8

Exceptions et gestion des fichiers : sauvegarder et charger un niveau

Vous allez sûrement vouloir charger des niveaux et en ajouter facilement de nouveaux aux jeux. Vous voudrez aussi probablement que l'expérience de jeu qu'auront les utilisateurs s'inscrive dans la durée en enregistrant leur progression ou leur score.

Ce chapitre apporte toutes les réponses à vos questions en ce qui concerne le stockage de données sur les périphériques physiques (disque dur, carte mémoire, etc.). Il commence par une partie théorique sur les emplacements de stockage offerts par XNA, les fichiers XML et la sérialisation. Vous découvrirez ensuite les *Gamer Services*. Enfin, vous passerez à la pratique en apprenant à gérer et à utiliser fichiers et répertoires.

Le stockage des données

Dans cette première partie, vous allez découvrir la théorie liée aux différents emplacements de stockage mis à disposition par XNA, ainsi qu'aux différentes méthodes de sauvegarde de données.

Les espaces de stockage

Le framework XNA met à votre disposition deux espaces de stockage bien distincts :

- Le dossier du jeu dans lequel se situe l'exécutable, ainsi que tout le contenu que vous aurez créé (textures, sons, etc.).

Zune

Sur le Zune, cet espace de stockage se limite à 16 Mo. Cela correspond à la mémoire vive qu'un jeu peut utiliser.

- Le dossier de l'utilisateur où se situent les sauvegardes ou la configuration préférée d'un joueur.

  Sous Windows, il s'agit du répertoire `SavedGames` situé dans le répertoire personnel de l'utilisateur connecté sur le PC. À l'intérieur de ce répertoire se trouve un dossier correspondant au nom du jeu exécuté. Enfin, la dernière ramification correspond au numéro du joueur (`PlayerIndex`). Si aucun n'est spécifié, les fichiers se trouvent dans le répertoire `AllPlayers`, sinon dans un répertoire correspondant à ce numéro (`Player1`, `Player2`, `Player3` ou `Player4`).

  Sur Xbox, il s'agit du disque dur ou d'une carte mémoire : c'est le joueur qui choisira le périphérique de stockage qu'il souhaite utiliser.

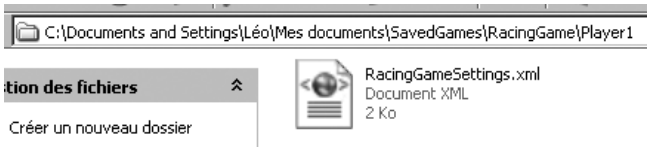


Figure 8-1

Fichier personnel de configuration du jeu Racing Game

Lecteur réseau

Si votre dossier personnel (`Documents and Settings\<Utilisateur>`) ne se situe pas sur l'ordinateur où vous exécuterez un jeu qui doit sauvegarder des données personnelles, mais sur un lecteur réseau, une exception sera levée rendant la sauvegarde impossible. À l'heure actuelle, l'exécution d'un jeu à travers le réseau n'est pas supportée par XNA.

Le format XML, un format intelligible qui simplifie le stockage de données XML (e*Xtensible Markup Language*, en français langage extensible de balisage) est utilisé pour contenir des données qui seront encadrées par des balises. Si vous vous êtes déjà aventuré dans la création d'un site web, vous avez probablement rencontré le HTML qui repose également sur des balises utilisées pour formater l'affichage de données.

L'autre spécificité de XML est qu'il n'y a pas une liste de balises définies : c'est à vous de créer celles qui vous seront utiles.

Ainsi, XML est utilisé dans des domaines très variés :

- Les fichiers de configuration de logiciels ou de jeux sont de plus en plus basés sur ce format. Certains utilisent même ces fichiers pour la configuration de l'interface utilisateur, permettant ainsi aux utilisateurs novices de la paramétrer très facilement.

- Dans les jeux vidéos, les fichiers de sauvegarde ou ceux qui décrivent les niveaux peuvent aussi utiliser le format XML.

