

This chapter describes view-related objects—view ports, view devices, and view groups—and the functions you can use to manipulate them. Read this chapter to learn how to draw any of the QuickDraw GX shapes you create, and how to control the representations of those shapes on any output devices.

Before reading this chapter, you should be familiar with the information in the chapter “Introduction to QuickDraw GX” in this book. You should also be familiar with shape objects, as discussed in the chapter “Shape Objects” in this book, and transform objects, as discussed in the chapter “Transform Objects.” Additional information related to color drawing is found in the chapter “Color and Color-Related Objects.”

This chapter constitutes the complete discussion of view-related objects for QuickDraw GX. Unlike for shape objects and style objects, there is no additional discussion of view ports, view devices, or view groups for graphic or typographic shapes in other books. The book *Inside Macintosh: QuickDraw GX Environment and Utilities* does, however, discuss drawing to Macintosh windows and how to relate a view port to a window. It also discusses matrix manipulation, which you can use to change the mapping property of a view port or view device.

This chapter introduces view ports, view devices, and view groups, discusses their properties, and shows how they are related to each other. It then discusses the different coordinate spaces you use to manipulate these objects. It then shows how to use QuickDraw GX functions to

- use view ports: create and manipulate them; manipulate their properties, including clip, mapping, dither, halftone, and parent and child view ports; and analyze a shape in a view port
- use view devices: create and manipulate them; manipulate their properties, including clip and mapping; and analyze a shape on a view device
- use view groups: create and manipulate them; set them up for offscreen drawing; and analyze a shape in a view group

About View Ports, View Devices, and View Groups

The view-related objects in QuickDraw GX exist to support drawing. They work together to provide device-independent drawing destinations for an application, while at the same time allowing access to device characteristics and permitting drawing destinations and devices to be grouped and manipulated in flexible ways.

These are the view-related objects:

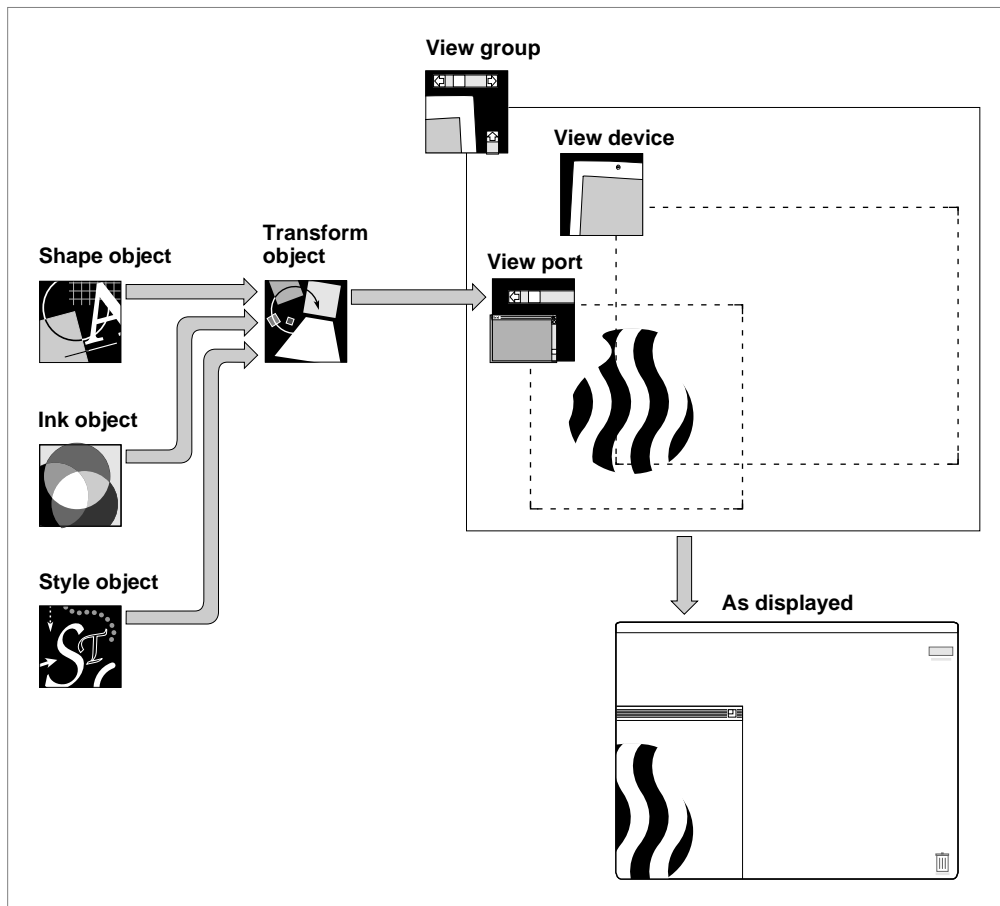
- **View port.** A view port represents a drawing destination, such as the content area within a window. View port objects are device-independent, and have mapping and clipping properties that position, modify, and mask the shapes drawn into them. A shape’s transform object defines the view ports that the shape is drawn into.

View-Related Objects

- **View device.** A view device represents a physical device, such as a monitor or printer, on which objects are drawn. Each view device object has mapping and clipping properties that define its resolution, mask its visible area, and position it in relation to view ports and other view devices.
- **View group.** A view group relates view ports and view devices to each other. Each view group object identifies a particular set of view ports and view devices; it also defines a coordinate space that positions view ports and view devices relative to each other. Drawing occurs in those locations where view ports and view devices within the same view group overlap.

Figure 7-1 shows the relationships among the view-related objects and the sequence of events that occur when a shape is drawn.

Figure 7-1 Objects used by the drawing mechanism



View-Related Objects

As Figure 7-1 shows, a shape's geometry, initially modified by information contained in the shape's style object and ink object, is further modified by the clip and the mapping of the transform object. That modified shape is then even further modified by the mapping and clip of one or more view ports, and modified once more by the mapping and clip of any view devices that intersect the view ports.

A shape cannot be drawn unless its transform object contains a reference to at least one view port. The transform object for the shape in Figure 7-1 references only one view port, so the shape is drawn only once. Transform objects are described in the chapter "Transform Objects" in this book.

Drawing occurs where the clip of a view port overlaps the clip of a view device within the same view group. The overlap is determined by the dimensions of the two clip shapes, both of which are defined in terms of the view group's coordinate space. In this example, the position of the shape in the view port and the overlap between the view port and the view device mean that only part of the shape is rendered.

The view group in Figure 7-1 represents a coordinate space for all view ports and view devices that may be visible to the user of your application. This particular view group is called the *onscreen view group* because the view devices represent actual screen devices. You can create view groups to draw offscreen as well.

View-related objects are different from most other QuickDraw GX objects in several ways:

- All the view-related objects can be shared, but they have no owner count and cannot be cloned. When you dispose of a view-related object, it is deleted from memory.
- View ports can be arranged hierarchically, and you can attach a view port hierarchy to a Macintosh window to simplify clipping, moving, and scrolling documents in a window.
- QuickDraw GX creates view device objects for all physical screen devices for you; for drawing to the screen, you normally never need to create a new view device.

About View Port Objects

A *view port* object represents the drawing destination for QuickDraw GX objects. A view port is analogous in some ways to a porthole on a ship, hence the name *port*. Objects seen through the porthole may have any extent, but only those parts within the boundaries of the porthole are visible.

View ports are device independent, and you normally need not take device characteristics such as pixel resolution into account when you draw.

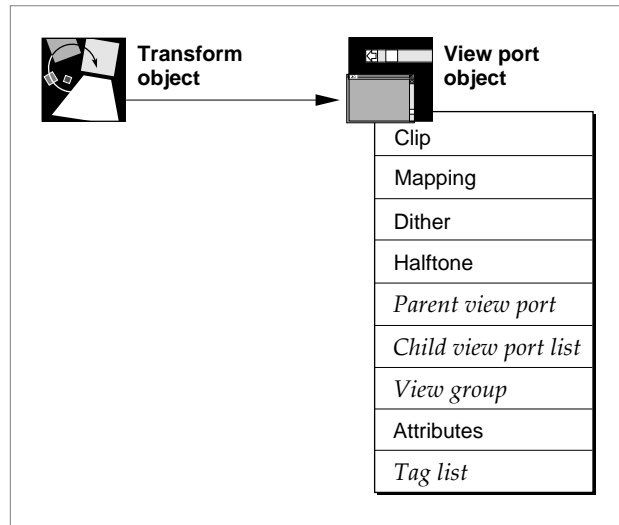
View Port Properties

The interface to view port objects is entirely procedural. You manipulate the information in a view port by modifying its properties using QuickDraw GX functions.

View-Related Objects

View port objects have nine accessible properties, as shown in Figure 7-2. Note that, because a view port is an object and not a data structure, the order of the properties as shown in Figure 7-2 is completely arbitrary. Properties in italics are references to other objects.

Figure 7-2 View port object properties



These are the accessible properties of a view port:

- **Clip.** A specialized shape geometry that controls the visibility of all shapes drawn to this view port. Only the parts of shapes that overlap with the clip remain visible when they are drawn. In the porthole analogy for a view port, the view port clip represents the transparent area of the porthole. The view port clip is further described in the next section, "*View Port Clip and Mapping.*"
- **Mapping.** A mathematical matrix that specifies the translation, scaling, skewing, rotation, and perspective of all shapes drawn to this view port. The view port mapping is further described in the next section, "*View Port Clip and Mapping.*"
- **Dither.** A value that specifies the dither setting of a view port. Dithering combines pixels of different colors to create the illusion of more colors than are actually supported by the hardware of an output device. For more information about the dither property, see the section "*Dither*" *beginning on page 7-10* .
- **Halftone.** A structure that specifies the halftone settings of the view port. Halftones use variable-sized dots of color to create the illusion of more colors than are actually supported by the hardware of an output device. For more information about the halftone property, see the section "*Halftone*" *beginning on page 7-13* .
- **Parent view port.** A reference to the view port that is the parent of this one. View ports exist in a hierarchical relationship that simplifies attaching them to windows and moving groups of them as units. For more information about the parent view port and view port hierarchies, see the section "*Parent and Child View Ports*" *on page 7-18* .

View-Related Objects

- **Child view port list.** A list of references to the view ports for which this view port is the immediate parent. View ports exist in a hierarchical relationship that simplifies attaching them to windows and moving groups of them as units. For more information about child view ports and view port hierarchies, see the section “*Parent and Child View Ports*” on page 7-18 .
- **View group.** A reference to the view group object to which this view port belongs. View groups are described in the section “About View Group Objects” beginning on page 7-29.
- **Attributes.** A set of flags that affect various characteristics of the shapes drawn to this view port. See the section “*View Port Attributes*” on page 7-20 for more information.
- **Tag list.** A list of references to custom information about this view port object, stored in private structures called tag objects. The chapter “Tag Objects” in this book describes tag objects in general and how you can use them to add custom information to objects.

QuickDraw GX provides functions to manipulate each of these view port object properties.

View Port Clip and Mapping

Like transform objects, view port objects have a clip property and a mapping property. A view port’s mapping and clip are applied to a shape after the transform clip and mapping have already been applied.

The clip and mapping properties for a view port follow the same general conventions as for transform objects. The clip property specifies a shape geometry that you use as a mask to restrict the visibility of a shape object when it is displayed or printed. The clip is equivalent to a primitive shape, a shape whose geometry and fill properties by themselves define the shape. Specifically, a clip can be a framed or filled geometric shape, a glyph shape, a 1-bit-per-pixel bitmap shape, or an empty or full shape. Primitive shapes are described in more detail in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The filled or framed parts of a view port clip define the areas in which a shape drawn to that view port show through. If the clip shape is a filled rectangle, for example, only the parts of a shape that are within the limits of that rectangle are visible. Commonly, view port clips are filled rectangles, because the visible parts of view ports commonly correspond to rectangular windows or panes of windows.

The mapping property of a view port is a 3×3 matrix that specifies one or more transformations that a view port applies to all shapes drawn into it. You can use the view port mapping to perform operations such as the following:

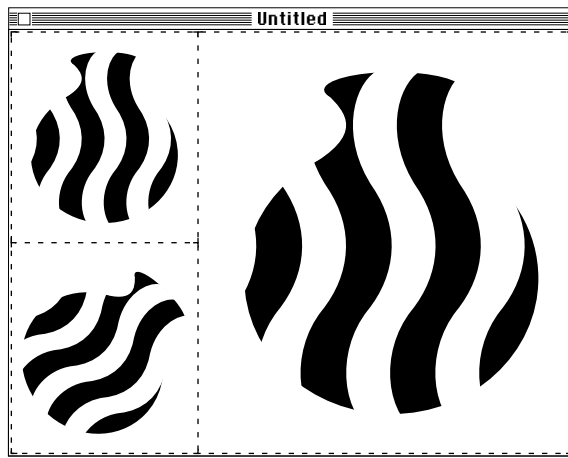
- translation, which changes the positions of shapes in the view port
- scaling, which shrinks or enlarges shapes horizontally or vertically or both
- rotation, which turns shapes about a fixed point
- skewing, which distorts shapes progressively along a single axis
- perspective, which distorts shapes to provide a three-dimensional effect

View-Related Objects

Most commonly, you use the view port mapping to position the view port (equivalent to positioning a window), and possibly to scale or rotate its contents. Skewing and perspective are less common, but you can use them for special effects. You can specify the identity mapping, a matrix whose elements have the value 1.0 along the diagonal and 0.0 elsewhere, to leave shapes drawn to this view port unchanged from the application of the transform mapping.

Figure 7-3 shows three different view ports in a single window, all used to display the same shape as that shown in Figure 7-1 on page 7-6. The shape is a vase, and its transform clip causes it to appear as wavy stripes.

Figure 7-3 Clipping and mapping in view ports



In Figure 7-3, each view port's clip shape is a rectangle that defines its pane in the window. The upper left view port uses an identity mapping. The lower left view port's mapping specifies a clockwise rotation. The right view port's mapping specifies scaling (equal in x and y dimensions) that enlarges the shape.

Dither

Dithering is a technique of assigning alternating colors to adjacent pairs or groups of pixels in a device's bitmap to achieve the illusion of a color that cannot be represented directly. For example, if a device only supports three shades of blue, dithering allows QuickDraw GX to assign those colors in a specific order to adjacent pixels, so that the mix of shades in the combined pixels approximates a desired but unsupported shade.








Dithering works one way in shapes that have a uniform color, and another way in bitmaps.

Dithering of Shapes Other Than Bitmaps

The dither property of a view port specifies a *dither level*, which is the maximum number of colors that QuickDraw GX can use in dithering when it draws a shape. The dither level can be between 1 and 16; a dither level of 1, the simplest, is equivalent to no dithering. A level of 0 is not permitted. Dithering has no effect if the device resolution is 32 bits per pixel.

Table 7-1 shows the pixel patterns that can occur, depending on the dither level. A dither level of 1 provides a solid pattern, which is effectively no dithering. A dither level of 2 provides a 2-by-2 repeating checkerboard pattern. A dither level of 3 provides a 3-by-3 repeating stripe pattern. A dither level of 4 provides a 4-by-2 repeating pattern. Note that the effective resolution of an image decreases as dither level increases, because each set of pixels that make up a unit of the dither pattern function as a single, larger pixel of dithered color.

Table 7-1 Dither levels and patterns

Dither level		Available patterns	
1			
2	Above patterns, plus:		
3	Above patterns, plus:		
4	Above patterns, plus:		

Implementation Note

Version 1.0 of QuickDraw GX supports a maximum of 16 colors in a dither pattern, although 4 is the practical maximum dither level, especially for grayscale drawing. ♦

View-Related Objects

QuickDraw GX does not necessarily use exactly the dither pattern specified by the dither property, unless the ink object attached to the shape drawn to the view port has its `gxForceDitherInk` attribute set. For example, if you specify a dither level of 4, QuickDraw GX may use any pattern from level 1 through level 4, as necessary, to create the illusion of additional colors. If the ink object's `gxForceDitherInk` attribute is set, however, only the level-4 pattern is used. Conversely, if the ink object's `gxSuppressDitherInk` attribute is set, no dithering occurs. Note also that you can affect the pixel alignment of the dither pattern with the ink object's `gxPortAlignDitherInk` attribute. For more information about these ink attributes, see the chapter "Ink Objects" in this book.

QuickDraw GX uses information from the color profile for the view device object to determine the supported colors and it chooses the appropriate colors to use in the dither for you. Although the results of dithering are controlled by the view device, you specify the dither level in the view port object so that you can simulate its effect, on the computer that is running the application, for a device that may not actually be present.

Dithering of Bitmaps

When you draw a bitmap shape, dithering works differently. Unlike with single-color shapes, dithering of bitmaps uses no specific pattern and recognizes no different dither levels. Dithering is off if the dither level is 1, and it is on if the dither level is greater than one.

Dithering of bitmaps uses the process of *error diffusion*, in which the error (the difference between the computed color of a given pixel and the nearest color available on the view device) is passed to adjacent pixels. The dithering algorithm starts at the top left of the visible part of the bitmap, and progresses through the bitmap, traveling left to right across one row of pixels and then right to left across the next lower row, and so on until the entire bitmap has been traversed. For each pixel, the algorithm adds the accumulated error (passed from the pixel above it and the pixel to the left of it) to the computed color, picks the closest available device color for that pixel, and passes the new error (the new computed color minus the available color) to the pixel to the right and the pixel below; half of the error goes to each.

Not all dither-related ink attributes are applicable to dithering of bitmaps. Setting or clearing the `gxPortAlignDitherInk` attribute or the `gxForceDitherInk` attribute of the ink object attached to a bitmap shape has no effect. Setting the `gxSuppressDitherInk` attribute, however, does have the effect of turning off dithering, even for bitmaps.

Ink attributes are described in the chapter "Ink Objects" in this book. The bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics* shows an example of drawing a dithered bitmap.

Drawbacks of Dithering

Dithering can provide good results in many cases, but it does have drawbacks:

- Dithering slows down the drawing operation slightly for non-bitmap shapes, because of the overhead of determining the pattern in which colors are assigned. Dithering significantly slows down the drawing of bitmaps.
- You cannot reliably use an ink transfer mode to reverse the effect of drawing a color that is dithered. Thus, transfer modes such as highlight mode may not be completely reversible where dithering occurs.
- In bitmap dithering, because QuickDraw GX determines which color to apply to a pixel using the color of the pixel next to it, a cumulative dithering error can occur due to the way a preceding pixel is changed by the dithering algorithm.
- Because clipping can interrupt the error diffusion in a bitmap dither, dithering a clipped shape can produce a different result from dithering the same unclipped shape. It can also mean that redrawing portions of an image, as when scrolling, can cause seams, or visible lines, between the separately drawn portions.

Note that dithering and halftones (described next) are mutually exclusive; if you choose both simultaneously, only a halftone is used.

Halftone

A *halftone* is a pattern of alternating colors of variable intensities in a fixed cell size, used to represent a variety of colors. Halftoning, like dithering, provides a method of representing color by alternating the available colors on view devices that support only a limited number of colors. Unlike a dither pattern, however, a halftone's fixed cell size means that its resolution is constant and adjacent halftones representing different colors mesh well. Also, unlike with dithers, you specify the colors that make up a halftone. If you use halftones, you should be familiar with how QuickDraw GX represents color, as described in the chapter "Color and Color-Related Objects" in this book. Also, note that dithering and halftones are mutually exclusive; if you choose both simultaneously, only a halftone is used.

A halftone consists of a pattern of variable-sized dots of one color against a background of another color. The halftone simulates a desired color (such as a specific intensity of gray), with the proper proportion of dot color (such as black) and background color (such as white). The colors are not limited to single components, however; a halftone can simulate beige, for example, by mixing pink with yellow. QuickDraw GX can attempt to reproduce any desired color by mixing the dot color and background color specified in a halftone. You can specify any color as the background and any other color as the dot color, and QuickDraw GX will find the best mixing proportion, given the specifications that you provide. Commonly, however, if you are using halftones you work with a single color component (such as blue for RGB space or yellow for CMYK space), and you specify that component as the dot color and black (for RGB) or white (for CMYK) as the background color.

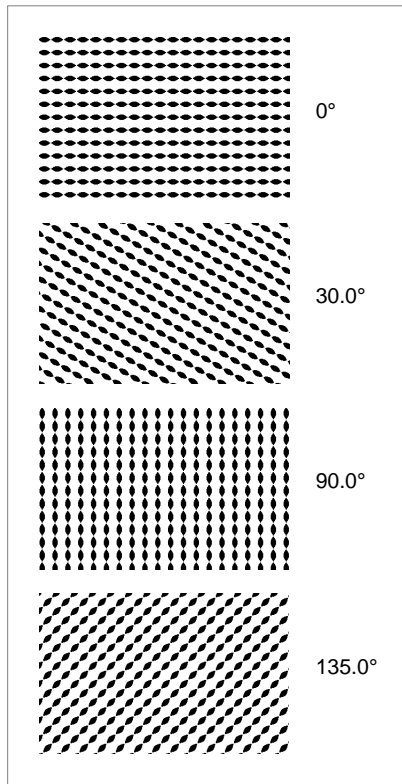
View-Related Objects

A halftone is described by several characteristics, specified in the `gxHalftone` structure:

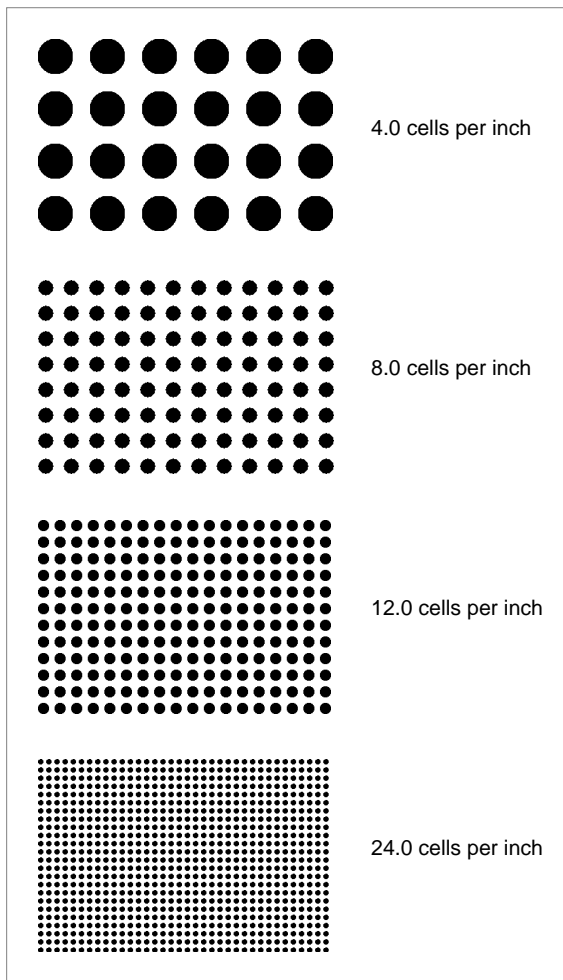
```
struct gxHalftone{
    Fixed      angle;          /* direction of halftone */
    Fixed      frequency;      /* cells per inch */
    gxDotType  method;         /* kind of pattern */
    gxTintType  tinting;        /* tint calculation method*/
    gxColor    dotColor;        /* color of dots */
    gxColor    backgroundColor; /* color of background */
    gxColorSpace tintSpace;     /* color space for tint */
};
```

The *angle* describes the orientation of the rows of dots in the halftone pattern. It is a fixed-point number between 0.0 and 360.0 that describes an angle, in degrees, clockwise from horizontal. Figure 7-4 shows several angles.

Figure 7-4 Halftone angle



Each cell in a halftone is an area that contains some proportion of background color and dot color. The *frequency* describes the size of the cells, in terms of numbers of dots per inch. You typically specify a frequency based on desired output quality and device resolution. Figure 7-5 shows examples of various frequencies.

Figure 7-5 Halftone frequency

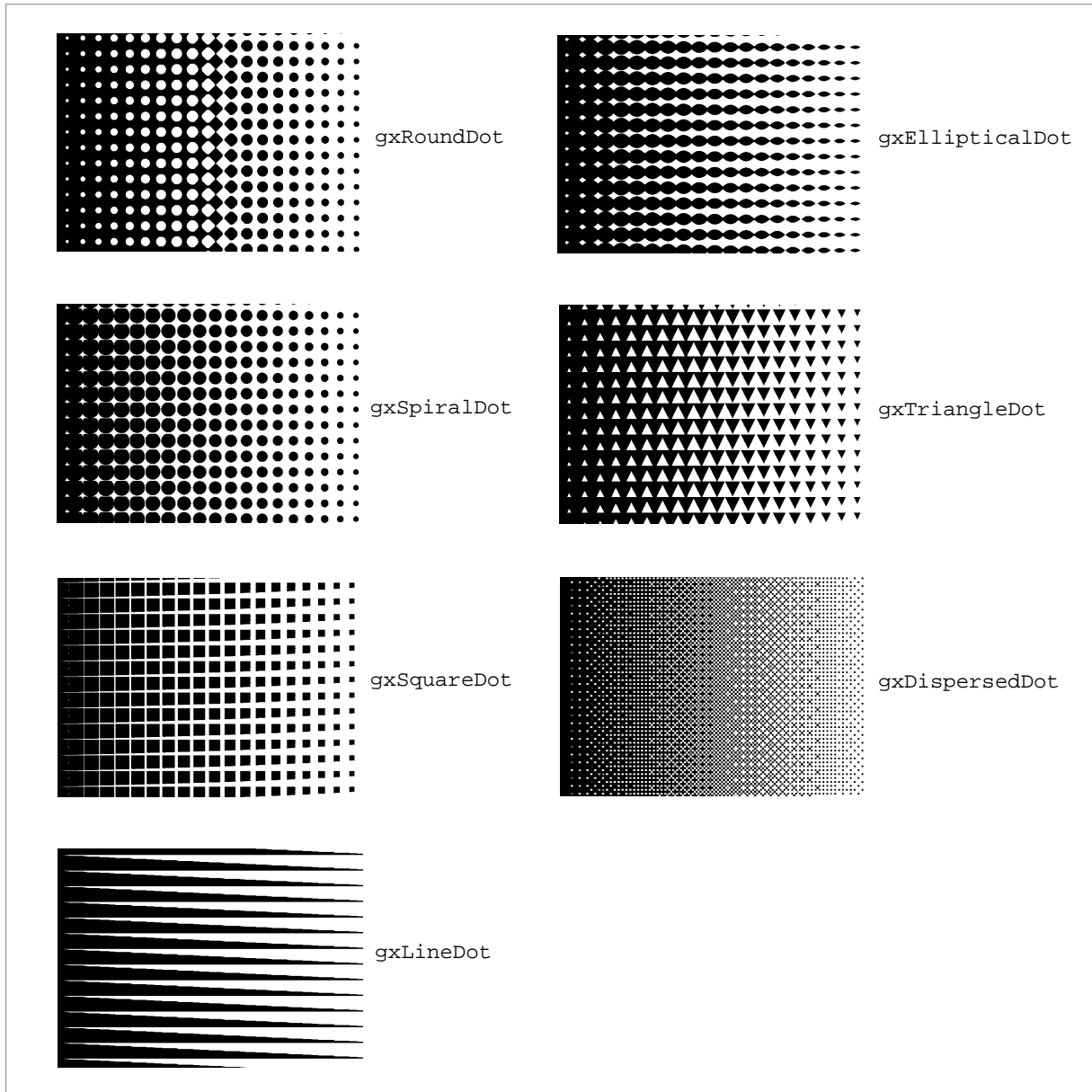
The *method*, or *dot type*, describes the halftone pattern itself and how it is filled: the shapes of the dots, the pattern of their arrangement, and the way in which a dot fills its cell as it enlarges. The supported methods are defined in the `gxDotTypes` enumeration:

```
enum gxDotTypes{
    gxRoundDot = 1,
    gxSpiralDot,
    gxSquareDot,
    gxLineDot,
    gxEllipticDot,
    gxTriangleDot,
    gxDispersedDot
};

typedef long gxDotType;
```

Figure 7-6 shows examples of these patterns.

