13

Améliorer le rendu avec le High Level Shader Language

Certains diront que les effets graphiques sont l'ingrédient indispensable à la réalisation d'un bon jeu, d'autres répliqueront qu'il ne s'agit que de fioritures visuelles qui exigent du matériel graphique de plus en plus puissant. Cependant, impossible de faire l'impasse sur des termes comme `glow`, `blur`, ou `bloom`. En effet, aujourd'hui, toutes les grandes productions en usent, et parfois en abusent.

Ce chapitre présente les shaders ainsi que le langage que vous utiliserez pour les programmer, à savoir le HLSL. Vous découvrirez aussi plusieurs exemples de shaders et comment les utiliser dans un projet XNA.

La première partie de ce chapitre explique ce que sont les shaders et présente le HLSL. Vous découvrirez ensuite la syntaxe du HLSL.

Les shaders et XNA

Un shader est un programme directement exécutable par la carte graphique. Il permet de transférer la gestion des lumières, ombres, textures, etc., au processeur graphique et de soulager de cette charge de travail le processeur de l'ordinateur. Ainsi, à partir du même contenu de base, mais en utilisant divers shaders, vous pouvez obtenir des rendus complètement différents.

Vertex shaders et pixel shaders

On distingue deux types de shaders : les *vertex shaders* et les *pixels shaders*.

Les vertex shaders sont exécutés sur tous les vertices d'un objet. Si vous dessinez un simple triangle à l'écran, le vertex shader sera exécuté pour les trois vertices qui composent ce triangle. Mais les vertex shaders peuvent également être exécutés sur des sprites en 2D : en effet, ceux-ci sont constitués de 4 vertices (un dans chaque coin du sprite).

Quant aux pixels shaders, ils sont exécutés sur tous les pixels visibles à l'écran de l'objet que vous dessinez, qu'il s'agisse d'un objet en 3D ou d'un sprite en 2D.

Chronologiquement, le processus est le suivant :

1. Le GPU (processeur graphique) reçoit des paramètres et des vertices (leur position, couleur, texture, etc., tout dépend du choix que vous avez fait).
2. Le vertex shader est d'abord exécuté. En sortie, vous disposez des vertices modifiés (ou non, tout dépend du shader).
3. Ces données passent ensuite à l'étape de rastérisation. Au cours de cette étape, les données vectorielles sont converties en données composées de pixels pouvant être affichés à l'écran.
4. Le résultat de la rastérisation est ensuite envoyé au pixel shader. Après l'exécution de ce programme, le GPU retourne la couleur qui devra être affichée.

Comme pour un processeur classique, le processeur graphique ne comprend que des instructions de bas niveau. Ainsi, pour faciliter le travail des développeurs, des langages de plus haut niveau (c'est-à-dire, plus intelligibles) à la syntaxe très proche des langages tels que le C ont été créés. Du côté d'OpenGL, le langage standardisé est le GLSL. Le fabricant de cartes graphiques NVidia a, quant à lui, créé le langage Cg. Enfin, de son côté, Microsoft a créé le HLSL (*High Level Shader Language*). C'est ce dernier langage qui est utilisé par l'API DirectX, et donc par XNA.

> **Définition**
> OpenGL est une API multi-plate-forme d'affichage en trois dimensions développée par Silicon Graphics.

Dans XNA, pour afficher quelque chose à l'écran vous devez passer obligatoirement par HLSL. Cependant, pour vous faciliter les choses (c'est le mot d'ordre de XNA après tout !) et vous éviter de devoir vous attaquer directement au HLSL, les développeurs ont créé les classes `SpriteBatch` et `BasicEffect`. En arrière-plan, ces classes travaillent directement avec des shaders. Vous pouvez d'ailleurs retrouver le code source de ces shaders à cette adresse : *http://creators.xna.com/fr-FR/education/catalog/*.

Ajouter un fichier d'effet dans XNA

Les fichiers d'effets écrits en HLSL sont automatiquement gérés par le Content Manager de XNA. Exactement comme vous ajoutez n'importe quel type de fichier (une police de caractères par exemple), vous pouvez ajouter un fichier d'effet à XNA (figure 13-1).

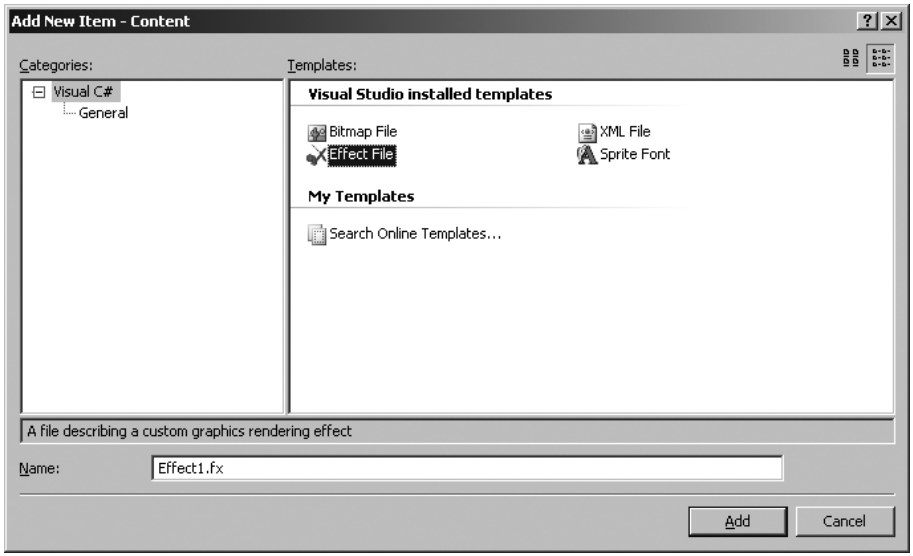


Figure 13-1

Ajout d'un fichier .fx au projet

Vous avez ensuite toute latitude pour éditer le fichier `.fx` directement dans Visual Studio, avec un bloc-notes ou bien, si vous désirez des fonctionnalités plus avancées, notamment pour déboguer un effet, via FX Composer de NVidia (figure 13-2).

FX Composer

Téléchargez FX Composer à l'adresse suivante :

http://developer.nvidia.com/object/fx_composer_home.html/

Puis, installez le programme comme une application classique.

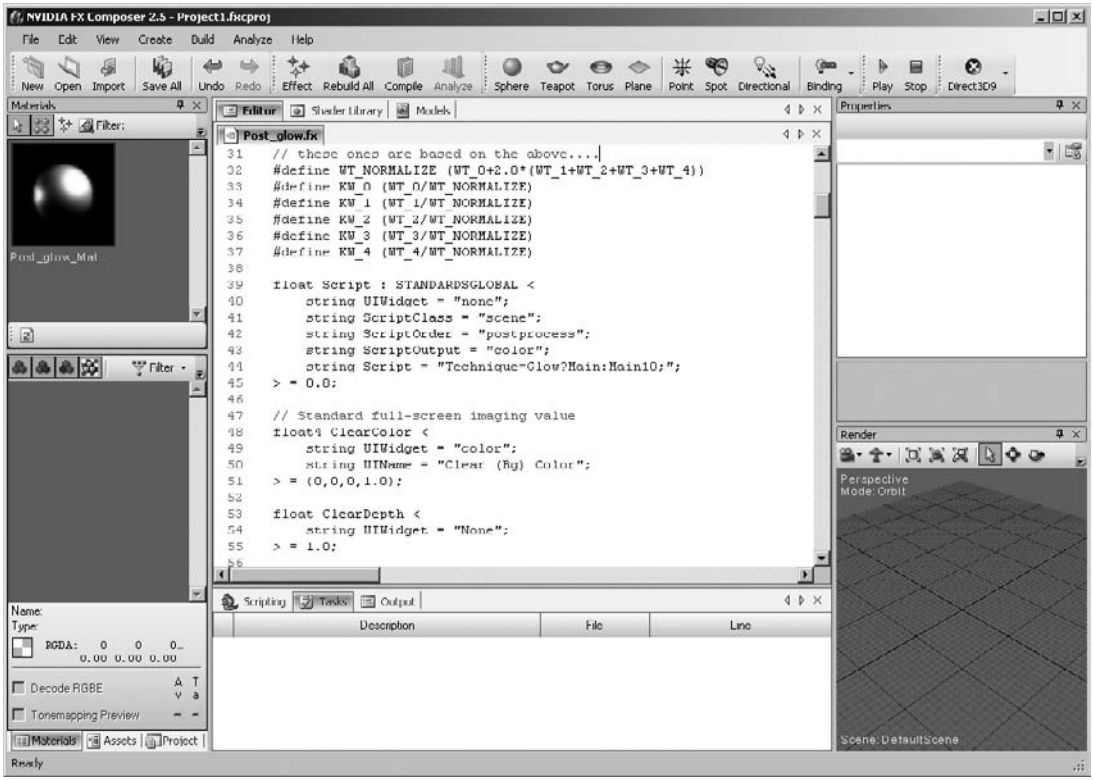


Figure 13-2

FX Composer, l'éditeur de shaders de NVidia

Syntaxe du langage HLSL

Vous allez maintenant découvrir la syntaxe de base du langage HLSL. Celle-ci étant très proche des langages C/C++/C#, vous l'assimilerez très vite.

