

11

QUICKDRAW PRELIMINARIES

Includes Demonstration Program PreQuickDraw

QuickDraw and Imaging

QuickDraw is a collection of system software routines that your application uses to perform most **imaging** operations on Macintosh computers. Imaging entails the construction and display of graphical information, including shapes, pictures, and text, which can be displayed on such output devices as screens and printers.

This chapter serves as a prelude to Chapter 12 — Drawing With QuickDraw, and introduces certain matters which need to be discussed before the matter of actually drawing with QuickDraw is addressed. These matters include the history of QuickDraw, RGB colours, colour and the video device, the colour graphics port, translation of RGB values, and graphics devices.

History of QuickDraw

As the system software has developed, QuickDraw has progressed through the following three main evolutionary stages:

- **Basic QuickDraw**, which was designed for the early black-and-white Macintoshes. System 7 added new capabilities to basic QuickDraw, including support for offscreen graphics worlds.
- The **original version of Color QuickDraw**, which was introduced with the first Macintosh II systems, and which could support up to 256 colours.
- The **current version of Color QuickDraw**, which was originally introduced as **32-bit Color QuickDraw**. This version has been expanded to support millions of colours.

The Appearance Manager requires that Color QuickDraw be present. Accordingly, this edition of Macintosh C assumes Color QuickDraw in all circumstances. Where the word "QuickDraw" is used, Color QuickDraw is invariably implied.

RGB Colours and Pixels

When using QuickDraw, you specify colours as **RGB colours**. An RGB (red-green-blue) colour is defined by its red, green and blue components. For example, when each of the red, green and blue components of a colour are at their maximum intensity (0xFFFF), the result is the colour white. When each of the components has zero intensity (0x0000), the result is the colour black.

You specify a colour to QuickDraw by creating an `RGBColor` structure in which you use three 16-bit unsigned integers to assign intensity values for the three additive¹ primary colours. The `RGBColor` data type is defined as follows:

```
struct RGBColor
{
    unsigned short red;           // Magnitude of red component.
    unsigned short green;       // Magnitude of green component.
    unsigned short blue;        // Magnitude of blue component.
};
typedef struct RGBColor RGBColor;
```

A **pixel** (picture element) is the smallest dot that QuickDraw can draw. Each colour pixel represents up to 48 bits in memory.

Colour and the Video Device

QuickDraw supports a variety of screens of differing sizes and colour capabilities. It is thus device-independent. Accordingly, you do not have to concern yourself with the capabilities of individual screens. For example, when your application uses an `RGBColor` structure to specify a colour by its red, green and blue components, with each component defined in a 16-bit integer, QuickDraw compares the resulting 48-bit value with the colours actually available on a video device (such as a plug-in video card or a built-in video interface) at execution time and then chooses the closest match. What the user finally sees depends on the characteristics of the actual video device and screen.

The video device that controls a screen may have either:

- **Indexed colours**, which support pixels of 1-bit, 2-bit, 4-bit, or 8-bit pixel depths². The indexed colour system was introduced with the Macintosh II, that is, at a time when memory was scarce and moving megabyte images around was impractical.
- **Direct colours**, which support pixels of 16-bit and 32-bit depths. Most video devices in the current day are direct colour devices. (However, as will be seen, there are circumstances in which a direct colour device will act like an indexed colour device.)

QuickDraw automatically determines which method is used by the video device and matches your requested 48-bit colour with the closest available colour.

Indexed Colour Devices

Video devices using indexed colours support a maximum of 256 colours at any one time, that is, with indexed colour, the maximum value of a pixel is limited to a single byte, with each pixel's byte specifying one of 256 (2⁸) different values.

¹ On a video device, the primary colours are referred to as additive because, when each of the three colour components is at maximum intensity, the result is the colour white. On a printer, the primary colours are referred to as subtractive because the colour black results when the three colour components are at maximum intensity.

² Pixel depth means the number of bits assigned to each pixel, and thus determines the maximum number of colours that can be displayed at the one time. A 4-bit pixel depth, for example, means that an individual pixel can be displayed in any one of 16 separate colours. An 8-bit pixel depth means that an individual pixel can be displayed in any one of 256 separate colours.

Video devices implementing indexed colour contain a data structure called a **colour lookup table** (or, more commonly, a **CLUT**). The CLUT, in turn, contains entries for all possible colour values.

256 colours is, for many images, sufficient for near-photographic quality. The problem is that the colours needed for one photographic image may not be appropriate for another. Because most indexed video devices use a **variable CLUT**, however, you can display one image using one set of 256 colours and then use system software to reload the CLUT with a second set of 256 colours that are appropriate for the next image.³ If your application needs this sort of control on indexed video devices, you can use the Palette Manager to arrange palettes (that is, sets of colours) for particular images and for video devices with differing colour capabilities.

If your application uses a 48-bit `RGBColor` structure to specify a colour, the Color Manager examines the colours available in the CLUT on the video device. Comparing the CLUT entries to the `RGBColor` structure you specify, the Color Manager determines which colour in the CLUT is closest, and gives QuickDraw the index to this colour. QuickDraw then draws with this colour.

Fig 1 illustrates this process. In Fig 1, the user selects a colour for some object in an application (1). Using a 48-bit `RGBColor` structure to specify the colour, the application calls a QuickDraw routine to draw the object in that colour (2). QuickDraw uses the Color Manager to determine what colour in the video device's CLUT comes closest to the requested colour (3).

At startup, the video device's declaration ROM supplies information for the creation of a `GDevice` structure (see below) that describes the characteristics of the device. The resulting structure contains a `ColorTable` structure that is kept synchronised with the card's CLUT.

The Color Manager examines the `GDevice` structure to find what colours are currently available (4) and to decide which colour comes closest to the one requested by the application. The Color Manager gets the index value for the best match and returns the value to QuickDraw (5), which puts the index value into those places in video RAM which store the object.

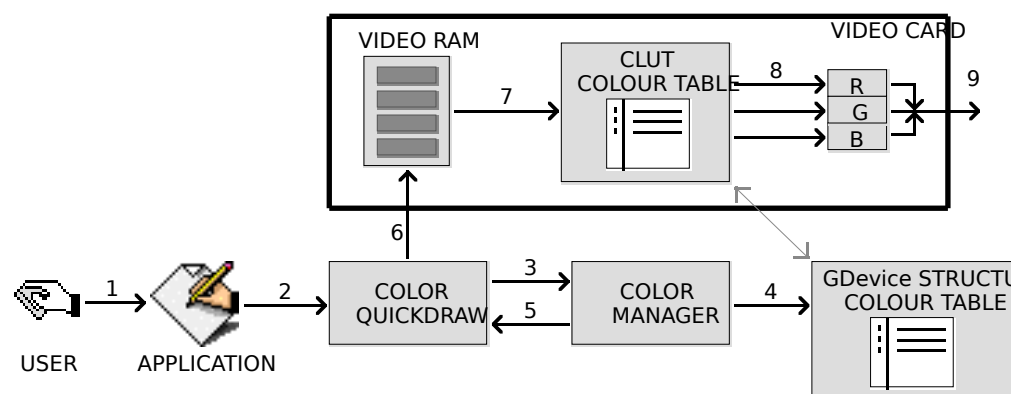


FIG 1 - INDEXED COLOUR SYSTEM

The video device continually displays video RAM by taking the index values, converting them to colours according to CLUT entries at those indexes (7), and sending them to the digital-to-analog converters (8) which produce a signal for the screen (9).

Direct Colour Devices

Video devices which implement direct colour eliminate the competition for limited colour lookup table spaces and remove the need for colour table matching. By using direct colour, video devices can support thousands or millions of colours.

³ Some Macintosh computers, such as grayscale PowerBook computers, have a fixed CLUT, which your application cannot change.

When you specify a colour using a 48-bit `RGBColor` structure on a direct colour system, QuickDraw truncates the least significant bits of its red, green and blue components to either 16 bits (five bits each for red, green and blue, with one bit unused) or 32 bits (eight bits for red, green and blue, with eight bits unused). (See Translation of RGB Colours to Pixel Values, below.) Using 16 bits, direct video devices can display 32,768 different colours. Using 32 bits, the device can display 16,777,215 different colours

Fig 2 illustrates the direct colour system. A user chooses a colour for some object (1) and, using a 48-bit `RGBColor` structure to specify the colour, the application uses a QuickDraw routine to draw the object in that colour (2).

QuickDraw knows from the `GDevice` structure (3) that the screen is controlled by a direct device in which pixels are, say, 32 bits deep, which means that eight bits are used for each of the red, green and blue components of the requested colour. Accordingly, QuickDraw passes the high eight bits from each 16-bit component of the 48-bit `RGBColor` structure to the video device (4), which stores the resulting 24-bit value in video RAM for the object. The video device continually displays video RAM by sending the 8-bit red, green and blue values for the colour to digital-to-analog converters (5) which produce a signal for the screen (6).

