

# Traitement d'images sous .NET, partie 1

Cet article a été réalisé à partir de Visual Studio 2005, donc du framework 2.0 mais reste parfaitement valable pour le framework 1.1

## 1. Introduction

Nous allons à travers une série d'articles acquérir quelques bases pour traiter les images. Dans cette première partie, nous nous attacherons à étudier l'accès aux pixels sous différents formats d'image.

## 2. L'accès aux pixels

### Introduction

Le framework .NET fournit nombre de classes dédiées au traitement des images. Ces classes contiennent du code faisant très largement appel à une API non managée GDI+ qui contient nombre de fonctions intéressantes. Aussi, si lors de vos développements, vous atteignez les limites des possibilités du framework ou que vous êtes à la recherche d'une meilleure performance, n'hésitez pas à faire de l'interop sur gdiplus.dll et gdi32.dll (l'ancienne API GDI).

La classe Bitmap possède une méthode « SetPixel » permettant d'accéder à un pixel de l'image. Un appel répété à cette méthode montre très vite ses limites. Par exemple, si nous souhaitons repeindre une image de 320 pixels par 200 en bleu, il nous faudra faire appel 64000 fois à la méthode « SetPixel » en passant les coordonnées.

On comprend ici l'intelligence des concepteurs de .NET de laisser la possibilité de produire du code non-managé (non-managé pour non géré par la CLR). On se doute que les couches de haut niveau de .NET font obstacle à la performance. Si on peut accepter que l'affichage d'un composant soit plus lent pour l'utilisateur de quelques millisecondes à des fins de simplification de développement et de réutilisation, on ne peut accepter la moindre milliseconde de perdu sur le traitement d'un pixel d'une image.

Nous trouvons deux méthodes LockBits et UnlockBits qui vont nous offrir une sorte de fenêtre par laquelle nous allons accéder aux données de l'image. Plus qu'une simple fenêtre, les données se trouvent fixées dans leur localisation et celles-ci sont « protégées » de l'accès simultanée en écriture. On va donc pouvoir les pointer directement en mémoire. Mais cette opération « LockBits » a un coût comme nous le verrons un peu plus loin.

Regardons dans un premier temps comment accéder à cette « fenêtre » :

La méthode LockBits va permettre de fournir les données brutes de la Bitmap :

```
unsafe
{
    BitmapData bmpData = bitmap.LockBits(new Rectangle(0, 0,
    bitmap.Width, bitmap.Height), ImageLockMode.ReadWrite,
    PixelFormat.Format32bppArgb);
```

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

L'accès aux données se fait dans un contexte non-managé (mode unsafe) car nous souhaitons pointer en mémoire une partie de l'image. On définit la zone que l'on souhaite manipuler (en général, la totalité de l'image) par la définition d'un rectangle. On précise le mode d'accès, ici lecture/écriture puis le format des données. Ce format peut varier en fonction de l'image (1 bits, 8 bits, 16 bits, 24 bits, 32bits). Il est bien évident qu'on cherchera à utiliser le mode le mieux adapté afin de réduire la quantité de données stockées en terme de poids (Grande influence sur la mémoire).

La méthode « LockBits » nous renvoie un objet avec un certain nombre de propriétés :

- ✚ Width est la largeur en pixels de l'image manipulée
- ✚ Height est la hauteur de l'image manipulée
- ✚ PixelFormat est le format dans lequel se trouvent les données de l'image
- ✚ Scan0 est l'adresse mémoire des données fixées.
- ✚ Stride est le nombre d'octets d'une largeur de l'image manipulée

Le contexte « unsafe » va nous permettre d'utiliser les pointeurs qui existent dans la plupart des langages tels que C, Pascal...

```
byte* pixel = (byte*)(void*)bmpData.Scan0;  
//syntaxe équivalente  
//byte* pixel = (byte*)bmpData.Scan0.ToPointer();  
pixel[0] = 255;  
pixel += 4;  
pixel[0] = 0;
```

Dans le code ci-dessus, nous pointons sur le premier octet de l'image et nous portons la valeur à 255. Nous nous déplaçons ensuite de 4 octets. Au 5<sup>ème</sup> octet de l'image, nous portons la valeur à 0.