Les variables HLSL

Comme la plupart des langages de programmation, le HLSL permet de déclarer des variables de différents types. Vous pouvez ainsi utiliser des variables de type `int`, `float`, `bool`, etc., et même définir vos propres types grâce aux structures.

Définir la matrice

Pour définir les matrices, deux manières différentes s'offrent à vous en HLSL :

- La première manière consiste à utiliser le mot-clé `floatLxC`, où `L` est le nombre de lignes et `C`, le nombre de colonnes. La ligne de code suivante sert donc à déclarer une matrice de 4 colonnes par 4 lignes.

```
float4x4 myMatrix;
```

- La seconde repose sur le mot-clé `matrix` auquel vous ajoutez le type des données et les dimensions de la matrice :

```
matrix<float,4,4> myMatrix;
```

Accéder au contenu de la matrice

Voici les deux manières d'accéder au contenu de la matrice :

- La première consiste à considérer la matrice comme un tableau de tableaux :

```
float element = myMatrix[0][0];
```

- La deuxième repose sur l'utilisation de la notation pointée « . », comme vous le faites en C# pour accéder aux membres d'une structure ou d'une classe :

```
float element = myMatrix._m23;
```

À la différence du C#, il n'existe pas de types `Vector2`, `Vector3`, etc. Vous utiliserez donc les types `vector` ou `floatX`, où `X` est le nombre de composantes utilisées. Ainsi, vous stockerez la position d'un vertex dans une variable de type `vector` ou `float3`.

Accéder aux composantes d'un vecteur

L'accès aux différentes composantes d'un vecteur se fait de deux manières :

- La première consiste à utiliser les crochets `[]`, comme vous le feriez pour un tableau en C#.

```
float value = myVector[0];
```

- La deuxième consiste à utiliser la notation pointée « . ». Les composantes peuvent être identifiées par deux noms différents. La première série de noms est `rgba` et la seconde, `xyzw`.

  Par exemple, si vous avez un vecteur `color`, vous pouvez accéder à son niveau de vert des deux manières suivantes :

```
float green = color.g;
float green = color.y;
```

Le *swizzling* sert à accéder à plusieurs composantes d'un vecteur en même temps. Dans les deux exemples suivants (qui utilisent les deux méthodes d'accès aux composantes), vous accédez aux composantes rouge et verte d'une couleur :

```
float2 redGreen = { color[0],color[1]};
float2 redGreen = color.rg;
```

Attention à l'erreur !

L'exemple de code suivant est invalide : vous ne pouvez pas mélanger les noms des deux groupes de noms de composantes.

```
float2 redGreen = color.xg;
```

Les structures de contrôle

Les structures de contrôle vous permettont d'effectuer des traitements avancés pour aboutir à des effets encore plus réussis. Vous retrouvez en HLSL une grande partie des structures de contrôle que vous connaissiez déjà en C# et qui s'utilisent de la même manière, à savoir :

- `If (si condition vérifiée, alors);`
- `While (tant que condition vérifiée, faire);`
- `Do (faire tant que condition vérifiée);`
- `For (pour X de I a J, faire).`

Les fonctions fournies pas le langage

Le langage HLSL fournit une très longue liste de fonctions faciles à utiliser dans les shaders. Le tableau 13-1 en cite quelques-unes. La liste complète se trouve dans la documentation de DirectX, à laquelle vous accéderez via la bibliothèque MSDN (voir annexe B).

Tableau 13-1 Exemples de fonctions fournies pas le langage

Fonction	Description
`Cos(x)`	Cosinus de x.
`Sin(x)`	Sinus de x.
`Mul(a,b)`	Multiplication de la matrice a avec la matrice b.
`Pow(x,y)`	Retourne x à la puissance y.

Sémantiques et structures pour formats d'entrée et de sortie

Les sémantiques servent à lier les entrées et sorties d'une fonction. Par exemple, elles servent à lier la sortie de l'application XNA à l'entrée du vertex shader. Dans le processus de rastérisation, d'autres sémantiques lient la sortie du vertex shader avec l'entrée. Pour finir cette courte présentation, signalons également que le pixel shader reçoit et renvoie lui aussi des sémantiques.

Vous déclarez vos propres formats d'entrée et de sortie en utilisant les structures. L'exemple ci-dessous définit le format d'entrée pour un vertex shader.

```
struct VertexInput
{
    float4 Position : POSITION;
    float2 TexCoord : TEXCOORD0;
};
```

Dans la structure précédente, vous constatez que deux sémantiques sont utilisées : une pour la position du vertex et l'autre pour les coordonnées de texture qui lui sont associées. La liste complète des sémantiques disponibles se trouve dans la documentation de DirectX, le tableau 13-2 présente un court extrait de cette liste.

Tableau 13-2 Sémantiques d'entrée pour vertex shader

Sémantique	Description
`COLOR`	Couleur diffuse ou spéculaire.
`NORMAL`	Normale au vertex.
`POSITION`	Position du vertex.
`TEXCOORD`	Coordonnées de texture associées au vertex.

Procédez de la même manière pour les données en sortie du vertex shader (`VertexOutput`), les données en entrée du pixel shader (`PixelInput`) et celles en sortie du pixel shader (`PixelOutput`). Attention, la liste des sémantiques n'est pas la même pour chacune de ces étapes.

Incompatibilité de versions

Certaines sémantiques ne sont pas toujours valables ou ne s'utilisent pas toujours de la même façon selon la version des vertex shaders ou pixel shaders. Vous devrez donc faire attention à la version des shaders que vous utiliserez lors de la compilation.

Écrire un vertex shader

Vous savez à présent tout ce qu'il faut savoir pour écrire un premier vertex shader. Il s'agit d'une simple fonction qui doit retourner un objet de type `VertexOutput`, qui est une structure que vous avez normalement déclarée.

```
VertexOutput vertexShader(VertexInput input)
{
    VertexOutput output;
    WorldViewProjection = mul(mul(World, View), Projection);
    output.Position = mul(Pos, WorldViewProjection);
    output.TexCoord = input.TexCoord;
    return(output);
}
```

Ce premier vertex shader multiplie les différentes matrices (`World`, `View` et `Projection`) pour transformer la position du vertex, puis recopier les coordonnées de textures.

Écrire un pixel shader

Le pixel shader est une fonction qui retourne une couleur, c'est-à-dire un vecteur à 4 dimensions.

```
float4 pixelShader(PixelInput input) : COLOR
{
    return tex2D(TextureSampler, input.TexCoord);
}
```

Le pixel shader ci-dessus retourne la couleur de la texture aux coordonnées de texture de l'objet `PixelInput`.

Finaliser un effet : les techniques et les passes

Comme nous l'avons au chapitre précédent, un fichier d'effets peut contenir plusieurs techniques. Une technique n'est rien de plus qu'un nom et un conteneur de passes. Une passe définit quel vertex shader et quel pixel shader doivent être utilisés. C'est aussi ici que vous choisissez la version des shaders (dans le cas présent, version 1 pour les deux).

```
technique Default
{
    pass P0
    {
      VertexShader = compile vs_1_1 vertexShader();
      PixelShader = compile ps_1_1 pixelShader();
    }
}
```

Vous savez à présent ce qui se cache derrière les termes de vertex shader ou pixel shader. Vous connaissez également la syntaxe du langage HLSL. Bref, vous avez toutes les clés en main pour écrire vos premiers fichiers d'effet.

Créer le fichier d'effet

En reprenant les différents éléments de base d'un fichier d'effets que nous avons vus dans la première partie de ce chapitre, vous reconstituez le fichier suivant. Ce premier fichier d'effet affiche le modèle à l'écran, ni plus, ni moins.

Fichier d'effet basique

```
float4x4 World : WORLD;
float4x4 View;
float4x4 Projection;
float4x4 WorldViewProjection : WORLDVIEWPROJECTION;
texture Texture;
```

```
sampler TextureSampler = sampler_state
{
    texture = <Texture>;
    magfilter = LINEAR;
    minfilter = LINEAR;
    mipfilter = LINEAR;
    AddressU = mirror;
    AddressV = mirror;
};

struct VertexInput
{
    float4 Position : POSITION;
    float2 TexCoord : TEXCOORD0;
};

struct VertexOutput
{
    float4 Position : POSITION;
    float2 TexCoord : TEXCOORD0;
};

VertexOutput vertexShader(VertexInput input)
{
    VertexOutput output;
    WorldViewProjection = mul(mul(World, View), Projection);
    output.Position = mul(input.Position, WorldViewProjection);
    output.TexCoord = input.TexCoord;
    return( output );
}

struct PixelInput
{
    float2 TexCoord : TEXCOORD0;
};

float4 pixelShader(PixelInput input) : COLOR
{
    return tex2D(TextureSampler, input.TexCoord);
}

technique Default
{
    pass P0
    {
      VertexShader = compile vs_1_1 vertexShader();
      PixelShader = compile ps_1_1 pixelShader();
    }
}
```

Voyons comment complexifier les choses… Dans un projet XNA, reprenez la classe `Cube` écrite au chapitre précédent. Modifiez la classe de manière à ne plus utiliser la classe `BasicEffect`, mais la classe `Effect` :

```
Effect effect;
```

Chargez ensuite l'effet grâce au Content Manager, comme si vous chargiez une texture, un modèle, etc.

```
effect = Content.Load<Effect>("FirstEffect");
```

Vous accédez aux différentes variables globales de l'effet via la propriété `Parameters`. Les variables sont ensuite indexées selon leur nom. Renseignez donc les variables `World`, `View`, `Projection` et `Texture`.

```
effect.Parameters["Projection"].SetValue(projection);
effect.Parameters["View"].SetValue(view);
effect.Parameters["World"].SetValue(Matrix.Identity);
effect.Parameters["Texture"].SetValue(texture);
```

Plusieurs techniques dans le même fichier d'effet

Si vous aviez eu plusieurs techniques dans le fichier, vous auriez pu définir la propriété `CurrentTechnique` grâce à la propriété `Techniques`, où les techniques sont indexées par leur nom.

```
effect.CurrentTechnique = effect.Techniques["Default"];
```

Tout est prêt, vous pouvez compiler le projet et le lancer : le cube s'affiche correctement.

