QuickDraw GX Mathematics

This chapter describes QuickDraw GX number formats, number-format conversions, mathematical functions, and functions that operate on mappings (transformation matrices). Read this chapter if your application requires the explicit use of any of the mathematical capabilities of QuickDraw GX.

Related information on how QuickDraw GX uses mappings can be found in the chapter "Transform Objects" and the chapter "View-Related Objects" in *Inside Macintosh: QuickDraw GX Objects*.

This chapter first describes the number formats used in QuickDraw GX. It then describes the number-format conversion macros and mathematical functions that are provided by QuickDraw GX. It then shows how to use QuickDraw GX macros and functions to provide

- fixed-point number conversions
- fixed-point operations
- operations on 64-bit numbers
- vector operations
- Cartesian and polar coordinate conversions
- random number generation
- roots of linear and quadratic equations
- bit analysis
- mapping operations

About QuickDraw GX Mathematics

QuickDraw GX supports 16-bit, 32-bit, and 64-bit fixed-point number formats. You can use QuickDraw GX macros for efficient number-format conversions. QuickDraw GX mathematical functions provide a full spectrum of operations. QuickDraw GX mapping functions allow you to manipulate the matrices that transform shapes.

Number Formats

QuickDraw GX accepts standard integer and floating-point number formats, and defines several fixed-point number formats.

Integer Formats

Some Quickdraw GX functions and data structures may make use of the standard C language integer formats `short`, `unsigned short`, `long`, and `unsigned long`. The **short number** format is a 16-bit signed or unsigned integer; the **long number** format is a 32-bit signed or unsigned integer. Numbers in these formats have the following ranges of values:

Format	Range
`short`	–32, 768 to 32,767
`unsigned short`	0 to 65,535
`long`	–2,147,483,648 to 2,147,483,647
`unsigned long`	0 to 4,294,967,295

Floating-Point Formats

QuickDraw GX supports conversion to and from the C language single precision floating-point format `float`; double precision floating-point format `double`; and extra precision floating-point format `extended`. QuickDraw GX macros that convert between floating-point numbers and `Fixed` or `fract` numbers can handle all three floating-point formats.

Fixed-Point Formats

QuickDraw GX defines 16-bit, 32-bit, and 64-bit **fixed-point number** formats. Fixed-point number formats are integers that are interpreted as real numbers. The conversion between integer number format and a fixed-point number format is described by bias. A **bias** is a number (commonly expressed as a power of 2) by which an integer is divided in order to obtain the real number it represents. For example, the bias for the `Fixed` number format is 16 bits, or 2^{16}. In this case, the integer must be divided by 2^{16} to obtain the real number represented. Therefore, `Fixed` 0x10000 = 65,536 / 2^{16}, or 1.0.

There are one 16-bit, two 32-bit, and one 64-bit number formats:

- The **`gxColorValue`** format for fixed-point numbers is a 16-bit unsigned integer. The values range from 0 to 65,535 to represent numbers from 0 to 1. This fixed-point number is described by a bias of 65,535. The integer must be divided by 65,535 to obtain the real number represented. (Its name derives from the fact that it is used to describe color-component values in a Quickdraw GX color structure; see the chapter "Colors and Color-Related Objects" in *Inside Macintosh: QuickDraw GX Objects* for more information.)
- The **`Fixed`** format for fixed-point numbers has 16 bits to the left and 16 bits to the right of the binary point. This corresponds to a fixed-point bias of 16 bits. `Fixed` format numbers range from –32,768 to [32,767 + (65,535 / 65,536)], or approximately 32,768.

- The **`fract`** format for fixed-point numbers has 2 bits to the left and 30 bits to the right of the binary point. This corresponds to a fixed-point bias of 30 bits. Numbers in `fract` format range from –2 to [2 – (2^{-30})] or –2 to [1 + (1,073,741,823 / 1,073,741,824)], or approximately 2.0.
- The **`wide`** format is a signed integer data type that has 64 bits. It can be given a bias, just as any other integer type can. With a bias of 0 bits, a `wide` format number represents an integer and can range from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. With a bias of 16 bits, a `wide` format number represents an extended version of the `Fixed` format (that is, it has the same precision but a larger range) and can range from –140,737,488,355,328 to [140,737,488,355,327 + (65,535 / 65,536)].

All of the fixed-point number formats except for `gxColorValue` are two's complement signed integers.

The `wide` data type is defined as a structure that contains an unsigned long integer as its low-order half and a signed long integer as its high-order half. You can convert a `long` into a `wide` in either of two ways:

- First, assign the `long` to the low half of the `wide`. Then, if the `long` is not negative, assign 0 to the high half of the `wide`; if the `long` is negative, assign –1 to the high half of the `wide`.
- Assign the `long` to the high half of the `wide`. Use the `WideShift` function to shift the bits of the `wide` rightward by 32 bits.

The WideShift function is described on page 8-51. The `wide` structure is described on page 8-35.

Working With Bias in Fixed-Point Operations

`Fixed` numbers have a bias of 16; `fract` numbers have a bias of 30; and `long` and `wide` numbers have a bias of 0. Unless stated otherwise, all biases will be powers of 2. For brevity, we use the convention of describing a bias by the exponent of 2; for example, we say "a bias of 16" instead of "a bias of 16 bits."Operations that are designed to work on a specific number format (such as `FixedMultiply` or `FractDivide` or `WideMultiply`) apply a bias to the result of their operations that reflects the number format they expect. If you understand how the bias is applied, you can use (and even mix) any of several different fixed-point number formats in these functions, and know what bias to use when interpreting the result:

- Operations on `Fixed` numbers (such as `FixedMultiply` and `FixedDivide`) use a bias of 16; operations on `fract` numbers (such as `FractMultiply` and `FractDivide`) use a bias of 30; operations on `long` and `wide` numbers (such as `MultiplyDivide` and `WideDivide`) use a bias of 0.
- When multiplying two fixed-point numbers, the bias of the result is the sum of the biases of the input numbers, minus the bias of the operation. Thus, the result of using `FixedMultiply` to multiply two `Fixed` numbers is a `Fixed` (= (16 + 16) – 16), as expected. On the other hand, the result of using `FixedMultiply` to multiply a `fract` and a `Fixed` is a `fract` (= (30 + 16) – 16), and the result of `FractMultiply` on a `fract` and a `Fixed` is a `Fixed` (= (30 + 16) – 30).

- When dividing two fixed-point numbers, the bias of the result is the operation bias plus the difference between the biases of the input numbers. Thus, as expected, the result of using `FixedDivide` to divide one `Fixed` number by another is a `Fixed` (= 16 + (16 – 16)). The result of `FixedDivide` on a `fract` divided by a `Fixed` is a `Fract` (= 16 + (30 – 16)), and the result of `FractDivide` on a `Fixed` divided by a `Fract` is a `Fixed` (= 30 + (16 – 30)).
- For operations that have no bias, the result is simply the sum or difference of the input biases. For example, if you use `MultiplyDivide` to multiply a `Fixed` by a `fract` and divide the result by a `Fixed`, the result will be a `fract` (= 16 + 30 – 16). If you use `WideMultiply` to multiply two `Fixed` numbers, the result will have a bias of 32 bits (= 16 + 16).

Remember also that using the standard C operators + and – to add or subtract fixed-point numbers is meaningful only if the numbers have the same bias. Thus, if you wish to add together a long integer and a `Fixed`, for example, you must first convert one format to the other, or convert both to a common format.

The functions referred to in this section are described in the section "Fixed-Point Operations" beginning on page 8-42, and "Operations on wide Numbers" beginning on page 8-49.

Number-Conversion Macros

QuickDraw GX provides a set of predefined **macros** for the conversion between different fixed-point number formats. This allows you the convenience of accessing the number-conversion formulas as if they were function calls.

Table 8-1 summarizes the number-format conversions that are supported.

Table 8-1 Macro number-format conversions

From number format	To number format
`Fixed`	`fract`
`Fixed`	floating-point
`Fixed`	integer
`fract`	floating-point
`fract`	`Fixed`
`fract`	`gxColorValue`
floating-point	`fract`
floating-point	`Fixed`
integer	`Fixed`
`gxColorValue`	`fract`

QuickDraw GX also provides macros that

- round a `Fixed` number to its nearest integer
- determine the greatest integer that is not greater than a given `Fixed` number
- use a function to determine the square root of a `Fixed` number

The use of QuickDraw GX macros is described in the section “Converting Number Formats” beginning on page 8-26. Each macro is described in the section “Number-Conversion Macros” beginning on page 8-36.

Mathematical Functions

QuickDraw GX provides mathematical functions for

- fixed-point operations on `Fixed`, `long`, and `fract` number formats
- fixed-point operations on a `wide` number format
- vector operations
- Cartesian and polar coordinate point conversions
- random number generation
- linear and quadratic roots
- bit analysis

A description of each QuickDraw GX mathematics function is provided in the section “Mathematical Functions” beginning on page 8-42.

Operations on Fixed, long, and fract Numbers

QuickDraw GX provides functions that perform operations on `Fixed`, `long`, and `fract` number formats. Functions are provided that

- determine the product of two numbers ($a \times b$)
- determine the quotient of two numbers (a / b)
- determine the product of two numbers and the quotient of a third number ($(a \times b) / c$)
- determine both the sine and cosine of an angle measured in degrees [sine(angle) and cosine(angle)]
- determine the square root of a number $(a)^{1/2}$
- determine the cube root of a number $(a)^{1/3}$
- determine the magnitude of a two-dimensional vector

The functions that perform operations on `Fixed`, `long`, and `fract` number formats are described in the section “Fixed-Point Operations” beginning on page 8-42.

Operations on wide Numbers

QuickDraw GX provides functions for operations on `wide` numbers. Functions are provided that

- determine the sum of two `wide` numbers (a + b)
- determine the difference between two `wide` numbers (a – b)
- determine the product, as a `wide` number, of two `long` numbers ($a \times b$)
- determine the quotient, as a `long` number (without remainder), of a `wide` number divided by a `long` number (a / b)
- determine the result, as a `long` quotient and a `long` remainder, of dividing a `wide` number by a `long` number (a / b + remainder)
- determine the square root of a `wide` number $(a)^{1/2}$
- negate a `wide` number (–a)
- shift bits in a `wide` number to the right or left
- determine the highest order bit in the absolute value of a `wide` number
- compare two `wide` numbers

The functions that perform operations on `wide` number formats are described in the section "Operations on wide Numbers" beginning on page 8-49.

Vector Operations

QuickDraw GX provides vector operation functions that

- determine the dot product of two vectors ($v_1 \bullet v_2$)
- determine the dot product of two vectors and divide by a number $(v_1 \bullet v_2)/a$

The use of QuickDraw GX vector operation functions is described in the section "Performing Vector Operations" beginning on page 8-29. These functions are described in the section "Vector Operations" beginning on page 8-54.

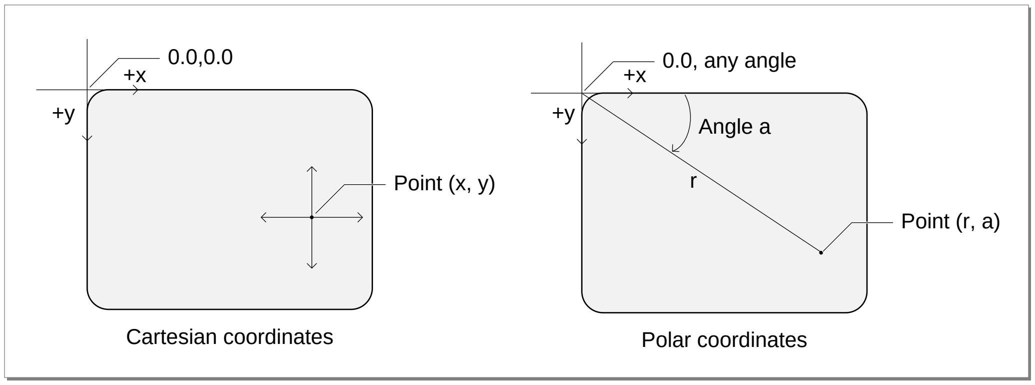
Cartesian and Polar Coordinate Conversion

You use Cartesian coordinates to specify points with QuickDraw GX. Some shapes, such as rectangles, are more easily drawn using Cartesian coordinates; however, some shapes that have symmetry about a point are more easily drawn with polar coordinates. For that reason, QuickDraw GX provides conversion routines so that you can work in either coordinate system.

For QuickDraw GX, **Cartesian coordinates** have a positive *x* direction to the right and a positive *y* direction downward (not upward, as in many other Cartesian coordinate systems). Cartesian coordinates are written in the order (*x*, *y*). The origin is at (0, 0). The `gxPoint` structure describes points using Cartesian coordinates.

Polar coordinates have the same origin point as Cartesian coordinates, but locations are specified differently. The polar coordinate of a point is specified by the length of the radius vector *r* from the origin to the point and the direction of the vector is specified by polar angle *a*. Angles in QuickDraw GX are measured clockwise in degrees from the Cartesian coordinate positive x-axis. The polar coordinate of a point specified by a vector of length *r* and direction *a* degrees from the x-axis is written as point (*r*, *a*). The polar origin point has the coordinates (0, *a*), where *a* is any angle. Points having polar coordinates are defined by the `gxPolar` structure. The `gxPolar` structure is described in the section "Constants and Data Types" beginning on page 8-35. The relationship of the Cartesian and polar coordinates is shown in Figure 8-1.

Figure 8-1 Cartesian and polar coordinates



The `gxPolar` location (*r*, *a*) corresponds to the `gxPoint` location ($r \times \cos(a)$, $r \times \sin(a)$). The mathematical relationship between the two coordinate systems is given by the expressions $r^2 = x^2 + y^2$ and $\tan(a / 2) = y / (r + x)$. The angle can also be defined by the more familiar term $\tan(a) = y / x$.

