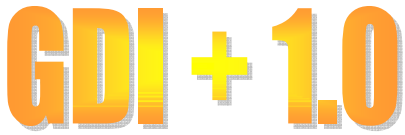

by Stephen Rodriguez

Matrix Class

General.

Our GDI+ Matrix class is housed within the "gdiPlus_MatrixClass.pbi" source file.

The class contains elements of the following c/c++ classes :

Matrix

This is not a *helper-class* (containing just simple macros etc.) but a full OOP class.

Interface / base-class.

Our matrix class exposes a single **interface** with the name gdiPlus_Matrix.

This interface extends our gdiPlus_BaseClass interface which exposes a single method :

GetGdiPlusHandle.i()

which can be used to retrieve the Matrix handle used natively by GDI+.

A note on GDI+ matrices.

A Matrix object represents a 3×3 matrix which, in turn, represents an **affine transformation** (a **linear transformation** followed by a **translation**).

A Matrix object stores only six of the 9 numbers in a 3×3 matrix because all 3×3 matrices that represent affine transformations have the same third column (0, 0, 1).

A general GDI+ matrix has the following form :

$$\begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ dx & dy & 1 \end{pmatrix}$$

where $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ represents the linear transformation (e.g. a shear or a rotation) and $\mathbf{b} = \begin{pmatrix} dx \\ dy \end{pmatrix}$ is the translation vector.

The affine transformation, *T*, can be expressed in multiplicative form as follows :

$$(X \quad Y \quad 1) = T(x \quad y \quad 1)$$

$$= (x \quad y \quad 1) \times \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ dx & dy & 1 \end{pmatrix}$$

(this is essentially the transpose of the way transformations are typically set up in mathematics, meaning that GDI+ is doing things backwards!)

'*gdiPlus_Matrix*' constructors.

The following all return (if successful) instances of our gdiPlus_Matrix class whose methods are listed in the following section. These correspond to constructors from the appropriate c/c++ wrapper class and so the reader is advised to look on the appropriate MSDN pages for detailed descriptions.

gdiPlus_CreateIdentityMatrix()

Creates the GDI+ matrix representing the identify transformation $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

gdiPlus_CreateMatrix(m11.f, m12.f, m21.f, m22.f, dx.f, dy.f)

*gdiPlus_CreateMatrixFromRectF(*rect.RectF, *dstplg.PointF)*

*rect must point to a RectF structure containing the elements of the linear transformation part of the affine transformation as follows :

\x holds the element from row 1 column 1
\y holds the element from row 1 column 2
\width holds the element from row 2 column 1
\height holds the element from row 2 column 2.

*dstplg must point to a PointF structure containing the displacement vector components *dx* and *dy*.

*gdiPlus_CreateMatrixFromRectI(*rect.RectI, *dstplg.PointI)*

Similar to the previous function but utilising RectI and PointI structures.

'*gdiPlus_Matrix*' methods.

Unless specified otherwise, the following all return a gdiPlus status code (beginning with *#Ok*).

All of these methods correspond to methods from the appropriate c/c++ wrapper classes and so the reader is advised to look on the appropriate MSDN pages for detailed descriptions.

Destroy()

Clone.i()

Returns, if successful, a new gdiPlus_Matrix object containing an exact copy of the original object.

Equals.i(matrix.gdiPlus_Matrix)

Returns *#True* if the elements of the specified matrix match the underlying matrix.

*GetElements.i(*elements)*

*elements must point to an array of 6 floats which will receive the 6 pertinent elements of the underlying matrix.

Invert.i()

Inverts the matrix (overwriting the original elements). Will fail if the matrix is not invertible.
The inverse matrix represents the inverse (reverse) of the underlying affine transformation.

IsIdentity.i()

Returns *#True* if this is the identity matrix.

IsInvertible.i()

Returns *#True* if the underlying matrix is invertible.

Multiply.i(matrix.gdiPlus_Matrix, matrixOrder=#MatrixOrderPrepend)

Composition of transformations is performed by matrix multiplication.
Prepending the multiplication means that the passed matrix transformation will be first in the order of composition.
matrixOrder should be set to one of the MatrixOrder constants, e.g. *#MatrixOrderPrepend*.

OffsetX.f()

Returns the horizontal translation value of this matrix, which is the element in row 3, column 1.

OffsetY.f()

Returns the vertical translation value of this matrix, which is the element in row 3, column 2.

Reset.i()

Resets the matrix with the elements of the identity matrix.

Rotate.i(angle.f, matrixOrder=#MatrixOrderPrepend)

Positive angles (degrees) for clockwise rotations.
Prepending the rotation means that the rotation will be first in the order of composition.

RotateAt.i(angle.f, *point.PointF, matrixOrder=#MatrixOrderPrepend)

Updates this matrix with the product of itself and a matrix that represents rotation about a specified point.
Positive angles (degrees) for clockwise rotations.
Prepending the rotation means that the rotation will be first in the order of composition.

Scale.i(scaleX.f, scaleY.f, matrixOrder=#MatrixOrderPrepend)

Prepending the scaling means that the scaling will be first in the order of composition.

SetElements.i(m11.f, m12.f, m21.f, m22.f, dx.f, dy.f)

Shear.i(shearX.f, shearY.f, matrixOrder=#MatrixOrderPrepend)

Prepending the shear means that the shear will be first in the order of composition.

TransformPointsF.i(*pts, count)

*pts must point to an array of PointF structures.

TransformPointsI.i(*pts, count)

*pts must point to an array of PointI structures.

TransformVectorsF.i(*pts, count)

As for TransformPointsF() but ignoring any translation element.

TransformVectorsI.i(*pts, count)

As for TransformPointsI() but ignoring any translation element.

Translate.i(offsetX.f, offsetY.f, matrixOrder=#MatrixOrderPrepend)

Prepending the translation means that the translation will be first in the order of composition.