10 Version 1.1

BASIC QUICKDRAW

Includes Demonstration Program BasicQuickDraw

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).

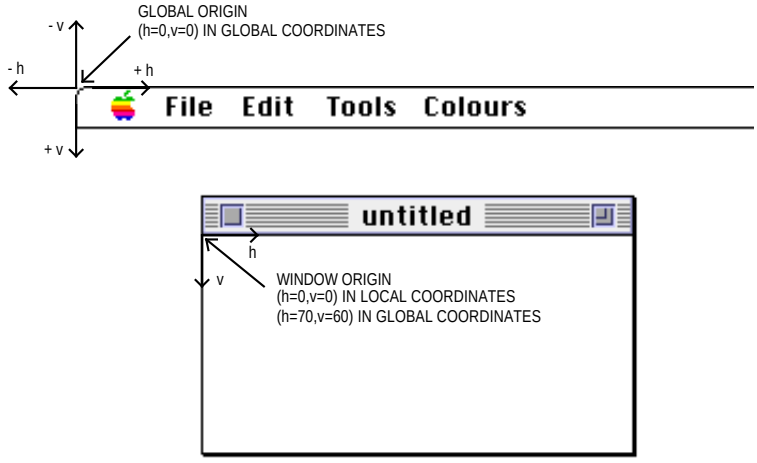

FIG 1 - LOCAL AND GLOBAL COORDINATE SYSTEMS

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.

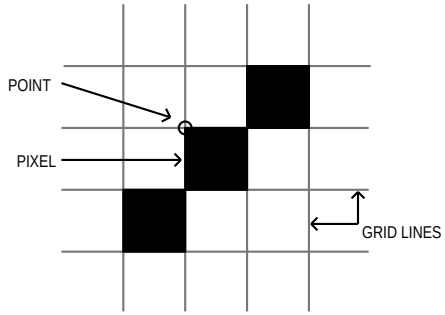

FIG 2 - POINTS AND PIXELS

The data type for points is `Point`:

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

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

Rectangles

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

The data type for rectangles is `Rect`:

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

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

If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is less than the left, the rectangle is an **empty rectangle**, that is, one that contains no data.

Regions

One of QuickDraw's most powerful features is to work with regions of arbitrary size, shape and complexity. A region is an arbitrary area, or set of areas, the outline of which is one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate ones,

and can even have holes in the middle. In the examples at Fig 3, the region on the left has a hole and the one on the right consists of two unconnected areas.

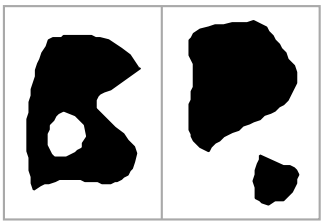

FIG 3 - TWO REGIONS

The data type for regions is `Region`:

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

typedef struct Region Region;
typedef Region *RgnPtr, **RgnHandle;
```

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`:

```
struct GrafPort
{
  short      device;      // Device-specific information. (0 = screen.)
  BitMap     portBits;    // BitMap.
  Rect       portRect;    // Port Rectangle.
  RgnHandle  visRgn;      // Visible region.
  RgnHandle  clipRgn;     // Clipping region.
  Pattern    bkPat;       // Background pattern.
  Pattern    fillPat;     // Fill pattern.
  Point      pnLoc;       // Pen location.
  Point      pnSize;      // Pen size.
  short      pnMode;      // Pen mode.
  Pattern    pnPat;       // Pen pattern.
  short      pnVis;       // Pen visibility.
  short      txFont;      // Font number for text.
  Style      txFace;      // Text's font style.
  SInt8      filler;
  short      txMode;      // Transfer mode for text.
  short      txSize;      // Font size for text.
  Fixed      spExtra;     // Spacing for full justification..
  long       fgColor;     // Foreground colour.
  long       bkColor;     // Background colour.
  short      colrBit;     // Color bit.
  short      patStretch;  // (Used internally.)
  Handle     picSave;     // Picture being saved. (Used internally.)
  Handle     rgnSave;     // Region being saved. (Used internally.)
  Handle     polySave;    // Polygon being saved. (Used internally.)
  QDProcsPtr grafProcs;   // Low-level drawing routines.
};
```

```
typedef struct GrafPort GrafPort;
typedef GrafPort *GrafPtr;
typedef GrafPtr WindowPtr
```

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:

```
struct BitMap
{
  Ptr    baseAddr;  // Pointer to bit image.
  short  rowBytes;  // Row width.
  Rect   bounds;    // Boundary rectangle.
};

