Enrichir les sprites : textures, défilement, transformation, animation

Dans le chapitre 3, nous avons commencé notre étude des sprites en XNA. Cependant, il ne s'agissait que d'un simple aperçu des possibilités qu'offre le framework.

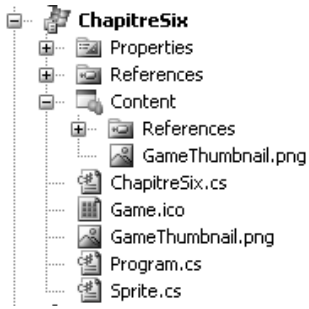
Dans ce chapitre nous allons découvrir les spécificités concernant l'affichage des textures à l'écran et en profiterons pour améliorer notre classe `Sprite`. Enfin, nous découvrirons comment dessiner du texte à l'écran.

Préparation de votre environnement de travail

Avant de se lancer dans le perfectionnement de vos connaissances des images dans XNA, vous devez d'abord préparer votre environnement de travail.

Créez tout d'abord un nouveau projet baptisé « ChapitreSix », renommez la classe `Game1` en `ChapitreSix` et importez la classe `Sprite` du chapitre 3. Pensez également à modifier l'espace de noms. Ajoutez ensuite une image au gestionnaire de contenu. Pour le début de ce chapitre, c'est l'image `GameThumbnail.png`, située à la racine de votre projet, qui sera utilisée. Créez alors un sprite qui utilise cette image et affichez-le. Votre projet devrait maintenant ressembler à celui visible sur la figure 6-1. Récapitulons ci-dessous le code de la classe `ChapitreSix`.

Figure 6-1
Votre projet devrait ressembler à ceci




```
public class ChapitreSix : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    Sprite sprite;

    public ChapitreSix()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }

    protected override void Initialize()
    {
        sprite = new Sprite(100, 100);
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        sprite.LoadContent(Content, "GameThumbnail");
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();
        sprite.Draw(spriteBatch);
        spriteBatch.End();

        base.Draw(gameTime);
    }
}
```

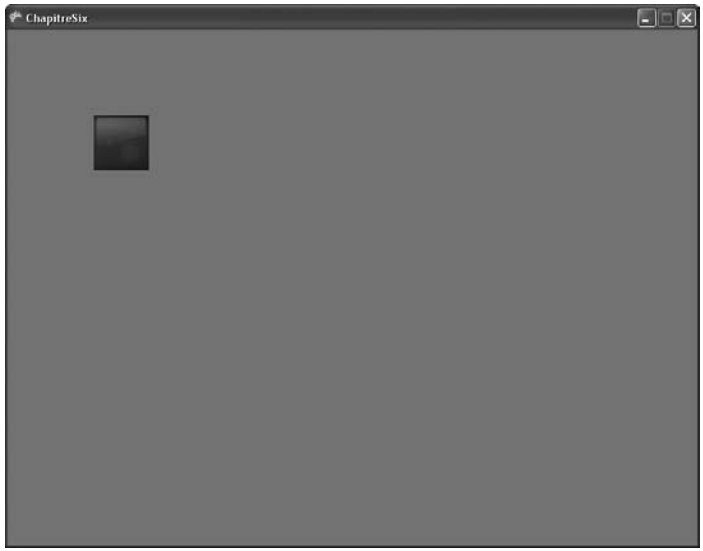
La méthode `Draw()` de la classe `SpriteBatch` dispose de sept surcharges. Cependant, jusqu'à présent nous n'en avons utilisé qu'une seule…

```
spriteBatch.Draw(texture, position, Color.White);
```

Rappelons, pour mémoire, que `texture` correspond à la texture à afficher, `position` à la position du coin supérieur gauche de l'image et `Color.White` à la teinte à appliquer à la texture, celle utilisée ici correspondant à une teinte nulle.

Figure 6-2

Un sprite affiché simplement avec la classe Sprite actuelle



Texturer un objet Rectangle

La surcharge de la méthode `Draw()` que nous savons utiliser pour le moment est classée comme étant la deuxième par Visual Studio. À quoi correspond donc la première ? Faites simplement défiler l'ensemble des surcharges avec les flèches haut et bas de votre clavier. Les paramètres attendus par cette première surcharge sont visibles sur la figure 6-3.

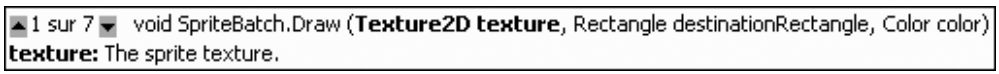


Figure 6-3

Le détail de la première surcharge

Seul le deuxième paramètre diffère de la surcharge que nous connaissons déjà. Là où la méthode attendait un `Vector2` correspondant à la position du coin supérieur gauche de la texture, elle attend à présent un objet de type `Rectangle` nommé `destinationRectangle`.

Nous avons croisé des objets de type `Rectangle` au chapitre précédent. Ils nous ont servi à déterminer s'il y avait collision entre les raquettes et la balle. Un rectangle comprenait en

fait une paire de coordonnées X et Y, ainsi qu'une largeur et une hauteur. Nous pouvons donc facilement déduire le rôle de ce rectangle dans cette surcharge : la texture le remplira, c'est-à-dire qu'elle occupera sa position et qu'elle s'adaptera à sa taille, en se redimensionnant si nécessaire.

Modifier la classe Sprite

Assez parlé, il est temps d'appliquer une texture à notre Rectangle.

1. Ajoutez un champ pour un objet de type `Rectangle` à votre classe `Sprite` ainsi qu'une propriété en lecture et écriture.

```
Rectangle destinationRectangle;
public Rectangle DestinationRectangle
{
    get { return destinationRectangle; }
    set { destinationRectangle = value; }
}
```

2. Surchargez maintenant le constructeur de la classe pour qu'il prenne un objet `Rectangle` en paramètre plutôt qu'une unique paire de coordonnées.

```
public Sprite(Rectangle destinationRectangle)
{
    this.destinationRectangle = destinationRectangle;
}
```

3. Et enfin, modifiez la méthode `Draw()` pour qu'elle utilise la première surcharge.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, destinationRectangle, Color.White);
}
```

4. Retournez dans la classe `ChapitreSix` et utilisez maintenant votre nouveau constructeur, puis exécutez le programme.

```
protected override void Initialize()
{
    sprite = new Sprite(new Rectangle(100, 100, 300, 300));
    base.Initialize();
}
```

Rien ne vous oblige à conserver les proportions de votre sprite ! L'exemple suivant redimensionne le sprite en modifiant sa hauteur.

```
protected override void Initialize()
{
    sprite = new Sprite(new Rectangle(100, 100, 300, 100));
    base.Initialize();
}
```

Figure 6-4
La taille de la texture est facilement modifiable

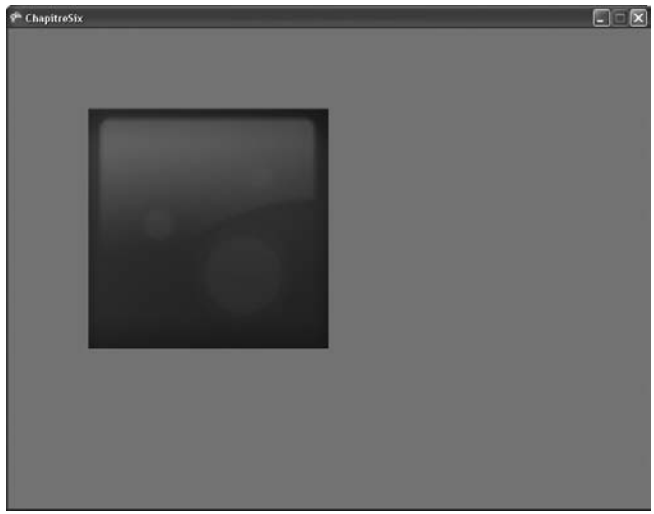
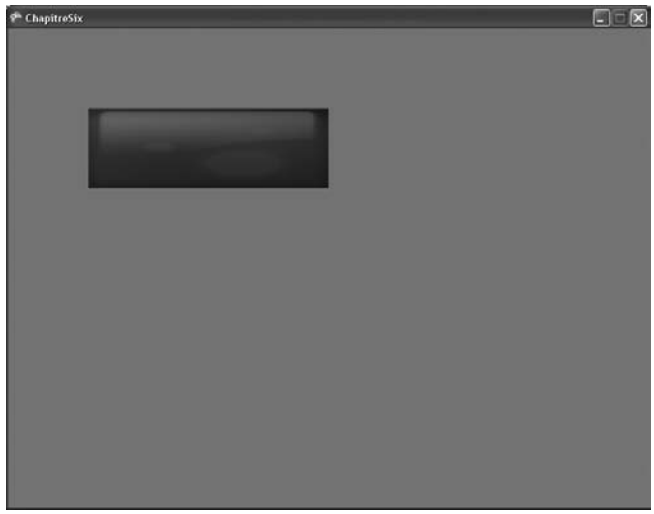


Figure 6-5
La texture est maintenant aplatie



Maintenant que nous avons fait le tour de la première surcharge, passons à la troisième. Celle-ci prend un nouveau paramètre en compte, de type `Rectangle?` et s'appelle `sourceRectangle`. Le point d'interrogation signifie que le paramètre peut être nul.

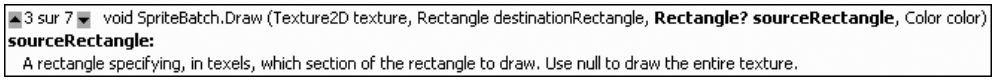


Figure 6-6
Le nouveau paramètre de la troisième surcharge

Comme vous pouvez le voir en anglais dans la description visible sur la figure 6-6, ce rectangle permet de sélectionner la portion de la texture qui devra être dessinée. Si vous décidez de passer `null` plutôt qu'un objet `Rectangle`, c'est toute la texture qui sera dessinée.

Testons cette nouvelle fonctionnalité :

1. Commencez par ajouter un nouveau champ à la classe. Il devra être de type `Rectangle?` et s'appeler `sourceRectangle`. Pensez également à ajouter des propriétés en lecture et en écriture pour cette nouvelle variable.

```
Rectangle? sourceRectangle = null;
public Rectangle? SourceRectangle
{
    get { return sourceRectangle; }
    set { sourceRectangle = value; }
}
```

2. Comme d'habitude, ajoutez maintenant une nouvelle surcharge du constructeur de la classe `Sprite`.