The use of the polar-to-Cartesian and Cartesian-to-polar coordinates functions are described in the section "Converting Between Cartesian and Polar Coordinates" beginning on page 8-29. These functions are described in the section "Cartesian and Polar Coordinate Point Conversions" beginning on page 8-56.

Random Number Generation

The QuickDraw GX random-number algorithm generates random integers in the range of 0 to $2^{count} - 1$, where *count* is the number of bits to be generated by the random number generator.

The sequence of values that the random number generator produces is dependent upon the initialization value called the **seed**. The algorithm uses the seed to calculate the next random number and a new seed. If no seed is provided, QuickDraw GX uses a default seed value of 0. To repeat a sequence of random numbers, you can use the same seed value.

QuickDraw GX provides random number generation functions that

- generate a sequence of random bits
- change the seed used by the random number algorithm
- determine the current seed for the random number algorithm

The use of the random number generation functions is described in the section "Generating Random Numbers" beginning on page 8-33. These functions are described in the section "Random Number Generation" beginning on page 8-58.

Roots of Linear and Quadratic Equations

QuickDraw GX provides mathematical functions that

- determine the root of a linear equation
- determine the roots of a quadratic equation

The linear and quadratic equation solving functions are described in the section "Linear and Quadratic Roots" beginning on page 8-60.

Bit Analysis

QuickDraw GX provides a mathematical function that allows you to determine the highest bit number that is set in a number.

The `FirstBit` function is described in the section "Bit Analysis" beginning on page 8-62.

Transformation Operations With Mappings

A **mapping** is a 3×3 perspective matrix that performs transformations of spatial locations in two dimensions. You can apply a mapping operation to a set of points either directly (as when directly modifying the geometry of a shape), or indirectly, by multiplying a mapping with another mapping (as when altering the mapping in the transform object associated with a shape).

QuickDraw GX uses mappings to perform the following transformations on shapes or other two-dimensional data:

- **Translation** shifts the position of a shape by the amount specified in the mapping. Translation functions allow you to specify either a relative shift along either coordinate axis, or an absolute shift to a new specified location.
- **Scaling** changes the size of a shape by the factor specified in the mapping. Scaling functions allow you to change size along either axis, and can also result in reflection about the coordinate axes.
- **Rotation** changes the angle of rotation of a shape by the amount specified in the mapping, rotating all points around a given point.
- **Skewing** changes the slant applied to a shape by the amount specified in the mapping. Skewing functions allow you to apply slant along either coordinate axis, relative to a given point. The term *shearing* is synonymous with skewing.
- **Perspective** modifies the positions of points to give a three-dimensional effect.

When you multiply two or more matrices to obtain a cumulative result, you **concatenate,** or accumulate the transformations of, both mappings. Matrix multiplication is not commutative. This means that $[A] \times [B] \neq [B] \times [A]$. As a result, the order that you concatenate is important. [A] is **postmultiplied** by [B] if [A] is replaced by $[A] \times [B]$. Conversely, [A] is **premultiplied** by [B] if [A] is replaced by $[B] \times [A]$. A mapping is applied to a point via postmultiplication (which is to say that points are row vectors); therefore, the default for applying one mapping to another is also postmultiplication.

Multiple concatenations can occur in QuickDraw GX, such as when drawing picture shapes or when drawing any shape through a hierarchy of view ports. If you are going to apply several mappings to a relatively large bitmap or other shape, it is advantageous to concatenate the mappings first (with the `MapMapping` function) and then apply the resultant mapping to the shape (with the `GXMapShape` function).

The motivation is speed. It is much faster to concatenate mappings than to apply a mapping to a large number of points. For bitmaps, an additional motivation is accuracy. Each time a shape is transformed, a certain amount of roundoff error is introduced. Because the pixels of a bitmap are at integral coordinates, the roundoff error is on the average of a quarter pixel, compared with thousandths of a pixel for fixed-point coordinates.

QuickDraw GX provides two groups of mapping functions. The first group allows you to copy and perform standard matrix operations on mappings. With these functions, you can

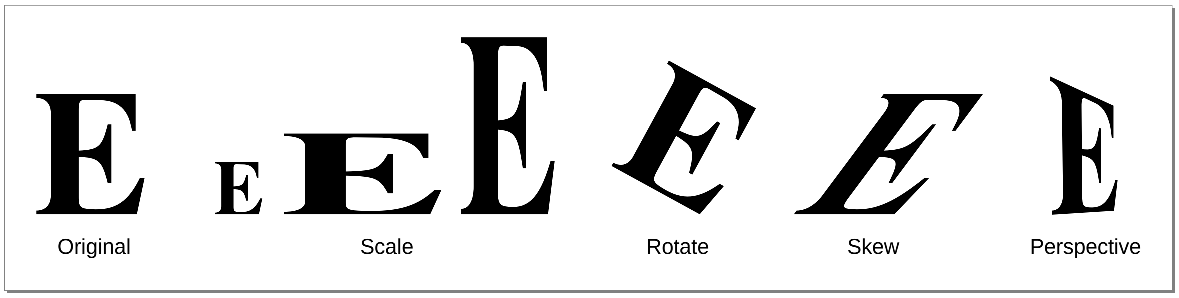
- make a copy of a mapping
- normalize a mapping
- reset a mapping to identity
- invert a mapping
- concatenate (postmultiply) a mapping to another mapping
- apply a mapping to each of a given set of points

The second group allows you to modify how a mapping transforms the objects or coordinate space it is applied to. With these functions, you can

- add translation to mapping
- modify a mapping to specify translation to an absolute location
- add horizontal and vertical scaling to a mapping
- add rotation to a mapping
- add horizontal and vertical skew to a mapping

Figure 8-2 shows an example of how modifying a mapping can modify the scaling, rotation, skewing, and perspective of a shape.

Figure 8-2 Transformation operations with a mapping matrix



Characteristics of a Mapping

QuickDraw GX achieves these two-dimensional transformations of shapes and points on a plane by matrix multiplication of each Cartesian point P by the mapping matrix [T] to generate a transformed point P´.

P(x, y) [T] = P´(x´, y´)

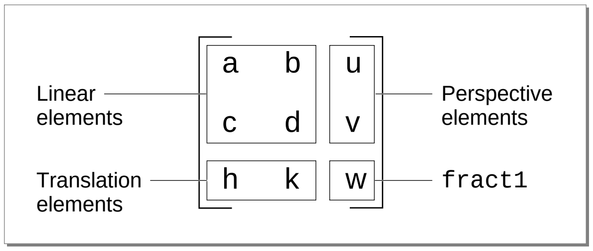
To multiply a two-dimensional point by a three-dimensional matrix, we first expand it to a three-dimensional point (x, y, 1). After multiplication, the resulting point is (x´, y´, z´), which normalizes to (x´/z´, y´/z, 1) or, in two dimensions, (x´/z´, y´/z).

The QuickDraw GX mapping is defined as

```
struct gxMapping { Fixed map[3][3];};
```

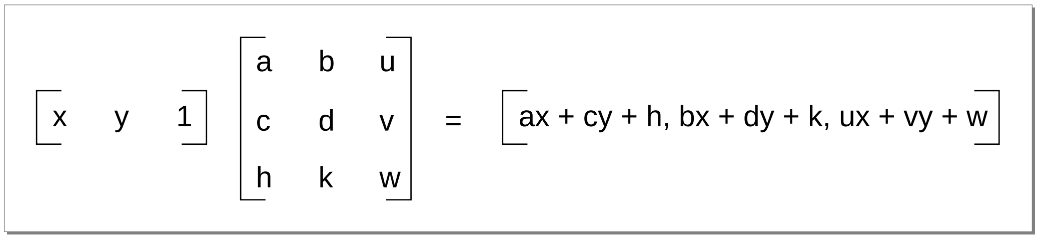
The mapping consists of linear elements a, b, c, and d; perspective elements u and v; translation elements h and k; and the scale factor w, which is commonly set to `fract1`. Although defined as containing only `Fixed` numbers, the rightmost column of the matrix—containing elements u, v, and w—consists of `fract` numbers. Figure 8-3 shows the elements of the matrix in place.

Figure 8-3 Mapping matrix elements



Point P(x, y) is transformed to point P´(x´, y´) by matrix multiplication of the row vector [x y 1] by the mapping matrix to yield the expanded general expression shown in Figure 8-4.

Figure 8-4 Applying a mapping matrix to a point



$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & b & u \\ c & d & v \\ h & k & w \end{bmatrix} = \begin{bmatrix} ax + cy + h, bx + dy + k, ux + vy + w \end{bmatrix}$$

The x and y elements of the transformed vector can be mapped back to the x and y-coordinates by dividing each element by the term ux + vy + w. The resulting general expression for the transformation of point P(x, y) to P´(x´, y´) is shown in Figure 8-5.

Figure 8-5 The point (x, y) as transformed by the mapping matrix

$$\begin{bmatrix} x & y \end{bmatrix} \longrightarrow \begin{bmatrix} \dfrac{ax + cy + h}{ux + vy + w} , & \dfrac{bx + dy + k}{ux + vy + w} \end{bmatrix}$$

A mapping is **normalized** whenever the transformation matrix element w has the value 1. Most QuickDraw GX mapping operations will be automatically normalized. However, mappings that an application generates itself might not be normalized. Subsequent operations with that mapping may be slow.

If a mapping does not specify perspective (that is, if its perspective elements u and v are zero), normalization of the transformation involves dividing the map by the absolute value of w, if possible. If this division is not possible (due to overflow) or if the mapping specifies perspective, normalization involves bit-shifting each element of the mapping to the left. The amount of shift provided by the minimum of the following two operations is selected:

- shift the minimum number of bits so that the absolute value of some element of the mapping is >= `fract1` (compared as `long` values).
- shift the maximum number of bits so that the sum of the absolute values of u and v is <= `fract1` – `fixed1` (compared as `long` values).

The identity mapping, or **identity matrix,** has the unique characteristic that it maps points to the same point. The identity matrix has all diagonal elements equal to 1 and all other matrix elements have the value 0. The identity matrix is shown in Figure 8-6.

Figure 8-6 The identity matrix

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x, y, 1 \end{bmatrix}$$

$$(x, y) \longrightarrow (x, y)$$

The **inverse of a mapping** is the mathematical inverse of the matrix. This means that if you concatenate a mapping with its inverse, you will get the identity matrix.

The rest of this section discusses the use of the mapping functions in modifying the translation, scaling, rotation, and skewing factors in a mapping. It ends with a discussion of how to modify the perspective factors in a mapping. For additional information about the use of mappings in the transform object and in view port and view device objects, see the chapters "Transform Objects" and "View-Related Objects," respectively, in *Inside Macintosh: QuickDraw GX Objects*.

Translation by a Relative Amount

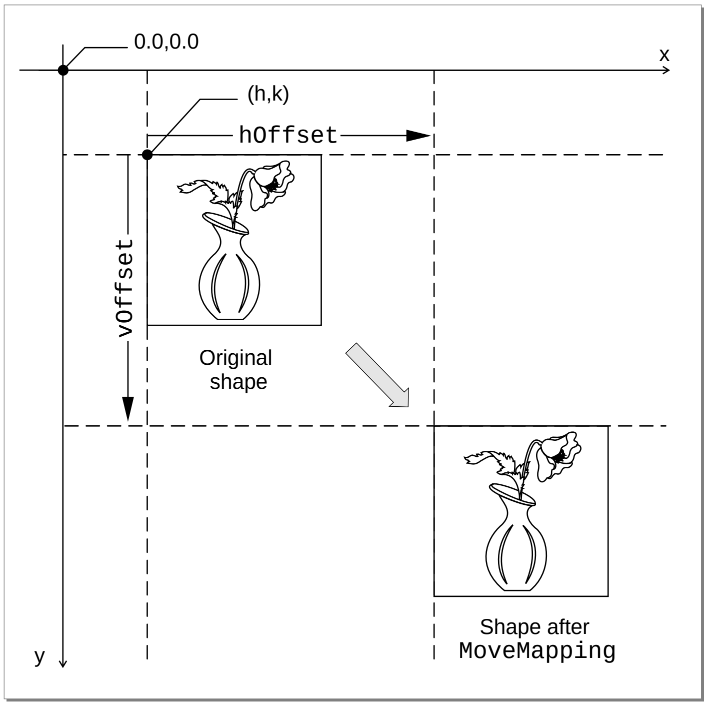
You can use the `MoveMapping` function to make a relative change (in both x and y) to the translation specified by a mapping. Matrix elements h and k control the amount of the translation. Figure 8-7 shows what happens to a mapping *M* when you call `MoveMapping` and specify horizontal and vertical offsets of `hOffset` and `vOffset`. A purely translational matrix is applied to the target mapping, so that the resultant mapping's translation is increased by the specified offsets.

Figure 8-7 Changing the translation specified by a mapping

$$\underset{\textbf{Original mapping}}{M} \times \underset{\textbf{Translation matrix}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \text{hOffset} & \text{vOffset} & 1 \end{bmatrix}} = \underset{\textbf{Transformed mapping}}{M'}$$

Figure 8-8 shows the use of the `MoveMapping` function to provide translation of a mapping by the increments given by the `hOffset` and `vOffset` parameters. The `MoveMapping` function is described on page 8-67.

