Traitement d'images en .NET (partie 2)

Bitmap et performance, quelques améliorations

Frédéric MELANTOIS (Tkfé), le 7 février 2006

Droits relatif à la diffusion :
L'ensemble de cet article ainsi que les codes mis à disposition sont la propriété de leur auteur. La reproduction même partielle de tout ou partie est interdite sans accord **écrit** préalable de l'auteur.

1. Introduction.

Nous avions vu lors de la première partie les manières d'accéder au pixel en fonction des formats d'images. Dans cette partie, nous nous attacherons à comparer les performances lors de l'accès au pixel puis à réaliser les premiers traitements sur une image.
Il est important d'avoir lu et assimilé les notions de la première partie pour bien suivre cet article.

2. Chargement d'une image et performance

Les remarques que je ferais dans ce chapitre sont peu abordées dans les différents articles sur Internet traitant de la manipulation des Bitmaps en mode unsafe. Si vous en tenez compte, vous parviendrez à gagner du temps dans vos traitements.

Le format d'images est très important comme nous allons le voir, à la fois pour le traitement proprement dit mais aussi sur l'opération interne de « LockBits ». Le choix du format pour l'accès au pixel est déterminant pour la performance du traitement de l'image.

Méthodes d'accès à l'image

Souvent, notre source d'image est un fichier sur le disque dur. Nous allons voir que la manière d'accéder à cette image a toute son importance. Examinons les quelques moyens fournis par le Framework .NET (Nous ne détaillerons pas ici la lecture et le décodage de fichier directement, comme on peut le faire en C pour réaliser un traitement d'images très performant ou encore charger des images aux formats non reconnus par l'API GdiPlus) dont nous disposons pour charger en mémoire une image se trouvant sur le disque dur :

```
Bitmap bitmap = (Bitmap)Bitmap.FromFile(@"G:\image24bits72dpi.jpg");//
Méthode 1
```

```
Bitmap bitmap = new Bitmap(Bitmap.FromFile(@"G:\image24bits72dpi.jpg"));//
Méthode 2

Bitmap bitmap = new Bitmap(@"G:\image24bits72dpi.jpg");//Méthode 3
```

Prenez pour tester une image 24 bits en 72 dpi. Que vous utilisiez Photoshop, PaintShop Pro, Photo Impact ou The Gimp, cela se fait facilement. Faîtes afficher le format de votre image après application des différentes méthodes. Les méthodes 1 et 3 conservent le format original de l'image, c'est à dire « Format24bppRgb ». La méthode 2 convertit le format 24 bits en « Format32bppArgb ».

Nous allons déterminer les temps d'exécution de notre code. L'idéal est d'utiliser un « profiler ». Il s'agit d'un outil capable d'analyser le temps moyen d'exécution de chaque ligne de votre code. Pour les utilisateurs de Visual Studio 2003, je vous renvoie vers des outils tels que DevPartner Community. Pour les heureux possesseurs des versions « haut de gamme » de VS 2005, vous avez des outils intégrés. Pour les autres, le Framework .NET 2.0 propose une classe intéressante « Stopwatch » dans le namespace « System.Diagnostics » dont le fonctionnement est le suivant :

```
Stopwatch s = new Stopwatch();
s.Start();
Bitmap bitmap = new Bitmap(@"G:\image24bits72dpi.jpg");
s.Stop();
MessageBox.Show(s.Elapsed.TotalMilliseconds.ToString());
```

Attention, ceci ne remplacera pas les services d'un vrai profiler!

Bref, votre profiler doit vous indiquer que les méthodes 1 et 3 sont les plus rapides, et plus particulièrement la méthode 3. Il est vrai que vous vous en doutiez mais il est bon de rappeler qu'il faut toujours « profiler » son code et plus particulièrement en traitement d'images. Et ce qui peut paraître évident, ne l'est pas forcément comme nous pourrons bientôt le constater dans les remarques qui suivront.