```
public Sprite(Rectangle destinationRectangle, Rectangle? sourceRectangle)
{
    this.destinationRectangle = destinationRectangle;
    this.sourceRectangle = sourceRectangle;
}
```

3. Et enfin, modifiez la méthode `Draw()`.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, destinationRectangle, sourceRectangle, Color.White);
}
```

4. La classe `Sprite` étant prête, il ne reste plus qu'à modifier la classe `ChapitreSix` qui l'utilise. Pour commencer, contentez vous de passer `null` comme nouveau paramètre.

```
protected override void Initialize()
{
    sprite = new Sprite(new Rectangle(100, 100, 64, 64), null);
    base.Initialize();
}
```

À présent, essayez d'afficher seulement le quart inférieur droit de la texture et de le redimensionner pour qu'il apparaisse dans un rectangle de 64 par 64 pixels.

```
protected override void Initialize()
{
    sprite = new Sprite(new Rectangle(100, 100, 64, 64), new Rectangle(32, 32, 32,
    ➥32));
    base.Initialize();
}
```

… ou encore la moitié haute du sprite sans la redimensionner.

```
protected override void Initialize()
{
    sprite = new Sprite(new Rectangle(100, 100, 64, 32), new Rectangle(0, 0, 64,
    ➥32));
    base.Initialize();
}
```

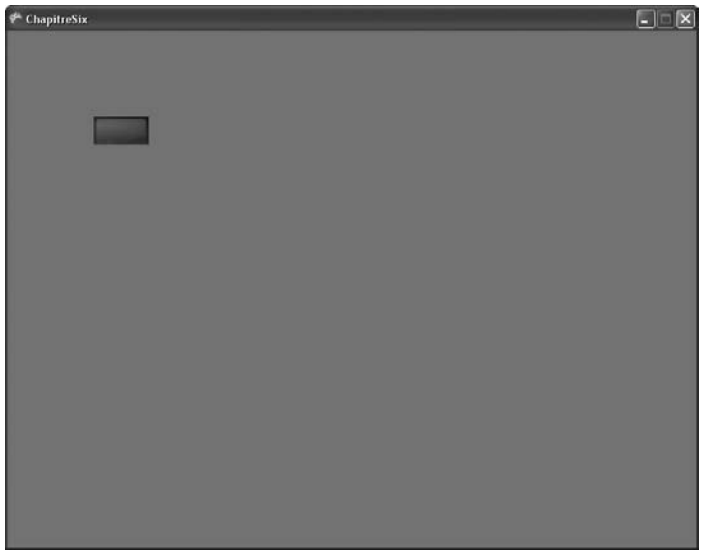


Figure 6-7

Ici, seule la moitié haute de la texture est affichée

Passons à la quatrième surcharge. Comme vous pouvez le constater sur la figure 6-8, elle est très similaire à la troisième surcharge, à la différence qu'elle prend un couple de coordonnées (`Vector2`) comme position de la texture à l'écran plutôt qu'un objet de type `Rectangle`. Adaptez votre classe pour qu'elle utilise cette surcharge plutôt que la troisième. La fonctionnalité de redimensionnement de la texture n'est pas disponible avec cette surcharge, cependant vous la retrouverez dans la cinquième surcharge ou sous la forme de changement d'échelle.

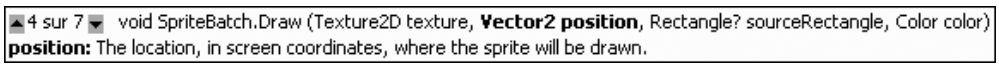


Figure 6-8

Le détail de la quatrième surcharge

La classe Sprite qui prend en charge une position de type Vector2

```
class Sprite
{
    Vector2 position;
    public Vector2 Position
    {
        get { return position; }
        set { position = value; }
    }

    Texture2D texture;
    public Texture2D Texture
    {
        get { return texture; }
    }

    Rectangle? sourceRectangle = null;
    public Rectangle? SourceRectangle
    {
        get { return sourceRectangle; }
        set { sourceRectangle = value; }
    }

    public Sprite(Vector2 position)
    {
        this.position = position;
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
    }

    public Sprite(float x, float y, Rectangle? sourceRectangle)
    {
        position = new Vector2(x, y);
        this.sourceRectangle = sourceRectangle;
    }

    public void LoadContent(ContentManager content, string assetName)
    {
        texture = content.Load<Texture2D>(assetName);
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, position, sourceRectangle, Color.White);
    }
}
```

Cette nouvelle fonction de votre classe s'utilisera de la manière suivante.

```
protected override void Initialize()
{
    sprite = new Sprite(new Vector2(100, 100), new Rectangle(0, 0, 64, 32));
    base.Initialize();
}
```

Pour l'instant, conservez cette version de la classe. Nous allons continuer de la faire évoluer au fur et à mesure du chapitre.

Faire défiler le décor : le scrolling

Nous allons maintenant découvrir une première utilisation de la sélection d'une portion de texture. Il s'agit du *scrolling*. Derrière ce mot anglais se cache tout simplement la notion de défilement de l'écran dans un jeu vidéo en deux dimensions. Cette technique est utile lorsque l'intégralité du niveau ne peut être affichée sur un seul écran ; l'arrière-plan se déplace alors suivant les mouvements du joueur.

Figure 6-9

Le jeu Super Tux utilise le scrolling

1. Pour commencer, rendez-vous sur le site du développeur et MVP (Microsoft *Most Valuable Professional)* George Clingerman : *http://www.xnadevelopment.com*. Naviguez ensuite jusqu'à la catégorie sprites et récupérez l'une des images d'arrière-plan qu'il met gracieusement à la disposition des internautes. Vous pouvez aussi bien utiliser une image de votre cru si vous le désirez.

Figure 6-10
L'arrière-plan qui va être utilisé

2. Ajoutez à votre projet les classes et interfaces nécessaires pour facilement récupérer les entrées clavier de l'utilisateur, modifiez les espaces de noms puis initialisez-les.

```
ServiceHelper.Game = this;
Components.Add(new KeyboardService(this));
```

L'arrière-plan que nous avons retenu pour illustrer le principe du défilement a une résolution de 400 par 300 pixels. Redimensionnez donc la fenêtre de manière à ce qu'elle n'affiche pas l'intégralité de l'image dans un seul écran. Utilisez par exemple une résolution de 200 par 300 pixels, il s'agira donc de faire un scrolling horizontal.

```
public ChapitreSix()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    ServiceHelper.Game = this;
    Components.Add(new KeyboardService(this));
    graphics.PreferredBackBufferWidth = 200;
    graphics.PreferredBackBufferHeight = 300;
}
```

4. Lors de l'initialisation de votre sprite, utilisez un rectangle qui commence aux coordonnées (0, 0) et qui correspond à la taille de l'écran.

```
sprite = new Sprite(new Vector2(0, 0), new Rectangle(0, 0, 200, 300));
```

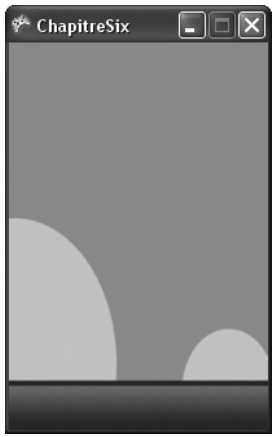
Enfin, dans la méthode `Update()`, modifiez la coordonnée X de la position du rectangle source. Augmentez-la si l'utilisateur appuie sur la flèche droite, diminuez-la dans le cas contraire. Remarquez que cette coordonnée est en dehors de la taille réelle de la texture, son extrémité (gauche ou droite selon les cas) est répétée à l'infini. Dans le cas présent, il s'agit donc ici de la couleur bleu ciel, la même que l'arrière-plan.

Une classe de test de scrolling

```
public class ChapitreSix : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
```

```
    SpriteBatch spriteBatch;
    Sprite sprite;

    public ChapitreSix()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        ServiceHelper.Game = this;
        Components.Add(new KeyboardService(this));
        graphics.PreferredBackBufferWidth = 200;
        graphics.PreferredBackBufferHeight = 300;
    }

    protected override void Initialize()
    {
        sprite = new Sprite(new Vector2(0, 0), new Rectangle(0, 0, 200, 300));
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        sprite.LoadContent(Content, "figure_6_11");
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Right))
            sprite.SourceRectangle = new Rectangle(sprite.SourceRectangle.Value.X
            ➥+ gameTime.ElapsedGameTime.Milliseconds / 10, 0, 200, 300);
        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Left))
            sprite.SourceRectangle = new Rectangle(sprite.SourceRectangle.Value.X
            ➥- gameTime.ElapsedGameTime.Milliseconds / 10, 0, 200, 300);

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();
        sprite.Draw(spriteBatch);
        spriteBatch.End();

        base.Draw(gameTime);
    }
}
```

Figure 6-11
Le début d'un clone de Super Mario



Créer des animations avec les sprites sheets

Avez-vous déjà entendu parler de *sprite sheet* (feuille de sprites) ? Peut-être pas, mais vous en avez sûrement déjà vu en fonctionnement. Il s'agit en fait d'une seule image sur laquelle figure toute une déclinaison d'un ou plusieurs sprites à différents instants t. L'ensemble des images d'un même sprite peut donc composer une animation.

Il est alors facile d'imaginer l'utilité des rectangles source dans ce cas de figure : déplacer le rectangle d'un sprite à l'autre dès qu'un intervalle de temps est révolu.

Commencez par récupérer une planche de sprites ou bien, si vous avez des talents de graphiste, faites-en une vous-même. Dans cette partie du chapitre, nous utiliserons à nouveau une création de George Clingerman (*http://www.xnadevelopment.com*).

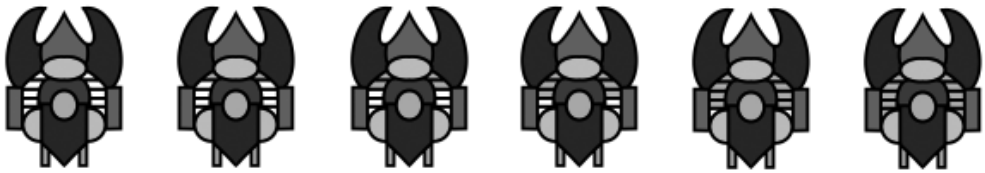


Figure 6-12
Une planche de sprite

Notre planche de sprite fait 600 par 100 pixels et comporte six états d'un sprite. Nos rectangles source devront donc être des carrés de 100 pixels de côté.

1. Dans la classe `Sprite`, créez deux champs : l'un nommé `index` de type `float` et l'autre nommé `maxIndex` de type `int`. Par défaut, ces deux variables doivent être initialisées à 0.

```
float index = 0;
int maxIndex = 0;
```

2. Surchargez la méthode `LoadContent()` pour permettre de définir un index maximum dans le cas d'une planche de sprite.

```
public void LoadContent(ContentManager content, string assetName, int maxIndex)
{
    texture = content.Load<Texture2D>(assetName);
    this.maxIndex = maxIndex;
}
```

3. Pour terminer, le traitement de l'animation va se faire dans la méthode `Update()`. À chaque fois qu'elle est appelée, stockez le nombre de millisecondes écoulées depuis le dernier appel dans la variable `index`. Si `index` à dépassé l'index maximum, fixez-le à 0.