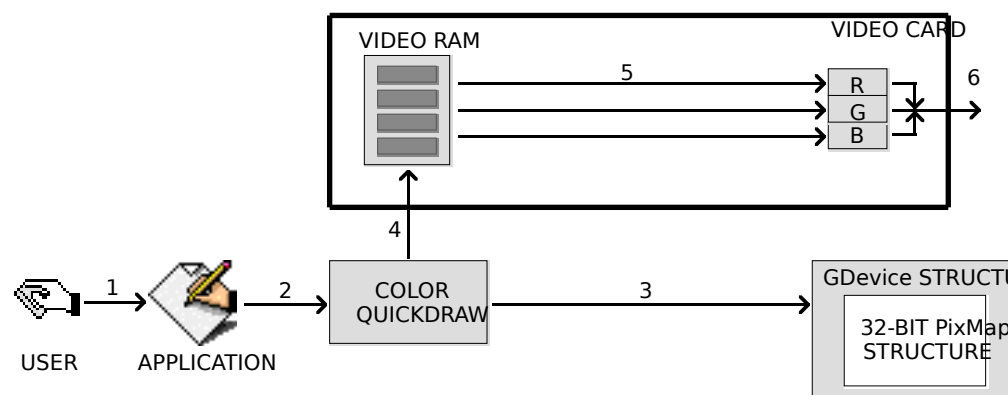


FIG 2 - DIRECT COLOUR SYSTEM

Direct colour not only removes much of the complexity of the CLUT mechanism for video device developers, but it also allows the display of thousands or millions of colours simultaneously, resulting in near-photographic resolution.

Direct Devices Operating Like Indexed Devices

Note that, when a user uses the Monitors and Sound control panel to set a direct colour device to use 256 colours (or less) as either a grayscale or colour device, the direct device creates a CLUT and operates like an indexed device.

Colour Graphics Port

QuickDraw performs its operations in a **colour graphics port**, a data structure of type `CGrafPort`.

Historical Note

There is a related type of graphics port called the **basic graphics port**, which was originally the drawing environment provided by basic QuickDraw. A basic graphics port is defined in a `GrafPort` structure. It contains the information basic QuickDraw needed to create and manipulate onscreen black-and-white images, or colour images that employed basic QuickDraw's eight-colour system.

Since the Appearance Manager requires that Color QuickDraw be present, the basic graphics port is now redundant.

A colour graphics port defines a complete drawing environment that determines where and how colour graphics operations take place. Amongst other things, a colour graphics port:

- Contains a handle to a **pixel map** which, in turn, contains a pointer to the area of memory in which your drawing operations take place.
- Contains a metaphorical graphics **pen** with which to perform drawing operations. (You can set this pen to different sizes, patterns and colours.)
- Holds information about text, which is styled and sized according to information in the graphics port.

The fields of a colour graphics port are maintained by QuickDraw. QuickDraw provides routines for changing and reading those fields. For example, routines are available to reshape and resize the pen, change the pen's pattern and colour, switch fonts, etc.

You can open many colour graphics ports at the same time. Each has its own local coordinate system, drawing pattern, background pattern, pen size and location, foreground colour, background colour, pixel map, etc. You can instantly switch from one graphics port to another using the function `SetPort`.

When you use Window Manager and Dialog Manager functions to create windows, dialog boxes, and alert boxes, those managers automatically create colour graphics ports for you

The `CGrafPort` structure is as follows:

```
struct CGrafPort
{
    short          device;           // Device-specific information.
    PixMapHandle   portPixMap;       // Handle to pixel map.
    short          portVersion;      // Flags and version number.
    Handle         grafVars;         // Handle to additional colour fields.
    short          chExtra;          // Extra width added to non-space characters.
    short          pnLocHFrac;       // Fractional horizontal pen position.
    Rect           portRect;         // Port rectangle.
    RgnHandle      visRgn;           // Visible region.
    RgnHandle      clipRgn;          // Clipping region.
    PixPatHandle   bkPixPat;         // Background pattern
    RGBColor       rgbFgColor;       // Requested foreground colour.
    RGBColor       rgbBkColor;      // Requested background colour
    Point          pnLoc;            // Pen location.
    Point          pnSize;           // Pen size.
    short          pnMode;           // Pattern mode.
    PixPatHandle   pnPixPat;         // Pen pattern.
    PixPatHandle   fillPixPat;       // Fill pattern.
    short          pnVis;            // Pen visibility.
    short          txFont;           // Font number for text.
    Style          txFace;           // Text font style.
    SInt8          filler;
    short          txMode;           // Text source mode.
    short          txSize;           // Font size for text.
    Fixed          spExtra;          // Extra width added to space characters.
    long           fgColor;          // Actual foreground colour.
    long           bkColor;          // Actual background colour.
    short          colrBit;          // Colour bit (reserved).
    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.)
    CQDProcsPtr   grafProcs;        // Pointer to low-level drawing routines.
};
typedef struct CGrafPort CGrafPort,*CGrafPtr;
typedef CGrafPtr CWindowPtr;
```

Main Field Descriptions

- portPixMap** A handle to a `PixMap` structure (see below) which describes the pixels in this colour graphics port.
- portVersion** In the highest two bits, flags set to indicate that this is a `CGrafPort` structure and, in the remainder of the field, the version number of QuickDraw that created this structure.
- grafVars** A handle to a `GrafVars` structure, which contains colour information additional to that contained in the `CGrafPort` structure itself, and which is used by QuickDraw and the Palette Manager. For example, one field contains the RGB colour for highlighting.
- portRect** The port rectangle that defines a subset of the pixel map to be used for drawing. All drawing done by your application occurs inside the port rectangle. (In a window's graphics port, the port rectangle is also called the **content region**.)

The port rectangle uses the local coordinate system defined by the **boundary rectangle** in the `portPixMap` field of the `PixMap` structure (see below). The upper-left corner (which, for a window, is called the window origin) of the port rectangle has a vertical coordinate of 0 and a horizontal coordinate of 0. The port rectangle usually falls within the boundary rectangle, but it is not required to do so.

- visRgn** The region of the graphics port that is actually visible on screen, that is, the part of the window not covered by other windows (see Fig 3). By default, the visible region is equivalent to the port rectangle.
- clipRgn** The graphics port's **clipping region**, an arbitrary region that you can use to limit drawing to any region within the port rectangle. The default clipping region is set arbitrarily large; however, you can change the clipping region using the function `ClipRect`. At Fig 3, for example, Window B's clipping region has been set by the application to prevent the scroll bar areas being over-drawn.

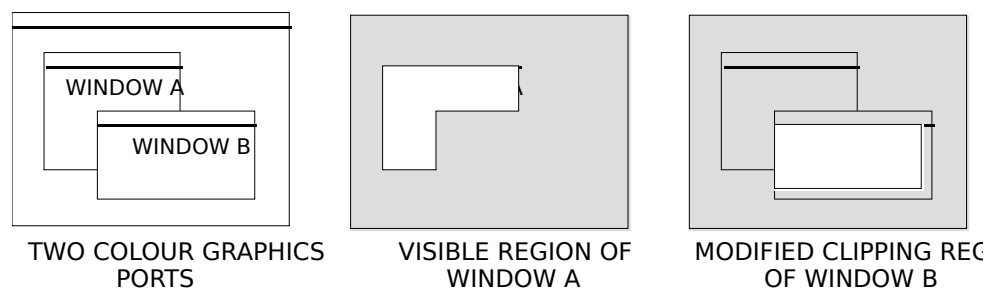


FIG 3 - VISIBLE REGION AND CLIPPING REGION

- bkPixPat** A handle to a `PixPat` structure that describes the background **pixel pattern**. Various QuickDraw functions use this pattern for filling scrolled or erased areas.
- rgbFgColor** An `RGBColor` structure that contains the *requested foreground colour*. By default, the foreground colour is black.
- rgbBkColor** An `RGBColor` structure that contains the *requested background colour*. By default, the background colour is white.
- pnLoc** The point where QuickDraw will begin drawing the next line, shape, or character. It can be anywhere on the coordinate plane.

