Formation C# – Delphi.NET – Delphi Win32
Développement & Sous-traitance

GDI+

Dessiner des points de couleur pour former du texte, dessiner des fenêtres, tout cela apparaît normal à tout utilisateur d'ordinateur. Pourtant il faut des millions d'opérations par seconde et des API pour contrôler chaque pixel et surtout, des développeurs pour les programmer… Windows et ses API procédurales sans réelle cohérence ne rendait pas toujours les choses aisées, ce qui fit la voie royale à des librairies comme la VCL de Borland par exemple. Il était temps que Windows se dote d'une API à la hauteur du défi que doivent relever les applications dans un monde où l'aspect visuel est devenu aussi important que le fonctionnel. Avec GDI+ et les classes du framework pour le gérer Windows franchit un cap essentiel, celui de la cohérence objet, celui des fonctions de haut niveau et d'une programmation à la mesure des attentes des développeurs. BDS, par son support natif du framework aussi bien

sous Delphi.NET que C#, permet au développeur de bénéficier sans effort de tous les services de GDI+.

Principes de base

GDI+ succède à GDI, *Graphical Device Interface*, interface pour périphérique graphique. Il s'agit donc d'un ensemble d'API permettant de contrôler les périphériques graphiques dont les plus courants sont les écrans et les imprimantes.

Si sous .NET tout est objet, pour un périphérique graphique tout est point… Dessiner des fenêtres ou écrire du texte sont des opérations excessivement complexes à tel point que les premiers PC utilisaient des caractères préformés enregistrés en mémoire morte pour écrire le texte à l'écran afin de décharger le CPU d'un travail jugé à l'époque bien trop lourd pour les processeurs. Aujourd'hui une application peut contrôler des fenêtres multiples décorées d'icônes, d'images animées, imprimer des documents complexes, afficher des représentations 3D et bien d'autres artifices, comme l'alpha blending, qui outre du matériel sophistiqué réclament une interface de programmation à la hauteur de la tâche à accomplir.

Les principes de base de GDI+ restent similaires à ceux de GDI, c'est-à-dire que le développeur dispose d'un contexte graphique, d'une surface de dessin et d'outils pour y écrire. La véritable différence se joue à deux niveaux : celui de la cohérence objet au lieu des API disparates de GDI et celui de la maturité des fonctions de haut niveau mises à disposition.

Les différents aspects de GDI+

GDI+ est à la base une amélioration de GDI supportée par Windows XP et Windows 2003 et pouvant être installée sur les versions précédentes. GDI+ offre trois pôles distincts que sont :

- Le graphisme vectoriel
- Le traitement des images
- La typographie.

La partie dessin vectoriel prend en charge les dessins fondés sur des formes géométriques en 2D, de la ligne aux formes comme le cercle en passant par les associations et combinaisons de toutes ses figures. Sous GDI+ la séparation entre paramétrage d'un dessin et exécution de celui-ci est nette, ce qui autorise une meilleure

structuration de la programmation et une plus grande réutilisation du code.

La partie support image de GDI+ s'occupe de traiter des images entières comme un tout. On trouve notamment ici le support des formats graphiques les plus populaires comme PNG ou JPEG là où GDI ne gérait que BMP.

La partie typographie de GDI+ comporte les API et les outils permettant de créer et de manipuler des polices de caractères.

Les nouveautés de GDI+

GDI+ offre de très nombreuses améliorations. L'anti-aliasing des lignes et courbes est certainement l'une des plus marquantes mais ce n'est pas, et de loin, la seule des évolutions.

Les brosses dégradées

Les brosses dégradées (*gradient brushes*) permettent désormais de remplir des espaces avec des dégradés de couleurs programmables.

Les lignes adoucies et les courbes de bézier

Appelées *cardinal spline* ces lignes sont formées d'un ensemble de points. À la différence des séries de points existant sous GDI, les lignes adoucies de GDI+ gère automatiquement une transition douce entre les segments pour former des courbes lissées plutôt que des lignes cassées. À côté de ces objets déjà très évolués GDI+ offre même un support complet des courbes de bézier.

Persistances des chemins graphiques

Sous GDI les chemins graphiques sont liés au contexte et sont détruits au fur et à mesure qu'ils sont dessinés. Sous GDI+ les chemins sont des objets indépendants non liés à un contexte graphique particulier. Ils peuvent être réutilisés à volonté même sur des contextes différents.

Matrices de transformation

GDI+ introduit la notion de matrice de transformation, un outil mathématique largement exploité dans les applications graphiques. En assurant le support natif de telles matrices GDI+ simplifie

grandement les opérations de translations ou déformations applicables à de nombreux outils comme les chemins par exemple.

Les régions extensibles

Sous GDI les régions sont définies en coordonnées du contexte graphique et ne supporte aucun traitement en dehors de la translation. Sous GDI+ les coordonnées sont exprimées en *world coordinates* et les régions ainsi définies peuvent subir toutes les transformations offertes par les matrices de transformation ainsi que le changement d'échelle.

Alpha blending

L'alpha blending est un paramètre supplémentaire qui est indiqué à la création d'une couleur. Il permet d'indiquer son niveau de transparence. De fait deux objets qui ne sont pas complètement opaques et qui se superposent deviennent visibles « par transparence », les couleurs des zones recouvertes se mélangeant en respectant les règles de la synthèse additive (un objet rouge et un objet bleu se recouvrant créeront une zone magenta).

Support des formats de fichier répandus

GDI souffrait d'un support rudimentaire pour le stockage et la lecture des fichiers graphiques. GDI+ offre de façon native le support des formats les plus populaires dont :

- BMP
- GIF
- JPEG
- EXIF
- PNG
- TIFF
- ICON
- WMF
- EMF

Les couleurs

Sous GDI+ les couleurs sont stockées sur quatre octets dans des structures de type `Color`. De l'adresse basse à l'adresse haute on trouve les octets : rouge, vert, bleu du système RGB puis l'octet Alpha. En terme de nombre de couleurs le développeur dispose de 16.777.216 couleurs, le paramètre `Alpha` servant à créer des transparences.

Le type `Color` possède sous la forme de propriétés une liste très large de couleurs « prête à consommer ». Elles sont stockées sous des noms américains comme `Red`, `Brown`, `Khaki`, etc. Les méthodes de `Color` permettent en outre de créer des couleurs à partir des paramètres Alpha et RGB.

Attention de ne pas confondre la structure `Color` et ses possibilités avec les capacités de la cible d'une opération de dessin qui dépend de ses réglages ou de sa sophistication. Même si utilisez 16 millions de couleurs pour créer un document à imprimer cela ne transformera pas votre imprimante jet d'encre monochrome en imprimante laser couleur ! Il en va de même pour les bitmap dont la nature dépend du mode choisi au moment de leur création.

De GDI à GDI+

Sous Win32 pour créer un dessin avec GDI il fallait avant tout obtenir un *handle* sur un contexte graphique (*device context*). On stockait ce contexte dans un handle de type `HDC` pour l'exploiter tout au long de la session de dessin. Une fois le contexte obtenu il fallait l'initialiser par un appel à `BeginPaint`. Il était tout aussi nécessaire d'obtenir des outils pour dessiner sur le contexte, en général des objets de type `Pen` ou `Brush`. Quand ces outils étaient créés il fallait les affecter au contexte pour qu'ils deviennent disponibles. Une fois la session de dessin terminée il convenait de libérer tous les outils utilisés ainsi que le contexte. L'ensemble de ces opérations produisait un code lourd, verbeux, répétitifs et assez fastidieux à tester et déboguer.

Grâce à la VCL les utilisateurs de Delphi bénéficient depuis longtemps d'une objectivation de plus haut niveau avec des classes comme `TCanvas` ou `TFont` évitant de coder chaque étape du processus bas niveau de dessin. Mais cela ne règle pas toutes les difficultés et les API de GDI redeviennent ce qu'elles sont dès lors qu'on s'écarte des outils de la VCL.

GDI+ sous .NET balaye toutes ces contraintes en proposant un ensemble de classes conçues pour collaborer et offrir un support clair aux développeurs. La classe principale s'appelle `Graphics`

qui peut être vue comme une équivalence à `TCanvas` de la VCL mais en plus abouti.

Programmation GDI+

Les assemblages GDI+

Sous .NET les API de GDI+ sont regroupées dans l'espace de noms `System.Drawing`. Ce dernier se décline en sous-espaces spécialisés :

- `System.Drawing` fournit l'ensemble des fonctions de base GDI+ ;
- `System.Drawing.Drawin2D` fournit les fonctions orientées 2D et celles destinées au dessin vectoriel ;
- `System.Drawing.Imaging` fournit les fonctions de gestion d'image ;
- `System.Drawing.Text` fournit les fonctions liées aux polices de caractères ;
- `System.Drawing.Printing` fournit les fonctions orientées impression.
- `System.Drawing.Desing` fournit des boîtes de dialogue et d'autres éléments d'interface pouvant être utilisés pour étendre l'IDE.

La classe Graphics et les bitmaps

La classe `System.Drawing.Graphics` est le pilier du dessin sous GDI+, elle fournit le contexte, les outils de base et les méthodes pour dessiner sur une surface.

Avant de dessiner il convient donc d'obtenir une instance de cette classe ce qui peut se faire de trois façons :

- Au sein d'un gestionnaire d'événement quelconque dans lequel on peut déclarer une variable `Graphics ;`
- Directement dans l'événement `Paint` d'une fenêtre Windows Forms qui fournit une instance de `Graphics` accrochée au contexte de la fiche ;

- En appelant la méthode `CreateGraphics` du contrôle Windows Forms sur lequel on souhaite dessiner (tous les contrôles héritant de cette méthode).