4. Enfin, modifiez l'objet `sourceRectangle` en changeant la position X par la valeur entière de la variable `index`. Voici le nouveau code complet de la classe `Sprite` :

La classe Sprite avec la possibilité d'utiliser une feuille de sprites

```
class Sprite
{
    Vector2 position;
    public Vector2 Position
    {
        get { return position; }
        set { position = value; }
    }

    Texture2D texture;
    public Texture2D Texture
    {
        get { return texture; }
    }

    Rectangle? sourceRectangle = null;
    public Rectangle? SourceRectangle
    {
        get { return sourceRectangle; }
        set { sourceRectangle = value; }
    }

    float index = 0;
    int maxIndex = 0;

    public Sprite(Vector2 position)
    {
        this.position = position;
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle)
    {
        this.position = position;
```

```
        this.sourceRectangle = sourceRectangle;
    }

    public Sprite(float x, float y, Rectangle? sourceRectangle)
    {
        position = new Vector2(x, y);
        this.sourceRectangle = sourceRectangle;
    }

    public void LoadContent(ContentManager content, string assetName)
    {
        texture = content.Load<Texture2D>(assetName);
    }

    public void LoadContent(ContentManager content, string assetName, int maxIndex)
    {
        texture = content.Load<Texture2D>(assetName);
        this.maxIndex = maxIndex;
    }

    public void Update(GameTime gameTime)
    {
        if (maxIndex != 0)
        {
            index += gameTime.ElapsedGameTime.Milliseconds * 0.001f;

            if (index > maxIndex)
                index = 0;

            sourceRectangle = new Rectangle((int)index * sourceRectangle.Value.X,
            ➥sourceRectangle.Value.Y, sourceRectangle.Value.Width,
            ➥sourceRectangle.Value.Height);
        }
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, position, sourceRectangle, Color.White);
    }
}
```

5. De retour dans la classe `ChapitreSix`, il vous reste à modifier la taille du rectangle dans l'initialisation du sprite, changer l'appel à la méthode `LoadContent()` et ajouter l'appel de la méthode `Update()`.

La classe de test des feuilles de sprites

```
public class ChapitreSix : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
```

```
    Sprite sprite;

    public ChapitreSix()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        ServiceHelper.Game = this;
        Components.Add(new KeyboardService(this));
        graphics.PreferredBackBufferWidth = 100;
        graphics.PreferredBackBufferHeight= 100;
    }

    protected override void Initialize()
    {
        sprite = new Sprite(new Vector2(0, 0), new Rectangle(0, 0, 100, 100));
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        sprite.LoadContent(Content, "figure_6_13", 6);
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        sprite.Update(gameTime);

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);

        spriteBatch.Begin();
        sprite.Draw(spriteBatch);
        spriteBatch.End();

        base.Draw(gameTime);
    }
}
```

Varier la teinte des textures

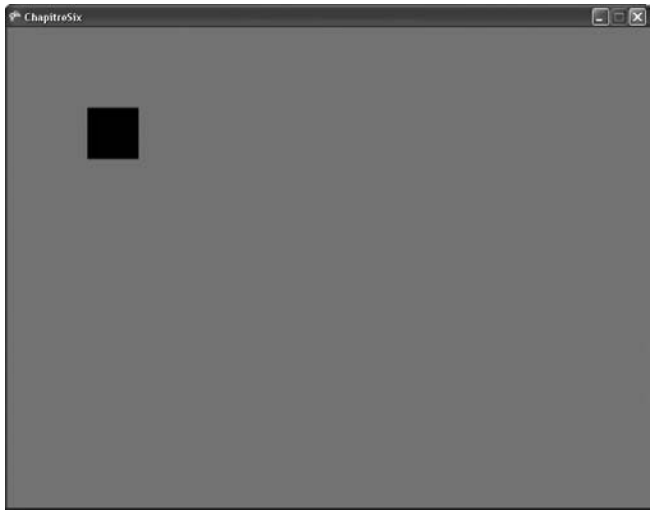
Avant d'étudier la prochaine surcharge de la méthode `Draw()`, il est bon de revenir sur le paramètre qui définit la teinte des textures. Jusqu'à présent, nous avons laissé ce paramètre à la valeur `Color.White`. Il est temps d'essayer de faire varier cette valeur et d'observer les résultats.

Reprenez la classe `Sprite` telle qu'elle était à la fin de la section « Texturer un objet Rectangle », puis modifiez sa méthode `Draw()` afin de régler la teinte sur `Color.Black` et exécutez l'application. Le résultat est visible sur la figure 6-13.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, position, sourceRectangle, Color.Black);
}
```

Figure 6-13

Avec une teinte noire, la texture semble… bien noire



Essayez la même chose mais cette fois-ci avec une teinte rouge.

```
public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, position, sourceRectangle, Color.Red);
}
```

Vous pouvez créer des effets intéressants sur votre sprite grâce aux teintes. Pour cela, ajoutez deux champs de type `Color` à votre classe `Sprite`. Le premier s'appelle `color` et contient la teinte courante du sprite. Le second s'appelle `nextColor` et, comme son nom l'indique, indique la prochaine teinte que votre sprite utilisera. L'effet que nous allons réaliser ici devra faire varier doucement la teinte du sprite d'une couleur à l'autre. Ainsi, le sprite s'illuminera puis s'assombrira.

```
Color color = Color.Gray;
Color nextColor = Color.White;
```

L'évolution de la couleur de la teinte a lieu dans la méthode `Update()`. Chaque couleur dispose de quatre composantes : rouge, vert, bleu et alpha. La composante alpha correspond à la transparence. À chaque appel de la méthode, il faut faire évoluer chaque composante (sauf la transparence qui n'a pas d'importance ici) vers la couleur objectif : soit en l'augmentant, soit en la diminuant. Lorsque la couleur courante correspond à la couleur objectif, on modifie cette dernière. N'oubliez pas de remplacer la couleur dans l'appel à la méthode `Draw()` de l'objet `spriteBatch`.

La classe Sprite prend maintenant en charge la fonction de variation de sa teinte

```
class Sprite
{
    Vector2 position;
    public Vector2 Position
    {
        get { return position; }
        set { position = value; }
    }

    Texture2D texture;
    public Texture2D Texture
    {
        get { return texture; }
    }

    Rectangle? sourceRectangle = null;
    public Rectangle? SourceRectangle
    {
        get { return sourceRectangle; }
        set { sourceRectangle = value; }
    }

    Color color = Color.Gray;
    Color nextColor = Color.White;

    public Sprite(Vector2 position, Rectangle? sourceRectangle)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
    }

    public Sprite(float x, float y, Rectangle? sourceRectangle)
    {
        position = new Vector2(x, y);
        this.sourceRectangle = sourceRectangle;
    }

    public void LoadContent(ContentManager content, string assetName)
    {
        texture = content.Load<Texture2D>(assetName);
```

```
    }

    public void Update()
    {
        if (color.R < nextColor.R)
            color.R++;
        else if(color.R > nextColor.R)
            color.R--;

        if (color.G < nextColor.G)
            color.G++;
        else if (color.G > nextColor.G)
            color.G--;

        if (color.B < nextColor.B)
            color.B++;
        else if (color.B > nextColor.B)
            color.B--;

        if (color == Color.White)
            nextColor = Color.Gray;
        else if (color == Color.Gray)
            nextColor = Color.White;
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, position, sourceRectangle, color);
    }
}
```

Dans la classe `ChapitreSix`, n'oubliez pas d'ajouter l'appel à la méthode `Update()` de votre objet `sprite`.

```
protected override void Update(GameTime gameTime)
{
    sprite.Update();
    base.Update(gameTime);
}
```

Les possibilités d'application de cet effet sont vastes : vous pouvez l'utiliser pour les boutons de votre GUI, les menus de votre jeu ou encore un système de cycle jour/nuit dans un jeu en deux dimensions.

Avant de continuer notre découverte des possibilités d'affichage des textures avec XNA, adaptez votre classe pour qu'elle soit la plus générale possible : enlevez donc la variable `nextColor`, ajoutez une propriété pour la variable `color` et surchargez le constructeur.

La classe Sprite plus générique que la précédente

```
class Sprite
{
    Vector2 position;
```

```
    public Vector2 Position
    {
        get { return position; }
        set { position = value; }
    }

    Texture2D texture;
    public Texture2D Texture
    {
        get { return texture; }
    }

    Rectangle? sourceRectangle = null;
    public Rectangle? SourceRectangle
    {
        get { return sourceRectangle; }
        set { sourceRectangle = value; }
    }

    Color color = Color.White;
    public Color Color
    {
        get { return color; }
        set { color = value; }
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
        this.color = color;
    }

    public void LoadContent(ContentManager content, string assetName)
    {
        texture = content.Load<Texture2D>(assetName);
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, position, sourceRectangle, color);
    }
}
```

Opérer des transformations sur un sprite

Les trois prochaines surcharges se ressemblent fortement. La seule différence entre elles concerne la mise à l'échelle. Dans la cinquième, cela se fait via un rectangle de destination comme au début de ce chapitre, dans la sixième, cela se fait grâce à un coefficient de type `float` et dans la dernière, cela se fait grâce à un `Vector2`. Le résultat est le même dans les deux cas, c'est donc à vous de choisir celle qui convient le mieux à vos besoins.

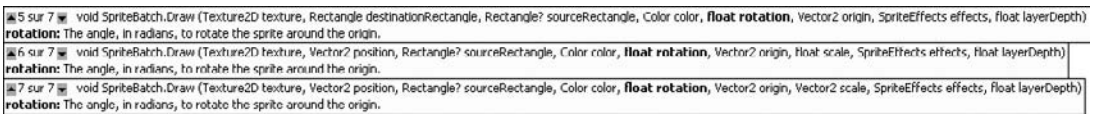


Figure 6-14

Le détail des paramètres attendus par les trois dernières surcharges

Rotation

Tout d'abord la rotation. Celle-ci doit être exprimée en radian et s'effectue autour du point d'origine.

Le point d'origine est le prochain paramètre à étudier. Pour que l'origine soit le coin supérieur gauche de l'écran, le couple de valeurs doit être 0 et 0.

> **Bon à savoir**
>
> Vous n'êtes pas obligé de créer un vecteur pour certaines valeurs particulières. En effet, il existe deux valeurs prédéfinies, l'une ayant ses composantes à 0 et l'autre ayant ses composantes à 1.

Le prochain paramètre attendu par la surcharge concerne l'échelle du sprite. Si vous lui passez une valeur en inférieure à 1, il sera rétréci, dans le cas d'une valeur égale à 1, sa taille ne sera pas modifiée et enfin, dans le cas d'une valeur supérieure à 1, il sera agrandi.

Vient ensuite le tour de l'énumération `SpriteEffects`. Celle-ci peut prendre trois valeurs : `None`, `FlipVertically` et `FlipHorizontally`. `None` ne changera rien au rendu de votre image, `FlipVertically` inversera l'image en la faisant tourner de 180° autour de l'axe horizontal et, enfin, `FlipHorizontally` inversera l'image en la faisant tourner de 180° selon l'axe vertical.

Le dernier paramètre, `layerDepth`, est un nombre réel compris entre 0 et 1 déterminant l'ordre de l'affichage des différents éléments. Une texture qui se voit attribuer un nombre proche de 0 sera dessinée par-dessus une texture ayant un `layerDepth` proche de 1. Cependant, pour que ce paramètre rentre vraiment en compte, vous devez modifier un autre élément lors de l'appel à la méthode `Begin()` de l'objet `spriteBatch`.

Testons à présent ces fonctionnalités une par une. La ligne suivante utilise la septième surcharge de la fonction et dessine un sprite sans aucune modification particulière.