L'importance du format de l'image

Commençons par accéder en mode unsafe à notre image et faisons en sorte de profiler le « LockBits » en fonction des méthodes 1, 2 et 3 définies plus haut :

```
int width = bitmap.Width;
int height = bitmap.Height;
unsafe
{
        BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0, width,
height), ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
      bitmap.UnlockBits(bmpData);
}
```

Regardons ce que dit le profiler. Il indique qu'avec les méthodes 1 et 3, nous avons un temps plus court au niveau du « LockBits » qu'avec la méthode 2. Comment expliquer cette différence ? La raison est simple, c'est que pour les formats de l'image obtenue avec les méthodes 1 et 3 correspondent parfaitement au format défini dans notre « LockBits », soit en 24 bits. La méthode 2 transforme l'image avec un format 32 bits.
En clair, si nous voulons perdre le moins de temps possible, il faut que le format de l'image source corresponde au format que nous aurons défini dans le « LockBits ». Or, ceci n'est pas

d'une évidence : Connaissez-vous par avance le format de l'image que vous allez charger ? La réponse est non.
Il existe une solution pour répondre à ce problème : Premièrement, détecter le format de l'image source et y répondre par un traitement adapté à ce format. C'est à dire que vous allez devoir développer le traitement autant de fois que vous avez de formats. On peut évidemment dans les cas simples traiter de manière générique (avec une même méthode), mais vous y perdrez en performance et cela deviendra un vrai casse-tête dès que vos algorithmes seront plus compliqués.
La solution la moins fastidieuse est de convertir une bonne fois pour toute l'image source dans le format bien adapté à ce que vous aurez développé pour vos traitements.

En résumé, si avec votre image source, vous ne faîtes qu'un seul traitement, développez de manière spécifique pour tous les formats d'images. Si vous êtes amenez à avoir plusieurs traitements sur votre image source, convertissez votre image (perdez donc un peu de temps processeur) dans le format avec lequel vous aurez codé votre bibliothèque de traitements.

Comportement de l'API GDI+ : influence de la taille, de l'encodage et de la résolution d'une image ?

Dans nos conclusions, nous n'avons pas tenu compte de la taille de l'image, de la résolution dpi de l'image, ou plus exactement, nous avons considéré qu'elle correspondait à celle de l'écran, c'est à dire 72 dpi. Mais qu'en est-il pour des résolutions plus élevées tel que par exemple 300 dpi ? Il est important de tester car le plus souvent, vous souhaiterez traiter des images venant par exemple de votre appareil photo numérique.

Nous allons prendre une image source venant d'un appareil numérique. Dans notre exemple, l'image sera au format 24 bits mais en 230 dpi au lieu de 72 dpi comme vu précédemment. Nous allons effectuer un traitement (conversion en niveau de gris) et nous allons surtout profiler notre code.

```
Bitmap bitmap = new Bitmap(@"G:\image24bits230dpi.jpg");
```

Notre traitement sera le suivant :

```
int width = bitmap.Width;
int height = bitmap.Height;

unsafe
{
    BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0, width,
height), ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);

      int offset = width % 4;

    byte* pix = (byte*)(void*)bmpData.Scan0;
      for (int y = 0; y < height; y++)
      {
            for (int x = 0; x < width; x++)
            {
                  pix[0] = pix[1] = pix[2] = (byte)(.3 * pix[0] + .59 *
pix[1] + .11 * pix[2]);

                  pix += 3;
            }
            pix += offset;
```

```
        }
        bitmap.UnlockBits(bmpData);
}
```

Exécutez ce traitement et retenez le temps moyen de traitement (325 millisecondes par exemple pour mon image).

Transformons notre image source comme ceci :

```
Bitmap bit = new Bitmap(@"G:\image24bits230dpi.jpg");

Bitmap bitmap = new Bitmap(bit);
```

L'image passe du format 24 bits au format 32 bits. Adaptons notre traitement au nouveau format de l'image :

```
int width = bitmap.Width;
int height = bitmap.Height;

unsafe
{
    BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0, width,
height), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);

        byte* pix = (byte*)(void*)bmpData.Scan0;
        for (int y = 0; y < height; y++)
        {
                for (int x = 0; x < width; x++)
                {
                        pix[0] = pix[1] = pix[2] = (byte)(.3 * pix[0] + .59 *
pix[1] + .11 * pix[2]);

                        pix += 4;
                }
        }
        bitmap.UnlockBits(bmpData);
}
```

Si nous exécutons ce traitement et retenons le temps moyen de traitement, nous constatons qu'il est plus faible (225 millisecondes) que pour le traitement en 24 bits.
Or, on pouvait s'attendre à ce que la structure 32 bits soit plus complexe donc plus longue à traiter.

Pourquoi ce type de comportement ? il ne paraît pas logique.