| | |
|------------|--|
| pnSize | The vertical height and horizontal width of the graphics pen. The default size is a 1-by-1 pixel square. If either the pen width or height is 0, the pen does not draw. |
| pnMode | The pen transfer mode , that is, a Boolean or arithmetic operation that determines how QuickDraw transfers the pen pattern to the pixel map during drawing operations. When the graphics pen draws into a pixel map, QuickDraw first determines what pixels in the pixel image are affected and finds their corresponding pixels in the pen pattern. QuickDraw then does a pixel-by-pixel comparison based on the transfer mode. QuickDraw stores the resulting pixel in its proper place in the image. |
| pnPixPat | A handle to a PixPat structure (see below) that describes the pixel pattern used by the graphics pen for drawing lines and framed shapes, and for painting shapes. |
| fillPixpat | A handle to a PixPat structure (see below) that describes the pixel pattern that is used when you call QuickDraw shape filling functions. |
| pnVis | The graphics pen's visibility , that is, whether it draws on the screen. |
| txFont | A font family ID number that identifies the font to be used in the graphics port. |
| txFace | The style of the text, for example, bold, italic, and/or underlined. |
| txMode | The transfer mode for text drawing, which functions much like the transfer mode specified in the pnMode field (see above). |
| txSize | The text size in pixels. The Font Manager uses this information to provide the bitmaps for text drawing. The value in this field can be represented by $\text{point size} \times \text{device resolution} / 72 \text{ dpi}$ where <i>point</i> is a typographical term meaning approximately 1/72 inch. |
| fgColor | The pixel value of the foreground colour supplied by the Color Manager. (See Colour and the Video Device, above, and Translation of RGB Colours to Pixel Values, below.) This is the colour actually displayed on the device, that is, it is the the best available approximation to the requested color in the rgbFgColor field. |
| bkColor | The pixel value of the background colour supplied by the Color Manager. (See Colour and the Video Device, above, and Translation of RGB Colours to Pixel Values, below.) This is the colour actually displayed on the device, that is, it is the the best available approximation to the requested color in the rgbBkColor field. |

Pixel Maps

QuickDraw draws in a **pixel map**. The portPixMap field of the CGrafPort structure contains a handle to a pixel map, which is a data structure of type PixMap. A PixMap structure contains a pointer to a **pixel image**, as well as information on the image's storage format, depth, resolution, and colour usage. The PixMap structure is as follows:

```

struct PixMap
{
    Ptr          baseAddr;      // Pointer to image data.
    short       rowBytes;      // Flags, and bytes in a row.
    Rect        bounds;        // Boundary rectangle.
    short       pmVersion;     // Pixel Map version number.
    short       packType;      // Packing format.
    long        packSize;      // Size of data in packed state.
    Fixed       hRes;          // Horizontal resolution in dots per inch.
}

```

```

Fixed      vRes;          // Vertical resolution in dots per inch.
short     pixelType;     // Format of pixel image.
short     pixelSize;    // Physical bits per pixel.
short     cmpCount;     // Number of components in each pixel.
short     cmpSize;      // Number of bits in each component.
long      planeBytes;   // Offset to next plane.
CTabHandle pmTable;     // Handle to a colour table for this image.
long      pmReserved;   // (Reserved.)
};
typedef struct PixMap PixMap,*PixMapPtr,**PixMapHandle;

```

Field Descriptions

baseAddr For an onscreen pixel image, a pointer to the first byte of the image. The pixel image that appears on the screen is normally stored on a graphics card rather than in main memory. Note that there can be several pixel maps pointing to the same pixel image, each imposing its own coordinate system on it.

A pixel image is analogous to the **bit image** used by basic QuickDraw. 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. 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. A bit image can be any length that is a multiple of the row's width in bytes. On black-and-white Macintoshes, the screen itself is one large visible bit image.

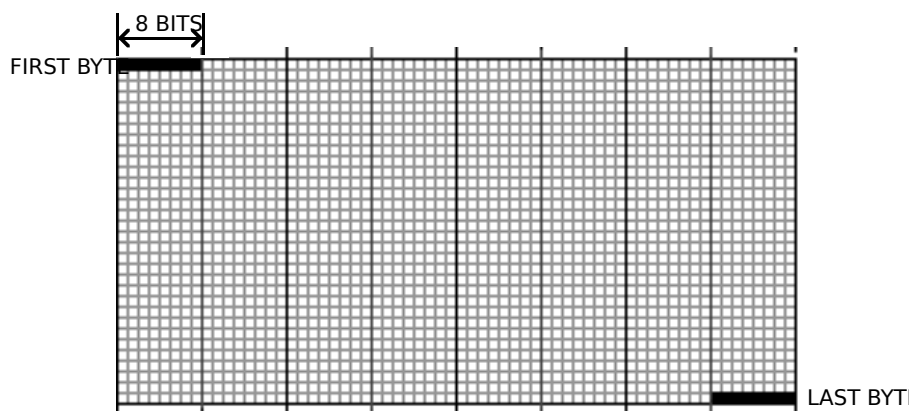


FIG 4 - A BIT IMAGE

A pixel image is essentially the same as a bit image, except that a number of bits, not just one bit, are assigned to each pixel. The number of bits per pixel in a pixel image is called the pixel depth.

rowBytes The offset in bytes from one row of the image to the next. The value must be even and less than 0x4000. For best performance it should be a multiple of 4. Bit 15 is used as a flag. If bit 15 = 1, the data structure is a PixMap structure, otherwise it is a BitMap structure. (The rowbytes bytes in a PixMap structure occupy the same bytes (fifth and sixth) as they do in a BitMap.)

bounds The boundary rectangle, which links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the pixel image into which QuickDraw can draw. All drawing in a colour graphics port occurs in the intersection of the boundary rectangle and the port rectangle (and, within that intersection, all drawing is cropped to the colour graphics port's visible region and its clipping region.)

As shown at Fig 5, QuickDraw assigns the entire screen as the boundary rectangle. The boundary rectangle shares the same local coordinate system as the port rectangle of the window.

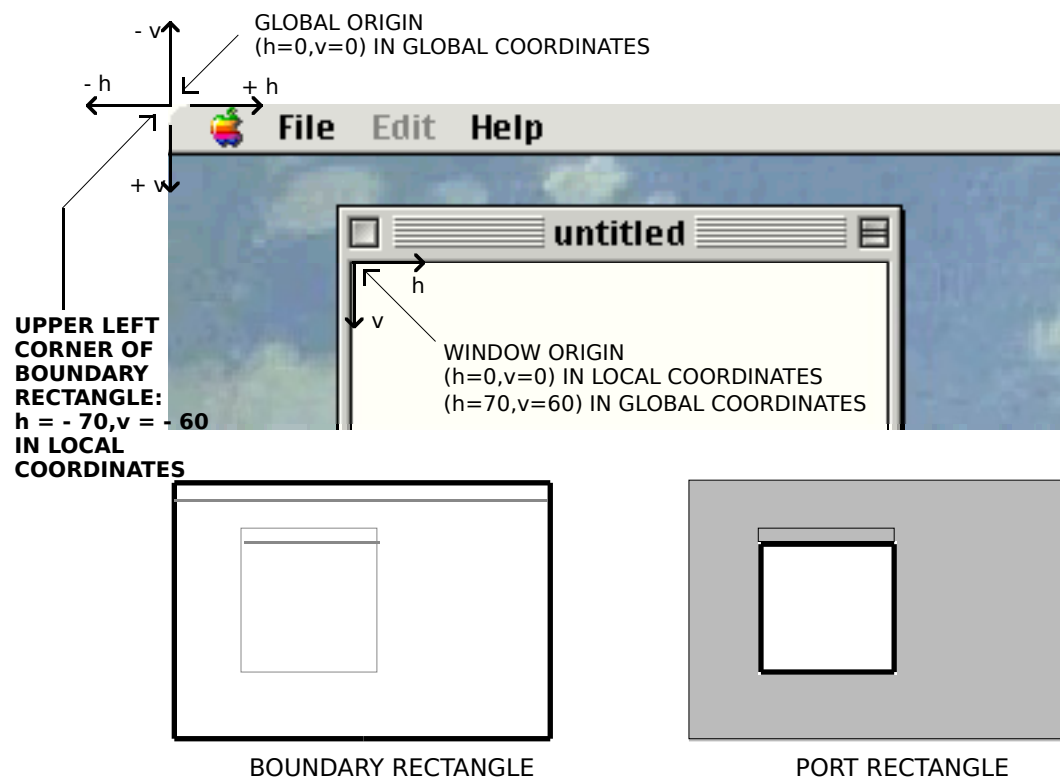


FIG 5 - LOCAL AND GLOBAL COORDINATE SYSTEMS, THE BOUNDARY RECTANGLE, AND THE PORT RECTANGLE

Do not use the `bounds` field to determine the size of the screen; instead, use the `gdRect` field of the `GDevice` structure (see below) for the screen.