```
spriteBatch.Draw(texture, position, sourceRectangle, Color.White, 0, Vector2.Zero,
➥Vector2.One, SpriteEffects.None, 0);
```

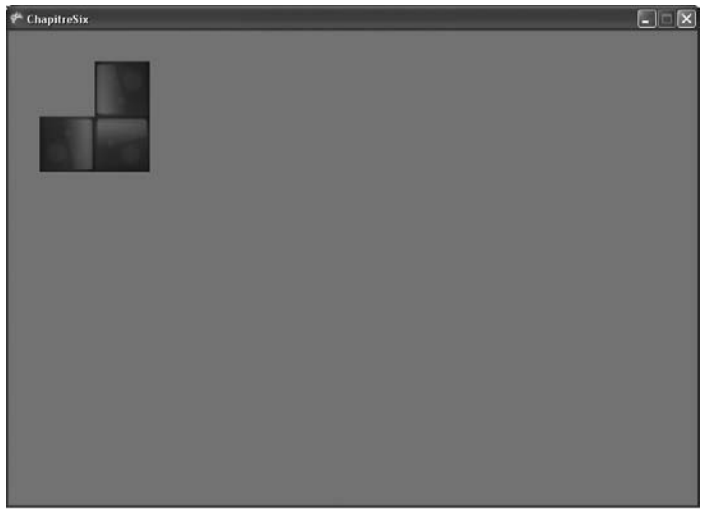
Faisons nos premiers pas dans l'utilisation des rotations. Pour l'instant ne vous préoccupez pas de l'origine.

1. Commencez par ajouter un champ de type `float` qui contiendra la valeur de la rotation et ajoutez également une nouvelle surcharge du constructeur qui prendra en compte cet élément.

```
float rotation = 0;

public float Rotation
{
    get { return rotation; }
    set { rotation = value; }
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, float rotation)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
    this.rotation = rotation;
}

public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, position, sourceRectangle, Color.White, rotation,
    ➥Vector2.Zero, Vector2.One, SpriteEffects.None, 0);
}
```

2. Pour tester la rotation, créez trois sprites. Le premier ne subira aucune rotation, le second une rotation de Π/2, soit 90° dans le sens horaire et le dernier une rotation de – Π/2, soit 90° dans le sens anti-horaire.

Figure 6-15
Premier essai avec les rotations



```
Sprite sprite;
Sprite sprite2;
Sprite sprite3;

public ChapitreSix()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    ServiceHelper.Game = this;
    Components.Add(new KeyboardService(this));
    sprite = new Sprite(new Vector2(100, 100), null, 0);
    sprite2 = new Sprite(new Vector2(100, 100), null, MathHelper.PiOver2);
    sprite3 = new Sprite(new Vector2(100, 100), null, -MathHelper.PiOver2);
}
```

3. Procédez comme dans la première étape pour l'origine de la rotation. Commencez par vous occuper de la classe `Sprite`.

```
Vector2 origin = Vector2.Zero;
public Vector2 Origin
{
    get { return origin; }
    set { origin = value; }
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
➥rotation, Vector2 origin)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
    this.color = color;
    this.rotation = rotation;
    this.origin = origin;
}

public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, position, sourceRectangle, color, rotation, origin,
    ➥Vector2.One, SpriteEffects.None, 0);
}
```

4. Pour tester les rotations en modifiant l'origine, modifiez l'instanciation de votre deuxième sprite. En prenant le point de coordonnées (32, 32), placez l'origine au milieu de la texture : le sprite tournera donc sur lui-même.

```
sprite2 = new Sprite(new Vector2(100, 100), null, Color.White, MathHelper.PiOver2,
➥new Vector2(32,32));
```

Le résultat de ce test est visible sur la figure 6-16.

Figure 6-16
Rotations en modifiant l'origine

Échelle

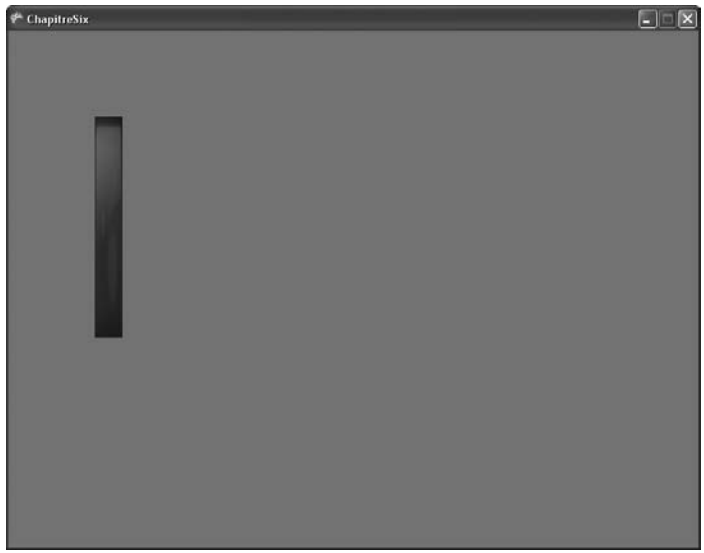
Progressons encore dans l'intégration de nouvelles fonctions à notre classe `Sprite` en nous occupant cette fois-ci de l'échelle. Cette fonctionnalité peut être utilisée, par exemple, pour redimensionner vos sprites en fonction de la résolution d'écran choisie par le joueur.

```
Vector2 scale = Vector2.One;
public Vector2 Scale
{
    get { return scale; }
    set { scale = value; }
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
➥rotation, Vector2 origin, Vector2 scale)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
    this.color = color;
    this.rotation = rotation;
    this.origin = origin;
    this.scale = scale;
}

public void Draw(SpriteBatch spriteBatch)
{
    spriteBatch.Draw(texture, position, sourceRectangle, color, rotation, origin,
    ➥scale, SpriteEffects.None, 0);
}
```

Cette fois-ci, vous n'avez plus besoin que d'un seul et unique sprite pour faire ce test. Dans l'exemple ci-dessous, le sprite est volontairement disproportionné.

```
sprite = new Sprite(new Vector2(100, 100), null, Color.White, 0, Vector2.Zero, new
➥Vector2(0.5f, 4));
```

Figure 6-17
Modification de l'échelle du sprite



Inversion

Procédez toujours de la même manière pour le prochain paramètre. Grâce à lui, si vous disposez d'une texture représentant la marche d'un personnage de la gauche vers la droite, vous pourrez l'inverser selon l'axe vertical et ainsi obtenir la marche de la droite vers la gauche.

```
SpriteEffects effect = SpriteEffects.None;
public SpriteEffects Effect
{
    get { return effect; }
    set { effect = value; }
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
➥rotation, Vector2 origin, Vector2 scale, SpriteEffects effect)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
    this.color = color;
    this.rotation = rotation;
    this.origin = origin;
    this.scale = scale;
```

```
        this.effect = effect;
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, position, sourceRectangle, color, rotation, origin,
        ➥scale, effect, 0);
    }
```

Dans cet exemple, l'inversion se fait selon l'axe horizontal : le haut de la texture va se retrouver en bas.

```
sprite = new Sprite(new Vector2(100, 100), null, Color.White, 0, Vector2.Zero,
➥Vector2.One, SpriteEffects.FlipVertically);
```

Enfin, il ne reste plus qu'à ajouter la définition de la profondeur. C'est cette notion qui définira l'ordre d'affichage des différents sprites à l'écran. Ci-dessous, vous retrouvez le code complet de la classe `Sprite` qui possède maintenant de puissantes fonctions avancées.

La classe Sprite qui prend en compte toutes les techniques vues dans ce chapitre

```
class Sprite
{
    Vector2 position;
    public Vector2 Position
    {
        get { return position; }
        set { position = value; }
    }

    Texture2D texture;
    public Texture2D Texture
    {
        get { return texture; }
    }

    Rectangle? sourceRectangle = null;
    public Rectangle? SourceRectangle
    {
        get { return sourceRectangle; }
        set { sourceRectangle = value; }
    }

    Color color = Color.White;
    public Color Color
    {
        get { return color; }
        set { color = value; }
    }

    float rotation = 0;
    public float Rotation
    {
        get { return rotation; }
        set { rotation = value; }
    }
```

```
Vector2 origin = Vector2.Zero;
public Vector2 Origin
{
    get { return origin; }
    set { origin = value; }
}

Vector2 scale = Vector2.One;
public Vector2 Scale
{
    get { return scale; }
    set { scale = value; }
}

SpriteEffects effect = SpriteEffects.None;
public SpriteEffects Effect
{
    get { return effect; }
    set { effect = value; }
}

float layerDepth = 0;
public float LayerDepth
{
    get { return layerDepth; }
    set { layerDepth = value; }
}

public Sprite(Vector2 position)
{
    this.position = position;
}

public Sprite(Vector2 position, Rectangle? sourceRectangle)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
    this.color = color;
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
➥rotation)
{
    this.position = position;
    this.sourceRectangle = sourceRectangle;
    this.color = color;
    this.rotation = rotation;
}

public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
➥rotation, Vector2 origin)
```

```
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
        this.color = color;
        this.rotation = rotation;
        this.origin = origin;
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
    ➥rotation, Vector2 origin, Vector2 scale)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
        this.color = color;
        this.rotation = rotation;
        this.origin = origin;
        this.scale = scale;
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
    ➥rotation, Vector2 origin, Vector2 scale, SpriteEffects effect)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
        this.color = color;
        this.rotation = rotation;
        this.origin = origin;
        this.scale = scale;
        this.effect = effect;
    }

    public Sprite(Vector2 position, Rectangle? sourceRectangle, Color color, float
    ➥rotation, Vector2 origin, Vector2 scale, SpriteEffects effect, float layerDepth)
    {
        this.position = position;
        this.sourceRectangle = sourceRectangle;
        this.color = color;
        this.rotation = rotation;
        this.origin = origin;
        this.scale = scale;
        this.effect = effect;
        this.layerDepth = layerDepth;
    }

    public void LoadContent(ContentManager content, string assetName)
    {
        texture = content.Load<Texture2D>(assetName);
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, position, sourceRectangle, color, rotation,
        ➥origin, scale, effect, layerDepth);
    }
}
```

Pour tester cette dernière fonctionnalité, vous aurez besoin de deux sprites. L'un aura une valeur de profondeur de 1 et l'autre 0.