Rechargeons les images, sans effectuer le traitement de conversion en niveau de gris, sauvegardez par la méthode « Save » votre image. Comparez alors dans les deux cas les images sauvegardées. Aidez-vous d'un visualiseur d'images, observez en particulier les propriétés.

Dans le premier cas, nous obtenons :

Image	
Width	2080 pixels
Height	1544 pixels
Horizontal Resolution	230 dpi
Vertical Resolution	230 dpi
Bit Depth	24
Frame Count	1

Ce qui est conforme à l'image initiale.
Et dans le deuxième cas :

Image	
Width	2080 pixels
Height	1544 pixels
Horizontal Resolution	96 dpi
Vertical Resolution	96 dpi
Bit Depth	32
Frame Count	1

Les images ne sont pas aux mêmes résolutions !

Dans le deuxième cas, pour conserver la résolution initiale, il vous suffit d'écrire le code :

```
bitmap.SetResolution(bit.HorizontalResolution, bit.VerticalResolution);
```

Et de nouveau l'image sauvegardée aura une résolution identique à l'image initiale.

Si vous faites de nouveau une série de tests, vous allez vous apercevoir que dans le premier cas, le chargement de l'image est plus court que dans le deuxième cas. A cela, rien de surprenant, puique nous faisons une opération en plus :

```
Bitmap bitmap = new Bitmap(bit);
```

Par contre, ce qui est surprenant, c'est que le temps de traitement (et non de chargement de l'image) est plus long dans le premier cas que dans le deuxième cas. Pour le premier cas, l'image reste au format 24 bits et on applique un traitement spécifique au format 24 bits, dans le deuxième cas, l'image passe au format 32 bits et subit un traitement spécifique au 32 bits. Dans le premier cas, nous sommes à 285 millisecondes en moyenne, alors que dans le deuxième cas, nous sommes à 205 millisecondes.

Vous pouvez vous dire, que ceci est lié au format d'image initiale. J'ai donc recommencé mes tests avec une image qui était initialement au format 32 bits. Les résultats sont quasi – similaires.

Quelles conclusions en tirer ? Ils semblent que pour certaines images, le temps de traitement soit augmenté si on travaille directement dessus. Il faut émettre des hypothèses pour essayer de déterminer quel facteur intervient dans ce phénomène :

Dans un premier temps, j'ai pensé que la résolution intervenait. J'ai refait une série de test avec la même image à des résolutions différentes. Même phénomène.

Le type d'encodage peut-il alors intervenir ? Une série de test sur des images de type JPEG avec des encodages différents donne les mêmes résultats. Un fichier BMP non compressé ne change rien au comportement.

Par contre, si vous prenez une image de taille beaucoup plus petite le phénomène ne se produit plus. Il semble que le comportement décrit plus haut soit lié à la taille de l'image.

Passé une taille critique, un traitement sur une image directement à partir de la source prend beaucoup plus de temps que le traitement sur une nouvelle instance de l'image.

Il est difficile d'expliquer le phénomène. L'explication se trouve sans doute dans l'API « GDIplus.dll » et sa manière de gérer les images en mémoire.

Conseils

Donc en général, si vous comptez effectuer plusieurs traitements sur l'image, il convient de créer une nouvelle Bitmap comme dans le deuxième cas ci-dessus. Le temps de traitement gagné doit compenser le coût de création nouvelle d'une image.
Attention, toutefois à cette conclusion hative. Pour le traitement d'images aux tailles gigantesques, si vous n'avez pas besoin de les afficher, il conviendra d'attaquer vous-même le fichier physique en décodant les données par vous-même. C'est ce que fait évidemment GDI+ mais il y a création d'un objet graphique en mémoire, ce qui devient pénalisant quand celui-ci est de taille trop importante.

Le clonage d'image

On ne doit pas passé sous silence, que le clonage d'une image donne par la suite des résultats catastrophiques en terme de performance sur les traitements appliqués à celle-ci.

S'il vous venez à l'idée d'utiliser la méthode « Clone » de la classe Bitmap,

```
Bitmap bitmap = (Bitmap) bit.Clone();
```

Le temps de traitement deviendrait très important ! (680 millisecondes dans notre exemple)
Par contre, le clonage permet de garder par exemple toutes les attributs liés à l'image tel que le nom de l'appareil ayant numérisé, le logiciel ayant créé l'image, etc ...