typedef struct BitMap BitMap;
typedef BitMap *BitMapPtr, **BitMapHandle;
```

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.

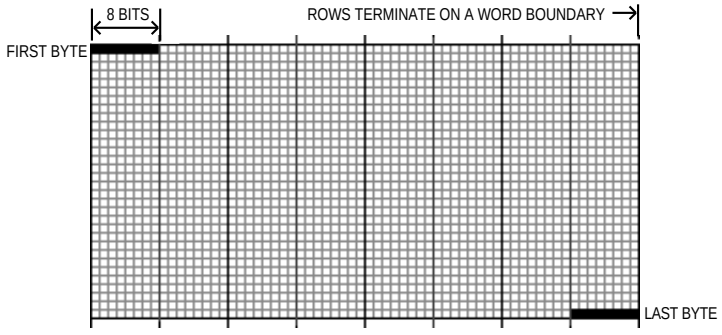

FIG 4 - A BIT IMAGE

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).

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

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

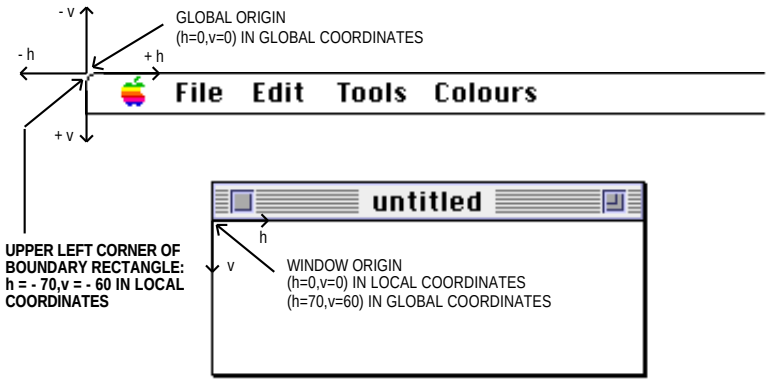

FIG 5 - LOCAL AND GLOBAL COORDINATE SYSTEMS AND THE BOUNDARY RECTANGLE

`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.

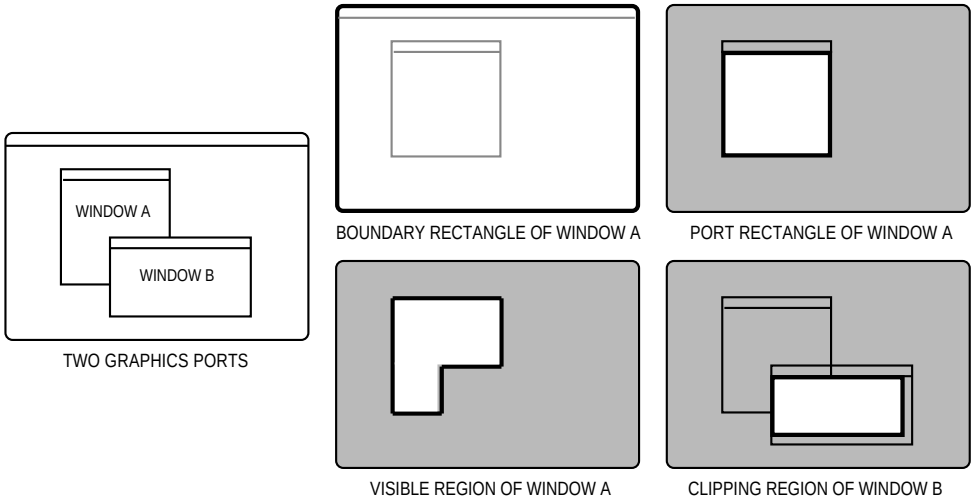

FIG 6 - BOUNDARY RECTANGLE, PORT RECTANGLE, VISIBLE REGION AND CLIPPING REGION

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.

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-32767 pixel square. Note that, if either the width or height is set to 0, the pen does not draw.

---

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

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`.

> **Note:** Patterns were originally defined as:
>
> ```
> typedef unsigned char Pattern[8];
> ```
>
> With the introduction of the Universal Headers, the definition was changed to:
>
> ```
> struct Pattern
> {
>   UInt8 pat[8];
> }
> typedef struct Pattern Pattern;
> ```
>
> The old array definition of `Pattern` would cause 68000-based CPUs to crash in certain circumstances. The new structure definition may require changes in 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 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.

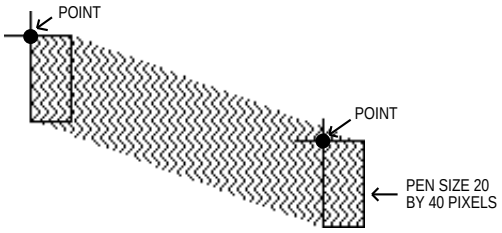

FIG 9 - A LINE DRAWN WITH A BIT PATTERN

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.)

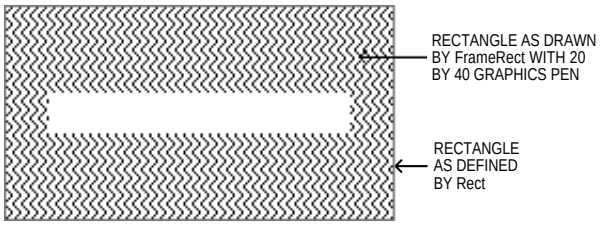

FIG 10 - A RECTANGLE DRAWN BY THE FrameRect PROCEDURE

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**).

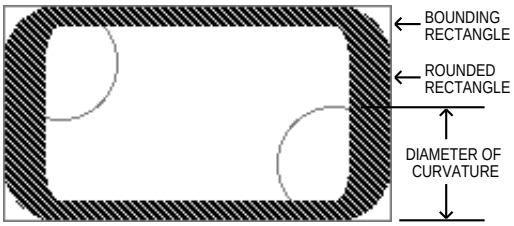

FIG 11 - A ROUNDED RECTANGLE

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`).

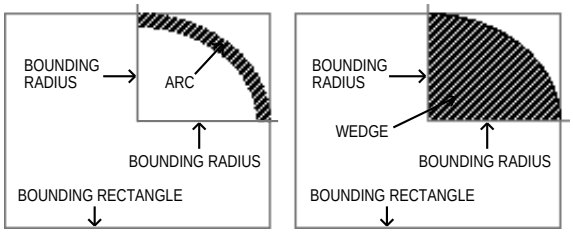

FIG 12 - AN ARC AND A WEDGE

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.

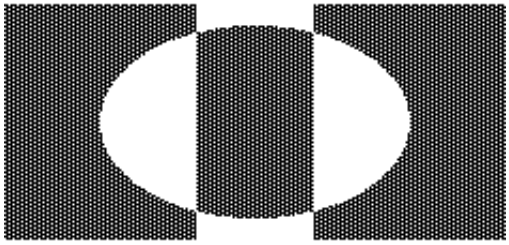

FIG 14 - DRAWING A REGION

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, an oval, and some text.

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:

```
short fontNum;

GetFNum("\pPalatino",&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.