```
sprite = new Sprite(new Vector2(100, 100), null, Color.White, 0, Vector2.Zero,
➥Vector2.One, SpriteEffects.FlipVertically, 0);
sprite2 = new Sprite(new Vector2(140, 140), null, Color.White, 0, Vector2.Zero,
➥Vector2.One, SpriteEffects.None, 1);
```

Cependant, ce code n'est pas encore opérationnel. Essayez les deux versions de la méthode `Draw()` ci-dessous.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    sprite.Draw(spriteBatch);
    sprite2.Draw(spriteBatch);
    spriteBatch.End();

    base.Draw(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    spriteBatch.Begin();
    sprite2.Draw(spriteBatch);
    sprite.Draw(spriteBatch);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Vous constaterez que les valeurs de profondeur que vous avez fixées pour chacun des deux sprites ne sont pas appliquées. Dans la première méthode `Draw()`, c'est le deuxième sprite qui chevauche le premier alors que dans la seconde version de la méthode, c'est le premier qui chevauche le deuxième.

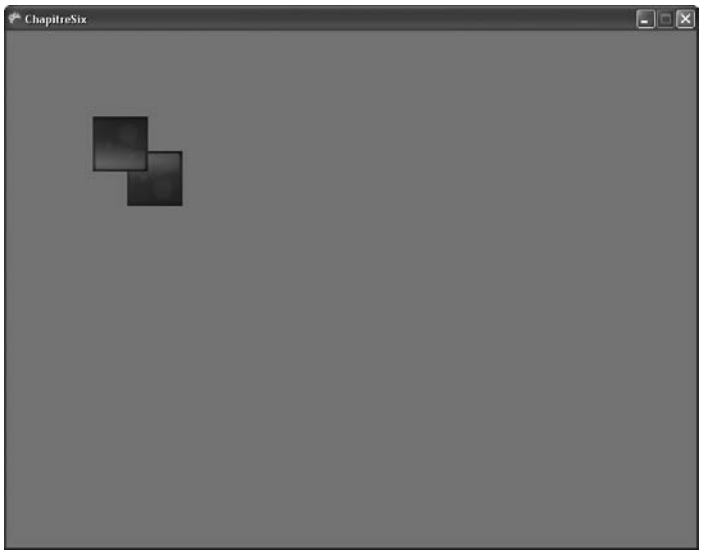
Il faut paramétrer votre objet `spriteBatch` pour que cette fonctionnalité soit effective. Cela se fait par l'intermédiaire de la méthode `Begin()`.

```
spriteBatch.Begin(SpriteBlendMode.None, SpriteSortMode.BackToFront,
➥SaveStateMode.None);
```

Le premier paramètre concerne la transparence. Le second, celui qui vous intéresse, indiquera l'ordre d'affichage des sprites. Il peut prendre comme valeurs `SpriteSortMode.BackToFront` (affichage des sprites les plus profonds d'abord) ou `SpriteSortMode.FrontToBack` (affichage des sprites en partant du premier plan). Enfin, le dernier paramètre permet d'enregistrer l'état du périphérique graphique, il ne sera pas utilisé ici.

À présent, vous constatez que quel que soit l'ordre des lignes d'appel de la méthode `Draw()` des deux sprites, c'est le premier sprite qui est situé au-dessus du second.

Figure 6-18
L'ordre d'affichage des sprites est maintenant respecté



Afficher du texte avec Spritefont

En lecteur curieux, vous vous êtes certainement rendu compte que la classe `SpriteBatch` dispose d'une méthode `DrawString()`. Celle-ci prend comme paramètre un objet de type `SpriteFont`, le texte à afficher, un objet `Vector2` correspondant à la position du coin supérieur gauche de la chaîne et, enfin, la couleur de la chaîne de caractères. Un objet `SpriteFont` sert à indiquer quelle texture utiliser pour dessiner du texte.

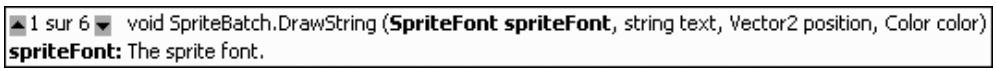


Figure 6-19
Il existe une méthode qui permet d'afficher simplement du texte

Pour créer un `SpriteFont`, commencez par ajouter un fichier dédié au gestionnaire de contenu (figure 6-20). Ouvrez ensuite ce nouveau fichier, qui est en fait un fichier XML.

```
<?xml version="1.0" encoding="utf-8"?>
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">
    <FontName>Kootenay</FontName>
    <Size>14</Size>
    <Spacing>0</Spacing>
    <UseKerning>true</UseKerning>
```

```
    <Style>Regular</Style>
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

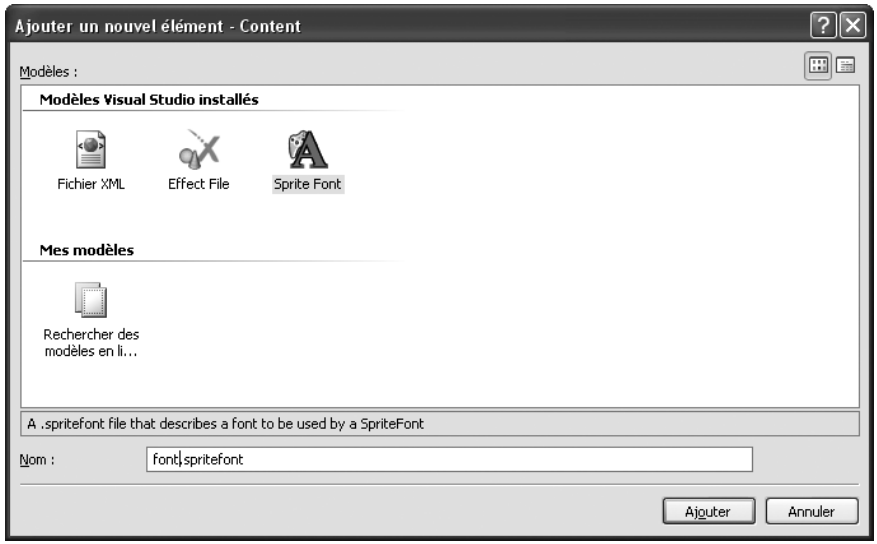


Figure 6-20

Ajout d'un SpriteFont au projet

À partir de ce fichier, définissez la police de caractères que vous souhaitez utiliser, sa taille, l'espace entre les caractères, le style ou encore les caractères qui seront disponibles.

> **En pratique**
>
> La police de caractères que vous voulez employer doit se situer dans le répertoire `Fonts` de Windows. Le Content Manager la transformera ensuite en fichier `.xnb` afin de pouvoir l'utiliser avec votre projet.

La portion de code ci-dessous montre que modifier et adapter une police à ses besoins est réellement intuitif avec XNA. Ici, nous modifions la police utilisée en changeant simplement le nom situé entre les balises `<FontName>`, ainsi que la taille de la police en modifiant le nombre placé entre les balises `<Size>`.

```
<?xml version="1.0" encoding="utf-8"?>
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">
```

```
    <FontName>Verdana</FontName>
    <Size>26</Size>
    <Spacing>10</Spacing>
    <UseKerning>true</UseKerning>
    <Style>Bold</Style>
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Vous n'avez plus qu'à charger la police en déclarant un objet de type `SpriteFont` et enfin compléter la méthode `DrawString()`.

Classe de test du SpriteFont

```
public class ChapitreSix : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SpriteFont font;

    public ChapitreSix()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        ServiceHelper.Game = this;
        Components.Add(new KeyboardService(this));
    }

    protected override void Initialize()
    {
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        font = Content.Load<SpriteFont>("font");
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);
```

```
        spriteBatch.Begin();
        spriteBatch.DrawString(font, "test", new Vector2(0, 0), Color.White);
        spriteBatch.End();

        base.Draw(gameTime);
    }
}
```

Afficher le nombre de FPS

Le nombre de FPS (*Frames Per Second*), c'est-à-dire la quantité d'images affichées par seconde, deviendra vite une de vos plus grandes obsessions. En effet, plus ce nombre est élevé, plus une animation semble fluide. Si vous effectuez un grand nombre de calculs qui ralentissent l'apparition de chaque image, votre jeu risque de donner une impression de saccades.

L'objectif de votre jeu devrait être d'avoisiner les 60 FPS. Par défaut, XNA bridera votre jeu pour qu'il ne dépasse pas cette limite. Vous pouvez considérer que 30 FPS est également acceptable, cependant, évitez de descendre en dessous de ce nombre. Pour information, au cinéma, la norme est de 24 images par secondes.

La mesure du nombre de FPS permet de rapidement juger des performances de votre jeu. Cependant, cela ne pourra pas vraiment vous aider pour optimiser votre code ou comparer la vitesse de différents algorithmes.

Le composant que vous allez maintenant développer vous permettra d'afficher à l'écran le nombre courant de FPS. Ajoutez une nouvelle classe au projet et faites-la dériver de `DrawableGameComponent`. Préparez un objet de type `SpriteBatch` et un autre de type `SpriteFont`.

Il existe de nombreuses méthodes pour calculer le nombre d'images par seconde. La technique utilisée ici est celle proposée par Shawn Hargreaves sur son blog (*http://blogs.msdn.com/shawnhar/*). Elle consiste à compter le nombre de passages dans la méthode `Draw()` puis, à chaque seconde, en déduire le nombre de FPS. La classe `TimeSpan` est accessible dans l'espace de noms `System`, n'oubliez pas de l'ajouter !

Composant d'affichage du nombre de FPS

```
class FPSComponent : DrawableGameComponent
{
    SpriteBatch spriteBatch;
    SpriteFont spriteFont;

    int frameRate = 0;
    int frameCounter = 0;
    TimeSpan elapsedTime = TimeSpan.Zero;

    public FPSComponent(Game game)
        : base(game)
    {
    }
```

```
    public override void Initialize()
    {
        spriteBatch = new SpriteBatch(Game.GraphicsDevice);
        spriteFont = Game.Content.Load<SpriteFont>("font");
    }

    protected override void LoadContent()
    {
    }

    public override void Update(GameTime gameTime)
    {
        elapsedTime += gameTime.ElapsedGameTime;

        if (elapsedTime > TimeSpan.FromSeconds(1))
        {
            elapsedTime -= TimeSpan.FromSeconds(1);
            frameRate = frameCounter;
            frameCounter = 0;
        }
    }

    public override void Draw(GameTime gameTime)
    {
        frameCounter++;

        spriteBatch.Begin();
        spriteBatch.DrawString(spriteFont, frameRate.ToString() + " FPS", new
        ➥Vector2(0, 0), Color.White);
        spriteBatch.End();
    }
}
```

Pour l'utiliser, vous n'avez plus qu'à l'ajouter à la liste des composants.