Figure 8-8 Translation by a relative amount with `MoveMapping`



Translation to a Specified Point

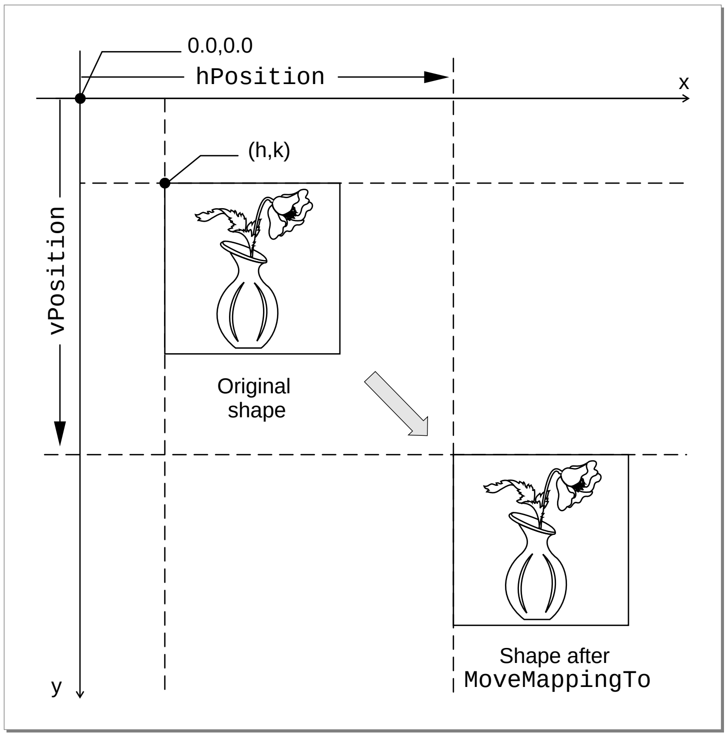
You can specify translation of the origin to a given point by using the `MoveMappingTo` function. Moving the origin means that the point (0, 0) will become the point (h, k) after the mapping is applied to it. Matrix elements h and k again control the amount of translation. Figure 8-9 shows what happens to a mapping *M* when you call `MoveMappingTo` and specify the desired location (`hPosition`, `vPosition`). A relative translation of (–h / w, –k / w) is applied to the target mapping to bring its origin to (0, 0), and then a relative translation of (`hPosition`, `vPosition`) is applied. The resultant mapping ends up with translational values of `hPosition` and `vPosition`.

Figure 8-9 Setting the origin specified by a mapping

$$\underbrace{M}_{\textbf{Original mapping}} \times \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ (\text{hPosition} - \frac{h}{w}) & (\text{vPosition} - \frac{k}{w}) & 1 \end{bmatrix}}_{\textbf{Translation matrix}} = \underbrace{M'}_{\textbf{Transformed mapping}}$$

Figure 8-10 shows the use of the `MoveMappingTo` function to move the origin to a specific location. Note that this figure assumes that the origin of the shape—point (0.0, 0.0) in its geometry—is at its upper left corner. The `MoveMappingTo` function is described on page 8-68.

Figure 8-10 Translation to a specific origin location



Scaling

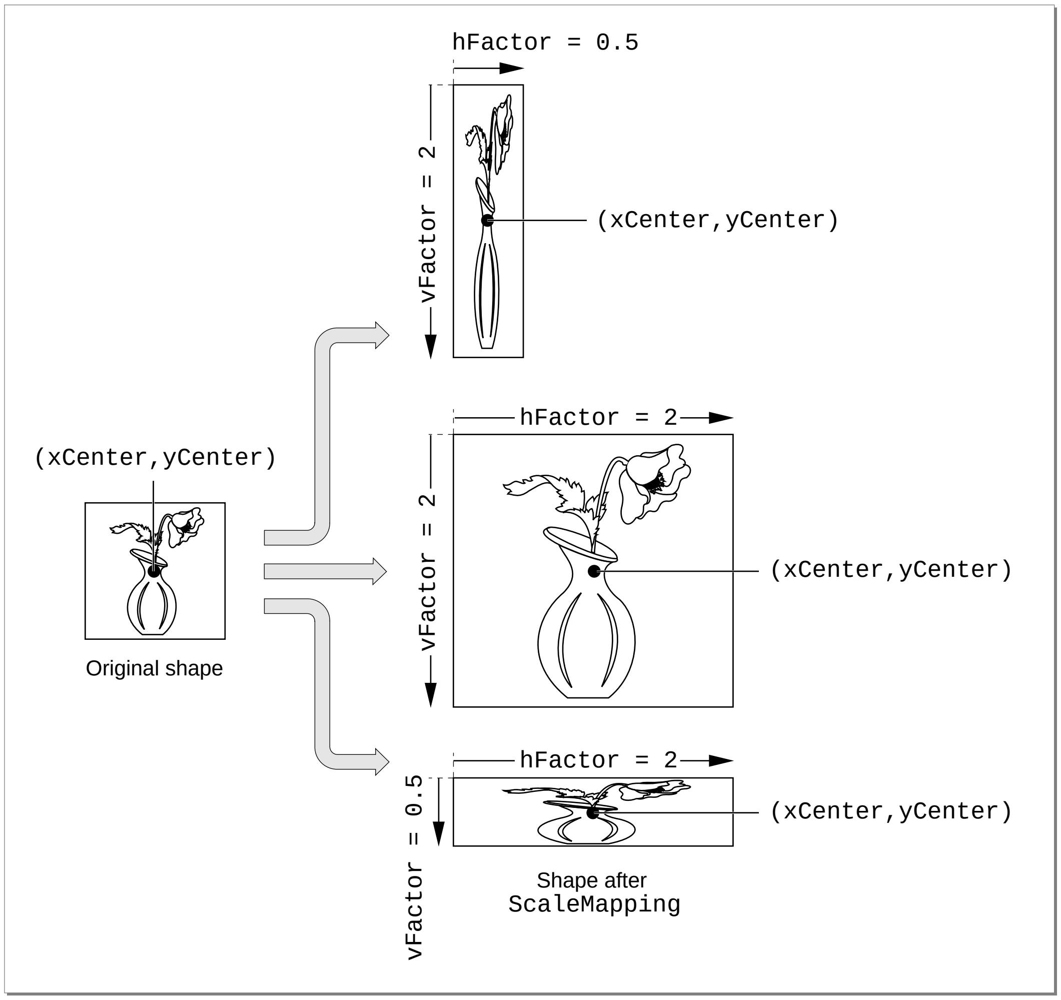
You can use the `ScaleMapping` function to modify the scaling factors in a mapping. Matrix elements a and d in the mapping matrix control the degree of the scaling in the horizontal and vertical directions, respectively. Figure 8-11 shows what happens to a mapping *M* when you call `ScaleMapping` with horizontal and vertical scaling factors of `hFactor` and `vFactor` and a center of scaling at (`xCenter`, `yCenter`). First, a relative translation of –`xCenter` and –`yCenter` moves the center of scaling to (0, 0); then a purely scaling matrix multiplies the scaling by `hFactor` and `vFactor`; finally, another relative translation moves the center of scaling by +`xCenter` and +`yCenter`. In effect, the center of scaling is moved to (0, 0), the scaling is applied, and the scaling center is then moved back to where it was.

Figure 8-11 Changing the amount of scaling specified by a mapping

$$\begin{array}{ccccccccc}
M & \times & \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ -\text{xCenter} & -\text{yCenter} & 1\end{bmatrix} & \times & \begin{bmatrix}\text{hFactor} & 0 & 0\\ 0 & \text{vFactor} & 0\\ 0 & 0 & 1\end{bmatrix} & \times & \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ \text{xCenter} & \text{yCenter} & 1\end{bmatrix} & = & M' \\
\textbf{Original mapping} & & \textbf{Translation matrix} & & \textbf{Scaling matrix} & & \textbf{Translation matrix} & & \textbf{Transformed mapping}
\end{array}$$

Figure 8-12 shows the use of the `ScaleMapping` function scale for various horizontal and vertical factors, in which the center of scaling corresponds to the center of the shape. The `ScaleMapping` function is described on page 8-69.

Figure 8-12 Scaling horizontally and vertically



Note that if `vFactor` equals `hFactor`, scaling is uniform in both directions. If `vFactor` is not equal to `hFactor`, distortion of the image occurs, as shown in Figure 8-12.

The mapping matrix also accommodates **reflection** transformations. If `hFactor` is negative, a reflection about the vertical axis occurs. If `vFactor` is negative, a reflection about the horizontal axis occurs. If both `vFactor` and `hFactor` are negative, a 180° rotation occurs.

Rotation

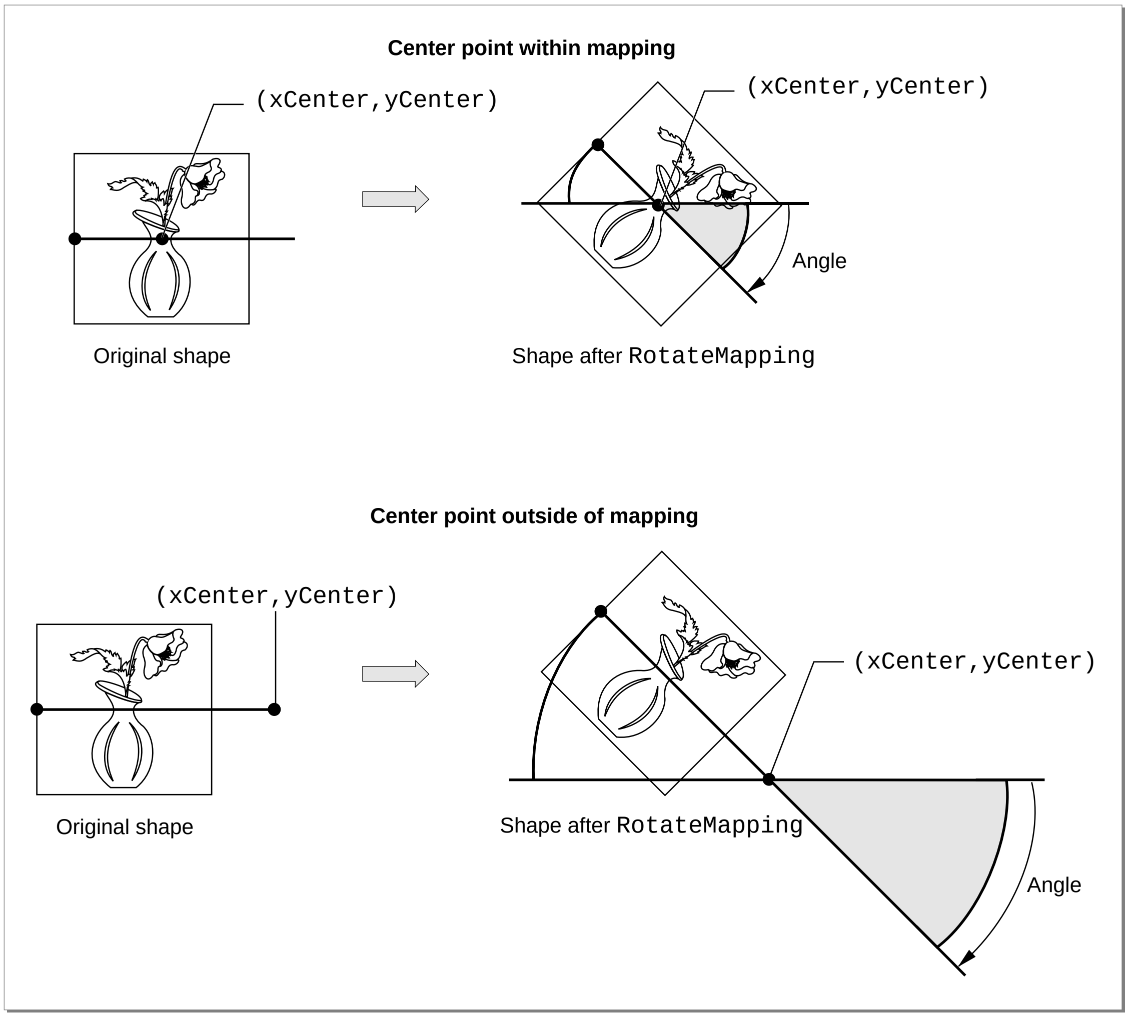
You can use the `RotateMapping` function to modify the rotation specified by a mapping. Matrix elements a, b, c, and d together specify the angle of rotation. Figure 8-13 shows what happens to a mapping *M* when you call `RotateMapping` to rotate by an angle β about a rotational origin of `xCenter` and `yCenter`. First, a relative translation of –`xCenter` and –`yCenter` moves the center of rotation to (0, 0); then a purely rotational matrix adds β to the amount of rotation already specified in the mapping; finally, another relative translation moves the center of rotation by +`xCenter` and +`yCenter`, back to where it was.

Figure 8-13 Changing the degree of rotation specified by a mapping

$$\underset{\textbf{Original mapping}}{M} \times \underset{\textbf{Translation matrix}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\text{xCenter} & -\text{yCenter} & 1 \end{bmatrix}} \times \underset{\textbf{Rotation matrix}}{\begin{bmatrix} \cos(\beta) & \sin(\beta) & 0 \\ -\sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{bmatrix}} \times \underset{\textbf{Translation matrix}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \text{xCenter} & \text{yCenter} & 1 \end{bmatrix}} = \underset{\textbf{Transformed mapping}}{M'}$$

Figure 8-14 shows the use of the `RotateMapping` function to change the rotation of a mapping. Note that positive values of the angle parameter cause clockwise rotation (consistent with y values increasing downward), and note also that changing the center of rotation can significantly change the final position of the rotated objects. The `RotateMapping` function is described on page 8-70.