Ci-dessous, vous retrouvez un fichier de configuration fictif, utilisant le XML, qui pourrait correspondre à la résolution de l'écran dans un de vos jeux.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<GameConfiguration>
<ScreenSize width="800" height="600" />
</GameConfiguration >
```

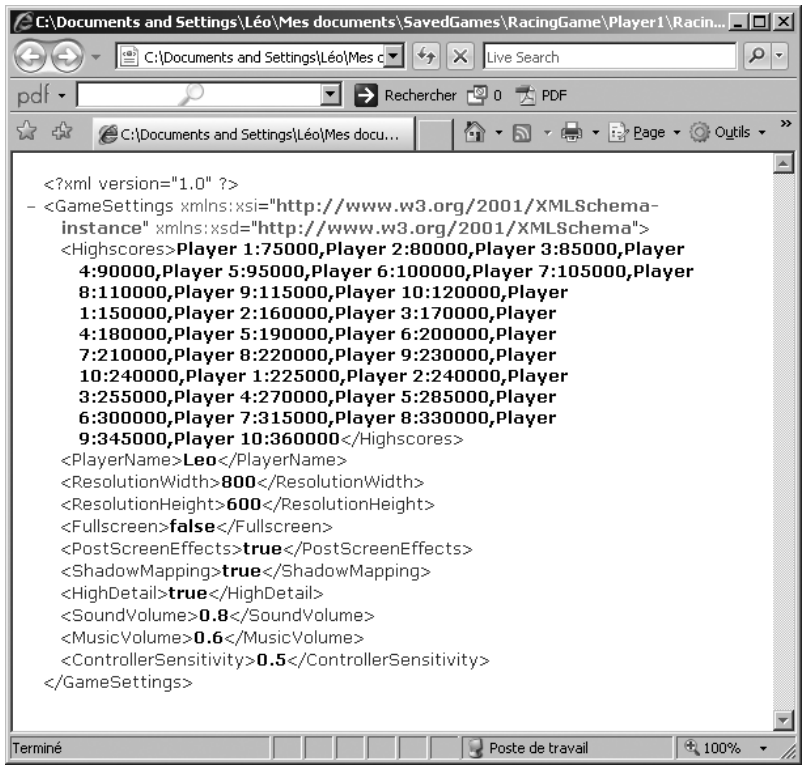


Figure 8-2

Un fichier XML ouvert dans Internet Explorer 7

Cependant, beaucoup d'informaticiens ne se sont pas encore ralliés à la cause de XML. Certains préféreront utiliser des formats utilisant des séparateurs de données. L'exemple qui suit correspond au fichier de configuration fictif de l'exemple précédent, mais cette fois-ci dans le format CSV (*Comma Separated Values*, valeurs séparées par des virgules).

```
800,600
```

D'autres préféreront les fichiers INI, pourtant progressivement abandonnés par Microsoft depuis Windows 95. Dans ce genre de fichier, vous définissez des sections puis vous

affectez des valeurs à différentes variables. Ci-dessous, la configuration de la résolution de l'écran au format INI.

