4

Interactions avec le joueur

Dans un jeu vidéo, l'interaction avec le joueur peut se faire via différents périphériques.

Ce chapitre a pour but de vous apprendre à contrôler les périphériques utilisables avec XNA. Ce sera aussi l'occasion de découvrir un nouvel aspect de la programmation objet que vous appliquerez directement pour la gestion du clavier. Nous verrons également des exemples d'interactions avancées intéressantes à mettre en place dans vos jeux.

Utiliser les périphériques

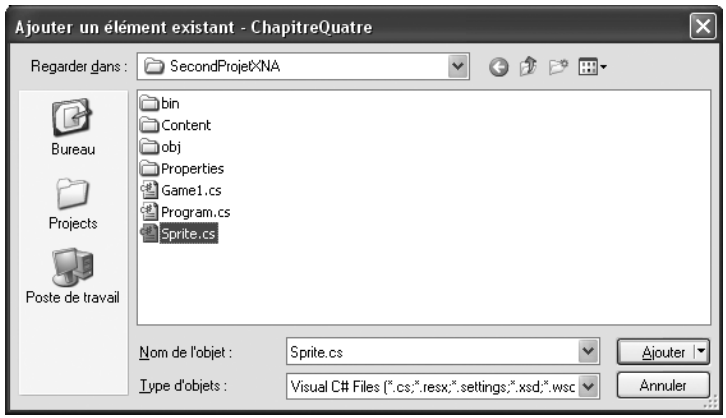
Cette première partie va vous présenter les différents périphériques compatibles avec XNA et leur utilisation. N'oubliez pas que, pour la Xbox 360, la souris et le clavier ne constituent pas des périphériques de base, vous devrez alors plutôt vous concentrer sur la gestion de la manette. En revanche, la situation est inversée si vous développez pour Windows. Toutefois, vous allez constater que les périphériques se gèrent très facilement avec XNA. Le portage de votre code d'une machine vers une autre est, de ce fait, un jeu d'enfant !

Le clavier

Le clavier est un périphérique de base pour un jeu sur PC. Il est beaucoup utilisé dans les jeux de tirs ou de course.

Commencez par créer un nouveau projet basé sur XNA. Importez votre classe `Sprite`, ajoutez une image au projet puis utilisez-la en créant un nouveau sprite.

Figure 4-1
Récupérez facilement une classe écrite précédemment



Pense-bête

Lorsque vous récupérerez une classe créée dans un précédent projet pour l'utiliser dans un nouveau, n'oubliez pas de modifier la déclaration de l'espace de noms pour pouvoir l'utiliser sans devoir ajouter de directive `using` au nouveau projet.

Comme les classes qui vont être utilisées ici font partie de l'espace de noms `Microsoft.Xna.Framework.Input`, n'oubliez pas d'ajouter une directive `using`, cela simplifiera l'écriture du code.

Le framework met à votre disposition l'objet `Keyboard` qui possède la méthode `GetState()` et – comble de bonheur – il dispose également de la classe `KeyboardState`. Une fois l'état du clavier récupéré, vous aurez accès à l'ensemble des touches pressées au moment de l'appel à la méthode et pourrez également déterminer si une touche donnée est pressée ou non.

L'ensemble des touches de votre clavier (et même certaines touches auxquelles vous n'avez jamais fait attention) sont disponibles via l'énumération `Keys`. Rappelons qu'une énumération est un type de données constitué d'un ensemble de constantes. Voici un exemple de déclaration d'énumération :

```
enum CouleurDesYeux
{
    bleu,
    marron,
    vert,
    gris
};
```

L'accès à une valeur de l'énumération se fait en écrivant le nom de l'énumération suivi d'un point et de la valeur voulue :

```
CouleurDesYeux.bleu;
```

Dans l'exemple suivant, l'utilisateur a maintenant la possibilité de quitter le jeu en appuyant sur la touche Échap de son clavier (à laquelle nous accédons via `Keys.Escape`) :

```
protected override void Update(GameTime gameTime)
{
    KeyboardState KState = Keyboard.GetState();

    if (KState.IsKeyDown(Keys.Escape))
        this.Exit();

    base.Update(gameTime);
}
```