Figure 7-6 Halftone dot types



The tinting, or *tint type*, specifies how the input color (the original color to which the halftone is applied) is to be approximated by a ratio of dot color and background color. The *tint* is a calculated value from zero to one: if the tint is zero, the halftone is composed only of the background color (the dots are infinitesimally small); if the tint is one, the halftone is composed entirely of the dot color (the dots fill their cells entirely).

View-Related Objects

The *tint color* is the actual color represented by the combination of dot and background, and is therefore a weighted average of the dot color and background color. The tint color may be only an approximation to—or even just a single component of—the input color. The `gxTintTypes` enumeration, described on page 7-67, defines these tinting choices:

- **Luminance.** The tint color is the input color’s luminance. The gray closest to the luminance of the input color is used to calculate the tint value. This tinting method is used for making grayscale halftones from grayscale or color images.
- **Color component.** The tint color is some intensity of any one of the components of the input color. This tinting method is used for making halftoned color separations.
- **Color average.** The tint color is the average of the components of the input color, determined by adding up the color components and dividing them by the number of components. This tint type is used with RGB only, and follows this formula:

$$\text{tint} = 1 - (R + G + B) / 3$$

Color average is different from luminance in that, for example, the luminance of an RGB color is not exactly the average of its component intensities. This tinting method is used for making grayscale halftones from color images.

- **Color mixture.** The tint color is a point on the line (in color space) connecting the dot color and the background color. The orthogonal projection of the input color onto that line locates the tint color, and the tint (the proportion of dot to background) is defined by the position of that projection point on the line. This is the formula:

$$\text{tint} = 1 - D1 / (D1 + D2)$$

where

$D1$ = input color-dot color distance

$D2$ = input color-background color distance

This tinting method is used for getting the closest possible representation of any input color using any dot and background colors.

The *dot color* and *background color* are, respectively, the color of the dots and the color of the background used to form the halftone. In the halftone structure, they are full color specifications, not just single color-component values. In setting up a halftone structure, you need to specify these colors in a way that is meaningful considering the tint type you have chosen. For example, if you are creating a halftoned color separation, you typically use a dot color that is the same color as the color component specified in the tint type, and a background color of white.

The *tint space* describes the color space the input color is converted to before the tint value is determined. For instance, you can set the tint space to CMYK space to separate out the cyan portion of an image that may have been created in RGB space. It is not necessary for the input colors or the view device colors to be set to CMYK space, only the halftone.

Note that halftoning occurs for a shape only if its view port contains a valid halftone structure, and if its ink object’s `gxSuppressHalftoneInk` attribute is cleared. The `gxSuppressHalftoneInk` attribute is described in the chapter “Ink Objects” in this book.

Parent and Child View Ports

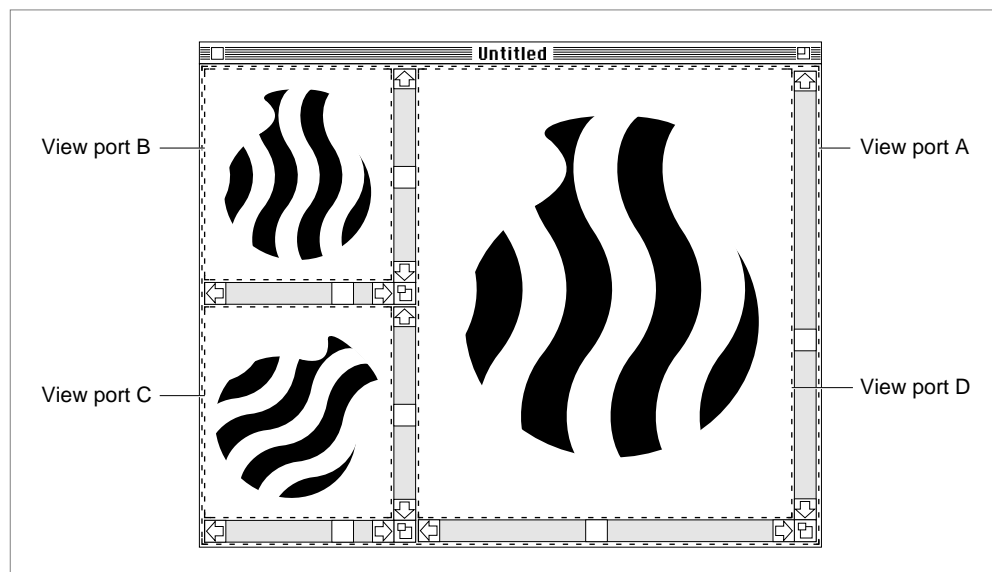
Two view port properties, the parent view port and the child view port list, allow you to arrange view ports in a hierarchy. The primary advantage of this capability is that QuickDraw GX manages the positional relationships among several related view ports for you.

In a view port object, the hierarchical relationship is represented by parent and child view port references. Each view port can reference one parent view port and any number of child view ports. When you move a view port by altering its clip and mapping, QuickDraw GX moves all its child view ports (and their child view ports, if any) accordingly. If the parent view port of your view port is attached to a window, QuickDraw GX moves your view port (and its children) to match movements of the parent whenever the user moves the window.

A view port hierarchy consists of a root view port, which is one with no parent view port, and all of its child view ports. If a child view port is also a parent view port, its children are part of the hierarchy too, and so on. Any parent view port in a hierarchy also defines a subhierarchy that consists of itself as the root, its child view ports, their child view ports, and so on.

Consider, for example, the window shown in Figure 7-7. Like in Figure 7-3 on page 7-10, it displays three different views of a vase. In this case, however, all the views are scrollable, which requires four view ports in a hierarchy.

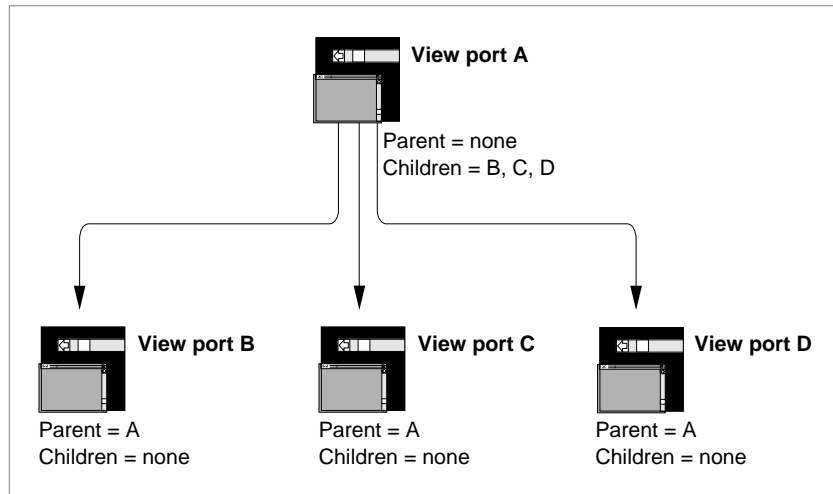
Figure 7-7 Hierarchical view ports in a window



View-Related Objects

The four view ports associated with the window in Figure 7-7 are arranged in the simple hierarchy shown in Figure 7-8.

Figure 7-8 A view port hierarchy



View port A encompasses the entire content area of the window. It does not have a parent, so it is the root of the hierarchy. View port A has three child view ports: B, C, and D. View ports B, C, and D all have the same parent, view port A. None of them, however, have child view ports of their own.

This hierarchical organization allows QuickDraw GX to automatically move all view ports when the window is moved. It also allows you to support scrolling in view port B, C, or D with minimal effort. When the user scrolls pane B, for example, the translation in view port B's mapping is changed to reflect the shape's new position in the window. No changes are required to the other child view ports or to view port A to implement scrolling. See the section "View Port Objects and Windows" beginning on page 7-21 for more information on view port hierarchies and windows.

When you set up a view port hierarchy, you create the root view port by calling the `GXNewWindowViewPort` function if you want the view port to be associated with a window, or the `GXNewViewPort` function otherwise. You create child view ports by calling the `GXNewViewPort` function for each, and then using the `GXSetViewPortParent` and `GXSetViewPortChildren` functions to organize them into a hierarchy.

The following rules apply when you set up a view port hierarchy:

- You cannot create a circular relationship among view ports. For example, a parent view port cannot also be a child view port within its own hierarchy.
- The view ports in a hierarchy must all be in the same view group.

View-Related Objects

View Port Attributes

Each view port object has a set of attributes, a group of flags that specify different aspects of display behavior. View port attributes allow you to specify drawing in gray only, to constrain shapes to integral pixel locations, or to enable color matching for shapes drawn to the port. Table 7-2 lists the constants for the view port attribute and describes what each one means. The constants are defined in the `gxPortAttributes` enumeration.

Table 7-2 View port attributes

Constant	Value	Explanation
<code>gxGrayPort</code>	0x0001	If set, QuickDraw GX only allows grays to be drawn to the view port; it converts colors into a gray color space. Color spaces are described in the chapter “Color and Color-Related Objects” in this book.
<code>gxAlwaysGridPort</code>	0x0002	If set, QuickDraw GX sets the <code>gxDeviceGridStyle</code> style attribute for all shapes drawn to the view port. This has the effect of constraining a shape to integral pixel values, thus avoiding distortion due to rounding of fractional coordinates. For more information about the <code>gxDeviceGridStyle</code> attribute, see the geometric styles chapter of <i>Inside Macintosh: QuickDraw GX Graphics</i> .
<code>gxEnableMatchPort</code>	0x0004	If set, QuickDraw GX performs color matching for all shapes drawn to this view port. Note that you must set this attribute for color matching to occur; color matching is off by default in view ports. Color matching is described in the chapter “Color and Color-Related Objects” in this book.

The Default View Port Object

When you first create a view port object, you must assign it to a specific view group. Other than that, the view port has these default properties:

- No parent view port.
- An empty child view port list.
- A clip shape that is a full shape. The clip has no effect.
- A mapping that is the identity mapping. The mapping has no effect.

View-Related Objects

- A dither level of 1.
- A `nil` halftone (no halftone).
- No attributes set.
- An empty tag list.

When you create a transform object, or if you just use the original default shape when you create a shape object, QuickDraw GX uses the default transform and assigns a default view port to the transform's view port list. That view port is in the onscreen view group and has the default properties just listed. That means that you can simply create a shape object and immediately draw it to the screen, without creating any view port. However, for most application purposes you need to restrict drawing to the interiors of windows. To do that, you can create a view port each time the user opens a window, and then alter the default shape object for each shape type to make sure that it references a transform whose view port list includes that view port or a child view port of it. Alternatively, you can explicitly assign the proper view port to the transform of each shape after the shape is created.

For more information about the onscreen view group, see "Onscreen and Offscreen View Groups" on page 7-29.

View Port Objects and Windows

In most cases on the Macintosh, when your application draws to the screen it draws into a Macintosh window. (You do not need a window for offscreen drawing or when printing.) Most of the view ports you create, therefore, are in some way associated with windows. QuickDraw GX allows you to associate a view port with a window, tying it to the window and establishing it as the root of a view port hierarchy.

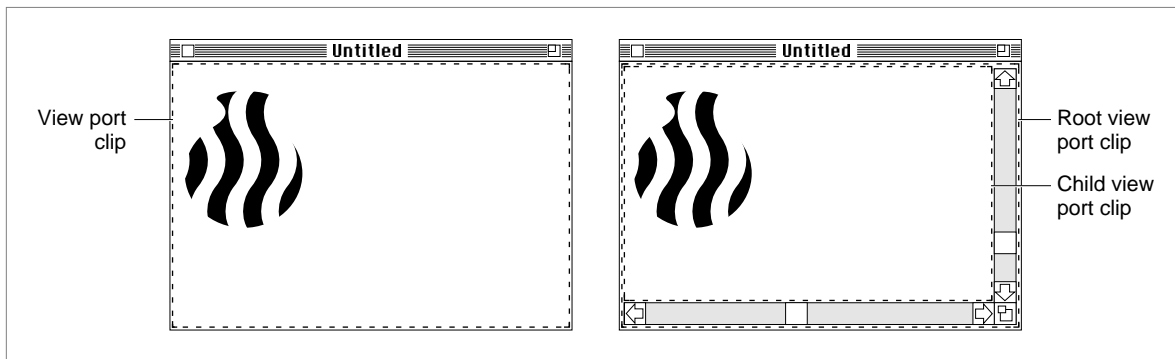
To attach a view port to a window, call the `GXNewWindowViewPort` function. This function sets up the view port so that drawing occurs only in the content area of the window (everything except for the title bar), effectively as if the view port's clip were equal to the window's visible region. When the user moves the window or changes its size, QuickDraw GX automatically moves the view port and adjusts its drawing limits to match the visible region. QuickDraw GX does not allow you to modify the clip or mapping of that view port.

If you add child view ports to the view port hierarchy, they are also moved as the window is moved. However, if the window is changed in shape, you need to adjust the clips of the child view ports to coincide with the new window dimensions.

View-Related Objects

Figure 7-9 shows a shape object drawn in two windows. In the window on the left, the shape is drawn directly to the window's view port; in the window on the right, the shape is drawn to a child view port of the window's view port.

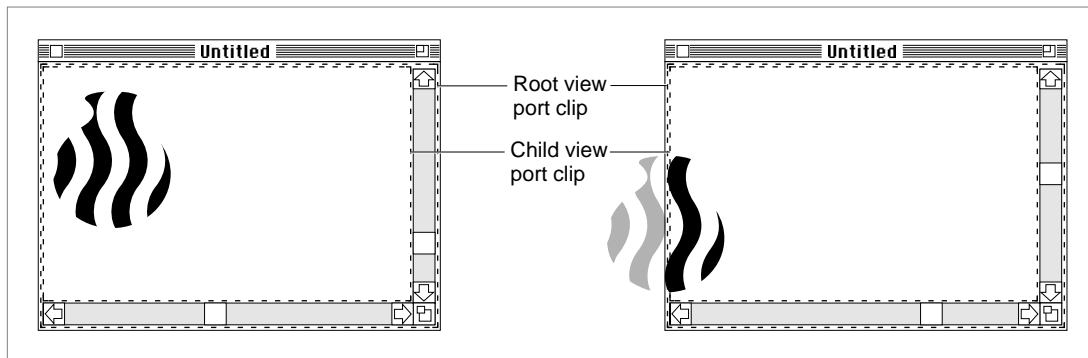
Figure 7-9 View ports in windows



One reason to draw only into a child view port is that it facilitates drawing tasks such as scrolling. Using a child view port helps to separate window management from content management when drawing. You use the parent view port for tracking window movement and visibility, and you manipulate the child view port's properties directly, without concern for the position or visibility of the parent view port. To implement scrolling, for example, you can follow these steps:

1. Create a child view port for your window's view port.
2. Draw your shapes to the child view port.
3. Alter the child view port's mapping to reflect the translation caused by scrolling.

Figure 7-10 shows these steps schematically. Note that the scroll bars are part of the content area of the window, and adjusting them means drawing into the parent view port. Note also that the child view port clip is smaller than the content area of the window, so that drawing into it does not draw over the scroll bars.

Figure 7-10 Adjusting a child view port's mapping to handle scrolling

You need not adjust the child view port's clip after scrolling because the clip's position is not changed when the mapping is altered; you need to adjust the clip only when the dimensions of the child view port's drawing area are changed (such as when the window is resized). Remember also that you need to adjust the mapping of a child view port only when there is relative movement between the child view port and its parent; if the user simply moves the window, you do not need to adjust the child view port because QuickDraw GX handles this for you.

For information about how clipping and mapping interact, see the section "About Drawing, Coordinate Conversion, and Clipping" beginning on page 7-30.

For information about the `GXNewWindowViewPort` function, see the environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. For an example that uses this function, see page 7-41.

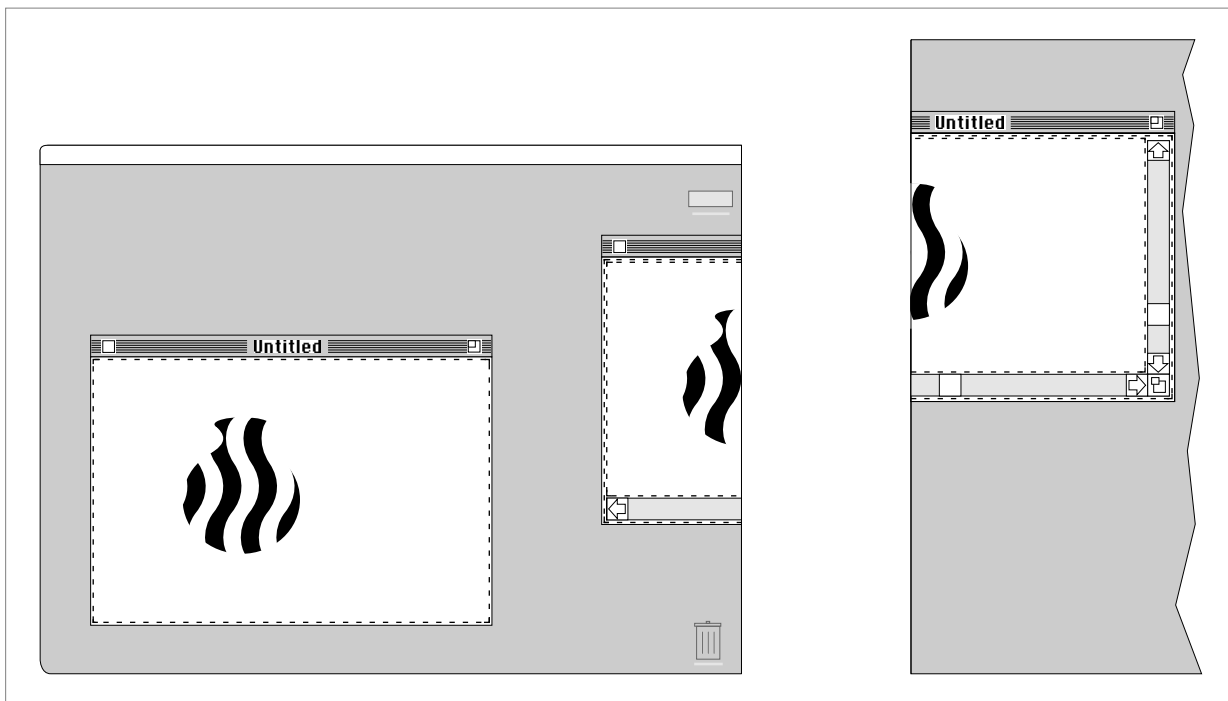
Drawing is Not Restricted to Windows

QuickDraw GX view ports are not restricted to windows; you can draw anywhere in a coordinate space. This feature makes it easy to take complete control of a view device and draw anywhere on it. For example, you can support dragging of objects between windows in this way. On the other hand, because most QuickDraw GX applications must share a view device with windows from other applications, you typically want to restrict drawing to window content areas. ♦

About View Device Objects

A *view device* object represents an output device such as a monitor or printer. When a shape is drawn, it appears on a view device, although its actual drawing destination is a view port. The intersection of the view port and view device determine where and how much of the shape is drawn. Figure 7-11 shows how a single view device can display more than one view port, and how a single view port can overlap more than one view device.

Figure 7-11 View ports overlapping view devices

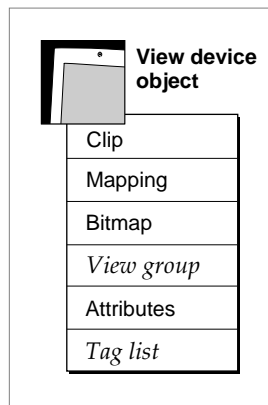


For drawing, you do not need to be concerned whether a view port overlaps one or more separate view devices; you just draw to the view port and QuickDraw GX handles it for you. On the other hand, if a view port does not intersect any device, the shapes drawn to the view port are not rendered at all.

View Device Properties

View device objects have six accessible properties, as shown in Figure 7-12. Note that, because a view device is an object and not a data structure, the order of the properties as shown in Figure 7-12 is completely arbitrary. Properties in italics are references to other objects.

Figure 7-12 View device object properties



These are the accessible properties:

- **Clip.** A specialized shape geometry that defines the active imaging area of the view device. Only the parts of the view device's bitmap that overlap with the clip can be drawn to. The view device clip is further described in the next section, "*View Device Clip and Mapping.*"
- **Mapping.** A mathematical matrix that specifies the translation, scaling, rotation, skewing, and perspective of shapes drawn on this view device. The view device mapping is further described in the next section, "*View Device Clip and Mapping.*"
- **Bitmap.** A bitmap structure that represents the imaging area of the view device. The view device bitmap is further described in the section "View Device Bitmap" on page 7-26.
- **View group.** A reference to the view group object to which this view device belongs. View groups are described in the section "About View Group Objects" beginning on page 7-29.
- **Attributes.** A set of flags that affect the state of activity and memory use of this view device. See the section "*View Device Attributes*" on page 7-27 for more information.
- **Tag list.** A list of references to custom information about this view device object, stored in private structures called tag objects. The chapter "Tag Objects" in this book describes tag objects in general and how you can use them to add custom information to objects.

View-Related Objects

Note that QuickDraw GX sets the properties for all onscreen view devices (view device objects that represent physical display devices present on the user's system). You cannot change the properties of those view devices.

View Device Clip and Mapping

Like transforms and view ports, view device objects have a clip property and a mapping property. A view device's mapping and clip are applied to a shape after those of the transform and the view port have already been applied.

The clip and mapping properties for a view device follow the same general conventions as for transform objects. The clip property specifies a mask that restricts the area on the device in which drawing or printing takes place. The clip is equivalent to a primitive shape, a shape whose geometry and fill properties by themselves define the shape. Specifically, a clip can be a framed or filled geometric shape, a glyph shape, a 1-bit-per-pixel bitmap shape, or an empty or full shape. Primitive shapes are described in more detail in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The filled or framed parts of the clip define the areas in which drawing can occur. In most cases the view device clip is simply a filled rectangle, often covering exactly the imageable area of the device. You can, however, restrict drawing to a single portion of the device by making the clip shape smaller.

The mapping property of a view device is a 3×3 matrix that specifies one or more transformations that the view device applies to shapes drawn into it. You can use the view device mapping like other mappings, to perform translation, scaling, rotation, skewing, or perspective. However, in most cases the view device mapping is used only to position the view device relative to other view devices and view ports, and to define its pixel size (an identity mapping usually means that pixel size is 72 per inch). You normally do not need to modify a view device's mapping, although it is possible for view device objects that you create yourself.

View Device Bitmap

The bitmap property of a view device is stored as a bitmap structure (type `gxBitmap`) that represents the imaging area of the device. The bitmap specifies the height, width, and pixel depth of the view device. The upper left corner of the pixel image is the upper left corner of the imaging area of the device; if the view device object has an identity mapping, that also corresponds to location (0.0, 0.0) in the view group to which the view device belongs.

The bitmap also specifies, possibly by using a reference to a color set object, the color of each pixel and the set of available colors on the device. (Bitmaps with fewer than 16 bits per pixel must use a color set.) The bitmap may also include a reference to a color profile object that defines the color response characteristics of the device.

View-Related Objects

The end result of a drawing operation is the assignment of pixel values to the bitmap of a view device, followed by the transfer of those pixels to the screen or onto paper. In screen drawing or in printing, you can use transfer modes that either ignore or take into account the current pixel values of the bitmap, which themselves may be the products of previous drawing actions.

When you retrieve the bitmap property of a view device object, QuickDraw GX returns it to you as a bitmap shape that includes the information from the bitmap structure in the view device.

Bitmap shapes are described in the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. Color set objects and color profile objects are described in the chapter “Color and Color-Related Objects” in this book.

View Device Attributes

Each view device object has a set of attributes, a group of flags that influence device behavior. View device attributes allow you to make a device active or inactive, and to specify whether or not the pixel image needs to be stored in directly accessible memory. Table 7-3 lists the constants for the view device attributes and describes what each one means. The constants are defined in the `gxDeviceAttributes` enumeration.

Table 7-3 View device attributes

Constant	Value	Explanation
<code>gxDirectDevice</code>	0x01	If set, QuickDraw GX puts the pixel image of the view device’s bitmap in directly accessible memory, if possible.
<code>gxRemoteDevice</code>	0x02	If set, QuickDraw GX puts the pixel image of the view device’s bitmap in remote memory, such as on an accelerator card or video controller card, if possible.
<code>gxInactiveDevice</code>	0x04	If set, QuickDraw GX makes the device inactive, meaning that no drawing occurs on it. As an object, however, the device retains its existence and all its properties. QuickDraw GX sets this attribute for view device objects whose <code>GDevice</code> records mark them as inactive Macintosh graphics devices. The relation of graphics devices to view devices is described in the Macintosh environment chapter of <i>Inside Macintosh: QuickDraw GX Environment and Utilities</i> ; the <code>GDevice</code> record itself is described in <i>Inside Macintosh: Imaging with QuickDraw</i> .

The Default View Device Object

When you first create a view device object, you must assign it to a specific view group, and you must give it a reference to a bitmap shape—which you have set up—that describes the size, pixel depth, and location of the device’s imaging area. Otherwise, the view device has these default properties:

- A clip shape that is a full shape. The clip has no effect.
- A mapping that is the identity mapping. This means that the upper left corner of the imaging area is at (0.0, 0.0) in global space, and the device resolution is 72 pixels per inch.
- No attributes set.
- An empty tag list.

For view devices that you do not create, QuickDraw GX defines their properties to be consistent with the physical devices they represent. See “View Device Objects and Physical Devices” (next).