Vous venez de voir les bases élémentaires pour manipuler une image. Bien évidemment, pour modifier au mieux son image, il faut en connaître les caractéristiques :

- ✚ Sous quel format est mon image ?
- ✚ Quand s'arrête la zone de données ? (il ne faut pas aller modifier la mémoire en dehors de notre zone).

Le format a une incidence sur la façon d'accéder à un pixel. Par exemple, 1 octet peut contenir 8 pixels d'une image en noir et blanc. 4 octets seront nécessaire pour stocker 1 pixel d'une image en 32bits. 1 pixel d'une image 24 bits contient 3 octets. 1 pixel d'une image 16 bits contient 2 octets.

Les données sont stockées suivant des rangées virtuelles. Si les données n'occupent pas totalement la rangée car le nombre d'octet est inférieur, on a des octets en surplus. J'appellerai ce nombre Offset.

Si pour une image 32 bits, de part la structure de 4 octets, on conçoit parfaitement que l'Offset est nul, il n'en est pas de même pour les autres formats d'images.

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

### 🚦 Gestion des images en 32 bits

Les images en 32 bits sont assez facilement gérables. Un pixel est codé en 32 bits sous quatre composantes : le bleu, le vert, le rouge, la couche alpha. Cette dernière permet de gérer la transparence. Chaque composante est codée donc sous 8 bits, c'est à dire un octet, en c# un « byte ». Donc si l'on souhaite récupérer un pixel sans le décomposer, on utilisera un « uint ». Par contre, si l'on souhaite gérer chaque composante, soit on décomposera l'« uint » soit on récupérera des « bytes ».

Prenons l'exemple d'une image dont on va intervertir la composante bleue avec la composante rouge.

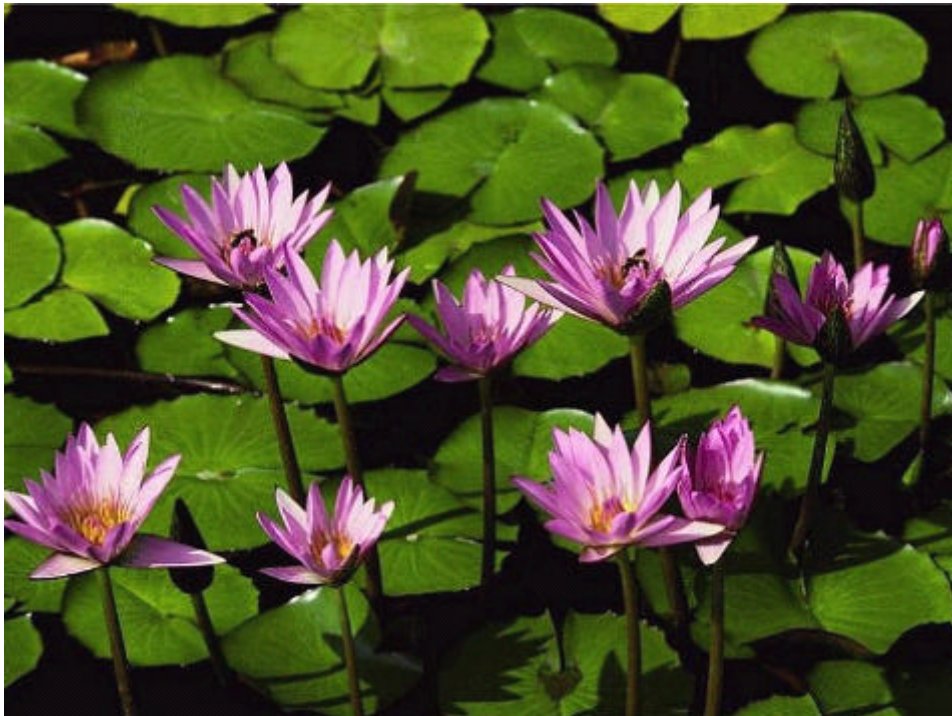


Image initiale

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1



Image traitée

Voici le code, si nous accédons aux données de l'image par « byte » :

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

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

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

Dans l'ordre, on accède au bleu (NewPixel[0]), vert (NewPixel[1], rouge (NewPixel[2]) et la couche alpha (NewPixel[3]).