Vous pouvez également écrire une forme plus contractée, qui n'utilise pas de variable pour stocker l'état du clavier. Il faut alors faire intervenir l'objet `Keyboard` comme suit :

```
if (Keyboard.GetState().IsKeyDown(Keys.Escape))
    this.Exit();
```

Voyons à présent comment déplacer notre sprite. Ajoutons une variable qui contiendra sa vitesse de déplacement.

```
float speed = 0.1f;
```

Le reste est très simple. Selon la touche fléchée sur laquelle il appuie, l'utilisateur déplace le sprite dans la direction qu'il souhaite, tout en modulant cette translation par le temps qui s'est écoulé depuis la dernière frame, comme vous l'avez vu au chapitre 3.

```
protected override void Update(GameTime gameTime)
{
    KeyboardState KState = Keyboard.GetState();

    if (KState.IsKeyDown(Keys.Left))
        sprite.Update(new Vector2(-1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed, 0));
    if (KState.IsKeyDown(Keys.Right))
        sprite.Update(new Vector2(1 * gameTime.ElapsedGameTime.Milliseconds * speed,
        ➥0));
    if (KState.IsKeyDown(Keys.Up))
        sprite.Update(new Vector2(0, -1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed));
    if (KState.IsKeyDown(Keys.Down))
        sprite.Update(new Vector2(0, 1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed));

    base.Update(gameTime);
}
```

Pour traiter une combinaison de touches, utilisez la méthode `GetPressedKeys()` qui renvoie un tableau de `Keys`. L'exemple suivant affiche la liste des touches sur lesquelles le joueur a appuyé à la place du titre de la fenêtre.

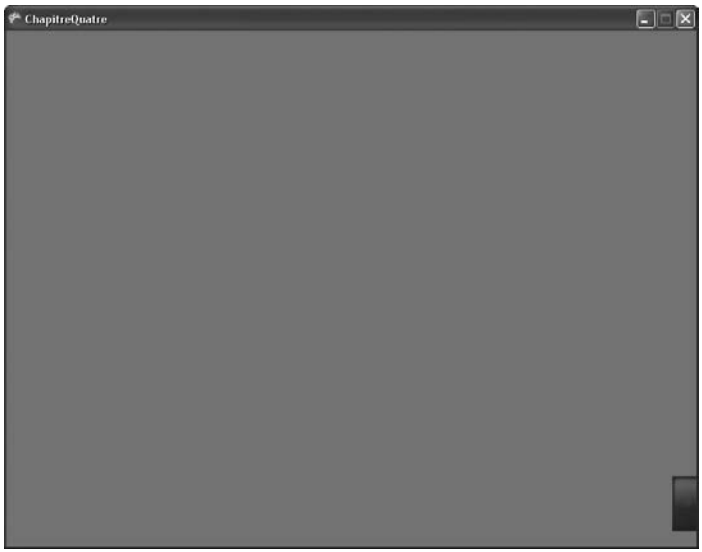


Figure 4-2

Le joueur peut maintenant déplacer son sprite où bon lui semble.

```
KeyboardState KState = Keyboard.GetState();

this.Window.Title = "";
foreach (Keys key in KState.GetPressedKeys())
    this.Window.Title += key.ToString();
```

La souris

Il est temps à présent d'étudier la souris. Ce périphérique est particulièrement adapté aux jeux de tir et de gestion.

Cette fois-ci encore, vous pouvez utiliser un objet `MouseState` de manière à récupérer l'état de la souris mis à disposition par l'objet `Mouse`. De la même manière que pour le clavier, vous pouvez grâce à cet objet connaître l'état de la souris : les boutons utilisés, la position de la souris ou encore le nombre d'interventions sur la molette.

Dans un premier temps, il faut modifier votre classe `Sprite`. Jusqu'à présent, vous ne pouviez qu'appliquer des translations à votre Sprite or, dans l'exemple suivant, vous souhaitez pouvoir modifier directement sa position. Ajoutez donc une propriété en lecture et en écriture concernant la position de votre sprite.

