

Gestion de la transparence d'images en 32 bits (ou le

« compositing » d'images 32 bits en mode unsafe)

Cet article est composé de ce document et d'un fichier contenant les sources des démonstrations vous permettant d'approfondir quelque peu vos connaissances.

A noter qu'il est essentiel de tester et de lire les commentaires de code des démonstrations pour bien comprendre cet article.

Enfin, il est nécessaire que le lecteur soit quelque peu familier du namespace « System.Drawing.Imaging » pour comprendre avec plus de facilité cet article.

1. Théorie

De quoi s'agit-il ?

On appelle « Compositing » (ou composition) d'images » la superposition d'images ayant une transparence (du moins pour l'image superposée). Autant dire que nous le retrouvons partout dans l'infographie d'aujourd'hui.

Nous allons faire un peu de théorie pour comprendre les mécanismes de la composition d'images (en particulier 32bits).

La théorie

Dans le cadre du « compositing d'images », une notion de transparence pour l'image superposée est nécessaire pour composer les deux images en une seule. Cette transparence est quantifiée par ce que l'on appelle la couche alpha.

Chaque pixel de l'image superposée possède un coefficient alpha (de 0 à 255 pour une image 32 bits). Si la valeur est égale à 0, la transparence est totale pour le pixel. Si la valeur est maximum, le pixel est opaque et donc le pixel de l'image sous-jacente ne peut pas être visible.

Si on a deux images A et B avec des couches alpha et qu'on les superpose, l'image résultante C (image B se superpose à l'image A) se calcule par la formule suivante :

$$C = \text{alphaB} B + (1 - \text{alphaB}) \text{alphaA} A$$

alphaA représentant la couche alpha de l'image A.

alphaB représentant la couche alpha de l'image B.

Dans la formule, alphaA et alphaB ont pour bornes 0 et 1

On peut transposer la formule dans le cas d'une image 32bits (8 bits pour la composante bleu, 8 bits pour la composante verte, 8 bits pour la composante rouge et 8 bits pour la composante alpha d'un pixel).

Cette formule devient :

$$C = (\text{alphaB} B + (((255 - \text{alphaB}) \text{alphaA} A) / 255)) / 255$$

Qui se simplifie pour une meilleure performance et donc une approximation par :

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

$$C = (\text{alpha}B B + (((255 - \text{alpha}B) \text{alpha}A A) \gg 8)) \gg 8$$

Il y a approximation car un décalage de 8 représente une division par 256 et non 255. Cette approximation est faite généralement par la plupart des logiciels. A noter, que si nous avons du traitement d'images sérieux à réaliser (Imagerie médicale ou imagerie satellite), nous ne pourrions nous permettre cette approximation.

Vous avouerez que cette dernière formule même approximée représente énormément de calcul si on doit l'appliquer à tous les pixels d'une image ! Il faut savoir que la multiplication est une opération coûteuse.

D'où l'idée de réduire le calcul en considérant que l'on peut déjà avoir effectué le calcul $\text{alpha}A A$.

On a alors :

$$A' = (\text{alpha}A A) \gg 8$$

La composition d'image se trouve alors simplifiée en :

$$C = (\text{alpha}B B + (255 - \text{alpha}B) A') \gg 8$$

On gagne ici énormément de temps pour le calcul si toutes les valeurs possibles de A' sont déjà pré-calculées et donc en mémoire. On parle alors de mode pré-multiplié.

Si on fait de même avec l'image B :

$$B' = (\text{alpha}B B) \gg 8$$

On obtient :

$$C = B' + ((255 - \text{alpha}B)A') \gg 8$$

A l'inverse, si le calcul n'est pas déjà fait, on parle de mode non-pré-multiplié.

2. Que nous propose .Net ?

On retrouve dans le Framework .NET cette notion de mode pré-multiplié ou non dans les différents formats de pixels.

L'énumération `PixelFormat` du namespace « `System.Drawing.Imaging` » fait état de plusieurs modes pré-multipliés (par exemple : `Format32bppPArgb` ou encore `Format64bppPArgb`).

Si une image est créée suivant le format `Format32bppPArgb`, ses composantes Rouge, Bleu et Verte sont affectées par le coefficient alpha par la formule A' précédente. Par exemple, la valeur de rouge sera modifiée en $\text{alpha} \times \text{Rouge} / 255$.