Figure 8-14 Rotating about different center points


Center point within mapping
(xCenter,yCenter)
(xCenter,yCenter)
Angle
Original shape
Shape after RotateMapping
Center point outside of mapping
(xCenter,yCenter)
(xCenter,yCenter)
Original shape
Shape after RotateMapping
Angle

Skewing

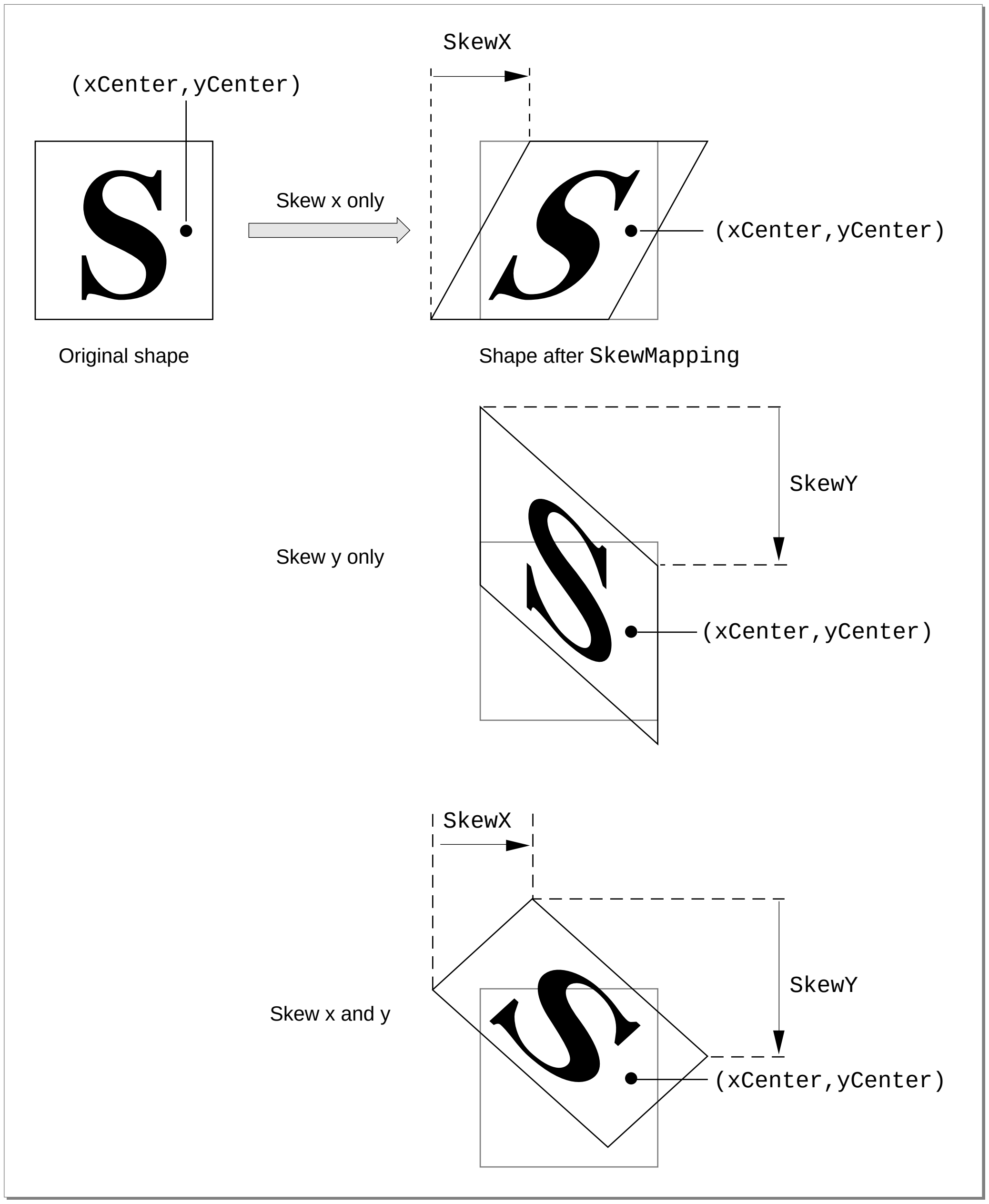
You can use the `SkewMapping` function to modify the skewing imposed by a mapping. Matrix elements b and c control the amount of the skew. Element b controls skew in the y direction and element c controls skew in the x direction. Figure 8-15 shows what happens to a mapping *M* when you call `SkewMapping` with x and y skew factors of `xSkew` and `ySkew`, and a skew origin (the point at which no shearing takes place) of `xCenter` and `yCenter`. First, a relative translation of –`xCenter` and –`yCenter` moves the center of skewing to (0, 0); then a purely skewing matrix modifies the amount of skew already specified in the mapping; finally, another relative translation moves the center of skewing by +`xCenter` and +`yCenter`, back to where it was.

Figure 8-15 Changing the amount of skew specified by a mapping

$$\underset{\textbf{Original mapping}}{M} \times \underset{\textbf{Translation matrix}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\text{xCenter} & -\text{yCenter} & 1 \end{bmatrix}} \times \underset{\textbf{Skew matrix}}{\begin{bmatrix} 1 & \text{ySkew} & 0 \\ \text{xSkew} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}} \times \underset{\textbf{Translation matrix}}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \text{xCenter} & \text{yCenter} & 1 \end{bmatrix}} = \underset{\textbf{Transformed mapping}}{M'}$$

Figure 8-16 shows the use of the `SkewMapping` function to change the skew specified by a mapping. (Note that the skew in the x direction in Figure 8-16 is negative; as y decreases—upward—the amount of shear in the x direction increases.) The `SkewMapping` function is described on page 8-71.

Figure 8-16 Skewing a shape both horizontally and vertically



Perspective

You can manipulate the elements of a mapping to modify its specification of perspective. The matrix elements u, v, and w determine how the perspective will appear when the mapping is applied. The action performed on a point by a mapping whose perspective elements are nonzero is shown in Figure 8-18.

Figure 8-17 Changing the perspective specified by a mapping

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x, y, xu + yv + 1 \end{bmatrix}$$

$$(x, y) \longrightarrow \left(\frac{x}{xu + yv + 1} , \frac{y}{xu + yv + 1} \right)$$

There is currently no QuickDraw GX function that modifies the perspective-controlling elements of a mapping for you. If you wish to create perspective, you need to modify the individual matrix elements directly.

Using QuickDraw GX Mathematics

This section describes how you can use QuickDraw GX number formats, macros, and functions in your application.

Converting Number Formats

You can use QuickDraw GX macros to convert between `Fixed`, `fract`, integer, floating-point, and `gxColorValue` number formats. Macros are also provided to round, truncate, and compute the square root of a fixed-point number.

For example, you can use the `IntToFixed` macro to convert an integer to a `Fixed` format and you can use the `FloatToFixed` macro to convert from a floating-point format to a `Fixed` format. The functionality of the `FloatToFixed` macro is also provided as the shortened `fl` macro. The functionality of the `IntToFixed` macro is also provided as the shortened `ff` macro.

The `ff` macro is especially useful when you are coding specific points in your application. For example, it's easier to define a line in your application using the `ff` macro:

```
gxLine lineData = {ff(25), ff(25) , ff(125), ff(125)};
```

than to use the equivalent, but much longer `IntToFixed` macro:

```
gxLine lineData = {IntToFixed(25), IntToFixed(25),
                   IntToFixed(125), IntToFixed(125)};
```

For constants, using `ff` is faster and more efficient than using `fl`, because `ff` is evaluated at compile time, whereas `fl` is evaluated at run time.

The `IntToFixed` macro is described on page 8-37. The `FloatToFixed` macro is described on page 8-39. The `fl` macro is described on page 8-39. The `ff` macro is described on page 8-38.

Performing Fixed-Point Operations

You can use QuickDraw GX functions to provide operations on `Fixed`, `long`, `fract` and `wide` numbers. The equivalent QuickDraw GX fixed-point functions for functions in the Macintosh Mathematical Utilities is shown in Table 8-2.

Table 8-2 QuickDraw GX and Macintosh Toolbox fixed-point functions

QuickDraw GX	Macintosh Mathematical Utilities
`FractDivide`	`FracDiv`
`FractMultiply`	`FracMul`
`FractSquareRoot`	`FracSqrt`
`FixedDivide`	`FixDiv`
`FixedMultiply`	`FixMul`
`WideMultiply`	`LongMul`

The Macintosh Mathematical Utilities are described in *Inside Macintosh: Operating System Utilities.*

Some functions combine multiple functions into a single function to increase calculation speed over that obtained using sequential function calls. For example, the `FractSineCosine` function returns both the sine and cosine of an angle.

Some functions support the use of 64-bit numbers to increase the accuracy of calculations. For example, the `WideAdd` function returns the 64-bit sum of two 64-bit numbers, and the `WideDivide` function returns the quotient of a 64-bit number and a 32-bit number. The `MultiplyDivide` function uses a 64-bit intermediate result to increase accuracy of the calculation and to prevent premature overflow.

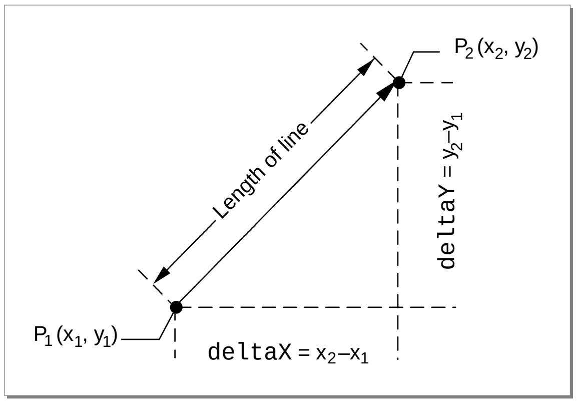
The `MultiplyDivide`, `Magnitude`, and `VectorMultiplyDivide` functions are derivatives of other functions. For example, `MultiplyDivide (x, y, z)` is the same as:

```
wide temp;
WideDivide (WideMultiply(x, y, &temp), z, 0)
```

The final argument of 0 specifies that the returned number will be rounded with no remainder.

You can use the `Magnitude` function to determine the magnitude (length) of a two-dimensional vector, or the distance between two points on a plane. Figure 8-18 shows the use of function parameters `deltaX` and `deltaY`.

Figure 8-18 Determining the length of a line with the `Magnitude` function

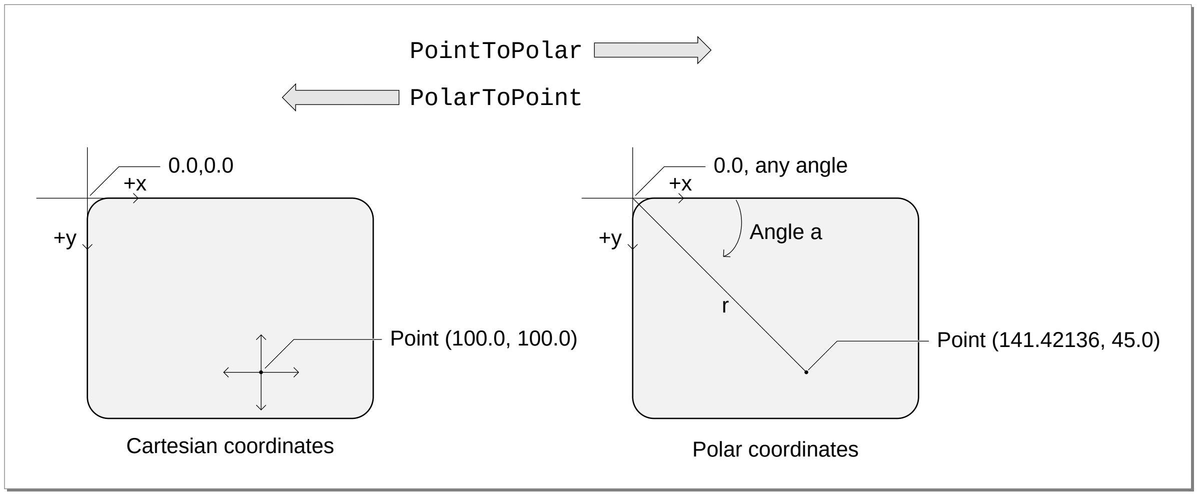


Functions that provide arithmetic operations on fixed-point numbers are described in the section “Fixed-Point Operations” beginning on page 8-42. Functions that provide operations on `wide` numbers are described in the section “Operations on wide Numbers” beginning on page 8-49. The `Magnitude` function is described on page 8-45.

Converting Between Cartesian and Polar Coordinates

You can use QuickDraw GX functions to convert between Cartesian and polar coordinates. The `PolarToPoint` function converts a point in polar coordinates to Cartesian coordinates, (r, a) to (x, y). The `PointToPolar` function converts a point in Cartesian coordinates to polar coordinates, (x, y) to (r, a). The `gxPolar` point (r, a) corresponds to the `gxPoint` point ($r \times \cos(a)$, $r \times \sin(a)$). Since $r^2 = x^2 + y^2$ and $\tan(a) = y \,/\, x$, the `gxPoint` structure (100, 100) corresponds to the `gxPolar` structure (141.42136, 45). Figure 8-19 shows the Cartesian coordinate of point (100, 100) and the polar coordinate of identical point (141.42136, 45).

Figure 8-19 Converting between Cartesian and polar coordinates



The Cartesian and polar coordinate systems are described in the section “Cartesian and Polar Coordinate Conversion” beginning on page 8-10. The `PolarToPoint` function is described on page 8-56. The `PointToPolar` function is described on page 8-57.

Performing Vector Operations

You can use the `VectorMultiply` function to obtain the dot product of two vectors with 64-bit accuracy. The function takes six parameters: the first parameter specifies the number of long numbers to multiply, and the third and fifth parameters specify the step size to use when walking the arrays to which the second and fourth parameters point.

For example:

`VectorMultiply(4,a,1,b,2,&c)` sets the `wide` number pointed to by the parameter `c` to the following value:

```
a[0] * b[0] +a [1] * b[2] +a[2] * b[4] + a[3] * b[6]
```

If the count is negative, the sign of the terms in the dot product are alternated. `VectorMultiply(–4,a,1,b,2,&c)` sets the `wide` parameter `c` to the following value and the result is returned in `c`:

`a[0] * b[0] – a[1] * b[2] + a[2] * b[4] – a[3] * b[6]`

You can also use `VectorMultiply` to determine the cross-product of a pair of vectors, as in Listing 8-1.

Listing 8-1 Calculating a cross-product with `VectorMultiply`

```
gxPoint *CrossProduct(const gxPoint *a, gxPoint *b, )
{
   wide temp;
   WideShift(VectorMultiply(-2, &a->x, 1, &b->y, -1, &temp), 16);
}
```

You can also use `VectorMultiply` to work with mappings. Listing 8-2 is a sample function that applies a mapping to a single point.