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.

---

[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.

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.

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.

[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.

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

```
struct Pattern
{
  UInt8   pat[8];
};

typedef struct Pattern Pattern;
typedef Pattern *PatPtr;
typedef PatPtr *PatHandle;
```

> **Note:** Patterns were originally defined as:
>
> ```
> typedef unsigned char Pattern[8];
> ```
>
> The new struct definition was introduced with the Universal Headers. The old array definition of `Pattern` would cause 68000-based CPUs to crash in certain circumstances. The new definition may require changes in older source code in order to compile.

Point

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

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

Rect

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

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

Region

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

typedef struct Region Region;
typedef Region *RgnPtr, **RgnHandle;
```

GrafPort

```
struct GrafPort
{
  short      device;     // Device-specific information. (0 = screen.)
  BitMap     portBits;   // BitMap.
  Rect       portRect;   // Port Rectangle.
```

```
  RgnHandle   visRgn;     // Visible region.
  RgnHandle   clipRgn;    // Clipping region.
  Pattern     bkPat;      // Background pattern.
  Pattern     fillPat;    // Fill pattern.
  Point       pnLoc;      // Pen location.
  Point       pnSize;     // Pen size.
  short       pnMode;     // Pen mode.
  Pattern     pnPat;      // Pen pattern.
  short       pnVis;      // Pen visibility.
  short       txFont;     // Font number for text.
  Style       txFace;     // Text's font style.
  SInt8       filler;
  short       txMode;     // Transfer mode for text.
  short       txSize;     // Font size for text.
  Fixed       spExtra;    // Spacing for full justification..
  long        fgColor;    // Foreground colour.
  long        bkColor;    // Background colour.
  short       colrBit;    // Color bit.
  short       patStretch; // (Used internally.)
  Handle      picSave;    // Picture being saved. (Used internally.)
  Handle      rgnSave;    // Region being saved. (Used internally.)
  Handle      polySave;   // Polygon being saved. (Used internally.)
  QDProcsPtr  grafProcs;  // Low-level drawing routines.
};

typedef struct GrafPort GrafPort;
typedef GrafPort *GrafPtr;
typedef GrafPtr WindowPtr
```

BitMap

```
struct BitMap
{
  Ptr         baseAddr;   // Pointer to bit image.
  short       rowBytes;   // Row width.
  Rect        bounds;     // Boundary rectangle.
};

typedef struct BitMap BitMap;
typedef BitMap *BitMapPtr, **BitMapHandle;
```

Polygon

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

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

PenState

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

typedef struct PenState PenState;
```

Routines

Initialising QuickDraw

```
void        InitGraf(void *globalPtr);
```

Opening and Closing Basic Graphics Ports

```
void       OpenPort(GrafPtr port);
void       InitPort(GrafPtr port);
void       ClosePort(GrafPtr port);
```

Saving and Restoring Graphics Ports

```
void       GetPort(GrafPtr *port);
void       SetPort(GrafPtr port);
```

Managing BitMaps, Port Rectangles and Clipping Regions

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

Manipulating Points in Graphics Ports

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

Managing the Graphics Pen

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

Changing the BackGround Bit Pattern

```
void       BackPat(const Pattern *pat);
```

Drawing Lines

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

Creating and Managing Rectangles

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

Drawing Rectangles

```
void       FrameRect(const Rect *r);
void       PaintRect(const Rect *r);
```

```
void       FillRect(const Rect *r,ConstPatternParam pat);
void       InvertRect(const Rect *r);
void       EraseRect(const Rect *r);
```

Drawing Rounded Rectangles

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

Drawing Ovals

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

Drawing Arcs and Wedges

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

Creating and Managing Polygons

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

Drawing and Painting Polygons

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

Creating and Managing Regions

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

Drawing Regions

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

Setting Text Characteristics

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

Drawing Text

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

Measuring Text

```
short      CharWidth(short ch);
short      StringWidth(ConstStr255Param s);
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

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

Copying Images

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

Drawing With the Eight-Color System

```
void       ForeColor(long color);
void       BackColor(long color);
void       ColorBit(short whichBit);
```

Determining Whether QuickDraw has Finished Drawing

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

Getting Pattern Resources

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

Demonstration Program

```
1  // ############################################################################################
2  // BasicQuickDraw.c
3  // ############################################################################################
4  //
5  // This program:
6  //
7  // •  Opens a window in which the results of various basic QuickDraw drawing operations
8  //    are displayed.
9  //
10 //    Individual drawing operations (eg, draw lines, draw rectangles, draw polygons, etc)
11 //    are selected from a pull-down menu titled "Demonstration".
12 //
13 // •  Quits when the user selects Quit from the File menu or clicks the window's close
14 //    box.
15 //
16 // 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 transfer modes demonstration.
//
// •  A 'PICT' resource (purgeable) used for the CopyBits demonstration.
//
// •  'STR#' resources (purgeable) containing strings used by the CopyBits and text
//    demonstrations.
//
// ##############################################################################################

// ……… includes

#include <Fonts.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <SegLoad.h>
#include <ToolUtils.h>
#include <Devices.h>

// ……… defines

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

#define MAXLONG         0x7FFFFFFF

// ……… global variables

Boolean   gDone;
WindowPtr gWindowPtr;
Boolean   gDrawWithMouseActivated;