Pour compliquer encore un peu les choses, modifions ce premier fichier afin qu'il prenne en charge la lumière ambiante.

Commencez par ajouter une variable globale dans le fichier. Cette variable devra contenir la couleur de la lumière d'ambiance.

```
float4 AmbientColor : COLOR0;
```

Vous n'avez plus qu'à appliquer la lumière d'ambiance dans le pixel shader.

```
float4 pixelShader(PixelInput input) : COLOR
{
    return (tex2D(TextureSampler, input.TexCoord) * AmbientColor);
}
```

Le fichier d'effet est maintenant prêt. Repassez dans le fichier `cube.cs` et définissez la variable `AmbientColor`.

```
effect.Parameters["AmbientColor"].SetValue(0.5f);
```

Une dernière petite modification pour finir ce premier fichier d'effet : nous allons afficher le cube en mode fil de fer (figure 13-3). Pour cela, il suffit d'ajouter la ligne suivante dans la passe :

```
FillMode = Wireframe;
```

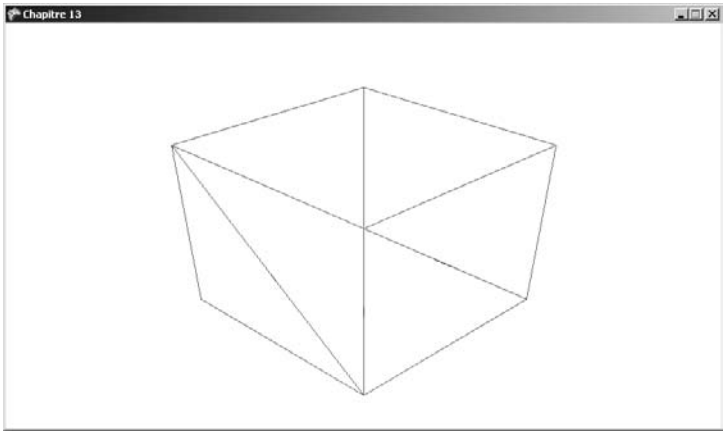


Figure 13-3
Le cube en mode fil de fer

De la même manière, vous pouvez désactiver le *culling*, ce qui fera bien apparaître toutes les faces du cube (figure 13-4).

```
CullMode = none;
```

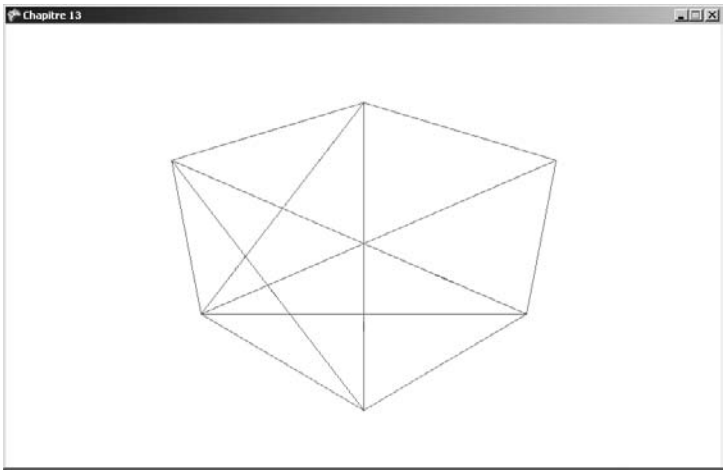


Figure 13-4
Le cube en mode fil de fer sans culling

Faire onduler les objets

Le premier fichier d'effets que vous venez de réaliser n'a rien d'extraordinaire. Vous allez maintenant créer un effet plus intéressant que ceux proposés par la classe `BasicEffect`.

Le but de l'effet suivant est de donner un effet de vague à des vertices : celui-ci oscillera en fonction du temps grâce à la fonction sinus.

1. Commencez par ajouter une nouvelle variable globale au fichier qui servira à stocker le temps.

```
float Timer : TIME;
```

2. Modifiez ensuite le vertex shader pour qu'il change les coordonnées du vertex, qui est quant à lui passé en paramètre. Pour cela, copiez sa position dans un vecteur temporaire.
3. Modifiez la composante `x`, le calcul utilise la fonction sinus qui prend en paramètre la composante `y` et le temps courant.
4. Dernière chose, passez ce jeu de coordonnées mis à jour au vertex de sortie.

```
VertexOutput vertexShader(VertexInput input)
{
    float4 Pos = float4(input.Position.xyz,1);
    Pos.x += sin(Pos.y + Timer);

    VertexOutput output;
    WorldViewProjection = mul(mul(World, View), Projection);
    output.Position = mul(Pos, WorldViewProjection);
    output.TexCoord = input.TexCoord;
    return( output );
}
```

5. Dans le code C#, à chaque passage dans la méthode `Draw()`, passez le nombre de secondes écoulées au total.

```
effect.Parameters["Timer"].SetValue((float)gameTime.TotalGameTime.TotalSeconds);
```

Vous pouvez ensuite essayer le projet et admirer le résultat (figure 13-5).

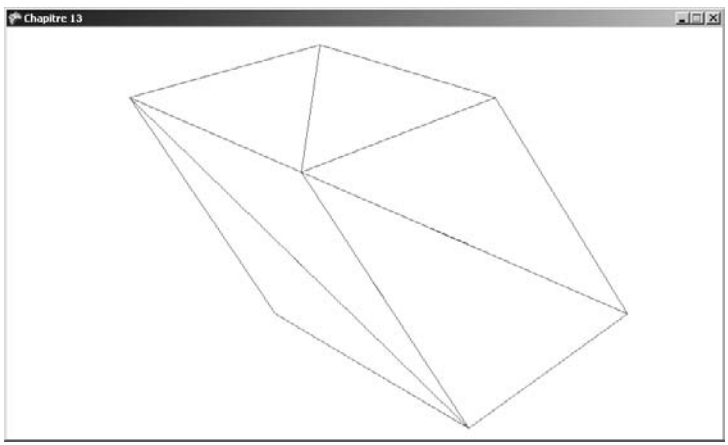


Figure 13-5
Le cube déformé par le nouvel effet

La texture en négatif

À présent, voyons comment modifier le pixel shader de manière à ce que le cube apparaisse en négatif (figure 13-6). Pour cela, vous devez effectuer une simple inversion des couleurs : soustrayez chacune des composantes de la couleur du pixel à 1.

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = 1 - tex2D(TextureSampler, input.TexCoord);
    return( color );
}
```

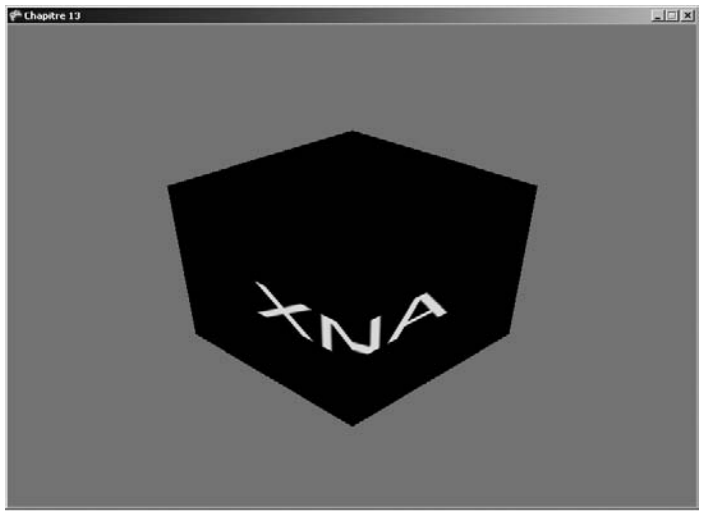


Figure 13-6

Le cube est maintenant affiché en négatif

Jouer avec la netteté d'une texture

Le pixel shader suivant fait varier la netteté de l'image (figure 13-7). Il suffit de modifier la couleur du pixel courant en fonction des pixels voisins.

1. Commencez par déclarer une variable globale dans le fichier d'effets.