Listing 8-2 Applying a mapping to one point

```
gxPoint *MapPoint(const gxMapping *map, gxPoint *pt)
{
   fixed temp[3] = { 0, 0, fixed1 };
   *(gxPoint *)temp = *pt;
   wide dot;
   fixed p = WideShift(VectorMultiply(3, temp, 1, &map[0][2],
                      3, &dot), 30);
   pt->x = WideDivide(VectorMultiply(3, temp, 1, &map[0][0],
                      3, &dot), p, nil);
   pt->y = WideDivide(VectorMultiply(3, temp, 1, &map[0][1],
                      3, &dot), p, nil);
   return pt;
}
```

The `VectorMultiply` function is described on page 8-54. Functions that perform vector operations are described in the section “Vector Operations” beginning on page 8-54.

Shifting the Bits of a wide Number

You can use the `WideShift` function to shift bits in a `wide` format number. Listing 8-3 shows how to use the `WideShift` function to provide a fixed-point version of the `VectorMultiply` function.

Listing 8-3 Using the `WideShift` function to create a fixed-point `VectorMultiply` function

```
Fixed VectorFixMul(long count, Fixed *vector1, long step1,
                     Fixed *vector2, long step2)
{
   wide temp;
   return WideShift(VectorMultiply(count, vector1, step1,
                     vector2, step2, &temp), 16)->lo;
}
```

Listing 8-4 shows how to use the `WideShift` function in a multiplication function for a fixed-point number with a fixed-point bias of 6 bits.

Listing 8-4 Using the `WideShift` function in a fixed-point multiplication function

```
long MultiplyDot6(long a, long b)
{
   wide temp;
   return (long)WideShift(WideMultiply(a, b, &temp), 6)->lo;
}
```

Listing 8-5 shows how to use the `WideShift` function in a division function for a fixed-point number with a fixed-point bias of 6 bits. Listing 8-6 gives an alternative, but equivalent, approach.

Listing 8-5 Using the `WideShift` function to create a fixed-point division function

```
long DivideDot6(long a, long b)
{
   wide temp;
   temp.hi = (temp.lo = a) < 0 ? -1 : 0;  /* sign extend a */
   return WideDivide(WideShift(&temp, -6), b, 0);
}
```

Listing 8-6 shows how to use the `WideShift` function for a second fixed-point division function with a fixed-point bias of 6 bits. Listing 8-5 gives an alternative, but equivalent, approach.

Listing 8-6 Using the `WideShift` function to create a second fixed-point division function

```
long DivideDot6(long a, long b)
{
   wide temp;
   temp.hi = a;
   temp.lo = 0;
   return WideDivide(WideShift(&temp, 26), b, 0);
}
```

Determining the Highest Order Bit of a wide Number

You can use the `WideScale` function to obtain the bit number of the highest order bit in the absolute value of a `wide` number. Listing 8-3 shows how to use the `WideScale` function in a function that multiplies two numbers in `long` format. If the product is too big to fit in a `long`, the function shifts the product so that it fits into a `long` and returns the bit shift. This operation can be termed *pseudo-floating-point*.

Listing 8-7 Using the `WideScale` function to create a pseudo-floating-point function

```
long FloatMul(long a, long b, long *product)
{
   wide temp;
   long shift = WideScale(WideMultiply(a, b, &temp)) - 30;
   if (shift > 0)
      WideShift(&temp, shift);
   else
      shift = 0;
   if (product) *product = temp.lo;
   return shift;
}
```

The `WideScale` function is described on page 8-53.

Generating Random Numbers

You can use the QuickDraw GX random number functions to return a sequence of random numbers. The `RandomBits` function generates random integers in the range of 0 to $2^{count} - 1$, where *count* is the number of bits in the integer to be generated by the random number generator.

The `SetRandomSeed` function allows you to use a seed other than the default seed. If the `SetRandomSeed` function is not used, the initial seed will always be 0. You can use the `GetRandomSeed` function to return the value of the current seed.

Listing 8-8 is a sample function that generates an unsigned random number between zero and the value passed in the `limit` parameter. It uses the `RandomBits` function, and it also uses the `WideMultiply` function, correcting for the fact that `WideMultiply` works with signed long integers whereas this random generator uses unsigned longs.

Listing 8-8 A random number generator

```
unsigned long RandomLong(unsigned long limit)
{
   wide temp;
   unsigned long random = RandomBits(32, 0);

   /* This treats random and limit as signed */
   WideMultiply(random, limit, &temp);
   if ((long)limit < 0)
      temp.hi += random;   /* correct for the “sign” of limit */
   if ((long)random < 0)
      temp.hi += limit;    /* correct for the “sign” of random */
   return temp.hi;
}
```

The general topic of random numbers and the functions you use to generate them generation are discussed in the section “Random Number Generation” beginning on page 8-58.

Analyzing the Bits in a Number

You can use the `FirstBit` function to determine the highest bit number that is set in a 32-bit number. The following examples demonstrate the use of this function with the parameter `x`:

If `x` is 1, the highest order bit that is set is bit number 0,
so `FirstBit(1)` = 0, as shown below.

```
FirstBit(00000000000000000000000000000001) = 0x0000
```

If `x` is 2, the highest order bit that is set is bit number 1, so `FirstBit(2)` = 1, as shown below.

```
FirstBit(00000000000000000000000000000010) = 0x0001
```

If `x` is 3, the highest order bit that is set is bit 1, so `FirstBit(3)` = 1, as shown below.

```
FirstBit(00000000000000000000000000000011) = 0x0001
```

If no bits in the number are set, `FirstBit` returns a value of –1.

You can also use `FirstBit` to find the *last* (= lowest-order) bit that is set in a number. Listing 8-9 is an example of such a function.

Listing 8-9 Determining the lowest bit of a number

```
short LastBit(unsigned long x)
{
   if (x == 0)
      return 32;
   return FirstBit(x & -x);
}
```

The `FirstBit` function is described on page 8-62.

Resetting a Mapping

You can use the `ResetMapping` function to reset a mapping. The following code example first uses the `ResetMapping` function to initialize the destination to the identity matrix, and then uses `RotateMapping` to calculate a resultant mapping that rotates by a given angle about a specified center.

```
gxMapping *RotationMap(gxMapping *dest, Fixed angle,
                        gxPoint *center)
{
    return RotateMapping(ResetMapping(dest), angle,
                           center->x,center->y);
}
```

The `ResetMapping` function is described on page 8-64.

QuickDraw GX Mathematics Reference

This section describes the constants, data types, structures, macros, and functions that relate to QuickDraw GX mathematics.

Constants and Data Types

This section describes the constants and data types that are used to define QuickDraw GX mathematical number formats and the transformation matrix.

Number Formats and Constants

QuickDraw GX provides `Fixed`, `fract`, and `gxColorValue` number formats. Polar coordinates are defined by the `gxPolar` structure. A structure consisting of two `long` values defines the `wide` number format.

```
typedef long fract;

typedef unsigned short gxColorValue;

struct gxPolar {
   Fixed radius;
   Fixed angle;
};

struct wide {
   long hi;
   unsigned long lo;
};
```

For convenience, QuickDraw GX provides constants for the value 1.0 for `Fixed`, `fract`, and `gxColorValue` types:

```
#define fixed1        ((Fixed) 0x00010000)
#define fract1        ((fract) 0x40000000)
#define gxColorValue1 ((gxColorValue) 0xFFFF)
```

QuickDraw GX also provides constants for the largest and smallest possible values for `Fixed` and `fract` numbers:

```
#define gxPositiveInfinity ((Fixed) 0x7FFFFFFF)
#define gxNegativeInfinity ((Fixed) 0x80000000)
```

The Mapping Structure

QuickDraw GX defines a transformation matrix with the `gxMapping` structure:

```
struct gxMapping {
   Fixed map[3][3];
};
```

Field descriptions

`map`	A 3 × 3 array of `Fixed` numbers whose values determine the translation, scaling, rotation, skewing, and perspective operations that can be applied to two-dimensional data. Although defined as containing only `Fixed` numbers, the rightmost column of the matrix consists of `fract` numbers. Furthermore, element `[3][3]` is commonly set to `fract1`.

The use of the mapping matrix is described further in the section “Transformation Operations With Mappings” beginning on page 8-12.

Number-Conversion Macros

QuickDraw GX defines macros for conversion between fixed-point number formats. It also provides macros to round and truncate numbers, as well as a macro that uses the `FractSquareRoot` function to compute the square root of a `Fixed` number.

Format Conversions

The macros in this section convert between `Fixed`, `fract`, integer, floating-point, and `gxColorValue` numbers.

FixedToFract

You can use the `FixedToFract` macro to convert a fixed number to a `fract` number.

```
#define FixedToFract(a) ((fract) (a) << 14)
```

`a`	A `Fixed` number to be converted to a `fract` number, $-2 \le a < 2$.
macro result	A `fract` number having the same value as the fixed number.

FractToFixed

You can use the `FractToFixed` macro to convert a `fract` number to a `Fixed` number.

```
#define FractToFixed (a) ((Fixed) (a) + 8192L >> 14)
```

`a`	A `fract` number to be converted to a `Fixed` number.
macro result	A `Fixed` number having the closest value to the `fract` number.

FixedToInt

You can use the `FixedToInt` macro to convert a `Fixed` number to an integer.

```
#define FixedToInt(a) ((short) ((Fixed) (a) + fixed1/2 >> 16))
```

`a`	A `Fixed` number to be converted to an integer.
macro result	An integer having the closest value to the `Fixed` number.

IntToFixed

You can use the `IntToFixed` macro to convert an integer to a `Fixed` number.

```
#define IntToFixed(a) ((Fixed)(a) << 16)
```

`a`	An integer to be converted to a `Fixed` number.
macro result	A `Fixed` number having the same value as the integer.

SPECIAL CONSIDERATIONS

QuickDraw GX also defines a shorthand version of this macro. `IntToFixed(a)` can also be coded as `ff(a)`.

SEE ALSO

The `ff` macro is described next.

ff

You can use the `ff` macro to convert an integer to a `Fixed` number.

```
#define ff(a) ((Fixed)(a) << 16)
```

`a`	An integer to be converted to a `Fixed` number.
macro result	A `Fixed` number having the same value as the integer.

DESCRIPTION

The `ff` macro converts an integer `a` to a `Fixed` number. This macro name is shorthand notation for the `IntToFixed` macro, and provides identical functionality.

SEE ALSO

For an example of how to use the `ff` macro, see the section “Converting Number Formats” beginning on page 8-26.

The `IntToFixed` macro is described in the previous section.

FixedToFloat

You can use the `FixedToFloat` macro to convert a `Fixed` number to a floating-point number.

```
#define FixedToFloat(a) ((float)(a) / fixed1)
```

`a`	A `Fixed` number to be converted to a floating-point number.
macro result	A floating-point number having the same value as the `Fixed` number.

FloatToFixed

You can use the `FloatToFixed` macro to convert a floating-point number to a `Fixed` number.

```
#define FloatToFixed(a) ((Fixed)((float) (a) * fixed1))
```

`a`	A floating-point number to be converted to a `Fixed` number.
macro result	The closest `Fixed` number to the floating-point number.

SPECIAL CONSIDERATIONS

QuickDraw GX also defines a shorthand version of this macro. The `FloatToFixed` macro can also be coded as `fl(a)`.

SEE ALSO

The `fl` macro is described next.

fl

You can use the `fl` macro to convert a floating-point number to a `Fixed` number.

```
#define fl(a) ((Fixed)((float) (a) * fixed1))
```

`a`	A floating-point number to be converted to a `Fixed` number.
macro result	The closest `Fixed` number to the floating-point number.

DESCRIPTION

The `fl` macro converts a floating-point number `a` to a `Fixed` number. This macro name is shorthand notation for the `FloatToFixed` macro, and provides identical functionality.

SEE ALSO

The `FloatToFixed` macro is described in the previous section.

FractToFloat

You can use the `FractToFloat` macro to convert a `fract` number to a floating-point number.

```
define FractToFloat(a) ((float)(a)/fract1)
```

`a`	A `fract` number to be converted to a floating-point number.
macro result	A floating-point number having the closest value to the `fract` number.

FloatToFract

You can use the `FloatToFract` macro to convert a floating-point number to a `fract` number.

```
define FloatToFract(a) ((fract)((float)(a)*fract1))
```

`a`	A floating-point number to be converted to a `fract` number.
macro result	A `fract` number having the closest value to the floating-point number.

ColorToFract

You can use the `ColorToFract` macro to convert a `gxColorValue` number to a `fract` number.

```
#define ColorToFract(a) (((fract)(a)<<14) + ((fract)(a) +2 >>2))
```

`a`	A `gxColorValue` number to be converted to a `fract` number.
macro result	A `fract` number having the same value as the `gxColorValue` number.

FractToColor

You can use the `FractToColor` macro to convert a `fract` number to a `gxColorValue` number.

```
#define FractToColor(a) ((gxColorValue)((a)-((a)>>16)+8191>>14))
```

`a`	A `fract` number to be converted to a `gxColorValue` number.
macro result	The closest `gxColorValue` number to the `fract` number.

Rounding, Truncating, and Square Root Operations

The macros in this section round, truncate, and determine the square root of fixed numbers.

FixedRound

You can use the `FixedRound` macro to round a `Fixed` number to its nearest integer.

```
#define FixedRound(a) ((short) ((Fixed)(a) + fixed1/2 >> 16))
```

`a`	The number to be rounded.
macro result	The closest integer to the `Fixed` number.

FixedTruncate

You can use the `FixedTruncate` macro to obtain an integer that is the greatest integer that is not greater than the given `Fixed` number.

```
#define FixedTruncate(a) ((short)((Fixed)(a) >> 16))
```

`a`	The number that is to be truncated.
macro result	The largest integer that is not greater than the `Fixed` number.