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

3. Demo 1 : Application par l'objet Graphics

L'exemple que nous allons présenter ici est des plus classiques. L'exemple de la démo 2 le sera moins et vous permettra de comprendre (de manière simplifiée) comment peut fonctionner à plus bas niveaux GDI+ (l'API sur laquelle s'appuie la plupart des classes du namespace « System.Drawing »).

Comment faire de la superposition d'images 32 bits avec .NET ? Cela se fait très facilement grâce à la classe Graphics.

```
public static Image Add(Image imageFond, Image imageOver)
{
    Image img = imageFond.Clone() as Image;
    //Graphics nous simplifie la vie ;-)
    using (Graphics g = Graphics.FromImage(img))
    {
        //superposition de l'image
        g.DrawImage(imageOver, new Rectangle(0,0, img.Width, img.Height));
    }
    imageFond.Dispose();
    imageOver.Dispose();
    return img;
}
```

Difficile de faire plus simple. La compréhension de l'exposé théorique précédent n'est même pas nécessaire pour arriver à ses fins.

Compliquons la tâche. Essayons d'introduire une valeur alpha générale pour tous les pixels de l'image superposée (en anglais, on appelle cela le « fading »). On peut s'apercevoir qu'on le fait avec une impressionnante facilité :

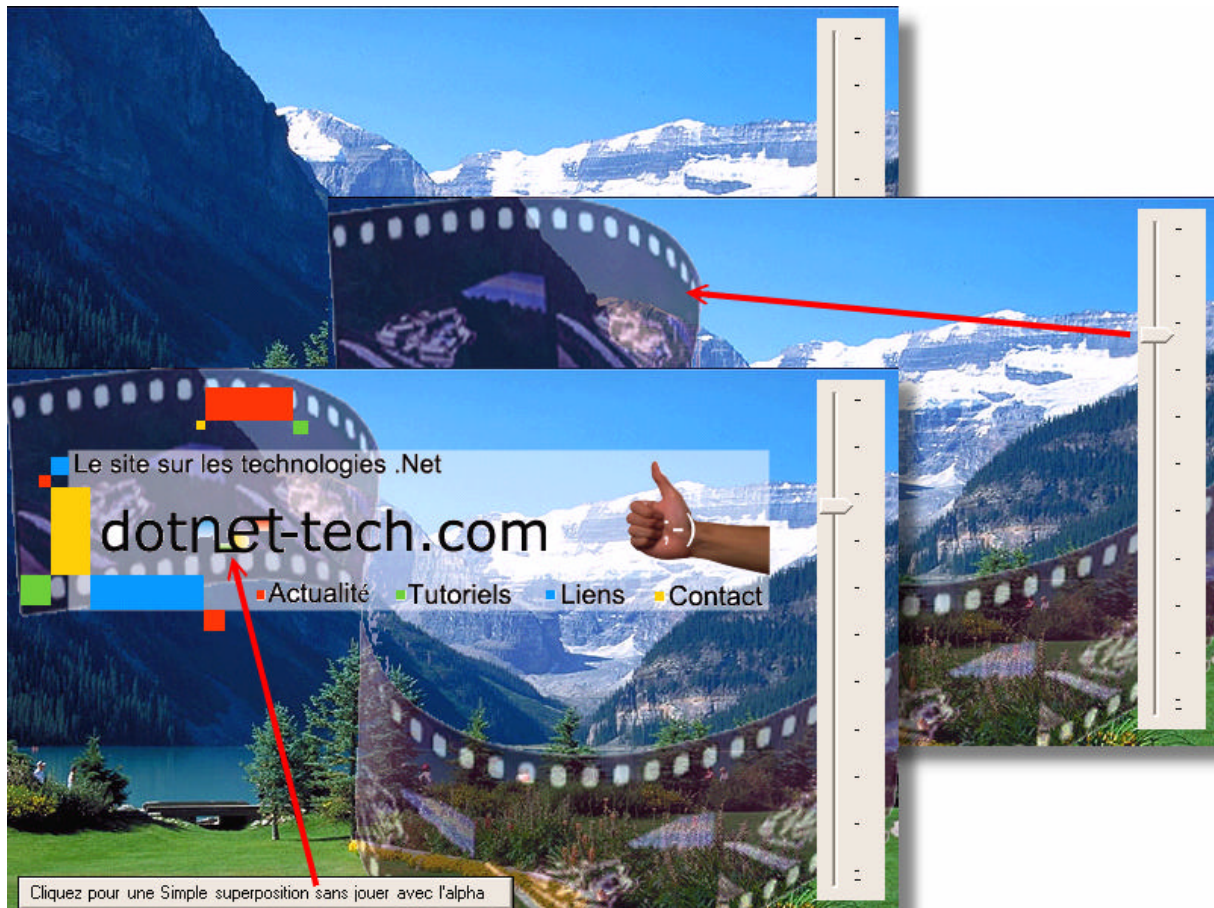
```
using (Graphics g = Graphics.FromImage(img))
{
    //on va modifier les attributs de l'image
    using (ImageAttributes at = new ImageAttributes())
    {
        ColorMatrix m = new ColorMatrix();
        //modification de la matrice de couleur (ce qui concerne la couche alpha)
        m.Matrix33 = alpha;
        at.SetColorMatrix(m);
        //on superpose l'image avec de nouveaux attributs
        g.DrawImage(imageOver, new Rectangle(0,0, img.Width, img.Height), 0, 0, img.Width, img.Heig
    }
}
```