- `pmVersion` The version number of QuickDraw that created this `PixMap` structure.
- `packType` The packing algorithm used to compress image data.
- `packSize` The size of the packed image in bytes.
- `hRes` The horizontal resolution of the image in pixels per inch, abbreviated as dpi (dots per inch). By default, the value here is `0x00480000` (for 72 dpi), but QuickDraw supports `PixMap` structures of other resolutions. For example, `PixMap` structures for scanners can have dpi resolutions of 150, 200, 300, or greater.
- `vRes` Describes the vertical resolution. (See `hRes`).
- `pixelType` The storage format for a pixel image. Indexed pixels are indicated by a value of 0. Direct pixels are specified by a value of `RGBDirect`, or 16. In the `PixMap` structure of the `GDevice` structure (see below) for a direct device, this field is set to the constant `RGBDirect` when the screen depth is set.
- `pixelSize` The pixel depth, that is, the number of bits used to represent a pixel. Indexed pixels can have sizes of 1, 2, 4, or 8 bits. Direct pixel sizes are 16 or 32 bits.
- `cmpCount` The number of components used to represent a colour for a pixel. With indexed pixels, each pixel is a single value representing an index in a colour table, so this field contains the value 1. With direct pixels, each pixel contains three components (one integer each for the intensities of red, green, and blue), so this field contains the value 3.
- `cmpSize` Specifies how large each colour component is. For indexed devices, it is the same value as that in the `pixelSize` field. For direct devices, each of the three colour components can be either 5 bits for a 16-bit pixel (one of these 16 bits is unused), or 8 bits for a 32 bit pixel (8 of these 32 bits are unused). (See Translation of RGB Colours to Pixel Values, below.)

planeBytes QuickDraw does not support multiple-plane images, so the value of this field is always 0.

pmTable Contains a handle to the ColorTable structure. ColorTable structures define the colours available for pixel images on indexed devices. (The Color Manager stores a colour table for the currently available colours in the graphic's device's CLUT. You use the Palette Manager to assign different colour tables to your different windows.)

You can create colour tables using either ColorTable structures or 'clut' resources. Pixel images on direct devices do not need a colour table because the colours are stored right in the pixel values. In such cases, pmTable points to a dummy colour table.

Pixel Patterns and Bit Patterns

Pixel Patterns

Three fields in the colour graphics port structure (pnPixPat, fillPixPat, and bkPixPat,) hold handles to a pixel pattern, a structure of type PixPat.

Pixel patterns, which define a repeating design, can use colours at any pixel depth, and can be of any width and height that is a power of 2. You can create your own pixel patterns in your program code, but it is usually simpler and more convenient to store them in resources of type 'ppat'. Fig 6 shows an 8-by-8 pixel 'ppat' resource being created using Resorcerer.

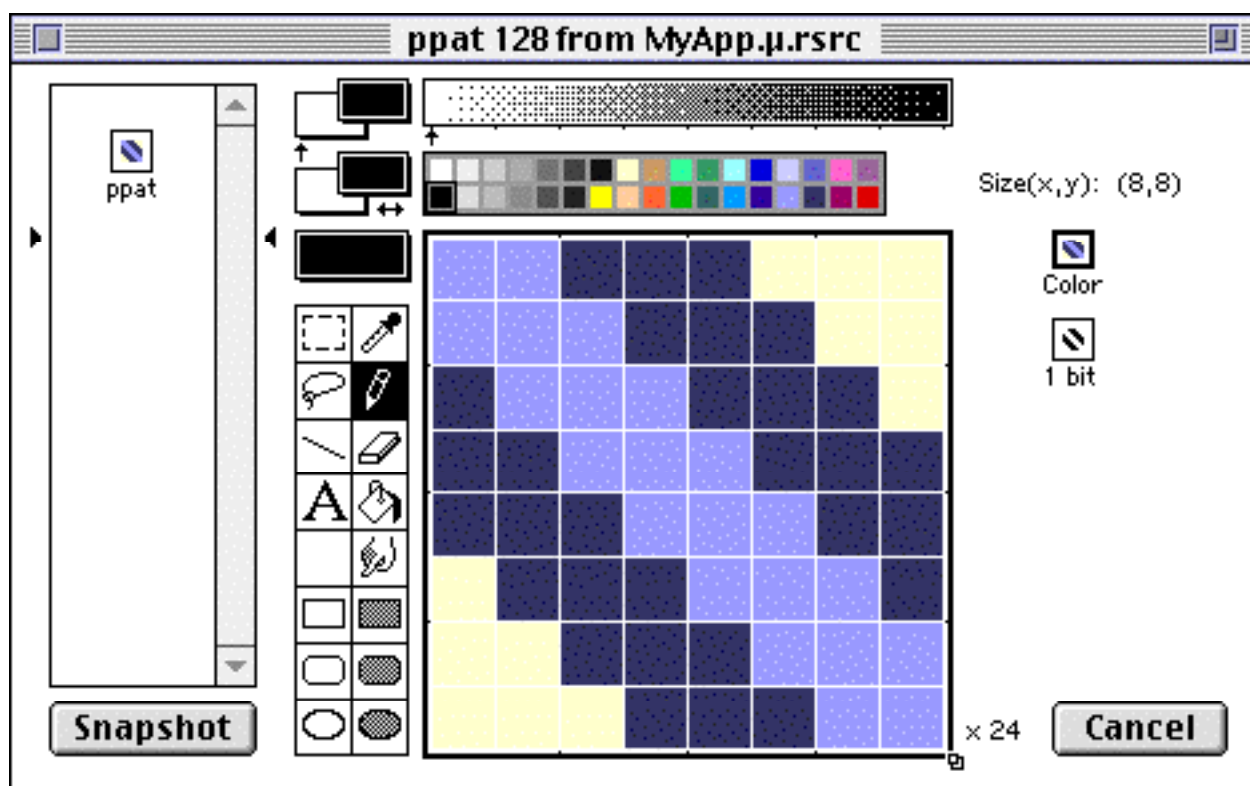


FIG 6 - CREATING A 'ppat' RESOURCE USING RESORCERER

Bit Patterns

Bit patterns date from the era of the black-and-white Macintosh, but may be assigned to the pnPixPat, fillPixPat, and bkPixPat fields of a colour graphics port. (PixPat structures can contain bit patterns as well as pixel patterns.) Bit patterns are defined in data structures of type Pattern, a 64-pixel image of a repeating design organised as an 8-by-8 pixel square.

Five bit patterns are pre-defined as QuickDraw global variables. The five pre-defined patterns are available not only through the QuickDraw globals but also as system

resources stored in the System resource file. Fig 7 shows images drawn using some of the 38 available system-supplied bit patterns.



FIG 7 - RECTANGLES DRAWN USING BIT PATTERNS IN THE SYSTEM RESOURCE FILE

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#'. Fig 8 shows a 'PAT' resource being created using Resorcerer, together with the contents of the pat field of the structure of type Pattern that is created when the resource is loaded.

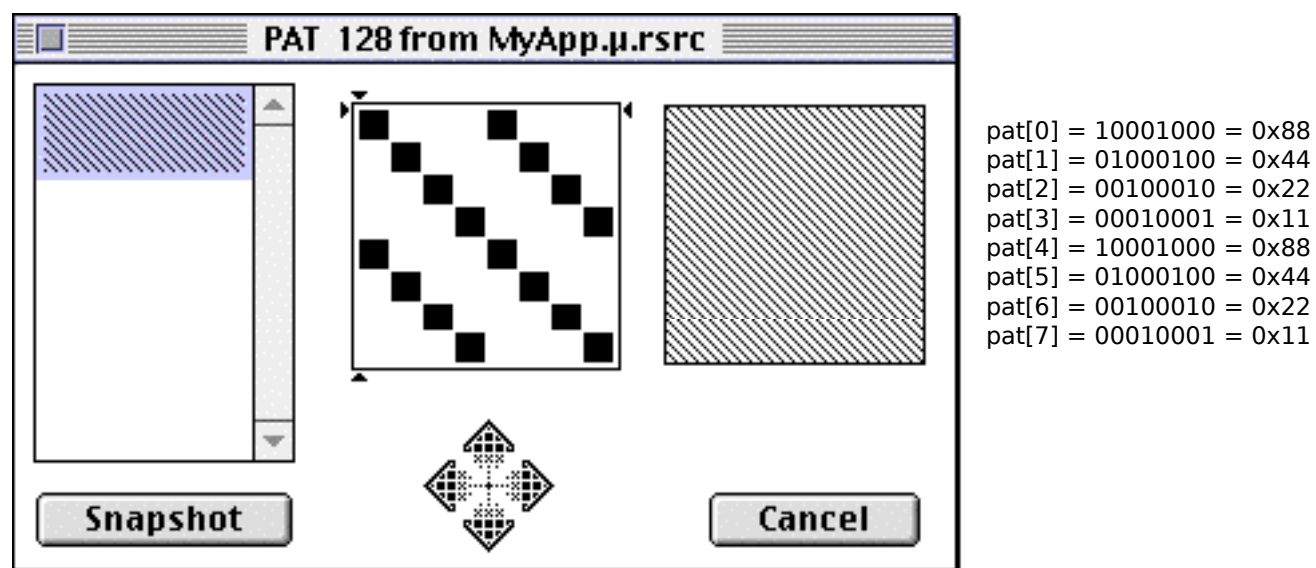


FIG 8 - CREATING A 'PAT' RESOURCE USING RESORCERER

Creating Colour Graphics Ports

Your application creates a colour graphics port using either the `GetNewCWindow`, `NewCWindow`, or `NewGWorld` function. These functions automatically call `OpenCPort` (which opens the port) and `InitCPort` (which and initialises the port).

