

GDI+ : le double buffering

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.

1. Introduction

La problématique du scintillement d'écran

Dès que l'on fait de la programmation graphique, nous sommes très souvent confrontés au problème du scintillement de l'écran lorsque des parties d'écrans sont mises à jour. Plus le traitement d'affichage est long, plus le problème de rafraîchissement rapide de l'écran est important.

Le pourquoi

Il y a quelques années, avant l'arrivée de Windows, les développeurs se servaient des interruptions DOS pour interagir sur l'écran. Le souci d'optimisation était total du fait de la faible capacité des processeurs et de la mémoire. La plupart des routines graphiques étaient en assembleur. Le mode graphique le plus utilisé pour les jeux vidéos était le 320*200 en 256 couleurs soit 16ko en mémoire. La mémoire vidéo étant de 64ko à l'époque, ce mode offrait donc 4 blocs mémoires de 16ko dont 1 bloc correspondait à ce qui était affiché à l'écran. On se servait tout naturellement des autres blocs pour écrire dedans et les déplacer dans le bloc d'affichage.

Si je vous parle de tout cela, c'est certes avec nostalgie, mais c'est surtout pour vous faire comprendre qu'à cette époque il était naturel (indispensable) de préparer son affichage en écrivant ses dessins dans une zone mémoire et basculer le bloc mémoire vers la zone d'affichage le moment venu. Aujourd'hui, on parle de la technique du « double buffering » mais ceci n'a rien de nouveau comme vous pouvez le constater.

L'arrivée de processeurs plus fortement cadencées, de cartes vidéos plus puissantes, de beaucoup plus de mémoire a permis l'arrivée de Windows, greffé sur le DOS dans un premier temps, puis en tant que système d'exploitation à part entière. La production d'interface graphique est devenue plus aisée. Nous en avons même oublié les bonnes vieilles techniques d'optimisation. Mais quand nos graphiques deviennent plus complexes, si nous affichons directement, le scintillement vient nous rappeler que ces techniques sont toujours d'actualité.

Cette question revient très souvent dans les newsgroups. C'est pourquoi je me suis décidé à écrire cet article. Nous examinerons les différentes techniques possibles sous le Framework 1.1 mais aussi sur ce que nous propose le Framework 2.0 (en beta, au moment de la rédaction de cet article).

.NET passionnément, tout simplement

GDI+ : le double buffering

2. Demo 1 : Un exemple de scintillement

Nous allons prendre l'exemple d'un dessin sur un formulaire tout entier. Faisons en sorte que le calcul prenne du temps. J'ai choisi le problème connu de « la courbe du chien » pour illustrer cet article. J'ai un peu surchargé le tout par des remplissages de polygones.

Nous dessinons sur le formulaire en surchargeant la méthode OnPaint.

C#

```
protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
    //..on dessine directement
}
```

VB

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)
    e.Graphics.SmoothingMode = SmoothingMode.AntiAlias
    '..on dessine directement
End Sub
```

Un Timer va nous permettre de rafraîchir à intervalle régulier le formulaire.

C#

```
private void TimerRepaint_Tick(object sender, System.EventArgs e)
{
    _angle += 2;
    if (_angle > 360)
        _angle = 0;
    this.Invalidate();
}
```

VB

```
Private Sub TimerRepaint_Tick(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles TimerRepaint.Tick