Maintenant, essayons de faire la même chose en capturant non plus des « byte » mais des « uint » :

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

unsafe
```

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

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

    uint* newPixel = (uint*)(void*)bmpData.Scan0;

    for (int y = 0; y < height; y++)
        for (int x = 0; x < width; x++)
            {
                newPixel[0] = (uint)(((newPixel[0] & 0x000000ff) << 16) |
(newPixel[0] & 0x0000ff00) | ((newPixel[0] & 0x00ff0000) >> 16) |
(newPixel[0] & 0xff000000));
                newPixel++;
            }

    bitmap.UnlockBits(bmpData);
}
```

Nous avons utilisés ici des opérations élémentaires qui permettront d'avoir un code assembleur très optimisé lors de l'exécution. Nous avons utilisé le Et Logique afin d'appliquer un masque qui nous permet de recueillir seulement l'information qui nous intéresse, une des composantes de couleur. Ensuite, il suffit de réaliser des décalages de bits pour intervertir la composante bleue et la composante rouge. « 0x000000ff » est le masque utilisé pour capturer la composante bleue. « 0x0000ff00 » est utilisé pour capturer le vert. « 0x00ff0000 » est utilisé pour capturer le rouge. « 0xff000000 » est utilisé pour capturer la couche alpha.

Une des questions que vous vous posez : Quelle est la méthode la plus rapide ? Pointer un « byte » ou pointer un « uint » ?

Dans notre exemple précédent, la capture par « uint » est la méthode la plus rapide. Mais il faut savoir que cela ne sera pas toujours le cas, cela dépendra de vos algorithmes. Une des règles principales du traitement d'images est l'optimisation. Il faut profiler son code pour savoir où on peut gagner du temps.

### Gestion des images en 24 bits

Les images en 24 bits sont les plus courantes. Chaque pixel est constitué de trois composantes : le bleu, le vert, le rouge. Il n'y a pas de couche alpha.

L'image est stockée sous forme de rangée d'« uint ». On comprend vite que puisque un « uint » peut être découpé en quatre « byte », il se peut que les composantes des pixels ne peuplent pas entièrement une rangée de l'image. Le nombre de « byte » non utilisé dans la rangée définit l'offset, c'est à dire le décalage qu'il sera nécessaire de faire pour passer à la rangée suivante.

Si « width » est la largeur de l'image en pixel, ce décalage se calcule facilement par la formule suivante :

width modulo 4

Une autre méthode est d'utiliser la propriété Stride de la BitmapData :

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

Stride – (width\*3)

La première méthode demande évidemment moins d'instruction, on gagne un peu de temps.

Prenons notre exemple d'inversion de couleurs et appliquons-la à une image 24bits :

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

    //int offset = bmpData.Stride - (width * 3);
    int offset = width % 4;