Une fois l'instance de `Graphics` obtenue il suffit alors d'utiliser ses méthodes ainsi que les autres classes mises à la disposition du développeur par le framework .NET pour dessiner.

Libération des ressources

Bien entendu il est inutile de libérer les instances utilisées puisque le ramasse-miettes s'en charge mais sous GDI+ il est souhaitable d'appeler la méthode `Dispose` des objets graphiques pour relâcher les ressources réservées. Certains objets GDI+ peuvent consommer d'énormes ressources, laisser de telles variables sortir de la portée en se reposant uniquement sur le GC peut créer un temps de latence durant lequel un engorgement mémoire (ou ressource graphique) peut se produire. Pour éviter de telles situations libérez systématiquement les objets GDI+ lorsqu'ils ne sont plus utilisés. Sous Delphi.NET appelez Dispose ou Free, sous C# appelez `Dispose` où plus finement créez des blocs `using` effectuant cette opération pour vous (voir l'aide du langage C# fournit avec BDS pour plus d'information sur cette syntaxe).

Lorsque qu'on utilise la classe `Graphics` pour dessiner sur l'espace d'un contrôle il faut garder à l'esprit que le dessin n'est pas persistent et qu'il sera effacé au moindre recouvrement. Il est de la responsabilité du développeur de prévoir cette situation en rafraîchissant le dessin lorsque cela est nécessaire.

Il n'est obligatoire de travailler directement sur l'espace d'un contrôle et un objet `Graphics` peut fort bien être créé en relation avec un objet `Bitmap` pour dessiner « en arrière scène » et produire le graphique qu'une fois celui-ci totalement formé. C'est d'ailleurs une technique largement utilisée sous GDI ou avec la VCL pour produire des affichages francs sans scintillement.

Pour créer un objet `Bitmap` en mémoire il suffit d'appeler son constructeur en fixant sa taille et sa profondeur de bits :

```
// C#
Bitmap bmp = new Bitmap( width, height, pixelformat );

//Delphi.NET
var bmp ;
begin
bmp := Bitmap.Create(width,height,pixelformat) ;
…
```

Les arguments `width` et `height` sont exprimés en pixels en déterminent respectivement la largeur et la hauteur du bitmap. L'argument `pixelformat` est de type `System.Drawing.Imaging.PixelFormat` et spécifie la profondeur en bits de l'image, c'est-à-dire le nombre de couleurs supportées. Il est possible de créer un layer pour la couche alpha gérant la transparence. Le tableau 16.1 donne la liste des formats les plus utiles supportés par GDI+. Certains formats non listés ici

sont assez exotiques mais peuvent être utiles dans des cas très précis, consultez la documentation du framework pour en prendre connaissance.

Tableau 16.1

Liste des formats bitmap supportés les utiles

VALEUR	DESCRIPTION
`Format16bppGrayScale`	Ce format créé une bitmap en dégradé de gris supportant 65536 nuances de gris (sur 16 bits donc).
`Format16bppArgb1555`	Format 16 bits offrant 32768 couleurs avec 5 bits par canal RGB et 1 bit pour le canal alpha.
`Format16bppRgb565`	Format 16 bits avec 5 bits pour les canaux R (rouge) et B (bleu) et 6 bits pour le canal G (vert). Les 65536 couleurs de ce mode offrent un rendu différent des modes 16 bits classiques puisque le canal vert possède un réglage plus « fin » que les autres (6 bits au lieu de 5).
`Format1bppIndexed`	Format bicolore sur 1 bit par pixel. La table de couleur associée peut attribuer les deux couleurs utilisées librement.
`Format24bppRgb`	Format plus classique utilisant 16 bits par pixel pour chaque canal du mode RGB. Le nombre de couleurs utilisables est de 16 777 216.
`Format32bppArgb`	Format 32 bits, 8 bits par canal RGB et 8 bits pour le canal alpha.
`Format48bppRgb`	Format 48 bits, 16 bits sont attribués pour chaque canal RGB.
`Format4bppIndexed`	Format 4 bits indexés par pixel. Une palette est associée au bitmap.
`Format64bppArgb`	Format 64 bits, 16 bits pour chaque canaux RGB et 16 bits pour le canal alpha.
`Format8bppIndexed`	Format 8 bits indexés par pixel. C'est le format 256 couleurs classique possédant une palette.

Exemple de création d'un objet `Graphics` associé à une image bitmap en 24 bits de 200 par 150 pixels :

```
// C#
Bitmap bmp = new Bitmap(200,150,
     System.Drawing.Imaging.PixelFormat.Format24bppRgb) ;
Graphics dessin = Graphics.FromImage(bmp) ;

// Delphi.NET
var bmp :Bitmap ; dessin : Graphics ;
begin
bmp := Bitmap.Create(200,150,
     System.Drawing.Imaging.PixelFormat.Format24bppRgb) ;
dessin := Graphics.FromImage(bmp) ;
…
```

Le système de coordonnées

Par défaut GDI+ utilise un système de coordonnées similaire à celui de GDI, notamment une position zéro se trouvant en haut à gauche de la surface à dessiner. On retrouve aussi une structure `Rectangle` permettant de stocker les coordonnées d'une surface. Mais si les similarités sont grandes, GDI+ étend très largement le concept même de coordonnées et propose de nombreuses options pour les fixer, les manipuler et les stocker que son ancêtre GDI ne possède pas.

Le tableau 16.2 liste les structures de stockage de coordonnées ainsi que leurs principales propriétés. Les structures (en terme de langage) sont des objets à part entière et exposent d'autres propriétés et méthodes que nous vous laissons découvrir dans la documentation. Nous ne nous intéresserons ici qu'aux plus courantes.

Tableau 16.2

Les structures de stockage

STRUCTURE	PROPRIETES PRINCIPALES	DESCRIPTION
`Point`	`X, Y : integer`	Position d'un point
`PointF`	`X, Y : single`	Position d'un point exprimée en décimal
`Size`	`Width, Height : integer`	Taille d'un objet
`SizeF`	`Width, Height : single`	Taille d'un objet exprimée en décimal
`Rectangle`	`Left, Right, Top, Bottom, Width, Height, X, Y, Location, Size`	Description d'une zone rectangulaire
`RectangleF`	Idem	Idem exprimé en décimal

Si par défaut GDI+ interprète les coordonnées en pixels cela n'est plus une restriction du système lui-même. Il existe en effet trois systèmes de coordonnées s'additionnant : *world coordinates*, *Page* et *Device*.

Lorsqu'on indique des coordonnées à GDI+ comme celles d'un point, nous sommes en mode *world coordinates*. C'est-à-dire un mode découplé de la réalité de l'affichage ou de l'impression, un « monde » (*world*) ayant un sens localement pour l'application. Mais lorsqu'il s'agit de traduire les coordonnées sur le périphérique de sortie, GDI+ peut appliquer des transformations. Par défaut il n'y a aucune transformation et on peut considérer, pour un travail à l'écran par exemple, que les coordonnées sont exprimées en pixels. Mais si le programmeur fixe des transformations, appelées *world transformations*, GDI+ transformera les coordonnées en fonction de celles-ci. Dès lors une distance horizontale de « 10 unités A »

entre deux points pourra fort bien devenir dans le système transformé une distance de « 120 unités B » en diagonal (effet d'échelle et effet de déformation conjugués).

Cette première transformation convertit les *world coordinates* en coordonnées de type page. Reste ensuite à convertir ces dernières en coordonnées propre au périphérique (*device coordinates*) ce que GDI+ prend aussi en charge. Cette transformation est appelée *page transformation*. Le chemin de transformation des coordonnées de GDI+ est illustré figure 16.1.

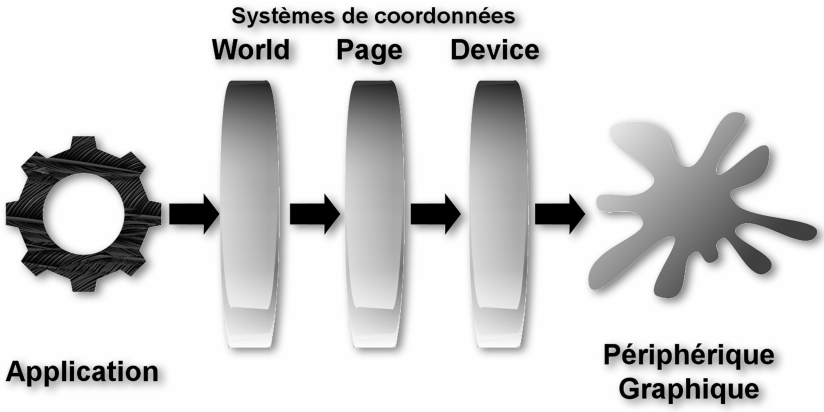

Figure 16.1

Les transformations de coordonnées sous GDI+

En fixant le mode de coordonnées le développeur peut contrôler comment les positions seront interprétées *in fine*. Par exemple s'il choisit l'unité `Inch` au niveau de la propriété `PageUnit` de l'objet `Graphics`, toutes les coordonnées seront traduites en pouces quel que soit le périphérique de sortie. Un objet de 1 pouce à l'écran mesurera aussi 1 pouce s'il est imprimé, peu importe la résolution de l'imprimante. Il est possible de combiner les effets de tous ces procédés pour fixer par exemple les coordonnées de la page en pouces en même temps qu'une matrice de translation est affectée avec un facteur différent pour les abscisses et les ordonnées. Tout cela peut très vite devenir complexe pour qui les cours de mathématiques ne sont plus très frais en mémoire... GDI+ ou non, la programmation des graphiques reste principalement fondée sur du calcul dont la sophistication reste modeste mais s'écarte des simples additions ou divisions utilisées le plus généralement en informatique de gestion…

Les transformations

Nous avons vu que le système de coordonnées se découpait en trois systèmes s'enchaînant pour partir d'une source, l'application, à une destination, le périphérique graphique. Les coordonnées world sont celles utilisées par l'application, elles sont transformées en

coordonnées de page par un processus appelé world transformation. Puis ces coordonnées pages subissent une nouvelle transformation, dite page transformation, qui aboutit à des coordonnées de type device (périphérique).