```
public ChapitreSix()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    ServiceHelper.Game = this;
    Components.Add(new KeyboardService(this));
    Components.Add(new FPSComponent(this));
```

Avant de terminer ce chapitre, un dernier détail concernant l'optimisation de la taille des polices de caractère. Pour celle chargée d'afficher le nombre de FPS, Shawn Hargreaves propose de ne spécifier dans le fichier `.spritefont` que les caractères qui seront utilisés.

```
<?xml version="1.0" encoding="utf-8"?>
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">
    <FontName>Arial</FontName>
    <Size>14</Size>
    <Spacing>2</Spacing>
    <Style>Regular</Style>
```

```
    <CharacterRegions>
      <CharacterRegion>
        <Start>F</Start>
        <End>F</End>
      </CharacterRegion>
      <CharacterRegion>
        <Start>P</Start>
        <End>P</End>
      </CharacterRegion>
      <CharacterRegion>
        <Start>S</Start>
        <End>S</End>
      </CharacterRegion>
      <CharacterRegion>
        <Start> </Start>
        <End> </End>
      </CharacterRegion>
      <CharacterRegion>
        <Start>0</Start>
        <End>9</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

En résumé

Dans ce chapitre vous avez d'abord découvert plusieurs techniques pour enrichir l'affichage de vos sprites :

- n'en sélectionner qu'une portion ;
- modifier leur teinte ;
- effectuer des rotations ;
- modifier leur échelle ;
- les inverser selon un axe ;
- modifier leur ordre d'affichage.

Vous avez également appris à afficher du texte et appliqué cette technique pour créer un composant qui indique le nombre de FPS.

7

La sonorisation

Ce chapitre porte sur un élément fondamental d'un bon jeu : le son. En effet, un bon environnement sonore est indispensable pour donner de la profondeur et du réalisme à votre jeu et favoriser l'immersion du joueur, qu'il s'agisse de la musique ou des bruitages. Pour pousser la logique à son extrême, sachez qu'il existe même des jeux qui ne comportent que du son et aucun élément graphique !

Jusqu'à la version 2 de XNA, le son était géré par l'API XACT, directement tirée de DirectX. Depuis l'arrivée de XNA 3.0, les développeurs ont accès à une nouvelle API pour gérer le son. Cette dernière est plus simple d'utilisation (vous pouvez manier les éléments sonores comme des textures, des SpriteFont, etc.), mais ne vous offrira pas des fonctions aussi avancées que XACT. Pour que vous soyez également en mesure de comprendre le code source d'un jeu écrit pour XNA 2.0, nous couvrirons ces deux méthodes.

Travailler avec XACT

Aussi appelé *Microsoft Cross-Platform Audio Creation Tool*, XACT est la partie du framework qui sert à créer des projets audio aussi bien pour la Xbox 360 que pour Windows. Comme nous l'avons précisé en introduction de ce chapitre, jusqu'à l'arrivée de la version 3.0 du framework, c'était la seule solution pour gérer le son dans XNA.

Mais ce n'est pas parce qu'une nouvelle API est arrivée dans le framework que XACT est un mauvais outil ! Au contraire, il est inclus avec le framework pour permettre d'utiliser les mêmes pistes son et ne pas avoir à redévelopper votre jeu lors du portage entre les différentes plates-formes couvertes par XNA.

Zune

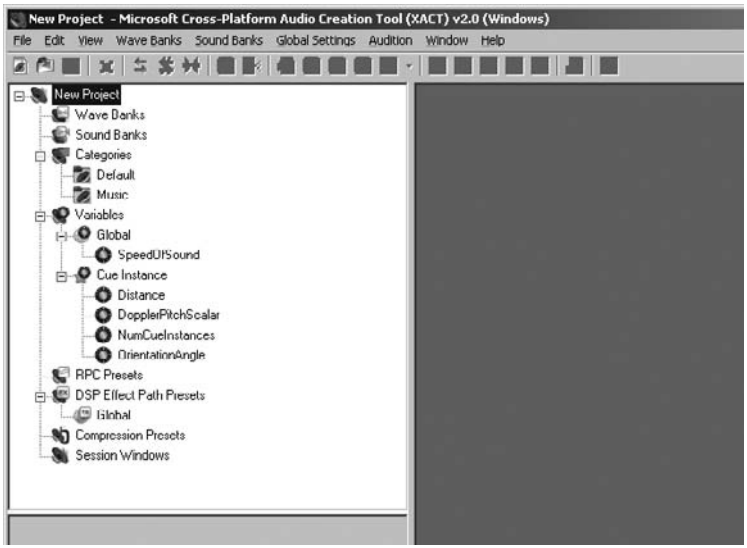
L'utilisation de XACT est impossible pour un projet de jeu Zune.

Créer un projet sonore

Commencez par démarrer le logiciel, qui se trouve au même endroit que les autres utilitaires livrés avec XNA, situés par défaut dans le dossier `C:\Program Files\Microsoft XNA\XNA Game Studio\v3.0\Tools`, ou via le menu Démarrer :

1. Cliquez sur Démarrer.
2. Cliquez sur Tous les programmes.
3. Dans Microsoft XNA Game Studio 3.0 Tools, sélectionnez Microsoft Cross-Platform Audio Creation Tool (XACT).

Figure 7-1
L'interface de XACT



L'interface de l'utilitaire se divise en quatre zones. De haut en bas, nous trouvons :

- La barre des menus qui sert à gérer les projets et l'apparence du logiciel.
- L'explorateur du projet grâce auquel vous inspecterez sous forme d'arborescence tous les éléments du projet.
- L'explorateur de propriétés où vous pourrez modifier les paramètres de tous les éléments du projet.
- La zone centrale qui, bien évidemment, vous permettra de travailler sur les éléments du projet.

En pratique

L'agencement des différentes parties de l'utilitaire est très similaire à l'agencement par défaut dans Visual Studio.

Voyons comment créer un projet sonore :

1. Créez un nouveau projet via le menu File puis New Project. La plupart des éléments de l'interface sont maintenant actifs. Notez que XACT génère deux fichiers de paramétrage du projet : un pour la Xbox 360 et l'autre pour Windows (figure 7-2).

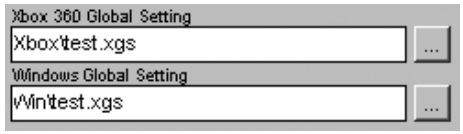


Figure 7-2
Un fichier de configuration par plate-forme

Avant de continuer, il faut connaître un peu la terminologie liée à l'organisation logique des éléments sonores d'un projet XACT.

Tableau 7-1 Signification des éléments d'un projet XACT

Nom	Description
Wave	Il s'agit d'un fichier audio.
Wave Bank	Il s'agit du regroupement logique dans un seul fichier de plusieurs fichiers audio.
Cue	Permet de jouer des sons.
Sound Bank	C'est le regroupement logique de plusieurs Wave Bank et de Cue.

2. Ajoutez une nouvelle banque de sons (Wave Bank). Pour cela, deux solutions s'offrent à vous : via le menu Wave Banks>New Wave Bank, ou bien par l'explorateur de projets grâce à un clic droit sur Wave Banks, puis un clic sur New Wave Bank. Vous pouvez maintenant consulter les propriétés de votre banque de sons. Vous y retrouvez le nom de la banque, une description, son type, sa taille et la méthode de compression utilisée.

3. Ensuite, ajoutez un fichier à la banque : cliquez droit dans la zone de travail puis sur Insert Wave File(s). Les formats supportés sont `.wav`, `.aif`, `.aiff`. Vous voyez à présent diverses informations sur les pistes, notamment un détail de la compression disponible dans l'explorateur de propriétés.

Le format `.wav` est un format de stockage audio défini par Microsoft et IBM. Il peut contenir des données aux formats MP3, WMA, etc. Les formats `.aif` et `.aiff` sont équivalents au `.wav`, mais sont développés par Apple.
Si vous avez de la musique dans votre jeu, déclarez-la telle quelle : déplacez-la dans la catégorie Music de l'arborescence.

