Offscreen Pixel Maps Drawing into an offscreen pixel map

Macintosh Technical Note #120 describes the **Color QuickDraw** graphics environment in great detail. The Note is over 50 pages in size, and will probabably grow. The information below is an overview of that Note.

The Macintosh has memory-mapped video. What you see on the screen is just the visual representation of a part of memory that is reserved for the video hardware. If you change the contents of a memory location in this part of memory, then you will see the corresponding location on the screen change when the video hardware draws the next frame or field of video. The Macintosh uses QuickDraw to draw on-screen images by writing to the video memory.

QuickDraw can be told to draw to any RAM address space. When you write to the screen RAM, you might be writing to a NuBus card, but it can still be considered RAM. QuickDraw can be told to draw into a part of memory that isn't reserved for video hardware, such as into your application heap. If you draw into memory in your heap, you won't see any results. Why would you want to do this? There are many uses for offscreen drawing. You might want to provide storage for a paint-style document or to smoothly animate an image.

To do offscreen drawing, you will need to build three data structures: a CGrafPort, a PixMap and a GDevice. You will also need a table to define the colors involved with drawing to and copying from the offscreen image: the ColorTable and the inverse color table. You will also need a buffer to hold the image itself.

You should build a new CGrafPort when you draw offscreen rather than using an existing CGrafPort . Some people use just one CGrafPort to share between on-screen and offscreen graphics environments, and switch their PixMap structures to switch between drawing on-screen and offscreen. That works, but if the offscreen and on-screen environments have a different clipRgn, visRgn, pen characteristic, portRect or any other characteristics that are different, then they must be updated anyway. If you create a CGrafPort that is dedicated to one graphics environment, then a simple call to **SetPort** will switch everything for you.

Graphics Devices and the GDevice structure describe color environments. In theory, there are virtually three hundred trillion colors available to your application with **Color QuickDraw** through the 48-bit RGBColor record. In reality, there are never this many colors available, in fact there might be only two. Colors are mapped from theoretical colors that you specify to the pixel value of the closest available color in the current color environment. But finding the closest available color to an RGBColor in a color table means searching the entire color table for the one closest color. If this color look-up is done often, the performance hit can be quite drastic. An example of a very bad case would be **CopyBits**, where every pixel value in the source image is converted to an RGBColor by looking it up in the color table of the source PixMap. If the color table of the destination PixMap has to be searched to find the closest available color for every pixel in the source PixMap, then the performance of even the most straightforward **CopyBits** call is many times slower than it has to be.

To avoid this performance hit, the current GDevice provides an inverse table and a device type which are used to determine the available set of colors. Inverse tables are anti-color tables. Where color tables give you a color for a

given pixel value, inverse tables give you a pixel value for a given color. Every ColorTable has a corresponding inverse table. The device type specifies whether the color environment uses the indexed-color, fixed-color or direct-color model. In the direct-color model, the inverse table is empty.

When you specify a color, **Color QuickDraw** takes the RGBColor specification and converts it into a value that can be used as an index into the inverse table of the current GDevice. To do this, the top few significant bits of each color component are used to generate a 16-bit word, which is used to index into the inverse table. The value in the inverse table at that index is the pixel value which best represents that color in the current color environment. The number of bits of each component that are used is determined by what's called the "resolution" of the inverse table. Almost always, the resolution of an inverse table is four bits, meaning the most significant four bits of each component are used to form the index into the inverse table.

The same process is used when **CopyBits** is called with an index-color destination. Each pixel in the source pixel image is converted to an RGBColor either by doing a table look-up of the source pixel map's color table if the source pixel image uses indexed colors, or the pixel value is expanded to an RGBColor record if the source pixel image uses direct colors. The resulting RGBColor is then used to look up a pixel value in the inverse table of the current GDevice, and this pixel value is put into the destination pixel image.

Usually, inverse-table look-up involves an extra step to find what are called "hidden colors" using proprietary information that is stored at the end of the inverse table. With an inverse-table resolution of four, only sixteen shades of any particular component can be distinguished, and that's often not enough. An inverse-table with a resolution of five is much larger, but it still only gives you 32 shades of any component. Hidden colors are looked up after the normal inverse-table look-up to give a much more accurate representation of the specified color in the current color environment than the inverse-table look-up alone can produce. The hidden colors are ignored if arithmetic transformations or dithering is used.

When a new color table is assigned to a PixMap or when its existing color table is modified, then a new corresponding inverse table should be generated for the GDevice that will be used when drawing into that environment. Normally, this happens automatically without you having to do any more than inform **Color QuickDraw** of the change.

The following example demonstrates how to draw something in an offscreen pixel map, and then use **CopyBits** to copy it back to the screen. It handles the case of multiple screens with different pixel depths. Before making any calls to **Color QuickDraw**, you must make sure it is present.

> **Note:** For optimal performance, you want to make sure that the source and destination pixel maps are aligned.