Par défaut ces trois systèmes de coordonnées ont une origine en 0,0 située dans le coin supérieur gauche de la surface.

Cette sophistication n'aurait pas de sens si elle ne servait pas un ou plusieurs buts. Et c'est bien le cas sous GDI+ puisque les transformations de coordonnées servent notamment à gérer les mises à l'échelle, les translations, les transformations, et même le choix de l'unité de mesure pour le périphérique de sortie.

Ce sont toutes ces transformations que nous allons étudier ici, elles sont essentielles à la fois par les services qu'elles rendent et par la suppression d'un code parfois complexe à écrire.

Translations et unités de mesure

La translation est la transformation la plus simple, elle s'effectue fixant la valeur de la translation sur les deux axes (X et Y) par un appel à `Graphics.TranslateTransform`.

Pour illustrer cette fonction nous allons dessiner dans l'espace d'une fiche une grille repère puis un trait. Nous pourrons appliquer la translation à la ligne ou bien à la ligne et au repère. La figure 16.02 montre la grille et la ligne affichée dans leur état par défaut, c'est-à-dire dans un mode où les trois systèmes de coordonnées sont identiques et où aucune translation n'est appliquée. Passons pour le moment le dessin de la grille des axes et regardons uniquement l'instruction qui dessine la ligne diagonale :

```
e.Graphics.DrawLine(pen.create(color.Red),0,0,200,200);
```

Cette instruction est placée dans le gestionnaire de l'événement `Paint` de le fiche (Windows Forms dans cet exemple, sous Delphi.NET). Au sein de cet événement nous recevons un paramètre `e`, objet décrivant l'ensemble des informations spécifiques à l'événement `Paint`. L'une des propriétés de cet objet s'appelle `Graphics` et retourne une instance de la classe de même nom affectée à la surface de la fiche. Il suffit ainsi de l'utiliser pour dessiner sur cette dernière. L'ordre lui-même que nous utilisons ici est `DrawLine`, l'une des nombreuses méthodes de la classe `Graphics`. Le premier paramètre est le stylo à utiliser, nous le créons à la volée en initialisant sa couleur en rouge. Les quatre autres valeurs sont les coordonnées des points de départ et arrivé de la ligne à tracer. On notera que nous utilisons ici l'une des variantes disponibles de `DrawLine`, d'autres versions permettent de passer des structures `Point` ou `PointF` pour des

coordonnées décimales plutôt qu'entières. Le résultat de cet action est le dessin d'une ligne allant du point 0,0 au point 200,200, soit une diagonale partant du coin en haut à gauche de la fiche et s'arrêtant à 200 pixels à gauche et autant en partant du haut de l'espace client de la fiche.

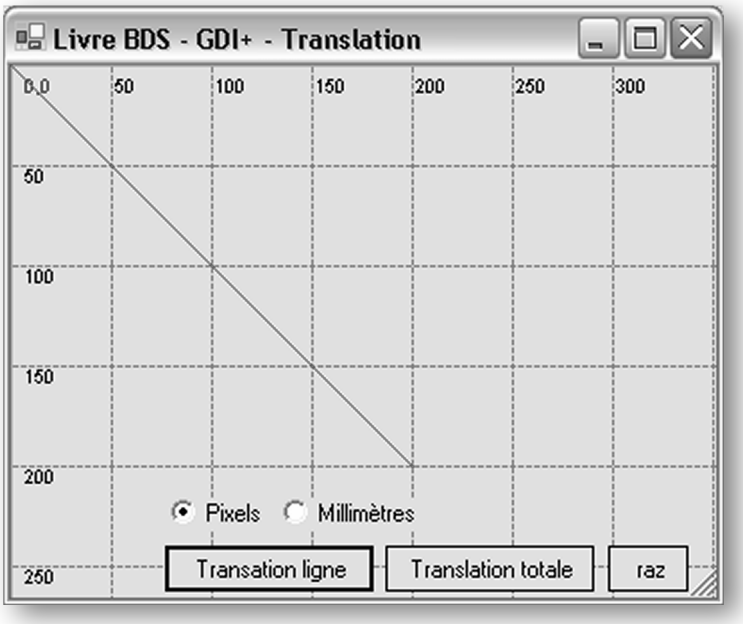

Figure 16.2

La grille répère et la ligne sans aucune translation

En cliquant sur le bouton « translation ligne » nous allons appliquer uniquement à la ligne une translation de 100 unités (ici des pixels) sur l'axe des abscisses et de 50 unités (toujours des pixels) sur l'axe des ordonnées. Le résultat est montré figure 16.3 où l'on peut voir que la ligne, bien que toujours dessinée de la même façon, par le même ordre `DrawLine` avec pour origine 0,0 et pour extrémité 200,200, se trouve décalée de 100 pixel à droite et 50 pixels vers le bas. Si la grille n'est pas affectée par la translation c'est que nous avons appliqué la translation après le dessin de cette dernière, ce qui est une technique intéressante. La translation en elle-même est triviale puisqu'elle est opérée par l'instruction suivante :

```
e.Graphics.TranslateTransform(100,50);
```

Le premier paramètre indique la translation sur l'axe des abscisses et le second sur l'axe des ordonnées.

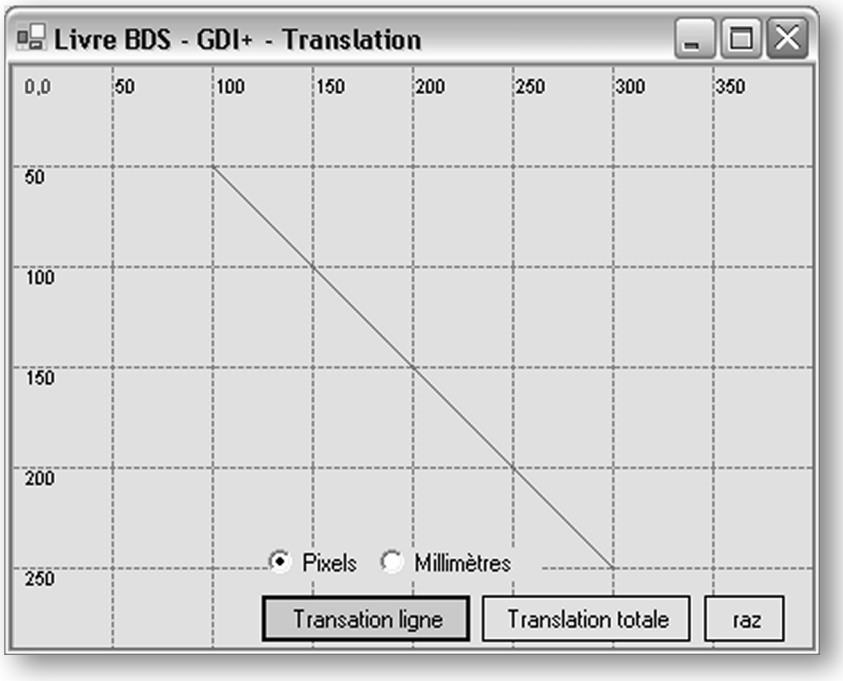

Figure 16.3

Translation appliquée à la ligne

Si nous appliquons la translation en début de séquence de dessin afin qu'elle s'applique aussi à la grille repère nous obtenons ce que montre la figure 16.4. Désormais le dessin de la grille repère est en accord avec la ligne et montre bien une origine en 0,0 pour celle-ci, c'est bien tout l'ensemble qui est maintenant affecté par la translation.

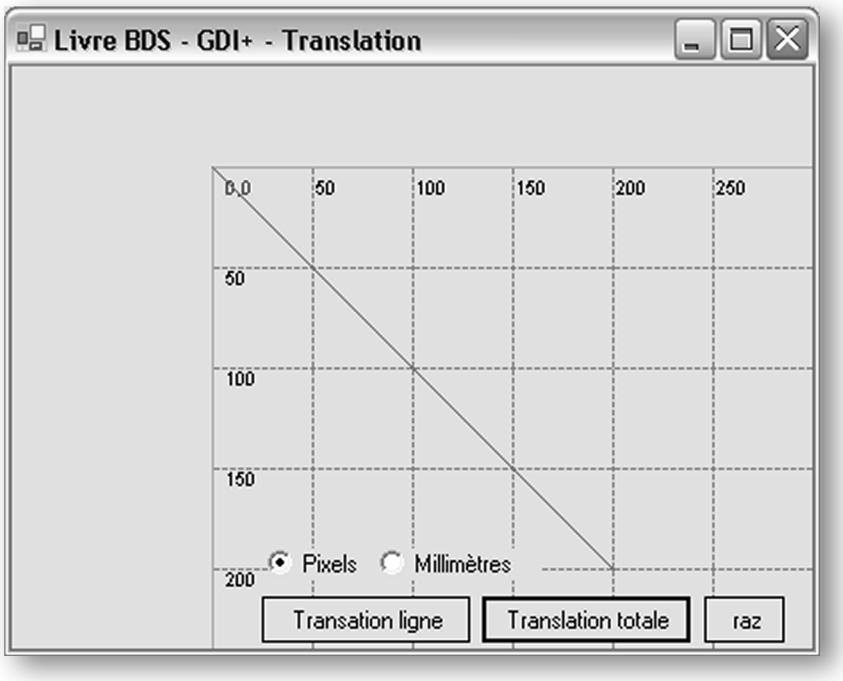

Figure 16.4

Translation appliquée à l'ensemble du dessin

Les unités servant à dessiner la grille et la ligne sont celles par défaut, soit des pixels. Et comme nous l'avons dit, par défaut aussi, l'ensemble des trois systèmes de coordonnées est aligné sur la même base et la même unité de mesure.