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

À présent, vous avez tous les outils en main pour transformer votre sprite en curseur. À chaque appel de la méthode `Update()`, il suffit de remplacer la position du sprite par celle de la souris.

```
protected override void Update(GameTime gameTime)
{
    MouseState MState = Mouse.GetState();
    sprite.Position = new Vector2(MState.X, MState.Y);

    base.Update(gameTime);
}
```

Notez que, comme pour le clavier, vous n'êtes pas obligé de stocker l'état de la souris et pouvez l'exploiter directement.

```
sprite.Position = new Vector2(Mouse.GetState().X, Mouse.GetState().Y);
```

En ce qui concerne les clics de la souris, vous les détectez en utilisant l'énumération `ButtonState` qui prend deux états : `Pressed` ou `Released`.

```
if (MState.LeftButton == ButtonState.Pressed)
this.Window.Title = "Gauche";
if (MState.MiddleButton == ButtonState.Pressed)
this.Window.Title = "Milieu";
if (MState.RightButton == ButtonState.Pressed)
this.Window.Title = "Droit";
```

Pour finir, vous pouvez récupérer les mouvements qu'effectue l'utilisateur avec sa molette via la propriété `ScrollWheelValue`. Celle-ci retourne une valeur entière qui s'incrémentera si l'utilisateur fait tourner la molette vers le haut et qui se décrémentera dans le cas contraire.

```
this.Window.Title = MState.ScrollWheelValue.ToString();
```

La manette de la Xbox 360

La manette de la Xbox 360 est le contrôleur de base que tous les joueurs de la console possèdent. Si vous développez un jeu pour la console, vous devrez donc adapter son fonctionnement à cette manette.

Le fonctionnement de la manette est similaire à celui du clavier et de la souris. Vous stockez son état dans un objet de type `GamePadState` que vous récupérerez de l'objet `GamePad`. Cependant, comme il peut y avoir plusieurs manettes connectées à la console, il faut spécifier celle qui vous intéresse grâce à l'énumération `PlayerIndex`.

```
GamePadState GPState = GamePad.GetState(PlayerIndex.One);
```

Tout d'abord, vous pouvez vérifier qu'une manette est bien connectée à la console en utilisant la propriété `IsConnected`.

```
GamePadState GPState = GamePad.GetState(PlayerIndex.Two);

if (GPState.IsConnected)
    this.Window.Title = "La manette 2 n'est pas connectée à la console";
```

Vous avez accès à deux méthodes classiques, l'une permettant de savoir si un bouton est pressé, l'autre s'il ne l'est pas. Le choix du bouton se fait grâce à l'énumération `Buttons`, laquelle vous permet également d'accéder à l'intégralité des boutons d'une manette Xbox 360.

```
if (GPState.IsButtonDown(Buttons.A))
    this.Exit();

if (GPState.IsButtonUp(Buttons.B))
    this.Window.Title = "Le bouton B n'est pas pressé";
```

Comme d'habitude, il est tout à fait possible de s'affranchir du stockage de l'état de la manette.

```
if (GamePad.GetState(PlayerIndex.One).IsButtonDown(Buttons.A))
    this.Exit();
```

Vous pourriez également estimer l'état d'un bouton avec l'énumération `ButtonState`. Un bouton a soit l'état `Pressed`, soit l'état `Released`.

```
if (GPState.Buttons.A == ButtonState.Pressed)
    this.Exit();
```

De la même manière, vous pourrez récupérer l'état du pad directionnel. Le cas des diagonales est géré, il peut donc y avoir deux directions pressées.

```
if (GPState.DPad.Down == ButtonState.Pressed)
    Window.Title = "Vers le haut";
```

L'état des gâchettes analogiques gauche et droite peut aussi être récupéré. La propriété renverra un `float`, compris entre 0 et 1, 1 signifiant que la gâchette est complètement enfoncée. Cette variation de valeur pour être utilisée, par exemple, pour l'accélération ou la décélération dans un jeu de course.

```
this.Window.Title = GPState.Triggers.Left.ToString();
```

En ce qui concerne les sticks analogiques, il est possible de récupérer un objet de type `Vector2` correspondant à la distance qui les sépare de la position initiale des sticks.

```
this.Window.Title = GPState.ThumbSticks.Left.X + " ; " + GPState.ThumbSticks.Left.Y;
```

Enfin, la fonction `SetVibration` de l'objet `GamePad` permet de faire vibrer la manette. Elle attend comme paramètres la manette concernée, la vitesse à appliquer au moteur gauche et celle à appliquer au moteur droit. Notez également que la fonction renvoie un booléen vous indiquant si les vibrations ont eu lieu. Lorsque vous souhaitez arrêter les vibrations, il vous suffira de passer une vitesse nulle en paramètre de la fonction.