```
[ScreenSize]
Width=800
Height=600
```

Libre à vous d'utiliser le format que vous préférez. Vous pouvez même en inventer un, ou utiliser des fichiers binaires (qui ne pourront pas être lus directement par un éditeur de texte). Retenez cependant que XML est un langage très simple à utiliser et très générique.

Sérialisation et désérialisation

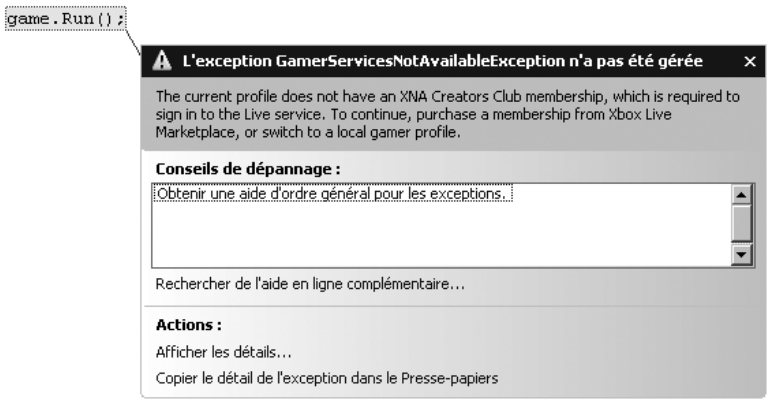
La sérialisation (en anglais *serialization*) est un processus qui permet d'enregistrer l'état complet d'un objet à un moment donné pour le sauvegarder ou l'envoyer vers le réseau ou un périphérique. L'état d'un objet signifie l'ensemble des valeurs de ses champs. L'opération inverse, qui consiste à reformer l'objet, s'appelle la désérialisation.

Les données peuvent être sérialisées sous de nombreuses formes : fichiers binaires, XML, etc.

Les exceptions

Essayez de vous connecter avec un compte qui n'est pas membre du XNA Creators Club. Que se passe-t-il ? La fenêtre du jeu se ferme et le focus est donné à Visual Studio où une bien étrange boîte de dialogue est apparue. Cela signifie qu'une exception a été levée, en l'occurrence `GamerServicesNotAvailableException`.

Figure 8-3
Une exception a été levée



Quand une erreur survient, une exception est levée. À partir de ce moment, l'exécution normale est interrompue et un gestionnaire d'exceptions est recherché dans le bloc d'instructions courant. S'il n'est pas trouvé, la recherche se poursuit dans le bloc englo-

bant celui-ci ou, à défaut, dans le bloc de la fonction appelante, et ainsi de suite. Si la recherche n'aboutit pas, une boîte de dialogue signalant l'exception s'affiche.

Si votre jeu avait été exécuté en mode Release, par exemple par un de vos amis dont vous auriez aimé avoir l'avis, une boîte de dialogue peu commode se serait affichée (figure 8-4).

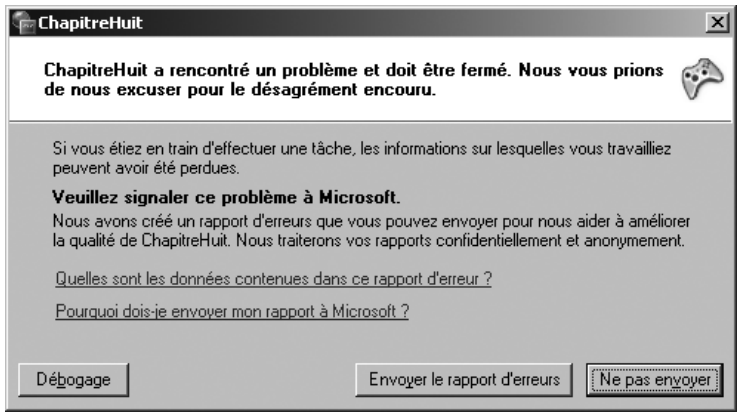


Figure 8-4

Voici la fenêtre qui apparaîtra si vous gérez mal vos exceptions

Vous allez donc devoir protéger votre jeu de ces arrêts brutaux en ajoutant autant de gestionnaires d'exceptions que nécessaire. En pratique, il faut en ajouter chaque fois qu'une portion de code est susceptible de rencontrer un problème : tentative de connexion au réseau impossible, accès à des données qui n'existent pas, division par zéro, dépassement de l'indice maximum lorsque vous manipulez des tableaux, etc. La liste peut être très longue…

L'extrait de code ci-dessous vous présente la structure basique d'un gestionnaire d'exceptions. Le code susceptible de générer une exception, et qui doit dont être testé, est celui présent dans le bloc `try`. Vous pouvez ensuite récupérer l'exception pour la traiter dans le bloc `catch`.

```
try
{
    // Portion de code pouvant lancer une exception
}
catch(Exception e)
{
    // Traitement de l'exception
}
```

L'exemple suivant divise une variable `a` par une variable `b`. Ici vous connaissez la valeur de `b`, or cela n'est pas toujours le cas. Dans le doute, il est préférable d'ajouter un gestionnaire d'exceptions pour se prémunir contre une division par zéro. Si une exception est levée, sa description est affichée dans le titre de la fenêtre du jeu.

```
int a = 10;
int b = 0;
try
{
    a /= b;
}
catch (Exception e)
{
    Window.Title = e.Message;
}
```

Dans l'intitulé de la fenêtre, vous pouvez lire :

```
Tentative de division par zéro.
```

En C#, les exceptions particulières dérivent toutes de la classe `Exception`. L'exemple précédent a levé une exception de type `DivideByZeroException`, mais aucun bloc `catch` qui lui correspond n'est présent. Il en existe tout de même un traitant une exception de type `Exception`, c'est ce bloc qui sera exécuté.

Vous pouvez donc cumuler les blocs `catch` de manière à effectuer un traitement spécial pour certaines erreurs. L'extrait de code ci-dessous reprend l'exemple précédent en affichant un message spécial si l'exception levée est une division par zéro et un message plus générique s'il s'agit d'une autre exception (sait-on jamais).

```
int a = 10;
int b = 0;
try
{
    a /= b;
}
catch (DivideByZeroException e)
{
    Window.Title = "Je le savais";
}
catch (Exception e)
{
    Window.Title = e.Message;
}
```

Cette fois-ci, dans l'intitulé de la fenêtre, vous pouvez lire le message personnalisé.