// ……… function prototypes

void    main                (void);
void    doInitManagers      (void);
void    doMouseDown         (EventRecord *);
void    doEvents            (EventRecord *);
void    doMenuChoice        (SInt32);
void    doDemonstrationMenu (SInt16);
void    doLines             (void);
SInt16  doRandomNumber      (SInt16);
void    doRectOval          (void);
void    doArcWedge          (void);
void    doPolygon           (void);
void    doRegion            (void);
void    doTransferMode      (void);
void    doCopyBits          (void);
void    doText              (void);
void    doBasicColours      (void);
```

```
93  void    doDrawWithMouse     (void);
94
95  // ############################################################################## main
96
97  void  main(void)
98  {
99    Handle      menubarHdl;
100   MenuHandle  menuHdl;
101   EventRecord eventRec;
102   Boolean     gotEvent;
103
104   // ……… initialise managers
105
106   doInitManagers();
107
108   // ……… see random number generator
109
110   GetDateTime((UInt32 *) (&qd.randSeed));
111
112   // ……… set up menu bar and menus
113
114   if(!(menubarHdl = GetNewMBar(rMenubar)))
115     ExitToShell();
116   SetMenuBar(menubarHdl);
117   DrawMenuBar();
118
119   if(!(menuHdl = GetMenuHandle(mApple)))
120     ExitToShell();
121   else
122     AppendResMenu(menuHdl,'DRVR');
123
124   // ……… open window
125
126   if(!(gWindowPtr = GetNewWindow(rWindow,NULL,(WindowPtr)-1)))
127     ExitToShell();
128
129   SetPort(gWindowPtr);
130
131   TextSize(10);
132
133   // ……… eventLoop
134
135   gDone = false;
136
137   while(!gDone)
138   {
139     gotEvent = WaitNextEvent(everyEvent,&eventRec,MAXLONG,NULL);
140     if(gotEvent)
141       doEvents(&eventRec);
142   }
143 }
144
145 // ########################################################################## doInitManagers
146
147 void  doInitManagers(void)
148 {
149   MaxApplZone();
150   MoreMasters();
151
152   InitGraf(&qd.thePort);
153   InitFonts();
154   InitWindows();
155   InitMenus();
156   TEInit();
157   InitDialogs(NULL);
158
159   InitCursor();
160   FlushEvents(everyEvent,0);
161 }
162
163 // ############################################################################### doEvents
164
165 void  doEvents(EventRecord *eventRecPtr)
166 {
167   WindowPtr windowPtr;
168   SInt8     charCode;
```

```
169
170   windowPtr = (WindowPtr) eventRecPtr->message;
171
172   switch(eventRecPtr->what)
173   {
174     case mouseDown:
175       doMouseDown(eventRecPtr);
176       break;
177
178     case keyDown:
179     case autoKey:
180       charCode = eventRecPtr->message & charCodeMask;
181       if((eventRecPtr->modifiers & cmdKey) != 0)
182         doMenuChoice(MenuKey(charCode));
183       break;
184
185     case updateEvt:
186       BeginUpdate(windowPtr);
187       EndUpdate(windowPtr);
188       break;
189   }
190 }
191
192 // ############################################################################## doMouseDown
193
194 void  doMouseDown(EventRecord *eventRecPtr)
195 {
196   WindowPtr windowPtr;
197   SInt16    partCode;
198
199   partCode = FindWindow(eventRecPtr->where,&windowPtr);
200
201   switch(partCode)
202   {
203     case inMenuBar:
204       doMenuChoice(MenuSelect(eventRecPtr->where));
205       break;
206
207     case inSysWindow:
208       SystemClick(eventRecPtr,windowPtr);
209       break;
210
211     case inContent:
212       if(windowPtr != FrontWindow())
213         SelectWindow(windowPtr);
214       else
215         if(gDrawWithMouseActivated)
216           doDrawWithMouse();
217       break;
218
219     case inDrag:
220       DragWindow(windowPtr,eventRecPtr->where,&qd.screenBits.bounds);
221       break;
222
223     case inGoAway:
224       if(TrackGoAway(windowPtr,eventRecPtr->where) == true)
225         gDone = true;
226       break;
227   }
228 }
229
230 // ############################################################################## doMenuChoice
231
232 void  doMenuChoice(SInt32 menuChoice)
233 {
234   SInt16  menuID, menuItem;
235   Str255  itemName;
236   SInt16  daDriverRefNum;
237
238   menuID = HiWord(menuChoice);
239   menuItem = LoWord(menuChoice);
240
241   if(menuID == 0)
242     return;
243
244   switch(menuID)
```

```
  {
    case mApple:
      GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
      daDriverRefNum = OpenDeskAcc(itemName);
      break;

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

    case mDemonstration:
      doDemonstrationMenu(menuItem);
      break;
  }

  HiliteMenu(0);
}

// ########################################################################## doDemonstrationMenu

void  doDemonstrationMenu(SInt16 menuItem)
{
  gDrawWithMouseActivated = false;

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

    case iRectAndOval:
      doRectOval();
      break;

    case iArcAndWedge:
      doArcWedge();
      break;

    case iPolygon:
      doPolygon();
      break;

    case iRegion:
      doRegion();
      break;

    case iTransferMode:
      doTransferMode();
      break;

    case iCopyBits:
      doCopyBits();
      break;

    case iText:
      doText();
      break;

    case iBasicColour:
      doBasicColours();
      break;

    case iDrawWithMouse:
      FillRect(&(gWindowPtr->portRect),&qd.white);
      MoveTo(10,25);
      DrawString("\pClick in the window and drag the mouse to the right and down");
      gDrawWithMouseActivated = true;
      break;
  }
}