Le fichier physique peut être « locké »

Si vous voulez appliquer un traitement directement sur une image directement chargée, tant que l'objet existera en mémoire, elle ne pourra pas être modifiée ou même supprimée physiquement par un autre logiciel. Si vous voulez rendre à nouveau disponible le fichier physique de l'image, il vous faut effectuer un « Dispose() » sur la Bitmap chargée.

3. Quelques corrections sur l'image

La correction gamma

Vous avez déjà pu constater que les images que vous avez prises à partir de votre appareil photo numérique, n'ont pas le même rendu couleur suivant le support où celles-ci sont affichées. Le rendu d'un écran à un autre diffère. Et puis, s'il vous venait à l'idée d'imprimer votre photo sur votre imprimante ou bien de réaliser un tirage papier chez votre boutique photo préférée, vous seriez sans doute déçu du rendu des couleurs. Cela vient du fait que le calibrage varie fortement d'un support à un autre.

Un appareil photo a tendance a capter une image avec une intensité moindre que l'intensité réelle. Les couleurs apparaissent alors moins saturées que dans la réalité. En ce qui concerne, l'écran de votre ordinateur, les couleurs apparaissent plus saturées que pour l'image initiale. Assez souvent, les deux phénomènes s'auto-compensent. Mais il y a des cas, où il est nécessaire d'appliquer une correction. Il s'agit de la correction gamma :

$$I_new = I_max * (I_entree/I_max)^{1/Gamma}$$

Gamma : correction gamma
I_entree : Intensité actuelle
I_max : Intensité maximum
I_new : Intensité nouvelle

Le résultat attendu, à partir de la formule, est que si Gamma est inférieur à 1, on aura un assombrissement des couleurs claires. Au contraire, si Gamma est supérieure à 1, les couleurs sombres seront éclaircies.

Voici un exemple :