FixedSquareRoot

You can use the `FixedSquareRoot` macro to determine the square root of a fixed number.

```
#define FixedSquareRoot(a) ((Fixed)FractSquareRoot(a) + 64 >>7)
```

`a`	The number for which the square root is to be determined.
macro result	The square root of the number.

Mathematical Functions

This section describes the QuickDraw GX functions you can use to perform

- fixed-point operations
- `wide` number operations
- vector operations
- mapping operations
- random number generation
- bit analysis

Fixed-Point Operations

QuickDraw GX provides an assortment of fixed-point mathematical functions that you can use in your application.

FixedMultiply

You can use the `FixedMultiply` function to return the product of two numbers.

```
Fixed FixedMultiply (Fixed multiplicand, Fixed multiplier);
```

`multiplicand`	The number to be multiplied by the multiplier.
`multiplier`	The number by which the multiplicand is to be multiplied.
function result	The product of two numbers.

DESCRIPTION

The `FixedMultiply` function multiplies two fixed numbers. The format of the `Fixed` number returned depends on the respective number formats of the multiplicand and multiplier. The operation has a bias of 16 bits; in general, the bias of the resulting number is the sum of the biases of the input numbers, shifted right by 16 bits. If either the multiplicand or the multiplier is `Fixed`, the result of the `FixedMultiply` function will be the same fixed-point format as the other parameter (`long`, `Fixed`, or `fract`).

Table 8-3 shows the bias of the product for different combinations of formats. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it.

Table 8-3 `FixedMultiply` product bias

	long	**fixed**	**fract**
long	---	long	---
fixed	long	fixed	fract
fract	---	fract	---

SPECIAL CONSIDERATIONS

The `FixedMultiply` function does not pin its result in the case of an overflow; the result returned is modulo 65,536.

FixedDivide

You can use the `FixedDivide` function to return the quotient of a dividend and divisor.

```
Fixed FixedDivide (Fixed dividend, Fixed divisor);
```

`dividend`	The number to be divided.
`divisor`	The number by which the dividend is to be divided.

function result The quotient of the dividend and the divisor.

DESCRIPTION

The `FixedDivide` function divides the `dividend` parameter by the `divisor` parameter and returns the quotient. The format of the fixed number returned depends on the respective number formats of the `dividend` and `divisor` parameters. The operation has a bias of 16 bits; in general, the bias of the resulting number is the difference between the biases of the input numbers, shifted left by 16 bits. If the `divisor` parameter is fixed, then the result will be the same fixed-point format as the dividend. If both the dividend and divisor are the same fixed-point format, the result will be in `Fixed` format.

Table 8-4 shows the bias for the quotient of two numbers that are of dissimilar formats. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it.

Table 8-4 `FixedDivide` quotient bias

Denominator	**Numerator**		
	long	fixed	fract
long	fixed	---	---
fixed	long	fixed	fract
fract	---	---	fixed

SPECIAL CONSIDERATIONS

In the case of overflow, `FixedDivide` pins its result to either the `gxPositiveInfinity` or `gxNegativeInfinity` constant.

MultiplyDivide

You can use the `MultiplyDivide` function to multiply two numbers and divide by a third number.

```
long MultiplyDivide (long multiplicand, long multiplier,
                     long divisor);
```

`multiplicand`	The number to be multiplied by the multiplier.
`multiplier`	The number by which the multiplicand is multiplied.
`divisor`	The number by which the product is divided.

function result The quotient of the product of two numbers and the divisor.

DESCRIPTION

The `MultiplyDivide` function calculates the quotient of the product of two numbers (parameters `multiplicand` and `multiplier`) and a divisor.

The function uses a 64-bit intermediate result to maintain accuracy and to prevent premature overflow. The parameters do not need to all be the same fixed-point format. The operation has a bias of 0 bits; if the divisor is the same format as the multiplier, the result is the same format as the multiplicand. If the divisor is the same format as the multiplicand, the result is in the same format as the multiplier.

SPECIAL CONSIDERATIONS

In the case of overflow, `MultiplyDivide` pins its result to either the `gxPositiveInfinity` or `gxNegativeInfinity` constant.

Magnitude

You can use the `Magnitude` function to obtain the magnitude of a vector, the length of a line, or the distance between two points.

```
unsigned long Magnitude (long deltaX, long deltaY);
```

`deltaX`	The difference in the x-coordinates of the vector's end points.
`deltaY`	The difference in the y-coordinates of the vector's end points.
function result	The magnitude of the vector.

DESCRIPTION

The `Magnitude` function returns ($\texttt{deltaX}^2 + \texttt{deltaY}^2)^{1/2}$, the Euclidean distance between two points whose x-coordinates are separated by `deltaX` and whose y-coordinates are separated by `deltaY`.

The fixed-point format of the result is the same as the fixed-point format for both of the parameters. Make sure that the two parameters use the same format.

FractSineCosine

You can use the `FractSineCosine` function to obtain both the sine and cosine of an angle measured in degrees.

```
fract FractSineCosine (Fixed degrees, fract *cosine);
```

`degrees`	The angle in degrees for which the cosine and sine are required.
`cosine`	A pointer to the location where the cosine of the angle is required.

function result The sine of the angle specified.

DESCRIPTION

Given the `degrees` parameter in degrees, the `FractSineCosine` function returns the sine as the function result and the cosine in the `cosine` parameter. Values for the `degrees` parameter are specified in degrees, not radians. The range of the angle is –32,768 to +32,769.999 degrees.

FractSquareRoot

You can use the `FractSquareRoot` function to calculate the square root of a `fract` number.

```
fract FractSquareRoot (fract source);
```

`source`	The number for which the square root is required.

function result The square root of the `fract` number.

DESCRIPTION

The `FractSquareRoot` function returns the square root of the `fract` number specified by the `source` parameter. The number is interpreted as unsigned and in the range 0 through $4 - (2^{-30})$. This means that bit 31 has a weight of 2, instead of –2. The result is an unsigned number in the range of 0 through 2.

FractCubeRoot

You can use the `FractCubeRoot` function to calculate the cube root of a `fract` number.

```
fract FractCubeRoot (fract source);
```

`source`	The `fract` number for which the cube root is required.
function result	The cube root of the `fract` number. This number is a signed value.

DESCRIPTION

The `FractCubeRoot` function returns the cube root of a `fract` number.

FractMultiply

You can use the `FractMultiply` function to calculate the product of two numbers.

```
fract FractMultiply (fract multiplicand, fract multiplier);
```

`multiplicand`	The number to be multiplied by the multiplier.
`multiplier`	The number by which the multiplicand is to multiplied.
function result	The product of two numbers.

DESCRIPTION

The `FractMultiply` function calculates the product of two numbers, specified in the `multiplicand` and `multiplier` parameters. If the parameters are a and b, the product a × b is returned.

The format of the number returned depends on the respective number formats of the `multiplicand` and `multiplier` parameters. The operation has a bias of 30 bits; in general, the bias of the resulting number is the sum of the biases of the input numbers, shifted right by 30 bits. Thus if either the `multiplicand` or `multiplier` parameter is `fract`, then the result is the same fixed-point format as the other argument.

Table 8-5 shows the bias of the FractMultiply result. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it

Table 8-5 `FractMultiply` result bias

	long	**fixed**	**fract**
long	**---**	**---**	long
fixed	**---**	**---**	fixed
fract	long	fixed	fract

SPECIAL CONSIDERATIONS

`FractMultiply` does not pin its result in the case of an overflow; the result returned is modulo 4.

FractDivide

You can use the `FractDivide` function to return the quotient of a dividend and divisor.

```
fract FractDivide (fract dividend, fract divisor);
```

`dividend`	The number to be divided.
`divisor`	The number by which the dividend is to be divided.
function result	The quotient of two numbers.

DESCRIPTION

The `FractDivide` function divides the `dividend` parameter by the `divisor` parameter and returns the quotient. If the `dividend` parameter is a and the `divisor` parameter is b, the quotient a / b is returned.

The format of the number returned depends on the respective number formats of the dividend and divisor. The operation has a bias of 30 bits; in general, the bias of the resulting number is the difference between the biases of the input numbers, shifted left by 30 bits. Thus if the divisor is a `fract`, the result is the same format as the dividend. If the `divisor` and the `dividend` parameters are the same format, the result is in `fract` format, as shown inTable 8-6. The dashed line indicates that the resulting bias is not equivalent to long, fixed, or fract. Use the rules of the operation to determine it.

Table 8-6 `FractDivide` result bias

Denominator	Numerator		
	long	fixed	fract
long	fract	---	---
fixed	---	fract	---
fract	long	fixed	fixed

SPECIAL CONSIDERATIONS

In the case of division of a large number by a very small number,the `FractDivide` function pins its result to either the `gxPositiveInfinity` or the `gxNegativeInfinity` constant.

Operations on wide Numbers

QuickDraw GX provides an assortment of 64-bit mathematical functions for your use. You can use `wide` functions to increase the accuracy of calculations.

WideAdd

You can use the `WideAdd` function to add two `wide` numbers.

```
wide *WideAdd(wide *target, const wide *source);
```

`target`	A pointer to the number to be added to. On return, contains the sum of the two numbers.
`source`	A pointer to the number that is to be added to the target number.
function result	A pointer to the result (also a pointer to the target number).

DESCRIPTION

The `WideAdd` function adds the `wide` number in the `source` parameter to the `wide` number in the `target` parameter and returns the target pointer.

WideSubtract

You can use the `WideSubtract` function to subtract one `wide` number from another.

```
wide *WideSubtract(wide *target, const wide *source);
```

`target`	A pointer to the number to be subtracted from. On return, contains the difference between the two numbers.
`source`	A pointer to the number that is to be subtracted from the number at target.
function result	A pointer to the target number.

DESCRIPTION

The `WideSubtract` function subtracts the source number from the target number and returns a pointer to the target number.

WideNegate

You can use the `WideNegate` function to change a `wide` number to its negative.

```
wide *WideNegate(wide *target);
```

`target`	A pointer to the number to be negated. On return, contains the negated number.
function result	A pointer to the target number.

WideShift

You can use the `WideShift` function to shift bits in a `wide` number.

```
wide *WideShift(wide *target, long shift);
```

`target`	A pointer to the number for which the bits are to be shifted. On return, contains the shifted number.
`shift`	The number of bits by which the target is to be shifted to the right.

function result A pointer to the target number.

DESCRIPTION

The shift direction is to the right (a decrease in magnitude) if the `shift` parameter is greater than 0, and to the left if the `shift` parameter is less than 0. The result of a right shift is rounded.

WideMultiply

You can use the `WideMultiply` function to calculate the `wide` product of two `long` numbers.

```
wide *WideMultiply(long multiplicand, long multiplier,
                     wide *target);
```

`multiplicand`	The number to be multiplied by the multiplier.
`multiplier`	The number by which the multiplicand is to be multiplied.
`target`	A pointer to the location where the product of the two numbers is to be stored.

function result A pointer to the target value, which holds the result.

DESCRIPTION

The operation has a bias of 0 bits. The bias of the result is the sum of the biases of the inputs.

WideDivide

You can use the `WideDivide` function to calculate the `long` quotient and `long` remainder for a `wide` dividend and `long` divisor.

```
long WideDivide(const wide *dividend, long divisor,
                long *remainder);
```

`dividend`	A pointer to the `wide` number to be divided.
`divisor`	The number by which the dividend is to be divided.
`remainder`	A pointer to a location to store the remainder of the division.
function result	The quotient of the division.

DESCRIPTION

The `WideDivide` function divides the dividend by the divisor and returns the quotient in the function result and the remainder in the `long` number pointed to by the `remainder` parameter. If the dividend is a and the divisor is b, the quotient a / b is returned with a remainder. The operation has a bias of 0 bits; the bias of the result is the difference between the biases of the dividend and the divisor. The bias of the remainder is the same as the bias of the dividend.

If an overflow occurs, the result is pinned to the closest infinity and the remainder is set to `gxNegativeInfinity` (an impossible remainder).

If the `remainder` parameter is `nil`, no remainder is returned and the `WideDivide` function returns a rounded quotient. Passing `(long *)-1` in the `remainder` parameter is the same as passing `nil` except in the case of an overflow, in which case `gxNegativeInfinity` is returned.

WideWideDivide

You can use the `WideWideDivide` function to calculate a `wide` quotient and `long` remainder for a `wide` dividend and a `long` divisor.

```
wide *WideWideDivide(wide *dividend, long divisor,
      long *remainder);
```

`dividend`	A pointer to the `wide` number to be divided.
`divisor`	The number by which the dividend is to be divided.
`remainder`	A pointer to a location to store the remainder of the division.
function result	A pointer to the quotient (also to the dividend).

DESCRIPTION

The `WideWideDivide` function returns the quotient of the dividend and divisor as its function result and places the remainder in the `remainder` parameter. If the `remainder` parameter is `nil`, `WideWideDivide` returns the rounded quotient. The quotient replaces the dividend. The operation has a bias of 0 bits; the bias of the result is the difference between the biases of the dividend and the divisor. The bias of the remainder is the same as the bias of the dividend.

If the `remainder` parameter is `nil`, no remainder is returned and the `WideDivide` function returns a rounded quotient. Passing `(long *)-1` in the `remainder` parameter is the same as passing `nil`.

Note that this function cannot result in overflow.

WideSquareRoot

You can use the `WideSquareRoot` function to calculate the square root of a `wide` number.

```
unsigned long WideSquareRoot(const wide *source);
```

`source`	A pointer to the number for which the square root is to be calculated.
function result	A number that is the square root of the number in the argument.

DESCRIPTION

The `WideSquareRoot` function returns the square root of the `wide` number pointed to by the `source` parameter. The source value for this function must be an unsigned `wide` value ranging from 0 to $2^{64} – 1$, not $–2^{63}$ to $2^{63} – 1$. If you supply a non-integer value for this function, its bias must be an even number of bits.