View Device Objects and Physical Devices

View device objects are different from some other QuickDraw GX objects in that QuickDraw GX creates all that you need for ordinary drawing. Unless you are drawing offscreen or want unusual onscreen effects, you do not have to create a view device.

At startup, QuickDraw GX creates a view device object for each physical display device attached to the system, assigning the device to the onscreen view group (see “Onscreen and Offscreen View Groups” on page 7-29). QuickDraw GX sets the view device mapping and clip properties to reflect the device’s pixel size, dimensions, and position in relation to other view devices. QuickDraw GX initializes the view device bitmap, assigning it a pixel image of appropriate size and pixel depth, and a color set and color profile based on information provided by the device’s driver. (If the user changes the relative positions of display devices by some means such as the Monitors control panel, QuickDraw GX automatically updates the mappings and clips of the view devices to reflect the change.)

If you need information about any display device, you can first obtain a list of all the view device objects in the onscreen view group. You can then use the object references in that list to examine the properties of each device as needed.

View device objects are also associated with printers. To access the view device of a printer—if, for example, you want to mimic its characteristics with an offscreen view device—you use functions described in the advanced printing features chapter of *Inside Macintosh: QuickDraw GX Printing*.

For offscreen drawing, you do need to create your own view device objects and initialize all of their properties, including their bitmaps. Note that if you create a bitmap for offscreen drawing whose characteristics exactly match those of an onscreen view device, you can quickly transfer the results of your offscreen drawing to the screen by simply drawing that bitmap onscreen.

About View Group Objects

A *view group* object exists to relate view ports and view devices. It defines the set of view ports and view devices that can interact with each other, and it provides the basis for their relative positioning.

A view group represents a two-dimensional coordinate plane called *global space*. Global space imposes physical dimensions on and defines the spatial relationships among the view ports and view devices that belong to a view group.

You can have several view groups. Each defines a drawing world, allowing you to separate groups of view ports and view devices and draw to each group without conflict.

View Groups Have No Properties

As QuickDraw GX objects, view groups have no directly accessible properties. A view group is more like an ID number than an object containing information. Although you can at any time obtain a list of view ports and view devices that belong to a given view group, you can think of that information as coming from the individual view port and view device objects in the group, rather than from the view group object itself.

Likewise, the dimensional and positioning information imposed by a view group is all contained in the mapping matrices of the individual view ports and view devices that belong to the view group.

The only kinds of manipulation that you can perform directly on view group objects are creating them, disposing of them, and passing their references as function parameters.

Onscreen and Offscreen View Groups

QuickDraw GX creates one view group for you: the *onscreen view group*, whose reference is defined by the `gxScreenViewDevices` constant. It includes all physical screen devices. You draw to view ports in this view group for all onscreen drawing. If a transform object's view port list property is not changed from its default value, the only view port in the list is the default view port, which is in this view group. Thus, by default, shapes are drawn onscreen to view ports and onto view devices in this view group.

You can also create any number of offscreen view groups. For example, you can build an image by creating an offscreen view group that mirrors the onscreen view group and draw into view ports and onto view devices exactly as the drawing is done with the onscreen view group. The only difference is that your shapes appear in the bitmap of your offscreen view devices instead of onscreen. When you are ready to transfer those drawn shapes to the screen, you can draw the bitmaps of the offscreen view devices as bitmap shapes into view ports in the onscreen view group.

View-Related Objects

Clipping and Offscreen Drawing

For onscreen view ports attached to windows, QuickDraw GX takes care of restricting drawing to each window's visible areas, even in cases where windows overlap. When you create an offscreen view group with offscreen view ports, you need to take care of all clipping yourself, including cases in which view ports in different hierarchies overlap and those in front must clip those behind. ♦

To help you track all view ports and view devices for all onscreen and offscreen view groups, QuickDraw GX provides another predefined view group reference, defined by the `gxAllViewDevices` constant. You use it to specify all view groups when you want a list of all view ports or all view devices in all view groups. You cannot use this constant to set a view port or view device because `gxAllViewDevices` does not actually refer to a specific view group.

About Drawing, Coordinate Conversion, and Clipping

When you draw a QuickDraw GX shape, you are converting its internal representation into an image on an output device. QuickDraw GX uses information in the shape object and several other objects, including the view-related objects, to control how the shape is rendered. In brief, when you execute a drawing command QuickDraw GX follows this sequence of tasks:

1. It extracts the geometry of the shape object.
2. It applies stylistic and color information from the style object and ink object.
3. It applies the clip, and then the mapping, from the transform object.
4. It applies the mapping, and then the clip, from one or more view port objects.
5. It applies the mapping, and then the clip, from one or more view device objects.

The mapping operations specified in the transform, view port, and view device objects are concatenated during drawing, meaning that the operation is applied at one stage and the result is then used as input to the next calculation, and so on. Mapping is thus cumulative.

The clipping mechanism computes the intersection of the clip shapes of the transform object, view port objects, and view device objects. Each time a clip is applied, the visibility of a shape can only be further restricted.

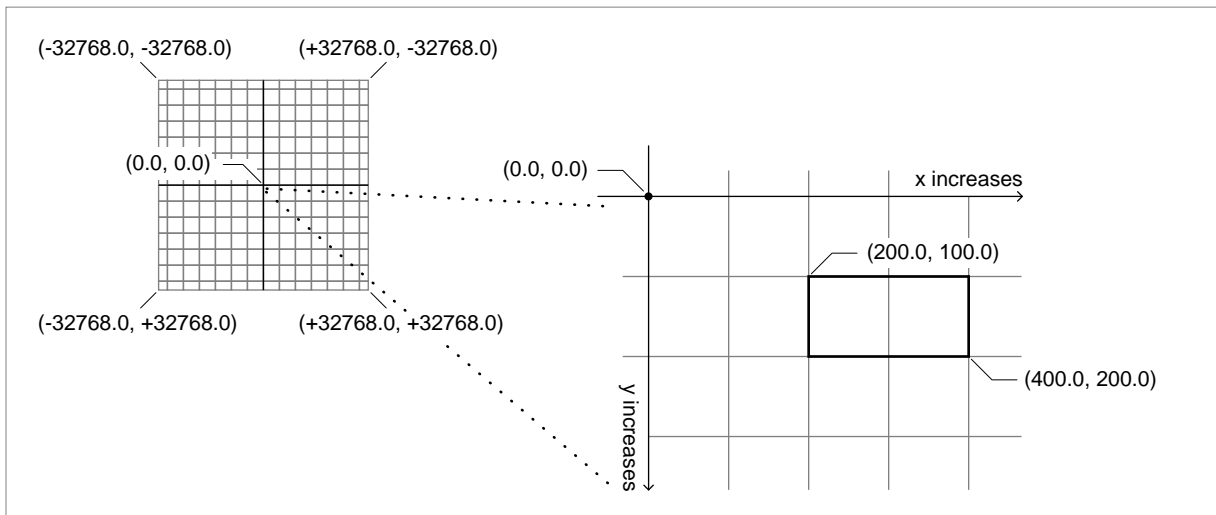
The following sections discuss this process in more detail, describing how QuickDraw GX uses four separate coordinate spaces to specify a shape during the stages of the drawing process, and what effects you can control at each stage.

QuickDraw GX Coordinates

All coordinates in QuickDraw GX are specified with fixed-point numbers in the range of $-32,768.0$ to approximately $32,768.0$. Fixed-point numbers and the functions for manipulating them are described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. For any coordinate space, point $(0.0, 0.0)$ represents the origin of the space. Points that lie to the right of the origin increase in a positive direction along the x-axis; points that lie below the origin increase in a positive direction along the y-axis. Coordinates are always written in the order (x, y) .

Figure 7-13 shows the general layout of the QuickDraw GX coordinate plane, with an expanded portion that shows a rectangle 200 units wide by 100 units high, whose upper-left corner is at the point $(200.0, 100.0)$.

Figure 7-13 The QuickDraw GX coordinate plane



QuickDraw GX allows you to work in four coordinate spaces: geometry space, local space, global space, and device space. You can work separately in each space as appropriate for specific purposes; QuickDraw GX automatically converts among them when drawing.

The following discussion of coordinate spaces follows the progress of a drawing operation. It uses as an example the rendering of a single shape in a single window on a single view device. The shape, as finally displayed, is shown in Figure 7-19 on page 7-39. More complex possibilities, such as displaying in multiple windows and on multiple devices, are discussed as they arise.

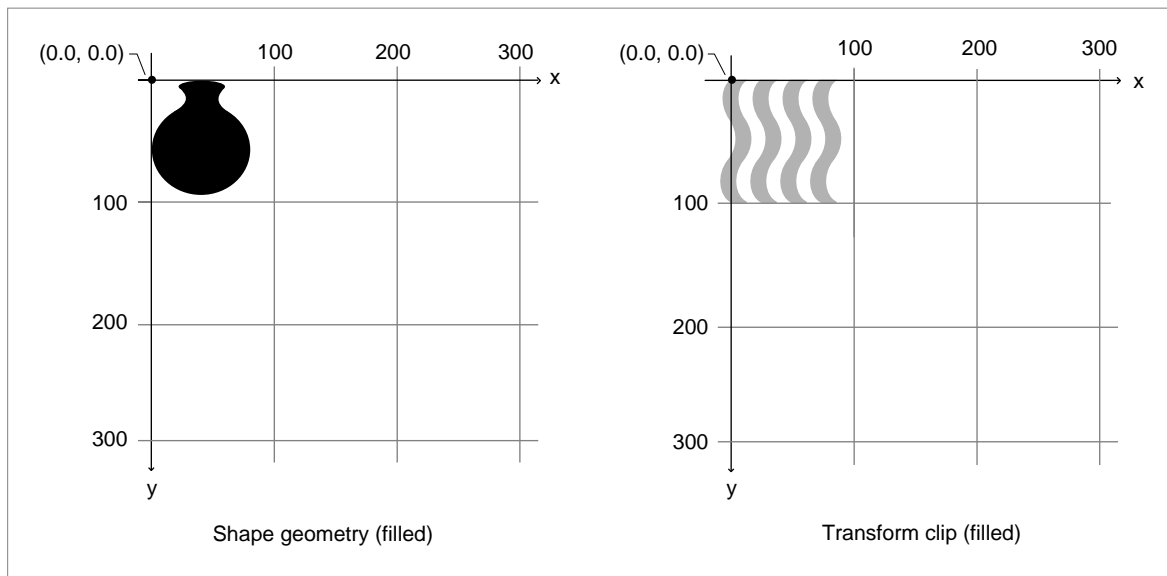
Geometry Space

Geometry space is the space within which the position and dimensions of a shape object are first defined. QuickDraw GX starts the drawing process by using the values in a shape's geometry; those values define the shape's fundamental dimensions, and their coordinate space is called geometry space.

No distance metric, such as points per inch, is defined for geometry space. Thus, the absolute size of a shape is undefined in geometry space. Also, every shape has its own geometry space; you cannot compare the sizes of two shapes based on their dimensions in geometry space alone. A rectangle 10 units wide in geometry space could end up ten times wider than a rectangle 100 units wide, once both have been drawn.

The left side of Figure 7-14 shows the geometry of a shape object, a filled path shape in the form of a vase. In geometry space, the vase is approximately 100 units by 100 units, and the upper-left corner of its bounding rectangle is at about (0.0, 0.0).

Figure 7-14 A shape geometry and a transform clip geometry



The right side of Figure 7-14 shows the geometry of the clip shape for a transform object. The clip shape is a filled path shape, of approximately the same dimensions and location as the vase shape. The next section shows how the clip shape modifies the appearance of the vase shape (assuming the vase shape object references the transform object containing this clip).

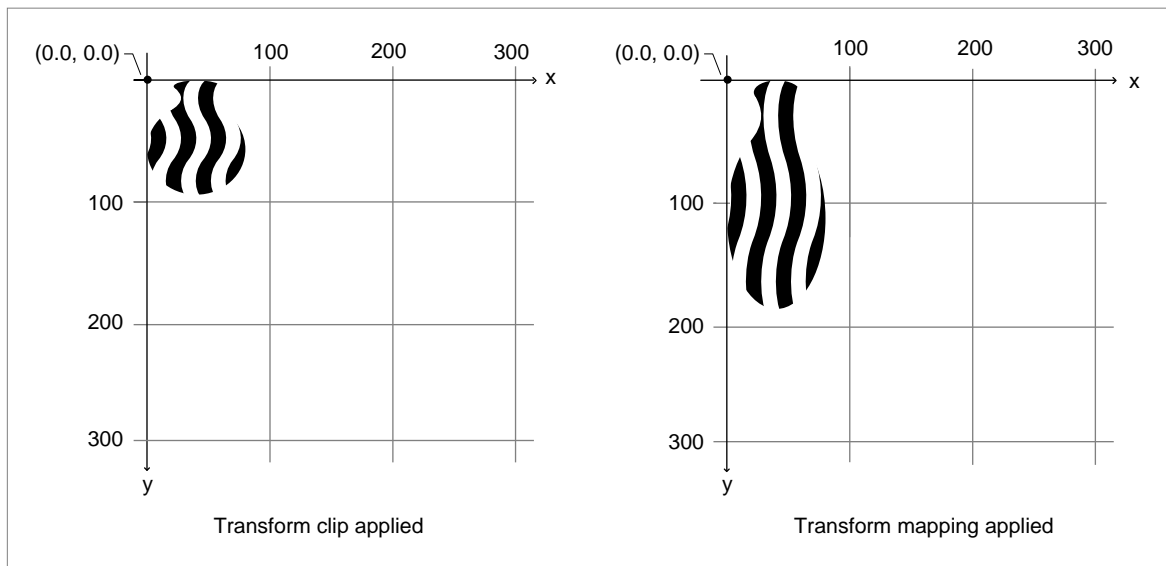
Local Space

Local space defines the location and dimensions of a shape after it has been modified by the mapping property of its associated transform object. Because mappings can translate, scale, rotate, skew, and otherwise distort geometries, the dimensions of a shape in local space can be quite different from what they are in geometry space.

As the first stage of drawing a shape, QuickDraw GX modifies the shape's geometry by applying information from the style object attached to the shape, and then applying first the clip and then the mapping contained in the transform object attached to the shape. Applying the mapping converts the shape from geometry space to local space. Because the transform clip is applied *before* the transform mapping, the dimensions of the clip shape are considered to be in geometry space. When you define the clip of a transform object, you size it and position it in terms of the dimensions of the shape's geometry.

The left side of Figure 7-15 shows the same vase shape as in Figure 7-14, this time after the transform clip has been applied to it. At this point the shape is still in geometry space—its overall position and dimensions unchanged, but its appearance modified by the clip.

Figure 7-15 Applying the transform's clip and mapping to a shape



The right side of Figure 7-15 shows the vase shape after the transform mapping has been applied to it. In this particular example, the only effect of the transform mapping is to scale the shape by a factor of 2.0 in the vertical direction, about an origin at (0.0, 0.0) in geometry space. The vase is now in local space.

View-Related Objects

Local space, like geometry space, has no metric; the absolute size of a shape object is still undefined after the transform mapping has been applied. You can, however, compare the sizes of two shape objects that share the same transform object. For example, if two path shapes have the same geometry and reference the same transform object, they are the same size.

You typically use the transform's clip and mapping for application-specific purposes related to moving, masking, and distorting shapes within a document. With the transform clip you define what parts of the shape geometry are to be visible, and with the transform mapping you choose how to move, orient, and distort that visible part of the shape, usually in relation to other shapes in the same document.

Several shape objects can reference the same transform object. This allows you to move, scale, rotate, and otherwise change an entire group of shapes in unison, by altering a single transform mapping.

Some shape types have specific additional definitions of local space:

- Picture shapes, which consist of a hierarchy of other shapes, can have more than one transform object. In such a case, QuickDraw GX performs clipping and mapping operations on all transforms in turn from the bottom of the hierarchy to the top; the result of all those mapping transformations is considered local space for the shape. See the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics* for information about the transform hierarchy in picture shapes.
- Glyph shapes can have mappings in their geometries (tangent array) and in their associated style objects (text faces). In such cases, QuickDraw GX applies those mappings before applying the clip and mapping of the transform object to convert the glyph shape to local space. See the glyph shapes and typographic styles chapters of *Inside Macintosh: QuickDraw GX Typography* for information about the tangent array and text face mappings.

The transform object includes a reference to at least one view port object, and local space orients a shape within its view port. Local space is the coordinate system local to that view port—hence its name. Thus, the vase example in this section would have the same local coordinates—its bounding rectangle would have corners at about (0.0, 0.0) and (100.0, 200.0)—no matter how the view port itself might be scaled or distorted by its own mapping when it is converted to global space.

The fact that local space is the interior coordinate space of a view port means that you can compare the sizes of two shapes in local space even if they do not share the same transform—as long as they share the same view port. If two shapes have the same dimensions in local space and their transforms reference the same view port, they are the same size regardless of the actual values in their geometries or transform mappings.

Global Space

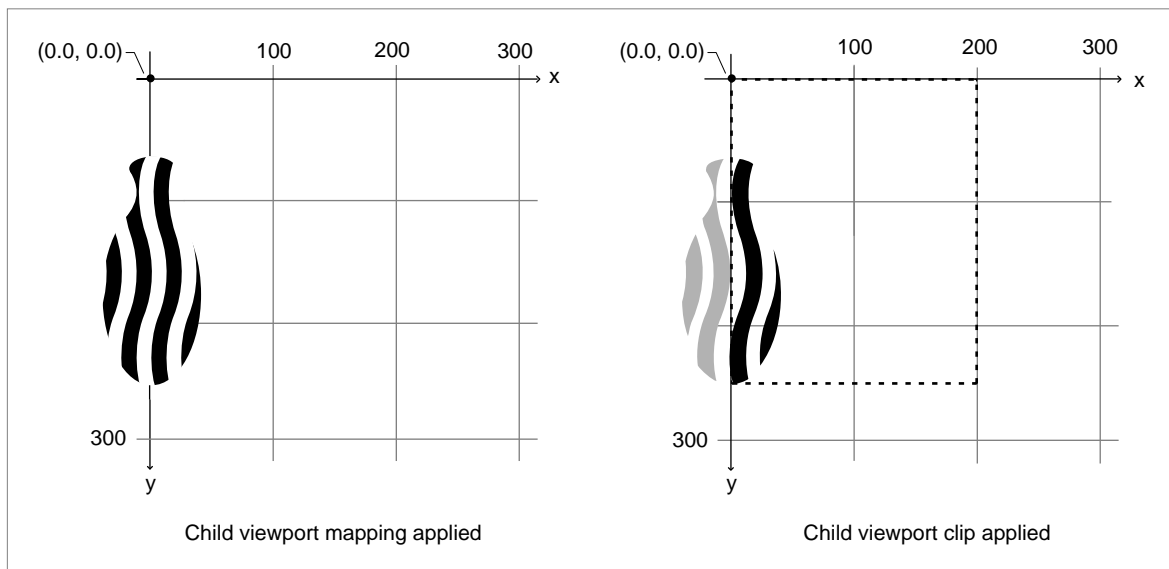
Global space contains the location and dimensions of a shape after the mapping in its associated view port has been applied. Global space defines the real-world location and dimensions of a shape; coordinate values in global space represent distance in points (72 per inch) from the origin of the view group that the view port is part of.

As this stage in drawing a shape, QuickDraw GX converts the shape from local space to global space. It modifies the shape's dimensions by applying first the mapping and then the clip contained in the view port object attached to the shape's transform. Because the view port clip is applied *after* the view port mapping, the dimensions of the clip shape are considered to be in global space. When you define the clip of a view port object, you size it and position it in terms of global space (the view port's position compared to view devices), not local space (the shape's position in its view port).

If the view port to which drawing occurs is a child view port in a view port hierarchy, QuickDraw GX performs mapping and clipping operations on all view ports in turn from that child view port through the top (root) view port; the result of all those mapping transformations is considered global space for the shape. See the section "Parent and Child View Ports" beginning on page 7-18 for information about view port hierarchies.

The example vase shape shown in the previous figures is drawn into the child view port of a simple two-level hierarchy. The left side of Figure 7-16 shows the vase shape after the child view port mapping has been applied to it. In this particular example, the effect of the view port mapping is to move the shape downward and to the left by approximately 50 units, representing a scrolling of the shape from its original position. There is no scale factor or other distortion in this case, so the dimensions of the shape are unchanged. The shape is not yet in global space, however, because another mapping (from the parent view port) must be applied.

Figure 7-16 Applying the child view port's mapping and clip to a shape



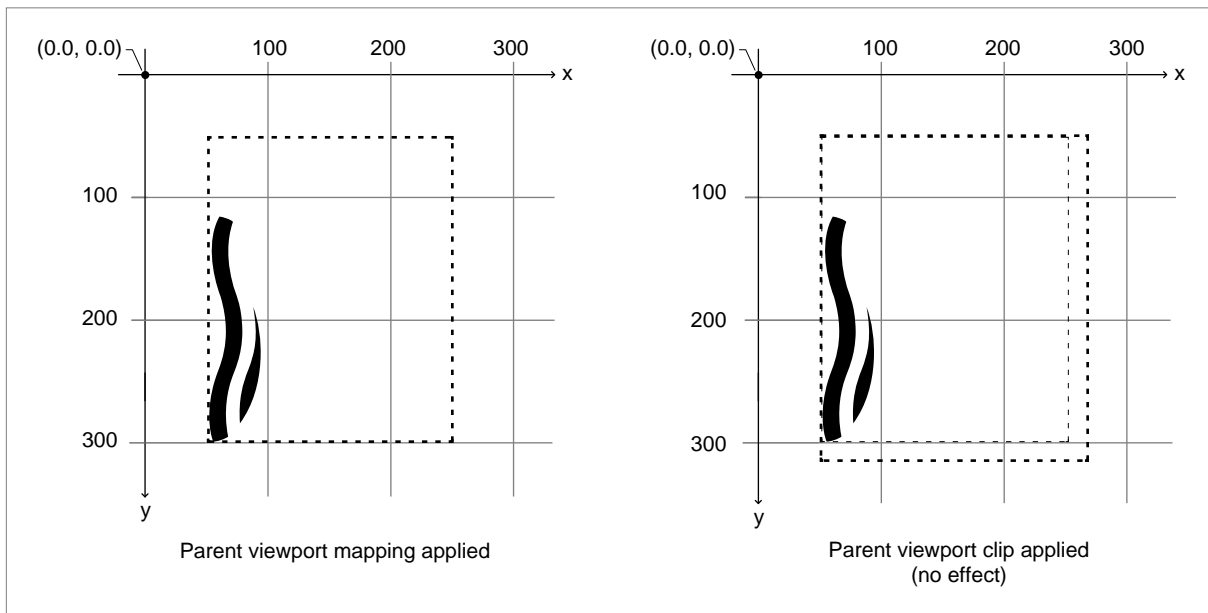
View-Related Objects

The right side of Figure 7-16 shows the vase shape after the child view port clip has been applied to it. The view port clip in this case is a rectangle that defines the visible portion of the child view port. As Figure 7-16 shows, the clip cuts out the left half of the vase (shaded gray), meaning that part of the shape has been scrolled out of view in its view port. The clip's dimensions, although not yet in global space (because the parent view port mapping has not yet been applied), are "global" to the child view port; changing the child view port's mapping, for example, does not change the position of its clip in relation to its parent view port. Therefore, to scroll a shape in a view port, you need only change the view port's mapping, not its clip.

This example shows a single shape drawn into a single view port, but more complex arrangements are possible. For example, as shown in Figure 7-3 on page 7-10, a single transform object can reference several view port objects, allowing a single shape to appear simultaneously (perhaps with different scaling or orientation) in several view ports.

Figure 7-17 completes the process of conversion from local to global space. The left side of Figure 7-17 shows the vase shape after the parent view port mapping has been applied to it. In this example, the effect of the view port mapping is to move the shape to the right and downward by approximately 50 units, representing the actual location of the shape in global space. Again, there is no scale factor or other distortion applied in this case, so the dimensions of the shape are unchanged. Because this view port is the root view port, the shape is now in global space and its dimensions can be measured. The visible part of the shape is approximately 50 points by 200 points in size, or about 0.7 by 2.8 inches.

Figure 7-17 Applying the parent view port's mapping and clip to a shape



View-Related Objects

The right side of Figure 7-17 shows the vase shape after the parent view port clip has been applied to it. This view port clip is a rectangle that defines, in global space, the content area of the window to which the parent view port is attached. As is typical for a simple window that supports scrolling, the clips of the child view port and parent view port differ only by the areas of the scroll bars; the child view port clip fits inside the scroll bars so that drawing into it does not obliterate the scroll bars. In this case, the application of the parent view port clip has no effect on the visibility of the vase shape because the child view port clip is entirely contained within it.

As this example shows, you typically use the parent view port's mapping to position the window you are drawing into, and its clip to restrict drawing to the interior of the window. You use mappings of child view ports to scroll, scale, or move shapes in relation to the parent view port, and you use their clips to restrict the shapes' visibilities in relation to the parent view port. If a parent view port is attached to a window (through the `GXNewWindowViewPort` call), QuickDraw GX itself manipulates both the clip and mapping of the parent view port to make sure its location and drawable area correspond to the visible parts of the content area of the window. (Strictly speaking, QuickDraw GX prevents drawing from occurring outside of the visible part of the content area of the window, but it does not necessarily use the view port's clip to do so; if you retrieve the clip of a window view port, it is not guaranteed to be equal to either the window's port rectangle or its visible region.)

Global space is view-group space. Keep in mind these ways in which view groups and global space define the interactions among view ports and view devices:

- Once a shape's dimensions have been converted to global space, it has an absolute size and a specific spatial relationship to all other shapes in that view group, whether or not the shapes share the same local space (view port).
- Global-space dimensions are device-independent and therefore resolution independent; for typical drawing operations, you need never know the resolutions of the devices you are drawing to.
- Within a view group, the clips of view ports and view devices can overlap in any combination. Drawing occurs automatically wherever the visible portions of any view port and any view device in that view group overlap.
- More than one view group can exist simultaneously, allowing for offscreen drawing. Furthermore, the view ports referenced by the transform of a single shape need not all be in the same view group, allowing for simultaneous onscreen and offscreen drawing of a shape.

To draw the device-independent shapes in a view group with maximum accuracy on view devices of varying positions and resolutions requires conversion from global space to device space, as described next.