Translation of RGB Colours to Pixel Values

As previously stated, the `baseAddr` field of the `CGrafPort` structure contains a pointer to the beginning of the onscreen pixel image. When your application specifies an RGB colour for a pixel in the pixel image, QuickDraw translates that colour into a value appropriate for display on the user's screen. QuickDraw stores this value in the pixel. The **pixel value** is a number used by system software and a graphics device to represent a colour. The translation from the colour you specify in an `RGBColor` structure to a pixel value is performed at the time you draw the colour. The process differs for direct and indexed devices as follows:

- When drawing on indexed devices, QuickDraw calls the Color Manager to supply the index to the colour that most closely matches the requested colour in the current device's CLUT. This index becomes the pixel value for that colour.

- When drawing on direct devices, QuickDraw truncates the least significant bits from the red, green and blue fields of the RGBColor structure. The result becomes the pixel value that QuickDraw sends to the graphics device.

Your application never needs to handle pixel values. However, to clarify the relationship between RGBColor structures and the pixels that are actually displayed, the following presents some examples of the derivation of pixel values from RGBColor structures.

Derivation of Pixel Values on Indexed Devices

Fig 9 shows the translation of an RGBColor structure to an 8-bit pixel value on an indexed device.

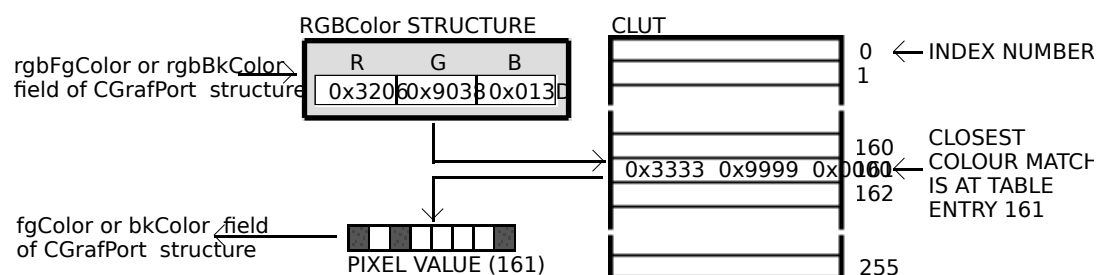


FIG 9 - TRANSLATING AN RGBColor STRUCTURE TO AN 8-BIT PIXEL VALUE

The application might later use GetCPixel to determine the colour of a particular pixel. As shown at Fig 10, the Color Manager uses the index number stored as the pixel value to find the RGBColor structure stored in the CLUT for that pixel's colour. Also as shown at Fig 10, this is not necessarily the exact colour first specified.

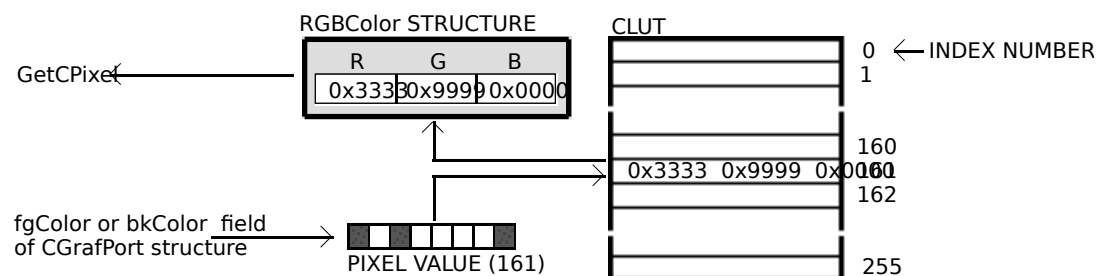


FIG 10 - TRANSLATING AN 8-BIT PIXEL VALUE ON AN INDEXED DEVICE TO AN RGBColor STRUCTURE

Derivation of Pixel Values on Direct Devices

Fig 11 shows how QuickDraw converts an RGBColor structure into a 16-bit pixel value on a direct device by storing the most significant 5 bits of each 16-bit field of the 48-bit RGBColor structure in the lower 15 bits of the pixel value, leaving an unused high bit. Fig 11 also shows how QuickDraw expands a 16-bit pixel value to a 48-bit RGBColor structure by dropping the unused high bit of the pixel value and inserting three copies of each 5-bit component and a copy of the most significant bit into each 16-bit field of the RGBColor structure. Note that the result differs, in the least significant 11 bits, from the original 48-bit value.

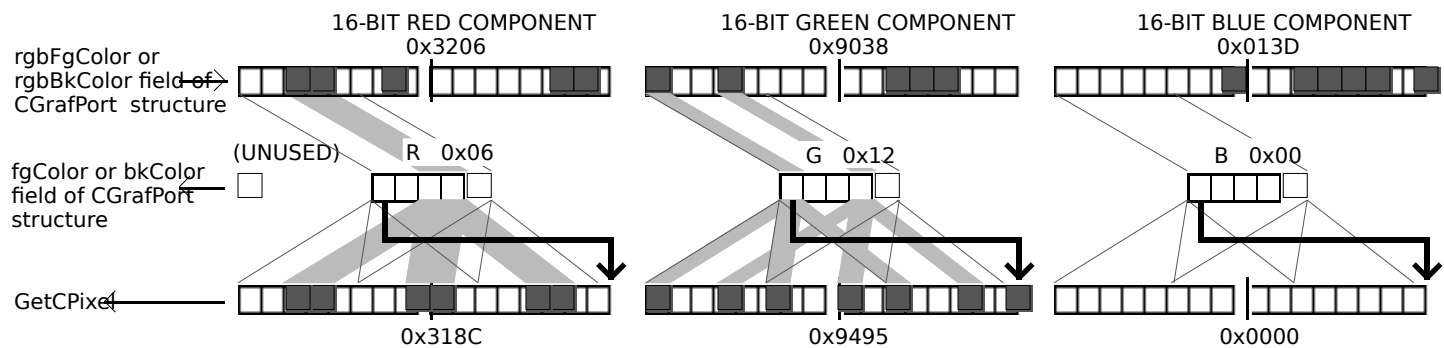


FIG 11 - TRANSLATING AN RGBColor STRUCTURE TO A 16 BIT PIXEL VALUE, AND FROM A 16-B STRUCTURE, ON A DIRECT DEVICE

Fig 12 shows how QuickDraw converts an RGBColor structure into a 32-bit pixel value on a direct device by storing the most significant 8 bits of each 16-bit field of the structure into the lower 3 bytes of the pixel value, leaving 8 unused bits in the high byte of the pixel value. Fig 12 also shows how QuickDraw expands a 32-bit pixel value to an RGBColor structure by dropping the unused high byte of the pixel value and doubling each of its 8-bit components. Note that the resulting 48-bit value differs in the least significant 8 bits of each component from the original RGBColor structure.

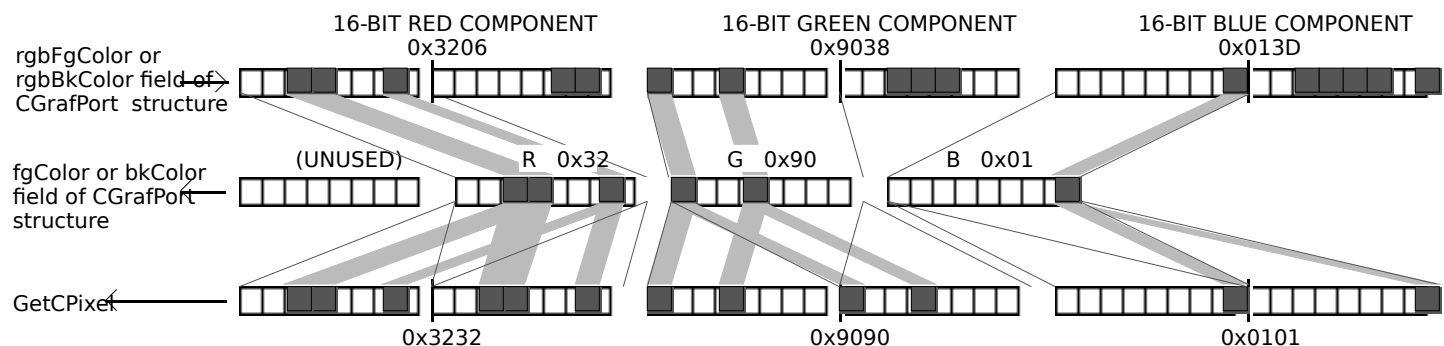


FIG 12 - TRANSLATING AN RGBColor STRUCTURE TO A 32 BIT PIXEL VALUE, AND FROM A 32-B STRUCTURE, ON A DIRECT DEVICE

Colours on Grayscale Screens

When QuickDraw displays a colour on a grayscale screen, it computes the luminance, or intensity of light, of the desired colour and uses that value to determine the appropriate gray value to draw.