```
float SharpAmount;
```

2. Puis, utilisez le pixel shader suivant.

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = tex2D( TextureSampler, input.TexCoord);
    color += tex2D( TextureSampler, input.TexCoord - 0.0001) * SharpAmount;
    color -= tex2D( TextureSampler, input.TexCoord + 0.0001) * SharpAmount;
    return( color );
}
```

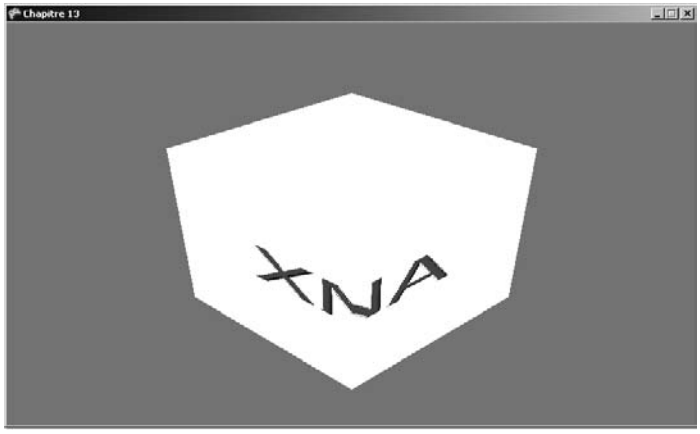


Figure 13-7
Modification de la netteté du cube

3. Pour apprécier pleinement l'impact de cet effet sur le cube, ajoutez une variable au projet C#.

```
float sharpAmount = 0;
```

4. Pour faire varier l'ampleur de l'effet, augmentez la valeur de la variable précédente si le joueur appuie sur la touche `PageUp` et faites-la diminuer s'il presse la touche `PageDown`.

```
public void Update(GameTime gameTime)
{
    if (Keyboard.GetState().IsKeyDown(Keys.PageUp))
    {
        sharpAmount += 1;
        effect.Parameters["SharpAmount"].SetValue(sharpAmount);
    }
    else if (Keyboard.GetState().IsKeyDown(Keys.PageDown))
    {
        sharpAmount -= 1;
        effect.Parameters["SharpAmount"].SetValue(sharpAmount);
    }
}
```

Flouter une texture

Le flou (*blur* en anglais) s'obtient presque de la même manière que l'effet précédent : la couleur du pixel est déterminée en fonction de la couleur des pixels voisins (figure 13-8), sauf que cette fois ci vous n'utiliserez pas de seuil de netteté. Pour vous permettre de bien visualiser l'effet, la distance séparant le pixel courant et les pixels voisins sera dynamique.

```
float Distance;
```

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = tex2D( TextureSampler,
    float2(input.TexCoord.x+Distance, input.TexCoord.y+Distance));
    color += tex2D( TextureSampler,
    float2(input.TexCoord.x-Distance, input.TexCoord.y-Distance));
    color += tex2D( TextureSampler,
    float2(input.TexCoord.x+Distance, input.TexCoord.y-Distance));
    color += tex2D( TextureSampler,
    float2(input.TexCoord.x-Distance, input.TexCoord.y+Distance));
    color = color / 4;
    return( color );
}
```

Comme pour l'effet précédent, le paramètre dynamique sera modifié si le joueur appuie sur la touche `PageUp` ou sur la touche `PageDown`.

```
float distance = 0;

public void Update(GameTime gameTime)
{
    if (Keyboard.GetState().IsKeyDown(Keys.PageUp))
    {
        distance += 0.001f;
        effect.Parameters["Distance"].SetValue(distance);
    }
    else if (Keyboard.GetState().IsKeyDown(Keys.PageDown))
    {
        distance -= 0.001f;
        effect.Parameters["Distance"].SetValue(distance);
    }
}
```

Vous remarquez que sur la figure suivante, la texture appliquée au cube est floutée.

Figure 13-8
Le cube flouté

Modifier les couleurs d'une texture

Les pixels shaders suivants inversent les couleurs d'une texture.

Ici, il renvoie la couleur avec toutes ses composantes dans l'ordre classique.

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = tex2D(TextureSampler, input.TexCoord);
    return (color.rgba);
}
```

Dans le pixel shader suivant, la composante rouge est inversée avec la composante verte.

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = tex2D( TextureSampler, input.TexCoord);
    return( color.grba );
}
```

Il est également possible d'inverser le rouge et le bleu.

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = tex2D( TextureSampler, input.TexCoord);
    return( color.bgra );
}
```

Un effet intéressant consiste à transformer la texture pour que ses couleurs varient uniquement entre le noir et le blanc (on parle également de niveaux de gris). Votre premier réflexe est peut-être de faire la moyenne des trois composantes. Cependant, l'œil humain ne percevant pas de la même manière les trois composantes, vous devez leur appliquer des coefficients. La Commission internationale de l'éclairage conseille d'utiliser les coefficients suivants :

- 0,3 pour le rouge ;
- 0,59 pour le vert ;
- 0,11 pour le bleu.
- Pour convertir une texture en niveau de gris (comme sur la figure 13-9), vous pouvez donc écrire le pixel shader suivant (la fonction `dot` fait le produit scalaire de deux vecteurs).

```
float4 pixelShader(PixelInput input) : COLOR
{
    float4 color = tex2D( TextureSampler, input.TexCoord);
    color.rgb = dot(color.rgb, float3(0.3, 0.59, 0.11));
    return( color );
}
```

Sur la figure suivante, la texture appliquée au cube apparaît en niveaux de gris.

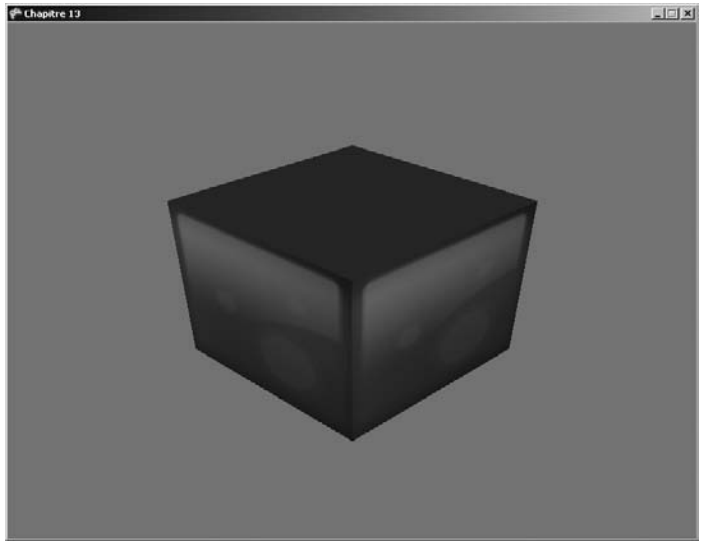


Figure 13-9
Une texture transformée en niveaux de gris

En résumé

Vous connaissez à présent tout ce qu'il faut pour intégrer des premiers effets à vos jeux, à savoir :

- ce qu'est un shader ;
- la syntaxe du langage HLSL ;
- comment écrire un vertex shader et un pixel shader et comment les intégrer aux jeux.

A

Visual C# Express 2008

Si vous débutez en C#, ou si vous développez habituellement sous Linux, vous n'avez peut-être encore jamais eu à utiliser un outil de la suite Visual Studio.

Cette annexe présente l'interface de Visual C# Express 2008, son éditeur de texte, ses fonctionnalités pour vous faire gagner du temps lorsque vous développez et son débogueur.

Différencier solution et projet

Un projet regroupe un ou plusieurs fichiers et vise la création d'un assemblage (appelé *assembly* en anglais). Un assemblage peut être un fichier .exe, mais aussi une DLL (pour *Dynamic Link Libraries*, bibliothèque de liens dynamiques en français).

Les DLL (attention, il ne s'agit pas forcément de fichiers portant l'extension .dll) correspondent aux projets de type Bibliothèque de classes (figure A-1), ou plus particulièrement pour XNA, aux projets de type « Windows/Xbox 360/Zune Game Library » (figure A-2). Une DLL ne comporte pas de point d'entrée comme la fonction `Main`, elle ne contient que du code réutilisable par plusieurs applications.

Dans tout cet ouvrage, nous avons travaillé soit avec XNA ou bien en mode console. Nous avons même utilisé le moteur physique FarseerPhysics, un projet créé par un autre développeur et qui générait une DLL.

Une solution est automatiquement créée lorsque l'on crée un projet. Elle peut bien entendu contenir plusieurs projets, comme lorsque vous écrivez une DLL parallèlement à un projet exécutable et que la bibliothèque devra être utilisée dans ce projet. Dans ce cas, il est conseillé de choisir l'option Créer le répertoire pour la solution, ce qui isole les différents projets de la solution dans des sous-répertoires, rendant le tout beaucoup plus lisible.