Tout se transforme !

Si nous utilisons pour nos démonstrations des graphiques rudimentaires telles des lignes il faut savoir que sous GDI+ tout se transforme par application des mêmes procédés, de la même logique. Ainsi les points, les rectangles, les courbes, le texte, les couleurs, les textures, et même les images peuvent subir translations et transformations. Le terrain de jeu que déroule devant nous GDI+ est tellement vaste que nous n'en verrons qu'une parcelle. À vous d'investiguer en vous distrayant. Dessiner a toujours été le passe-temps favori des enfants et les informaticiens sont de grands enfants…

Toutefois il nous est possible de décider de travailler en millimètres plutôt qu'en pixels. Nous voulons que chaque unité désigne un millimètre à l'écran. Donc un morceau de ligne de dix pixels devra, à l'écran, dessiner un segment mesurant exactement un centimètre, ce qui peut être constaté avec une règle.

Pour appliquer une unité différente il faut modifier la propriété `PageUnit` de l'objet `Graphics`, ce qui sera pris en considération par la transformation de coordonnées de type page. Si nous revenons à l'affichage par défaut de notre exemple (voir figure 16.2) mais en appliquant une unité de page millimétrique, le résultat correspond à un zoom avant (un millimètre étant plus grand qu'un pixel écran), ce qui est visible figure 16.5. Le zoom est tel que nous voyons en très gros le coin supérieur gauche de l'affichage de la figure 16.2.

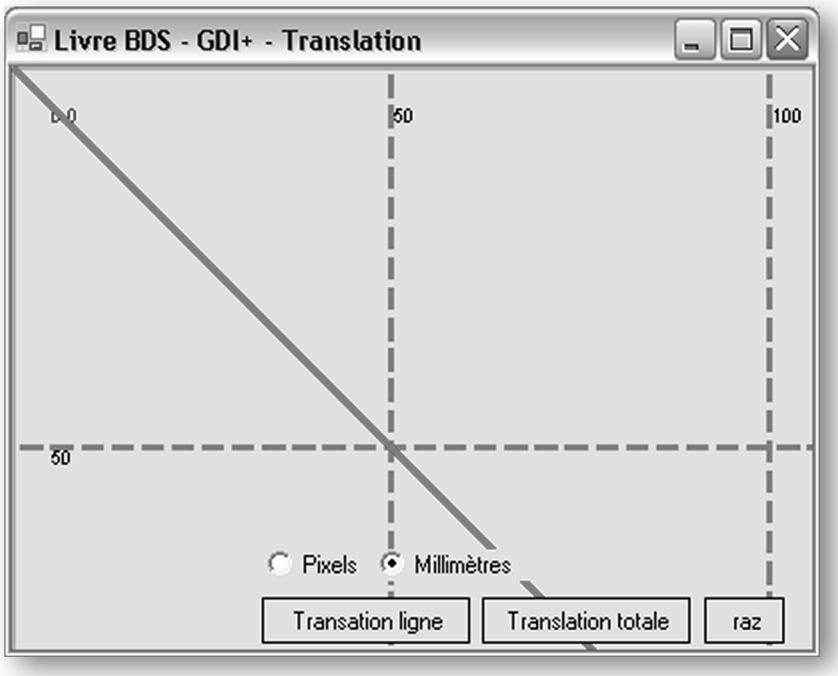

Figure 16.5

Passage des coordonnées en mode millimétrique

Notre séquence de dessin n'a en aucun cas été modifiée, notamment la ligne commence toujours en 0,0 et se termine toujours en 200,200. Toutefois ces coordonnées sont désormais interprétées en millimètre sur le périphérique de sortie et non plus en pixel. D'ailleurs, l'auteur ne reculant devant aucun sacrifice, il a vérifié que sur son écran 50 unités deviennent bien 50 millimètres soit 5 centimètres. La figure 16.6 nous le prouve (ce n'est pas un montage mais bien un cliché, la règle tenue dans une main et l'appareil photo dans l'autre…). Et ce qui est vrai ici pour cet écran en particulier qui possède une résolution de 96 dpi, serait vrai sur un écran plus classique en 72 dpi et resterait vrai si nous imprimions le graphique sur une jet d'encre ou une laser 600 dpi. Avec GDI+ il est ainsi possible de concevoir des graphiques et des états imprimés très précis, à la fraction de millimètres près (c'est pourquoi le système de coordonnées accepte des valeurs décimales), le tout sans effort.

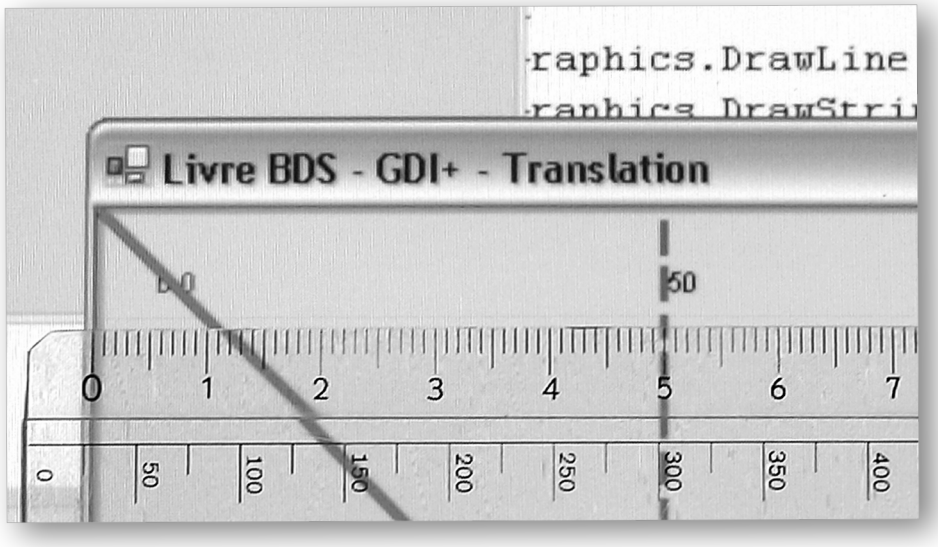

Figure 16.6

Une mesure conforme à l'attente

Les matrices de transformation

Les matrices sont des objets mathématiques rectangulaires constitués de lignes et colonnes contenant des valeurs dont la position dans la grille ainsi formée possède une signification particulière. Du point de vue de la programmation une matrice est représentée sous la forme d'un tableau de numériques à deux dimensions.

GDI+ utilise ce concept pour effectuer des transformations. Ce procédé est très courant en programmation graphique et la prise en charge directe par les API de ces objets est un soulagement appréciable.

Du côté du framework les matrices GDI+ sont représentées par la classe `Matrix` (espace de nom `System.Drawing.Drawin2D`), un tableau de dimensions 3x3 représentant une transformation affine. La classe possède plusieurs constructeurs surchargés permettant l'initialisation de la matrice en utilisant des entiers, des décimaux, des points, des rectangles ou des combinaisons de telles valeurs.

Transformation affine

Une transformation affine est une transformation qui conserve la colinéarité, c'est-à-dire qu'un point sur une ligne avant transformation reste sur une ligne après transformation, et qui conserve le ratio des distances, c'est-à-dire qu'un point se trouvant au milieu de deux autres avant la transformation est toujours au milieu de ces deux points après transformation. La contraction ou l'expansion, la dilatation, la réflexion, la rotation, le cisaillement et d'autres transformations sont des transformations affines.

Comme toute classe, `Matrix` expose des propriétés et des méthodes. Ses propriétés permettent d'obtenir la liste des éléments (`Elements`) sous la forme d'un tableau ou l'offset sur les deux axes (`OffsetX` et `OffsetY`). Les méthodes offertes couvrent les besoins les plus fréquents en matière de calcul matriciel comme l'inversion (`Invert`), la multiplication (`Multiply`), la rotation (`Rotate`), le cisaillement (`Shear`), etc.

Dans le même esprit que nous avons traité l'exemple sur les translations nous allons créer une fiche avec une grille repère et nous allons dessiner un objet, ici un rectangle plein, que nous transformerons en lui appliquant une matrice. L'image obtenu par défaut est celle montrée figure 16.7.