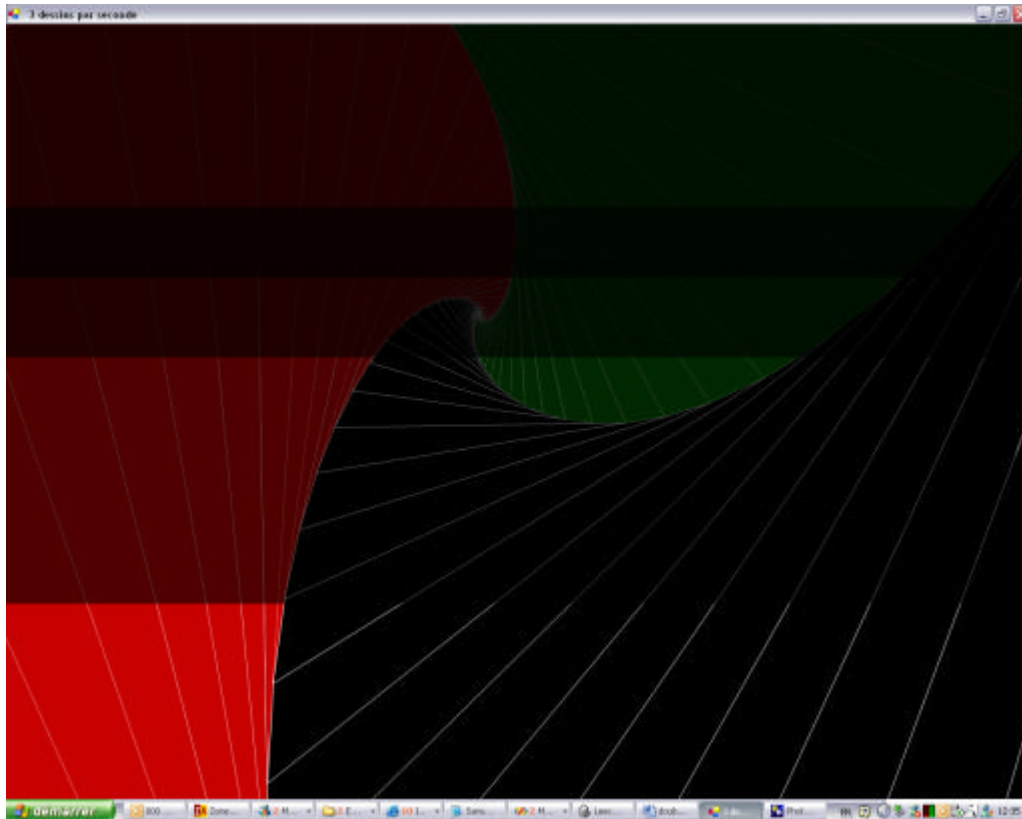
    _angle += 2.0
    If (_angle > 360) Then
        _angle = 0.0
    End If
    MyBase.Invalidate()

End Sub
```

Lorsque nous lançons l'application, nous obtenons un scintillement très désagréable de la fenêtre :

.NET passionnément, tout simplement

GDI+ : le double buffering



3. Demo 2 : Solution manuelle

Nous allons essayer d'éviter le scintillement en dessinant dans un premier temps en dehors de la zone d'affichage directe, par exemple sur une image. Puis nous transférerons l'image dessinée (c'est-à-dire le buffer) vers la zone d'affichage. Le principe est finalement très simple.

C#

```
private Bitmap _bitmapTemp;

protected override void OnPaint(PaintEventArgs e)
{
    //on crée une image de la taille du controle
    _bitmapTemp = new Bitmap(this.ClientSize.Width,
this.ClientSize.Height);
    //on crée un Graphics à partir de la Bitmap (= buffer)
    using (Graphics g = Graphics.FromImage(_bitmapTemp))
    {
        //On dessine
        Dessiner(g);
    }
    e.Graphics.DrawImage(_bitmapTemp, 0,0);
    _bitmapTemp.Dispose();
    _bitmapTemp = null;
    _cptDessinParSeconde++;
}

protected override void OnPaintBackground(PaintEventArgs pevent)
{
    //on neutralise le rafraichissement du background
}
```

.NET passionnément, tout simplement

GDI+ : le double buffering

VB