L'exemple suivant fait vibrer la manette durant 5 secondes.

```
int time = 0;

protected override void Update(GameTime gameTime)
{
```

```
        time += gameTime.ElapsedGameTime.Milliseconds;

        if (time < 5000)
        {
            if (!GamePad.SetVibration(PlayerIndex.One, 1, 1))
                this.Window.Title = "Impossible de faire vibrer la manette";
        }
        else
            this.Window.Title = "Temps écoulé";

        base.Update(gameTime);
    }
```

Vous ne le savez peut être pas mais la manette de la Xbox 360 peut également fonctionner sur votre ordinateur ; il est possible d'en brancher jusqu'à quatre et bien entendu de les utiliser dans XNA. Il faut pour cela utiliser un module USB que vous connecterez à votre ordinateur.

Utilisation de périphériques spécialisés

La création d'un bon jeu passe par la définition d'un *gameplay* innovant. L'utilisation de périphériques sortant de l'ordinaire permet d'accéder à cette phase d'innovation, nous en donnons pour preuve les jeux de rythmes musicaux qui rencontrent un grand succès depuis quelques années.

En ce qui concerne la Xbox 360, les périphériques spéciaux – qu'il s'agisse d'une guitare, d'une batterie, d'un tapis de danse ou autre – agissent comme une manette sur la console. C'est-à-dire qu'une guitare d'un jeu musical peut très bien faire office d'arme dans un jeu de tir par exemple. Ce n'est donc pas plus dur de développer un jeu prévu pour utiliser un de ces périphériques !

Vous devez toutefois faire attention à certains périphériques qui n'implémentent pas tous les boutons d'une manette classique. Voici la liste (provenant de Microsoft) des composants obligatoirement présents sur un périphérique :

- les boutons A, B, X et Y ;
- les boutons Back, Start et Xbox Guide ;
- le pad directionnel.

Cela signifie donc que toutes les autres parties d'une manette de Xbox 360, notamment les gâchettes ou les sticks directionnels, ne sont pas toujours présentes sur les périphériques de la console.

Les services avec XNA

La deuxième partie de ce chapitre va vous présenter ce qu'est un service dans XNA et comment l'utiliser. Vous créerez enfin un composant qui vous permettra de récupérer facilement tous vos services n'importe où dans le code de votre jeu.

Les interfaces en C#

Une interface contient des prototypes de méthodes et de propriétés et peut ainsi être comparée à un contrat. Une classe qui signe le contrat, c'est-à-dire qui implémente une interface, s'engage à fournir une implémentation du contenu de l'interface.

Retournez dans un projet en mode `console` et imaginez par exemple l'interface présentée ci-dessous. Remarquez bien que la méthode `Identification()` n'est pas définie, il y a seulement son prototype.

```
public interface IUseless
{
    string Identification();
}
```

Nom d'une interface

Ce n'est pas une règle officielle, mais généralement les développeurs font précéder le nom d'une interface par un I.

Il n'est pas non plus précisé quel type de valeur est concerné : c'est un nouveau pas vers la généricité. Dans l'exemple ci-dessous, deux classes qui implémentent cette interface sont présentées.

```
class Machine : IUseless
{
    string serialNumber;

    public Machine(string serialNumber)
    {
        this.serialNumber = serialNumber;
    }

    public string Identification()
    {
        return serialNumber;
    }
}

class Human : IUseless
{
     string name;
     string firstName;

     public Human(string name, string firstName)
     {
         this.name = name;
         this.firstName = firstName;
     }

     public string Identification()
     {
         return firstName + " " + name;
     }
}
```

La signature du contrat s'écrit donc comme un héritage (voir le chapitre 3 à ce sujet) :

```
Classe : Interface
```

Cependant, à la différence de l'héritage, une classe peut implémenter plusieurs interfaces. La syntaxe serait la suivante :

```
Classe : InterfaceA, InterfaceB
```

L'utilisation des classes se fait d'une manière tout à fait classique.

```
class Program
{
    static void Main(string[] args)
    {
        Human human = new Human("Dylan", "Bob");
        Console.WriteLine(human.Identification());

        Machine machine = new Machine("B563RF2");
        Console.WriteLine(machine.Identification());

        Console.Read();
     }
}
```

Comment utiliser les services

Les services ont été conçus pour récupérer, à partir de votre classe principale (celle qui hérite de `Game`), un objet qui implémente une interface donnée. Leur avantage est donc de faciliter l'utilisation des méthodes d'un objet sans avoir à le passer sans cesse en paramètre et sans utiliser de variable statique.