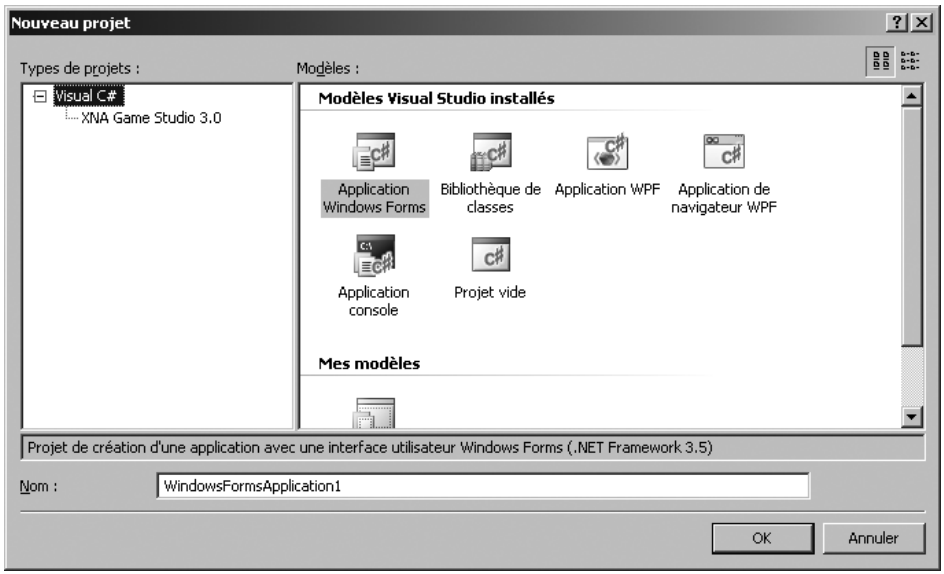


Figure A-1

Création d'une DLL .Net

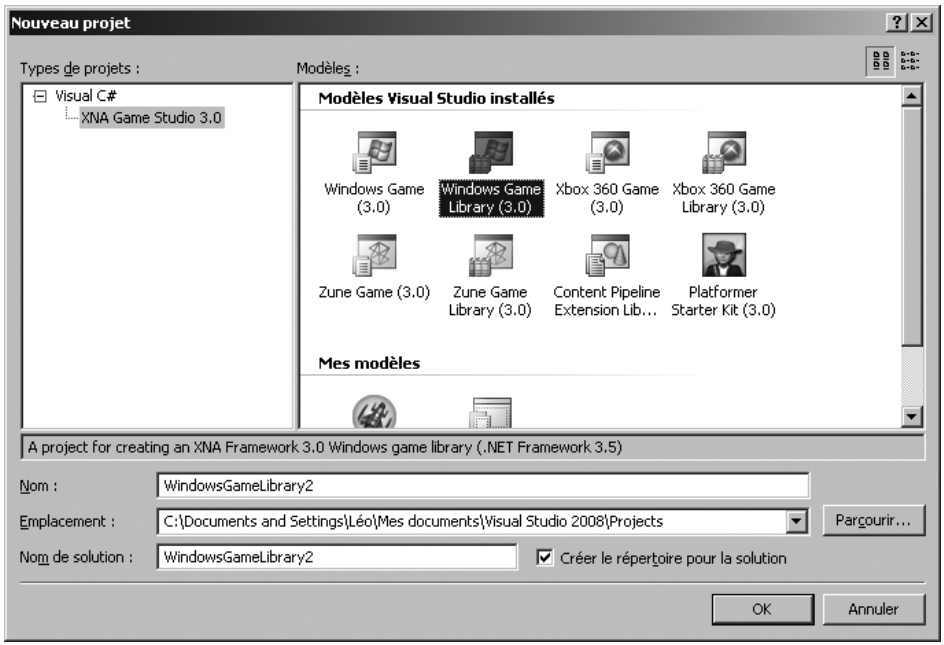


Figure A-2

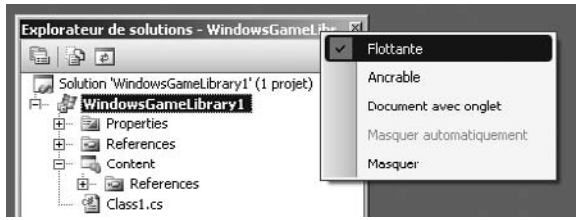
Création d'une DLL XNA

Personnaliser l'interface

L'interface de Visual C# Express 2008 est composée de plusieurs fenêtres (l'explorateur de solutions, l'explorateur de propriétés, etc.). Toutes ces fenêtres peuvent être remaniées comme vous le désirez : vous pouvez les déplacer, les cacher ou encore en afficher d'autres.

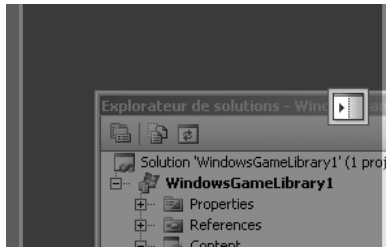
Par exemple, amusez-vous à fermer toutes les fenêtres pour avoir un écran totalement vierge. Allez ensuite dans le menu Affichage et sélection Explorateur de solution ou utilisez le raccourci clavier Ctrl + W, S. Cliquez avec le bouton droit sur la barre de titre de la fenêtre qui s'est ouverte (figure A-3) : vous pouvez définir si elle doit être flottante ou rattachable à un bord de la fenêtre (on parle également de fenêtre « ancrable »).

Figure A-3
Mode d'affichage de la fenêtre



Pour ancrer une fenêtre à un bord de la fenêtre, maintenez le clic sur sa barre de titre vers la marque du bord désiré, elle s'y accroche automatiquement (figure A-4).

Figure A-4
La fonction d'ancrage des fenêtres est très pratique



Vous avez la possibilité de n'afficher les fenêtres que lorsque votre souris passe sur le bord de la fenêtre. Lorsque vous cliquez avec le bouton droit sur la barre de titre de la fenêtre, choisissez l'option Masquer automatiquement.

Si vous disposez d'un écran assez grand et si vous aimez cette manière de travailler, vous pouvez afficher deux fichiers (ou plus) en parallèle, aussi bien verticalement qu'horizontalement (figure A-5).

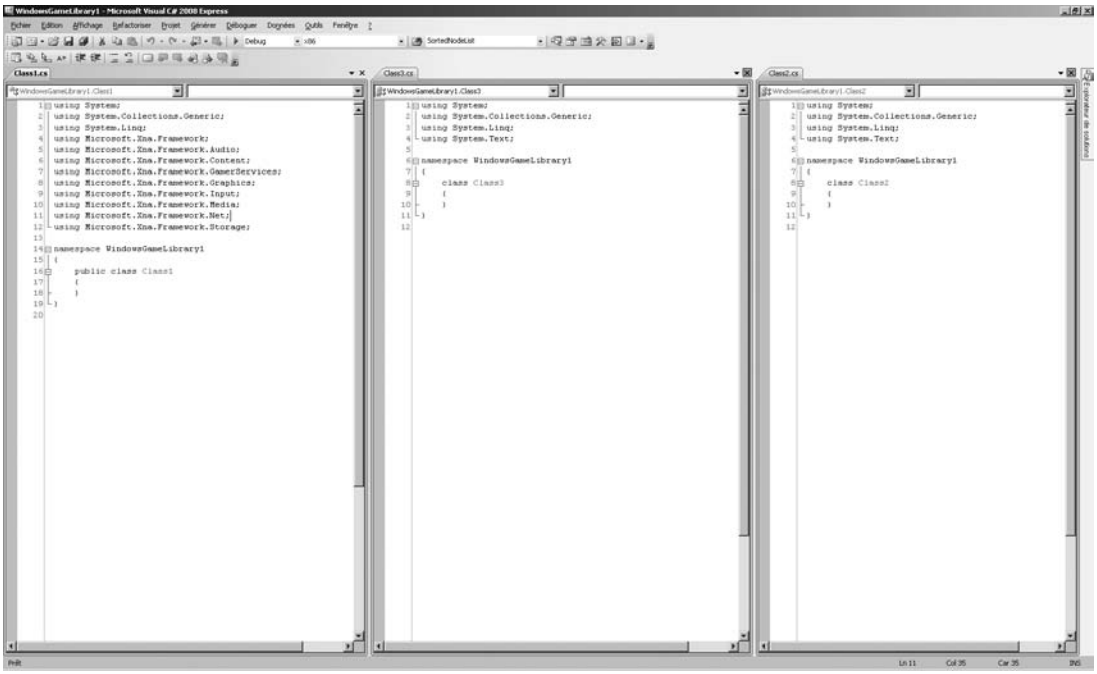


Figure A-5

Avoir plusieurs fichiers ouverts en même temps peut-il améliorer la productivité ?

L'éditeur de texte

L'éditeur de texte de Visual C# Express 2008 est un outil particulièrement paramétrable. La première chose que fait généralement un développeur après avoir installé l'IDE est de choisir d'afficher le numéro des lignes (option qui n'est malheureusement pas activée par défaut).

1. Cliquez sur le menu Outils puis sur Options : vous accédez à la fenêtre des options de l'éditeur de texte.
2. Dans l'arbre à gauche de la fenêtre, choisissez Éditeur de texte.
3. Sélectionnez C#.
4. Enfin, cochez la case Numéros de ligne (figure A-6).

Dans cette fenêtre, vous pouvez également définir la taille et le comportement des tabulations, l'affichage des erreurs ou encore le comportement d'IntelliSense.

> **IntelliSense**
>
> IntelliSense est le composant de Microsoft qui assure la saisie automatique de code et qui, lorsque vous écrivez un programme, vous propose une liste contextuelle d'éléments disponibles (propriétés, méthodes, etc.).