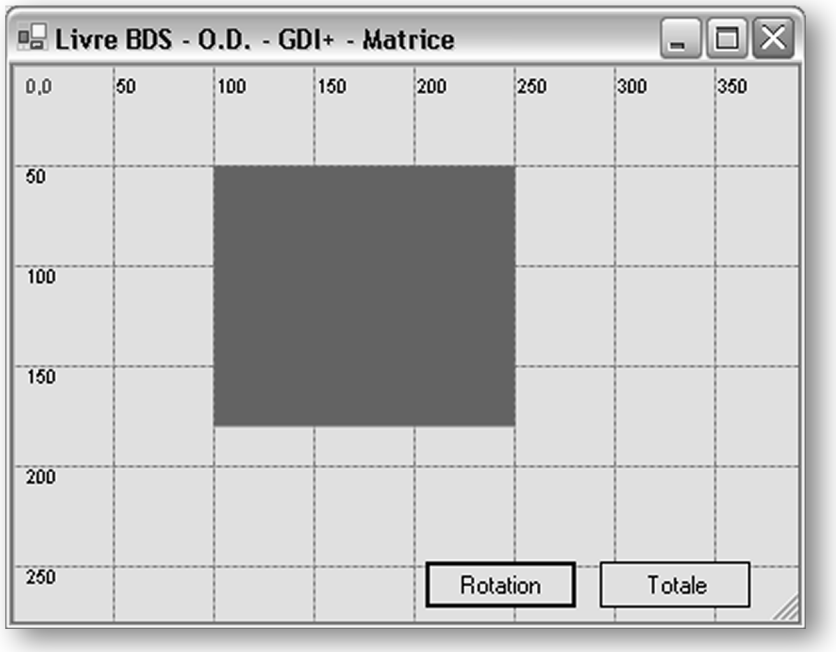

Figure 16.7

Le rectangle avant l'application de la transformation

Le principe reste identique à celui de l'application de l'exemple précédent, tout le travail est effectué dans la méthode `Paint` de la fiche. Le rectangle est dessiné de la façon la plus simple en créant une zone remplie d'une couleur :

```
e.Graphics.FillRectangle(Brushes.BlueViolet,100,50,150,130);
```

`FillRectangle` est l'une des innombrables méthodes de la classe `Graphics`, elle existe en plusieurs variantes. Celle utilisée ici prend en paramètre une brosse, la position X,Y du coin supérieur gauche de la figure et ses dimensions (largeur et hauteur respectivement).

En réalité, avec le principe des matrices de transformation de GDI+, nous n'allons pas faire une rotation du rectangle lui-même nous allons associer une matrice de rotation à l'espace de dessin, ce qui est différent. Tous les tracés que nous ferons ensuite seront affectés par la matrice sans que nous n'ayons une seule ligne de code à changer.

Pour créer une matrice de rotation nous utilisons le code suivant :

```
Var p :PointF ; m : Matrix ;
...
p := PointF.Create(100+(150/2),50+(130/2));
m := Matrix.Create;
m.RotateAt(60,p);
e.Graphics.Transform := m;
```

Il existe deux type de rotation, celle que nous utilisons ici, `RotateAt`, permet de fixer le centre de rotation. Nous avons ainsi

besoin d'un point pour désigner ce centre et c'est la raison d'être de la variable `p` de type `PointF` qui est initialisée par la création d'une instance de cette classe en fournissant les coordonnées X,Y du point. Ces valeurs sont calculées ici pour placer le point au centre du rectangle.

Ensuite nous créons une instance de la classe `Matrix` et nous invoquons sa méthode `RotateAt`. Le premier paramètre est l'angle, le second le point qui sera utilisé comme centre de rotation. Il ne nous reste plus qu'à affecter l'objet matrice ainsi initialisé à l'instance de `Graphics` passée en paramètre de la méthode `Paint` de la fiche. Dès lors tous les tracés effectués seront affectés par la matrice. Ce qui sera le cas du rectangle de cet exemple s'il est dessiné après avoir affecté la matrice. Tout comme pour l'exemple précédent nous dessinerons la grille repère avant la transformation pour bien visualiser l'effet de celle-ci mais un bouton permettra de rendre la transformation globale et d'affecter aussi la grille repère.

La figure 16.8 montre l'effet de la rotation sur le rectangle. Rappelons que nous le dessinons toujours de la même façon en passant toujours les mêmes coordonnées, la rotation est assurée automatiquement par GDI+ qui transforme toutes les coordonnées conformément à la matrice.

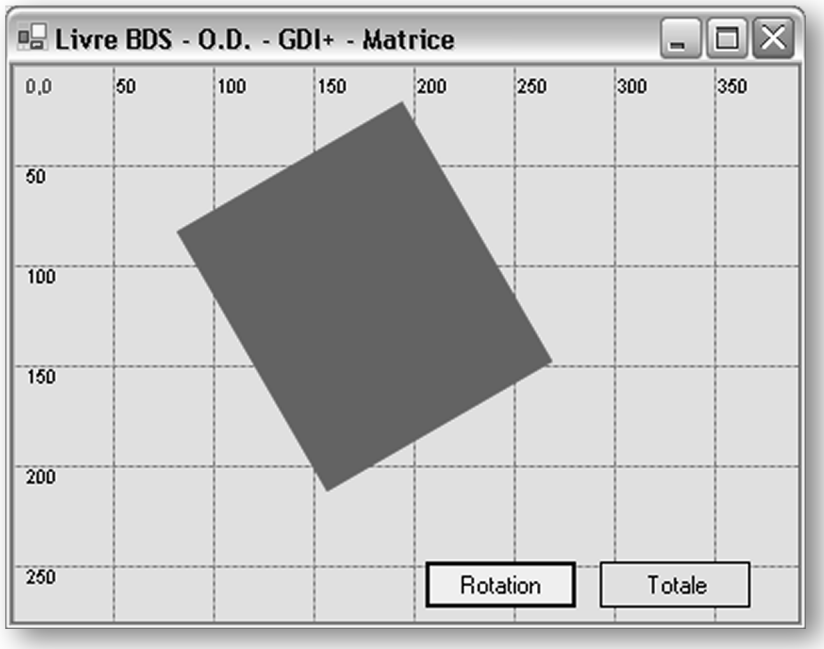

Figure 16.8

Application de la matrice de rotation au rectangle

La figure 16.9 montre ce qui arrive si nous affectons la matrice de transformation avant le dessin de la grille repère : cette dernière est elle aussi affectée par la rotation. Vous remarquerez que la grille

comporte des tracés horizontaux et verticaux, des lignes hachurées et des indications textuelles (les coordonnées) et que tout cela est automatiquement affecté par la matrice. C'en est presque magique…

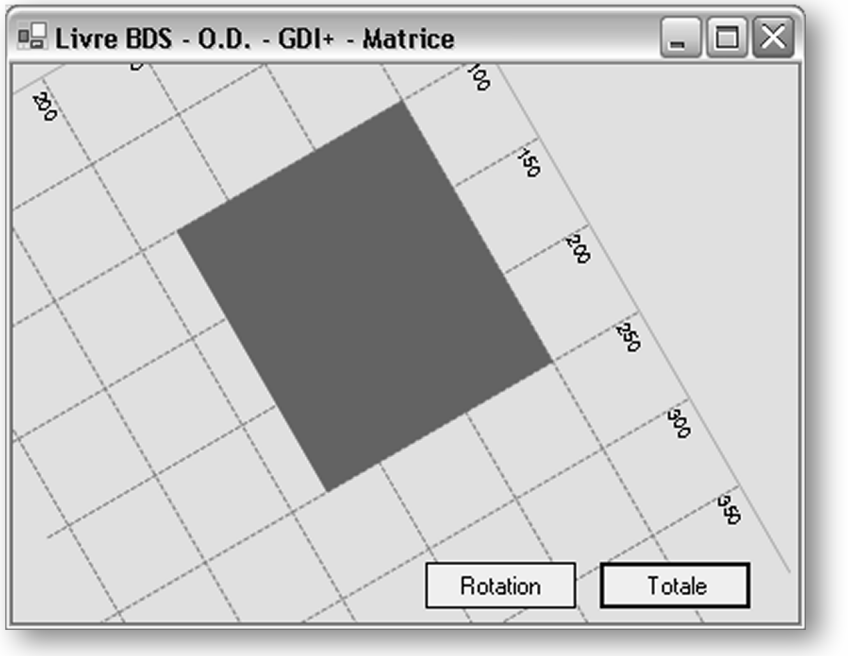

Figure 16.9

La matrice affecte l'ensemble du dessin

Les matrices permettent bien d'autres transformations comme le changement d'échelle par la méthode `Scale` qui s'utilise de la même façon que la méthode `RotateAt` montrée ici. La classe `Graphics` possède aussi des méthodes de transformation qui peuvent être utilisées pour affecter l'ensemble des tracés. Elles fonctionnent de la même façon.

Le choix d'effectuer les transformations depuis une instance de la classe `Graphics` ou directement comme nous l'avons fait ici dépend de ce qu'on souhaite obtenir. Les transformations effectuées depuis une instances de la classe `Graphics` sont dites globales puisqu'elles affectent sans distinction tous les tracés effectués. La propriété `Transform` est alors utilisée pour fixer les transformations globales.

Les transformations composites sont des transformations globales enchaînées. Les effets des propriétés `MultiplyTransform`, `RotateTransform`, `ScaleTransform` et `TranslateTransform` de l'objet `Graphics` peuvent être cumulée pour obtenir des transformations globales complexes sans avoir à les coder.

Les transformations locales sont celles qui n'affectent qu'un objet en particulier, par exemple un texte, une image, un chemin graphique ou autre. Tous les autres objets dessinés ne sont pas concernés par la transformation. Cela ouvre des possibilités infinies tout en simplifiant grandement l'écriture du code.

Le traitement des images

GDI+ permet de traiter les images comme tout autre objet et ce depuis et vers des formats beaucoup plus nombreux que ne le permettait GDI. Ainsi, les transformations vues plus haut sont tout à fait applicables aux images. Pour simple exemple, la figure 16.10 montre une image JPEG modifiée par une matrice de transformation cumulant à la fois une rotation et un changement d'échelle (doublement des longueurs sur l'axe des X et conservation de l'axe des Y sans changement).