```
Private _bitmapTemp As Bitmap

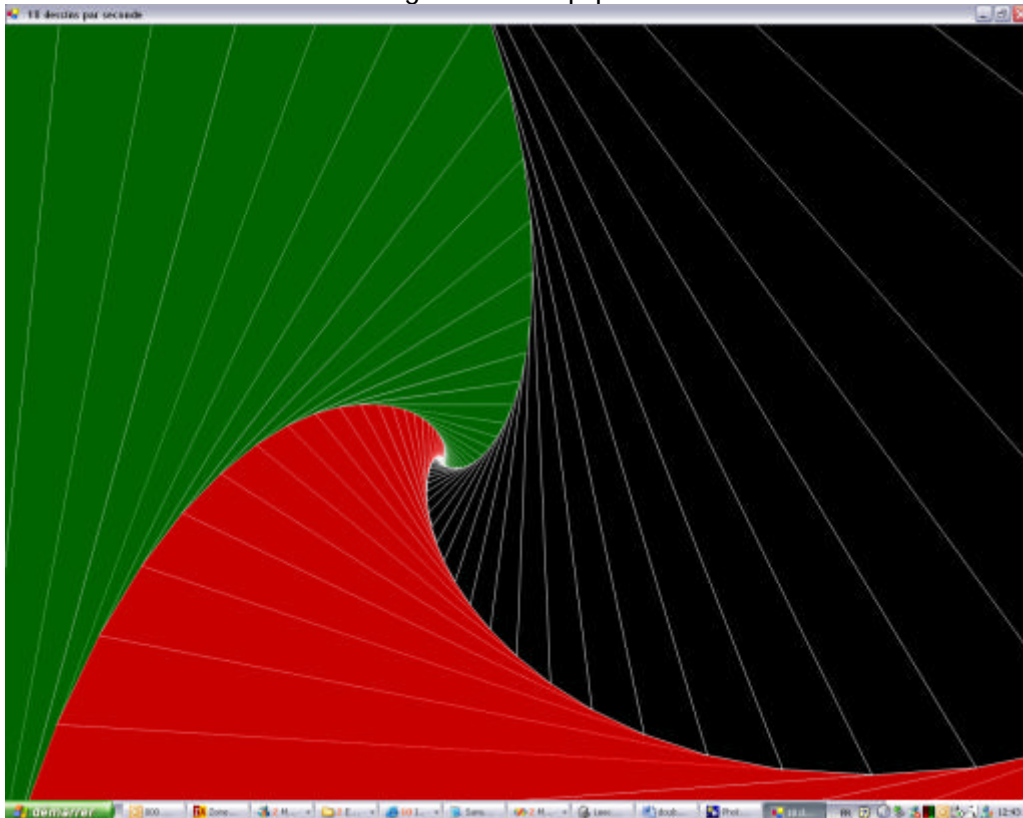
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)

    'on crée une image de la taille du controle
    _bitmapTemp = New Bitmap(MyBase.ClientSize.Width, _
MyBase.ClientSize.Height)
    'on crée un Graphics à partir de la Bitmap (= buffer)
    Dim g As Graphics = Graphics.FromImage(_bitmapTemp)
    'On dessine
    Dessiner(g)
    g.Dispose()
    e.Graphics.DrawImage(_bitmapTemp, 0, 0)
    _bitmapTemp.Dispose()
    _bitmapTemp = Nothing
    _cptDessinParSeconde += 1

End Sub

Protected Overrides Sub OnPaintBackground(ByVal e As PaintEventArgs)
    'on neutralise le rafraichissement du background
End Sub
```

Et nous obtenons un affichage beaucoup plus fluide :



4. Demo 3 : Solution automatique

La solution automatique nous est apportée par le framework. Le Double buffering est bien évidemment présent et il s'appuie pour quelques aspects sur la couche

.NET passionnément, tout simplement

GDI+ : le double buffering

native ce qui lui confère bien évidemment une meilleure performance. Il s'agit d'un algorithme générique, c'est-à-dire que dans certains cas, on préférera le Double Buffering manuel (en particulier pour l'invalidation de petites zones graphiques).

La mise en place de ce double buffering automatique est des plus simples :

C#

```
public Form1()
{
    InitializeComponent();
    this.SetStyle(ControlStyles.AllPaintingInWmPaint |
ControlStyles.UserPaint | ControlStyles.DoubleBuffer, true);
}
```

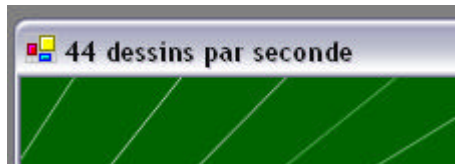
VB

```
Public Sub New()

    MyBase.New()
    InitializeComponent()
    Me.SetStyle(ControlStyles.AllPaintingInWmPaint Or _
ControlStyles.UserPaint Or ControlStyles.DoubleBuffer, True)