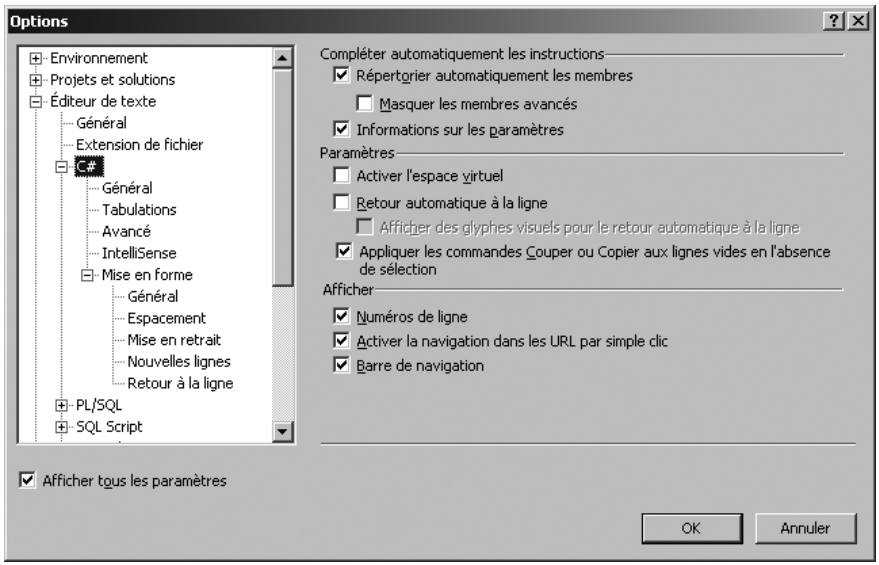


Figure A-6

Les options de l'éditeur de texte

Les directives `#region` et `#endregion` ne sont pas spécifiques à la suite de Microsoft, mais font partie intégrante du langage C#. Elles masquent une partie du code pour améliorer la lisibilité.

```
#region Test
public class Class1
{
}
#endregion
```

Si vous copiez l'exemple ci-dessous dans l'éditeur de texte de Visual C# Express 2008, vous verrez apparaîtrez un petit signe – à côté de l'expression `#region`. Lorsque vous cliquez sur ce signe, le contenu est automatiquement masqué et remplacé par le mot Test.

Dans l'éditeur de texte, vous pouvez également choisir de masquer le contenu d'une fonction, d'une classe, d'un espace de noms, etc., sans utiliser les régions (figure A-7).

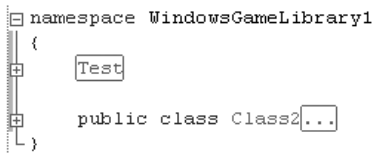


Figure A-7

L'implémentation de la fonction est cachée

Pour terminer avec l'éditeur de texte, lorsque vous ajoutez des instructions dans un fichier, une barre verticale jaune apparaît à côté de ces nouvelles lignes (ou des lignes modifiées). La barre deviendra verte une fois que vous aurez sauvegardé votre fichier. Les barres disparaissent dès que le fichier est fermé (figure A-8).

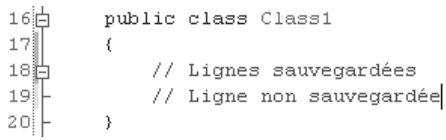


Figure A-8

Changements de couleur de la barre verticale

Les extraits de code

Les extraits de code (appelé *code snippet* en anglais) servent à insérer rapidement du code préconfiguré. L'extrait de code peut être une structure de base du langage (une boucle `for` par exemple) ou un extrait de code plus complet que vous aurez défini.

Pour insérer un *snippet* :

1. Cliquez avec le bouton droit.
2. Sélectionnez Insérez un extrait…, ou bien utilisez le raccourci clavier Ctrl + K, X.
3. Choisissez ensuite l'extrait de code qui vous intéresse.

Sur la figure A-9, vous voyez l'ajout d'un extrait de code pour les propriétés. Vous pouvez vous déplacer entre les différentes zones en couleur grâce à la touche Tab.

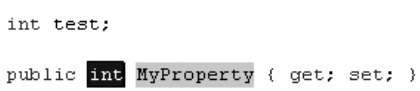


Figure A-9

Ajout d'un code snippet

Avec XNA, vous devez souvent réécrire toute un bloc de lignes concernant les directives `using`… Pour gagner du temps, vous allez donc créer un *code snippet* qui évitera d'avoir à réécrire toutes ces lignes.

L'extrait de code est représenté par un fichier XML qui doit porter l'extension `.snippet`. Le schéma du fichier est disponible sur le site de MSDN à cette adresse : *http://msdn.microsoft.com/en-us/library/ms171418.aspx*.