WideScale

You can use the `WideScale` function to obtain the bit number of the highest-order nonzero bit in the absolute value of a `wide` number.

```
short WideScale(const wide *w);
```

`w`	A pointer to the number whose scale is desired.
function result	The bit number of the highest order nonzero bit in the absolute value of `w`. The returned value is 63 if the highest-order bit is set, and 0 if the lowest order bit is set. If no bit is set, the return value is –1.

WideCompare

You can use the `WideCompare` function to compare the magnitudes of two 64-bit numbers.

```
short WideCompare(const wide *target, const wide *source);
```

`target`	A pointer to one of the two `wide` numbers to be compared.
`source`	A pointer to the second of the two `wide` numbers to be compared.

function result 1 if the target number is greater, –1 if the source number is greater, and 0 if the two numbers are equal.

Vector Operations

QuickDraw GX provides an assortment of vector mathematics functions for your use.

VectorMultiply

You can use the `VectorMultiply` function to obtain the dot product of two vectors with 64-bit accuracy.

```
wide *VectorMultiply(long count, const long *vector1, long step1,
const long *vector2, long step2, wide *dot);
```

`count`	The size of each vector.
`vector1`	A pointer to one of the two vectors.
`step1`	The index increment for the `vector1` vector.
`vector2`	A pointer to the second of two vectors.
`step2`	The index increment for the `vector2` vector.
`dot`	A pointer to the destination of the result.

function result A pointer to the dot product of the two vectors.

DESCRIPTION

The `VectorMultiply` function calculates the `wide` dot product of the parameters `vector1` and `vector2`. The size of each vector is given by the `count` parameter. The index increment is given by the parameters `step1` and `step2`, respectively. The `dot` parameter points to the destination `wide` number and is returned as the function result.

SEE ALSO

Examples of how to use the `VectorMultiply` function are provided in the section "Performing Vector Operations" beginning on page 8-29.

VectorMultiplyDivide

You can use the `VectorMultiplyDivide` function to calculate the quotient of the dot product of two vectors and a divisor.

```
long *VectorMultiplyDivide(long count, const long *vector1,
                           long step1, const long *vector2,
                           long step2, long divisor);
```

`count`	The size of each vector.
`vector1`	A pointer to one of the two vectors.
`step1`	The index increment for the `vector1` vector.
`vector2`	A pointer to the second of two vectors.
`step2`	The index increment for the `vector2` vector.
`divisor`	The number by which the dot product is to be divided.

function result The quotient of the dot product of two vectors and a divisor.

DESCRIPTION

The `VectorMultiplyDivide` function calculates the quotient of a dot product of parameters `vector1` and `vector2` and a `divisor` parameter. The size of each vector is given by the `count` parameter. The index increment is given by the parameters `step1` and `step2`, respectively. If the `count` parameter is negative, the terms are alternated. This is equivalent to

```
WideDivide(VectorMultiply(),divisor)
```

Cartesian and Polar Coordinate Point Conversions

QuickDraw GX provides two functions for converting Cartesian to polar coordinates.

PolarToPoint

You can use the `PolarToPoint` function to convert a point in polar coordinates to the identical point in Cartesian coordinates.

```
gxPoint *PolarToPoint(const gxPolar *ra, gxPoint *xy);
```

`ra`	A pointer to the point in polar coordinates.
`xy`	A pointer to the destination of the resulting point in Cartesian coordinates.
function result	A pointer to the converted point (also a pointer to the `xy` parameter).

DESCRIPTION

The `PolarToPoint` function converts the polar coordinate point (*r*, *a*) to the identical Cartesian coordinate point (*x*, *y*). The parameters of the `PolarToPoint` function are the `gxPolar` structure pointer `ra` and a `gxPoint` structure pointer `xy`.

If both pointers point to the same location, the source `gxPolar` structure will be converted to a `gxPoint` structure and will replace the `gxPolar` structure.

SEE ALSO

The `gxPolar` structure is described in the section "Constants and Data Types" beginning on page 8-35. Polar coordinate to Cartesian coordinate conversions are discussed in the section "Cartesian and Polar Coordinate Conversion" beginning on page 8-10. The `PointToPolar` function converts a point in Cartesian coordinates to the identical point in polar coordinates. The `PointToPolar` function is described next.

PointToPolar

The `PointToPolar` function converts a point in Cartesian coordinates to the identical point in polar coordinates.

```
gxPolar *PointToPolar(const gxPoint *xy, gxPolar *ra);
```

`xy`	A pointer to the Cartesian coordinate.
`ra`	A pointer to the destination of the resulting polar coordinate.
function result	The pointer passed in `ra`.

DESCRIPTION

The `PointToPolar` function converts the Cartesian coordinate point (*x*, *y*) to the identical polar coordinate point (*r*, *a*). The parameters of the `PointToPolar` function are a `gxPoint` structure pointer `xy` and a `gxPolar` structure pointer `ra`.

If both pointers point to the same location, the source `gxPoint` structure will be converted to a `gxPolar` structure and will replace the `gxPoint` structure.

SEE ALSO

The `gxPolar` structure is described in the section “Constants and Data Types” beginning on page 8-35. The `PolarToPoint` function converts a point in polar coordinates to the identical point in Cartesian coordinates. The `PolarToPoint` function is described in the previous section.

Random Number Generation

QuickDraw GX provides random number generation functions that can be used in your application.

RandomBits

You can use the `RandomBits` function to return a sequence of pseudorandom numbers.

```
unsigned long RandomBits(long count, long focus);
```

`count`	The number of bits in the number to be generated by the random number generator.
`focus`	The degree of clustering about the mean value.

function result A sequence of pseudorandom numbers.

DESCRIPTION

The `RandomBits` function returns random numbers in the range of 0 to $2^{\texttt{count}} - 1$. A focus of 0 generates numbers that are uniformly distributed.

A positive value for the `focus` parameter generates numbers that are clustered about the mean, analogous to averaging $2^{\texttt{focus}}$ uniform random numbers. A negative focus generates numbers that tend to avoid the mean.

If you define a value *limit* to be 1 << `count`, the result of the `RandomBits` function ranges from 0 to *limit* – 1. Its mean is (*limit* – 1) / 2. The mean is independent of the focus. If the focus is positive, the standard deviation of the numbers generated by the `RandomBits` function is approximately $(0.28868 \times limit) / e^{1.41421 \times \text{focus}}$. As the `focus` parameter gets bigger, two things happen:

- The values cluster about the mean.
- The values approximate a normal distribution (central limit theorem).

If the focus is negative, the `RandomBits` function result is computed as if it were positive; for results less than *limit* / 2, *limit* / 2 is added; for others, *limit* / 2 is subtracted. This causes the distribution to avoid the mean.

To generate a clustering of points around a given value, generate x and y offsets with

```
FractMultiply(radius, RandomBits(31, focus) - fract1);
```

The average distance will be $0.57735 \times \text{radius} / e^{1.41421 \times \text{focus}}$.

A good way to select a value for the focus is to experiment until the desired result is achieved.

SEE ALSO

The `SetRandomSeed` function sets the starting number seed for the random number generator algorithm. The `SetRandomSeed` function is described in the next section. The `GetRandomSeed` function returns the current starting number seed for the random number generator algorithm. The `GetRandomSeed` function is described on page 8-60.

SetRandomSeed

You can use the `SetRandomSeed` function to set the starting number for the random number generator algorithm.

```
void SetRandomSeed(const wide *seed);
```

`seed`	The pointer to the number to be used by the random number algorithm to generate random numbers.

DESCRIPTION

Random number generators are seeded with a value that is used by the algorithm to generate a random number. The seed is then used to generate the next random number.

The `SetRandomSeed` function allows you to select the seed used by the QuickDraw GX random number algorithm. If `SetRandomSeed` is not used, QuickDraw GX will select a default seed of 0. This results in the same sequence of random numbers each time `RandomBits` is called.

In order to obtain a different set of random numbers than those obtained using the default seed value or a previously set seed, use the `SetRandomSeed` function.

SEE ALSO

The `RandomBits` function uses the current seed to generate the next random number. The `RandomBits` function is described on page 8-58. The `GetRandomSeed` function returns the current seed. The `GetRandomSeed` function is described next.

GetRandomSeed

You can use the `GetRandomSeed` function to return the current seed for the random number generating algorithm.

```
wide *GetRandomSeed(wide *seed);
```

`seed`	A pointer to the current random number generator seed.
function result	The pointer passed in the `seed` parameter.

DESCRIPTION

The `GetRandomSeed` function returns the current seed for the random number generator and returns the pointer passed in `seed`.

SEE ALSO

The `RandomBits` function uses the current seed to generate the next random number. The `RandomBits` function is described in the previous section. The `SetRandomSeed` function changes the current seed. The `SetRandomSeed` function is described in the previous section.

Linear and Quadratic Roots

QuickDraw GX provides two functions that solve for the roots of linear and quadratic equations.

LinearRoot

You can use the `LinearRoot` function to obtain the root of a linear equation.

```
long LinearRoot(Fixed first, Fixed last, fract t[]);
```

`first`	The first coefficient.
`last`	The last coefficient.
`t`	An array of fract numbers. On return, it contains the roots of the equation.
function result	The number of roots of the linear equation. This value may be 0 or 1 (or –1 if all values of t are roots).

DESCRIPTION

The `LinearRoot` function computes any t between 0 and 1 in which a(1 – t) + bt = 0. The coefficient a is the parameter `first`. The coefficient b is the parameter `last`. The function returns the number of roots between 0 and 1.

Any root is returned in the `t` array, which only needs to hold one value. If both a and b are zero, the function returns the number –1, indicating that a(1 – t) + bt = 0 for all t.

QuadraticRoot

You can use the `QuadraticRoot` function to calculate the roots of a quadratic equation.

```
long QuadraticRoot(Fixed first, Fixed control, Fixed last, fract
t[]);
```

`first`	The first coefficient.
`control`	The second coefficient.
`last`	The third coefficient.
`t`	An array of fract numbers. On return, it contains the roots of the equation.
function result	The number of roots of the quadratic equation. This value may be 0, 1, or 2 (or –1 if all values of t are roots).

DESCRIPTION

The `QuadraticRoot` function returns roots between 0 and 1 for quadratic equations having the form $a(1 – t)^2 + 2bt(1 – t) + ct^2 = 0$. The coefficient a is the parameter `first`. The coefficient b is the parameter `control`. The coefficient c is the parameter `last`.

All roots are returned in increasing order in the `t` array. The array can have at most two values. If a, b, and c are all zero, then the function returns the number –1, indicating that $a(1 – t)^2 + 2bt(1 – t) + ct^2 = 0$ for all t.

Bit Analysis

QuickDraw GX provides a function that allows you to analyze the bits in a number.

FirstBit

You can use the `FirstBit` function to determine the highest order bit that is set in a number.

```
short FirstBit (unsigned long x);
```

`x`	The number for which the first bit is to be determined.
function result	The bit number of the highest order bit of the number in the argument.

DESCRIPTION

The `FirstBit` function returns the bit number of the highest order bit in a number that is set, or –1 if the number is 0. The highest-order bit is bit 31; the lowest-order bit is bit 0.

DESCRIPTION

The use of the `FirstBit` function is described in the section “Analyzing the Bits in a Number” on page 8-33.

Mapping Functions

QuickDraw GX provides two groups of mapping functions. The first group allows you to manipulate mapping matrices and apply them to other mappings or to points. The second group allows you to modify the transformational properties of a mapping matrix.

Mappings are described in the section “Transformation Operations With Mappings” beginning on page 8-12.

For specific information on mapping matrices as applied to transform objects, view port objects, and view device objects, see the chapters “Transform Objects” and “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Manipulating and Applying Mappings

This section describes functions with which you can copy, normalize, reset, and invert a mapping. It also describes functions with which you can apply a mapping to another mapping, and apply a mapping to an array of points.

CopyToMapping

You can use the `CopyToMapping` function to copy a mapping from one location to another location.

```
gxMapping *CopyToMapping(gxMapping *target,
                        const gxMapping *source);
```

`target`	A pointer to the destination mapping. On return, it is a copy of the source mapping.
`source`	A pointer to the mapping to be copied.
function result	A pointer to the copied mapping, which is also the target mapping.

DESCRIPTION

The `CopyToMapping` function copies the mapping pointed to by the `source` parameter into the location pointed to by the `target` parameter. Note that it may be faster in C to simply copy the `gxMapping` structure into another `gxMapping` structure than to call this function.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

NormalizeMapping

You can use the `NormalizeMapping` function to normalize a mapping.

```
gxMapping *NormalizeMapping(gxMapping *target);
```

`*target`	A pointer to the mapping to be normalized. On return, it is the normalized mapping.
function result	A pointer to the normalized mapping, which is also the target mapping.

DESCRIPTION

The `NormalizeMapping` function normalizes the target mapping. If the mapping's perspective elements (u and v) are 0, each element of the mapping is divided by element w (`target–>[2][2]`). If the mapping has a nonzero perspective, each element is shifted to ensure that `fract1`$/2 < |u| + |v| + (|w| >> 15) \le$ `fract1`.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

ResetMapping

You can use the `ResetMapping` function to reset a mapping.

```
gxMapping *ResetMapping(gxMapping *target);
```

`target`	A pointer to the mapping that is to be reset. On return, contains the identity mapping.
function result	A pointer to the reset mapping, which is also the target mapping.

DESCRIPTION

The `ResetMapping` function resets the target mapping to the identity matrix.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

An example of the use of the `ResetMapping` function is provided on page 8-34.

InvertMapping

You can use the `InvertMapping` function to create an inverted copy of a mapping.

```
gxMapping *InvertMapping(gxMapping *target,
                         const gxMapping *source);
```

`target`	A pointer to a mapping structure. On return, contains the inverse of the mapping specified in the `source` parameter.
`source`	A pointer to the mapping to be inverted.
function result	A pointer to the inverted mapping, which is also the target mapping.