Le « compositing » géré par GDI+ fait sans doute appel à des tables de pré-calculs déjà chargées en mémoire. Le mode Pré-multiplié est à priori utilisé étant donnée la vitesse d'exécution de notre programme.

Le rendu visuel est le suivant :

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits



Comme vous vous en doutez, et puisqu'avec cette exemple des plus classiques, il n'y avait pas sujet à faire un article (quoi que, voir les liens dans « En savoir plus »), la démonstration n°2 va mettre en pratique l'exposé théorique que nous avons fait au début de cet article et vous ouvrir d'autres perspectives de développement.

4. Démo 2 : Application par le code unsafe

La démo 2 va nous montrer en effet une toute autre approche. Nous allons devoir écrire beaucoup plus de codes mais nous allons avoir une totale liberté de manœuvre pour traiter nos images.

Il est nécessaire pour suivre cette partie d'avoir des notions de manipulations d'images en mode unsafe. Je peux vous conseiller ces liens :

Unsafe at the limit (manipulation de pointeurs en c#)

<http://msdn.microsoft.com/library/en-us/dncscol/html/csharp10182001.asp>

Unsafe Image Processing (manipulation d'images 24 bits)

<http://msdn.microsoft.com/library/en-us/dncscol/html/csharp11152001.asp>

Avec les images en 32 bits, l'approche est la même sauf que l'on manipule 4 bytes (ou un uint) pour chaque pixel. Les 8 premiers bits sont consacrés à la composante bleue, puis les 8 autres à la verte, les 8 autres à la rouge et enfin les 8 dernières à

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

la composante alpha. Du fait, que l'on manipule 4 bytes, on n'a pas à gérer les fins de lignes comme dans le cas des images 24 bits (en particulier quand la longueur de l'image en pixels n'est pas divisible par 3). La manipulation d'une image 32bits s'en trouve donc facilitée :

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


for (int y = 0; y < height; y++)
    for (int x = 0; x < width; x++)
    {
        for (byte i=0; i<3; i++)
        {
            //inversion des composants Bleu, vert, rouge
            newPixel[i] = 255 - oldPixel[i];
            oldPixel++;
            newPixel++;
        }
        //deplacement du pointeur : on ne touche pas à l'alpha
        oldPixel++;
        newPixel++;
    }
}
```

Cet exemple produit un négatif de l'image sans toucher à la couche alpha.

Nous allons maintenant appliquer la théorie en travaillant en mode pré-multiplié avec nos images pour réduire le nombre de calcul :