End Sub
```

Lancez l'application et vous constaterez que cette solution automatique est très performante quand il s'agit de rafraîchir tout le contrôle. Vous pouvez en particulier comparer avec les solutions précédentes par le nombre de dessins créés par seconde (Voir la barre de titre du contrôle).



5. Un peu plus loin

🔧 Quelques explications

Une fenêtre ou un contrôle peut être rafraîchi à l'écran individuellement par le principe de la boucle de message de Windows. Examinons ce qui se passe en dehors de la couche .NET : lorsque nous invalidons un contrôle, un message est envoyé vers Windows. Celui-ci est alors mis dans la queue des messages. La boucle de message traite un par un les messages suivant la queue. Le message est envoyé vers une procédure Windows pour y être traitée. Celle-ci va permettre le rafraîchissement de la zone.

.NET ne participe pas à la mécanique complète. Le Framework n'intervient que pour signaler qu'il faut envoyer un message (c'est ce que l'on fait quand on invalide le control), et il agit à la réception d'un type de message (un mécanisme interne lance l'action de redessiner par exemple). Toutes les actions se passant entre celles-ci, se font dans la mécanique sous-jacente, c'est-à-dire hors code managé.

.NET passionnément, tout simplement

GDI+ : le double buffering

Où sont captés les messages ? Les messages sont réceptionnés par la procédure Windows sous-jacente. Nous disposons d'un accès par le code managé en surchargeant la méthode WndProc de notre contrôle. Cette méthode traite des messages reçus.

Le message

De quoi est constitué un message ?

Sous .NET, le message est une structure :

- HWND permet d'identifier la fenêtre pour qui le message est destiné
- Msg est l'identifiant permettant de connaître le type de message (15 correspond au redessin de la fenêtre)
- LParam et WParam contiennent des informations sur le message.
- Result correspond à la valeur de retour du message.

En C, le message a une structure légèrement différente :

- hwnd permet d'identifier la fenêtre pour qui le message est destiné
- message est l'identifiant permettant de connaître le type de message (15 correspond au redessin de la fenêtre)
- lParam et wParam contiennent des informations sur le message.
- Time correspond à l'heure où le message a été posté
- pt correspond à la position de la souris lors de l'envoi du message

La boucle de message

Celle-ci n'est absolument pas apparente sous .NET.

Il nous faut créer un projet win32 en c++ non managé pour avoir une idée de cette boucle de message :

C++

```
//...
// Boucle de messages principale :
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

return (int) msg.wParam;
```

GetMessage permet de réceptionner le dernier message dans la queue. La boucle continue jusqu'à ce que GetMessage réceptionne un message pour quitter l'application.

TranslateMessage permet de transformer le message.

DispatchMessage envoie le message à la procédure Windows.

.NET passionnément, tout simplement

GDI+ : le double buffering

La procédure Windows

Examinons la procédure Windows dans le code c++ par défaut :

C++

```
// WM_COMMAND - traite le menu de l'application
// WM_PAINT - dessine la fenêtre principale
// WM_DESTROY - génère un message d'arrêt et retourne
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam,
LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Analyse les sélections de menu :
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        // TODO : ajoutez ici le code de dessin...
        EndPaint(hWnd, &ps);
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Celle-ci correspond à un gros « switch case ». Ici, WM_PAINT correspond à la valeur 15. C'est ici que l'on dessine le formulaire.

Sous .NET, cette procédure windows n'est pas visible. Toutefois, il est possible d'ajouter des cas suivant le type de message (par exemple ceux que .NET ne propose pas de traiter).

Vous pouvez intercepter de nombreux messages : par exemple le changement de thèmes Windows (WM_THEMECHANGED, 0x31A), la perte du focus sur la fenêtre

.NET passionnément, tout simplement

GDI+ : le double buffering

(WM_KILLFOCUS, 0x0008), la demande de fin de session (WM_QUERYENDSESSION, 0x0011)...

Comment intercepter certains messages sous .NET ? Il nous suffit de surcharger la méthode WndProc du control.