Device Space

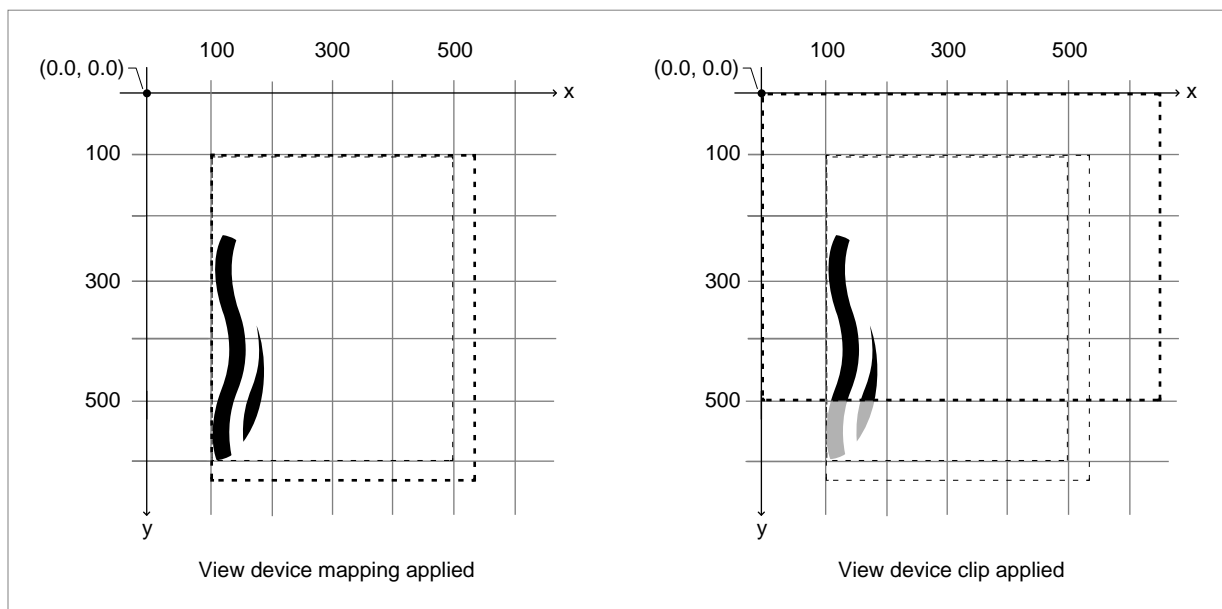
Device space defines the location and dimensions of a shape as displayed on a particular output device. The upper-left corner of the displayable area of a view device is at coordinate (0.0, 0.0) in device space. Unit distance between coordinates in device space represents one picture element, or pixel.

The view device's mapping defines both its location in global space (as a translation factor) and its pixel size (as a scaling factor). For example, if your device is a 144 pixels-per-inch high-resolution monitor, QuickDraw GX converts global space to device space when drawing by scaling each global-space point by 2.0, which is $144/72$. By default, if there is a single view device in a view group, the translation value in its mapping is 0, meaning that point (0.0, 0.0) in device space is also point (0.0, 0.0) in global space. The view device's clip is a (usually rectangular) shape representing the displayable area of the device.

As the final stage in drawing a shape, QuickDraw GX converts the shape from global space to device space. It modifies the shape's dimensions by applying first the mapping and then the clip of any view device object in the same view group whose clip overlaps the view port clip.

The example vase shape shown in the previous figures is drawn onto a single view device. The left side of Figure 7-18 shows the vase shape after the view device mapping has been applied to it. In this example, the view device mapping specifies no translation, but the pixel resolution is 144 ppi so it scales the shape by 2.0. The shape is now in device space, and its visible part is approximately 100 by 400 pixels in size (which is still about 0.7 by 2.8 inches).

Figure 7-18 Applying the view device's mapping and clip to a shape

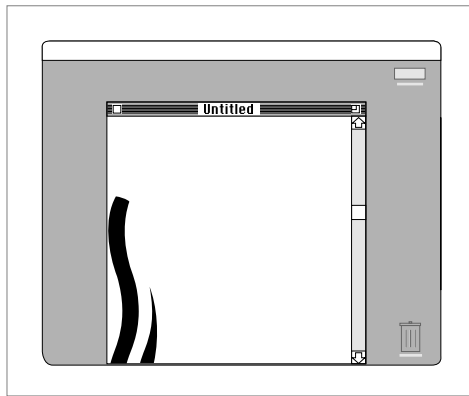


View-Related Objects

The right side of Figure 7-18 shows the vase shape after the view device clip has been applied to it. The view device clip for the monitor represents its imaging area, exclusive of the menu bar. In this case, the view device clip cuts off part of the visible area of the view port, so the lower part of the vase shape (shaded gray in Figure 7-18) is not drawn on this device.

Figure 7-19 shows the example vase shape as actually displayed, after all clipping and coordinate conversions have been applied. The clipped and stretched vase shape is partially scrolled out of view in its window, and the lower part of the window is clipped by the bottom edge of the monitor.

Figure 7-19 The shape as finally displayed



It is seldom necessary to work in device space because QuickDraw GX performs this conversion for you. QuickDraw GX also handles modifications to the view device mappings to match the monitor configuration. For example, if you use the Monitors control panel to change the relative positions of monitors on a Macintosh system, QuickDraw GX handles the changes for you.

Using View-Related Objects

View-related objects define the drawing environment for a QuickDraw GX application. Often, you set up your view ports, view devices and view groups when you set up other application structures. Because of the interrelationships between these objects, setting up an offscreen or onscreen environment and manipulating it involves creating and setting up several objects.

This section describes how you can

- create and use view ports, and analyze shapes in view ports
- create and use view devices, and analyze shapes on view devices
- create and use view groups for offscreen drawing, and analyze shapes in view groups

Using View Ports

This section demonstrates how to use QuickDraw GX view ports. It shows how you can

- create and manipulate view port objects and their properties
- get and set a view port's clip and mapping
- set up a view port hierarchy attached to a window
- support scrolling in a window
- identify the view devices of a view port and the view ports of a shape
- measure a shape in the local space of a view port

Creating and Manipulating View Port Objects

QuickDraw GX provides several functions with which you can create a new view port. To create a view port that is the root view port of a hierarchy and is attached to a Macintosh window, you use the function `GXNewWindowViewPort`, described in the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. To create child view ports for that root parent, or to create a root view port for offscreen drawing, you use the `GXNewViewPort` function. You can also create a new view port object by copying an existing one with the `GXCopyToViewPort` function; see Listing 7-2 on page 7-44 for an example of this method.

Replacing the default view port

If your application draws only within windows, you may want to replace the default transform object (which references the default view port) for each shape type. You could replace it with a transform object that directly references a window view port (a view port attached to a Macintosh window). Alternatively, you could replace it with a transform that references a view port you have designated as the “current” view port; you could then redirect drawing to a window view port by assigning the window view port as the parent of the current view port. Or you could take a different approach and explicitly assign a child view port of a window view port to each shape as it is created, using the `GXSetShapeViewPorts` function. ♦

Once you have created a view port object, you can customize its features using the techniques described in the following section.

You can test if two view port-object references refer to the same view port object by simply testing the references for equality. You can also test a view port for equality with another view port with the `GXEqualViewPort` function. For two view port objects to be equal, their mappings, clips, dithers, halftones, attributes, parent view ports, and view groups must be identical; if one view port is attached to a window, the other view port must be attached to the same window. The tag lists or child view ports of the view ports need not be identical. View port object copies created with the `GXCopyToViewPort` function are always equal to the view port from which they were copied.

View-Related Objects

To delete your application's reference to a view port object, call the `GXDisposeViewPort` function. Because view port objects have no owner count, calling `GXDisposeViewPort` actually releases the memory allocated for that view port object, and invalidates all other references to it. Therefore, once you have disposed of a view port, other transform objects that reference that view port will have invalid view port references in their view port lists. This causes no error when you try to draw; drawing simply does not occur to view ports whose references are invalid.

The following code fragment first creates a Macintosh window (`sampleWindow`), and then uses `GXNewWindowViewPort` to create a view port attached to it. When it no longer needs them, the code disposes of the view port and then the window. The `NewWindow` function and its parameters, and the `DisposeWindow` function, are described in the Window Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials*.

```
sampleWindow = NewWindow( nil, &windowRect, "\p", true,
                          documentProc, (WindowPtr)-1L, true, 0L );
aViewPort = GXNewWindowViewPort(sampleWindow);
.
.  /* use the window and view port */
.
GXDisposeViewPort(aViewPort);
DisposeWindow(sampleWindow);
```

The following line of code creates a view port that is not attached to a window. You might use this call to create a view port that is to be the child of another view port. The code assigns the new view port to the `gxScreenViewDevices` view group, the view group for all onscreen drawing:

```
myChildViewPort = GXNewViewPort(gxScreenViewDevices);
```

A more general way to assign the view group parameter is to first call the `GXGetViewPortViewGroup` function to determine the view group of the intended parent view port, and use that result as the parameter for `GXNewViewPort`. See, for example, Listing 7-5 on page 7-47.

The `GXNewViewPort` function is described on page 7-70. The `GXCopyToViewPort` function is described on page 7-72. The `GXEqualViewPort` function is described on page 7-73. The `GXDisposeViewPort` function is described on page 7-71.

Manipulating View Port Object Properties

This section describes how to manipulate the dither, halftone, view group, attributes, and tag list properties of a view port object:

- To manipulate the dither level, you use the functions `GXGetViewPortDither` and `GXSetViewPortDither`.
- To manipulate the halftone structure, you use the functions `GXGetViewPortHalftone` and `GXSetViewPortHalftone`.
- To manipulate the view group reference, you use the functions `GXGetViewPortViewGroup` and `GXSetViewPortViewGroup`.
- To manipulate the view port attributes, you use the functions `GXGetViewPortAttributes` and `GXSetViewPortAttributes`.
- To manipulate the view port tag list, you use the functions `GXGetViewPortTags` and `GXSetViewPortTags`.

How to manipulate other view port properties is described in subsequent sections, starting with “Getting and Setting a View Port’s Clip and Mapping” on page 7-44.

Getting and Setting a View Port’s Dither, Halftone, and Attributes

Listing 7-1 is an example of code that sets the dither level, halftone structure, and view port attributes of the view port `myViewPort`. For the halftone structure (`myHalfTone`), the code sets all of its values, including the background color and the dot color in HSV color space. The tint type selected, however, is luminance tint, meaning that only the lightness of the input color is used to calculate the proportion of dot and background to use for the halftone. The attributes specify a grayscale view port, meaning that the dot and background colors are also drawn in gray.

Listing 7-1 Changing a view port’s dither, halftone, and attributes

```
gxViewPort  myViewPort
gxHalftone  myHalfTone;

GXSetViewPortAttributes(myViewPort, gxGrayPort);
GXSetViewPortDither (myViewPort, 4);

myHalfTone.angle = ff(6);
myHalfTone.frequency = ff(24);
myHalfTone.method = gxDispersedDot;
myHalfTone.tinting = gxLuminanceTint;
myHalfTone.tintSpace = gxHSVSpace;
```

View-Related Objects

```

myHalfTone.backgroundColor.space = gxHSVSpace;
myHalfTone.backgroundColor.profile = nil;
myHalfTone.backgroundColor.element.hsv.value = 0xFFFF;
myHalfTone.backgroundColor.element.hsv.saturation = 0xCCCD;
myHalfTone.backgroundColor.element.hsv.hue = 0x8000;

myHalfTone.dotColor.space = gxHSVSpace;
myHalfTone.dotColor.profile = nil;
myHalfTone.dotColor.element.hsv.value = 0xFFFF;
myHalfTone.dotColor.element.hsv.saturation = 0xAD1C;
myHalfTone.dotColor.element.hsv.hue = 0xE4F9;

GXSetViewPortHalftone(myViewPort, &myHalfTone);

```

Note

Dithers and halftones are mutually exclusive. The halftone in this example overrides the dither, so dithering is not performed at drawing. ♦

The `GXGetViewPortDither` function is described on page 7-80; the `GXSetViewPortDither` function is described on page 7-80.

The `GXGetViewPortHalftone` function is described on page 7-81; the `GXSetViewPortHalftone` function is described on page 7-82.

The `GXGetViewPortAttributes` function is described on page 7-89; the `GXSetViewPortAttributes` function is described on page 7-90.

Getting and Setting a View Port's View Group

Listing 7-2 demonstrates the use of the `GXSetViewPortViewGroup` function. It is part of a routine that creates an offscreen view group (`newGroup`) that is a copy of an existing view group (`group`). For each view port in the onscreen view group, the routine creates a copy. It then uses `GXSetViewPortViewGroup` to assign the proper view group to the new view port. (Listing 7-10 on page 7-54 shows another part of the same routine.)

The routine uses the `count` variable to decrement through the list of view ports (`oldList`, retrieved through two consecutive calls to `GXGetViewGroupViewPorts`) belonging to the onscreen view group. The code simultaneously builds, for its own purposes, a list (`newList`) of view ports for the offscreen view group, using `GXCopyToViewPort` and `GXSetViewPortViewGroup` to copy each view port into the offscreen view group and set its view group property.

View-Related Objects

Listing 7-2 Copying the view ports from one view group to another

```

long          portCount = GXGetViewGroupViewPorts(group, nil);
long          count = portCount;
gxViewPort    *oldPortList = (void *)NewPtr(portCount *
                                         sizeof(gxViewPort));
gxViewPort    *oldList = oldPortList;
gxViewPort    *newPortList = (void *)NewPtr(portCount *
                                         sizeof(gxViewPort));
gxViewPort    *newList = newPortList;
GXGetViewGroupViewPorts(group, oldPortList);
while (count-- > 0)
    GXSetViewPortViewGroup(*newList++ = GXCopyToViewPort(nil,
                                         *oldList++), newGroup);

```

The `GXGetViewPortViewGroup` function is described on page 7-88. The `GXSetViewPortViewGroup` function is described on page 7-88.

Getting and Setting a View Port's Tag References

You can examine the list of references to tag objects currently associated with a view port object using the `GXGetViewPortTags` function. Once you create a tag object, you can attach it to a view port object using the `GXSetViewPortTags` function. You can attach as many tag objects as you like to a view port object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetViewPortTags` function is described on page 7-91. The `GXSetViewPortTags` function is described on page 7-92.

Getting and Setting a View Port's Clip and Mapping

The clip and mapping properties of a view port control the visibility and location of its contents. For onscreen view ports attached to Macintosh windows, you do not directly set the clip or mapping properties; you move or resize the window with Window Manager calls, and QuickDraw GX automatically updates the view port's clip and mapping. For child view ports of window view ports, however, and for all offscreen view ports, you must set the clip and mapping yourself.

You use the functions `GXGetViewPortMapping`, and `GXSetViewPortMapping` to set a view port mapping to move the contents of the view port, such as when scrolling. You also set the view port mapping to provide scaled, rotated, or otherwise altered views of the view port's contents. Listing 7-3 shows an example that uses those functions plus `GXGetViewPortClip` and `GXScaleMapping` to scale the view port `myViewPort` to 200 percent of its original size, about an origin at the center of the view port's clip.

Listing 7-3 Changing a view port's mapping

```

gxViewport      myViewport
gxMapping       myViewportMapping;
gxShape         myViewportFrame;
gxPoint         center;

GXGetViewportMapping(myViewport, &myViewportMapping);
myViewportFrame = GXGetViewportClip(myViewport);
GXGetShapeCenter(myViewportFrame, 0L, &center);
GXScaleMapping(&myViewportMapping, ff(2), ff(2),
               center.x, center.y);
GXSetViewportMapping(myViewport, &myViewportMapping);
GXDisposeShape(myViewportFrame);

```

Note that, because the `GXGetViewportClip` function creates a shape object, the code in Listing 7-3 disposes of the shape after using it. The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*; the `GXSetShapeMapping` function is described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Getting the global mapping

If a view port is a child view port in a hierarchy, its mapping converts from local space into the local space of its parent view port, not directly into global space. If you want to determine the resultant mapping obtained by concatenating the mappings of a view port and all its parents—a mapping from local space all the way into global space—you can use `GXGetViewportGlobalMapping`, which is described on page 7-79. For an example of its use, see Listing 7-11 on page 7-57. ♦

You can use the `GXSetViewportClip` function to set a view port clip to initialize or change the visible area of the view port. Listing 7-4 is a routine that sets up the clip of a child view port (`gcontentViewport`) whose parent is the root view port attached to a Macintosh window (`theWindow`). The routine makes the clip the same size as the window's content area, minus the width of the scroll bars on the window's side and bottom.

The listing uses the application-defined function `GetWindowBoundsShape` to determine the rectangle shape corresponding to the content area of the window. That function retrieves a QuickDraw rectangle corresponding to the port rectangle of the window, and then converts it to a QuickDraw GX rectangle using the `GXConvertQDPoint` function.

Listing 7-4 Setting a view port clip

```

void ResetContentViewPortClip (WindowPtr theWindow)
{
    gxRectangle    viewRect;
    gxShape        contentViewPortClipShape;

    /* get the size of the window port rect */
    GetWindowBoundsShape(theWindow, &viewRect);

    /* Adjust the rectangle to accommodate the scroll bars */
    viewRect.right -= ff(kScrollBarWidth - 1);
    viewRect.bottom -= ff(kScrollBarWidth - 1);

    /* assign it as the clip shape */
    contentViewPortClipShape = GXNewRectangle(&viewRect);
    GXSetViewPortClip(gContentViewPort, contentViewPortClipShape);
    GXDisposeShape (contentViewPortClipShape);
}

```

The `GXConvertQDPoint` function is described in the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. The `GXNewRectangle` function, which creates a rectangle shape, is described in the geometric shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The `GXGetViewPortClip` function is described on page 7-74; the `GXSetViewPortClip` function is described on page 7-75.

The `GXGetViewPortMapping` function is described on page 7-77; the `GXSetViewPortMapping` function is described on page 7-78.

Setting Up the View Port Hierarchy for a Window

Setting up a view port hierarchy means assigning the appropriate parent view port and child view port references to all view ports involved. The functions you use are `GXGetViewPortParent`, `GXSetViewPortParent`, `GXGetViewPortChildren`, and `GXSetViewPortChildren`. Take these steps to set up a simple hierarchy in which a child view port is used for drawing a window's content:

1. Create the child view port in the window view port's view group.
2. Create a clip shape and assign it to the child view port. Set the child view port's mapping.
3. Assign the window view port as the parent of the child view port.
4. Dispose of the clip shape.

Note that you do not have to add the child view port to the window view port's list of children; when you set the parent view port property of the child view port, QuickDraw GX adds the child view port to the parent's list of child view ports.

Listing 7-5 is an example that sets up such a hierarchy. It creates a view port with the `GXNewViewPort` function, and uses `GXGetViewPortViewGroup` to find out what view group to assign it to. The code assigns properties to the view port with `GXSetViewPortClip`, `GXSetViewPortMapping`, and `GXSetViewPortParent`. The window view port is `windowParentViewPort`, and the rectangle `viewRect` defines a clipping area that is the size of the window minus the area reserved for scroll bars.

Listing 7-5 Setting up a view port for a window

```
gxRectangle    viewRect;
gxViewPort    windowParentViewPort;
gxShape        contentViewPortShape;
. /*
.     Create window view port with GXNewWindowViewPort. Make
.     viewRect equal to port rectangle minus scroll bars.
. */
gcontentViewPort = GXNewViewPort
                    (GXGetViewPortViewGroup(windowParentViewPort));
contentViewPortShape = GXNewRectangle(&viewRect);
GXSetViewPortClip(gcontentViewPort, contentViewPortShape);
GXSetViewPortMapping(gcontentViewPort, nil);
GXSetViewPortParent(gcontentViewPort, windowParentViewPort);
GXDisposeShape (contentViewPortShape);
```

Once you have set up a hierarchy, if you want to draw into the child view port—and thus onscreen—you must place a reference to the child view port in a transform object's view port list, and the shapes you draw must reference that transform.

The `GXGetViewPortParent` function is described on page 7-84; the `GXSetViewPortParent` function is described on page 7-84.

The `GXGetViewPortChildren` function is described on page 7-86; the `GXSetViewPortChildren` function is described on page 7-87.

Supporting Scrolling in a Window

To support scrolling in a view port attached to a Macintosh window, you need to create a child view port of the window view port, and draw into it rather than into its parent. QuickDraw GX prevents you from changing the mapping or clip of a view port directly attached to a Macintosh window.

When the user scrolls the window, you manipulate the child view port's mapping to scroll the content. When the user resizes the window, you manipulate the child view port's clip to fit the new window shape. When the user moves the window, you do nothing; QuickDraw GX takes care of positioning both parent and child view ports.

View-Related Objects

Listing 7-6 is an example of a scrolling routine that scrolls the contents of a child view port (gcontentViewPort) by specified vertical and horizontal amounts (hScroll and vScroll), in response to mouse-down events in the scroll bars of a window (theWindow). The event-dispatching routine calls this scrolling routine after it has calculated how much scrolling is required. After the scrolling routine executes, a separate routine (not shown) updates the appearance of the scroll bars.

Listing 7-6 Supporting scrolling in a child view port

```
void DoScroll(WindowPtr theWindow, short hScroll, short vScroll)
{
    Rect        scrollRect;
    Point       scrollPt;
    RgnHandle    myRgn;
    gxMapping    viewPortMapping;

    if ((hScroll == 0) && (vScroll == 0)) return;

    /*
       Get the child view port's mapping, adjust it for the
       horizontal and vertical scroll, then reassign it to the
       view port. The next drawing action will then reflect the
       scrolled positions of shapes in the view port.
    */
    GXGetViewPortMapping(gcontentViewPort, &viewPortMapping);
    MoveMapping(&viewPortMapping, ff(hScroll), ff(vScroll));
    GXSetViewPortMapping(gcontentViewPort, &viewPortMapping);

    /*
       Shift the pixels representing the already drawn contents of
       window, so that less will have to be drawn when the window
       is updated. Only the parts scrolled into view (specified by
       the update region myRgn) will need to be redrawn.
    */
    scrollRect = theWindow->portRect;
    scrollRect.right -= (kScrollBarWidth-1);
    scrollRect.bottom -= (kScrollBarWidth-1);

    SetPort(theWindow);
    myRgn = NewRgn();
    ScrollRect(&scrollRect, hScroll, vScroll, myRgn);
}
```

View-Related Objects

```

/* update origin position in app's extended window record */
SetPt(&scrollPt, hScroll, vScroll);
AddPt(scrollPt, &((MyWindowPeek)theWindow)->origin);

/* redraw the window and dispose of the region handle */
DrawWindow(theWindow);
DisposeRgn(myRgn);
}

```

Identifying a View Port's View Devices

The `GXGetViewPortViewDevices` function returns a list of all view devices that could be affected by shapes drawn in a given view port. Within the view group of the view port, all view device objects whose clip areas overlap the clip area of the view port appear in the list returned by this function.

You can use `GXGetViewPortViewDevices` to determine whether a given view device can be affected by drawing into a given view port. You can also use it to examine the properties of all devices that you might draw to, perhaps in order to assign appropriate properties to offscreen view devices.

Listing 7-7 is a library function (`SetShapeFastXorTransfer`) that uses another library function (`SetInkFastXorTransfer`) to set up a shape's color and an XOR transfer mode so that drawing with that color will cause a specified highlight color to replace the background color in the destination. The `SetInkFastXorTransfer` function is not shown here. Listing 7-7 is shown because it uses `GXGetViewPortViewDevices` to get a list of the view devices a view port can draw to, although it actually uses only the first view device in the list.

Listing 7-7 Setting a shape color for XOR highlighting

```

void SetShapeFastXorTransfer(gxShape source, gxColor *background,
                           gxColor *result)
{
    long          viewPortCount, viewDeviceCount;
    void          *buffer;
    gxViewPort    vp;
    gxViewDevice  vd;
    gxInk         inky;

    /* get size of view port list, then allocate buffer for it */
    viewPortCount = GXGetTransformViewPorts(
                      GXGetShapeTransform(source), nil);
    buffer = NewPtr(sizeof(gxViewPort) * viewPortCount);
}

```

View-Related Objects

```

/* check for memory error (not shown), then get list itself */
GXGetTransformViewPorts(GXGetShapeTransform(source),
                        (gxViewport *)buffer);

/* get no. of view devices, then allocate buffer for list */
viewDeviceCount = GXGetViewportViewDevices(
                    vp = *(gxViewport *)buffer, nil);
.
. /* check for memory error (not shown), then get list itself */
.
GXGetViewportViewDevices(vp, (gxViewDevice *)buffer);
vd = *(gxViewDevice *)buffer;
DisposePtr(buffer);

/* get shape's ink; if shared, assign a copy of the ink */
if (GXGetInkOwners(inky = GXGetShapeInk(source)) > 1)
{
    GXSetShapeInk(source, inky = GXNewInk());
    GXDisposeInk(inky);
}
/* set ink's transfer mode and suppress dithering */
SetInkFastXorTransfer(inky, vd, vp, background, result);
GXSetShapeInkAttributes(source,
                        GXGetShapeInkAttributes(source) |
                        gxSuppressDitherInk);
}

```

The `GXGetViewportViewDevices` function is described on page 7-94.

Identifying a Shape's View Ports

The `GXGetShapeGlobalViewPorts` function returns a list of all view ports that a shape could actually appear in if it were drawn. If a shape's transform references a view port, and if that view port's clip does not totally exclude the shape from the visible part of the view port, the view port appears in the list returned by this function.

You can use `GXGetShapeGlobalViewPorts` to avoid the overhead of drawing shapes that cannot be visible. You can also use it as an input to the `GXGetShapeGlobalViewDevices` function to determine all the devices on which a given shape can appear.

The `GXGetShapeGlobalViewPorts` function is described on page 7-95. The `GXGetShapeGlobalViewDevices` function is described on page 7-115.

Measuring a Shape in Local Space

The `GXGetShapeLocalBounds` function measures the bounding rectangle of a shape in local coordinates—that is, after the transform mapping has been applied to the shape geometry. You can use `GXGetShapeLocalBounds` to compare the positions and sizes of two shapes in the same view port, even if they do not share the same transform object. (To compare the positions and sizes of two shapes in different view ports, use the `GXGetShapeGlobalBounds` function; to measure a shape on a view device, use `GXGetShapeDeviceBounds`.)

Listing 7-8 is a function in a shape-editing program. It draws a gray box representing the bounding rectangle for each shape in a list of shapes passed to it. It calls `GXGetShapeLocalBounds` for each shape, and then defines and draws a rectangle shape whose geometry matches that bounding rectangle. Regardless of how each shape has been modified by its own transform mapping, `GXGetShapeLocalBounds` returns a rectangle (whose default transform has an identity mapping) that exactly matches the transformed shape's bounding rectangle when drawn in the view port.

Listing 7-8 makes use of the library function `SetShapeCommonColor` to set the bounding rectangle's color.