```
unsafe
{
    BitmapData bmpDataOld=oldBitmap.LockBits(new Rectangle(0,0,width,height),
        ImageLockMode.ReadWrite,PixelFormat.Format32bppPArgb);
    BitmapData bmpDataOver=overBitmap.LockBits(new Rectangle(0,0,width,height),
        ImageLockMode.ReadWrite,PixelFormat.Format32bppPArgb);
    BitmapData bmpDataNew=newBitmap.LockBits(new Rectangle(0,0,width,height),
        ImageLockMode.ReadWrite,PixelFormat.Format32bppPArgb);

    byte * newPixel = (byte*) (void *)bmpDataNew.Scan0;
    byte * oldPixel = (byte *) (void *)bmpDataOld.Scan0;
    byte * overPixel = (byte *) (void *)bmpDataOver.Scan0;
}
```

 Mode pré-multiplié

Il nous faut appliquer maintenant la formule théorique que nous avons vue précédemment :

$$C = B' + ((255 - \alpha B)A') \gg 8$$

B' et A' sont les images dont les composantes sont pré-multipliés.

Concrètement, nous allons obtenir pour notre code :

$$\text{newPixel}[i] = (\text{byte}) (((\text{oldPixel}[i] * (255 - \text{overPixel}[3])) \gg 8) + \text{overPixel}[i])$$

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

i représente l'index d'une des composantes (bleue, verte, rouge, alpha)
overPixel[3] est la composante alpha du pixel considéré.

A noter, que nous faisons une optimisation par l'approche byte + byte = byte, ce qui n'est évidemment pas toujours vrai, mais vrai pour nos bornes de calculs. On évite ainsi un test de condition sur l'addition des deux bytes (est-elle supérieure à 255 ?). Toute optimisation est bonne en traitement d'images quand on sait que le traitement s'exécute sur des dizaines de milliers de pixels. Il est bon de s'équiper d'un « profiler » afin de savoir où telle routine prend du temps.

Et nous allons continuer à optimiser...

La multiplication est une opération coûteuse. Puisque la plupart des images que nous traiterons en général ont une taille dépassant 256x256 pixels, nous pouvons nous servir, de ce que l'on appelle, une table de pré-calculs. Il s'agit tout simplement d'un tableau que nous allons construire avant le traitement de l'image et sur lequel nous allons appuyer pour faire des affectations au lieu d'opérations, ce qui réduit considérablement le temps.

```
//fabrication de la table de pré-calcul
private static byte[][] GetArrayForMult()
{
    byte [][] tab = new byte[256] [];

    for (int i=0; i<256; i++)
    {
        tab[i] = new byte[256];
        for (int j=0; j<256; j++)
        {
            tab[i][j] = (byte) ((i*(255-j))>>8);
        }
    }
    return tab;
}
```

A noter que nous avons suivi la recommandation d'utiliser des tableaux imbriqués plutôt qu'un tableau multidimensionnel conformément à ce document :

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenetchapt05.asp>

On y retrouve une partie de la formule précédente. Il ne nous reste plus qu'à transmettre ces tableaux à notre routine de traitements. Et nous obtenons :

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

```
for (int y = 0; y < height; y++)
    for (int x = 0; x < width; x++)
    {
        for (byte i=0; i<4; i++)
        {
            //calcul sans table de pré-calcul : beaucoup plus long!!!!
            //int val = (oldPixel[i]*(255-overPixel[3]))/255 + overPixel[i];
            newPixel[i] = (byte)(tab[oldPixel[i]][overPixel[3]]+overPixel[i]);
        }
        oldPixel+=4;
        overPixel+=4;
        newPixel+=4;
    }
```

C'est-à-dire une affectation, une addition, un cast.

Il nous reste à gérer le « fading » de la démonstration 1. C'est-à-dire qu'on applique un coefficient alpha à l'image superposée. Vous allez me dire que c'est extrêmement simple puisqu'il suffit de multiplier nos composantes bleue, verte, rouge et alpha par le float dont les bornes sont situées entre 0 et 1. Et cela donne :

```
for (int y = 0; y < height; y++)
    for (int x = 0; x < width; x++)
    {
        byte over = (byte)(alpha*(float)overPixel[3]);
        for (byte i=0; i<4; i++)
        {
            newPixel[i] = (byte)(tab[oldPixel[i]][over]+(byte)(alpha*(float)overPixel[i]));
        }
        oldPixel+=4;
        overPixel+=4;
        newPixel+=4;
    }
```

C'est-à-dire un calcul extrêmement lent puisque nous avons des multiplications encore à gérer et en plus avec des floats suivi d'un ensemble de casts !

Notre réflexe premier est donc de penser à nouveau à une table de pré-calculs. Sauf que dans ce cas-là, nous avons à gérer un float...

L'idée serait peut-être de nous servir de notre table de pré-calcul déjà existante. Comment s'en servir ?

Postulons que le résultat à obtenir est :

tab[overPixel[i]][calcAlpha] c'est-à-dire une affectation

Il nous faut donc calculer calcAlpha.

Nous savons (par notre postulat) que :

tab[overPixel[i]][calcAlpha] = alpha * overPixel[i]

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

Appelons overPixel[i] : y pour simplifier la lecture, nous avons donc