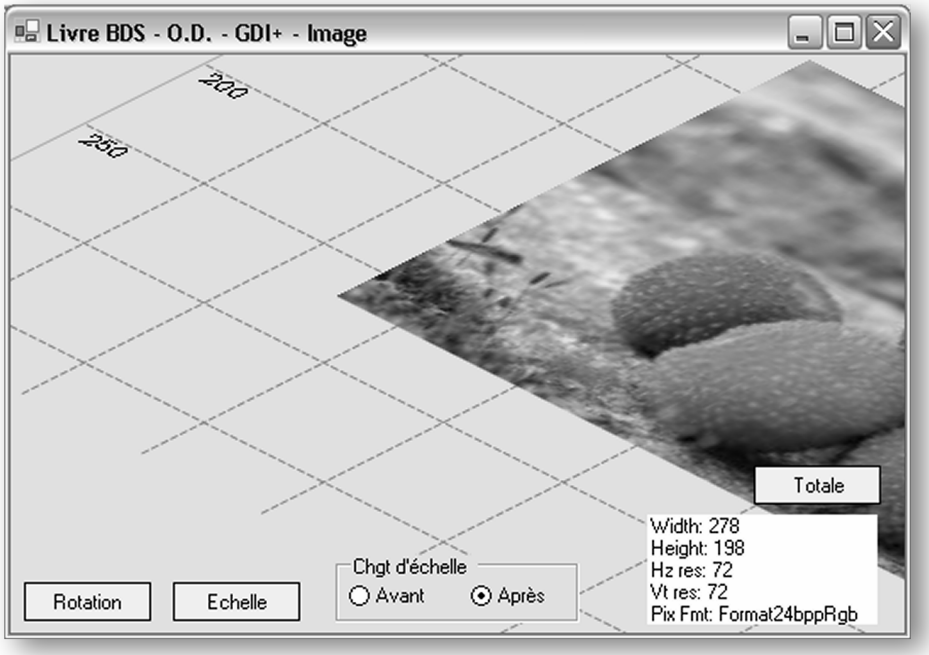

Figure 16.10

Transformation d'une image

Dans l'exemple présenté figure 16.10 nous vous montrons l'effet cumulé des deux transformations matérialisées par deux boutons « Rotation » et « Echelle ». Le cadre en bas à droite liste les principales caractéristiques de l'image, notamment ses dimensions, sa résolution et sa profondeur en bits par pixel. Le bouton « Totale » a la même fonction que dans les exemples précédents, à savoir appliquer ou non les transformations à la grille repère. Sur l'image présentée ce bouton a été cliqué et on peut voir l'effet de la déformation sur la grille, ses repères, et le texte qui jalonne les graduations.

La création du bitmap contenant l'image qui sera traitée est effectuée de la façon suivante :

```
var b : System.Drawing.Bitmap ;
b := system.Drawing.Bitmap.Create('arbouses.jpg');
```

Le cadre des informations de l'image est une liste (classe `ListBox`) remplie en utilisant les propriétés de l'objet bitmap :

```
listbox1.Items.Add('Width: '+b.Width.tostring);
listbox1.Items.Add('Height: '+b.Height.tostring);
listbox1.Items.Add('Hz res: '+b.HorizontalResolution.tostring);
listbox1.Items.Add('Vt res: '+b.VerticalResolution.tostring);
listbox1.Items.Add('Pix Fmt: '+
                     system.object(b.PixelFormat).tostring);
```

L'affichage est effectué dans le gestionnaire de l'événement `Paint` de la fiche (Windows Form) qui nous passe en paramètre un argument `e` possédant un objet `Graphics` relié à la surface de la fiche. Pour afficher le bitmap nous utilisons l'instruction suivante :

```
e.Graphics.DrawImage(b, r );
```

La variable `b` est le bitmap qui a été créé précédemment. La variable `r` est de type `System.Drawing.Rectangle`, elle permet de définir un rectangle dans lequel l'image sera affichée. La méthode `DrawImage` possède des dizaines de variantes, celle utilisée ici n'en est qu'une parmi d'autres que nous vous laissons découvrir.

Le rectangle que nous créons est calculé pour afficher automatiquement l'image au centre de l'espace de la fiche :

```
 r := rectangle.Create(
 (self.ClientRectangle.width-b.size.Width) div 2,
 (self.ClientRectangle.height-b.size.Height) div 2,
  b.size.width,b.size.Height);
```

Enfin, la transformation est effectuée par le biais d'une matrice créée de la façon suivante :

```
m := Matrix.Create;
p :=
   PointF.Create(self.ClientRectangle.width / 2,
   self.ClientRectangle.height / 2);
m.RotateAt(45,p);
if ScalePrepend then m.Scale(2,1,Matrixorder.Prepend)
                else m.Scale(2,1,Matrixorder.Append)
e.Graphics.Transform := m; // affectation de la matrice
```

La variable `m` est de type `Matrix`, la variable p de type `PointF`. Le point est utilisé pour définir le centre de la transformation de rotation. Nous le calculons de telle façon à ce qu'il se trouve au centre de l'image.

Le changement d'échelle est effectué en cumulant l'effet de la méthode `Scale` de la matrice avec l'effet de `RotateAt` appliqué juste avant. Les paramètres de la méthode `Scale` sont, respectivement, le facteur sur l'axe des X (ici 2 donc doublement des tailles apparentes), facteur sur l'axe des Y (ici 1 donc aucun changement) et enfin l'ordre dans lequel l'effet que nous ajoutons est traité vis-à-vis des effets déjà enregistrés dans la matrice.

Le rôle de la variable `ScalePrepend` de type booléen est justement de permettre la mise en évidence de la différence de rendu final selon le choix. La variable est basculée par le groupe de boutons radio visible en bas de fiche.

Toutes les transformations sont applicables aux images comme aux autres éléments graphiques offrant une large palette d'effets prêts à l'emploi et ne nécessitant presque pas de code.

Le traitement des couleurs

Les matrices de transformation ne s'appliquent pas seulement aux objets graphiques et aux images, elles peuvent aussi être appliquées aux composantes couleurs. Vous allez peut être vous demander à quoi peuvent servir une translation, une rotation, un changement d'échelle sur des couleurs… Selon l'effet, selon la matrice cela va permettre de modifier la balance RGB de l'image, de la rendre partiellement transparente, d'augmenter le bleu si elle tire trop sur le rouge, etc. Autant de traitements essentiels qu'on retrouve d'ailleurs dans la plupart des logiciels de dessin.

Sous GDI+, comme nous l'avons déjà dit, les couleurs sont représentées sur 4 octets, les trois composantes du système RGB et la composante alpha. Chaque canal acceptant des valeurs de 0 à 255, ce qui nous offre 256 niveaux de transparence pour chacune des 16.777.216 de couleurs (256^3). Les matrices de transformation des couleurs acceptent une autre notation dans laquelle chaque paramètre évolue entre 0 et 1 en décimal.

Pour modifier les couleurs d'une image par une matrice de transformation GDI+ nous offre une classe spécifique, `ColorMatrix`. Cette dernière définit une matrice 5x5 (une matrice 4x4 ne pouvant permettre que des transformations linéaires ce qui serait limitatif). Dans une telle matrice, chaque ligne représente une composante, la cinquième ligne servant aux translations notamment devant avoir tous ces éléments à 0 sauf le dernier. Une matrice identité (ne faisant aucune transformation) est donc remplie de zéros avec une ligne diagonale de « 1 » du coin supérieur gauche au coin inférieur droit. La figure 16.11 donne l'exemple de deux matrices de transformation des couleurs. La première, à gauche, est une matrice identité, la seconde, à droite,

modifie les composantes couleurs de la façon suivante : la composante rouge est non modifiée (1), la composante verte est multipliée par 2, la composante bleue est multipliée par 3 et l'opacité est divisée par 2 (multipliée par 0,5).

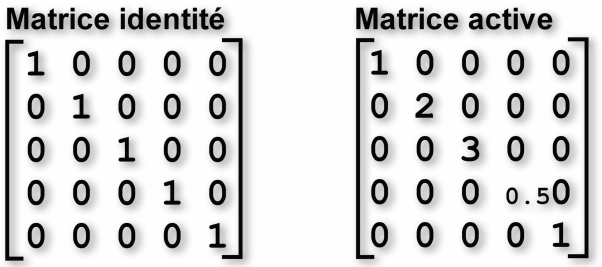

Figure 16.11

Exemples de matrices de transformation des couleurs

Si la dernière ligne de la seconde matrice était la suivante :

```
10 15 12 0 1
```

cela permettrait non plus de multiplier mais d'ajouter des valeurs aux composantes. Dans cet exemple on ajouterait 10 à la composante rouge, 15 à la composante verte, 12 à la composante bleue et rien à la composante alpha. Les multiplications et additions peuvent être cumulées dans la même matrice.

En utilisant un objet `ImageAttributes` auquel on associe une matrice de modification des couleurs par sa propriété `SetColorMatrix` on peut afficher le dessin d'une image en y appliquant les transformations programmées dans la matrice. L'objet `ImageAttributes` est alors utilisé en tant que paramètre de la méthode `DrawImage` de l'objet `Graphics`.

En quelques lignes de code il est ainsi possible de recréer les fonctions de correction RGB d'un logiciel de dessin, c'est ce que montre la figure 16.12.