Listing 7-8 Locating the bounding rectangles of a list of shapes in a view port

```
void ShowLocalBounds(gxShape *plstShape, long shapeCount)
{
    register gxShape      *pShape, rectShape;
    gxRectangle           bounds;

    pShape = plstShape + shapeCount - 1;    /* no. of last shape */

    /* define a framed gray rectangle shape for the bounds */
    rectShape = GXNewShape(gxRectangleType);
    GXSetShapeFill(rectShape, gxClosedFrameFill);
    SetShapeCommonColor(rectShape, gxGray);

    /* go through shape list, get and draw local bounds for each */
    while (shapeCount--)
    {
        GXGetShapeLocalBounds(*pShape--, &bounds);
        GXSetRectangle(rectShape, &bounds);
        GXDrawShape(rectShape);
    }
    GXDisposeShape(rectShape);
}
```

View-Related Objects

The `GXGetShapeLocalBounds` function is described on page 7-96.

The `GXGetShapeGlobalBounds` function is described on page 7-125; the

`GXGetShapeDeviceBounds` function is described on page 7-116.

Using View Devices

This section demonstrates how to use QuickDraw GX view devices. It shows how you can

- create and manipulate view device objects and their properties
- get and set a view device's clip and mapping
- identify the view devices of a shape
- measure a shape in the device space of a view device
- hit-test a shape on a device

Creating and Manipulating View Device Objects

Normally, your application needs to create view device objects only for offscreen drawing. QuickDraw GX creates view device objects for all attached screen devices at startup. If you do need to create a view device object, QuickDraw GX provides the `GXNewViewDevice` function, to which you must supply a view group reference and a bitmap shape representing the device's imaging area and characteristics. You can also create a new view device object by copying an existing one, using the `GXCopyToViewDevice` function.

Once you have created a view device object, you can customize its features using the techniques described in the following sections.

You can test if two view device-object references refer to the same view device object by simply testing the references for equality. You can also test a view device for equality with another view device with the `GXEqualViewDevice` function. For two view device objects to be equal, their clips, mappings, bitmap shapes, and attributes must be identical, and they must be in the same view group, represent the same Macintosh graphics device (same `GDevice` record), and point to the same pixel image, color set, and color profile. Their tag lists need not be identical. View device object copies created with the `GXCopyToViewDevice` function are always equal to the view device from which they were copied.

To delete your application's reference to a view device object, call the `GXDisposeViewDevice` function. Because view device objects have no owner count, calling `GXDisposeViewDevice` actually releases the memory allocated for that view device object, and invalidates all other references to it.

View-Related Objects

Listing 7-9 is a portion of a printer driver routine that sets up a default printing view device. It first sets up a bitmap structure with default values, and then creates a bitmap shape to pass to `GXNewViewDevice`. The view group for this view device is `gxScreenViewDevices`, because it is to be used for printing, not offscreen drawing.

Listing 7-9 Creating a new view device

```
gxBitmap      aBitmap;
gxViewDevice  vd;
aBitmap.pixelSize = 1;
aBitmap.rowBytes = 0;
aBitmap.width    = 0;
aBitmap.height   = 0;
aBitmap.image     = nil;
aBitmap.space     = gxNoSpace;
aBitmap.set       = nil;
aBitmap.profile   = nil;

theBitmap = GXNewBitmap(&aBitmap, nil);
/* error-check here (not shown) */
.
.
vd = GXNewViewDevice(gxScreenViewDevices, theBitmap);
/* error-check here (not shown) */
.
.
```

When the driver is finished with the view device, it disposes of it with this line:

```
GXDisposeViewDevice(vd);
```

Listing 7-10 demonstrates creating a new view device by using the `GXCopyToViewDevice` function. Like Listing 7-2 on page 7-44, it is part of a routine that creates an offscreen view group (`newGroup`) that is a copy of an existing view group (`group`). For each view device in the onscreen view group, the routine creates a copy and assigns it to the offscreen view group.

The routine decrements the count variable to control incrementing through the list of view devices (`list`) belonging to the onscreen view group.

Listing 7-10 Copying the view devices from one view group to another

```

long          deviceCount = GXGetViewGroupViewDevices(group, nil);
long          count = deviceCount;
gxViewDevice *deviceList = (void *)NewPtr(deviceCount *
                                         sizeof(gxViewDevice));
gxViewDevice *list = deviceList;
GXGetViewGroupViewDevices(group, deviceList);
while (count-- > 0)
{
    GXSetViewDeviceViewGroup(*list = GXCopyToViewDevice(nil,
                                                         *list), newGroup);
    list++;
}

```

The `GXNewViewDevice` function is described on page 7-98. The

`GXDisposeViewDevice` function is described on page 7-99.

The `GXCopyToViewDevice` function is described on page 7-100. The

`GXEqualViewDevice` function is described on page 7-101.

Manipulating View Device Object Properties

This section describes how to manipulate the bitmap, view group, and attributes properties of a view device object:

- To manipulate the bitmap structure, you use the functions `GXGetViewDeviceBitmap` and `GXSetViewDeviceBitmap`.
- To manipulate the view group reference, you use the functions `GXGetViewDeviceViewGroup` and `GXSetViewDeviceViewGroup`.
- To manipulate the view device attributes, you use the functions `GXGetViewDeviceAttributes` and `GXSetViewDeviceAttributes`.
- To manipulate the view device tag list, you use the functions `GXGetViewDeviceTags` and `GXSetViewDeviceTags`.

How to manipulate other view device properties is described in subsequent sections, starting with “Getting and Setting a View Device’s Clip and Mapping” on page 7-56.

Getting and Setting a View Device's Bitmap

The following code fragment is a function that uses `GXGetViewDeviceBitmap` to gain access to a copy of the color set of a view device, clone its reference (so it won't be deleted when its bitmap is disposed of), and then return it as a function result. The function needs the bitmap shape itself only temporarily, and therefore disposes of it after extracting the color set reference from it.

```
gxColorSet GetViewDeviceColorSet(gxViewDevice source)
{
    register gxShape    bitmapShape =
                                GXGetViewDeviceBitmap(source);
    register gxColorSet result = GetShapeColorSet(bitmapShape);
    if (result)
        GXCloneColorSet(result);
    GXDisposeShape(bitmapShape);
    return result;
}
```

The following code fragment is a function that uses `GXSetViewDeviceBitmap` to assign a color profile to a view device. The function disposes of its reference to the bitmap shape after assigning it to the view device.

```
void SetViewDeviceColorProfile(gxViewDevice target,
                                gxColorProfile profile)
{
    register gxShape bitmapShape = GXGetViewDeviceBitmap(target);

    SetShapeColorProfile(bitmapShape, profile);
    GXSetViewDeviceBitmap(target, bitmapShape);
    GXDisposeShape(bitmapShape);
}
```

The `GXGetViewDeviceBitmap` function is described on page 7-107; the `GXSetViewDeviceBitmap` function is described on page 7-108.

Getting and Setting a View Device's View Group

You can use the `GXGetViewDeviceViewGroup` function to retrieve the view group that a view device belongs to, and you can use the `GXSetViewDeviceViewGroup` to change the view group of a view device. Listing 7-10 on page 7-54 shows an example of using `GXSetViewDeviceViewGroup` to reassign the copy of a view device from one view group to another.

The `GXGetViewDeviceViewGroup` function is described on page 7-109; the `GXSetViewDeviceViewGroup` function is described on page 7-109.

Getting and Setting a View Device's Attributes and Tag References

You can examine the attributes of a view device object using the `GXGetViewDeviceAttributes` function. You can set the attributes of a view device object using the `GXSetViewDeviceAttributes` function. By setting attributes, you can influence whether the device bitmap is placed on an accelerator card and whether the device is active or inactive.

You can examine the list of references to tag objects currently associated with a view device object using the `GXGetViewDeviceTags` function. Once you create a tag object, you can attach it to a view device object using the `GXSetViewDeviceTags` function. You can attach as many tag objects as you like to a view device object.

Tag objects and the basic functions for manipulating them are described in the chapter “Tag Objects” in this book. That chapter also lists the common tag types defined and reserved by Apple Computer, Inc.

The `GXGetViewDeviceAttributes` function is described on page 7-110; the `GXSetViewDeviceAttributes` function is described on page 7-111. View device attributes are described in the section “View Device Attributes” on page 7-27.

The `GXGetViewDeviceTags` function is described on page 7-112. The `GXSetViewDeviceTags` function is described on page 7-113.

Getting and Setting a View Device's Clip and Mapping

The clip and mapping properties of a view device control its active imaging area, its scale (pixel size), and its position in global space. For onscreen view devices and printing view devices, you cannot change the clip or mapping properties; they are set by QuickDraw GX. For offscreen view devices, you can set the clip and mapping yourself. The functions you use are `GXGetViewDeviceClip`, `GXSetViewDeviceClip`, `GXGetViewDeviceMapping`, and `GXSetViewDeviceMapping`.

Listing 7-11 is a utility routine that returns a mapping matrix that converts from local space (or from the identity mapping, if the view port is `nil`) to device space (or to global space, if the view device is `nil`). It uses `GXGetViewDeviceMapping` to retrieve the view device's mapping matrix. (If the view device is `nil`, the routine uses the mapping matrix returned by the `GXGetViewPortGlobalMapping` function.) The routine also makes use of the `MapMapping` function, described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Listing 7-11 Returning the mapping from local to device space

```

static void GetSpaceMapping(gxViewPort port, gxViewDevice device,
                           gxMapping *map)
{
    if (port)
        GXGetViewPortGlobalMapping(port, map);
    else
        ResetMapping(map);

    if(device)
    {
        gxMapping temp;
        MapMapping(map, GXGetViewDeviceMapping(device, &temp));
    }
}

```

The following code fragment is part of a printer driver routine that sets up a default view device. This section of code rescales the mapping matrix (vdMapping) of the view device (vd) from the default resolution (72 ppi, as specified by the identity matrix) to the horizontal and vertical resolution of the printer (kHorizHighRes and kVertHighRes). To do the scaling, the code uses the ScaleMapping function, described in the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. It then uses GXSetViewDeviceMapping to assign the scaled mapping to the view device.

```

Fixed    xScale;
Fixed    yScale;

xScale = FixRatio(kHorizHighRes, 72);
yScale = FixRatio(kVertHighRes, 72);

ResetMapping(&vdMapping);
ScaleMapping(&vdMapping, xScale, yScale, ff(0), ff(0));
GXSetViewDeviceMapping(vd, &vdMapping);

```

The GXGetViewDeviceClip function is described on page 7-102; the GXSetViewDeviceClip function is described on page 7-103.

The GXGetViewDeviceMapping function is described on page 7-105; the GXSetViewDeviceMapping function is described on page 7-106.

View-Related Objects

Identifying a Shape's View Devices

The `GXGetShapeGlobalViewDevices` function returns a list of all view devices that a shape would actually appear in if it were drawn. The function can test the shape against all the shape's view ports, or you can specify a single view port for the test.

You can use `GXGetShapeGlobalViewDevices` to avoid the overhead of testing the drawing characteristics (such as the colors) of shapes on devices that they cannot be drawn to.

Listing 7-12 is part of a library routine that sets up a data structure for offscreen drawing through a given view port. This part of the code creates a full shape—which covers all of coordinate space—and passes it to `GXGetShapeGlobalViewDevices` to derive a count of all view devices that could be drawn on through the given port. (Note that, for the purpose of retrieving all the view devices of a view port, you could also use the `GXGetViewportViewDevices` function.)

Listing 7-12 Setting up a data structure for offscreen drawing

```
viewPortBuffer NewViewPortBuffer(gxViewport port)
{
    viewPortBuffer          buffersHandle;
    viewPortBufferRecord    *buffers;
    gxTransform             xform;
    gxShape                 area;
    long                   deviceCount;
    short                  i;

    NilParamReturnNil(port);          /* error check port parameter */
    area = GXNewShape(gxFullType);
    xform = GXNewTransform();
    GXSetTransformViewPorts(xform, 1, &port);
    GXSetShapeTransform(area, xform);
    GXDisposeTransform(xform);
    deviceCount = GXGetShapeGlobalViewDevices(area, port, nil);
    .
    .  /* continued as Listing 7-13 on page 7-61 */
    .
}
```

The `GXGetShapeGlobalViewDevices` function is described on page 7-115.

Measuring a Shape in Device Space

You can use view device functions to measure a shape on a device in three ways:

- The `GXGetShapeDeviceBounds` function measures the position and size of the bounding rectangle of a shape on a device, in device coordinates.
- The `GXGetShapeDeviceArea` function determines the area of a shape (in pixels) on a device.
- The `GXGetShapeDeviceColors` function determines the colors with which a shape would be drawn on a device.

The `GXGetShapeDeviceBounds` function determines whether any part of a shape intersects a view device, and if so, returns the bounding rectangle of that part of the shape in device coordinates. You can thus use `GXGetShapeDeviceBounds` to measure the size of a shape on a view device and to compare it with other shapes on the device. (To measure a shape in the local space of a view port, you can use the `GXGetShapeLocalBounds` function; to measure a shape in the global space of a view group, use `GXGetShapeGlobalBounds`.)

The following is a fragment of a function that converts a QuickDraw GX shape on a device into a QuickDraw picture. It uses `GXGetShapeDeviceBounds` to get the shape's bounding rectangle on the device, converts that rectangle into a QuickDraw `rect` structure, further converts it to QuickDraw local coordinates, and uses that to define the picture bounding rectangle. After that, the function converts the shape itself (not shown).

```
GXGetShapeDeviceBounds(theShape, 0, 0, &shapeBounds);
picRect.left          = FixedToInt(shapeBounds.left);
picRect.top           = FixedToInt(shapeBounds.top);
picRect.right         = FixedToInt(shapeBounds.right);
picRect.bottom        = FixedToInt(shapeBounds.bottom);
GlobalToLocal((Point*) &picRect.top);
GlobalToLocal((Point*) &picRect.bottom);

thePicture = OpenPicture(&picRect);
.
.  /* convert the shape (not shown) */
.
```

The QuickDraw functions `GlobalToLocal` and `OpenPicture`, and the data types `Point` and `Rect` are described in *Inside Macintosh: Imaging With QuickDraw*.

Note

You do not need to write special functions to convert QuickDraw pictures into QuickDraw GX shapes. You can use the QuickDraw-to-QuickDraw GX translator for that; see the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*. ♦

View-Related Objects

The `GXGetShapeDeviceBounds` function is described on page 7-116. The

`GXGetShapeDeviceArea` function is described on page 7-118.

The `GXGetShapeDeviceColors` function is described on page 7-119.

`GXGetShapeGlobalBounds` function is described on page 7-125.

The `GXGetShapeLocalBounds` function, described on page 7-96.

Hit-Testing a Shape on a Device

The `GXHitTestDevice` function is one of several hit-testing functions provided by QuickDraw GX. Hit-testing in general is described in the chapter “Introduction to QuickDraw GX” in this book; shape parts for hit-testing are described in the chapter “Transform Objects” in this book.

You use `GXHitTestDevice` instead of `GXHitTestShape` or `GXHitTestPicture`—or before them—when it is important to take into account whether a shape is actually visible on a device. Unlike `GXHitTestShape` and `GXHitTestPicture`, `GXHitTestDevice` accounts for clipping and does not return successful hits for shapes that are not actually drawn.

Another significant difference is that the tolerance for `GXHitTestDevice` defines a rectangular area of pixels, not the circular geometry area used by `GXHitTestShape` and `GXHitTestPicture`. Thus you can use the tolerance value for `GXHitTestDevice` as something like a clip area, expanding it to cover an entire window or contracting it to one or a few complete pixels.

What `GXHitTestDevice` does not do that `GXHitTestShape` and `GXHitTestPicture` do is analyze the parts of a shape. If you are hit-testing in order to highlight specific parts of a shape, for example, you can first call `GXHitTestDevice` to determine which shape was actually hit, and then call `GXHitTestShape` or `GXHitTestPicture` to determine the part or parts to highlight.

The `GXHitTestDevice` function is described on page 7-120. The `GXHitTestShape` function is described in the chapter “Shape Objects” in this book. The `GXHitTestPicture` function is described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Using View Groups

This section demonstrates how to use QuickDraw GX view groups. It shows how you can

- create and manipulate view group objects
- set up an offscreen drawing environment
- measure a shape in a view group

Creating and Manipulating View Group Objects

QuickDraw GX provides the `GXNewViewGroup` function to allow you to create a new view group object, and the `GXDisposeViewGroup` function to delete it. Normally, you create a view group only for the purpose of offscreen drawing.

Listing 7-13 is a continuation of the routine in Listing 7-12 on page 7-58 that sets up a data structure for offscreen drawing through a given view port. This part of the code fills in various fields of the `buffers` data structure and calls `GXNewViewGroup` to create an offscreen view group. In addition, it calls `GXNewViewPort` to create an offscreen view port (`slavePort`) in the new group, and `GXGetShapeGlobalViewDevices` to copy all drawable devices from the original onscreen view port (`port`) into a list in the data structure.

Listing 7-13 Setting up a data structure for offscreen drawing

```
.
.  /* continued from Listing 7-12 on page 7-58 */
.
    buffersHandle = (viewPortBuffer) NewHandle(sizeof
                                                (viewPortBufferRecord) + (deviceCount - 1) *
                                                sizeof(gxViewDevice));

    NilParamReturnNil(port);          /* error-check the handle */
    HLock((Handle) buffersHandle);
    buffers = *buffersHandle;
    buffers->group = GXNewViewGroup();
    buffers->masterPort = port;
    buffers->slavePort = GXNewViewPort(buffers->group);
    buffers->area = area;
    buffers->draw = GXNewShape(gxPictureType);
    buffers->deviceCount = deviceCount;
    GXSetViewPortDither(buffers->slavePort,
                        GXGetViewPortDither(port));
    GXGetShapeGlobalViewDevices(area, port, buffers->devices);
```

Once you have created a view group object, you can assign view ports and view devices to it with the `GXSetViewPortViewGroup` function (as described under “Manipulating View Port Object Properties” beginning on page 7-42) and the `GXSetViewDeviceViewGroup` function (as described under “Manipulating View Device Object Properties” beginning on page 7-54).

View-Related Objects

At any time, you can retrieve a list of view ports or view devices that belong to a view group by calling the functions `GXGetViewGroupViewPorts` and `GXGetViewGroupViewDevices`. See Listing 7-2 on page 7-44 for an example of the use of `GXGetViewGroupViewPorts`; see Listing 7-10 on page 7-54 for an example of the use of `GXGetViewGroupViewDevices`.

When you have finished using an offscreen view group, you delete it with the `GXDisposeViewGroup` function. For an example of the use of `GXDisposeViewGroup`, see page 7-63.

The `GXNewViewGroup` function is described on page 7-122. The `GXDisposeViewGroup` function is described on page 7-122.

The `GXGetViewGroupViewPorts` function is described on page 7-123. The `GXGetViewGroupViewDevices` function is described on page 7-124.

Setting Up an Offscreen View Group

This section shows how to set up a view port for offscreen drawing. Examples of most of the steps shown here have already been presented elsewhere in this chapter, although not all together.

Offscreen drawing requires creating a new view group, so that drawing does not conflict with the screen devices view group. You must also create a view port to draw into. To set up the view port, you must create a view device object; however, for offscreen drawing the view device object need not correspond to any physical device. Follow this typical sequence of steps to set up an offscreen view group:

1. Create a new view group.
2. Create a new view device in this view group, specifying a bit map that represents the area that you may want to copy onscreen later.
3. Create a new view port in this view group.
4. Retrieve the bitmap as a shape object of its own, so that you can later draw it directly onscreen.
5. Create a transform object for your shapes, and assign the view port to its view port list.

Listing 7-14 is a partial example of a routine that sets up an offscreen drawing environment. It does not show how the bitmap shape (`bitShape`) for the device is set up. See the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics* for information on bitmap shapes.

Listing 7-14 Setting up a view port and view group for offscreen drawing

```

gxShape      myDraw, bitShape;
gxTransform  myXform;
gxViewDevice myDevice;
gxViewPort   myPort;
gxViewGroup  myGroup;

.
.  /* set up bitmap shape for offscreen view device */
.
myGroup = GXNewViewGroup();
myDevice = GXNewViewDevice(myGroup, bitShape);
myPort = GXNewViewPort(myGroup);
myDraw = GXGetViewDeviceBitmap(myDevice);
myXform = GXNewTransform();
GXSetTransformViewPorts(myXform, 1, &myPort);

```

The `myDraw` shape represents your offscreen drawing buffer. Whenever you draw a shape that has `myXform` as its transform object, drawing takes place offscreen, in the pixel image associated with `myDraw`. If you added a view port in the onscreen view group to the view port list of `myXform`, drawing could take place both offscreen and onscreen simultaneously.

It is useful to have a direct reference to `myDraw` because you can draw it itself, which would have the effect of transferring the offscreen buffer onto the screen (if `myDraw` references a transform object that references onscreen view ports).

When you are finished with offscreen drawing, you can dispose of the objects you have created:

```

GXDisposeShape(myDraw);
GXDisposeTransform(myXform);
GXDisposeViewGroup(myGroup);

```

You do not have to explicitly dispose of your offscreen view port or view device, because calling `GXDisposeViewGroup` causes QuickDraw GX to dispose of all of its view ports and view devices.

Measuring a Shape in Global Space

The `GXGetShapeGlobalBounds` function measures the bounding rectangle of a shape in global coordinates—that is, after the transform mapping has been applied to the shape geometry, and after all view port mappings have been applied. You can thus use `GXGetShapeGlobalBounds` to compare the true positions and sizes of any two shapes in the same view group, even if they do not share the same view port or do not appear on the same view device. (To compare the positions and sizes of two shapes in the same view port, you can use the `GXGetShapeLocalBounds` function; to measure a shape on a view device, use `GXGetShapeDeviceBounds`.)

View-Related Objects

Listing 7-15 is a library function used in offscreen drawing. The function returns the device characteristics (a bitmap structure plus an offset) of a particular “area,” the intersection of an offscreen view device and an offscreen view port. Area-characteristics structures are used by this library to store device-specific drawing information for each area within the offscreen view port occupied by the pixel image of a device. This listing uses `GXGetShapeGlobalBounds` to determine the intersection of the specified shape (area) with the specified view device (device), which determines the size of the image stored in the area-characteristics structure (x).

Listing 7-15 Returning the characteristics of an offscreen device area

```
static areaCharacteristics GetAreaCharacteristics(gxShape area,
                                                gxViewDevice device,
                                                gxViewPort port)
{
    areaCharacteristics x; /* a bitmap structure & location */
    gxRectangle bounds;
    gxShape bitShape;
    gxMapping map;
    .
    .
    . /* get device bitmap and shape bounds on device */
    bitShape = GXGetViewDeviceBitmap(device);
    GXGetShapeGlobalBounds(area, port, nil, &bounds);

    /* fill out the area-characteristics structure */
    GXGetBitmap(bitShape, &x.bits, nil);
    if (x.bits.space == gxIndexedSpace)
        GXCloneColorSet(x.bits.set);
    if (x.bits.profile)
        GXCloneColorProfile(x.bits.profile);
    GXDisposeShape(bitShape);
    x.offset.x = bounds.left;
    x.offset.y = bounds.top;
    x.bits.width = FixedRound(bounds.right) -
                    FixedRound(bounds.left);
    x.bits.height = FixedRound(bounds.bottom) -
                    FixedRound(bounds.top);

    /* map the area offset back to local space, store in x */
    InvertMapping(&map, GXGetViewPortGlobalMapping(port, &map));
    MapPoints(&map, 1, &x.offset);
    return x;
}
```

The `GXGetShapeGlobalBounds` function is described on page 7-125.

The `GXGetShapeLocalBounds` function is described on page 7-96. The `GXGetShapeDeviceBounds` function is described on page 7-116.

View-Related Objects Reference

This section provides reference information to the structures and functions that allow you to create and manipulate view related objects and alter their properties. It includes

- descriptions of the constants and data types that are specific to view port, view device, and view group objects
- descriptions of the QuickDraw GX functions that operate on view port objects
- descriptions of the QuickDraw GX functions that operate on view device objects
- descriptions of the QuickDraw GX functions that operate on view group objects

Constants and Data Types

This section describes the data types that you use to obtain and provide information about view port, view device, and view group objects.

The View Port Object

QuickDraw GX provides you with access to an individual view port object through a `gxViewPort` reference:

```
typedef struct gxPrivateViewPortRecord *gxViewPort;
```

In this type definition, `gxViewPort` is a type-checked reference, not an actual pointer to any defined structure. The contents of the view port object are private.

The Halftone Structure

Halftones are described by the `gxHalftone` structure:

```
struct gxHalftone{
    Fixed          angle;
    Fixed          frequency;
    gxDotType      method;
    gxTintType      tinting;
    gxColor        dotColor;
    gxColor        backgroundColor;
    gxColorSpace   tintSpace;
};
```

View-Related Objects

Field descriptions

angle	The orientation of the rows of dots in the halftone pattern. It is a fixed-point number between 0.0 and 360.0 that describes an angle, in degrees, clockwise from horizontal.
frequency	The size of the cells, in terms of numbers of dots per inch. It can be any positive value.
method	The halftone pattern itself and how it is filled: the shapes of the dots, the pattern of their arrangement, and the way in which a dot fills its cell as it enlarges. The supported methods are defined in the <code>gxDotTypes</code> enumeration, described next.
tinting	The type of calculation by which the input color is to be approximated by a ratio of dot color and background color. Tint types are defined in the <code>gxTintTypes</code> enumeration, described on page 7-67.
dotColor	The color of the dots used to form the halftone.
backgroundColor	The color of the background used to form the halftone.
tintSpace	The color space the input color is converted to before the halftone calculations are made.

The halftone structure is described further in the section “Halftone” beginning on page 7-13.

Dot Types

The `gxDotTypes` enumeration defines the possible halftone dot types:

```
enum gxDotTypes{
    gxRoundDot = 1,
    gxSpiralDot,
    gxSquareDot,
    gxLineDot,
    gxEllipticDot,
    gxTriangleDot,
    gxDispersedDot
};

typedef long gxDotType;
```

The meaning of each dot type is evident from its name, except perhaps for `gxDispersedDot`. The `gxDispersedDot` dot type uses a seemingly random pattern of small dots that gradually fill up each cell as the tint value increases. For a visual representation of each of these dot types, see Figure 7-6 on page 7-16.

Tint Types

The `gxTintTypes` enumeration defines the possible ways of calculating the tint color (the color to be represented by a ratio of dot and background color) for a halftone.

```
enum gxTintTypes{
    gxNoTint,
    gxLuminanceTint,
    gxAverageTint,
    gxMixtureTint,
    gxComponent1Tint,
    gxComponent2Tint,
    gxComponent3Tint,
    gxComponent4Tint
};

typedef long gxTintType;
```

Constant descriptions

<code>gxNoTint</code>	No tint color. In a halftone structure with all fields set to 0, the tinting field has a value of <code>gxNoTint</code> .
<code>gxLuminanceTint</code>	The tint color is the input color's luminance.
<code>gxAverageTint</code>	The tint color is the average of all components of the input color.
<code>gxMixtureTint</code>	Project the input color onto the foreground-background color axis in tint color space. That projection point is the tint color.
<code>gxComponent1Tint</code>	Use only component 1 of the input color as the tint color.
<code>gxComponent2Tint</code>	Use only component 2 of the input color as the tint color.
<code>gxComponent3Tint</code>	Use only component 3 of the input color as the tint color.
<code>gxComponent4Tint</code>	Use only component 4 of the input color as the tint color.