```
tab[y][calcAlpha] = alpha * y
```

Nous remplaçons notre tableau par la formule :

```
y * (255-calcAlpha)>>8 = alpha * y
```

ou encore

```
(255-calcAlpha)/255 = alpha
```

donc

```
calcAlpha = 255 - (alpha *255)
```

Il nous suffit de l'appliquer à notre traitement et nous aurons minimisé énormément le coût de l'opération de « fading » par de simples affectations et casts.

```
//l'alpha est rapporté aux bornes [0,255]
//pour le calcul, voir l'article
byte calcAlpha = (byte) (255f*(1f-alpha));

unsafe
{
    BitmapData bmpDataOld=oldBitmap.LockBits(new Re
```

Et dans les boucles :

```
for (int y = 0; y < height; y++)
    for (int x = 0; x < width; x++)
    {
        byte over = tab[overPixel[3]][calcAlpha];
        for (byte i=0; i<4; i++)
        {
            //calcul si on n'a pas à gérer un alpha supplémentaire
            //int val = (oldPixel[i]*(255-overPixel[3])/255 + overPixel[i];
            //le même calcul avec la table de pré-calcul
            //byte val = (byte) (tab[oldPixel[i]][overPixel[3]]+overPixel[i]);
            newPixel[i] = (byte) (tab[oldPixel[i]][over]+tab[overPixel[i]][calcAlpha]);
        }
        oldPixel+=4;
        overPixel+=4;
        newPixel+=4;
    }
}
```

En profilant le code, on s'aperçoit que l'on gagne du temps en passant par la variable intermédiaire "over".

A noter que nous obtenons une performance presque satisfaisante : 10% à 15% de temps de calcul en plus par rapport à la démonstration 1.

Cette approche nous permet de mieux comprendre l'aspect théorique mais aussi de pouvoir personnaliser, si on le souhaite, le « compositing » ou composition

.NET passionnément, tout simplement

Gestion de la transparence d'images en 32 bits

d'images. On peut par exemple, réaliser avec cette version de code unsafe du « compositing » sur un seul canal (Rouge ou Bleu ou Vert) ou encore mettre des conditions à notre composition (exemple : Evaluer la luminosité des pixels de l'image sous-jacente et la luminosité des pixels de l'image à superposer, et de modifier en conséquence le pixel se superposant pour assurer une meilleure visibilité de l'ensemble de l'image !).

A vos claviers, et bon code !

5. Conclusion

Nous venons de montrer le fonctionnement des superpositions d'images 32bits avec effet de transparence. L'étude théorique des mécanismes nous a conduit à gérer nous-même cette transparence via le mode unsafe. En réalisant encore quelques optimisations, nous pourrions nous lancer sur la construction d'un logiciel d'images avec effet de couches ou calques (pratiquement comme sous Photoshop).

6. En savoir plus

Après avoir écrit mon article, je me suis mis à rechercher des liens à vous proposer, et je suis tombé sur une série d'articles magnifiques (sans le mode unsafe d'où l'intérêt au moins de mon article ;-) qui complétera parfaitement celui-ci :

Transparency Tutorial with c# part 1

<http://www.codeproject.com/cs/media/CsTranspTutorial1.asp>

Transparency Tutorial with c# part 2

<http://www.codeproject.com/cs/media/CsTranspTutorial2.asp>

Transparency Tutorial with c# part 3

<http://www.codeproject.com/cs/media/CsTranspTutorial3.asp>

Pour l'optimisation et la performance, je vous recommande ces liens :

Writing Faster Managed Code: Know What Things Cost

<http://msdn.microsoft.com/library/?url=/library/en-us/dndotnet/html/fastmanagedcode.asp>

Writing High-Performance Managed Applications : A Primer

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/highperfmanagedapps.asp>

Improving .NET Application Performance and Scalability (mon livre de chevet)

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenet.asp>

N'hésitez pas à me contacter :

Frédéric Mélantois

Email : [fmelantois\[arobase\]free.fr](mailto:fmelantois[arobase]free.fr)