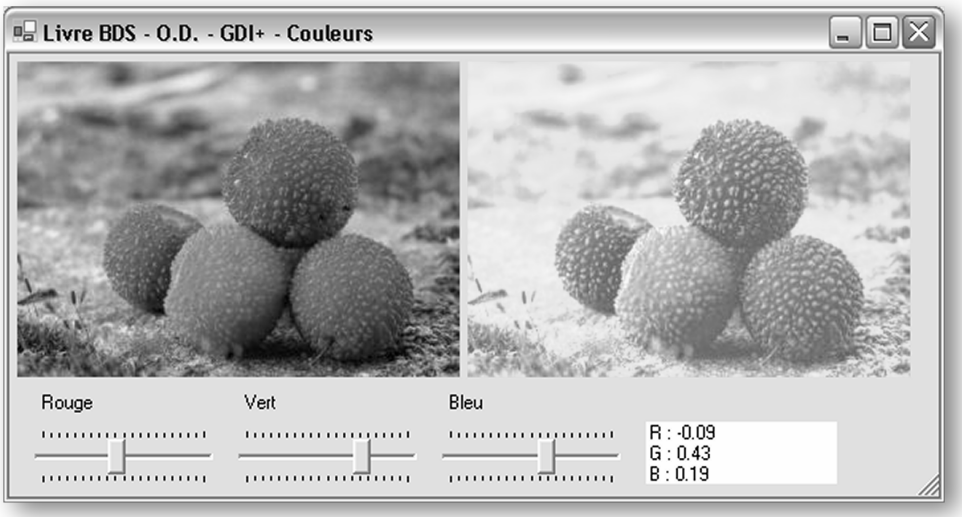

Figure 16.12

Correction RGB par une matrice de transformation de couleurs

Le code nécessaire à cet exemple comporte bien entendu des artifices liés à l'interface, nous en ferons abstraction d'autant que, comme pour tous les exemples de ce livre, vous retrouver le code original complet sur le cd-rom fourni.

Voici le gestionnaire de l'événement `Paint` de la fiche :

```
procedure TWinForm4.TWinForm4_Paint(sender: System.Object;
                  e: System.Windows.Forms.PaintEventArgs);
 var r1,r2:rectangle;
     b : System.Drawing.Bitmap;
     cm : ColorMatrix;
     matrice : array[,] of single;
     m : array of array of single;
     IA : ImageAttributes;
     i,ii : integer;
begin
  b := system.Drawing.Bitmap.Create('arbouses.jpg');
  r1 := rectangle.Create(5,5,b.size.width,b.size.Height);
  r2 :=
   rectangle.Create(10+b.size.width,5,b.size.width,b.size.Height);
  e.Graphics.DrawImage(b, r1 );
  matrice := new(array[,] of single, ((1, 0, 0, 0, 0),
                                      (0, 1, 0, 0, 0),
                                      (0, 0, 1, 0, 0),
                                      (0, 0, 0, 1, 0),
                                      (cR,cG,cB,0,1)));
  setlength(m,5,5);
  for i:= 0 to 4 do for ii:=0 to 4 do m[i,ii] := matrice[i,ii];
  cm := ColorMatrix.Create(m);
  IA := ImageAttributes.Create;
  IA.SetColorMatrix(cm);
  e.Graphics.DrawImage(b, r2,
        0,0,b.width, b.Height, GraphicsUnit.Pixel, IA );
  b.Dispose;
  IA.Dispose;
end;
```

Éclaircissons les lignes de code présentées ci-dessus :

Tout d'abord nous créons un objet bitmap tout en le chargeant depuis une image disque au format JPEG. L'image est maintenant disponible dans la variable `b`.

Ensuite nous créons deux rectangles. Le premier servira à afficher l'image originale, le second l'image transformée. Ces rectangles sont stockés dans les variables `r1` et `r2`.

L'image non modifiée est alors affichée par la méthode `DrawImage` de l'objet `Graphics` en utilisant le rectangle `r1`.

Pour l'image modifiée il nous faut maintenant créer une matrice. Cette dernière est créée en utilisant une syntaxe d'initialisation introduite depuis Delphi 2005. Cela permet de mieux visualiser la matrice que si nous avions initialiser chaque cellule l'une après l'autre.

Toutefois il reste un petit problème qui sera peut être résolu dans la version de BDS en votre possession. La nouvelle syntaxe des tableaux dynamiques multidimensionnels se note comme en C# (`array[,]` pour un tableau à deux dimensions), or cette notation n'est pas compatible avec les `array of array`. La méthode qui attend la matrice en paramètre accepte un paramètre de ce dernier type ce qui créé une difficulté. Pour ne pas nous priver de la clarté de la nouvelle initialisation des tableaux dynamique tout en restant compatible avec le type attendu, nous commençons par créer la variable matrice avec la nouvelle syntaxe. Nous utilisons ensuite une double boucle `for` pour transférer les éléments de cette matrice vers la variable `m` qui sera utilisée in fine pour les opérations de dessin. Cela n'est pas très propre mais cette façon de faire est toujours préférable en terme de lisibilité du code que d'utiliser l'ancienne méthode qui aurait consisté à initialiser chacune des 25 cellules l'une après l'autre.

Puis un objet `ColorMatrix` est créé et initialisé à l'aide de la matrice. Dans la foulée un objet `ImageAttributes` est lui aussi créé. Nous utilisons sa méthode `SetColorMatrix` pour lui transmettre la matrice de transformation des couleurs.

Enfin l'image est dessinée en utilisant le second rectangle `r2` et les objets graphiques sont relâchés par un appel à leur méthode `Dispose`.

Revenons un instant sur la matrice, lorsque nous l'initialisation nous lui passons une matrice identité (une diagonale de 1 parmi des 0) mais sa dernière ligne commence par des noms de variables : `cR`, `cG`, `cB`. Chacune de ces dernières est le paramètre correcteur pour une composante (rouge, vert, bleu). Nous les avons définis au niveau de l'objet fiche et c'est la manipulation des curseurs (`Trackbar`) qui en modifie les valeurs puis invalide la fiche pour qu'elle se redessine (appel à `Invalidate`).

Autant dire que cette méthode, s'il est simple à mettre en œuvre, n'est pas à conseiller dans une application réelle. Si vous tester l'application (ce que nous vous conseillons de faire), vous verrez que tout déplacement de curseur provoque un scintillement très peu esthétique. Cela est du au fait que nous dessinons tout dans l'événement `Paint` de la fiche et directement sur l'espace de celle-ci. Une application réelle afficherait l'image non modifiée une seule fois, dans un composant `PictureBox` par exemple, au lieu de la redessiner à chaque fois et elle utiliserait certainement un

système de temporisation pour éviter que l'image corrigée ne soit redessinée systématiquement à chaque modification de la valeur d'un curseur alors que ce dernier est encore en cours de déplacement. Mais il s'agit là de méthodes classiques qui n'ont rien de spécifiques à GDI+.

Une fois le principe des matrices de transformation de couleurs compris on peut l'utiliser dans de nombreuses situations en exploitant toutes les possibilités liées aux matrices : rotation, cisaillement, etc.

Le texte

Le texte se manipule aussi simplement que le reste sous GDI+ et, comme tout objet graphique, il peut être soumis à des transformations de tout genre : mise à l'échelle, rotation, cisaillement, etc.

En réalité nous avons déjà traité cet aspect dans le présent article : de nombreux exemples affiche une grille repère en fond d'écran pour mieux matérialiser les transformations. Cette grille n'est autre qu'une série de lignes dessinées par la méthode `DrawLine` et du texte indiquant régulièrement les valeurs sur chaque axe. Ce texte est dessiné par la méthode `DrawString`. Comme vous l'avez vu au fil des exemples précédents, lorsqu'une transformation est appliquée avant le dessin de la grille repère, le texte lui aussi suit le mouvement et se trouve translaté, déformé ou affecté par une opération de rotation.

Dès lors il serait inutile de reprendre les mêmes exemples pour vous démontrer les mêmes effets. Regardez à nouveaux ceux que nous présentons plus haut, jouer avec le code source et profitez d'un système totalement cohérent où les mêmes méthodes s'appliquent aussi bien à une image qu'à un bout de texte…

Un exemple de syntaxe de manipulation de texte pour ne pas vous laisser totalement sur votre faim si vous n'avez pas votre ordinateur sous la main pour inspecter le code source fourni sur le cd-rom :

```
e.Graphics.DrawString('0,0',
       system.Drawing.Font.Create('arial', 7),
       system.Drawing.SolidBrush.Create(Color.Blue), 5, 5);
```

Cette instruction est celle qui est utilisée dans le dessin de la grille repère pour écrire « 0,0 » à l'origine des deux axes. Les paramètres sont, dans l'ordre : le texte à afficher, l'objet fonte (créé ici directement), l'objet brosse (créé aussi directement) et les coordonnées.

Si l'objet Graphics est modifié par des transformations, le texte le sera aussi.

Les brosses et les crayons

Jusqu'ici nous en avons utilisés mais nous n'en avons pas parlé, rendons leur justice, il est temps d'étudier les brosses et les crayons.

En effet, la majorité des dessins, en dehors de l'affichage des bitmap, utilise des brosses, qui définissent le remplissage, et des crayons, qui définissent le contour.

Les brosses

Ce sont les brosses qui définissent la façon dont l'intérieur d'une figure, d'un tracé (lorsqu'il est fermé), est dessiné. Lorsqu'on remplit une zone rectangulaire par exemple il faut un crayon (que nous verrons plus loin) pour tracer le contour de la figure et une brosse pour décrire comment l'espace intérieur sera peint.

Les brosses définissent à la fois un style de dessin et une couleur. Le style peut être plein, hachuré, strié, etc, et la couleur peut être n'importe quelle couleur supportée par GDI+. Toutefois ce dernier ajoute quelques nouveautés comme la gestion des dégradés, effet très intéressant pour donner un peu plus de vie aux surfaces ou créer des rendus en fausse 3D (comme un cylindre par exemple).

Le framework met à notre disposition de nombreux types de brosses toutes dérivées de `System.Drawing.Brush` qui est une classe abstraite ne pouvant être instanciée directement. La figure 16.13 donne un aperçu de plusieurs types de brosses différentes.