C#

```
protected override void WndProc(ref Message m)
{
    if (m.Msg == 15)
    {
        //15 = message pour redessiner le control
        cpt++; //on fait ce que l'on veut
    }

    base.WndProc (ref m);
}
```

VB

```
Protected Overrides Sub WndProc(ByRef m As Message)

    If (m.Msg = 15) Then
        '15 = message pour redessiner le control
        Me.cpt += 1 'on fait ce que l'on veut
    End If
    MyBase.WndProc(m)

End Sub
```

En comprenant un peu mieux ce fonctionnement, on peut réaliser plus facilement quelques optimisations. Par exemple, dans notre démonstration n°2, il est inutile d'invalider la fenêtre tant que l'image (le buffer) n'a pas été complètement peinte. Car on ajouterait des messages inutiles dans la mécanique sous-jacente (donc du temps processeur). Il convient donc dans notre exemple d'émettre la condition suivante :

C#

```
if (_bitmapTemp == null)
    this.Invalidate();
```

VB

```
If (_bitmapTemp Is Nothing) Then
    MyBase.Invalidate()
End If
```

6. Les nouveautés du Framework 2.0

La solution automatique

Le Framework 2.0 apporte son lot de nouveautés. Tous les contrôles sont dotés d'une nouvelle propriété : DoubleBuffered, qu'il suffit de mettre à true, pour que le Double Buffering soit géré.

.NET passionnément, tout simplement

GDI+ : le double buffering

Sinon, le « SetStyle » est toujours disponible. S'ajoute de nouveaux ControlStyles dont le ControlStyles.OptimizedDoubleBuffer. Ce dernier usage est recommandé si vous développez vos propres contrôles.

C#

```
this.SetStyle(ControlStyles.AllPaintingInWmPaint |  
ControlStyles.UserPaint | ControlStyles.OptimizedDoubleBuffer, true);
```

VB

```
Me.SetStyle(ControlStyles.AllPaintingInWmPaint Or ControlStyles.UserPaint _  
Or ControlStyles.OptimizedDoubleBuffer, True)
```

Je vous laisse tester les différences qu'il peut exister entre le « DoubleBuffer » et le « OptimizedDoubleBuffer ».

La solution manuelle

Le Framework 2.0 nous offre les outils pour gérer notre propre double buffering.

C#

```
protected override void OnPaint(PaintEventArgs e)  
{  
    BufferedGraphicsContext cnt = BufferedGraphicsManager.Current ;  
  
    BufferedGraphics bg = cnt.Allocate(this.CreateGraphics(),  
this.DisplayRectangle);  
    Dessiner(bg.Graphics);  
  
    bg.Render(this.CreateGraphics());  
    bg.Dispose();  
  
    _cptDessinParSeconde++;  
}
```

VB

```
Protected Overrides Sub OnPaint(ByVal e As PaintEventArgs)  
  
    Dim cnt As BufferedGraphicsContext = BufferedGraphicsManager.Current  
    Dim bg As BufferedGraphics = cnt.Allocate(MyBase.CreateGraphics(), _  
MyBase.DisplayRectangle)  
    Dessiner(bg.Graphics)  
  
    bg.Render(MyBase.CreateGraphics())  
    bg.Dispose()  
  
    _cptDessinParSeconde += 1  
  
End Sub
```

L'objet BufferedGraphics gère un buffer en mémoire. Sa méthode Render permet de basculer ce qui est en mémoire vers la zone d'affichage.

La classe responsable de la gestion des BufferedGraphics est la classe BufferedGraphicsContext. Elle permet d'allouer un espace pour ces buffers.

.NET passionnément, tout simplement

GDI+ : le double buffering

La classe BufferedGraphicsManager permet par exemple de récupérer le BufferedGraphicsContext courant.

On peut bien évidemment travailler avec plusieurs BufferedGraphics.

7. En savoir plus

Vous pouvez adapter le code de cet article pour réaliser votre propre économiseur d'écran :

How to develop a screen saver in C#

<http://www.codeproject.com/csharp/scrframework.asp>

Autres liens :

The WM_PAINT message

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/gdi/pantdraw_4k6d.asp

Custom Control Painting and Rendering

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

La FAQ GDI+ de Bob Powell

<http://www.bobpowell.net/faqmain.htm>

Visual Studio 2005 Library

<http://msdn2.microsoft.com/library/default.aspx>

8. Notes

Certains produits mentionnés ne sont pas encore commercialisés. Ils sont en phase de test. Si vous souhaitez obtenir Visual Studio 2005 en version beta ou en version finale dès sa disponibilité, vous pouvez souscrire un abonnement MSDN

<http://www.microsoft.com/france/msdn/abonnements/presentation.asp>

N'hésitez pas à me contacter :

Frédéric Mélantois

Email : fmelantois@free.fr