    byte* newPixel = (byte*)(void*)bmpData.Scan0;
    byte swap;
    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            swap = newPixel[2];
            newPixel[2] = newPixel[0];
            newPixel[0] = swap;
            newPixel+=3;
        }
        newPixel += offset;
    }
    bitmap.UnlockBits(bmpData);
}
```

On passe d'un pixel à l'autre dans une même rangée par un saut de 24 bits, c'est à dire 3 « byte ». Quand on atteint la fin de rangée, on passe à la rangée suivante, en faisant un saut équivalent au décalage (« offset »).

### Gestion des images en 8 bits (couleurs indexées)

Il s'agit des images en 256 couleurs ou à 255 niveaux de gris. Elles sont beaucoup moins utilisés aujourd'hui mais leurs heures de gloire furent la fin des années 80, où les cartes vidéos commençaient à gérer les 256 couleurs (en dehors du célèbre mode 320x200 en 256 couleurs des cartes VGA utilisé dans nombre de jeux vidéos).

Quant aux images 256 niveaux de gris, elles étaient très largement utilisés : Par exemple, les images Spot dont chaque canal (Rouge, vert, bleu, infrarouge par exemple) était rendu par une image brut non compressé (format .ima). Chaque niveau de gris était représenté par un octet. A l'époque, comme on disposait de peu de couleurs, on ne pouvait représenter la richesse visuelle de 3 canaux superposés. Il fallait souvent se choisir des « plages » de détails en spécifiant par exemple, que l'on voulait 216 couleurs (6x6x6). On réduisait la richesse d'un canal de 0 à 255 en 6 valeurs possibles : 0,51,102,153,204,255. Plusieurs techniques étaient possibles :

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

- on basculait les valeurs de 0 à 50 vers la valeur 0, les valeurs 51 à 101 vers la valeur 51 et ainsi de suite... On appelle ceci une **segmentation automatique**.
- on définissait une valeur de début d'application, par exemple la valeur 100. Toutes les valeurs inférieures à 100 se voyaient attribuées la valeur 0. On définissait une valeur de fin d'application, par exemple la valeur 220. Toutes les valeurs supérieures à 220 se voyaient attribuées la valeur 255. Entre 100 et 220, on répartissait les pixels de manière égalitaire sur les valeurs restantes (51,102,153,204). Pour aider à choisir les valeurs de début et de fin d'application, on disposait de l'histogramme (Nombre de pixels par valeur de gris). On appelle ceci une **segmentation bornée**.
- On choisissait chaque plage de l'histogramme. Par exemple, de 0 à 100, on attribuait la valeur 0, de 101 à 110, la valeur 51, de 111 à 117, la valeur 102, etc... On appelle une **classification**.

La classification permet de mettre en évidence des détails. C'est ainsi que sur des images satellites, on peut isoler la végétation, voire même les types de végétation. On peut aussi repérer les zones de failles de l'écorce terrestre (on a pu en détecter ainsi alors que sur le terrain, les géologues n'avaient pas repérés celles-ci).

Aujourd'hui, on retrouve ces problèmes de réduction du nombre de couleurs par exemple quand on veut créer une image GIF pour le web. La problématique est la suivante : comment à partir de mon image 32 bits, peut-on créer une image 8 bits ? A partir d'une image 32bits, on fabrique une nouvelle image 8bits avec une palette à définir :

```
public static Bitmap Convert8Bits(Bitmap bitmap)
{
    int width = bitmap.Width;
    int height = bitmap.Height;

    Bitmap bitmap8bits = new Bitmap(width, height,
PixelFormat.Format8bppIndexed);
    ColorPalette pal = bitmap8bits.Palette;
    byte[] tab = new byte[6] { 0, 51, 102, 153, 204, 255 };
}
```

On définit dans « tab » les 6 valeurs de composantes qui seront utilisés pour la palette 216 couleurs. On crée la palette de la nouvelle image comme ceci :

```
for (int b=0; b < 6; b++)
    for (int g=0; g < 6; g++)
        for (int r = 0; r < 6; r++)
        {
            pal.Entries[(36*b)+(6*g)+r] =
Color.FromArgb(0,tab[r],tab[g],tab[b]);
        }
```

Nous allons devoir répartir les valeurs des composantes de l'image 32 bits sur une plage de 6 valeurs en répartissant plus équitablement que par une simple division par 51. Nous stockons le résultat pour les entrées de 0 à 255 dans un tableau. Il s'agit d'une table de pré-calcul qui évite de répéter les opérations par le nombre de pixels total de l'image (et même par ces composantes, dans le cas présent) ! On transforme un calcul par une affectation pour chaque composante de pixels :

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

```
Byte[] canalTab = new byte[256];

    int m;
    for (int i = 0; i < 256; i++)
    {
        m = i / 51;
        m += (i%51>0 && Math.Abs(i-tab[m]) > Math.Abs(i-tab[m + 1])) ?
1 : 0;
        canalTab[i] = Convert.ToByte(m);
    }
```

Puis :

```
unsafe
{
    BitmapData bmpDataOld = bitmap.LockBits(new Rectangle(0, 0,
width, height), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);
    BitmapData bmpDataNew = bitmap8bits.LockBits(new Rectangle(0,
0, width, height), ImageLockMode.ReadWrite, PixelFormat.Format8bppIndexed);

    byte* oldPixel = (byte*)(void*)bmpDataOld.Scan0;
    byte* newPixel = (byte*)(void*)bmpDataNew.Scan0;

    int offsetNew = bmpDataNew.Stride - (width);