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
```

```
    <Header>
      <SnippetTypes>
        <SnippetType>Expansion</SnippetType>
      </SnippetTypes>
      <Title>Directives using pour XNA</Title>
      <Shortcut>usingxna</Shortcut>
      <Description>Insere automatiquement les directives using pour XNA.
      ➥</Description>
      <Author>Leonard LABAT</Author>
    </Header>
    <Snippet>
      <Code Language="csharp"><![CDATA[using Microsoft.Xna.Framework;
      using Microsoft.Xna.Framework.Audio;
      using Microsoft.Xna.Framework.Content;
      using Microsoft.Xna.Framework.GamerServices;
      using Microsoft.Xna.Framework.Graphics;
      using Microsoft.Xna.Framework.Input;
      using Microsoft.Xna.Framework.Media;
      using Microsoft.Xna.Framework.Net;
      using Microsoft.Xna.Framework.Storage;]]></Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
```

< ![CDATA[]]>

Le CDATA évite l'exécution d'une partie du fichier XML. Ce qui se situe dans cette balise ne sera donc pas parsé.

Cliquez ensuite sur Outils, puis sur Gestionnaire des extraits de code, et enfin sur le bouton Importer… Allez chercher le fichier `.snippet`. L'extrait de code est maintenant prêt à être utilisé.

Refactoriser

Imaginez la situation suivante : vous vous êtes lancé dans la création d'une application (un jeu, une application Windows, etc.) et vous êtes déjà bien avancé. Seulement, au bout d'un moment, vous vous rendez compte que vous avez fait une erreur de design, ou vous n'êtes pas satisfait d'un résultat : les modifications à apporter au projet risquent donc de vous prendre beaucoup de temps.

La refactorisation (de l'anglais *refactoring*) permet de remanier facilement le code. Les différentes possibilités de refactorisation sont accessibles depuis le menu Refactoriser lorsque vous avez sélectionné un élément du code.

Si vous avez lu ce livre en entier, vous avez déjà rencontré une technique de refactorisation : le renommage. En effet, par défaut dans tout projet XNA, la classe de base s'appelle `Game1`. Si vous avez également créé une variable qui s'appelle `Game1Int`, et que finalement

vous décidez de renommer la classe en `MyGame`, Visual C# Express 2008 procédera de manière intelligente et ne renommera que la classe et ses utilisations.

Il existe plusieurs autres techniques :

- extraire la méthode, ce qui vous permettra de générer automatiquement une méthode à partir d'instructions que vous aurez sélectionnées ;
- extraire l'interface, ce qui vous permettra de créer une interface à partir du nom d'une fonction publique ;
- supprimer les paramètres, ce qui vous permettra de supprimer un argument de la signature d'une fonction, etc.

Déboguer une application

Avant de déboguer un projet, vous devez vous assurer que vous le générez en mode *Debug*. Pour cela, vérifiez que c'est bien Debug qui est sélectionné dans la liste Configuration de solutions (figure A-10).

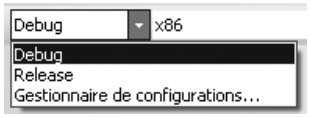


Figure A-10

Sélection de la configuration à appliquer

> **Attention**
>
> Avant de distribuer votre application, recompilez-la en mode *Release*. L'assemblage ainsi généré est optimisé et ne comprend pas de données relatives au débogage.

Pour lancer une application et utiliser le débogueur, cliquez sur le menu Déboguer, puis sur Démarrer le débogueur, ou utilisez le raccourci clavier F5. Pour stopper le débogage, retournez dans le même menu et choisissez cette fois Arrêter le débogage, ou utilisez le raccourci clavier Maj + F5.

Les points d'arrêt (*break points* en anglais) sont un élément essentiel du débogueur : vous en placez sur chaque instruction où le programme doit suspendre son exécution. Ils sont donc pratiques pour vérifier que le programme passe bien par certains endroits.

Pour placer un point d'arrêt sur une ligne, cliquez sur la colonne située à gauche de celle-ci ou utilisez le raccourci F9. Le retrait d'un point d'arrêt se fait de la même manière. Lorsque vous avez ajouté un point d'arrêt sur une ligne, celle-ci s'affiche en rouge dans l'éditeur de texte.

Placez un point d'arrêt quelque part dans le code (par exemple, dans la méthode `Update()` si vous travaillez sur un jeu avec XNA) et lancez le débogueur. Dès que le programme atteint le point d'arrêt, la fenêtre de Visual C# Express 2008 passe au premier plan. La ligne où s'est arrêté le programme est maintenant visible en jaune dans l'éditeur de texte (figure A-11).

Figure A-11

L'exécution s'est suspendue sur le point d'arrêt

Vous pouvez afficher le contenu d'une variable simplement en passant dessus le curseur de la souris. Pour les objets plus complexes, faites dérouler la liste des informations grâce au signe + (figure A-12).

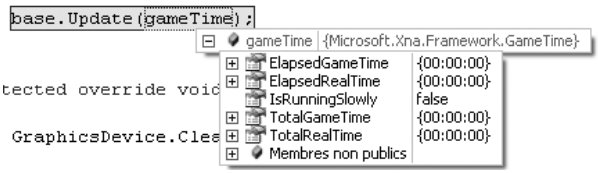


Figure A-12

Visualisation de la valeur d'une variable

Vous pouvez aussi voir le contenu des variables via les fenêtres espions. Pour ajouter un espion sur une variable, faites un clic droit sur la variable et choisissez Ajouter un espion. Son contenu est alors visible dans la fenêtre Espion (figure A-13). La fenêtre Variables locales présente automatiquement les variables définies dans le contexte courant.

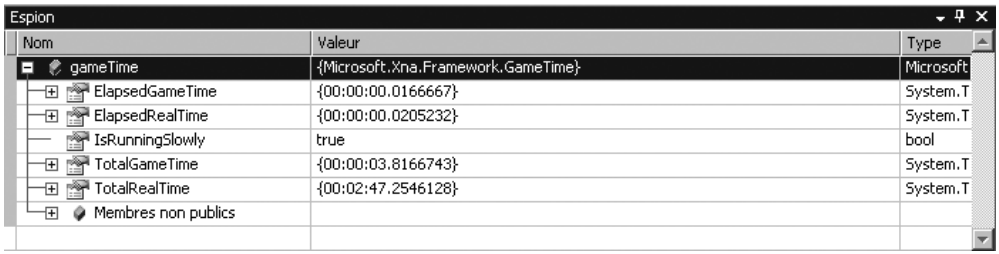


Figure A-13

Utilisation d'un espion

Il ne s'agit ici que d'une présentation rapide du débogueur de Visual C# Express 2008. Il possède bien d'autres fonctionnalités et le framework .Net dispose aussi des classes `Debug` et `Trace` qui pourraient vous être utiles…

Raccourcis clavier utiles

La meilleure façon d'augmenter votre productivité est de maîtriser les différents raccourcis clavier de Visual Studio. Le tableau A-1 énumère les plus couramment utilisés.

Toujours plus de raccourcis

Si vous avez envie d'abandonner complètement votre souris et de devenir un as du clavier, Microsoft a mis à la disposition des développeurs C# un poster qui regroupe tous les raccourcis clavier disponibles dans Visual Studio 2008.

http://www.microsoft.com/downloads/details.aspx?familyid=E5F902A8-5BB5-4CC6-907E-472809749973&displaylang=en

Tableau A-1 Raccourcis clavier les plus couramment utilisés

Raccourci	Description
Ctrl + M,O	Masquer une zone.
Ctrl + M,M	Déplier une zone masquée.
Ctrl + E,C	Commenter la zone sélectionnée.
Ctrl + E,U	Décommenter la zone sélectionnée.
TAB	Insérer un snippet.
Shift + Alt + C	Afficher la fenêtre d'ajout de classe.
Ctrl + J	Afficher IntelliSense.
Ctrl + F4	Fermer le fichier ouvert.
Ctrl + Shift + B	Générer le projet.
F5	Lancer le débogage.

B

Les bienfaits de la documentation

Dans cette annexe, vous allez apprendre où trouver de la documentation et comment générer la documentation de vos projets.

L'incontournable MSDN

La première chose à connaître lorsque vous débutez avec XNA ou, d'une manière plus générale, avec la programmation .Net, c'est l'existence de MSDN (*Microsoft Developer Network*). MSDN, c'est tout simplement :

- Un site web, qui est disponible en français (*http://msdn.microsoft.com/fr-fr/*) et sur lequel vous retrouvez des actualités, des dossiers ou encore des événements à propos des technologies Microsoft (figure B-1).
- Une gigantesque bibliothèque de documentations qui contient des tutoriaux et un guide de référence sur toutes les classes du framework .Net ou XNA (figure B-2). Cette base de connaissances est également disponible en téléchargement lorsque vous installez un outil de la suite Visual Studio.
- Des forums de discussion où vous pourrez trouver de l'aide auprès des experts de la communauté.
- Des blogs où les auteurs postent régulièrement leurs dernières trouvailles ou études, et ce, pas uniquement sur des technologies Microsoft.

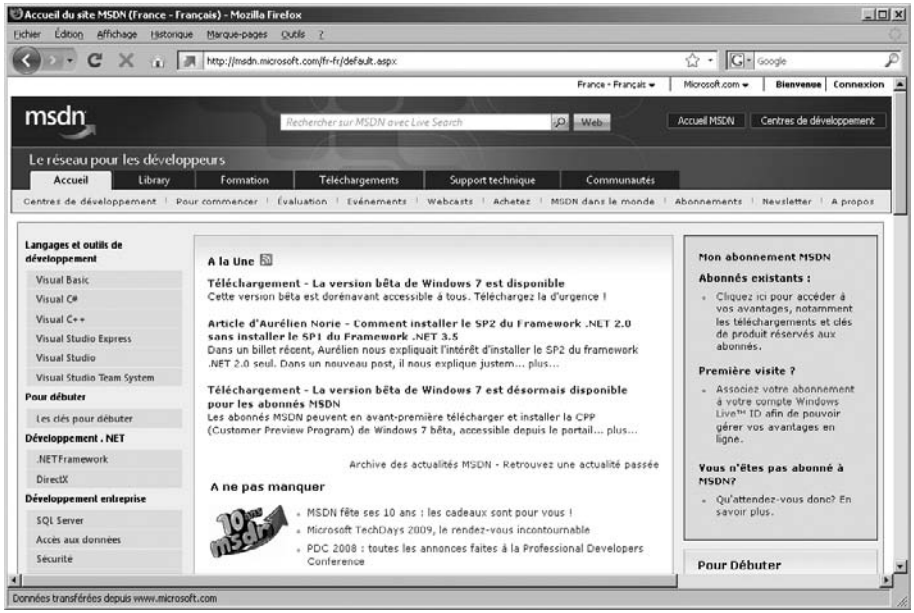


Figure B-1

Le site web français de MSDN

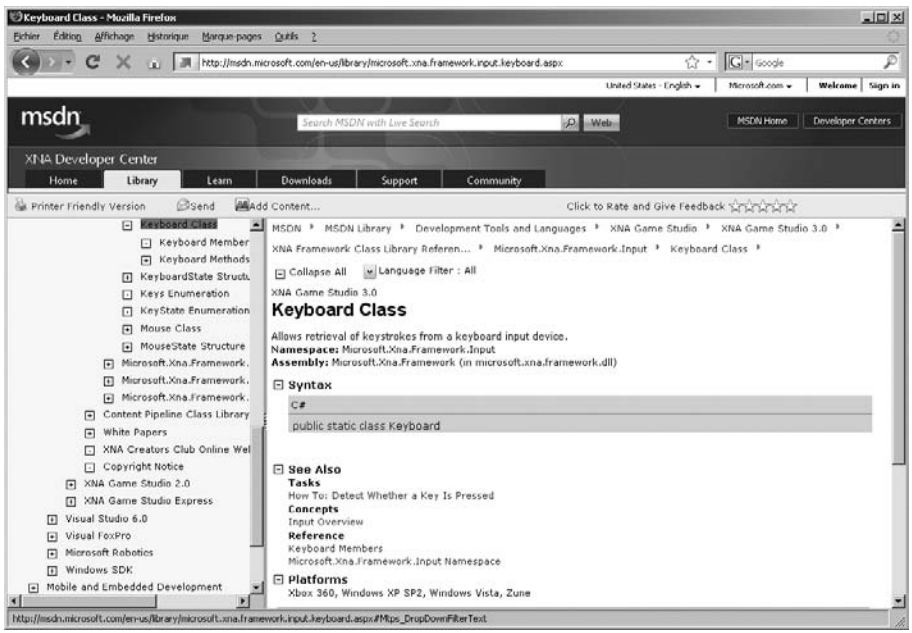


Figure B-2

La documentation complète du framework est disponible

MSDN propose un abonnement qui permet, entre autres, à ceux qui en bénéficient d'accéder aux tout derniers logiciels de Microsoft en avant-première.

Enfin, MSDNAA (MSDN *Academic Alliance*) est un programme qui permet aux universités et écoles adhérentes de proposer à leurs étudiants l'accès à divers logiciels Microsoft. Si vous êtes étudiant et que votre structure enseignante vous le permet, vous avez par exemple accès à un abonnement étudiant à l'XNA Creator Club pour tester vos jeux sur Xbox 360 (vous ne pourrez cependant pas vendre vos jeux).

Ressources sur le Web

La communauté XNA est grande, les blogs et les sites des gourous du framework sont très nombreux et leur contenu est à la hauteur. Dans le tableau B-1, nous avons essayé de dresser une courte liste de ces ressources, à visiter et explorer régulièrement en profondeur.

Tableau B-1 Sites et blogs incontournables

Site	Langue	Description
http://ziggyware.com/	En anglais	Actuellement, presque deux cents articles publiés. Une véritable mine d'or.
http://blogs.msdn.com/shawnhar/	En anglais	Le blog de Shawn Hargreaves.
http://msmvps.com/blogs/valentin/default.aspx	En français	Le blog de Valentin Billotte, MVP.
http://www.c2i.fr/	En français	Le site web de Richard Clark, MVP.
http://blog.emmanueldeloget.com/	En français	Le blog d'Emmanuel Deloget.
http://www.xnadevelopment.com/	En anglais	Le site web de George Clingerman, MVP.
http://leonard-labat.blogspot.com/	En français	Le blog de l'auteur de ce livre.

MVP

Le terme MVP (*Microsoft Most Valuable Professional*) est un titre décerné chaque année par Microsoft à des professionnels des technologies Microsoft indépendants. Les experts qui reçoivent ce titre sont récompensés pour le partage de leurs connaissances auprès d'autres membres des communautés d'utilisateurs.

Figure B-3

Le logo généralement présent sur le blog ou le site d'un MVP

Générer de la documentation

Vous vous en doutez sûrement, apprendre à programmer sans support de documentation est mission impossible. De la même manière, il est difficilement imaginable que vous puissiez apprendre à vous servir d'une bibliothèque sans pouvoir vous appuyer sur une source de documentation. Vous aurez donc certainement envie, vous aussi, d'utiliser les fonctionnalités de génération de documentation de la suite Visual Studio afin de distribuer vos projets accompagnés de cette source d'informations.

Nous l'avons vu au début du livre, il existe un format de commentaire particulier dédié à la documentation, les trois slashs à la suite :