// ################################################################################ doLines

void  doLines(void)
{
```

```
321     SInt16    top, left, bottom, right, a, b, c;
322     RgnHandle oldClipRgn;
323     Rect      newClipRect;
324     Pattern   systemPattern;
325     SInt32    finalTicks;
326
327     FillRect(&(gWindowPtr->portRect),&qd.white);
328
329     PenMode(patCopy);
330
331     left    = gWindowPtr->portRect.left + 10;
332     top     = gWindowPtr->portRect.top + 10;
333     right   = gWindowPtr->portRect.right - 10;
334     bottom  = gWindowPtr->portRect.bottom - 10;
335
336     oldClipRgn = NewRgn();
337     GetClip(oldClipRgn);
338     SetRect(&newClipRect,left,top,right,bottom);
339     ClipRect(&newClipRect);
340
341     for(a=1;a<39;a++)
342     {
343       b = doRandomNumber(gWindowPtr->portRect.right - gWindowPtr->portRect.left);
344       c = doRandomNumber(gWindowPtr->portRect.right - gWindowPtr->portRect.left);
345
346       GetIndPattern(&systemPattern,sysPatListID,a);
347       PenPat(&systemPattern);
348       PenSize(a * 2,1);
349
350       MoveTo(b,gWindowPtr->portRect.top);
351       LineTo(c,gWindowPtr->portRect.bottom);
352
353       Delay(15,&finalTicks);
354     }
355
356     SetClip(oldClipRgn);
357     DisposeRgn(oldClipRgn);
358
359     SetWTitle(gWindowPtr,"\pClick Mouse for More Lines");
360     while(!Button()) ;
361     SetWTitle(gWindowPtr,"\pBasic QuickDraw");
362
363     FillRect(&(gWindowPtr->portRect),&qd.white);
364     PenSize(1,1);
365     PenPat(&qd.black);
366     PenMode(patXor);
367
368     for(a=left,b=right;a<right+1;a++,b--)
369     {
370       MoveTo(a,top);
371       LineTo(b,bottom);
372     }
373
374     for(a=bottom,b=top;b<bottom+1;a--,b++)
375     {
376       MoveTo(left,a);
377       LineTo(right,b);
378     }
379   }
380
381   // ############################################################################ doRandomNumber
382
383   SInt16  doRandomNumber(SInt16 range)
384   {
385     SInt32 randomNumber;
386
387     randomNumber = Random();
388     if(randomNumber < 0)
389       randomNumber *= -1;
390
391     return((randomNumber * range) / 32767);
392   }
393
394   // ############################################################################## doRectOval
395
396   void  doRectOval(void)
```

```
{
  Rect    theRect;
  SInt32  finalTicks;
  Pattern systemPattern;

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

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

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

  MoveTo(10,15);
  DrawString("\pFrameRect");
  FrameRect(&theRect);
  Delay(30,&finalTicks);

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

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

  MoveTo(255,154);
  DrawString("\pFrameOval");
  OffsetRect(&theRect,245,0);
  PenSize(40,20);
  PenPat(&qd.dkGray);
  FrameOval(&theRect);

  SetWTitle(gWindowPtr,"\pClick Mouse For Invert and Erase");
  while(!Button()) ;
  SetWTitle(gWindowPtr,"\pBasic QuickDraw");
  SetRect(&theRect,10,145,490,154);
  EraseRect(&theRect);
  SetRect(&theRect,255,160,490,270);
  Delay(30,&finalTicks);

  MoveTo(10,154);
  DrawString("\pInvertRoundRect");
  OffsetRect(&theRect,-245,0);
  InvertRoundRect(&theRect,120,60);
  Delay(30,&finalTicks);

  MoveTo(255,154);
  DrawString("\pEraseOval");
  OffsetRect(&theRect,245,0);
  EraseOval(&theRect);
  Delay(30,&finalTicks);
}

// ############################################################################ doArcWedge

void  doArcWedge(void)
{
  Rect    theRect;
  SInt16  a;
  SInt32  finalTicks;
  Pattern systemPattern;