    for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
            newPixel[0] = (byte)((36 * canalTab[oldPixel[2]]) +
(6 * canalTab[oldPixel[1]]) + canalTab[oldPixel[0]]);
            oldPixel += 4;
            newPixel++;
        }
        newPixel += offsetNew;
    }
    bitmap8bits.UnlockBits(bmpDataNew);
    bitmap.UnlockBits(bmpDataOld);
}
return bitmap8bits;
}
```

Cette technique ne donne pas les meilleurs résultats visuels. Il existe d'autres techniques comme la quantification où on répartit les composantes du pixel de l'image 32 bits en 3 bits par exemple (8 niveaux) pour le rouge et le vert et 2 bits (4 niveaux) pour le bleu. On a alors  $8 \times 8 \times 4 = 256$  couleurs pour la palette. Nous détaillerons cette technique dans un prochain article.

Après avoir vu comment accéder à une image 8 bits et comment convertir une image 32 bits en une image 8 bits, regardons de plus près la modification de la palette d'une image 8 bits :

Comment inverser les canaux Rouge et bleu dans une image 8 bits ? Il suffit de manipuler la palette de l'image.

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

Il n'est malheureusement pas possible de manipuler directement la palette d'une Bitmap existante à travers le framework .NET.

```
int length = bitmap.Palette.Entries.Length;
uint c;
for (int i = 0; i < length; i++)
{
    c = (uint)bitmap.Palette.Entries[i].ToArgb();
    bitmap.Palette.Entries[i] = Color.FromArgb((int)(((c & 0x000000ff) <<
16) | (c & 0x0000ff00) | ((c & 0x00ff0000) >> 16) | (c & 0xff000000)));
}
```

Ce code ne produira aucun effet sur l'image !

Il est nécessaire de créer une nouvelle image 8bits pour obtenir une nouvelle référence de palette !

```
private static ColorPalette GetPalette(Color[] tabcoul)
{
    Bitmap bitmap = new Bitmap(1,1,PixelFormat.Format8bppIndexed);
    ColorPalette pal = bitmap.Palette;

    uint c;
    for (int i = 0; i < 256; i++)
    {
        c = (uint)tabcoul[i].ToArgb();
        pal.Entries[i] = Color.FromArgb((int)(((c & 0x000000ff) << 16)
| (c & 0x0000ff00) | ((c & 0x00ff0000) >> 16) | (c & 0xff000000)));
    }
    bitmap.Dispose();
    return pal;
}
...
bitmap.Palette = GetPalette(bitmap.Palette.Entries);
```

Regardons si nous ne pourrions pas via Interop modifier la palette de la Bitmap. La dll GDIplus met à notre disposition cette méthode :

```
[DllImport("gdiplus.dll", CharSet = CharSet.Unicode, SetLastError = true,
ExactSpelling = true)]
internal static extern int GdipSetImagePalette(HandleRef image, IntPtr
palette);
```

L'utilisation de celle-ci avec la Bitmap est impossible car la classe Bitmap ne met pas à disposition l'adresse de l'image en mémoire.

```
int length = bitmap.Palette.Entries.Length;
IntPtr ptr = Marshal.AllocHGlobal(8 + (4 * 256)); // 1 int = 4 octets
Marshal.WriteInt32(ptr, bitmap.Palette.Flags); //flag du type de palette
Marshal.WriteInt32(ptr, 256); //taille de la palette
uint c;
for (int i = 0; i < 256; i++)
{
    c = (uint)bitmap.Palette.Entries[i].ToArgb();
    Marshal.WriteInt32(ptr, (int)(((c & 0x000000ff) << 16) | (c &
0x0000ff00) | ((c & 0x00ff0000) >> 16) | (c & 0xff000000)));
}
```

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