Example

```
#define    OffLeft     30
#define    OffTop      30
#define    OffBottom   250
#define    OffRight    400
```

```
/* These constants for the bounds of the offscreen PixMap are chosen because
    we know what the extent of the drawing will be and we want to restrict the
    size of the map as much as possible. */

/* typedef BitMapPtr for use during CopyBits operation */
typedef BitMap *BitMapPtr;

long        offRowBytes, sizeOfOff;
Ptr         myBits;
Rect        destRect, globRect, bRect;
short       theDepth, i, err;
CGrafPort   myCGrafPort;
CGrafPtr    myCGrafPtr;
CTabHandle  ourCMHandle;
GDHandle    theMaxDevice, oldDevice;
Point       tempP;


/* Create a color window on screen. */

    myWindow = GetNewCWindow(SomeID,nil,(WindowPtr) -1);

    /* set to this port for the LocalToGlobals that follow */
    SetPort((WindowPtr) myWindow);

    SetRect(&bRect,OffLeft,OffTop,OffRight,OffBottom);
    if (!SectRect(&(*myWindow).portRect,&bRect,&globRect))
        ExitToShell();              /*nothing to do, clean up and EXIT*/

/* Since there are no topLeft or bottomRight fields for a Rect, you should*/
/* use the macros which give you access to the equivalent that are provided */
/* in the THINK C environment in THINK.h. The example below does not */
/* assume access to these handy macros. */

    tempP.v = globRect.top;
    tempP.h = globRect.left;
    LocalToGlobal(&tempP);
    globRect.top = tempP.v;
    globRect.left = tempP.h;

    tempP.v = globRect.bottom;
    tempP.h = globRect.right;
    LocalToGlobal(&tempP);
    globRect.bottom = tempP.v;
    globRect.right = tempP.h;

    theMaxDevice = GetMaxDevice(&globRect);
                                    / *get the maxDevice*/

    oldDevice = GetGDevice();       /* save theGDevice so we */
                                    /*   can restore it later*/
    SetGDevice(theMaxDevice);       /* Set to the maxdevice*/

/* Now you can set up the offscreen pixel map.*/
```

```
myCGrafPtr = &myCGrafPort;     /* initialize this color port */
OpenCPort(myCGrafPtr);         /* open a new color port, */
                               /*  this calls InitCPort */
theDepth = (**(*myCGrafPtr).portPixMap).pixelSize;

/* Bitshift and adjust for local coordinates */
offRowBytes = (((theDepth * (OffRight - OffLeft)) + 15) >> 4) << 1;
sizeOfOff = (long) (OffBottom - OffTop) * offRowBytes;
OffsetRect(&bRect, - OffLeft, - OffTop);

myBits = NewPtr(sizeOfOff);

/* Remember to be a PixMap */
(**(*myCGrafPtr).portPixMap).baseAddr = myBits;
(**(*myCGrafPtr).portPixMap).rowBytes = offRowBytes + 0x8000;
(**(*myCGrafPtr).portPixMap).bounds = bRect;

ourCMHandle = (**(**theMaxDevice).gdPMap).pmTable;
err = HandToHand(&((Handle) ourCMHandle));
/* Real programs do error checking here */
for (i = 0; i <= (**ourCMHandle ).ctSize; ++i)
    (**ourCMHandle ).ctTable[i].value = i;
(**ourCMHandle ).ctFlags &= 0x7fff;
(**ourCMHandle ).ctSeed = GetCTSeed();
/* This code is necessary for converting GDevice cluts to Pixmap cluts */

(**(*myCGrafPtr).portPixMap).pmTable = ourCMHandle;
SetPort((GrafPtr) myCGrafPtr);

/*****************************************************/
/*                                                   */
/*    function for setting the wanted color          */
/*                                                   */
/*****************************************************/
RGBColor FillInColor(short r, short g, short b)
{   /*FillInColor*/

    RGBColor  theColor;

    theColor.red = r;
    theColor.green = g;
    theColor.blue = b;
    return (theColor);
}

/*********************************************************/
/*                                                       */
/*  Drawing routine that makes the background blue       */
/* then draws a red oval, white oval, and green oval     */
/* After drawing to the offscreen it CopyBits to the     */
/*                 screen                                */
/*                                                       */
/*********************************************************/
void DrawIt()
{
```

```
        Rect        OvalRect;
        RGBColor    myRed,myBlue,myWhite,myGreen,myBlack;

        myRed = FillInColor(-1,0,0);
        myBlue = FillInColor(0,0,-1);
        myGreen = FillInColor(0,-1,0);
        myWhite = FillInColor(-1,-1,-1);
        myBlack = FillInColor(0,0,0);
        PenMode(patCopy);
        RGBBackColor(&myBlue);
        EraseRect(&thePort->portRect);
        RGBBackColor(&myWhite);
        RGBForeColor(&myRed);
        SetRect(&OvalRect,30,30,190,150);
        PaintOval(&OvalRect);

        InsetRect(&OvalRect,1,20);
        EraseOval(&OvalRect);

        RGBForeColor(&myGreen);
        InsetRect(&OvalRect,40,1);
        PaintOval(&OvalRect);
        RGBForeColor(&myBlack);

        SetPort((WindowPtr) myWindow);
        SetGDevice(oldDevice);

        destRect = bRect;
        OffsetRect(&destRect,OffLeft,OffTop);
        CopyBits((BitMapPtr) *(*myCGrafPtr).portPixMap,
            &(*myWindow).portBits,&bRect, &destRect, 0, nil);

        return;
    }


/*  Once again, you clean up as a final act. */

    CloseCPort(myCGrafPtr);
    DisposPtr(myBits);
    DisposHandle((Handle) ourCMHandle);
```