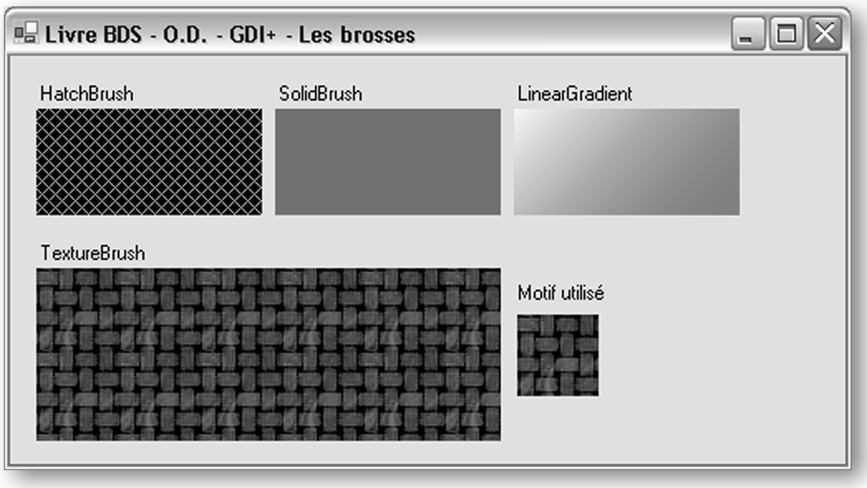

Figure 16.13

Différents types de brosses

La brosse la plus élémentaire est bien entendu celle qui définit un remplissage solide de la zone par une couleur précise. Le code ci-dessous est celui qui réalise l'affichage du second rectangle en haut de la fiche montrée figure 16.13.

```
procedure TWinForm.Panel2_Paint(sender: System.Object;
                  e: System.Windows.Forms.PaintEventArgs);
var Brosse : SolidBrush;
begin
  Brosse := SolidBrush.Create(System.Drawing.Color.RoyalBlue);
  e.Graphics.FillRectangle(Brosse,panel2.get_DisplayRectangle);
  brosse.Dispose;
end;
```

Une brosse solide se créée en instanciant la classe `SolidBrush`. Le constructeur prend en paramètre une couleur qu'on peut définir de nombreuses façons, ici nous avons pioché dans la liste des couleurs système.

Comme les brosses solides sont des objets utilisés très fréquemment le framework nous propose un stock de brosses toutes prêtes qui évitent d'avoir à appeler le constructeur. Ces brosses dérivent de la classe `System.Drawing.Brushes`. Ainsi nous aurions pu écrire le code ci-dessus de la façon suivante :

```
var Brosse : Brush;
begin
  Brosse := Brushes.RoyalBlue;
  ...
```

Les brosses s'utilisent dans de nombreux contextes et nous avons ici choisi l'un des plus simples, l'utilisation de la méthode `FillRectangle` qui remplit une zone rectangulaire en utilisant une brosse déjà définie et les coordonnées du rectangle. On remarquera qu'en fin d'utilisation de la brosse nous appelons sa méthode `Dispose` pour la libérer.

La brosse hachurée de la figure 16.13 est obtenue par le code suivant qui ne diffère du précédent que par l'utilisation d'une autre sous classe de `Brush`, `HatchBrush`.

```
procedure TWinForm.Panel1_Paint(sender: System.Object;
                  e: System.Windows.Forms.PaintEventArgs);
var Brosse : HatchBrush;
begin
  Brosse :=
   HatchBrush.Create(HatchStyle.OutlinedDiamond,
                     System.Drawing.Color.Tan);
  e.Graphics.FillRectangle(Brosse,panel1.get_DisplayRectangle);
  brosse.Dispose;
end;
```

Les brosses hachurées sont créées en indiquant, outre une couleur, un style de hachure. Ce dernier est choisi parmi les divers éléments de l'énumération `HatchStyle`.

Pour créer une brosse dégradée, la séquence reste similaire à la différence de la classe utilisée comme le montre le code ci-dessous :

```
procedure TWinForm.Panel3_Paint(sender: System.Object;
                  e: System.Windows.Forms.PaintEventArgs);
var Brosse : LinearGradientBrush;
begin
  Brosse :=
   LinearGradientBrush.Create(panel3.get_DisplayRectangle,
               Color.Yellow,Color.Red,Single(45.0));
  e.Graphics.FillRectangle(Brosse,panel1.get_DisplayRectangle);
  brosse.Dispose;
end;
```

La classe `LinearGradientBrush` permet de régler à la fois les deux couleurs du dégradé, la zone à remplir et l'angle du dégradé. La taille de la zone est celle de l'échantillon de dégradé qui est généré en mémoire. Lorsqu'on applique un échantillon plus grand ou plus petit que la zone définitive on obtient des effets différents difficiles à décrire avec des mots, à vous d'expérimenter !

À noter qu'il existe une autre brosse dégradée, `PathGradientBrush` qui au lieu de créer le dégradé dans une zone rectangulaire fait varier la couleur en suivant un `Path` graphique (chemin). Cela permet des effets qui présentaient de réelles difficultés de codage sous GDI.

Autre type de brosse particulière attractive : `TextureBrush`. Cette classe définit une brosse qui sera utilisée pour créer des effets de texture, notamment en partant d'un bitmap (généré par l'application ou lu depuis un fichier graphique). L'exemple qui occupe l'espace inférieur de la figure 16.13 utilise une telle brosse pour remplir un rectangle d'une texture dont l'échantillon d'origine est affiché à sa droite pour référence. Le code réalisant ce remplissage est le suivant :

```
procedure TWinForm.Panel4_Paint(sender: System.Object;
                  e: System.Windows.Forms.PaintEventArgs);
var img : System.Drawing.Image;
    brosse: TextureBrush;
begin
 img := Image.FromFile('brosse.jpg');
 brosse := TextureBrush.Create(img);
 img.Dispose;
 e.Graphics.FillRectangle(brosse,panel4.get_DisplayRectangle);
 brosse.Dispose;
end;
```

Les crayons

Les crayons sont avec les brosses les outils de base pour dessiner. C'est avec eux qu'on trace tous les traits et ils sont donc à ce titre d'un emploi encore plus fréquent que les brosses (tous les tracés n'ayant pas forcément de zone de remplissage).

Comme un trait est une chose simple il n'existe qu'une seule classe : `System.Drawing.Pen`. Toutefois ces propriétés en font un outil très complet et versatile comme le montre le programme exemple visible figure 16.14.

En jouant sur les propriétés des crayons on peut former facilement des lignes hachurées, des flèches, des lignes se terminant par des cercles, des triangles, etc.

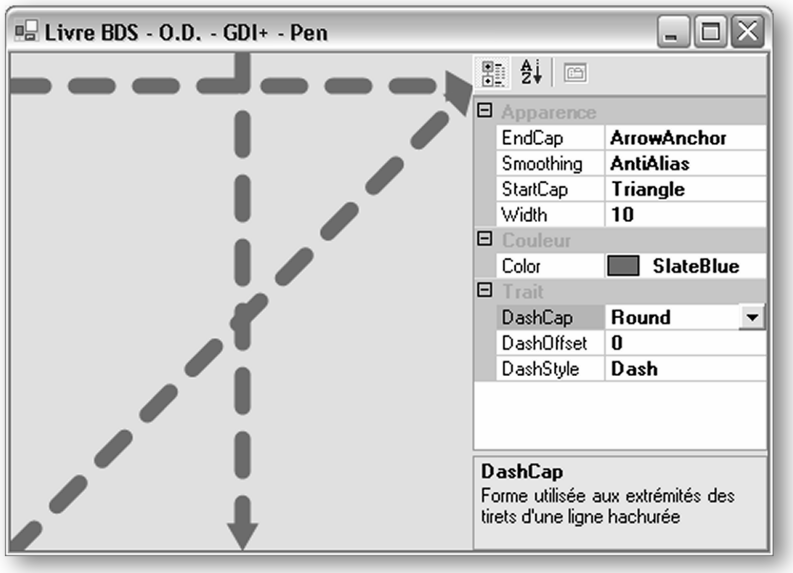

Figure 16.14

Utilisation de la classe Pen

D'un point de vue technique il n'y a pas beaucoup à dire sur cette classe qui est très simple. Nous vous conseillons plutôt de jouer avec notre application exemple. Celle-ci utilise notamment des astuces qui ne sont pas propres à GDI+ mais qui pourront vous être très utiles. Notamment nous avons conçu une classe proxy pour être affichée par la `PropertyGrid`. Lorsqu'une modification est détectée par l'instance de cette classe, elle la reporte sur l'instance de `Pen` utilisées pour dessiner les lignes dans la partie gauche de la fiche. La classe proxy utilise des attributs personnalisés pour définir des catégories et des explications affichées dans la partie inférieure de la grille.

Conclusion

GDI+ est une librairie fort bien servit par le framework .NET auquel elle n'est pas directement liée à l'origine (on peut l'utiliser depuis Delphi Win32 ou C++ notamment). La cohérence objet ajoutée par .NET en fait un ensemble très agréable à utiliser. Ici nous avons vu comment fonctionne cette librairie, comment afficher des lignes, des zones, du texte, des images, comment modifier les couleurs d'une photo, comme effectuer des rotations, des translations et bien d'autres techniques qui auraient réclamé des pages de code complexe sous GDI. GDI+ contient encore beaucoup d'autres choses que nous n'avons pas vues ici comment par exemple les courbes de bézier. En fait, en faisant l'inventaire de ce que propose System.Drawing et ses sous espaces de noms, on a l'impression de disposer d'Adobe PhotoShop et Illustrator en pièces détachées… Bien entendu entre se faire offrir une boîte de Lego et construire la tour Eiffel à l'échelle ½ il y a une certaine marge… à vous de la franchir !