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

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

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

```
473
474     MoveTo(10,15);
475     DrawString("\pFrameArc");
476     for(a=0;a<270;a++)
477       FrameArc(&theRect,135,a);
478     Delay(30,&finalTicks);
479
480     MoveTo(255,15);
481     DrawString("\pFillArc");
482     OffsetRect(&theRect,245,0);
483     GetIndPattern(&systemPattern,sysPatListID,16);
484     FillArc(&theRect,315,270,&systemPattern);
485     Delay(30,&finalTicks);
486     OffsetRect(&theRect,-30,0);
487     FillArc(&theRect,225,90,&systemPattern);
488   }
489
490   // ############################################################################## doPolygon
491
492   void  doPolygon(void)
493   {
494     PolyHandle  polygonHdl;
495     SInt32      finalTicks;
496     Pattern     systemPattern;
497
498     FillRect(&(gWindowPtr->portRect),&qd.white);
499
500     PenSize(10,30);
501     PenPat(&qd.gray);
502     PenMode(patCopy);
503
504     polygonHdl = OpenPoly();
505     MoveTo(10,20);
506     LineTo(225,40);
507     LineTo(100,120);
508     LineTo(215,248);
509     LineTo(10,248);
510     LineTo(50,200);
511     ClosePoly();
512
513     MoveTo(10,15);
514     DrawString("\pFramePoly");
515     FramePoly(polygonHdl);
516     Delay(30,&finalTicks);
517
518     MoveTo(265,15);
519     DrawString("\pFillPoly");
520     OffsetPoly(polygonHdl,255,0);
521     GetIndPattern(&systemPattern,sysPatListID,9);
522     FillPoly(polygonHdl,&systemPattern);
523
524     KillPoly(polygonHdl);
525   }
526
527   // ############################################################################## doRegion
528
529   void  doRegion(void)
530   {
531     RgnHandle   regionHdl;
532     Rect        theRect;
533     SInt32      finalTicks;
534
535     FillRect(&(gWindowPtr->portRect),&qd.white);
536     PenPat(&qd.gray);
537     PenMode(patCopy);
538
539     regionHdl = NewRgn();
540
541     OpenRgn();
542     SetRect(&theRect,10,20,100,130);
543     FrameRect(&theRect);
544     SetRect(&theRect,155,20,245,130);
545     FrameRect(&theRect);
546     SetRect(&theRect,55,30,200,120);
547     FrameOval(&theRect);
548     CloseRgn(regionHdl);
```

```
549
550     MoveTo(10,15);
551     DrawString("\pFrameRgn");
552     PenPat(&qd.black);
553     PenSize(10,20);
554     FrameRgn(regionHdl);
555     Delay(30,&finalTicks);
556
557     MoveTo(255,15);
558     DrawString("\p1. FillRgn");
559     OffsetRgn(regionHdl,245,0);
560     FillRgn(regionHdl,&qd.dkGray);
561     Delay(30,&finalTicks);
562
563     MoveTo(10,154);
564     DrawString("\p2. InsetRgn (10 horizontal, 10 vertical)");
565     OffsetRgn(regionHdl,-245,140);
566     InsetRgn(regionHdl,10,10);
567     PenPat(&qd.dkGray);
568     PaintRgn(regionHdl);
569     Delay(30,&finalTicks);
570
571     MoveTo(255,154);
572     DrawString("\p3. InsetRgn (-10 horizontal, -10 vertical)");
573     OffsetRgn(regionHdl,245,0);
574     InsetRgn(regionHdl,-10,-10);
575     PenPat(&qd.dkGray);
576     PaintRgn(regionHdl);
577
578     DisposeRgn(regionHdl);
579   }
580
581   // ############################################################################## doTransferMode
582
583   void  doTransferMode(void)
584   {
585     Handle  crossIconHdl, squareIconHdl;
586     Rect    destRect;
587     SInt16  a, b;
588     BitMap  squareIconMap;
589     SInt32  finalTicks;
590     SInt16  sourceMode = 0;
591     Str255  sourceString;
592
593     FillRect(&(gWindowPtr->portRect),&qd.white);
594
595     PenSize(1,1);
596     PenPat(&qd.gray);
597     PenMode(patOr);
598
599     if(!(crossIconHdl = GetIcon(rCrossIcon)))
600     {
601       SysBeep(10);
602       return;
603     }
604
605     if(!(squareIconHdl = GetIcon(rSquareIcon)))
606     {
607       SysBeep(10);
608       return;
609     }
610
611     SetRect(&destRect,120,8,190,78);
612     PlotIcon(&destRect,crossIconHdl);
613     FrameRect(&destRect);
614     MoveTo(200,48);
615     DrawString("\pDestination");
616
617     SetRect(&destRect,270,8,340,78);
618     PlotIcon(&destRect,squareIconHdl);
619     FrameRect(&destRect);
620     MoveTo(350,48);
621     DrawString("\pSource");
622
623     for(a=91;a<192;a+=100)
624       for(b=30;b<391;b+=120)
```

```
625       {
626         SetRect(&destRect,b,a,b+70,a+70);
627         PlotIcon(&destRect,crossIconHdl);
628       }
629
630     HLock(squareIconHdl);
631
632     squareIconMap.baseAddr = *squareIconHdl;
633     squareIconMap.rowBytes = 4;
634     SetRect(&squareIconMap.bounds,0,0,31,31);
635
636     for(a=91;a<192;a+=100)
637       for(b=30;b<391;b+=120)
638       {
639         Delay(30,&finalTicks);
640         SetRect(&destRect,b,a,b+70,a+70);
641         CopyBits(&squareIconMap,&qd.thePort->portBits,&squareIconMap.bounds,&destRect,
642                  sourceMode++,NULL);
643         GetIndString(sourceString,rModeStringList,sourceMode);
644         MoveTo(b,a+82);
645         DrawString(sourceString);
646       }
647
648     HUnlock(squareIconHdl);
649   }
650
651   // ############################################################################## doCopyBits