```
}  
GdiplusImagePalette(new IntPtr(bitmap, pointerdelimage), ptr);  
Marshal.FreeHGlobal(ptr);
```

Nous ne pouvons malheureusement pas obtenir « pointerdelimage » !!

Si l'on souhaite gagner du temps en évitant de créer un bitmap 8 bits de 1x1 pixel, il nous faut redéfinir toute notre propre classe Bitmap. Nous aurons lors de la création de notre image via l'API Gdiplus, en retour, l'adresse de l'image en mémoire. Tout ceci pour une simple palette ! Dommage.

On ne peut conseiller cette approche qu'à ceux qui veulent réaliser des traitements par lots d'un nombre important d'images. Pour les opérations ponctuelles, on se contentera de récupérer la référence de la palette d'une nouvelle image, comme présentée plus haut.

Voici en illustration l'image 8 bits dont on a modifié par la suite la palette :



Image 8 bits originale

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1



Image 256 couleurs traitée

### 🔧 Gestion des images en 4 bits (couleurs indexées)

La gestion des images 4 bits (16 couleurs) n'est pas beaucoup plus compliquée que celles de 8 bits. La spécificité réside dans le fait que dans un byte que l'on lira de l'image, nous aurons 2 pixels.

Nous allons illustrer l'accès à l'image 4bits, en intervertissant comme précédemment le canal rouge avec le canal bleu puis nous convertirons l'image obtenue en une image 32 bits. Voici le code commenté :

```
//création d'une nouvelle palette
private static ColorPalette GetPalette(Color[] tabcoul)
{
    Bitmap bitmap = new Bitmap(1, 1, PixelFormat.Format4bppIndexed);
    ColorPalette pal = bitmap.Palette;

    uint c;
    for (int i = 0; i < 16; i++)
    {
        c = (uint)tabcoul[i].ToArgb();
        pal.Entries[i] = Color.FromArgb((int)((c & 0x000000ff) << 16)
| (c & 0x0000ff00) | ((c & 0x00ff0000) >> 16) | (c & 0xff000000));
    }
    bitmap.Dispose();
    return pal;
}

public static Bitmap Convert32Bits(Bitmap bitmap4Bits)
{
    int width = bitmap4Bits.Width;
    int height = bitmap4Bits.Height;

    //on va intervertir les composantes dans la palette de l'image 4bits
    bitmap4Bits.Palette = GetPalette(bitmap4Bits.Palette.Entries);
}
```

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

```
//création de l'image 32bits qui va accueillir
Bitmap bitmap32Bits = new
Bitmap(width,height,PixelFormat.Format32bppArgb);

//on définit trois tables de précalculs
//qui donne les composantes RGB de la palette
byte[] red = new byte[16];
byte[] green = new byte[16];
byte[] blue = new byte[16];

for (int i = 0; i < 16; i++)
{
    red[i] = bitmap4Bits.Palette.Entries[i].R;
    blue[i] = bitmap4Bits.Palette.Entries[i].B;
    green[i] = bitmap4Bits.Palette.Entries[i].G;
}

unsafe
{
    BitmapData bmpDataOld = bitmap4Bits.LockBits(new Rectangle(0,
0, width, height), ImageLockMode.ReadWrite, PixelFormat.Format4bppIndexed);
    BitmapData bmpDataNew = bitmap32Bits.LockBits(new Rectangle(0,
0, width, height), ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);

    int offset = bmpDataOld.Stride - (width/2);

    byte* oldPixel = (byte*)(void*)bmpDataOld.Scan0;
    byte* newPixel = (byte*)(void*)bmpDataNew.Scan0;

    int quatreBits;

    for (int y = 0; y < height; y++)
    {
        for (int x = 1; x <= width; x++)
        {
            //une fois sur 2, on récupère les 4 bits
            //à gauche ou à droite pour l'octet lu
            quatreBits = (x % 2 == 0) ? oldPixel[0] >> 4 :
oldPixel[0] & 0x0F;

            //on affecte les composantes de l'image 32 bits
            //par les tables de pré-calculs
            newPixel[0] = blue[quatreBits];
            newPixel[1] = green[quatreBits];
            newPixel[2] = red[quatreBits];
            newPixel[3] = 255;//couche alpha
            oldPixel+=x%2;//une fois sur 2, on avance le
pointeur
                newPixel += 4;
            }
            oldPixel += offset;
        }
        bitmap4Bits.UnlockBits(bmpDataOld);
        bitmap32Bits.UnlockBits(bmpDataNew);
    }
    return bitmap32Bits;
}
```

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

### 🔧 Gestion des images en 1 bit (noir et blanc)

Les images en noir et blanc sont codées suivant le principe un pixel, un bit. Un « byte » contient alors 8 pixels. Nous allons illustrer l'accès aux images 1 bit par un exemple de conversion.

Nous allons convertir chaque pixel d'une image 32 bits en niveau de gris. Et nous allons appliquer un algorithme de dithering de Bayer, c'est à dire qu'on applique un motif à l'image afin de n'avoir que des pixels en 0 ou 1 pour la nouvelle image 1 bit. Voici le code commenté :