DESCRIPTION

The `InvertMapping` function creates a copy of the source mapping, inverts it, and returns the inverted mapping in the `target` parameter. If both the `source` and `target` parameters point to the same `gxMapping` structure, that mapping will be replaced by its inverse. If the mapping is not invertible, the function returns nil and the target is not changed.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

MapMapping

You can use the `MapMapping` function to concatenate two mappings.

```
gxMapping *MapMapping(gxMapping *target, const gxMapping *source);
```

`target`	A pointer to the mapping to be modified. On return, contains the result of the concatenation.
`source`	A pointer to the mapping to be concatenated with the target mapping.
function result	A pointer to the resultant mapping, which is also the target mapping.

DESCRIPTION

The `MapMapping` function postmultiplies the target mapping by the source mapping, and returns the result in the `target` parameter.

The result of passing the function result of `MapMapping` tothe `GXMapShape` function is equivalent to passing the result of one call to `GXMapShape` to another call to `GXMapShape`, as shown below (for the shape `s`):

```
GXMapShape(s, target);
GXMapShape(s, source);
```

The same results would be obtained more efficiently and perhaps more accurately by making these calls:

```
MapMapping(target, source);
GXMapShape(s, target);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The `GXMapShape` function is described in the chapter Transform Objects in *Inside Macintosh: QuickDraw GX Objects.*

MapPoints

You can use the `MapPoints` function to apply a mapping to each of the points in an array.

```
void MapPoints(const gxMapping *source, long count,
               gxPoint vector[]);
```

`source`	A pointer to the mapping that is to be applied to the array of points.
`count`	The number of points in the array.
`vector`	The array of points to which the mapping is to be applied. On return, the array contains the transformed points.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`
`parameter_is_nil`
`number_of_points_exceeds_implementation_limit`

Warnings
`map_points_out_of_range`

SEE ALSO

For an example of a function that applies a mapping to a single point, see Listing 8-2 on page 8-30.

Modifying Mappings

This section describes functions with which you can modify the translational, scaling, rotational, and skewing properties of a mapping.

Similar functions are available that allow you to directly modify the transformational properties of the mapping in the transform object associated with a QuickDraw GX shape. See the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects* for more information.

MoveMapping

You can use the `MoveMapping` function to change the horizontal and vertical translation factors of a mapping by given amounts.

```
gxMapping *MoveMapping(gxMapping *target, Fixed hOffset,
                        Fixed vOffset);
```

`target`	A pointer to the mapping to be modified. On return, points to the modified mapping.
`hOffset`	The horizontal translation to add to the mapping.
`vOffset`	The vertical translation to add to the mapping.

function result A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `MoveMapping` function postmultiplies the target mapping by a mapping that adds `hOffset` to the x translation and `vOffset` to the y translation of the target mapping.

Passing the result of this function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXMoveShape` function.

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `MoveMapping` function is described in the section "Translation by a Relative Amount" beginning on page 8-17.

The `GXMapShape` and `GXMoveShape` functions are described in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects*.

MoveMappingTo

You can use the `MoveMappingTo` function to assign specific values to the horizontal and vertical translation factors of a mapping.

```
gxMapping *MoveMappingTo(gxMapping *target, Fixed hPosition,
                         Fixed vPosition);
```

`target`	A pointer to the mapping that is to be modified. On return, points to the modified mapping.
`hPosition`	The horizontal translation to be assigned to the target mapping.
`vPosition`	The vertical translation to be assigned to the target mapping.
function result	A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `MoveMappingTo` function postmultiplies the target mapping by a mapping that assigns `hPosition` to the x translation and `vPosition` to the y translation of the target mapping. This function sets the translational origin of the mapping; the point (0, 0), when postmultiplied by the mapping that results from this function, will be at location (`hPosition`, `vPosition`).

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `MoveMappingTo` function is described in the section “Translation to a Specified Point” beginning on page 8-18.

ScaleMapping

You can use the `ScaleMapping` function to change the horizontal and vertical scale factors of a mapping.

```
gxMapping *ScaleMapping(gxMapping *target,
                        Fixed hFactor, Fixed vFactor,
                        Fixed xCenter, Fixed yCenter);
```

`target`	A pointer to the mapping that is to be modified. On return, points to the modified mapping.
`hFactor`	The horizontal scaling factor to apply. A value of 1.0 means no scale change in the x direction.
`vFactor`	The vertical scaling factor to apply. A value of 1.0 means no scale change in the y direction.
`xCenter`	The x-coordinate of the center of scaling.
`yCenter`	The y-coordinate of the center of scaling.

function result A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `ScaleMapping` function postmultiplies the target mapping by a mapping that specifies a horizontal scaling factor of `hFactor` and a vertical scaling factor of `vFactor`, about the point (`xCenter`, `yCenter`). Note that if `hFactor` is 1, `xCenter` irrelevant; likewise, if `vFactor` is 1, `yCenter` is irrelevant.

These scaling factors are in addition to any preexisting scaling factors in the target mapping. The center of scaling is the point that does not move when the scaling is applied.

Passing the result of the `ScaleMapping` function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXScaleShape` function. For example, you could make these calls (for the shape `s`):

```
ScaleMapping(target, hFactor, vFactor, xCenter, yCenter);
GXMapShape(s, target);
```

or, you could make these equivalent calls:

```
GXMapShape(s, target);
GXScaleShape(s, hFactor, vFactor, xCenter, yCenter);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `ScaleMapping` function is described in the section “Scaling” beginning on page 8-20.

The `GXMapShape` and `GXScaleShape` functions are described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

RotateMapping

You can use the `RotateMapping` function to change the rotation specified by a mapping.

```
gxMapping *RotateMapping(gxMapping *target, Fixed angle,
                         Fixed xCenter, Fixed yCenter);
```

`target`	A pointer to the mapping to be modified. On return, points to the modified mapping.
`angle`	The amount of rotation (in degrees clockwise) to be added to the mapping.
`xCenter`	The x-coordinate of the center of rotation.
`yCenter`	The y-coordinate of the center of rotation.

function result	A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `RotateMapping` function postmultiplies the target mapping by a mapping that specifies a rotation (clockwise if positive) by a specified number of degrees about the point (`xCenter`, `yCenter`).

The rotation is in addition to any preexisting rotation specified by the target mapping.

Passing the result of this function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXRotateShape` function. For example, you could make these calls (for the shape `s`):

```
RotateMapping(target, angle, xCenter, yCenter);
GXMapShape(s, target);
```

or, you could make these equivalent calls:

```
GXMapShape(s, target);
GXRotateShape(s, angle, xCenter, yCenter);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `RotateMapping` function is described in the section “Rotation” beginning on page 8-22.

The `GXMapShape` and `GXRotateShape` functions are described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

SkewMapping

You can use the `SkewMapping` function to change the horizontal and vertical skew specified by a mapping.

```
gxMapping *SkewMapping(gxMapping target,
                       Fixed skewX, Fixed skewY,
                       Fixed xCenter, Fixed yCenter);
```

`target`	A pointer to the mapping that is to be modified. On return, points to the modified mapping.
`skewX`	The scaling factor that determines the amount of skew in the x direction. A value of 0 means no horizontal skew.

`skewY`	The scaling factor that determines the amount of skew in the y direction. A value of 0 means no vertical skew.
`xCenter`	The x-coordinate of the center of skewing.
`yCenter`	The y-coordinate of the center of skewing.
function result	A pointer to the modified mapping, which is also the target mapping.

DESCRIPTION

The `SkewMapping` function postmultiplies the target mapping by a mapping that specifies a horizontal skew factor of `skewX` and a vertical skew factor of `skewY`, about the point (`xCenter`, `yCenter`). Note that if `skewX` is 0, `yCenter` irrelevant; likewise, if `skewY` is 0, `xCenter` is irrelevant.

These skew factors are in addition to any preexisting skew specified in the target mapping. The center of skewing specifies the point at which no translation takes place because of the skewing.

Passing the result of the `SkewMapping` function to the `GXMapShape` function is equivalent to calling the `GXMapShape` function and then calling the `GXSkewShape` function. For example, you could make these calls (for the shape `s`):

```
SkewMapping(target, hFactor, vFactor, xCenter, yCenter);
GXMapShape(s, target);
```

or, you could make these equivalent calls:

```
GXMapShape(s, target);
GXSkewShape(s, skewX, skewY, xCenter, yCenter);
```

ERRORS, WARNINGS, AND NOTICES

Errors
`mapping_is_nil`

SEE ALSO

The use of the `SkewMapping` function is described in the section “Skewing” beginning on page 8-24.

The `GXMapShape` and `GXSkewShape` functions are described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects.*

Summary of QuickDraw GX Mathematics

Constants and Data Types

Number Formats and Constants

```
typedef long fract;

typedef unsigned short gxColorValue;

struct gxPolar {
   Fixed radius;
   Fixed angle;
};

struct wide {
   long hi;
   unsigned long lo;
};

#define fixed1             ((Fixed) 0x00010000)        /* = 1.0 for Fixed */

#define fract1             ((fract) 0x40000000)        /* = 1.0 for fract */

#define gxColorValue1      ((gxColorValue) 0xFFFF) /* 1.0 for gxColorValue*/

#define gxPositiveInfinity ((Fixed) 0x7FFFFFFF)    /* for Fixed and fract */

#define gxNegativeInfinity ((Fixed) 0x80000000)    /* for Fixed and fract */
```

The Mapping Structure

```
struct gxMapping {
   Fixed map[3][3];
};
```

Number-Conversion Macros

Format Conversions

```
#define FixedToFract(a)     ((fract) (a) << 14)
#define FractToFixed(a)     ((Fixed) (a) + 8192L >> 14)
#define FixedToInt(a)       ((short) ((Fixed) (a) + fixed1/2 >> 16))
#define IntToFixed(a)       ((Fixed)(a) << 16)
#define ff(a)               ((Fixed)(a) << 16)
#define FixedToFloat(a)     ((float)(a) / fixed1)
#define FloatToFixed(a)     ((Fixed)((float) (a) * fixed1))
#define fl(a)               ((Fixed)((float) (a) * fixed1))
#define FractToFloat(a)     ((float)(a)/fract1)
#define FloatToFract(a)     ((fract)((float)(a)*fract1))
#define ColorToFract(a)     (((fract)(a)<<14) + ((fract)(a) +2 >>2))
#define FractToColor(a)     ((gxColorValue)((a)-((a)>>16)+8191>>14))
```

Rounding, Truncating, and Square Root Operations

```
#define FixedRound(a)       ((short) ((Fixed)(a) + fixed1/2 >> 16))
#define FixedTruncate(a)    ((short)((Fixed)(a) >> 16))
#define FixedSquareRoot(a)  ((Fixed)FractSquareRoot(a) + 64 >>7)
```

Mathematical Functions

Fixed-Point Operations

```
Fixed FixedMultiply         (Fixed multiplicand, Fixed multiplier);
Fixed FixedDivide           (Fixed dividend, Fixed divisor);
long MultiplyDivide         (long multiplicand, long multiplier,
                             long divisor);
unsigned long Magnitude     (long deltaX, long deltaY);
fract FractSineCosine       (Fixed degrees, fract *cosine);
fract FractSquareRoot       (fract source);
fract FractCubeRoot         (fract source);
fract FractMultiply         (fract multiplicand, fract multiplier);
fract FractDivide           (fract dividend,fract divisor);
```

Operations on wide Numbers

```
wide *WideAdd                (wide *target, const wide *source);
wide *WideSubtract           (wide *target, const wide *source);
wide *WideNegate             (wide *target);
wide *WideShift              (wide *target, long shift);
wide *WideMultiply           (long multiplicand, long multiplier,
                              wide *target);
long WideDivide              (const wide *dividend, long divisor,
                              long *remainder);
wide *WideWideDivide         (wide *dividend, long divisor, long *remainder);
unsigned long WideSquareRoot
                             (const wide *source);
short WideScale              (const wide *w);
short WideCompare            (const wide *target, const wide *source);
```

Vector Operations

```
wide *VectorMultiply         (long count, const long *vector1, long step1,
                              const long *vector2, long step2, wide *dot);
long *VectorMultiplyDivide   (long count, const long *vector1, long step1,
                              const long *vector2, long step2, long divisor);
```

Cartesian and Polar Coordinate Point Conversions

```
gxPoint *PolarToPoint        (const gxPolar *ra, gxPoint *xy);
gxPolar *PointToPolar        (const gxPoint *xy, gxPolar *ra);
```

Random Number Generation

```
unsigned long RandomBits     (long count, long focus);
void SetRandomSeed           (const wide *seed);
wide *GetRandomSeed          (wide *seed);
```

Linear and Quadratic Roots

```
long LinearRoot              (Fixed first, Fixed last, fract t[]);
long QuadraticRoot           (Fixed first, Fixed control, Fixed last,
                              fract t[]);
```

Bit Analysis

```
short FirstBit               (unsigned long x);
```

Mapping Functions

Manipulating and Applying Mappings

```
gxMapping *CopyToMapping    (gxMapping *target, const gxMapping *source);
gxMapping *NormalizeMapping (gxMapping *target);
gxMapping *ResetMapping     (gxMapping *target);
gxMapping *InvertMapping    (gxMapping *target, const gxMapping *source);
gxMapping *MapMapping       (gxMapping *target, const gxMapping *source);
void MapPoints              (const gxMapping source, long count,
                             gxPoint vector[]);
```

Modifying Mappings

```
gxMapping *MoveMapping      (gxMapping *target,
                             Fixed hOffset, Fixed vOffset);
gxMapping *MoveMappingTo    (gxMapping *target,
                             Fixed hPosition, Fixed vPosition);
gxMapping *ScaleMapping     (gxMapping *target,
                             Fixed hFactor, Fixed vFactor,
                             Fixed xCenter, Fixed yCenter);
gxMapping *RotateMapping    (gxMapping *target, Fixed angle,
                             Fixed xCenter, Fixed yCenter);
gxMapping *SkewMapping      (gxMapping target, Fixed skewX, Fixed skewY,
                             Fixed xCenter, Fixed yCenter);
```