A grayscale device can be a colour graphics device that the user sets to grayscale by using the Monitors and Sound control panel. For such a graphics device, Colour QuickDraw places an evenly spaced set of grays in the graphics device's CLUT.

By using the GetCTable function, your application can obtain the default colour tables for various graphics devices, including grayscale devices.

Graphics Devices and GDevice Structures

As previously stated, QuickDraw provides a device-independent interface. Your application can draw images in the graphics port for a window and QuickDraw automatically manages the path to the screen — even if the user is using multiple screens. QuickDraw communicates with a video device, such as a plug-in video card or a built-in video interface, by automatically creating and managing a structure of data type GDevice.

Types of Graphics Device

A **graphics device** is anything into which QuickDraw can draw. There are three types of graphics device:

- **Video devices**, such as video cards and built-in video interfaces, that control screens.
- **Offscreen graphics worlds**, which allow your application to build complex images offscreen before displaying them.⁴
- **Printing graphics ports** for printers.⁵

GDevice Structure

For a video device or an offscreen graphics world, QuickDraw automatically creates, and stores state information in, a GDevice structure. Note that printers do not have GDevice structures.

When the system starts up, QuickDraw uses information supplied by the Slot Manager to create and initialise a GDevice structure for each video device found during startup. When you use the `NewGWorld` function to create an offscreen graphics world, QuickDraw automatically creates a GDevice structure.

All existing GDevice structures are linked together in a list called a **device list**. The global variable `DeviceList` holds a handle to the first structure in the list. At any given time, exactly one graphics device is the **current device** (also called the **active device**), that is, the one in which drawing is actually taking place. A handle to its GDevice structure is stored in the global variable `TheGDevice`. By default, the GDevice structure corresponding to the first video device found is marked as the current device.

Your application generally never needs to create GDevice structures; however, it may need to examine GDevice structures to determine the capabilities of the user's screens. The GDevice structure is as follows:

```
struct GDevice
{
    short      gdRefNum;          // Reference Number of Driver.
    short      gdID;              // Client ID for search procedures.
    short      gdType;           // Type of device (indexed or direct).
    ITabHandle gdITable;         // Handle to inverse lookup table for Color Manager.
    short      gdResPref;        // Preferred resolution.
    SProcHndl  gdSearchProc;     // Handle to list of search functions.
    CProcHndl  gdCompProc;       // Handle to list of complement functions.
    short      gdFlags;          // Graphics device flags.
    PixMapHandle gdPMap;         // Handle to pixel map for displayed image.
    long       gdRefCon;         // Reference value.
    Handle     gdNextGD;        // Handle to next GDevice structure.
    Rect       gdRect;          // Device's global boundaries.
    long       gdMode;           // Device's current mode.
    short      gdCCBytes;        // Width of expanded cursor data.
    short      gdCCDepth;        // Depth of expanded cursor data.
    Handle     gdCCXData;        // Handle to cursor's expanded data.
    Handle     gdCCXMask;        // Handle to cursor's expanded mask.
    long       gdReserved;       // (Reserved. Must be 0.)
};
typedef struct GDevice GDevice;
typedef GDevice *GDPtr, **GDHandle;
```

Main Field Descriptions

`gdType` The general type of graphics device. See Flag Bits of `gdType` Field, below.

⁴ See Chapter 13 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

⁵ See Chapter 15 — Printing.

| | |
|----------|---|
| gdITable | Points to an inverse table . An inverse table is a special Color Manager data structure arranged in such a manner that, given an arbitrary RGB colour, its pixel value (that is, its index number in the CLUT) can be found quickly. |
| gdFlags | Device attributes (that is, whether the device is a screen, whether it is the main screen, whether it is set to black-and-white or colour, whether it is the active device, etc.) See Flag Bits of gdType Field, below. |
| gdPMap | Contains a handle to the pixel map (PixMap) structure, which contains the dimensions of the image buffer, along with the characteristics of the graphics device (resolution, storage format, pixel depth, and colour table. QuickDraw automatically synchronises the pixel map's colour table (ColorTable) structure with the CLUT on the video device. |
| gdNextGD | A handle to the next graphics device in the device list. If this is the last graphics device in the device list, this field contains 0. |
| gdRect | The boundary rectangle of the graphics device represented by the GDevice structure. The main screen has the upper-left corner of the rectangle set to (0,0). All other graphics devices are relative to this point. |

Flag Bits of gdType Field

| Constant | Bit | Meaning |
|-----------------|------------|--------------------|
| clutType | 0 | CLUT device. |
| fixedType | 1 | Fixed CLUT device. |
| directType | 2 | Direct device. |

Flag Bits of gdFlags Field

| Constant | Bit | Meaning |
|-----------------|------------|--|
| gdDevType | 0 | 0 = black-and-white. 1 = colour. |
| burstDevice | 7 | Device supports block transfer. |
| ext32Device | 8 | Device must be used in 32-bit mode. |
| ramInit | 10 | Device was initialised from RAM. |
| mainScreen | 11 | Device is the main screen. |
| allInit | 12 | All devices were initialised from 'scrm' resource. |
| screenDevice | 13 | Device is a screen device. |
| noDriver | 14 | GDevice structure has no driver. |
| screenActive | 15 | Device is current device. |

Setting a Device's Pixel Depth

The gdPMap field of the GDevice structure contains a handle to a PixMap structure which, in turn, contains the PixelSize field to which is assigned the pixel depth of the device.

The Monitors and Sound control panel is the user interface for changing the pixel depth of video devices. Since the user can control the capabilities of the video device, your application should be flexible, that is, although it may have a preferred pixel depth, it should do its best to accommodate less than ideal conditions. Your application can use the SetDepth function to change the pixel depth, but it should not do so without the consent of the user. Before calling SetDepth, you should use the HasDepth function to determine whether the available hardware can support the pixel depth you require.

Other Graphics Managers

In addition to the QuickDraw functions, several other collections of system software functions are available to assist you in drawing images.

Palette Manager

To provide more sophisticated colour support on indexed graphics devices, your application can use the Palette Manager. The Palette Manager allows your application to specify sets of colours that it needs on a window-by-window basis. On a video device that uses a variable CLUT, your application can use the Palette Manager to display any number of palettes (that is, sets of colours) consisting of 256 colours each. Remember, though, that only one set of colours (palette) can be displayed at any one time.

Color Picker Utilities

To solicit colour choices from users, your application can use the Color Picker Utilities. The Color Picker Utilities also provide functions that allow your application to convert between colours specified in `RGBColor` structures and colours specified for other colour models, such as the CMYK (cyan, magenta, yellow, black) model used for many colour printers. (See Chapter 23 — Miscellany.)

Coping With Multiple Monitors

Image optimisation and window dragging in a multiple monitors environment is addressed at Chapter 23 — Miscellany.

Relevant QuickDraw Constants, Data Types, and Functions

Constants

Flag Bits of `gdType` Field of `GDevice` Structure

| | |
|-------------------------|-----|
| <code>clutType</code> | = 0 |
| <code>fixedType</code> | = 1 |
| <code>directType</code> | = 2 |

Flag Bits of `gdFlags` Field of `GDevice` Structure

| | |
|---------------------------|------|
| <code>gdDevType</code> | = 0 |
| <code>burstDevice</code> | = 7 |
| <code>ext32Device</code> | = 8 |
| <code>ramInit</code> | = 10 |
| <code>mainScreen</code> | = 11 |
| <code>allInit</code> | = 12 |
| <code>screenDevice</code> | = 13 |
| <code>noDriver</code> | = 14 |
| <code>screenActive</code> | = 15 |

Pixel Type

| | | |
|------------------------|------|---|
| <code>RGBDirect</code> | = 16 | // 16 and 32 bits-per-pixel <code>pixelType</code> value. |
|------------------------|------|---|

Data Types

Colour Graphics Port

`struct CGrafPort`


```

{
    short          device;                // Device-specific information.
    PixMapHandle  portPixMap;            // Handle to pixel map.
    short         portVersion;           // Flags and version number.
    Handle        grafVars;             // Handle to additional colour fields.
    short         chExtra;               // Extra width added to non-space characters.
    short         pnLocHFrac;           // Fractional horizontal pen position.
    Rect          portRect;             // Port rectangle.
    RgnHandle     visRgn;               // Visible region.
    RgnHandle     clipRgn;              // Clipping region.
    PixPatHandle  bkPixPat;             // Background pattern
    RGBColor      rgbFgColor;           // Requested foreground colour.
    RGBColor      rgbBkColor;           // Requested background colour
    Point         pnLoc;                // Pen location.
    Point         pnSize;               // Pen size.
    short         pnMode;               // Pattern mode.
    PixPatHandle  pnPixPat;             // Pen pattern.
    PixPatHandle  fillPixPat;          // Fill pattern.
    short         pnVis;                // Pen visibility.
    short         txFont;               // Font number for text.
    Style         txFace;               // Text font style.
    SInt8         filler;
    short         txMode;               // Text source mode.
    short         txSize;               // Font size for text.
    Fixed         spExtra;              // Extra width added to space characters.
    long          fgColor;              // Actual foreground colour.
    long          bkColor;              // Actual background colour.
    short         colrBit;              // Colour bit (reserved).
    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.)
    CQDProcsPtr  grafProcs;            // Pointer to low-level drawing routines.
};
typedef struct CGrafPort CGrafPort,*CGrafPtr;
typedef CGrafPtr CWindowPtr;