For more information about halftone tints, see the section "Halftone" beginning on page 7-13.

View-Related Objects

View Port Attributes

The view port attributes are a set of flags that modify the behavior of the view port object. Constants for all recognized view port attributes are defined in the `gxPortAttributes` enumeration:

```
enum gxPortAttributes {
    gxGrayPort          = 0x0001,    /* convert to gray space */
    gxAlwaysGridPort    = 0x0002,    /* use gxDeviceGridStyle */
    gxEnableMatchPort   = 0x0004     /* perform color matching */
};

typedef long gxPortAttribute;
```

The individual view port attributes are described in Table 7-2 on page 7-20.

The View Device Object

QuickDraw GX provides you with access to an individual view device object through a `gxViewDevice` reference:

```
typedef struct gxPrivateViewDeviceRecord *gxViewDevice;
```

In this type definition, `gxViewDevice` is a type-checked reference, not an actual pointer to any defined structure. The contents of the view device object are private.

View Device Attributes

The view device attributes are a set of flags that modify the behavior of the view device object. Constants for all recognized view device attributes are defined in the `gxDeviceAttributes` enumeration:

```
enum gxDeviceAttributes{
    gxDirectDevice      = 0x01,    /* pixel image must be accessible */
    gxRemoteDevice      = 0x02,    /* pixel image may be on card */
    gxInactiveDevice    = 0x04     /* device is inactive */
};

typedef long gxDeviceAttribute;
```

The individual view device attributes are described in Table 7-3 on page 7-27.

The View Group Object

QuickDraw GX provides you with access to an individual view group object through a `gxViewGroup` reference:

```
typedef struct gxPrivateViewGroupRecord *gxViewGroup;
```

In this type definition, `gxViewGroup` is a type-checked reference, not an actual pointer to any defined structure. The contents of the view group object are private.

View Group Types

QuickDraw GX provides two predefined view group references for you, defined by the following constants:

```
#define gxAllViewDevices          ((gxViewGroup) 0)
#define gxScreenViewDevices      ((gxViewGroup) 1)
```

Constant descriptions

`gxAllViewDevices`

Not an actual reference, this constant represents all view groups, both offscreen and onscreen. You can use this constant when you want to use the `GXGetViewGroupViewPorts` function or the `GXGetViewGroupViewDevices` function to determine all the view ports or all the view devices for all view groups. You cannot use this constant to set a view port or view device.

`gxScreenViewDevices`

A reference to the view group that includes view device objects for all physical display devices. Only by drawing to view ports in this view group can you perform onscreen drawing. This is the one view group that QuickDraw GX provides for you.

View Port Functions

This section describes the QuickDraw GX functions you use with view port objects. Using the functions described here, you can

- create and manipulate view port objects
- manipulate view port object properties
- retrieve the view devices that intersect a view port
- retrieve the view ports that intersect a shape
- measure a shape in a view port

View-Related Objects

To associate a view port object directly with a QuickDraw GX transform object, use the `GXGetTransformViewPorts` and `GXSetTransformViewPorts` functions. To associate a view port object indirectly with a QuickDraw GX shape object, use the `GXGetShapeViewPorts` and `GXSetShapeViewPorts` functions. All four functions are described in the chapter “Transform Objects” in this book.

Creating and Manipulating View Port Objects

The functions described in this section allow you to create and manipulate view port objects. With the functions in this section, you can

- create and dispose of view ports
- copy view ports
- test view ports for equality

GXNewViewPort

You can use the `GXNewViewPort` function to create a new view port.

```
gxViewPort GXNewViewPort(gxViewGroup group);
```

`group` A reference to the view group in which to create the view port.

function result A reference to the newly created view port.

DESCRIPTION

The `GXNewViewPort` function creates a new view port with default properties and assigns it to the specified view group. All other properties are set to their default values:

- no parent view port or child view port
- a clip that is a full shape
- a mapping that is the identity mapping
- a dither level of 1
- no halftone
- no attributes set
- an empty tag list

To create a view port in the onscreen view group, pass the value `gxScreenViewDevices` for the `group` parameter. To obtain an offscreen view group reference to pass to this function, use the `GXNewViewGroup` function.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewViewPort` function creates a view port object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`invalid_viewGroup_reference`

SEE ALSO

For examples of the use of this function, see page 7-41, Listing 7-5 on page 7-47, and Listing 7-14 on page 7-63.

To dispose of a view port, use the `GXDisposeViewPort` function, described next. To dispose of all the view ports in a view group, use the `GXDisposeViewGroup` function, described on page 7-122.

The `gxScreenViewDevices` view group reference is described in the section “View Group Types” on page 7-69. The `GXNewViewGroup` function is described on page 7-122.

GXDisposeViewPort

You can use the `GXDisposeViewPort` function to delete a view port object.

```
void GXDisposeViewPort(gxViewPort target);
```

`target` A reference to the view port object to dispose of.

DESCRIPTION

The `GXDisposeViewPort` function disposes of the target view port. If the target view port is a parent, its children are removed from their position in the hierarchy and each is made the root of a new hierarchy.

SPECIAL CONSIDERATIONS

If the target view port is a parent associated with a window, its child view ports lose their association with the window.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

View-Related Objects

SEE ALSO

For an example of the use of this function, see page 7-41.

For information about view port hierarchies, see “Parent and Child View Ports” beginning on page 7-18.

GXCopyToViewPort

You can use the `GXCopyToViewPort` function to create a copy of an existing view port object.

```
gxViewPort GXCopyToViewPort(gxViewPort target, gxViewPort source);
```

target A reference to the view port to copy the source contents into. If you specify `nil` for this parameter, the `GXCopyToViewPort` function creates a new view port object.

source A reference to the view port whose contents you want to copy.

function result A reference to the copy (that is, the target view port) of the source view port.

DESCRIPTION

The `GXCopyToViewPort` function copies the contents of an existing view port object to another, or it creates a new view port object and copies the contents of an existing view port object to it. The function copies the clip, mapping, dither, halftone, attributes, and tag list (but not the parent or child view ports) of the view port object specified by the `source` parameter into the view port object specified by the `target` parameter. It clones, but does not copy, the tag objects in the tag list. The target view port is placed in the same view group as the source view port. The target view port is not associated with any window, whether or not the source view port is associated with one.

If you specify `nil` for the `target` parameter, the `GXCopyToViewPort` function creates a new view port object and copies the source properties into it.

You can use the `GXCopyToViewPort` function to create a copy of a view port object and then modify it without changing the original.

SPECIAL CONSIDERATIONS

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToViewPort` function creates a view port object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`invalid_viewPort_reference`
`viewPort_is_a_window` (debugging version)

SEE ALSO

For an example of the use of this function, see Listing 7-2 on page 7-44.

To create a new view port that has default values instead of being a copy of an existing view port, use the `GXNewViewPort` function, described on page 7-70.

To compare two view port objects for equality, use the `GXEqualViewPort` function, described in the next section.

GXEqualViewPort

You can use the `GXEqualViewPort` function to determine whether two view port objects are equal.

```
boolean GXEqualViewPort(gxViewPort one, gxViewPort two);
```

`one` A reference to a view port to test for equality.

`two` A reference to another view port to test for equality.

function result `true` if the view port specified by the `one` parameter is equal to the view port specified by the `two` parameter; `false` otherwise.

DESCRIPTION

The `GXEqualViewPort` function returns as its function result a Boolean value indicating whether the view port object specified by the `one` parameter is equal to the view port object specified by the `two` parameter.

For two view port objects to be equal, they must have identical mappings, clips, dithers, halftones, and attributes. They also must have the same parent view port, if any, and (therefore) be in the same view group. If one view port is attached to a window, the other view port must be attached to the same window. The tag lists or child view ports of the view ports need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

View-Related Objects

SEE ALSO

To make a copy of a view port object that is equal by the criteria of this function, use the `GXCopyToViewPort` function, described in the previous section.

Manipulating View Port Object Properties

The functions described in this section allow you to manipulate the properties of the view port object. With these functions you can

- get and set a view port's clip
- get and set a view port's mapping, and get its global mapping
- get and set a view port's dither and halftone, and get its halftone device angle
- get and set a view port's parent view port and list of child view ports
- get and set a view port's view group
- get and set a view port's attributes and tag list

GXGetViewPortClip

You can use the `GXGetViewPortClip` function to examine the clip property of a view port object.

```
gxShape GXGetViewPortClip(gxViewPort source);
```

source A reference to the view port whose clip you wish to examine.

function result A reference to a newly created shape object that is a copy of the source view port's clip.

DESCRIPTION

The `GXGetViewPortClip` function returns a shape object whose geometry defines the clip associated with the view port. The function returns `nil` if there is no clip. The clip shape is a copy of the view port's clip; changing this shape does not change the view port's clip.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetViewPortClip` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

out_of_memory
invalid_viewPort_reference

SEE ALSO

For an example of the use of this function, see Listing 7-3 on page 7-45.

To set the view port's clip, use the `GXSetViewPortClip` function, described next.

GXSetViewPortClip

You can use the `GXSetViewPortClip` function to set the clip property of a view port object.

```
void GXSetViewPortClip(gxViewPort target, gxShape clip);
```

target A reference to the view port whose clip you wish to set.

clip A reference to a shape object whose geometry describes the clip to be assigned.

DESCRIPTION

The `GXSetViewPortClip` function copies information from the shape object referenced by the `clip` parameter into the clip property of the view port object referenced by the `target` parameter. You can specify `nil` for the `clip` parameter, in which case this function sets the clip property of the target view port to a full clip. (A full clip indicates that QuickDraw GX should not apply view port clipping to shapes drawn to this view port.)

Although a filled rectangle shape is most typical for a view port clip, the new clip shape may be a geometric shape, a bitmap shape, or a glyph shape. It may not be a picture, text, or layout shape.

- If you specify a geometric shape, it must be in primitive form—that is, all the stylistic information about the shape must be incorporated into the shape's geometry—because this function copies only the geometry-related information from the shape you specify. It does not copy the information contained in the shape's style. You can convert a shape to its primitive form using the `GXPrimitiveShape` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*. You can also specify an empty or full shape for a clip.

View-Related Objects

- If you specify a bitmap shape, it must have a pixel size of 1 and its color profile reference must be `nil`. In the bitmap, pixel values of 0 obscure drawing; pixel values of 1 do not restrict visibility. The `GXSetViewPortClip` function copies the pixel image from the bitmap to the clip property of the target view port.
- If you specify a glyph shape, this function uses information from the glyph shape's style object as well as its style list to determine the size, form, and position of the glyph outlines; those outlines are then used to clip drawing. The style list cannot have `nil` entries. A style object referenced by the glyph shape cannot be complex—that is, it cannot have a cap, join, dash, pattern, text face, font variation, tag list, or any of the properties used only by layout shapes.

Because it is copied into the view port, changing the clip shape after calling `GXSetViewPortClip` does not affect the view port's clip.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>invalid_viewPort_reference</code>	
<code>colorProfile_must_be_nil</code>	(debugging version)
<code>bitmap_pixel_size_must_be_1</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)
<code>viewPort_is_a_window</code>	(debugging version)

Notices (debugging version)

`clip_already_set`
`tags_in_shape_ignored`

SEE ALSO

For examples of the use of this function, see Listing 7-4 on page 7-46 and Listing 7-5 on page 7-47.

For information about geometric shapes and bitmap shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For information about glyph shapes, see *Inside Macintosh: QuickDraw GX Typography*.

To retrieve a copy of the view port clip, use the `GXGetViewPortClip` function, described in the previous section.

GXGetViewPortMapping

You can use the `GXGetViewPortMapping` function to retrieve the mapping for a view port object.

```
gxMapping *GXGetViewPortMapping(gxViewPort source,
                                gxMapping *map);
```

source A reference to the view port whose mapping you wish to examine.

map A pointer to a mapping structure. On return, this mapping contains a copy of the information from the mapping property of the source view port.

function result A pointer to a copy of the mapping property of the source view port. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetViewPortMapping` function copies the mapping matrix information from the mapping property of the source view port object into the mapping structure pointed to by the `map` parameter. The function also returns as its function result a pointer to this mapping structure.

To make changes to the source view port's mapping property, you can alter the information returned by this function, and then use the `GXSetViewPortMapping` function to reassign the altered mapping to the source view port.

If the source view port is the root view port of a view port hierarchy, this function gives the same results as `GXGetViewPortGlobalMapping`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`
`parameter_is_nil` (debugging version)

SEE ALSO

For examples of the use of this function, see Listing 7-3 on page 7-45 and Listing 7-6 on page 7-48.

To set a view port's mapping, use the `GXSetViewPortMapping` function, described next.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXSetViewportMapping

You can use the `GXSetViewportMapping` function to assign a mapping to a view port object.

```
void GXSetViewportMapping(gxViewport target,
                        const gxMapping *map);
```

<code>target</code>	A reference to the view port object you want to assign the mapping to.
<code>map</code>	A pointer to a mapping structure containing the mapping matrix to assign to the target view port.

DESCRIPTION

The `GXSetViewportMapping` function copies information from the mapping structure pointed to by the `map` parameter into the mapping property of the target view port.

You can specify `nil` for the `map` parameter, in which case this function sets the mapping property of the target view port as follows:

- If the clip shape is a full shape, which specifies no clipping, the function sets the mapping property to the identity mapping.
- If a clip exists, the function sets the mapping's translation component to the upper-left corner of the clip. It sets the other components of the mapping to identity.

You can provide arbitrary values for the elements of the mapping structure pointed to by the `map` parameter, with one exception: the lower-right element of the matrix (element `[2][2]`) may not be 0.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>invalid_viewPort_reference</code>	
<code>viewPort_is_a_window</code>	(debugging version)

SEE ALSO

For examples of the use of this function, see Listing 7-3 on page 7-45 and Listing 7-5 on page 7-47.

To get a view port's mapping, use the `GXGetViewportMapping` function, described in the previous section.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetViewPortGlobalMapping

You can use the `GXGetViewPortGlobalMapping` function to examine the resultant mapping after concatenating the mapping properties of a view port object with all its parent view ports.

```
gxMapping *GXGetViewPortGlobalMapping(gxViewPort source,
                                       gxMapping *map);
```

source A reference to the view port whose global mapping you wish to examine.

map A pointer to a mapping. On return, this parameter contains the concatenation of all the mapping properties of this view port and the view ports from which it descends.

function result A pointer to a copy of the `map` parameter. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetViewPortGlobalMapping` function returns the global mapping of the source view port in its view group. This mapping is the result of concatenating the view port's mapping with that of its parent and that of its parent's parent, and so on; the concatenation ascends the view port's branch in the hierarchy, from its position to the root view port, inclusive.

This function is useful when you want to determine how a shape is mapped through the root view port into the view group; that is, what its position and dimensions are in global space.

If the source view port is the root view port of a view port hierarchy, this function gives the same results as `GXGetViewPortMapping`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-11 on page 7-57.

To get a view port's mapping without concatenating it with the mappings of parent view ports, use the `GXGetViewPortMapping` function, described on page 7-77.

For a discussion of coordinate systems, global space, and view port hierarchies, see the section "Global Space" beginning on page 7-34.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetViewPortDither

You can use the `GXGetViewPortDither` function to examine the dither level of a view port object.

```
long GXGetViewPortDither(gxViewPort source);
```

`source` A reference to the view port whose dither level you wish to examine.

function result The view port's dither level.

DESCRIPTION

The `GXGetViewPortDither` function returns the view port dither level, which is a value between 0 and 16, inclusive. The values 0 and 1 both mean do not dither.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-13 on page 7-61.

To set the dither level, use the `GXSetViewPortDither` function, described next.

For information about the dither property, see “Dither” beginning on page 7-10.

GXSetViewPortDither

You can use the `GXSetViewPortDither` function to assign a dither level to a view port object.

```
void GXSetViewPortDither(gxViewPort target, long level);
```

`target` A reference to the view port whose dither level you wish to set.

`level` The new dither level.

DESCRIPTION

The `GXSetViewPortDither` function specifies the default dither level for the target view port. You can specify a level in the range of 0 to 16, inclusive. Levels 0 and 1 specify no dithering, otherwise the level specifies the maximum number of pixels in the dither pattern.

SPECIAL CONSIDERATIONS

You can set the ink object's attribute, `gxSuppressDitherInk`, if you want to ignore the view port's dither level for shapes drawn with a specific ink. For more information about ink object attributes, see the chapter "Ink Objects" in this book.

Dithering does not occur on 32 bit-per-pixel devices, regardless of the dither level.

SPECIAL CONSIDERATIONS

Version 1.0 of QuickDraw GX does not guarantee useful results for dither levels greater than 4.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`dither_already_set`

SEE ALSO

For examples of the use of this function, see Listing 7-1 on page 7-42 and Listing 7-13 on page 7-61.

To get a view port's dither level, use the `GXGetViewPortDither` function, described in the previous section.

For information about the dither property, see "Dither" beginning on page 7-10.

GXGetViewPortHalftone

You can use the `GXGetViewPortHalftone` function to examine the halftone property of a view port object.

```
boolean GXGetViewPortHalftone(gxViewPort source,
                              gxHalftone *data);
```

source A reference to the view port whose halftone you wish to examine.

data A pointer to a halftone structure. On return, the structure contains the view port's halftone information.

function result true if the halftone exists; otherwise false.

View-Related Objects

DESCRIPTION

The `GXGetViewPortHalftone` function copies the view port's halftone into the structure you provide, pointed to by the `data` parameter. If a halftone does not exist, the contents of the structure pointed to by the `data` parameter are not changed.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewPort_reference`

SEE ALSO

For information about the halftone property, see “Halftone” beginning on page 7-13.

To set the halftone of a view port, use the `GXSetViewPortHalftone` function, described next.

GXSetViewPortHalftone

You can use the `GXSetViewPortHalftone` function to assign a halftone property to a view port object.

```
void GXSetViewPortHalftone(gxViewPort target,
                           const gxHalftone *data);
```

<code>target</code>	A reference to the view port whose halftone you wish to change.
<code>data</code>	A pointer to a halftone structure containing the data with which to set the view port's halftone property.

DESCRIPTION

The `GXSetViewPortHalftone` function sets the halftone for the target view port. If the `data` parameter is set to `nil`, halftones are not used when drawing to this view port.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewPort_reference</code>	
<code>frequency_parameter_out_of_range</code>	(debugging version)
<code>tinting_parameter_out_of_range</code>	(debugging version)
<code>method_parameter_out_of_range</code>	(debugging version)
<code>space_may_not_be_indexed</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)

Notices (debugging version)

`halftone_already_set`

SEE ALSO

For an example of the use of this function, see Listing 7-1 on page 7-42.

For information about the halftone property, see “Halftone” beginning on page 7-13.

To set the halftone of a view port, use the `GXGetViewPortHalftone` function, described in the previous section.

GXGetHalftoneDeviceAngle

You can use the `GXGetHalftoneDeviceAngle` function to determine the actual angle a halftone is drawn with on a particular view device.

```
Fixed GXGetHalftoneDeviceAngle(gxViewDevice source,
                               const gxHalftone *data);
```

<code>source</code>	A reference to the view device whose halftone angle you wish to examine.
<code>data</code>	A pointer to a halftone structure that specifies the characteristics of a halftone.

function result The halftone angle as it would be drawn on the view device.

DESCRIPTION

The `GXGetHalftoneDeviceAngle` function returns the actual angle that the specified halftone would be drawn with on the source view device. The contents of the halftone structure pointed to by the `data` parameter are not changed.

The halftone angle on the view device may be different from the one you set with the `GXSetViewPortHalftone` function because the view device’s resolution interacts with the halftone frequency, which specifies the dot size in cells per inch, and the view device’s grid pattern.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

SEE ALSO

For information about the halftone property, see “Halftone” beginning on page 7-13.

To get the halftone of a view port, use the `GXGetViewPortHalftone` function, described on page 7-81.

To obtain a list of view devices that intersect a view port, use the `GXGetViewPortViewDevices` function, described on page 7-94.

GXGetViewPortParent

You can use the `GXGetViewPortParent` function to retrieve the parent view port of a view port object.

```
gxViewPort GXGetViewPortParent(gxViewPort source);
```

`source` A reference to the view port whose parent you want to examine.

function result A reference to the source view port's parent view port.

DESCRIPTION

The `GXGetViewPortParent` function returns the parent view port of the source view port, or `nil` if the view port has no parent (that is, if it is the root view port in a hierarchy).

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For information about view port hierarchies and the parent view port property, see “Parent and Child View Ports” beginning on page 7-18.

To set the parent of a view port, use the `GXSetViewPortParent` function, described next.

To get the child view port list of a view port, use the `GXGetViewPortChildren` function, described on page 7-86. To set the child view port list of a view port, use the `GXSetViewPortChildren` function, described on page 7-87.

GXSetViewPortParent

You can use the `GXSetViewPortParent` function to assign a parent view port to a view port object.

```
void GXSetViewPortParent(gxViewPort target, gxViewPort parent);
```

`target` A reference to the view port whose parent you want to set.

`parent` A reference to the target view port's new parent.

DESCRIPTION

The `GXSetViewPortParent` function replaces the target's parent view port with the view port specified in the `parent` parameter. It also adds the target view port to the parent's list of child view ports. If the target view port is in a different view group from the new parent view port, the target view port is reassigned to the parent's view group, which also causes the target's child view ports to be assigned to the new parent's view group as well.

If you set the `parent` parameter to `nil`, this function sets the target view port to have no parent; the target view port then becomes the root of a new view port hierarchy.

SPECIAL CONSIDERATIONS

View port hierarchies cannot contain circular references; that is, a view port cannot have itself or any of its descendents as its parent.

You cannot assign a parent view port to a window view port (one attached to a Macintosh window).

The view ports in a hierarchy must all be in the same view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewPort_reference</code>	
<code>viewPort_is_a_window</code>	(debugging version)
<code>viewPort_cannot_contain_itself</code>	(debugging version)

SEE ALSO

For an example of the use of this function, see Listing 7-5 on page 7-47.

For information about view port hierarchies and the parent view port property, see "Parent and Child View Ports" beginning on page 7-18.

To get the parent of a view port, use the `GXGetViewPortParent` function, described in the previous section.

To get the child view port list of a view port, use the `GXGetViewPortChildren` function, described next. To set the child view port list of a view port, use the `GXSetViewPortChildren` function, described on page 7-87.

GXGetViewPortChildren

You can use the `GXGetViewPortChildren` function to retrieve the list of child view ports of a view port object.

```
long GXGetViewPortChildren(gxViewPort source, gxViewPort list[]);
```

`source` A reference to the view port whose child view ports you wish to examine.

`list` An array of view port references. On return, contains the child view port list of the source view port.

function result The number of references in the child view port list of the source view port.

DESCRIPTION

The `GXGetViewPortChildren` function retrieves the list of child view ports of the source view port. It also returns, as its function result, the number of references in the list. The list and the number returned by this function do not include children of children.

The view ports are placed in the list array in the order they were added as children.

If you set the `list` parameter to `nil`, `GXGetViewPortChildren` does not retrieve a list of references; it only returns the number of child view port objects. Therefore, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For information about view port hierarchies and the child view port list property, see “Parent and Child View Ports” beginning on page 7-18.

To set the child view port list of a view port, use the `GXSetViewPortChildren` function, described next.

To get the parent of a view port, use the `GXGetViewPortParent` function, described on page 7-84. To set the parent of a view port, use the `GXSetViewPortParent` function, described in the previous section.

GXSetViewportChildren

You can use the `GXSetViewportChildren` function to assign a list of child view ports to a view port object.

```
void GXSetViewportChildren(gxViewport target, long count,
                          const gxViewport list[]);
```

<code>target</code>	A reference to the view port whose children you wish to set.
<code>count</code>	The number of references in the child view port list.
<code>list</code>	The array of view port references that is the new child view port list.

DESCRIPTION

The `GXSetViewportChildren` function sets each of the view ports specified in the `list` parameter to have the target view port as its parent, and assigns the list as the child view port list of the target view port. Previous children of the target view port are assigned to have no parent.

If the `target` parameter is set to `nil`, each view port in the list is assigned to have no parent. If a child view port in the list is in a different view group than the target view port's view group, the child view port is changed to be in the view group of the target view port.

SPECIAL CONSIDERATIONS

View port hierarchies cannot contain circular references; that is, a view port cannot have itself or any of its ancestors as its child.

The view ports in a hierarchy must all be in the same view group.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>invalid_viewPort_reference</code>	
<code>viewPort_cannot_contain_itself</code>	(debugging version)

SEE ALSO

For information about view port hierarchies and the child view port list property, see “Parent and Child View Ports” beginning on page 7-18.

To get the child view port list of a view port, use the `GXGetViewportChildren` function, described in the previous section.

To get the parent of a view port, use the `GXGetViewportParent` function, described on page 7-84. To set the parent of a view port, use the `GXSetViewportParent` function, described on page 7-84.

GXGetViewPortViewGroup

You can use the `GXGetViewPortViewGroup` function to determine the view group that a view port is part of.

```
gxViewGroup GXGetViewPortViewGroup(gxViewPort source);
```

`source` A reference to the view port whose view group you wish to examine.

function result A reference to the view group that the source view port is part of.

DESCRIPTION

The `GXGetViewPortViewGroup` returns a reference to the source view port's view group. If it is the onscreen view group, the returned value is `gxScreenViewDevices`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

To set a view port's view group, use the `GXSetViewPortViewGroup` function, described next.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

GXSetViewPortViewGroup

You can use the `GXSetViewPortViewGroup` function to assign a view port object to a new view group.

```
void GXSetViewPortViewGroup(gxViewPort target, gxViewGroup group);
```

`target` A reference to the view port whose view group you wish to change.

`group` A reference to the view group to which the view port is to be assigned.

DESCRIPTION

The `GXSetViewportViewGroup` function assigns the target view port to the specified view group. Child view ports of the target view port are also assigned to that view group.

To assign a view port to the onscreen view group, pass the value `gxScreenViewDevices` for the group parameter. To obtain an offscreen view group reference to pass to this function, use the `GXNewViewGroup` function.

SPECIAL CONSIDERATIONS

The view ports in a hierarchy must all be in the same view group.

You cannot change the view group of a window view port (one attached to a Macintosh window).

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewPort_reference</code>	
<code>invalid_viewGroup_reference</code>	
<code>viewPort_is_a_window</code>	(debugging version)

Notices (debugging version)

<code>viewPort_already_in_viewGroup</code>
--

SEE ALSO

For an example of the use of this function, see Listing 7-2 on page 7-44.