```
///
```

À la suite de ces slashs, vous écrivez une balise XML. Le tableau B-2 répertorie une partie des balises existantes.

Tableau B-2 Les balises de documentation

Balise	Description
`<summary>`	Utilisée pour donner la description complète d'une classe, fonction, propriété ou variable.
`<param>`	Utilisée pour la documentation d'un argument. Vous devez préciser le nom de l'argument. `<param name="arg1">Argument 1</param>`
`<returns>`	Utilisée pour documenter la valeur de retour d'une fonction.
`<value>`	Utilisée pour documenter une propriété.

`<paramref>`	Utilisée pour signaler qu'un mot, utilisé par exemple dans une balise `<summary>`, est un paramètre de la fonction.
	`<summary>Je parle de <paramref name="arg1" /> <summary>`
	`<param name="arg1">Argument 1</param>`

`<c>`	Utilisée pour documenter du code sur une ligne (la différence avec du texte classique se situe au niveau de l'affichage).
`<code>`	Même chose que `<c>`, mais supporte plusieurs lignes.
`<remarks>`	Utilisée pour ajouter des remarques ou commentaires particuliers.
`<exception>`	Utilisée pour préciser le type d'exception que la fonction est susceptible de lever.
`<exemple>`	Utilisée pour donner un exemple d'utilisation d'un élément.
`<see>`	Utilisée pour créer un lien vers un autre élément. `<summary>Avez-vous vu <see cref="method">ma méthode</see> ?`
`<seealso>`	Utilisée pour créer un lien en bas de page de la documentation.

Une fois votre classe correctement documentée, signalez à l'environnement de développement qu'il doit générer un fichier `.xml` contenant la documentation du projet.

1. Cliquez sur le menu Projet, puis sur Propriétés de …
2. Allez ensuite sur l'onglet Générer.
3. Cochez la case Fichier de documentation XML (figure B-4).

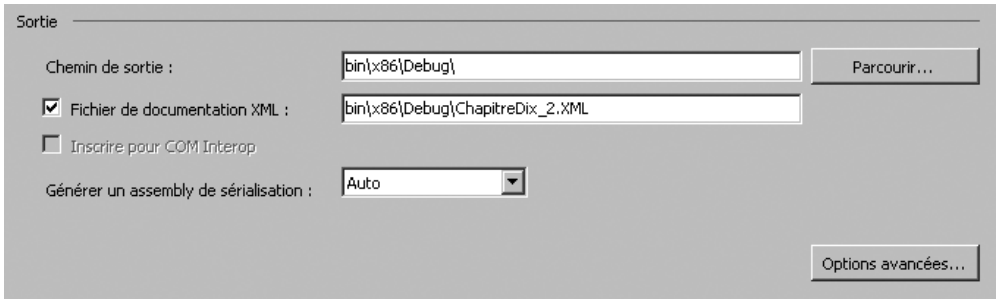


Figure B-4
Activation de la génération de documentation

4. Générez ensuite le projet et rendez-vous dans son répertoire de sortie : un fichier `.xml` a bien été créé.

```
<?xml version="1.0" ?>
- <doc>
- <assembly>
  <name>ChapitreDix_2</name>
  </assembly>
- <members>
- <member name="M:ChapitreDix_2.Program.Main(System.String[])">
  <summary>The main entry point for the application.</summary>
  </member>
  </members>
  </doc>
```

Pour transformer votre documentation en une version plus lisible, essayez par exemple l'utilitaire SandCastle (*http://www.codeplex.com/Sandcastle*) qui génère des fichiers HTML dans le style MSDN (figure B-5).

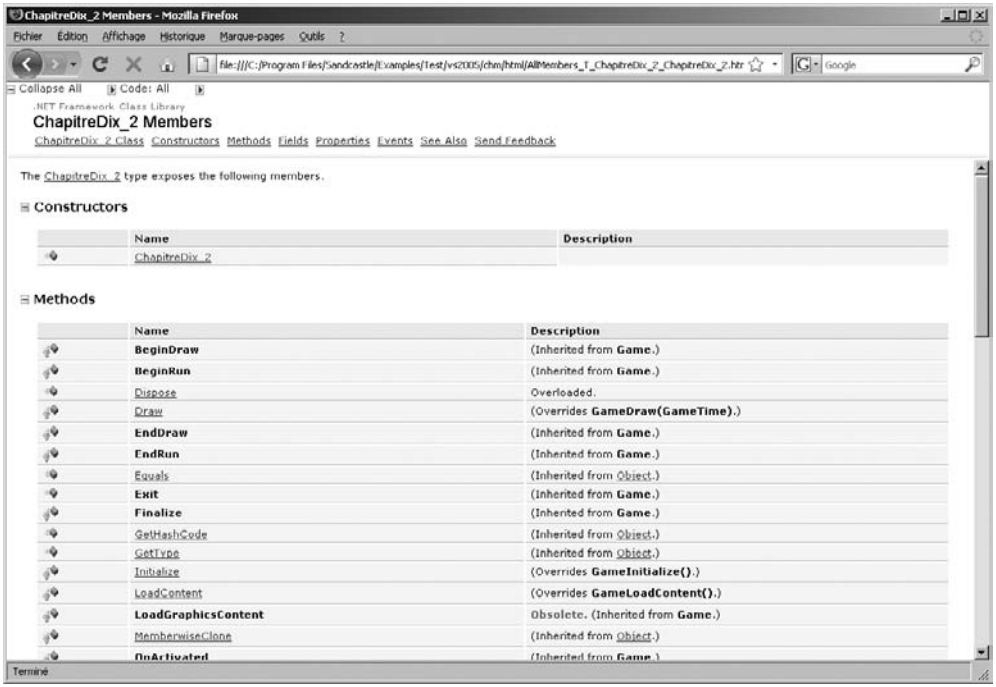


Figure B-5

Documentation générée avec SandCastle

Index

Développement XNA pour la Xbox et le PC

Grâce au tandem Live et XNA, la programmation de jeu vidéo pour PC et Xbox 360 est accessible au plus grand nombre : il n'est plus nécessaire d'investir dans de ruineux outils pour donner libre cours à ses idées de jeux et les réaliser. Cet ouvrage permettra au lecteur de s'approprier le framework XNA 3.0, mais également de comprendre comment s'organise un projet de développement de jeu vidéo.

Accéder aux dernières technologies de développement PC et Xbox 360 avec le framework XNA 3.0

Pour accompagner l'explosion du développement amateur favorisé par la plate-forme de distribution en ligne Live, Microsoft a mis au point le framework XNA pour fournir toutes les briques nécessaires à la création de jeu vidéo. Supports de référence du Live, Xbox 360 et PC sont, grâce à XNA, les deux plates-formes les plus propices pour les studios indépendants, les freelances et les particuliers qui souhaitent faire connaître, voire commercialiser, leurs réalisations.

Un manuel complet pour se lancer dans un projet de création de jeu vidéo

Ce livre accompagne le lecteur, débutant ou non, dans la conduite d'un projet de jeu en C#, qu'il s'agisse de programmer des événements, de créer un environnement sonore, ou de choisir ses moteurs graphique et physique et de les exploiter. L'auteur y détaille les techniques de programmation 2D et 3D. Il explore également les techniques graphiques et sonores avancées (effets, textures, défilement, transformations, animation, éclairage, design sonore, streaming) mais aussi certains algorithmes d'intelligence artificielle, sans oublier l'inclusion du mode multijoueur en réseau ou en écran splitté.

Au sommaire

À qui s'adresse cet ouvrage ?

– Aux étudiants en programmation qui désirent adapter leurs connaissances aux spécificités du développement de jeu pour PC et Xbox.

– Aux studios indépendants et freelances qui souhaitent passer à XNA.

– À l'amateur curieux qui a choisi XNA pour développer son premier jeu.

L. Labat

Passionné par le développement et les jeux vidéo, **Léonard Labat** assure une veille sur les technologies Microsoft en publiant régulièrement sur son blog (http://leonard-labat.blogspot.com/). Il évolue au sein du laboratoire des technologies .Net de SUPINFO (http://www.labo-dotnet.com/).

Conception : Nord Compo

EYROLLES