La gestion des services se fait tout simplement par la propriété `Services` de la classe `Game`. Elle correspond à un objet de type `GameServiceContainer` qui dispose des trois méthodes `AddService()`, `GetService()` et `RemoveService()`. Les services sont en fait stockés dans un objet `Dictionary<Type, Object>`.

Vous allez maintenant créer votre premier service qui vous servira à gérer le clavier. Dans la première partie de ce chapitre, nous avons présenté les deux possibilités qui existent pour récupérer les entrées de l'utilisateur : à savoir utiliser directement l'objet `Keyboard` ou alors passer par un objet `KeyboardState`. Or, dès que vous commencerez à travailler sur un projet conséquent, vous vous rendrez compte que l'accès répété à l'objet `Keyboard` peut être à l'origine de pertes de performances. En fait, vous pourriez n'y accéder qu'une fois et mettre à disposition de tout le monde l'objet `KeyboardState` : cela correspond parfaitement à la définition des services.

L'écriture du contrat se fait très rapidement : une fonction suffit pour savoir si une touche est effectivement pressée.

```
interface IKeyboardService
{
    bool IsKeyDown(Keys key);
}
```

Maintenant, ajoutez une nouvelle classe au projet et nommez-la `KeyboardService`. Cette classe devra implémenter l'interface `IKeyboardService`, mais devra aussi hériter de la classe `GameComponent`, laquelle implémente les interfaces `IGameComponent` et `IUpdateable`. Ces interfaces imposent respectivement l'implémentation des méthodes `Initialize()` et `Update()`. La classe `Game` dispose justement d'une collection de `GameComponent`, ainsi les méthodes citées précédemment seront automatiquement appelées pour les objets de cette collection.

Vous retrouverez ici la classe `KeyboardService`. Il n'y a rien de particulier à signaler ici : dans le constructeur, on se contente d'ajouter la classe courante à la collection de service de l'objet `Game` et dans la méthode `Update()`, on enregistre l'état du clavier. Enfin, la fonction `IsKeyDown()` se contentera de consulter la méthode éponyme de l'objet `KBState`.

```
class KeyboardService : GameComponent, IKeyboardService
{
    KeyboardState KBState;

    public KeyboardService(Game game)
        : base(game)
    {
        game.Services.AddService(typeof(IKeyboardService), this);
    }

    bool IsKeyDown(Keys key)
    {
        return KBState.IsKeyDown(key);
    }

    public override void Update(GameTime gameTime)
    {
        KBState = Keyboard.GetState();
        base.Update(gameTime);
    }
}
```

Ajoutez, dans la méthode `Initialize()` de votre classe Game, un nouvel objet `KeyboardService` à la collection de `Components`.

```
this.Components.Add(new KeyboardService(this));
```

Dans la méthode `Update()`, vous allez devoir accéder à votre service. La ligne suivante sert à récupérer un service depuis la collection de la classe `Game`. Il est important de garder en mémoire que la collection est un dictionnaire et que les valeurs sont donc indexées par type : il faut préciser celui de notre service, c'est le but de l'opérateur `typeof`.

```
this.Services.GetService(typeof(IKeyboardService))
```

Cependant, à ce stade, l'objet que vous récupérez n'est toujours pas du type `IKeyboardService`, vous allez donc devoir le convertir. Cette opération s'appelle le casting : en précisant le

type voulu entre parenthèses juste avant l'objet, vous pourrez accéder aux méthodes de l'objet.

Le casting

Lorsqu'il est nécessaire de convertir une donnée dans un type différent de celui choisi par la conversion automatique, il faut préciser de manière explicite le nouveau type : cette opération s'appelle le casting.