To get a view port's view group, use the `GXGetViewportViewGroup` function, described in the previous section.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

The `GXNewViewGroup` function is described on page 7-122.

GXGetViewportAttributes

You can use the `GXGetViewportAttributes` function to examine which attributes of a view port object are set.

```
gxPortAttribute GXGetViewportAttributes(gxViewport source);
```

`source` A reference to the view port whose attributes you wish to examine.

function result The view port attributes of the source view port.

View-Related Objects

ERRORS, WARNINGS, AND NOTICES

Errors

invalid_viewPort_reference

SEE ALSO

View port attributes are described in the section “View Port Attributes” on page 7-20.

To set a view port’s attributes, use the `GXSetViewPortAttributes` function, described next.

GXSetViewPortAttributes

You can use the `GXSetViewPortAttributes` function to set or clear the attributes of a view port object.

```
void GXSetViewPortAttributes(gxViewPort target,
                             gxPortAttribute attributes);
```

`target` A reference to the view port whose attributes you wish to set.

`attributes` The new view port attributes to be assigned.

DESCRIPTION

The `GXSetViewPortAttributes` function sets the attributes of the view port object referenced in the `target` parameter to those specified in the `attributes` parameter. If you pass `gxNoAttributes` for the `attributes` parameter, all attributes are cleared.

ERRORS, WARNINGS, AND NOTICES

Errors

invalid_viewPort_reference

parameter_out_of_range (debugging version)

Notices (debugging version)

attributes_already_set

SEE ALSO

For an example of the use of this function, see Listing 7-1 on page 7-42.

View port attributes are described in the section “View Port Attributes” on page 7-20.

To examine a view port’s attributes, use the `GXGetViewPortAttributes` function, described in the previous section.

GXGetViewPortTags

You can use the `GXGetViewPortTags` function to examine one or more of the tag objects associated with a view port object.

```
long GXGetViewPortTags(gxViewPort source, long tagType,
                       long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the view port whose tag list you want to examine.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetViewPortTags` function searches the tag list of the source view port object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetViewPortTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetViewPortTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetViewPortTags` function returns in it the tag references that were found.

Typically, you call this function once with a `nil` value for the `items` parameter to determine the number of matching tag references. Then you allocate an appropriately sized array and call the function a second time to obtain the references themselves.

View-Related Objects

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`invalid_viewPort_reference`
`index_is_less_than_one` (debugging version)
`count_is_less_than_one` (debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a view port, use the `GXSetViewPortTags` function, described next.

GXSetViewPortTags

You can use the `GXSetViewPortTags` function to add, remove, or replace tag objects associated with a view port object.

```
void GXSetViewPortTags(gxViewPort target, long tagType,
                       long index, long oldCount,
                       long newCount, const gxTag items[]);
```

<code>target</code>	A reference to the view port whose tag list you want to alter.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source shape’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

DESCRIPTION

The `GXSetViewPortTags` function allows you add tag references to a view port object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the view port object to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewPort_reference</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

Notices (debugging version)

`tag_already_set`

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a view port, use the `GXGetViewPortTags` function, described in the previous section.

Retrieving the View Devices That Intersect a View Port

The function described in this section allows you to determine the view devices that a view port can draw to.

GXGetViewPortViewDevices

You can use the `GXGetViewPortViewDevices` function to determine all of the view device objects that shapes drawn to a view port can display on.

```
long GXGetViewPortViewDevices(gxViewPort source,
                              gxViewDevice list[]);
```

source A reference to the view port whose view devices you wish to examine.

list An array of view device references. On return, contains the references to the view devices that the source view port can draw to.

function result The number of view device references in the `list` array.

DESCRIPTION

The `GXGetViewPortViewDevices` function determines which view devices can display the contents of the source view port, and places a list of references to those view devices in the `list` parameter. It also returns the number of view devices in the list. The view devices returned are those, in the same view group as the view port, whose clip areas intersect the view port's clip area.

If you set the `list` parameter to `nil`, `GXGetViewPortViewDevices` does not return a list of references; it only returns the number of view device references that would be in the list. Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewPort_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-7 on page 7-49.

Retrieving the View Ports That Intersect a Shape

The function described in this section allows you to determine which view ports can display a shape object.

GXGetShapeGlobalViewPorts

You can use the `GXGetShapeGlobalViewPorts` function to determine all of the view ports that intersect the area of a shape.

```
long GXGetShapeGlobalViewPorts(gxShape source, gxViewport list[]);
```

source A reference to the shape whose view ports you wish to examine.

list An array of view port references. On return, contains the references to the view ports that the shape can draw to.

function result The number of view port references in the `list` array.

DESCRIPTION

The `GXGetShapeGlobalViewPorts` function retrieves a list of the view ports that the source shape may draw to. If a view port is specified in the transform object associated with the shape, and if the transformed shape is not completely clipped by the view port's clip shape, the view port is put in the list returned by this function. The view ports need not all be in the same view group.

If you set the `list` parameter to `nil`, `GXGetShapeGlobalViewPorts` does not fill out the list of references; it only returns the number of view port references that would be in the list (which may be 0). Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

As one application of this function, you could first call `GXGetShapeGlobalViewPorts` to determine the view ports to which a shape is drawn. You then could use the view ports in calls to `GXGetShapeGlobalViewDevices`, to determine the view devices a given shape would be drawn to.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`

SEE ALSO

The `GXGetShapeGlobalViewDevices` function is described on page 7-115.

Measuring a Shape in Local Coordinates

The function described in this section allows you to measure the size of a shape as it appears in its view ports—in local coordinates, after its transform mapping has been applied. Other QuickDraw GX functions are available to measure shapes in other contexts:

- To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function, described on page 7-125.
- To determine a shape's bounding rectangle on a view device, use the `GXGetShapeDeviceBounds` function, described on page 7-116.
- To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetShapeLocalBounds

You can use the `GXGetShapeLocalBounds` function to determine the bounding rectangle of a shape in local coordinates.

```
gxRectangle *GXGetShapeLocalBounds(gxShape source,
                                   gxRectangle *bounds);
```

source A reference to the shape whose bounding rectangle you wish to determine in local coordinates.

bounds A pointer to a rectangle structure. On return, it contains the dimensions of the bounding rectangle.

function result A rectangle that defines the bounds of the shape in local coordinates. (It is the same as the rectangle returned in the `bounds` parameter.)

DESCRIPTION

The `GXGetShapeLocalBounds` function returns the bounding rectangle of the source shape after the shape's transform mapping and style have been applied. The dimensions of the rectangle are in the shape's local coordinates. The rectangle pointed to by the `bounds` parameter also receives the bounding rectangle in local coordinates.

View-Related Objects

To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function. To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function. To determine a shape's bounding rectangle on a view device, use the `GXGetShapeDeviceBounds` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`parameter_is_nil` (debugging version)

SEE ALSO

For an example of the use of this function, see Listing 7-8 on page 7-51.

The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeGlobalBounds` function is described on page 7-125. The `GXGetShapeDeviceBounds` function is described on page 7-116.

For information about coordinate spaces, see the section “About Drawing, Coordinate Conversion, and Clipping” beginning on page 7-30.

View Device Functions

This section describes the QuickDraw GX functions you use with view device objects. Using the functions described here, you can

- create and manipulate view device objects and their properties
- retrieve the view devices that intersect a shape
- measure and analyze a shape on a device, including hit-testing a shape on a device

Creating and Manipulating View Device Objects

The functions described in this section allow you to create and manipulate view device objects. With the functions in this section, you can

- create and dispose of view devices
- copy view devices
- test view devices for equality

GXNewViewDevice

You can use the `GXNewViewDevice` function to create a new view device object.

```
gxViewDevice GXNewViewDevice (gxViewGroup group,
                               gxShape bitmapShape);
```

`group` A reference to the view group in which to create the view device.

`bitmapShape` A reference to a bitmap shape that defines the view device's imaging area.

function result A reference to the newly created view device object.

DESCRIPTION

The `GXNewViewDevice` function creates a new view device object in the specified view group. The `bitmapShape` parameter references a bitmap shape whose bitmap structure specifies the height, width, and pixel depth (bits per pixel) of the device, plus any color set or color profile used by the device. The remaining properties have default values:

- a clip that is a full shape
- a mapping that is the identity mapping
- no attributes set
- an empty tag list

To obtain an offscreen view group reference to pass to this function, use the `GXNewViewGroup` function. To create a view device in the onscreen view group, pass the value `gxScreenViewDevices` for the `group` parameter.

SPECIAL CONSIDERATIONS

The bitmap shape that you pass to this function cannot not have a disk-based pixel image.

If no error occurs, the `GXNewViewDevice` function creates a view device object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>invalid_viewGroup_reference</code>	
<code>illegal_type_for_shape</code>	(debugging version)

SEE ALSO

For examples of the use of this function, see Listing 7-9 on page 7-53 and Listing 7-14 on page 7-63.

To dispose of a view device, use the `GXDisposeViewDevice` function, described next. To dispose of all the view devices in a view group, use the `GXDisposeViewGroup` function, described on page 7-122.

For information about bitmap shapes and the bitmap structure, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

The `GXNewViewGroup` function is described on page 7-122. The `gxScreenViewDevices` view group reference is described in the section “View Group Types” on page 7-69.

GXDisposeViewDevice

You can use the `GXDisposeViewDevice` function to delete a view device object.

```
void GXDisposeViewDevice(gxViewDevice target);
```

`target` A reference to the view device.

DESCRIPTION

The `GXDisposeViewDevice` function disposes of the view device and all associated memory structures, including the view device’s clip shape, bitmap, color set, and color profile.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

SEE ALSO

For an example of the use of this function, see page 7-53.

To dispose of all the view devices in a view group, use the `GXDisposeViewGroup` function, described on page 7-122.

GXCopyToViewDevice

You can use the `GXCopyToViewDevice` function to create a copy of an existing view device object.

```
gxViewDevice GXCopyToViewDevice (gxViewDevice target,  
                                gxViewDevice source);
```

target A reference to the view device to copy the source contents into. If you specify `nil` for this parameter, the `GXCopyToViewDevice` function creates a new view device object.

source A reference to the view device whose contents you want to copy.

function result A reference to the view device that is a copy of the source view device.

DESCRIPTION

The `GXCopyToViewDevice` function copies the contents of an existing view device object to another, or it creates a new view device object and copies the contents of an existing view device object to it. The function copies the clip, mapping, bitmap shape (including color space, color profile reference, and color set reference), attributes, and tag list from the source view device. It clones, but does not copy, the tag objects in the tag list.

In copying the bitmap, the `GXCopyToViewDevice` function does not copy the pixel image itself, just the properties of the bitmap shape. For the color set and color profile properties of the bitmap, and tag objects, `GXCopyToViewDevice` copies only the references, not the objects themselves.

If you specify `nil` for the `target` parameter, the `GXCopyToViewDevice` function creates a new view port object and copies the properties of the source view device into it.

SPECIAL CONSIDERATIONS

If you attempt to copy to a screen device, this function posts a `viewDevice_access_restricted` error.

If you specify `nil` for the `target` parameter and no error occurs, the `GXCopyToViewDevice` function creates a view device object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

`invalid_viewDevice_reference`

`viewDevice_access_restricted` (debugging version)

SEE ALSO

To create a new view device that has default properties instead of being a copy of an existing view device, use the `GXNewViewDevice` function, described on page 7-98.

To compare two view device objects for equality, use the `GXEqualViewDevice` function, described next.

GXEqualViewDevice

You can use the `GXEqualViewDevice` function to determine whether two view device objects are equal.

```
boolean GXEqualViewDevice(gxViewDevice one, gxViewDevice two);
```

`one` A reference to one view device to test for equality.

`two` A reference to another view device to test for equality.

function result `true` if the view device specified by the `one` parameter is equal to the view device specified by the `two` parameter; `false` otherwise.

DESCRIPTION

The `GXEqualViewDevice` function returns as its function result a Boolean value indicating whether the view device object specified by the `one` parameter is equal to the view device object specified by the `two` parameter.

For two view device objects to be equal, they must have identical clips, mappings, bitmap shapes, and attributes. They also must be in the same view group, represent the same Macintosh graphics device (same `GDevice` record), and point to the same pixel image, color set, and color profile. The view device objects are not equal if they point to different but equivalent pixel images, color sets, or color profiles. The tag lists of the view devices need not be identical.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewDevice_reference`

View-Related Objects

SEE ALSO

To make a copy of a view device object that is equal by the criteria of this function, use the `GXCopyToViewDevice` function, described in the previous section.

Macintosh graphics devices and the `GDevice` record are described in *Inside Macintosh: Imaging With QuickDraw*. The relationship of view devices to `GDevice` records is discussed in the Macintosh environment chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Manipulating View Device Object Properties

The functions described in this section allow you to create and manipulate view device objects. With these functions, you can get and set a view device's

- clip
- mapping
- bitmap
- view group
- attributes
- tag list

GXGetViewDeviceClip

You can use the `GXGetViewDeviceClip` function to examine the clip property of a view device object.

```
gxShape GXGetViewDeviceClip(gxViewDevice source);
```

source A reference to the view device whose clip you wish to examine.

function result A reference to a shape object whose geometry defines the view device's clip.

DESCRIPTION

The `GXGetViewDeviceClip` function returns a shape object that defines the geometry of the clip associated with the view device. The function returns `nil` if there is no clip. The clip shape is a copy of the view device's clip; changing this shape does not change the view device's clip.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetViewDeviceClip` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`invalid_viewDevice_reference`

SEE ALSO

To set a view device's clip, use the `GXSetViewDeviceClip` function, described next.

GXSetViewDeviceClip

You can use the `GXSetViewDeviceClip` function to set the clip property of a view device object.

```
void GXSetViewDeviceClip(gxViewDevice target, gxShape clip);
```

`target` A reference to the view device whose clip you wish to set.
`clip` A reference to a shape object whose geometry describes the clip to be assigned.

DESCRIPTION

The `GXSetViewDeviceClip` function copies information from the shape object referenced by the `clip` parameter into the clip property of the view device object referenced by the `target` parameter. You can specify `nil` for the `clip` parameter, in which case this function sets the clip property of the target view device to a full clip. (A full clip indicates that QuickDraw GX is not to apply view device clipping to shapes that reference this view device.)

Although a filled rectangle is the most common clip shape for a view device, the new clip shape may be a geometric shape, a bitmap shape, or a glyph shape. It may not be a picture, text, or layout shape.

- If you specify a geometric shape, it must be in primitive form—that is, all the stylistic information about the shape must be incorporated into the shape's geometry—because this function copies only the geometry-related information from the shape you specify. It does not copy the information contained in the shape's style. You can convert a shape to its primitive form using the `GXPrimitiveShape` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*. You can also specify an empty or full shape for a clip.

View-Related Objects

- If you specify a bitmap shape, it must have a pixel size of 1 and its color profile reference must be `nil`. In the bitmap, pixel values of 0 obscure drawing; pixel values of 1 do not restrict visibility. The `GXSetViewDeviceClip` function copies the pixel image from the bitmap to the clip property of the target view device.
- If you specify a glyph shape, this function uses information from the glyph shape's style object as well as its style list to determine the size, form, and position of the glyph outlines; those outlines are then used to clip drawing. The style list cannot have `nil` entries. A style object referenced by the glyph shape cannot be complex—that is, it cannot have a cap, join, dash, pattern, text face, font variation, tag list, or any of the properties used only by layout shapes.

You only need to call this function if you want to restrict the part of the device that displays your view ports—for example, to clip out the area reserved for live video in your offscreen copy of an onscreen view device.

Because it is copied into the view device object, changing the clip shape after calling `GXSetViewDeviceClip` does not affect the view device's clip.

SPECIAL CONSIDERATIONS

You cannot change the clip of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>colorProfile_must_be_nil</code>	(debugging version)
<code>bitmap_pixel_size_must_be_1</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)
<code>viewDevice_access_restricted</code>	(debugging version)

Notices (debugging version)

`clip_already_set`
`tags_in_shape_ignored`

SEE ALSO

For information about geometric shapes and bitmap shapes, see *Inside Macintosh: QuickDraw GX Graphics*. For information about glyph shapes, see *Inside Macintosh: QuickDraw GX Typography*.

To retrieve a copy of the clip property, use the `GXGetViewDeviceClip` function, described in the previous section.

GXGetViewDeviceMapping

You can use the `GXGetViewDeviceMapping` function to examine the mapping property of a view device object.

```
gxMapping *GXGetViewDeviceMapping(gxViewDevice source,  
                                   gxMapping *map);
```

source	A reference to the view device whose mapping you wish to examine.
--------	---

map	A pointer to a mapping structure. On return, the structure contains a copy of the mapping matrix of the source view device.
-----	---

function result A pointer to the mapping matrix of the source view device. (This value is the same as the value returned in the `map` parameter.)

DESCRIPTION

The `GXGetViewDeviceMapping` function copies the mapping matrix information from the mapping property of the source view device object into the mapping structure pointed to by the `map` parameter. The function also returns as its function result a pointer to this mapping structure.

To make changes to the source view device's mapping property, you can alter the information returned by this function, and then use the `GXSetViewDeviceMapping` function to reassign the altered mapping to the source view device.

ERRORS, WARNINGS, AND NOTICES

Errors

```
invalid_viewDevice_reference
parameter_is_nil           (debugging version)
```

SEE ALSO

For an example of the use of this function, see Listing 7-11 on page 7-57.

To set a view device's mapping, use the `GXSetViewDeviceMapping` function, described next.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXSetViewDeviceMapping

You can use the `GXSetViewDeviceMapping` function to assign a mapping to a view device object.

```
void GXSetViewDeviceMapping(gxViewDevice target,
                           const gxMapping *map);
```

<code>target</code>	A reference to the view device object you want to assign the mapping to.
<code>map</code>	A pointer to a mapping structure containing the mapping matrix to assign to the target view device.

DESCRIPTION

The `GXSetViewDeviceMapping` function copies the mapping structure pointed to by the `map` parameter into the mapping property of the target view device.

You can specify `nil` for the `map` parameter. If you do, the mapping is set to a translation that prevents the view device from overlapping any other view device in the view device's view group. This translation may cause the view device to move to an area adjacent to, but not overlapping, other view devices in this view group.

You can provide arbitrary values for the elements of the mapping structure pointed to by the `map` parameter, with one exception: the lower-right element of this matrix (element `[2][2]`) may not be 0.

SPECIAL CONSIDERATIONS

You cannot change the mapping of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES

Errors

<code>invalid_viewDevice_reference</code>	
<code>viewDevice_access_restricted</code>	(debugging version)

SEE ALSO

For an example of the use of this function, see page 7-57.

To retrieve a view device's mapping, use the `GXGetViewDeviceMapping` function, described in the previous section.

For information about the `gxMapping` structure, see the mathematics chapter of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

GXGetViewDeviceBitmap

You can use the `GXGetViewDeviceBitmap` function to retrieve the bitmap property of a view device object.

```
gxShape GXGetViewDeviceBitmap(gxViewDevice source);
```

`source` A reference to the view device whose bitmap you wish to examine.

function result A bitmap shape object that is a copy of the information in the view device's bitmap property.

DESCRIPTION

The `GXGetViewDeviceBitmap` function returns a bitmap shape that is a copy of the bitmap representing the imaging area of the specified device. The pixel image pointer in the bitmap structure of the bitmap shape may be `nil` if, for example, the image is on disk. The pointer is not `nil` if the image is associated with an onscreen device or was supplied by an application.

The shape returned by this function is a copy of the device's bitmap, so if you alter it you must reassign it to the view device with the `GXSetViewDeviceBitmap` function.

However, the pixel image of the bitmap shape returned by this function is *not* a copy; it is the same pixel image as that used by the view device. Thus if you draw to the view device you modify the pixel image of the returned shape, and likewise if you modify the pixel image of the returned shape you modify the pixel image of the view device. You can take advantage of this fact to draw an offscreen bitmap directly to an onscreen view device.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXGetViewDeviceBitmap` function creates a shape object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`invalid_viewDevice_reference`

SEE ALSO

For examples of the use of this function, see page 7-55 and Listing 7-14 on page 7-63.

To set a view device's bitmap property, use the `GXSetViewDeviceBitmap` function, described next.

For information about bitmap shapes, the bitmap structure, and pixel images, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXSetViewDeviceBitmap

You can use the `GXSetViewDeviceBitmap` function to set the bitmap property of a view device object.

```
void GXSetViewDeviceBitmap(gxViewDevice target,
                           gxShape bitmapShape);
```

`target` A reference to the view device whose bitmap you wish to set.

`bitmapShape` A reference to a bitmap shape object that specifies the new bitmap.

DESCRIPTION

The `GXSetViewDeviceBitmap` function sets the bitmap property in the view device to contain the information in the shape specified in the `bitmapShape` parameter. Only the bitmap structure in the geometry of the bitmap shape is copied into the view device.

SPECIAL CONSIDERATIONS

The bitmap shape you supply to this function cannot have a disk-based pixel image.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>viewDevice_access_restricted</code>	(debugging version)

SEE ALSO

For an example of the use of this function, see page 7-55.

To retrieve the bitmap property of a view device, use the `GXGetViewDeviceBitmap` function, described in the previous section.

For information about bitmap shapes, see the bitmap shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetViewDeviceViewGroup

You can use the `GXGetViewDeviceViewGroup` function to determine the view group that a view device is part of.

```
gxViewGroup GXGetViewDeviceViewGroup(gxViewDevice source);
```

`source` A reference to the view device whose view group you wish to examine.

function result A reference to the view group that the source view device is part of.

DESCRIPTION

The `GXGetViewDeviceViewGroup` function returns a reference to the source view device's view group. If it is the onscreen view group, the returned value is `gxScreenViewDevices`.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewDevice_reference`

SEE ALSO

To set a view device's view group, use the `GXSetViewDeviceViewGroup` function, described next.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

GXSetViewDeviceViewGroup

You can use the `GXSetViewDeviceViewGroup` function to assign a view device object to a specified view group.

```
void GXSetViewDeviceViewGroup(gxViewDevice target,  
                               gxViewGroup group);
```

`target` A reference to the view device whose view group you wish to change.

`group` A reference to the view group to which the view device is to be assigned.

View-Related Objects

DESCRIPTION

The `GXSetViewDeviceViewGroup` function changes the target view device to the specified view group.

SPECIAL CONSIDERATIONS

You cannot assign a view device to the onscreen view group; do not specify `gxScreenViewDevices` for the group parameter. Also, you cannot change the view group of a view device already in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewDevice_reference</code>	
<code>invalid_viewGroup_reference</code>	
<code>viewDevice_access_restricted</code>	(debugging version)

Notices (debugging version)

<code>viewDevice_already_in_viewGroup</code>
--

SEE ALSO

For an example of the use of this function, see Listing 7-10 on page 7-54.

To get a view device's view group, use the `GXGetViewDeviceViewGroup` function, described in the previous section.

The `gxScreenViewDevices` view group reference is described in the section "View Group Types" on page 7-69.

GXGetViewDeviceAttributes

You can use the `GXGetViewDeviceAttributes` function to determine which attributes of a view device object are set.

```
gxDeviceAttribute GXGetViewDeviceAttributes(gxViewDevice source);
```

source A reference to the view device whose attributes you wish to examine.

function result The view device attributes of the source view device.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewDevice_reference</code>

SEE ALSO

View device attributes are described in the section “View Device Attributes” on page 7-27.

To set a view device’s attributes, use the `GXSetViewDeviceAttributes` function, described next.

GXSetViewDeviceAttributes

You can use the `GXSetViewDeviceAttributes` function to set or clear the attributes of a view device object.

```
void GXSetViewDeviceAttributes(gxViewDevice target,
                               gxDeviceAttribute attributes);
```

`target` A reference to the view device whose attributes you wish to set.

`attributes` A reference to the view port’s attributes.

DESCRIPTION

The `GXSetViewDeviceAttributes` function sets the attributes of the view device object referenced in the `target` parameter to those specified in the `attributes` parameter. If you pass `gxNoAttributes` for the `attributes` parameter, all attributes are cleared.

You cannot change the attributes of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>invalid_viewDevice_reference</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>viewDevice_access_restricted</code>	(debugging version)

Notices (debugging version)

<code>attributes_already_set</code>

SEE ALSO

View device attributes are described in the section “View Device Attributes” on page 7-27.

To get a view device’s attributes, use the `GXGetViewDeviceAttributes` function, described in the previous section.

GXGetViewDeviceTags

You can use the `GXGetViewDeviceTags` function to examine one or more of the tag objects associated with a view device object.

```
long GXGetViewDeviceTags(gxViewDevice source, long tagType,
                        long index, long count, gxTag items[]);
```

<code>source</code>	A reference to the view device object whose tag list you want to examine.
<code>tagType</code>	The type of tag object to search for. A value of 0 indicates that you want to look for all tag types.
<code>index</code>	The (1-based) index of the first such tag reference to return.
<code>count</code>	The number of tag references to return.
<code>items</code>	An array to hold the returned tag references.

function result The number of tag references found that fit the criteria.

DESCRIPTION

The `GXGetViewDeviceTags` function searches the tag list of the source view device object for references to tag objects with the tag type specified by the `tagType` parameter. If you specify 0 for the `tagType` parameter, the `GXGetViewDeviceTags` function searches all tag types.

You can use the `index` and the `count` parameters to specify which tag references of the appropriate type the `GXGetViewDeviceTags` function should return. The `index` parameter indicates the first tag reference to return and the `count` parameter indicates how many tag references to return. The `index` parameter must be greater than 0. The `count` parameter must be greater than 0 or equal to the `gxSelectToEnd` constant (-1), which indicates that all tag references (starting with the tag reference indicated by the `index` parameter) should be returned.

The function result is the number of tag references found that fit the criteria. If you pass a value other than `nil` for the `items` parameter, the `GXGetViewDeviceTags` function returns in it the tag references that were found.

Typically, you call this function once with a `nil` value for the `items` parameter to determine the number of matching tag references. Then you allocate an appropriately sized array and call the function a second time to obtain the references themselves.

View-Related Objects

ERRORS, WARNINGS, AND NOTICES

Errors

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

Warnings

<code>index_out_of_range</code>
<code>count_out_of_range</code>

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To change the set of tag references associated with a view device, use the `GXSetViewDeviceTags` function, described next.

GXSetViewDeviceTags

You can use the `GXSetViewDeviceTags` function to add, remove, or replace tag objects associated with a view device object.