652
653   void  doCopyBits(void)
654   {
655     WindowPtr windowPtr;
656     GrafPtr   oldPort;
657     PicHandle pictureHdl;
658     Rect      sourceRect, destRect;
659     SInt32    finalTicks;
660
661     FillRect(&(gWindowPtr->portRect),&qd.white);
662
663     if(!(windowPtr = GetNewWindow(rSmallWindow,NULL,(WindowPtr)-1)))
664       ExitToShell();
665
666     GetPort(&oldPort);
667     SetPort(windowPtr);
668
669     if(!(pictureHdl = GetPicture(rPicture)))
670     {
671       DisposeWindow(windowPtr);
672       SysBeep(10);
673       return;
674     }
675
676     HNoPurge((Handle) pictureHdl);
677     SetRect(&sourceRect,65,40,165,182);
678     DrawPicture(pictureHdl,&sourceRect);
679     HPurge((Handle) pictureHdl);
680
681     SetWTitle(windowPtr,"\pClick Mouse for CopyBits");
682     while(!Button()) ;
683
684     SetRect(&destRect,20,21,210,272);
685
686     CopyBits(&windowPtr->portBits,&oldPort->portBits,&sourceRect,&destRect,
687              srcCopy,NULL);
688
689     SetWTitle(windowPtr,"\pClick Mouse to Close");
690     Delay(60,&finalTicks);
691     while(!Button()) ;
692
693     DisposeWindow(windowPtr);
694     SetPort(oldPort);
695   }
696
697   // ################################################################################ doText
698
699   void  doText(void)
700   {
```

```
  SInt16  windowCentre, a, fontNum, stringWidth;
  Str255  textString;

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

  windowCentre = ((FrontWindow())->portRect.right - (FrontWindow())->portRect.left) / 2;

  for(a=1;a<9;a++)
  {
    if(a == 1)
    {
      GetFNum("\pGeneva",&fontNum);
      TextFont(fontNum);
      TextFace(normal);
    }
    else if(a == 2)
      TextFace(bold);
    else if(a == 3)
    {
      GetFNum("\pTimes",&fontNum);
      TextFont(fontNum);
      TextFace(italic);
    }
    else if(a == 4)
      TextFace(underline);
    else if(a == 5)
    {
      GetFNum("\pHelvetica",&fontNum);
      TextFont(fontNum);
      TextFace(outline);
    }
    else if(a == 6)
      TextFace(shadow);
    else if(a == 7)
    {
      GetFNum("\pChicago",&fontNum);
      TextFont(fontNum);
      TextFace(condense);
    }
    else if(a == 8)
    {
      TextFace(extend);
      TextMode(grayishTextOr);
    }

    if(a < 7)
      TextSize(a * 2 + 10);
    else
      TextSize(12);

    GetIndString(textString,rTextStringList, a);
    stringWidth = StringWidth(textString);
    MoveTo(windowCentre - (stringWidth / 2), a * 35 - 10);
    DrawString(textString);
  }

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

// ########################################################################## doBasicColours

void  doBasicColours(void)
{
  SInt16  a;
  Rect    theRect;
  SInt32  finalTicks;

  FillRect(&(gWindowPtr->portRect),&qd.dkGray);
  PenPat(&qd.black);
  PenMode(patCopy);

  for(a=1;a<9;a++)
```

```
777     {
778       Delay(30,&finalTicks);
779       if(a == 1)  ForeColor(blackColor);
780       if(a == 2)  ForeColor(whiteColor);
781       if(a == 3)  ForeColor(redColor);
782       if(a == 4)  ForeColor(greenColor);
783       if(a == 5)  ForeColor(blueColor);
784       if(a == 6)  ForeColor(cyanColor);
785       if(a == 7)  ForeColor(magentaColor);
786       if(a == 8)  ForeColor(yellowColor);
787
788       SetRect(&theRect,35,a*28,465,a*28+23);
789       PaintRect(&theRect);
790     }
791
792   ForeColor(blackColor);
793 }
794
795 // ############################################################################ doDrawWithMouse
796
797 void  doDrawWithMouse(void)
798 {
799   Point   mouseDownMouse, previousMouse, currentMouse;
800   Rect    drawRect;
801   Pattern thePattern;
802
803   PenSize(1,1);
804   PenPat(&qd.gray);
805   PenMode(patXor);
806
807   GetMouse(&mouseDownMouse);
808   drawRect.left = drawRect.right  = mouseDownMouse.h;
809   drawRect.top  = drawRect.bottom = mouseDownMouse.v;
810
811   GetMouse(&previousMouse);
812
813   while(StillDown())
814   {
815     GetMouse(&currentMouse);
816
817     if(currentMouse.v != previousMouse.v || currentMouse.h != previousMouse.h)
818     {
819       FrameRect(&drawRect);
820
821       drawRect.right = currentMouse.h;
822       drawRect.bottom = currentMouse.v;
823
824       FrameRect(&drawRect);
825     }
826
827     previousMouse.v = currentMouse.v;
828     previousMouse.h = currentMouse.h;
829   }
830
831   FrameRect(&drawRect);
832
833   PenMode(patCopy);
834
835   PenSize(2,2);
836   PenPat(&qd.black);
837   ForeColor(redColor);
838   FrameRect(&drawRect);
839
840   InsetRect(&drawRect,10,10);
841   PenSize(8,8);
842   GetIndPattern(&thePattern,0,5);
843   PenPat(&thePattern);
844   ForeColor(blueColor);
845   FrameRoundRect(&drawRect,40,40);
846
847   InsetRect(&drawRect,16,16);
848   PenSize(14,14);
849   GetIndPattern(&thePattern,0,6);
850   PenPat(&thePattern);
851   ForeColor(greenColor);
852   PaintOval(&drawRect);
```

```
853
854     PenMode(patCopy);
855     ForeColor(blackColor);
856   }
857
858   // ##############################################################################################
```