Attention, cela peut parfois être source d'une perte de données. Si vous convertissez, par exemple, un nombre décimal de type `float` vers un `int`, vous perdrez la partie décimale du nombre.

```
((IKeyboardService)this.Services.GetService(typeof(IKeyboardService)))
➥.IsKeyDown(Keys.Up)
```

Il est à présent possible de réécrire la méthode du début de ce chapitre.

```
protected override void Update(GameTime gameTime)
{
    if (((IKeyboardService)this.Services.GetService(typeof(IKeyboardService)))
    ➥.IsKeyDown(Keys.Up))
        sprite.Update(new Vector2(0, -1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed));

    if (((IKeyboardService)this.Services.GetService(typeof(IKeyboardService)))
    ➥.IsKeyDown(Keys.Down))
        sprite.Update(new Vector2(0, 1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed));

    if (((IKeyboardService)this.Services.GetService(typeof(IKeyboardService)))
    ➥.IsKeyDown(Keys.Left))
        sprite.Update(new Vector2(-1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed, 0));

    if (((IKeyboardService)this.Services.GetService(typeof(IKeyboardService)))
    ➥.IsKeyDown(Keys.Right))
        sprite.Update(new Vector2(1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed, 0));

    base.Update(gameTime);
}
```

Enfin, notez qu'il n'est pas obligatoire d'écrire une interface pour un service. Il est possible de procéder directement ainsi :

```
game.Services.AddService(typeof(KeyboardService), new KeyboardService());
```

Les méthodes génériques

Comme vous venez de le constater, récupérer un service pour l'utiliser peut être ennuyeux à la longue. Vous allez maintenant apprendre à créer une classe qui vous en simplifiera l'utilisation.

Les méthodes génériques permettent d'écrire des méthodes qui effectueront exactement le même traitement, peu importe le type des paramètres passés. Vous avez utilisé une méthode générique chaque fois que vous avez chargé une texture pour vos sprites.

```
texture = content.Load<Texture2D>(assetName);
```

La déclaration d'une telle classe se fait de la manière suivante :

```
public static void Sample<T>(T param)
{
    //…
}
```

Vous pouvez également ajouter des conditions sur le type des paramètres. Le tableau 4-1 en dresse une liste :

Tableau 4-1 Différentes conditions possibles sur le type des paramètres

Contrainte	Description
`where T : struct`	Le type T doit être un type valeur.
`where T : class`	Le type T doit être un type référence.
`where T : new()`	Le type T doit disposer d'un constructeur public sans paramètres.
`where T : <classe de base>`	Le type T doit être dérivé de la classe spécifiée.
`where T : <interface>`	Le type T doit implémenter l'interface spécifiée.

Ajoutez une nouvelle classe au projet et nommez-la `ServiceHelper`. Elle fera office de classe statique et contiendra un champ statique qui sera utilisé pour stocker une référence vers votre classe `Game`.

```
static class ServiceHelper
{
    static Game game;

    public static Game Game
    {
        set { game = value; }
    }
}
```

Ajoutez une méthode statique et générique : elle devra ensuite ajouter à la collection de services l'objet qu'elle a reçu en paramètre.

```
public static void Add<T>(T service) where T : class
{
    game.Services.AddService(typeof(T), service);
}
```

Il faut ensuite ajouter une seconde méthode pour récupérer un service ; elle sera également statique et générique. Notez l'utilisation du mot-clé `as`, utile à la conversion de types références.

```
public static T Get<T>() where T : class
{
    return game.Services.GetService(typeof(T)) as T;
}
```

Ci-dessous, le code complet de la classe.

```
static class ServiceHelper
{
    static Game game;

    public static Game Game
    {
        set { game = value; }
    }

    public static void Add<T>(T service) where T : class
    {
        game.Services.AddService(typeof(T), service);
    }

    public static T Get<T>() where T : class
    {
        return game.Services.GetService(typeof(T)) as T;
    }
}
```

Vous devez maintenant modifier le constructeur de la classe `KeyboardService` pour qu'il passe par la classe `ServiceHelper`, plutôt qu'ajouter directement le service à la collection de l'objet `Game`.

```
public KeyboardService(Game game)
    : base(game)
{
    ServiceHelper.Add<IKeyboardService>(this);
}
```

Pour rendre cette classe utilitaire opérationnelle, il vous reste à définir la référence vers votre classe `Game`.

```
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    ServiceHelper.Game = this;
    Components.Add(new KeyboardService(this));
}
```