```

GrafVars

```

struct GrafVars
{
    RGBColor      rgbOpColor;           // Color for addPin, subPin and average.
    RGBColor      rgbHiliteColor;      // Color for highlighting.
    Handle        pmFgColor;           // Palette handle for foreground color.
    short         pmFgIndex;           // Index value for foreground.
    Handle        pmBkColor;           // Palette handle for background color.
    short         pmBkIndex;           // Index value for background.
    short         pmFlags;             // Flags for Palette Manager.
};
typedef struct GrafVars GrafVars,*GVarPtr,**GVarHandle;

```

Pixel Map

```

struct PixMap
{
    Ptr           baseAddr;            // Pointer to image data.
    short         rowBytes;            // Flags, and bytes in a row.
    Rect          bounds;              // Boundary rectangle.
    short         pmVersion;           // Pixel Map version number.
    short         packType;            // Packing format.
    long          packSize;            // Size of data in packed state.
    Fixed         hRes;                // Horizontal resolution in dots per inch.
    Fixed         vRes;                // Vertical resolution in dots per inch.
    short         pixelType;           // Format of pixel image.
    short         pixelSize;           // Physical bits per pixel.
    short         cmpCount;            // Number of components in each pixel.
    short         cmpSize;             // Number of bits in each component.
    long          planeBytes;          // Offset to next plane.
    CTabHandle    pmTable;             // Handle to a colour table for this image.
    long          pmReserved;          // (Reserved.)
};
typedef struct PixMap PixMap,*PixMapPtr,**PixMapHandle;

```

Color Table

```

struct ColorTable
{
    long          ctSeed;              // Unique identifier for table.
};

```

```

    short          ctFlags;           // High bit: 0 = PixMap, 1 = device.
    short          ctSize;           // Number of entries in ctTable minus 1.
    CSpecArray     ctTable;         // Array of ColorSpec structures.
};
typedef struct ColorTable ColorTable;
typedef ColorTable *CTabPtr;
typedef CTabPtr *CTabHandle;

```

ColorSpec

```

struct ColorSpec
{
    short          value;           // Index or other value.
    RGBColor       rgb;           // True color.
};
typedef struct ColorSpec ColorSpec;
typedef ColorSpec *ColorSpecPtr;
typedef ColorSpec CSpecArray[1];

```

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;

```

Pixel Pattern

```

struct PixPat
{
    short          patType;       // Type of pattern.
    PixMapHandle   patMap;       // The pattern's pixel map.
    Handle         patData;      // Pixel map's data.
    Handle         patXData;     // Expanded Pattern data (internal use).
    short          patXValid;    // Flags whether expanded Pattern valid.
    Handle         patXMap;     // Handle to expanded Pattern data (reserved).
    Pattern        pat1Data;    // Bit map's data.
};
typedef struct PixPat PixPat;
typedef PixPat *PixPatPtr;
typedef PixPatPtr *PixPatHandle;

```

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.

GDevice

```

struct GDevice
{
    short          gdRefNum;      // Reference Number of Driver.
    short          gdID;         // Client ID for search procedures.
    short          gdType;       // Type of device (indexed or direct).
    ITabHandle     gdITable;     // Handle to inverse lookup table for Color Manager.
    short          gdResPref;    // Preferred resolution.
    SProcHndl      gdSearchProc; // Handle to list of search functions.
};

```



```

doInitManagers();

// ..... set
up menu bar and menus

menubarHdl = GetNewMBar(rMenubar);
if(menubarHdl == NULL)
    ExitToShell();
SetMenuBar(menubarHdl);

menuHdl = GetMenuHandle(mApple);
if(menuHdl == NULL)
    ExitToShell();
else
    AppendResMenu(menuHdl,'DRVR');

if(!(doCheckMonitor()))
{
    GetIndString(theString,rIndexedStrings,sMonitorInadequate);
    doMonitorAlert(theString);
    menuHdl = GetMenuHandle(mDemonstration);
    DisableItem(menuHdl,0);
}
else
{
    if(gStartupPixelDepth > 8)
    {
        menuHdl = GetMenuHandle(mDemonstration);
        DisableItem(menuHdl,0);
    }
}

DrawMenuBar();

// ..... open windows, set font size, show windows, move windows

if(!(windowPtr = GetNewCWindow(rWindow,NULL,(WindowPtr)-1)))
    ExitToShell();

SetPort(windowPtr);
TextSize(10);

//
..... enter eventLoop

gDone = false;

while(!gDone)
{
    if(WaitNextEvent(everyEvent,&EventStructure,MAXLONG,NULL))
        doEvents(&EventStructure);
}

// ..... doInitManagers
void doInitManagers(void)
{
    MaxApplZone();
    MoreMasters();

    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);

    InitCursor();
    FlushEvents(everyEvent,0);

    RegisterAppearanceClient();
}

// ..... doEvents
void doEvents(EventRecord *eventStrucPtr)

```

```

{
    Slnt8    charCode;
    Slnt32   menuChoice;
    Slnt16   menuID, menuItem;
    Slnt16   partCode;
    WindowPtr windowPtr;
    Str255   itemName;
    Slnt16   daDriverRefNum;
    Rect     theRect;

    switch(eventStrucPtr->what)
    {
    case keyDown:
    case autoKey:
        charCode = eventStrucPtr->message & charCodeMask;
        if((eventStrucPtr->modifiers & cmdKey) != 0)
        {
            menuChoice = MenuEvent(eventStrucPtr);
            menuID = HiWord(menuChoice);
            menuItem = LoWord(menuChoice);
            if(menuID == mFile && menuItem == iQuit)
                gDone = true;
        }
        break;

    case mouseDown:
        if(partCode = FindWindow(eventStrucPtr->where,&windowPtr))
        {
            switch(partCode)
            {
            case inMenuBar:
                menuChoice = MenuSelect(eventStrucPtr->where);
                menuID = HiWord(menuChoice);
                menuItem = LoWord(menuChoice);

                if(menuID == 0)
                    return;

                switch(menuID)
                {
                case mApple:
                    if(menuItem == iAbout)
                        SysBeep(10);
                    else
                    {
                        GetMenuItemText(GetMenuHandle(mApple),menuItem,itemName);
                        daDriverRefNum = OpenDeskAcc(itemName);
                    }
                    break;

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

                case mDemonstration:
                    if(menuItem == iSetDepth)
                        doSetMonitorPixelFormat();
                    else if(menuItem == iRestoreDepth)
                        doRestoreMonitorPixelFormat();
                    break;
                }
                HiliteMenu(0);
                break;

            case inDrag:
                DragWindow(windowPtr,eventStrucPtr->where,&qd.screenBits.bounds);
                theRect = windowPtr->portRect;
                theRect.right = windowPtr->portRect.left + 250;
                InvalRect(&theRect);
                break;
            }
        }
        break;

    case updateEvt:
        windowPtr = (WindowPtr) eventStrucPtr->message;
        BeginUpdate(windowPtr);
        SetPort(windowPtr);

```



```

    case fixedType:
        DrawString("\pIt is is an indexed device with fixed CLUT.");
        break;

    case directType:
        DrawString("\pIt is a direct device.");
        break;
}

// .....
Get Handle to Pixel Map

    pixMapHdl = (**deviceHdl).gdPMap;

//
.....
determine pixel depth

    MoveTo(10,70);
    DrawString("\pPixel depth = ");
    pixelDepth = (**pixMapHdl).pixelSize;
    NumToString((SInt32) pixelDepth,theString);
    DrawString(theString);

// ..... Get Device's
Global Boundaries

    theRect = (**deviceHdl).gdRect;

// ..... determine bytes per row and total pixel image bytes

    MoveTo(10,90);
    bytesPerRow = (**pixMapHdl).rowBytes & 0x7FFF;
    DrawString("\pBytes per row = ");
    NumToString((SInt32) bytesPerRow,theString);
    DrawString(theString);

    MoveTo(10,105);
    DrawString("\pTotal pixel image bytes = ");
    NumToString((SInt32) bytesPerRow * theRect.bottom,theString);
    DrawString(theString);

// ..... convert device's global boundaries to coordinates of graphics port

    GlobalToLocal(&topLeft(theRect));
    GlobalToLocal(&botRight(theRect));

    MoveTo(10,125);
    DrawString("\pBoundary rectangle top = ");
    NumToString((SInt32) theRect.top,theString);
    DrawString(theString);