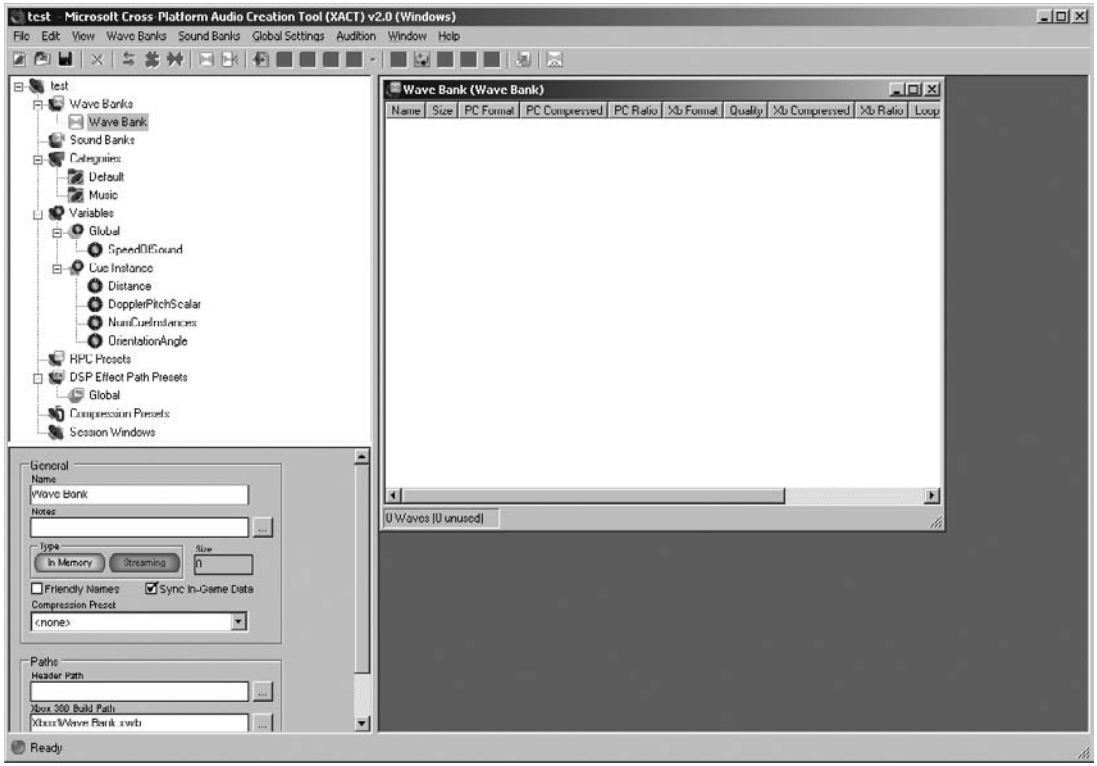


Figure 7-3

Ajout d'une banque de sons

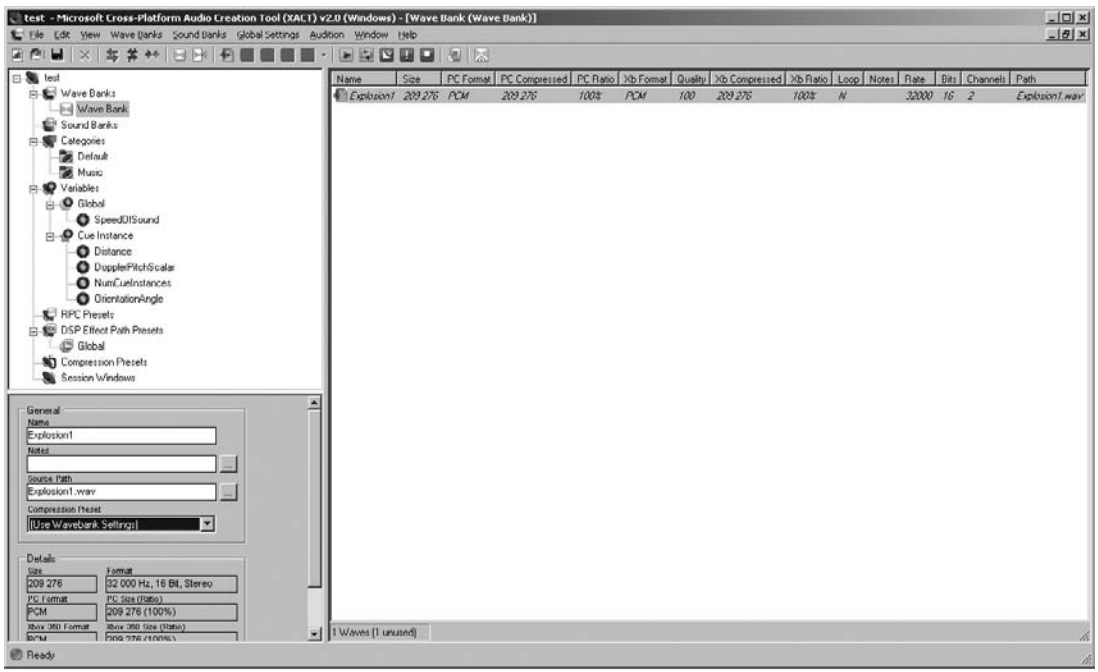


Figure 7-4

La banque de sons contient maintenant une piste

Écouter un son

Si vous voulez écouter une piste sonore à partir de XACT, vous devrez d'abord lancer l'utilitaire XACT Auditioning Utility disponible au même endroit.

4. Créez ensuite une Sound Bank (via le menu Sound Banks, puis New Sound Bank). Dans la fenêtre qui s'ouvre alors, vous distinguez deux zones : celle du haut contient les pistes audio et celle du bas rassemble les pistes Cue correspondantes. Pour ajouter une piste à la Sound Bank, glissez-déposez la piste vers la deuxième zone.

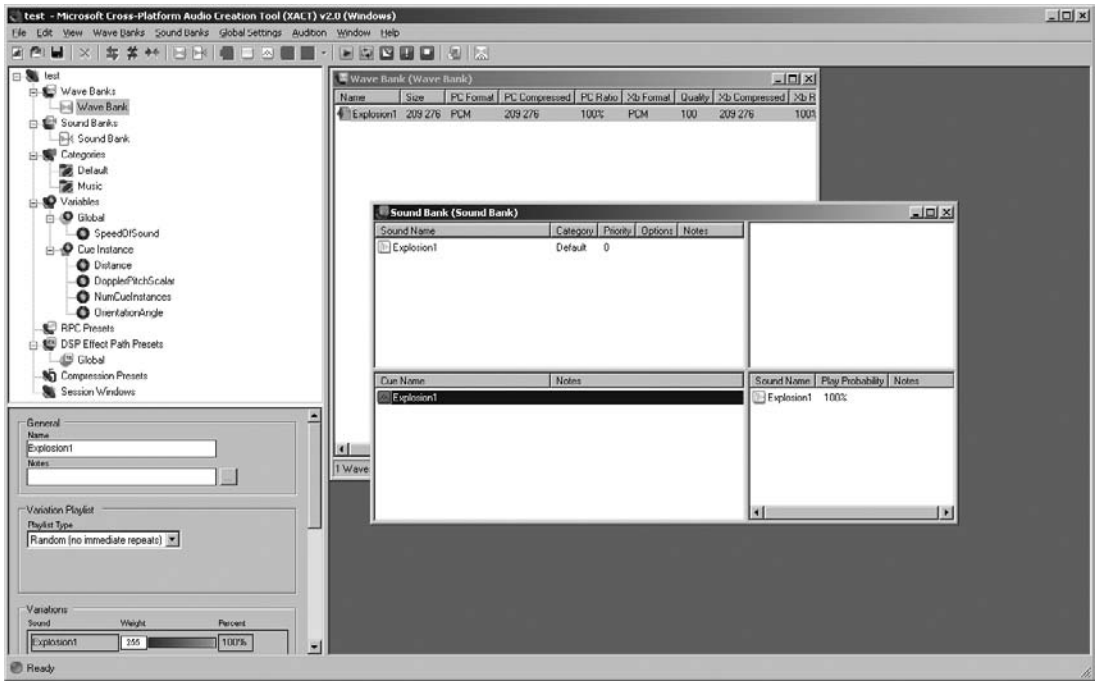


Figure 7-5

Ajout d'une piste à la Sound Bank

5. À présent, le projet est prêt à être généré : cliquez sur File, puis sur Build, ou utilisez le raccourci clavier F7. Les fichiers générés sont disponibles dans les sous-dossiers `Win` ou `Xbox`, à l'emplacement où vous avez sauvegardé votre projet XACT.

Lire les fichiers créés

Votre projet a été généré, il ne reste plus qu'à l'importer dans Visual Studio pour l'utiliser.

De retour dans Visual Studio, commencez par créer un nouveau projet afin de tester les possibilités sonores de XNA. L'organisation logique que nous avons créée précédemment

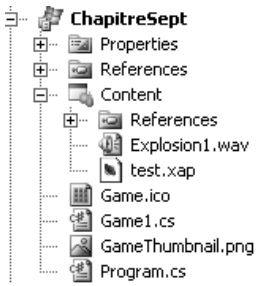
se charge aussi simplement qu'une texture. En effet, il suffit d'ajouter le fichier `.xap` du projet XACT dans le Content Manager, qui importe automatiquement le projet vers le jeu. Cependant, vous ne devez pas vous contenter d'ajouter ce fichier au projet, il faut aussi ajouter toutes les pistes audio à utiliser dans le projet.

En pratique

Dans le cas d'un projet de jeu complet, cette organisation logique est beaucoup plus complexe : vous devrez en effet gérer de nombreuses banques de sons (pour les bruitages d'un personnage ou de l'environnement, les musiques, etc.).

Figure 7-6

Voila à quoi devrait ressembler le projet



Il est enfin temps de passer à la programmation d'une classe test du projet sonore. Créez des objets de type `AudioEngine`, `SoundBank` et `WaveBank` en chargeant les fichiers générés par XACT. Attention, vous devez spécifier le chemin complet vers ces fichiers ! N'oubliez pas non plus les extensions de fichiers.

Vous pouvez ensuite lire une piste Cue simplement en utilisant la méthode `PlayCue ()` de l'objet de type `SoundBank`. N'oubliez pas d'appeler auparavant la méthode `Update()` de l'objet `AudioEngine` pour le mettre à jour. Celle-ci devra également être appelée dans la méthode `Update()` de la classe principale.

Classe test pour la lecture d'une piste Cue

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    AudioEngine engine;
    WaveBank waveBank;
    SoundBank soundBank;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        engine = new AudioEngine(@"Content\test.xgs");
        soundBank = new SoundBank(engine, @"Content\Sound Bank.xsb");
        waveBank = new WaveBank(engine, @"Content\Wave Bank.xwb");
    }
```

```
    protected override void Initialize()
    {
        engine.Update();
        soundBank.PlayCue("Explosion1");

        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        engine.Update();

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);
        base.Draw(gameTime);
    }
}
```

Exécutez le programme. Le son est lu sans problème. Charger et lire un son est finalement aussi simple que pour les textures !

Vous auriez également pu stocker la piste dans un objet de type `Cue` et récupérer celui-ci grâce à la méthode `GetCue` du `SoundBank`. La piste se lit ensuite grâce à la méthode `Play()` :

Classe test d'utilisation d'un objet de type Cue

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    AudioEngine engine;
    WaveBank waveBank;
    SoundBank soundBank;
    Cue sound;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        engine = new AudioEngine(@"Content\test.xgs");
        soundBank = new SoundBank(engine, @"Content\Sound Bank.xsb");
        waveBank = new WaveBank(engine, @"Content\Wave Bank.xwb");
    }
```

```
    protected override void Initialize()
    {
        engine.Update();
        sound = soundBank.GetCue("Explosion1");
        sound.Play();

        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        engine.Update();

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);
        base.Draw(gameTime);
    }
}
```

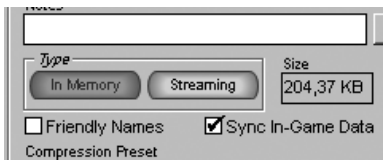
Lire les fichiers en streaming

La méthode que nous venons de voir charge les sons en mémoire. Cette solution est peu recommandée dès que le projet dispose d'un certain nombre de pistes, surtout si elles ne sont pas compressées. La seconde méthode que nous allons détailler permet de lire les données en streaming, c'est-à-dire en chargement continu.

1. Ouvrez de nouveau XACT et le projet de la première méthode.
2. Cependant, cette fois-ci, sélectionnez Streaming dans les propriétés de la `Wave Bank`.

Figure 7-7

Sélection du mode streaming



3. Reconstruisez le projet (F7).
4. Au niveau du code du projet, vous devez seulement appeler un autre constructeur pour la classe `WaveBank`.

```
waveBank = new WaveBank(engine, @"Content\Wave Bank.xwb", 0, 2);
```

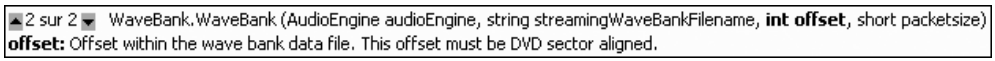


Figure 7-8

Le nouveau constructeur de la classe WaveBank

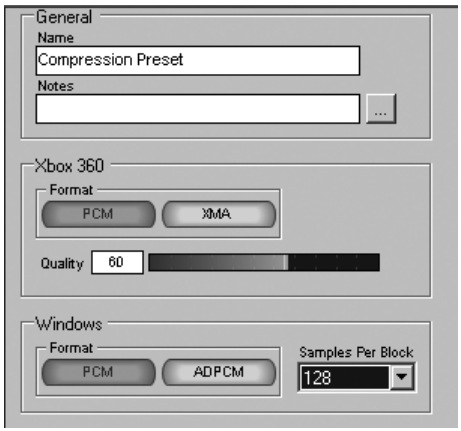
5. Le premier des deux nouveaux paramètres est l'*offset* de démarrage de la ressource Wave Bank. Ce paramètre est utile si vous lisez un son depuis un DVD, sinon laissez-le à 0. Le second paramètre est la taille du buffer utilisé pour le streaming. Vous pouvez considérer qu'il s'agit de la valeur qui déterminera la qualité du son. La valeur minimale est de 2 et il n'est pas conseillé de dépasser 16, faute de quoi le son risque de ne plus être diffusé de manière ininterrompue.