À présent, tout est prêt pour que vous puissiez utiliser plus simplement vos services. Ci-dessous, vous trouverez le corps de la méthode `Update()` dans sa nouvelle version. Beaucoup plus lisible, n'est-ce pas ?

```
protected override void Update(GameTime gameTime)
{
    if(ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Up))
        sprite.Update(new Vector2(0, -1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed));

    if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Down))
        sprite.Update(new Vector2(0, 1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed));

    if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Left))
        sprite.Update(new Vector2(-1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed, 0));

    if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Right))
        sprite.Update(new Vector2(1 * gameTime.ElapsedGameTime.Milliseconds *
        ➥speed, 0));

    base.Update(gameTime);
}
```

Toujours plus d'interactions grâce à la GUI

Pour conclure, sachez que les interactions avec le joueur ne passent pas seulement par les périphériques, mais également par le *gameplay* et les mécanismes de jeux que vous pouvez implémenter.

L'un des points les plus importants dans un jeu réside dans la communication avec le joueur, qui peut s'exercer par l'intermédiaire d'une GUI (*Graphical User Interface*).

Graphical User Interface

Les GUI ont été présentes dès les débuts du jeu vidéo, en effet ceux-ci on toujours eu besoin de donner des indications aux joueurs. Mais ces dernières ont fortement évolué depuis Pong, leur aspect graphique en faisant parfois de véritables petites œuvres d'art.

Aujourd'hui la tendance est plutôt de limiter fortement le nombre d'informations affichées à l'écran, afin d'augmenter le sentiment d'immersion (par exemple dans les jeux *Gears of War* ou *Mirror's Edge*).

Cependant, ce comportement n'est adapté qu'à certains types de jeux (jeux à la première personne notamment). Sauf gameplay particulier, il est difficile d'imaginer un jeu de gestion qui ne présente aucune information au joueur.

Il existe de nombreux projets de GUI disponibles sur Internet et utilisables dans vos jeux. En voici deux :

- XNA Simple Gui (*http://www.codeplex.com/simplegui*) qui, comme son nom l'indique, arbore un design très simpliste mais possède néanmoins une liste de contrôles assez intéressante (panel, boutons, etc.).
- WinForms (*http://www.ziggyware.com/news.php?readmore=374*) vous propose une grande liste de contrôles (et même des barres de progression ou des potentiomètres), ainsi qu'un design proche de celui des fenêtres de Windows XP.

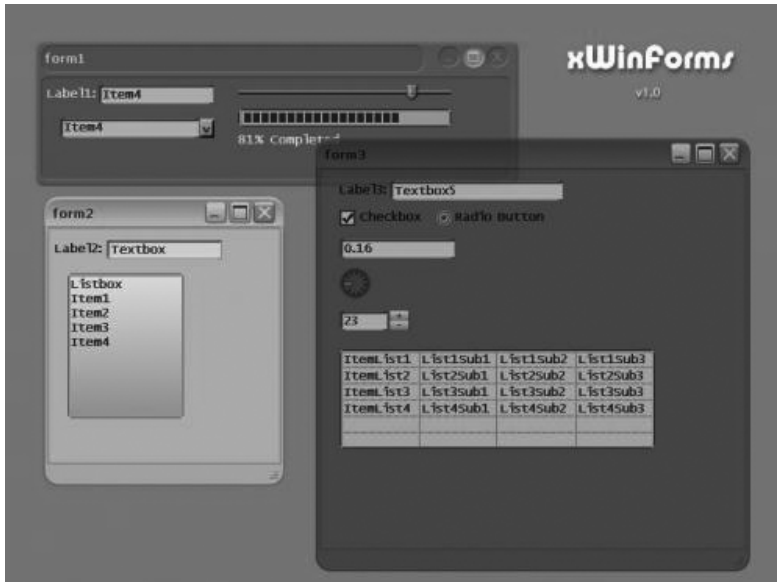


Figure 4-3

Le projet xWinForms, une interface graphique pour XNA

Il est également possible de faire interagir les joueurs entre eux en local ou même en réseau. Ces deux notions seront abordées au chapitre 11.

En résumé

Dans ce chapitre, vous avez découvert les différents périphériques compatibles avec XNA et leur utilisation dans vos jeux. Vous avez ensuite abordé la notion de services, que vous avez mise en pratique pour pouvoir exploiter plus facilement le clavier dans vos jeux.