```
Je le savais
```

Vous pouvez également ajouter un bloc `finally` qui sera exécuté, qu'une exception ait été levée ou non.

```
int a = 10;
int b = 0;
```

```
try
{
    a /= b;
}
catch (DivideByZeroException e)
{
    Window.Title = "Je le savais";
}
catch (Exception e)
{
    Window.Title = e.Message;
}
finally
{
    Window.Title = "Il est passé ici";
}
```

La levée d'une exception se fait en utilisant le mot-clé `throw` suivi d'un objet du type de l'exception voulue. Dans l'exemple suivant, la fonction `TestException` lèvera une exception si le paramètre `i` vaut 0.

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

    try
    {
        TestException(0);
    }
    catch (Exception e)
    {
        Window.Title = e.Message;
    }
}

private void TestException(int i)
{
    if (i == 0)
        throw new Exception("i vaut 0 !");
}
```

Personnaliser les exceptions

Vous pouvez créer vos propres exceptions selon vos besoins. Il suffit de les faire dériver de la classe Exception.

Les Gamer Services : interagir avec l'environnement

Les *Gamer Services* sont un ensemble de fonctionnalités qui permettent au jeu d'interagir avec son environnement : boîte de dialogue pour informer l'utilisateur, récupérer des messages de celui-ci, afficher sa liste d'amis et surtout, accéder à un périphérique de sauvegarde.

Dossier de l'utilisateur

Commencez par ajouter au jeu un nouveau composant de type `GamerServicesComponant`, faute de quoi vous ne pourrez pas utiliser le dossier de l'utilisateur.

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

Vous êtes à présent en mesure d'utiliser toutes les fonctionnalités mises à votre disposition par la classe `Guide` (de l'espace de noms `Microsoft.Xna.Framework.GamerServices`). Par exemple, le code ci-dessous affichera l'écran de connexion au Xbox LIVE (un abonnement XNA Creators Club est requis pour pouvoir se connecter). Vous serez ensuite en mesure de récupérer des informations à propos du joueur.

Figure 8-5

L'écran de connexion au Xbox LIVE s'affiche facilement

```
protected override void Initialize()
{
    base.Initialize();
    Guide.ShowSignIn(1, false);
}
```

Grâce aux *Gamer Services*, vous accédez au dossier de l'utilisateur. Essayez la classe ci-dessous. Si vous êtes sur Xbox 360, la méthode `BeginShowStorageDeviceSelector()` affichera l'écran de sélection du périphérique de sauvegarde. Ensuite, la méthode `EndShowStorageDeviceSelector()` renverra un objet `StorageDevice`. La méthode `OpenContainer()` de cet objet renverra un objet de type `StorageContainer` que vous utiliserez pour vos fichiers de sauvegarde.

Récupérer le dossier de l'utilisateur

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

    StorageDevice device;
    StorageContainer container;