```
public static Bitmap Convert1Bit(Bitmap bitmap32Bits)
{
    int width = bitmap32Bits.Width;
    int height = bitmap32Bits.Height;

    Bitmap bitmap1Bit = new Bitmap(width, height,
PixelFormat.Format1bppIndexed);

    //le pattern à appliquer sur des niveaux de gris de 64
byte[][] pattern = new byte[][]{
        new byte[]{0,32,8,40,34,2,10,42},
        new byte[]{48,16,56,24,50,18,58,26},
        new byte[]{12,44,4,36,14,46,6,38},
        new byte[]{60,28,52,20,62,30,54,22},
        new byte[]{3,35,11,43,1,33,9,41},
        new byte[]{51,19,59,27,49,17,57,25},
        new byte[]{15,47,7,39,13,45,5,37},
        new byte[]{63,31,55,23,61,29,53,21}};

    unsafe
    {
        BitmapData bmpDataOld = bitmap32Bits.LockBits(new Rectangle(0,
0, width, height), ImageLockMode.ReadWrite, PixelFormat.Format32bppRgb);
        BitmapData bmpDataNew = bitmap1Bit.LockBits(new Rectangle(0, 0,
width, height), ImageLockMode.ReadWrite, PixelFormat.Format1bppIndexed);

        byte* oldPixel = (byte*)(void*)bmpDataOld.Scan0;
        byte* newPixel = (byte*)(void*)bmpDataNew.Scan0;

        //l'offset de l'image 1 bit
        int offset = bmpDataNew.Stride - (width / 8);

        byte pixel;
        int newPix = 0;

        for (int y = 0; y < height; y++)
        {
            for (int x = 1; x <= width; x++)
            {
                //A partir des composantes du pixel
                //de l'image 32 bits, on détermine
                //le niveau de gris
                pixel = (byte)(.3 * oldPixel[0] + .59 * oldPixel[1]
+ .11 * oldPixel[2]);

                //on compare le pixel (réduit à 64 niveaux)
                //au pattern (en fonction de sa position
                //par rapport à l'image)
                if ((pixel >> 2) > pattern[x & 7][y & 7])
                    //on allume le pixel (Bit=1)
                    //dans le byte
                    newPix = newPix | (1 << (7-(x % 8)));
            }
        }
    }
}
```

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1

```
        if (x % 8 == 0)
        {
            //quand on a rempli 8 bits
            //on écrit le byte dans l'image
            newPixel[0] = (byte)newPix;
            //on avance le pointeur
            newPixel++;
            //on éteint tous les pixels
            //pour le nouveau cycle de 8 bits
            newPix = 0;
        }
        oldPixel += 4;
    }
    newPixel += offset;
}

bitmap32Bits.UnlockBits bmpDataOld;
bitmap1Bit.UnlockBits bmpDataNew;
}
return bitmap1Bit;
}
```

Nous obtenons le résultat suivant :



Si nous n'avions pas appliqué de dithering, mais simplement une binarisation en mettant à 1 tous les niveaux de gris supérieurs à 100, nous aurions obtenu cette image :

## .NET passionnément, tout simplement

Traitement d'images sous .NET, partie 1



Nous verrons dans un prochain article des techniques donnant de meilleurs résultats que le dithering de Bayer.

### 3. Conclusion

Nous avons passé en revue les accès aux pixels sur les formats d'images les plus répandus. Nous n'avons pas traité tous les cas mais cette présentation devrait vous permettre de réaliser toutes les conversions possibles entre deux formats différents. N'hésitez pas à utiliser des tables de pré-calculs afin d'éviter des calculs répétés dans le parcours des pixels.

### 4. 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>

Autres matrices pour le Dithering

<http://scanline.ca/dithering/>

N'hésitez pas à me contacter :

Frédéric Mélantois

Email : [fmelantois@free.fr](mailto:fmelantois@free.fr)

## **.NET passionnément, tout simplement**

Traitement d'images sous .NET, partie 1

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