Image Originale

Image corrigée avec gamma = 1.3

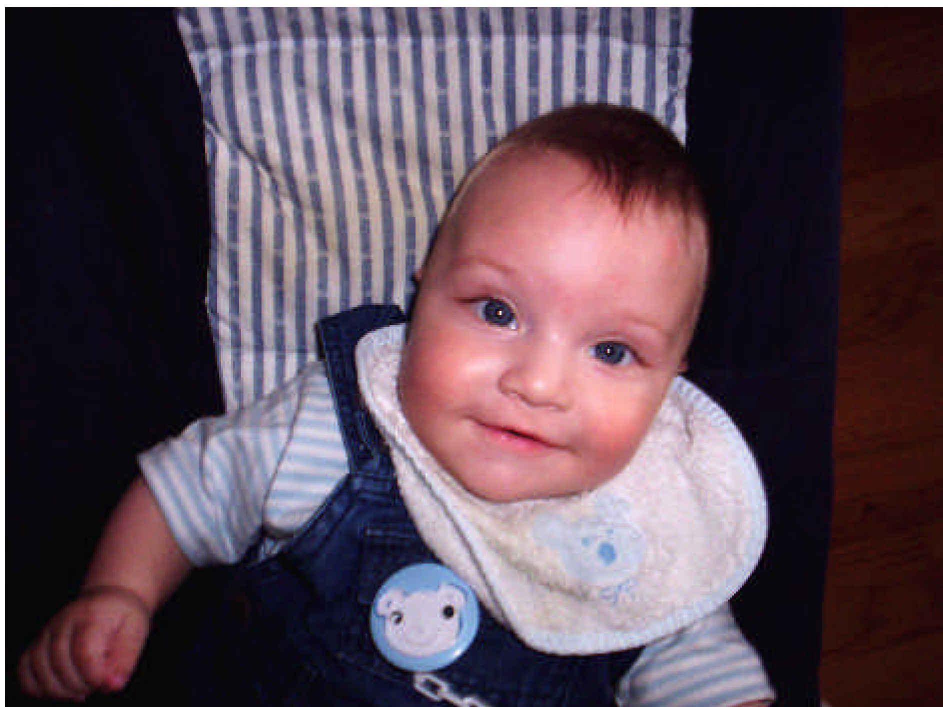


Image corrigée avec gamma = 0.6

Voici l'application de la formule :

```
public static bool Gamma(Bitmap bitmap,float red, float green, float blue)
{
      //En dehors de ces valeurs, une correction Gamme n'a pas d'intérêt
      if (red < .1 || red > 5) return false;
      if (green < .1 || green > 5) return false;
      if (blue < .1 || blue > 5) return false;
```

```
        //définition des tables de précalculs
        byte[] redGamma = new byte[256];
        byte[] greenGamma = new byte[256];
        byte[] blueGamma = new byte[256];

        //remplissage des tables de précalculs
        for (int i = 0; i < 256; ++i)
        {
                redGamma[i] = (byte) Math.Min(255, (int)(255f * Math.Pow(i /
255f, 1f/red)));
                greenGamma[i] = (byte) Math.Min(255, (int)(255f * Math.Pow(i /
255f, 1f/green)));
                blueGamma[i] = (byte) Math.Min(255, (int)(255f * Math.Pow(i /
255f, 1f/blue)));
        }

        int width = bitmap.Width;
        int height = bitmap.Height;

        unsafe
        {
                BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0, width,
height), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);

                byte* pix = (byte*)(void*)bmpData.Scan0;
                for (int y = 0; y < height; y++)
                {
                        for (int x = 0; x < width; x++)
                        {
                                pix[0] = blueGamma[pix[0]];
                                pix[1] = greenGamma[pix[1]];
                                pix[2] = redGamma[pix[2]];
                                pix += 4;
                        }
                }
                bitmap.UnlockBits(bmpData);
        }
        return true;
}
```

Attention à ne pas appliquer plusieurs fois la correction Gamma successivement sur une image. Si l'on souhaite, faire prévisualiser la correction, il vous faudra partir à chaque changement de l'image initiale.

Il est possible de faire de la correction Gamma avec le framework .NET de la manière suivante :

```
public static void SetGamma(Bitmap bitmap, float gamma)
{
        ImageAttributes imgatt = new ImageAttributes();
        imgatt.SetGamma(gamma);
        Graphics g = Graphics.FromImage(bitmap);
        g.DrawImage(bitmap, new Rectangle(0, 0, bitmap.Width, bitmap.Height),
0, 0, bitmap.Width, bitmap.Height, GraphicsUnit.Pixel, imgatt);
        g.Dispose();
}
```

Il faut savoir que cette méthode toute intégrée est plus lente que la précédente.

La luminosité

Modifier la luminosité globale d'une image est simple. Il suffit d'ajouter ou de soustraire une intensité lumineuse.

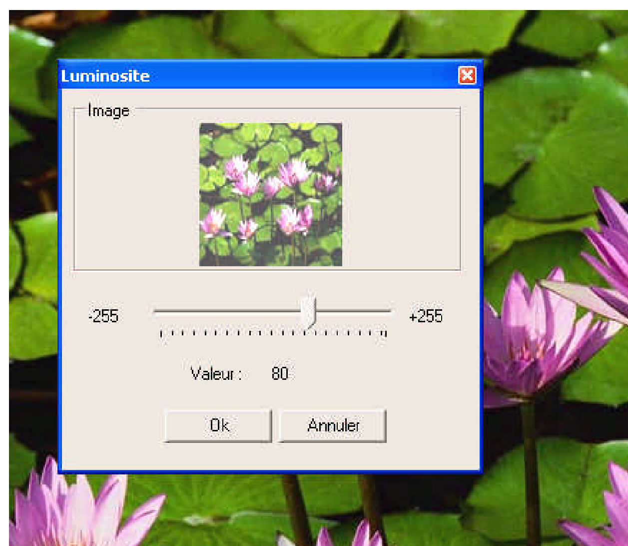
I_New = I_entree + I_corrective

I_New = Intensité de l'image résultat
I_Entree = Intensité actuelle
I_corrective = paramètre correctif dont les bornes sont entre – I_max et + I_max
I_max = Intensité maximum

Voici le code :

```
public static bool Luminosite(Bitmap bitmap, int lumiere)
{
	//En dehors de ces valeurs, on n'effectue pas
	if (lumiere < -255 || lumiere > 255)
		return false;

	//définition de la table de précalcul
	byte[] light = new byte[256];

	//remplissage des tables de précalculs
	for (int i = 0; i < 256; ++i)
	{
		int temp = i + lumiere;
		light[i] = (byte) ((temp < 0) ? 0 : (temp > 255) ? 255 : temp);
	}

	int width = bitmap.Width;
	int height = bitmap.Height;

	unsafe
	{
		BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0, width,
height), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);

		byte* pix = (byte*)(void*)bmpData.Scan0;
		for (int y = 0; y < height; y++)
		{
			for (int x = 0; x < width; x++)
			{
				pix[0] = light[pix[0]];
				pix[1] = light[pix[1]];
				pix[2] = light[pix[2]];
				pix += 4;
			}
		}
		bitmap.UnlockBits(bmpData);
	}
	return true;
}
```