```
void GXSetViewDeviceTags(gxViewDevice target, long tagType,
                        long index, long oldCount, long newCount,
                        const gxTag items[]);
```

<code>target</code>	A reference to the view device object whose tag list you want to alter.
<code>tagType</code>	The type of tag objects to replace. A value of 0 indicates that you want to replace tags of all types.
<code>index</code>	The (1-based) index of the first tag reference (to a tag object of the appropriate type) to replace.
<code>oldCount</code>	The number of tag references to replace. A value of 0 specifies that you want to insert tag references before the tag reference indicated by the <code>index</code> parameter, rather than replace tag references. A value of -1 (the <code>gxSelectToEnd</code> constant) specifies that all tag references of the requested type, starting with the tag reference indicated by the <code>index</code> parameter, should be replaced.
<code>newCount</code>	The number of tag references to insert. A value of 0 specifies that there are no tag references to insert; the existing tag references that match the criteria you specify are removed from the source shape’s tag list and disposed of.
<code>items</code>	An array of tag references to insert in the tag list.

View-Related Objects

DESCRIPTION

The `GXSetViewDeviceTags` function allows you add tag references to a view device object's tag list, to remove tag references from the list, or to replace tag references in the list with new tag references. In any of these three cases, the `target` parameter specifies the view port device to be modified, the `newCount` parameter specifies the number of tag references to add, and the `items` parameter provides the new tag references.

- To add tag references, set the `oldCount` parameter to 0. Use the `tagType` and the `index` parameters to specify where to add the new tag references. (For example, if you specify `nil` for the `tagType` parameter and 1 for the `index` parameter, this function inserts the new tag references before the current tag references. If you specify a value other than `nil` for the `tagType` parameter and a value of 2 for the `index` parameter, the function inserts the new tag references before the second tag reference with a tag type matching the `tagType` parameter.)
- To remove tag references, set the `newCount` parameter to 0 and the `items` parameter to `nil`. You can use the `index` and the `oldCount` parameters to specify which tag references (of the specified type) should be removed. The `index` parameter indicates the first tag reference (of the specified type) to remove and the `oldCount` parameter indicates how many tag references (of the specified type) to remove.
- To replace tag references, use the `tagType`, `index`, and `oldCount` parameters to indicate which tag references to replace, and use the `newCount` and `items` parameters to specify the new tag references to add. If `newCount` is greater than `oldCount`, the extra tag references are placed immediately adjacent to the last tag reference replaced.

You cannot change the tag list of a view device in the onscreen view group.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>invalid_viewDevice_reference</code>	
<code>tag_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)

Warnings

<code>index_out_of_range</code>
<code>count_out_of_range</code>

Notices (debugging version)

<code>tag_already_set</code>

SEE ALSO

Tag objects are discussed in the chapter “Tag Objects” in this book.

To examine the set of tag references associated with a view port, use the `GXGetViewDeviceTags` function, described in the previous section.

Retrieving the View Devices That Intersect a Shape

The function described in this section allows you to get the view devices on which a shape appears.

GXGetShapeGlobalViewDevices

You can use the `GXGetShapeGlobalViewDevices` function to determine which view devices intersect the area of a shape.

```
long GXGetShapeGlobalViewDevices(gxShape source, gxViewPort port,
                                  gxViewDevice list[]);
```

source A reference to the shape whose view devices you wish to retrieve.

port A reference to a view port that the shape is drawn to.

list An array of view device references. On return, the array lists the view devices that the shape is drawn on.

function result The number of view devices that the shape is drawn on.

DESCRIPTION

The `GXGetShapeGlobalViewDevices` function retrieves a list of view devices that the source shape is drawn on, through the specified view port. The view port must be in the view port list of the shape’s transform object. A returned view device object must have a clip that intersects the clip of the specified view port, and the shape drawn to the view port must also intersect the view device clip.

If the `port` parameter is set to `nil`, all view ports in the view port list of the shape’s transform object are used.

If you set the `list` parameter to `nil`, `GXGetShapeGlobalViewDevices` does not fill out the list of references; it only returns the number of view device references that would be in the list (which may be 0). Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

As one application of this function, you could call `GXGetShapeGlobalViewPorts` to determine the view ports to which a shape is drawn. You then could use one of these view ports in a call to `GXGetShapeGlobalViewDevices` to determine the view devices that the shape would be drawn to.

View-Related Objects

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`invalid_viewPort_reference`

SEE ALSO

For examples of the use of this function, see Listing 7-12 on page 7-58 and Listing 7-13 on page 7-61.

The `GXGetShapeGlobalViewPorts` function is described on page 7-95.

Measuring a Shape in Device Coordinates

The functions described in this section allow you to get a shape's bounding rectangle and its area, as measured on a view device. Other QuickDraw GX functions are available to measure shapes in other contexts:

- To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function, described on page 7-96.
- To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function, described on page 7-125.
- To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetShapeDeviceBounds

You can use the `GXGetShapeDeviceBounds` function to determine the bounding rectangle for the visible part of a shape on a view device.

```
boolean GXGetShapeDeviceBounds(gxShape source, gxViewPort port,
                               gxViewDevice device,
                               gxRectangle *bounds);
```

source A reference to the shape whose bounding rectangle you wish to test for inclusion on a device.

port A reference to a view port to which the shape is drawn.

device A reference to the view device that the shape displays on.

bounds A pointer to a rectangle structure. On return the structure contains the bounding rectangle, in device coordinates, for the part of the shape that appears on the device.

function result `true` if the bounding rectangle overlaps the view device clip; `false` if it does not.

DESCRIPTION

The `GXGetShapeDeviceBounds` function returns a value specifying whether the bounding rectangle of the shape drawn to the specified view port is within the clip area of the specified view device. The view port and view device must be in the same view group. The view port must be in the view port list of the shape's transform object.

You can specify `nil` for the `port` parameter, in which case `GXGetShapeDeviceBounds` includes all view ports in the view port list of the source shape's transform. You can specify `nil` for the `device` parameter, in which case `GXGetShapeDeviceBounds` includes any view device that intersects any of the specified view ports.

Unless you pass `nil` for the `bounds` parameter, the function also returns in the `bounds` parameter the part of the shape's bounding rectangle that displays on the device. The rectangle shape is defined in device coordinates.

To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function. To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function. To determine a shape's bounding rectangle in global coordinates, use the `GXGetShapeGlobalBounds` function.

SPECIAL CONSIDERATIONS

If the bounding rectangle of the source shape spans more than one device, this function posts an `unable_to_get_bounds_on_multiple_devices` warning.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>invalid_viewPort_reference</code>	
<code>invalid_viewDevice_reference</code>	
<code>inconsistent_parameters</code>	(debugging version)

Warnings

`unable_to_get_bounds_on_multiple_devices`

SEE ALSO

For an example of the use of this function, see page 7-59.

To calculate the area of a shape on a device, use the `GXGetShapeDeviceArea` function, described next.

The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeLocalBounds` function is described on page 7-96. The `GXGetShapeGlobalBounds` function is described on page 7-125.

For information about coordinate spaces, see the section “About Drawing, Coordinate Conversion, and Clipping” beginning on page 7-30.

GXGetShapeDeviceArea

You can use the `GXGetShapeDeviceArea` function to calculate the area of a shape on a given view device.

```
long GXGetShapeDeviceArea(gxShape source, gxViewPort port,
                          gxViewDevice device);
```

`source` A reference to the shape whose area on the device you want to determine.

`port` A reference to a view port that the shape is drawn to.

`device` A reference to the view device for which you wish to calculate the area.

function result The number of pixels that represent the shape on the device.

DESCRIPTION

The `GXGetShapeDeviceArea` function returns the number of pixels covered by the source shape in the view port on the specified view device. The shape object cannot be a bitmap or picture shape. The view port must be in the view port list of the shape's transform object.

This function accounts for just the pixels that would actually be affected if the shape were drawn. Spaces inside of framed or filled geometric shapes, or spaces within and between glyphs of typographic shapes, do not contribute to the calculated area.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`invalid_viewPort_reference`
`invalid_viewDevice_reference`
`illegal_type_for_shape` (debugging version)

SEE ALSO

To determine the bounding rectangle of a shape on a device, use the `GXGetShapeDeviceBounds` function, described in the previous section.

To determine the area of a shape in geometry-space coordinates, use the `GXGetShapeArea` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

Measuring the Colors and Pattern Width of a Shape on a Device

The function described in this section allows you to determine the exact colors and size of pattern that QuickDraw GX will use to draw a shape on a given view device.

GXGetShapeDeviceColors

You can use the `GXGetShapeDeviceColors` function to determine the set of colors with which a shape will be drawn on a given view device, as well as the width of any repeating pattern with which the shape will be drawn.

```
gxColorSet GXGetShapeDeviceColors(gxShape source,
                                   gxViewPort port,
                                   gxViewDevice device,
                                   long *width);
```

<code>source</code>	A reference to the shape whose colors you wish to determine.
<code>port</code>	A reference to a view port to which the shape is drawn.
<code>device</code>	A reference to the view device on which the shape is drawn.
<code>width</code>	A pointer to a long value. On return, the value is the width of the repeated pattern formed by dithering or halftoning, or by the pattern—if any—specified in the shape’s style object.

function result A reference to a color set that contains the colors with which the shape can be drawn.

DESCRIPTION

The `GXGetShapeDeviceColors` function returns a color set containing the colors that the shape would be drawn with through the view port onto the view device. The view port must be in the view port list of the shape’s transform object. If no shape sharing the ink of the source shape intersects the view port and view device, the function returns `nil`.

The `GXGetShapeDeviceColors` function returns only the number of unique colors in the dither pattern or the halftone pattern, not the size of the dither or the halftone. It also does not take transfer modes into account, or the colors already on the view device.

This function does not check that the shape actually intersects the view device; you may want to call the `GXGetShapeGlobalViewDevices` function first. The shape object cannot be a bitmap or picture shape.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
invalid_viewPort_reference
invalid_viewDevice_reference
```

View-Related Objects

SEE ALSO

For information about color sets, see “When Color Matching Occurs” beginning on page 4-31.

For information about dithers and halftones, see the sections “Dither” beginning on page 7-10 and “Halftone” beginning on page 7-13.

Patterns and the style object are described in the geometric styles chapter of *Inside Macintosh: QuickDraw GX Graphics*.

To determine the view devices that a shape is displayed on, use the `GXGetShapeGlobalViewDevices` function, described on page 7-115.

Hit-Testing a Shape on a Device

The function described in this section allows you to hit-test a shape in relation to the pixels of a view device.

GXHitTestDevice

You can use the `GXHitTestDevice` function to determine whether a point in device space is within a given tolerance of a shape displayed on that device.

```
gxShape GXHitTestDevice(gxShape target, gxViewPort port,
                        gxViewDevice device, const gxPoint *test,
                        const gxPoint *tolerance);
```

<code>target</code>	A reference to the shape to hit-test.
<code>port</code>	A reference to a view port that the shape is drawn to.
<code>device</code>	A reference to the view device on which the shape is drawn.
<code>test</code>	A pointer to a point structure specifying the location, in device coordinates (pixels), to hit-test the shape against.
<code>tolerance</code>	A pointer to a point structure specifying a rectangular shape whose size specifies the distance, in pixels, from the target shape that the test point can be and still be considered a successful hit.

function result A reference to the target shape if the shape was hit; otherwise `nil`.

DESCRIPTION

The `GXHitTestDevice` function returns the target shape within the specified view port if the hit is successful, otherwise it returns `nil`. All clipping, from transform through view port and view device, is taken into account in determining whether a hit is possible.

The test point represents a pixel location in view device coordinates. The tolerance represents a rectangular area of pixels, defining the “radius” of the total tolerance area.

View-Related Objects

You can think of four such rectangles as making up a larger rectangle centered on the hit point; if the distance from the hit point to the shape is within that larger rectangle, the hit is considered successful.

Negative values for the tolerance are permitted.

If the `port` parameter is set to 0, all view ports on the view device are tested. If the `device` parameter is set to 0, all view devices intersected by the view port are tested. If both `port` and `device` parameters are set to 0, all view ports that the shape is drawn to and all view devices drawn to by the target shape are tested.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`parameter_is_nil` (debugging version)

SEE ALSO

To hit-test individual parts of a shape's geometry, use the `GXHitTestShape` function, described in the chapter "Shape Objects" in this book. To hit-test the parts of a picture shape, use the `GXHitTestPicture` function, described in the picture shapes chapter of *Inside Macintosh: QuickDraw GX Graphics*. To hit-test the text of a layout shape, use the `GXHitTestLayout` function, described in the layout carets chapter of *Inside Macintosh: QuickDraw GX Typography*.

For more information on `GXHitTestDevice` and how it relates to the other hit-testing functions, see "Hit-Testing a Shape on a Device" on page 7-60.

View Group Functions

This section describes the QuickDraw GX functions you use with view group objects. Using the functions described here, you can

- create and dispose of view group objects
- determine the view ports and view devices that belong to a view group
- measure a shape in a view group's global space

Creating and Disposing of View Group Objects

The functions described in this section allow you to create and dispose of view groups.

GXNewViewGroup

You can use the `GXNewViewGroup` function to create a new view group object.

```
gxViewGroup GXNewViewGroup(void);
```

function result A reference to the new view group.

DESCRIPTION

The `GXNewViewGroup` function returns a unique view group reference. You can then use the new view group to create view ports and view devices that share the same global space. QuickDraw GX provides an onscreen view group, `gxScreenViewDevices`, for you. You only need to create offscreen view groups.

SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewViewGroup` function creates a view group object; you are responsible for disposing of that object when you no longer need it.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`

SEE ALSO

For examples of the use of this function, see Listing 7-13 on page 7-61 and Listing 7-14 on page 7-63.

For information about view groups, see “About View Group Objects” beginning on page 7-29.

To dispose of a view group, use the `GXDisposeViewGroup` function, described next.

GXDisposeViewGroup

You can use the `GXDisposeViewGroup` function to delete a view group object.

```
void GXDisposeViewGroup(gxViewGroup target);
```

target A reference to the view group.

DESCRIPTION

The `GXDisposeViewGroup` function deletes the view group, as well as all view devices and view ports that belong to that view group. If you create an offscreen view group with several view ports and view devices, you needn't dispose of those view ports and view devices when you are finished, as long as you dispose of the view group itself.

ERRORS, WARNINGS, AND NOTICES**Errors**

`invalid_viewGroup_reference`

SEE ALSO

For an example of the use of this function, see page 7-63.

For information about view groups, see “About View Group Objects” beginning on page 7-29.

Getting the View Ports and View Devices of a View Group

The functions described in this section allow you to find out what view ports and view devices belong to a view group.

GXGetViewGroupViewPorts

You can use the `GXGetViewGroupViewPorts` function to retrieve a list of the view ports that are associated with a view group object.

```
long GXGetViewGroupViewPorts(gxViewGroup source,
                             gxViewport list[]);
```

<code>source</code>	A reference to the view group whose view ports you wish to examine.
<code>list</code>	An array of view port references. On return, the array contains a list of references to the view ports belonging to the source view group.

function result The number of view port references in the `list` array.

DESCRIPTION

The `GXGetViewGroupViewPorts` function fills out a list of all the view ports in the source view group and returns, as its function result, the number of view ports in the list.

If you pass `gxAllViewDevices` for the `source` parameter, this function returns all view ports in all view groups.

View-Related Objects

If you set the `list` parameter to `nil`, `GXGetViewGroupViewPorts` does not fill out the list of references; it only returns the number of view port references that would be in the list. Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`invalid_viewGroup_reference`

SEE ALSO

To get a list of all the view devices in a view group, use the `GXGetViewGroupViewDevices` function, described next.

GXGetViewGroupViewDevices

You can use the `GXGetViewGroupViewDevices` function to retrieve a list of the view devices that are associated with a view group object.

```
long GXGetViewGroupViewDevices(gxViewGroup source,
                               gxViewDevice list[]);
```

<code>source</code>	A reference to the view group whose view devices you wish to examine.
<code>list</code>	An array of view device references. On return, the array contains a list of references to the view devices belonging to the source view group.

function result The number of view device references in the `list` array.

DESCRIPTION

The `GXGetViewGroupViewDevices` function fills out a list of all the view devices in the source view group and returns, as its function result, the number of view devices in the list.

If you pass `gxAllViewDevices` for the `source` parameter, this function returns all view devices in all view groups.

If you set the `list` parameter to `nil`, `GXGetViewGroupViewDevices` does not fill out the list of references; it only returns the number of view device references that would be in the list. Thus, you typically call this function twice: first to get the size of array to allocate for the `list` parameter, and second to retrieve the list itself.

ERRORS, WARNINGS, AND NOTICES

Errors`invalid_viewGroup_reference`

SEE ALSO

For an example of the use of this function, see Listing 7-10 on page 7-54.

To get a list of all the view ports in a view group, use the `GXGetViewGroupViewPorts` function, described in the previous section.

Measuring a Shape in Global Coordinates

The function described in this section allows you to determine the bounding rectangle of a shape in the global coordinates of its view group. Other QuickDraw GX functions are available to measure shapes in other contexts:

- To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function, described on page 7-96.
- To determine a shape's bounding rectangle on a view device, use the `GXGetShapeDeviceBounds` function, described on page 7-116.
- To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function, described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*.

GXGetShapeGlobalBounds

You can use the `GXGetShapeGlobalBounds` function to determine the bounding rectangle of a shape in global coordinates.

```
boolean GXGetShapeGlobalBounds(gxShape source,
                               gxViewport port,
                               gxViewGroup group,
                               gxRectangle *bounds);
```

<code>source</code>	A reference to the shape whose bounding rectangle you wish to determine in global coordinates.
<code>port</code>	A reference to a view port to which the shape is drawn.
<code>group</code>	A reference to the view group that defines the global coordinates.
<code>bounds</code>	A pointer to a rectangle structure. On return, the structure contains the bounding rectangle for the shape, in the global coordinates of the specified view group.

function result `true` if the bounding rectangle appears in global space; `false` if it does not.

View-Related Objects

DESCRIPTION

The `GXGetShapeGlobalBounds` function returns a value that specifies whether the bounding rectangle of the shape drawn to the specified view port appears anywhere in the global space of the specified view group. The view port must belong to the view group, and it must be referenced in the view port list of the shape's transform object.

The `GXGetShapeGlobalBounds` function also returns in the `bounds` parameter the bounding rectangle of that part of the shape that can be drawn through the specified view port. The function returns the bounding rectangle after the shape's transform clip, mapping and style have been applied, and after all view port mappings and clips have been applied, from the view port specified in the `port` parameter to the root view port in the view port hierarchy (if any). The dimensions of the rectangle are in the global coordinates of the view group.

If you specify `nil` for the `port` parameter, `GXGetShapeGlobalBounds` includes all view ports specified in the source shape's transform's view port list. If you specify `nil` for the `group` parameter, `GXGetShapeGlobalBounds` includes all view groups of all specified view ports.

To determine a shape's bounding rectangle in geometry-space coordinates, use the `GXGetShapeBounds` function. To determine a shape's bounding rectangle in local coordinates, use the `GXGetShapeLocalBounds` function. To determine a shape's bounding rectangle in device coordinates, use the `GXGetShapeDeviceBounds` function.

ERRORS, WARNINGS, AND NOTICES**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>invalid_viewPort_reference</code>	
<code>invalid_viewGroup_reference</code>	
<code>parameter_is_nil</code>	(debugging version)

Notices (debugging version)

<code>transform_references_disposed_viewPort</code>

SEE ALSO

For an example of the use of this function, see Listing 7-15 on page 7-64.

The `GXGetShapeBounds` function is described in the geometric operations chapter of *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeLocalBounds` function is described on page 7-96. The `GXGetShapeDeviceBounds` function is described on page 7-116.

Summary of View-Related Objects

Constants and Data Types

The View Port Object

```
typedef struct gxPrivateViewPortRecord *gxViewPort;
```

The Halftone Structure

```
struct gxHalftone{
    Fixed          angle;
    Fixed          frequency;
    gxDotType      method;
    gxTintType     tinting;
    gxColor        dotColor;
    gxColor        backgroundColor;
    gxColorSpace   tintSpace;
};
```

Dot Types

```
enum gxDotTypes{
    gxRoundDot = 1,
    gxSpiralDot,
    gxSquareDot,
    gxLineDot,
    gxEllipticDot,
    gxTriangleDot,
    gxDispersedDot
};
```

```
typedef long gxDotType;
```

Tint Types

```
enum gxTintTypes{
    gxNoTint,
    gxLuminanceTint,
    gxAverageTint,
```

View-Related Objects

```

    gxMixtureTint,
    gxComponent1Tint,
    gxComponent2Tint,
    gxComponent3Tint,
    gxComponent4Tint
};

typedef long gxTintType;

```

View Port Attributes

```

enum gxPortAttributes {
    gxGrayPort          = 0x0001,    /* convert to gray space */
    gxAlwaysGridPort    = 0x0002,    /* use gxDeviceGridStyle */
    gxEnableMatchPort   = 0x0004     /* perform color matching */
};

typedef long gxPortAttribute;

```

The View Device Object

```

typedef struct gxPrivateViewDeviceRecord *gxViewDevice;

```

View Device Attributes

```

enum gxDeviceAttributes{
    gxDirectDevice      = 0x01,    /* pixel image must be accessible */
    gxRemoteDevice      = 0x02,    /* pixel image may be on card */
    gxInactiveDevice    = 0x04     /* device is inactive */
};

typedef long gxDeviceAttribute;

```

The View Group Object

```

typedef struct gxPrivateViewGroupRecord *gxViewGroup;

```

View Group Types

```

#define gxAllViewDevices      ((gxViewGroup) 0)
#define gxScreenViewDevices  ((gxViewGroup) 1)

```

View Port Functions

Creating and Manipulating View Port Objects

```

gxViewport GXNewViewport      (gxViewport group);
void GXDisposeViewport        (gxViewport target);
gxViewport GXCopyToViewport    (gxViewport target, gxViewport source);
boolean GXEqualViewport        (gxViewport one, gxViewport two);

```

Manipulating View Port Object Properties

```

gxShape GXGetViewportClip      (gxViewport source);
void GXSetViewportClip          (gxViewport target, gxShape clip);
gxMapping *GXGetViewportMapping
                                (gxViewport source, gxMapping *map);
void GXSetViewportMapping        (gxViewport target, const gxMapping *map);
gxMapping *GXGetViewportGlobalMapping
                                (gxViewport source, gxMapping *map);
long GXGetViewportDither        (gxViewport source);
void GXSetViewportDither        (gxViewport target, long level);
boolean GXGetViewportHalftone
                                (gxViewport source, gxHalftone *data);
void GXSetViewportHalftone      (gxViewport target, const gxHalftone *data);
Fixed GXGetHalftoneDeviceAngle
                                (gxViewDevice source, const gxHalftone *data);
gxViewport GXGetViewportParent
                                (gxViewport source);
void GXSetViewportParent        (gxViewport target, gxViewport parent);
long GXGetViewportChildren      (gxViewport source, gxViewport list[]);
void GXSetViewportChildren      (gxViewport target, long count,
                                const gxViewport list[]);
gxViewport GXGetViewportViewGroup
                                (gxViewport source);
void GXSetViewportViewGroup      (gxViewport target, gxViewGroup group);
gxPortAttribute GXGetViewportAttributes
                                (gxViewport source);
void GXSetViewportAttributes    (gxViewport target,
                                gxPortAttribute attributes);
long GXGetViewportTags          (gxViewport source, long tagType, long index,
                                long count, gxTag items[]);

```

View-Related Objects

```
void GXSetViewportTags      (gxViewport target, long tagType, long index,
                           long oldCount, long newCount,
                           const gxTag items[]);
```

Retrieving the View Devices That Intersect a View Port

```
long GXGetViewportViewDevices(gxViewport source, gxViewDevice list[]);
```

Retrieving the View Ports That Intersect a Shape

```
long GXGetShapeGlobalViewPorts
                           (gxShape source, gxViewport list[]);
```

Measuring a Shape in Local Coordinates

```
gxRectangle *GXGetShapeLocalBounds
                           (gxShape source, gxRectangle *bounds);
```

View Device Functions

Creating and Manipulating View Device Objects

```
gxViewDevice GXNewViewDevice (gxViewGroup group, gxShape bitmapShape);
void GXDisposeViewDevice     (gxViewDevice target);
gxViewDevice GXCopyToViewDevice
                           (gxViewDevice target, gxViewDevice source);
boolean GXEqualViewDevice    (gxViewDevice one, gxViewDevice two);
```

Manipulating View Device Object Properties

```
gxShape GXGetViewDeviceClip (gxViewDevice source);
void GXSetViewDeviceClip    (gxViewDevice target, gxShape clip);
gxMapping *GXGetViewDeviceMapping
                           (gxViewDevice source, gxMapping *map);
void GXSetViewDeviceMapping (gxViewDevice target, const gxMapping *map);
gxShape GXGetViewDeviceBitmap
                           (gxViewDevice source);
void GXSetViewDeviceBitmap  (gxViewDevice target, gxShape bitmapShape);
gxViewGroup GXGetViewDeviceViewGroup
                           (gxViewDevice source);
void GXSetViewDeviceViewGroup
                           (gxViewDevice target, gxViewGroup group);
gxDeviceAttribute GXGetViewDeviceAttributes
                           (gxViewDevice source);
```

```

void GXSetViewDeviceAttributes
    (gxViewDevice target, gxDeviceAttribute
     attributes);
long GXGetViewDeviceTags
    (gxViewDevice source, long tagType, long index,
     long count, gxTag items[]);
void GXSetViewDeviceTags
    (gxViewDevice target, long tagType, long index,
     long oldCount, long newCount,
     const gxTag items[]);

```

Retrieving the View Devices That Intersect a Shape

```

long GXGetShapeGlobalViewDevices
    (gxShape source, gxViewPort port,
     gxViewDevice list[]);

```

Measuring a Shape in Device Coordinates

```

boolean GXGetShapeDeviceBounds
    (gxShape source, gxViewPort port,
     gxViewDevice device, gxRectangle *bounds);
long GXGetShapeDeviceArea
    (gxShape source, gxViewPort port,
     gxViewDevice device);

```

Measuring the Colors and Pattern Width of a Shape on a Device

```

gxColorSet GXGetShapeDeviceColors
    (gxShape source, gxViewPort port,
     gxViewDevice device, long *width);

```

Hit-Testing a Shape on a Device

```

gxShape GXHitTestDevice
    (gxShape target, gxViewPort port,
     gxViewDevice device, const gxPoint *test,
     const gxPoint *tolerance);

```

View Group Functions

Creating and Disposing of View Group Objects

```

gxViewGroup GXNewViewGroup    (void);
void GXDisposeViewGroup
    (gxViewGroup target);

```

View-Related Objects

Getting the View Ports and View Devices of a View Group

```
long GXGetViewGroupViewPorts (gxViewGroup source, gxViewPort list[]);  
long GXGetViewGroupViewDevices  
    (gxViewGroup source, gxViewDevice list[]);
```

Measuring a Shape in Global Space

```
boolean GXGetShapeGlobalBounds  
    (gxShape source, gxViewPort port,  
     gxViewGroup group, gxRectangle *bounds);
```