Compression

Si vous utilisez le mode In Memory, vous préférerez sûrement compresser les pistes pour prendre le moins de place possible en mémoire. Retournez au projet dans XACT. Dans l'explorateur de projet, faites un clic droit sur Compression Presets, puis sur New Compression Preset.

Figure 7-9

Propriété d'un nouveau preset de compression



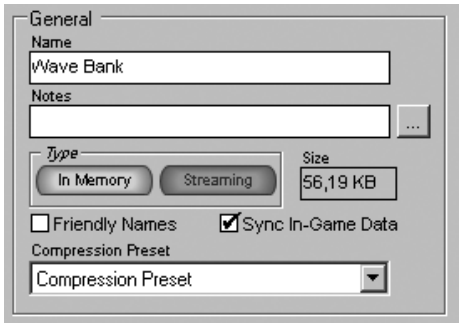
La compression n'est pas la même pour la Xbox que pour Windows. Dans les deux cas, PCM signifie que le fichier ne sera pas compressé.

Tableau 7-2 Les méthodes de compression

Compression	Description
XMA (Xbox 360)	Vous devez spécifier la qualité de la piste via un curseur. Plus la valeur est faible, plus la qualité est faible, mais plus la compression est importante. Par défaut, la valeur est de 60.
ADPCM (Windows)	Vous devez spécifier le nombre d'échantillons par bloc. Plus le nombre est grand, plus la qualité est satisfaisante, mais moins la compression est bonne. Par défaut, la valeur est de 128 échantillons par bloc.

Une fois que vous avez créé un *preset*, vous pouvez l'appliquer soit à une piste, soit à un Wave Bank. Vous n'avez plus qu'à reconstruire le projet.

Figure 7-10
Application d'un preset à une banque de sons



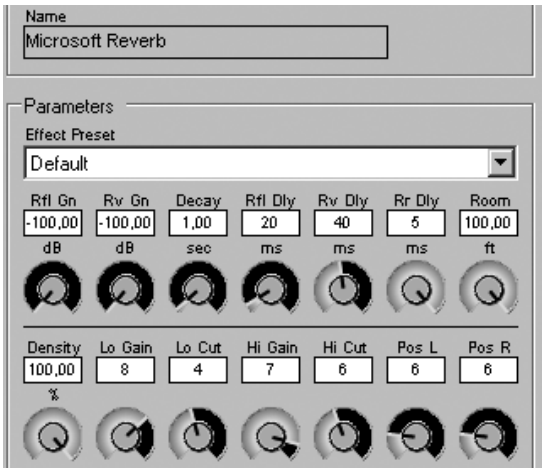
Ajouter un effet de réverbération

L'effet de « réverb », bien connu des musiciens, vise à donner l'impression d'être dans un lieu plus ou moins vaste. Il peut être ajouté aux pistes via XACT. Dans l'explorateur de projet, ces effets sont regroupés sous le nom « DSP Effect Path Presets ».

Voici comment ajouter un nouvel effet :

1 Dans XACT, cliquez avec le bouton droit sur la catégorie DSP Effect Path Presets de l'explorateur de projet, puis sélectionnez New Microsoft Reverb Project.

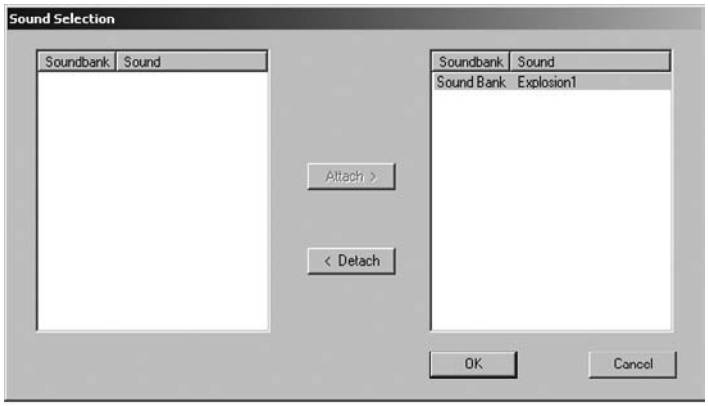
Figure 7-11
Le paramétrage de la réverbération peut être très poussé



2. À partir de là, vous pouvez régler très précisément les paramètres de la réverbération (figure 7-11). Vous pouvez aussi choisir d'utiliser les paramètres préréglés disponibles dans la liste déroulante Effect Preset, qui permettent d'obtenir facilement la réverbération d'une cave, d'une salle de concert, d'un hangar, etc.

3. Reste à appliquer l'effet à une piste. Cliquez avec le bouton droit sur le nom de l'effet dans l'explorateur de projet, puis cliquez sur Attach/Detach Sound(s)… À partir de la fenêtre qui s'est ouverte, attachez l'effet à chacune des pistes ou détachez-le.

Figure 7-12
La fenêtre de liaison pistes/effet



4. Pour rendre ces modifications utilisables dans le jeu, reconstruisez le projet (F7) ; aucune autre modification n'est nécessaire sous Visual Studio.

Le son avec la nouvelle API SoundEffect

Vous venez de voir que XACT est très simple d'utilisation. Cependant, les développeurs de XNA ont eu des remontées de nombreuses personnes qui trouvaient son utilisation un peu lourde pour des petits projets, notamment à cause de l'utilisation d'un outil externe et de la gestion de l'arborescence de la banque de sons.

Les développeurs ont donc ajouté une nouvelle API, baptisée SoundEffect, pour la gestion du son. Celle-ci rend le chargement et l'utilisation de pistes sonores aussi simples que dans le cas d'une texture, sans que vous ayez à gérer tout un projet sonore comme vous le feriez pour XACT. De plus, si vous avez l'intention de porter le projet pour Zune, sachez que seule cette nouvelle API est disponible en ce qui concerne le son.

Lire un son

La lecture d'un son se fera en utilisant un objet de type `SoundEffect`. Ajoutez simplement un fichier `.wav` au projet comme vous ajouteriez une texture (voir le chapitre 6), puis créez l'objet `SoundEffect` et chargez le son. La lecture se fait ensuite grâce à la méthode `Play()`.

Test de l'API SoundEffect

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SoundEffect soundEffect;
```

```
    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
    }

    protected override void Initialize()
    {
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        soundEffect = Content.Load<SoundEffect>("Explosion1");
        soundEffect.Play();
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);
        base.Draw(gameTime);
    }
}
```

Lire un morceau de musique

Grâce à la classe `MediaLibrary`, vous accédez à la musique présente dans la bibliothèque multimédia de l'utilisateur. Ces musiques doivent avoir été détectées au préalable par Windows Media Player. Vous pourrez ensuite jouer un morceau d'un disque présent dans cette bibliothèque multimédia grâce à la classe `MediaPlayer`.

Le code ci-dessous lit la première piste d'un album choisi au hasard dans la bibliothèque du joueur. Si ce dernier appuie sur la touche Espace et que la piste est en cours de lecture, elle est mise en pause. Si elle est déjà en pause, la lecture reprend.

Test de classe MediaPlayer

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
```

```
    SpriteBatch spriteBatch;

    MediaLibrary sampleMediaLibrary;
    Random rand;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        sampleMediaLibrary = new MediaLibrary();
        rand = new Random();
    }

    protected override void Initialize()
    {
        ServiceHelper.Game = this;
        Components.Add(new KeyboardService(this));
        int i = rand.Next(0, sampleMediaLibrary.Albums.Count - 1);
        MediaPlayer.Play(sampleMediaLibrary.Albums[i].Songs[0]);
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Space))
        {
            if (MediaPlayer.State == MediaState.Playing)
                MediaPlayer.Pause();
            else
                MediaPlayer.Resume();
        }

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
        base.Draw(gameTime);
    }
}
```

Pour un bon design sonore

À première vue, le son n'est pas l'élément qui vous causera le plus de soucis lors de la création d'un jeu vidéo : il suffit juste de charger les sons et de les jouer. Cependant, les joueurs n'ont pas tous les mêmes équipements audio !

De plus, certains joueurs n'aiment pas les musiques présentes dans les jeux et préfèrent écouter les leurs… Veillez donc à leur laisser la possibilité de les éteindre sans pour autant couper tous les bruitages, par exemple par l'intermédiaire d'un menu d'options.

Ci-dessous vous retrouverez quelques exemples de la liste des actions conseillées par le site du Creators Club pour le son des jeux de la Xbox 360 (la liste complète est disponible sur *http://creators.xna.com/en-US/education/bestpractices*). Certains de ces conseils s'appliquent aussi aux jeux développés pour les autres plates-formes.

- Essayez de tester la partie sonore du jeu sur le plus grand nombre de configurations possibles (stéréo, mono, casques, etc.).
- Assurez-vous, au moment de leur édition, que le volume des musiques et/ou des bruitages est normalisé afin que les joueurs n'aient pas à le modifier au cours du jeu. Pour définir cette norme, basez-vous sur le volume du son de démarrage de la Xbox 360 : réglez-le pour qu'il soit un peu fort et ajustez ensuite le volume du jeu en conséquence.
- Faites attention : sur la Xbox 360, la méthode `Play()` de la classe `MediaPlayer` est asynchrone, ce qui signifie que les musiques ne sont pas immédiatement lues. Si vous vérifiez l'état de l'objet `MediaPlayer` dans l'appel à la méthode `Update()` qui suit la lecture de la musique, il sera probablement toujours sur `MediaState.Stopped`. Ainsi, si vous voulez démarrer une nouvelle musique à la fin de la lecture de l'ancienne, n'utilisez pas la méthode précédente, mais attendez plutôt l'événement `ActiveSongChanged` puis, seulement à partir de ce moment-là, vérifiez l'état du `MediaPlayer`.
- Par défaut, le gestionnaire de contenu (le *Content Processor* pour être plus précis) compresse au maximum les éléments qui seront utilisés par l'API `SoundEffect`. Songez-y et souvenez-vous que la compression améliore la taille du jeu au détriment de la qualité du son.

En résumé

Dans ce chapitre, vous avez découvert :

- comment gérer un projet sonore avec XACT et utiliser ce projet dans XNA ;
- quels sont les avantages et inconvénients de l'API SoundEffect par rapport à XACT et comment utiliser cette API dans XNA ;
- comment utiliser la classe MediaPlayer pour jouer des musiques de la bibliothèque multimédia de l'utilisateur ;
- quelques bonnes pratiques à appliquer dans le design sonore d'un jeu.