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.

#define

Lines 44-66 establish constants related to menu, window, icon, string list, and picture resources, menu IDs, and menu item numbers. Line 67 defines MAXLONG as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

Global Variables

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.

main

The main function initialises the system software managers (Line 106), seeds the random number generator (Line 110), sets up the menus (Lines 114-122), opens the main window and sets its graphics port as the current port for drawing operations (Lines 126-129), sets the text size (Line 131), and enters the main event loop (Lines 135-142).

Random numbers are used in the application-defined function doLines. randSeed (Line 110) 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 111 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.

doEvents and doMouseDown

doEvents and doMouseDown perform minimal event handling consistent with the satisfactory operation of the drawing demonstration aspects of the program. Note that, at Lines 214-216, the application-defined function doDrawWithMouse is called if the global variable gDrawWithMouseActivated contains true.

doMenuChoice and doDemonstrationMenu

doMenuChoice and doDemonstrationMenu handle menu choices from the Apple, File and Demonstration menus. Note that, at Lines 309-312, the global variable gDrawWithMouseActivated is set to true when the Draw With Mouse menu item is chosen.

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 327, FillRect is called to fill the entire port rectangle with the pattern white. At Line 329, the pen mode is set to patCopy for the lines demonstration.

Lines 331-339 set the window's clipping region to a rectangle 10 pixels inside the port rectangle. Lines 331-334 assign appropriate values to four variables which will be used to define the Rect representing the new clipping region, Lines 336-337 save the old clipping region, and Line 338 defines the Rect which is used in the call to ClipRect at Line 339 to establish the new clipping region.

Lines 341-354 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 343-344), the next system pattern is retrieved (Line 346), the pen pattern is set to this pattern (Line 347), the width of the pen is increased (Line 348), and a line is drawn from somewhere at the top of the port rectangle to somewhere at the bottom of the port rectangle (Lines 350-351). The line drawing is, of course, clipped to the clipping region established at Line 339, 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 356-357).

Lines 364-378 illustrate a well-known but nonetheless exotic capability of the humble line when it operates in the pattern mode patXor. Lines 363-366 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 368-378 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 moire (watered silk) pattern.

doRandomNumber

doRandomNumber generates and returns a random number between 0 and the value passed to it. At Line 387, the function Random returns a random number between -32,767 to 32,767. If the number is negative, it is made positive at Lines 388-389. Line 391 changes the random number from one between 0 and 32,767 to one between 0 and the value received by doRandomNumber, and returns that value.

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 402-406 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 408 defines the Rect required as a parameter by the drawing routines.

Line 412 draws a framed rectangle.

Lines 417-419 offset the rectangle to the right, set the pen pattern to ltGray and paint a rectangle.

Lines 424-426 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 431-434 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 437-440), Lines 446-447 offset the rectangle so that it is back over the rounded rectangle and invert the rounded rectangle with a call to InvertRoundRect.

Lines 452-453 offset the rectangle so that it is back over the oval and erase the oval with a call to EraseOval.

doArcWedge

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

Lines 466-470 fill the port rectangle with the pattern white, and set the pen size, pattern, and mode. Line 472 defines the bounding rectangle for the arc and wedges.

Lines 476-477 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 482-487 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.

doPolygon

doPolygon draws a framed and filled polygon.

Lines 498-502 fill the port rectangle with the pattern white, and set the pen size, pattern, and mode.

Lines 504-511 initiate the recording of the polygon definition (Line 504), define the polygon (Lines 505-510), and stop the recording (Line 511). Note that, in this demonstration, the last vertex is not joined to the first vertex.

Line 515 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 520-522 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 524 deallocates the memory used to store the polygon.

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 535-537 fill the port rectangle with the pattern white, and set the pen pattern and mode.

Line 539 allocates memory for a new region and a region pointer, initialises the contents of the region and make it an empty rectangle.

Lines 541-548 initiate the recording of a region shape (Line 541), create a region definition comprising two rectangles and an overlapping oval (Lines 542-547) and terminate the recording (Line 548).

Lines 552-554 set the pen pattern and pen size and draw a framed region based on the region definition.

Lines 559-560 offset the region to the right and then draw a filled region with the pattern dkGray.

Lines 565-568 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 573-576 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 578 deallocates the memory used to store the region.

doTransferMode

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

Lines 593-597 fill the port rectangle with the pattern white, and set the pen size, pattern and pattern mode.

Lines 599-609 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 612 and 618 use PlotIcon to draw the icons, expanding them into the 71 pixel by 71 pixel rectangle defined at Lines 611 and 617. 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 623-628 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 630). The address of the square icon image data is then assigned to the baseAddr field of a BitMap record (Line 632), the rowBytes field is assigned the value 4 (Line 633), and the bounds field is assigned a rectangle defining the normal icon size (Line 634).

Lines 636-646 copy the bit image into the graphics port eight times, overdrawing the previously drawn cross icons. Line 640 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 641. 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 643-645 retrieve the appropriate string containing the relevant source mode from the 'STR#' resource and print this string under each image.

Line 648 unlocks the icon image data.

doCopyBits

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

Line 661 prepares the way by filling the port rectangle with the pattern white.

Line 663 opens a small window over the right side of the main window. Lines 666-667 save the current graphics port and make the new window's port the current graphics port.

Line 669 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 676), used immediately (Line 678), and immediately made purgeable again (Line 679). The picture is drawn in the current graphics port (the small window).

When the user clicks the mouse button (Line 682), a large rectangle is defined to represent the size and shape in which the copied image is to be drawn (Line 684). This is used in the call to CopyBits (Line 686), 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 691), Lines 693-694 dispose of the small window and reset the current graphics port to that of the main window.

doText

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

Line 704 prepares the way by filling the port rectangle with the pattern white.

Line 706 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 708-755 is a loop within which the text font, size and style are changed, a string is retrieved from a 'STR#' resource (Line 751), the width of the string in pixels is determined (Line 752), and the string is drawn centrally (from left to right) in the window (Lines 753-754).

Note that, the last time around the loop, the transfer mode is set to grayishTextOr (Line 743).

Lines 757-761 reset the font, size and transfer mode back to the settings which existed before doText was called.

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.)

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 803-805 set the pen size to 1 pixel wide and high, the pen pattern to gray, and the pen mode to patXOr.

Line 807 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 811 assigns the same mouse location to another Point variable, which will be used for comparison purposes within the while loop entered at Line 813.

The while loop continues to execute while the mouse button remains down. Within the loop, the current mouse location is retrieved (Line 815) and compared with the previous mouse location (Line 817). 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 (Lines 819-824). Because the drawing mode is patXor, the first call to FrameRect erases the old rectangle. Note that, because Lines 819-824 only execute if the mouse has moved, the flicker which would otherwise occur when the mouse is stationary is avoided. At Lines 827-828, and preparatory to the next Line 817 comparison, the current mouse position is assigned to the variable which holds the previous mouse position.

When the mouse button is released, Line 831 erases the final "rubber-band" rectangle. Lines 833-855 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.

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™.