    MoveTo(10,140);
    DrawString("\pBoundary rectangle left = ");
    NumToString((SInt32) theRect.left,theString);
    DrawString(theString);

    MoveTo(10,155);
    DrawString("\pBoundary rectangle bottom = ");
    NumToString((SInt32) theRect.bottom,theString);
    DrawString(theString);

    MoveTo(10,170);
    DrawString("\pBoundary rectangle right = ");
    NumToString((SInt32) theRect.right,theString);
    DrawString(theString);

// ..... Get Pointer to Colour
Graphics Port

    cgrafPtr = (CGrafPtr) windowPtr;

// ..... determine requested
background colour

    MoveTo(10,190);
    GetBackColor(&blueColour);
    DrawString("\pRequested background colour (rgb) = ");
    MoveTo(10,205);

```



```

NumToString((SInt32) blueColour.red,theString);
DrawString(theString);
DrawString("\p ");
NumToString((SInt32) blueColour.green,theString);
DrawString(theString);
DrawString("\p ");
NumToString((SInt32) blueColour.blue,theString);
DrawString(theString);

// ..... get actual
colour (pixel value)

pixelValue = cgrafPtr->bkColor;

// ..... if direct device, extract colour components, else retrieve colour table index

MoveTo(10,220);

if(deviceType == directType)
{
    if(pixelDepth == 16)
    {
        redComponent    = pixelValue >> 10 & 0x0000001F;
        greenComponent  = pixelValue >> 5 & 0x0000001F;
        blueComponent   = pixelValue & 0x0000001F;
    }
    else if (pixelDepth == 32)
    {
        redComponent    = pixelValue >> 16 & 0x000000FF;
        greenComponent  = pixelValue >> 8 & 0x000000FF;
        blueComponent   = pixelValue & 0x000000FF;
    }
}

DrawString("\pBackground colour used (rgb) = ");
MoveTo(10,235);

NumToString((SInt32) redComponent,theString);
DrawString(theString);
DrawString("\p ");

NumToString((SInt32) greenComponent,theString);
DrawString(theString);
DrawString("\p ");

NumToString((SInt32) blueComponent,theString);
DrawString(theString);
}
else if(deviceType == clutType || deviceType == fixedType)
{
    DrawString("\p Background colour used (color table index) = ");
    MoveTo(10,235);
    NumToString((SInt32) pixelValue,theString);
    DrawString(theString);
}

// ..... Get
Handle to Colour Table

colorTableHdl = (*pixMapHdl)->pmTable;

// ..... if any entries in colour table, draw the colours

MoveTo(250,20);
DrawString("\pColour table in GDevice's PixMap:");

entries = (*colorTableHdl)->ctSize;

if(entries < 2)
{
    MoveTo(260,105);
    DrawString("\pDummy (one entry) colour table only.");
    MoveTo(260,120);
    DrawString("\pTo get some entries, set the monitor to");
    MoveTo(260,135);
    DrawString("\p 256 colours, causing it to act like an");
    MoveTo(260,150);
    DrawString("\p indexed device.");
    SetRect(&theRect,250,28,458,236);
    FrameRect(&theRect);
}

```


Demonstration Program Comments

When this program is first run, the user should:

- Drag the window to various position on the main screen, noting the changes to the coordinates of the boundary rectangle.
- Open the Monitors and Sound control panel and, depending on the characteristics of the user's system:
 - Change between the available resolutions, noting the changes in the bytes per row and total pixel image bytes figures displayed in the window.
 - Change between the available colour depths, noting the changes to the pixel depth and total pixel image bytes figures, and the background colour used figures, displayed in the window.
- Note that, when 256 or less colours are displayed on a direct device (in colours and grays), the device creates a CLUT and operates like a direct device. In this case, the background colour used figure is the colour table entry (index), and the relevant colour in the colour table display is framed in white.

Assuming the user's monitor is a direct colour device, the user should then run the program again with the monitor set to display 256 colours prior to program start. The Demonstration menu and its items will be enabled. The user should then choose the items in the Demonstration menu to set the monitor to a pixel depth of 16 and back to the startup pixel depth.

main

Before DrawMenuBar is called, a call to the application-defined function doCheckMonitor assigns the startup pixel depth to a global variable and determines whether the main device supports 16-bit colour. If the main device does not support 16-bit colour, the Demonstration menu is disabled. If the main device does support support 16-bit colour, the Demonstration menu is disabled only if the current pixel depth is not 8 (256 colours) or less.

doEvents

In the case of a mouse-down event, in the inDrag case, when the user releases the mouse button, the left half of the window is invalidated, causing the left half to be redrawn with the new boundary rectangle coordinates.

doDisplayInformation

In the first three lines, RGB colours are assigned to the window's colour graphics port's rgbFgColor and rgbBkColor fields. The call to EraseRect causes the content region to be filled with the background colour.

Get Device List

The call to LMGetDeviceList gets a handle to the first GDevice structure in the device list. The device list is then "walked" in the while loop. For every video device found in the list, the variable videoDeviceCount is incremented. GetNextDevice gets a handle to the next device in the device list.

Get Main Device

LMGetMainDevice gets a handle to the startup device, that is, the device on which the menu bar appears.

The call to BitTest with the gdDevType flag determines whether the main (startup) device is a colour or black-and-white device. In the next block, the gdType field of the GDevice structure is examined to determine whether the device is an indexed device with a variable CLUT, an indexed device with a fixed CLUT, or a direct device (or a direct device set to display 256 colours or less and, as a consequence, acting like an indexed device).

Get Handle to Pixel Map

At the first line of this block, a handle to the GDevice structure's pixel map is retrieved from the gdPMap field.

In the next block, the pixel depth is extracted from the PixMap structure's pixelSize field.

Get Device's Global Boundaries

At the first line of this block, the device's global boundaries are extracted from the GDevice structure's gdRect field.

At the next block, the number of bytes in each row in the pixel map is determined. (The high bit in the rowBytes field of the PixMap structure is a flag which indicates whether the data structure is a PixMap structure or a BitMap structure.)

At the next block, the bytes per row value is multiplied by the height of the boundary rectangle to arrive at the total number of bytes in the pixel image.

The two calls to GlobalToLocal convert the boundary rectangle coordinates to coordinates local to the colour graphics port.

Get Pointer To Colour Graphics Port

The first line simply casts the windowPtr to a pointer to a colour graphics port so that, later on, the bkColor field can be accessed.

The next block gets the current (requested) background colour using the function GetBackColor, and then extracts the red, green, and blue components.

At the next line, the pixel value in the bkColor field of the colour graphics port is retrieved. This is an SInt32 value holding either the red, green, and blue components of the background colour actually used for drawing (direct device) or the colour table entry used for drawing (indexed devices).

For direct devices with a pixel depth of 16, the first 15 bits hold the three RGB components. For direct devices with a pixel depth of 32, the first 24 bits hold the RGB components. These are extracted in the if(deviceType == directType) block. For indexed devices the value is simply the colour table entry (index) determined by the Color Manager to represent the nearest match to the requested colour.

Get Handle To Colour Table

The first and fourth lines get a handle to the colour table in the GDevice structure's pixel map and the number of entries in that table.

The final block paints small coloured rectangles for each entry in the colour table. If the main device is an indexed device (or if it is a direct device set to display 256 colours or less), the colour table entry being used as the best match for the requested background colour is outlined in white.

doCheckMonitor

doCheckMonitor is called at program start to determine whether the main device supports 16-bit colour and, if it does, to assign the main device's pixel depth at startup to the global variable gStartupPixelDepth.

The call to LMGetMainDevice gets a handle to the main device's GDevice structure. The function HasDepth is used to determine whether the device supports 16-bit colour. The pixel depth is extracted from the pixelSize field of the PixMap structure in the GDevice structure.

doSetMonitorPixelDepth

doSetMonitorPixelDepth is called when the first item in the Demonstration menu is chosen to set the main device's pixel depth to 16.

If the current pixel depth determined at the first two lines is not 16, a string is retrieved from a 'STR#' resource and passed to the application-defined function doMonitorAlert, which displays a movable modal alert box advising the user that the monitor's bit depth is about to be changed to 16. When the user dismisses the alert box, SetDepth sets the main device's pixel depth to 16.

If the current pixel depth is 16, the last two lines display an alert box advising the user that the device is currently set to that pixel depth.

doRestoreMonitorPixelDepth

doRestoreMonitorPixelDepth is called when the second item in the Demonstration menu is chosen to reset the main device's pixel depth to the startup pixel depth.

If the current pixel depth determined at the first two lines is not equal to the startup pixel depth, a string is retrieved from a 'STR#' resource and passed to the application-defined function doMonitorAlert, which displays a movable modal alert box advising the user that the monitor's bit depth is about to be changed to the startup pixel depth. When the user dismisses the alert box, SetDepth sets the main device's pixel depth to the startup pixel depth.

If the current pixel depth is the startup pixel depth, the last two lines display an alert box advising the user that the device is currently set to that pixel depth.