Créer une interface utilisateur est facile :



L'exposition

Quand nous prenons une photo, celle-ci n'est pas toujours prise dans des conditions idéales de lumière. On peut corriger l'exposition de la photo non pas en ajoutant ou soustrayant par une valeur mais en appliquant une fonction non linéaire comme la fonction exponentielle.
La problématique est qu'il ne faut pas dépasser la valeur de l'intensité maximum. Il nous faut donc une fonction qui permette de se rapprocher de zéro et de tendre vers 1 suivant deux bornes définies.

On constate que la fonction $\mathbf{e}^{-x}$ tend vers 0 quand x se rapproche de l'infini (c'est à dire de la valeur maximum) et est égal à 1 quand x est égal à 0.

La formule suivante permet d'établir ce que nous voulons (c'est à dire mettre en valeur les pixels de fortes intensités) :

$$(1 - \mathbf{e}^{-x}) * I_max$$

Avec x = I_entree / I_max * Coef / 100

I_entree : Intensité actuelle
I_max : Intensité maximum
Coef : Coefficient positif que l'on applique

Si on applique cette formule au code, cela nous donne :

```
public static bool CorrectionExposition(Bitmap bitmap, int correction)
{
      //En dehors de ces valeurs, on n'effectue pas
      if (correction < 0 || correction > 2000)
            return false;

      int width = bitmap.Width;
      int height = bitmap.Height;

      //Table de pré-calcul
      byte[] tab = new byte[256];
      for (int i = 0; i <= 255; i++)
      {
```

```
            double d = Math.Exp(-(i / 255.0 * correction / 100.0));
            tab[i] = (byte)((1.0 - d) * 255);
      }

      unsafe
      {
            BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0, width, 
height), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);

            byte* pix = (byte*)(void*)bmpData.Scan0;

            for (int y = 0; y < height; y++)
            {
                  for (int x = 0; x < width; x++)
                  {
                        pix[0] = tab[pix[0]];
                        pix[1] = tab[pix[1]];
                        pix[2] = tab[pix[2]];
                        pix+=4;
                  }
            }
            bitmap.UnlockBits(bmpData);
      }
      return true;
}
```

Exécutons ce code sur une image dont on souhaite changer l'exposition :

Image originale prise le matin (voir l'ombre des arbres)

Image corrigée avec un coefficient de 180

Image corrigée avec un coefficient de 250

Essayez de faire la différence entre une image dont on a changé la luminosité et une image dont on change l'exposition. Dans le premier cas, tous les pixels sont affectés de manière égale par une valeur. Dans le deuxième cas, les pixels de forte intensité, sont beaucoup plus affectés par le changement que les pixels de faible intensité.

4. Conclusion

Il faut avoir à l'esprit qu'en traitement d'images, il faut toujours tester la performance de telle ou telle instruction. Ce qui peut paraître évident, ne l'est pas forcément. Munissez-vous d'un

« profiler » pour voir où vous perdez beaucoup de temps dans votre traitement. Et améliorez votre code. Dans un prochain article, je vous illustrerai par un exemple vécu, l'amélioration d'un traitement.
A travers cet article, vous avez pu aussi vous rendre compte que la façon de charger une image était très importante pour la suite de vos traitements. Vous n'utiliserez plus l'API GDI+, sans vous poser de questions. Il faut adapter les méthodes de chargement d'une image à vos besoins.
Enfin, nous avons abordé quelques améliorations que l'on peut apporter à une image. Suivront d'autres à travers les articles suivants.

5. En savoir plus

La FAQ de Bob Powell.net
http://www.bobpowell.net/

Unsafe Image Processing
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp11152001.asp

Qu'est-ce que la correction Gamma ?
http://www-inf.enst.fr/~vercken/couleurs/gamma.pdf

N'hésitez pas à me contacter :

Frédéric Mélantois
Email : fmelantois[arobase]free.fr

Blog : http://blog.developpeur.org/tkfe/