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

        this.Components.Add(new GamerServicesComponent(this));
    }

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

        Guide.BeginShowStorageDeviceSelector(new AsyncCallback(GetDevice), null);
    }

    private void GetDevice(IAsyncResult result)
    {
        if (result.IsCompleted)
        {
            try
            {
                device = Guide.EndShowStorageDeviceSelector(result);
                container = device.OpenContainer("ChapitreHuit");
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
                ➥string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }
}
```

Attention

La gestion du dossier de l'utilisateur dans un jeu sur la Xbox 360 peut s'avérer plus difficile que sous Windows. En effet, il se peut par exemple qu'un joueur débranche le périphérique sélectionné en plein jeu. Ne vous limitez donc pas sur l'utilisation de blocs `try … catch`.

Les méthodes asynchrones

Dans les exemples précédents, vous affichiez les détails des exceptions dans la barre de titre des fenêtres. Ceci n'est pas très esthétique et reste peu pratique pour le joueur qui ne remarquera pas forcément que le titre de la fenêtre à été modifié, surtout s'il joue en plein écran… La solution est donc d'afficher le message dans la fenêtre du jeu : dans une console ou bien dans une boîte de dialogue. Vous pourrez créer facilement une boîte de dialogue si vous employez dans votre jeu un projet de GUI qui met ce genre d'éléments à votre disposition, mais vous pouvez aussi simplement utiliser les *Gamer Services*.

La classe `Guide` possède une méthode `BeginShowMessage ()`. Le tableau ci-dessous présente les différents paramètres qu'elle attend.

Tableau 8-1 Paramètres de la méthode BeginShowMessage

Paramètre	Description
`PlayerIndex player`	Joueur concerné par la boîte de dialogue. Sous Windows, il ne peut s'agir que du joueur 1.
`String title`	Titre de la boîte de dialogue.
`String text`	Contenu textuel de la boîte de dialogue.
`IEnumerable<string> buttons`	Description à afficher sur chacun des boutons de la boîte de dialogue. Il peut y avoir au maximum trois boutons.
`Int focusButton`	Index (zéro étant la valeur minimale) du bouton qui doit avoir le focus.
`MessageBoxIcon icon`	Type de l'icône à afficher avec la boîte de dialogue (`Alert`, `Error`, `None`, `Warning`).
`AsyncCallback callback`	La méthode à appeler une fois que l'opération asynchrone est terminée.
`Object state`	Un objet créé par l'utilisateur pour identifier l'appel à la méthode.

Vous vous demandez sûrement à quoi sert le paramètre `callback` de type `AsyncCallback`. Généralement, l'appel à une fonction se fait de manière synchrone, c'est-à-dire qu'aucune autre instruction n'est exécutée avant que la fonction ne retourne une valeur :

1. Appel de la fonction.
2. Exécution de la fonction.
3. Le thread qui a appelé la fonction récupère la main.

Or, le traitement de certaines fonctions peut être assez long, notamment dans le cas des fonctions d'entrées-sorties ou de communication à travers le réseau. Il est donc nécessaire de les traiter en parallèle, c'est-à-dire de manière asynchrone.

Pour effectuer un traitement asynchrone, vous aurez donc besoin de trois méthodes. La première, dont le nom commence par `Begin`, commande l'exécution de l'opération. Une fois celle-ci terminée, un délégué (*delegate* en anglais, une variable permettant d'appeler une fonction) est appelé. Celui-ci appelle alors la méthode dont le nom commence par `End`.

Commencez par écrire la fonction qui appellera la méthode dont le nom commence par `End`. Elle doit prendre en paramètre une interface de type `IAsyncResult`, qui sera transmise à la méthode commençant par `End`. Cette interface dispose du booléen `IsCompleted` permettant de savoir si l'opération est terminée ou non.

```
private void EndShowMessageBox(IAsyncResult result)
{
    if (result.IsCompleted)
        Guide.EndShowMessageBox(result);
}
```

Il ne reste plus qu'à appeler la méthode commençant par `Begin` dans un bloc `catch`. N'oubliez pas de passer en paramètre le délégué qui servira à appeler la fonction précédente. Complétez le reste des paramètres selon vos besoins.

```
try
{
    TestException(0);
}
catch (Exception e)
{
    Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new String[]
    ➥{ "OK" }, 0, MessageBoxIcon.Error, new AsyncCallback(EndShowMessageBox), null);
}
```

Vous trouverez ci-dessous le code source complet de la classe illustrant la notion qui vient d'être présentée.

Utiliser des méthodes asynchrones

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

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

        this.Components.Add(new GamerServicesComponent(this));
    }

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

        try
        {
            TestException(0);
```

```
        }
        catch (Exception e)
        {
            Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
            ➥String[] { "OK" }, 0, MessageBoxIcon.Error, new
            ➥AsyncCallback(EndShowMessageBox), null);
        }
    }

    private void TestException(int i)
    {
        if (i == 0)
            throw new Exception("i vaut 0 !");
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }

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

Figure 8-6

La boîte de dialogue affiche le détail de l'exception

Grâce aux *Gamer Services*, vous accédez également à une boîte de dialogue permettant à l'utilisateur d'entrer une chaîne de caractères : il suffit d'utiliser les fonctions `BeginShowKeyboardInput ()` et `EndShowKeyboardInput()`. Le tableau ci-dessous répertorie tous les paramètres attendus par la première fonction.

Tableau 8-2 Paramètres de la fonction BeginShowKeyboardInput

Paramètre	Description
`PlayerIndex player`	Joueur concerné par la boîte de dialogue. Sous Windows, il ne peut s'agit que du joueur un.
`String title`	Titre de la boîte de dialogue.
`String text`	Contenu textuel de la boîte de dialogue.
`String defaultText`	Texte à afficher dans la boîte de dialogue lorsque celle-ci s'ouvre.
`AsyncCallback callback`	La méthode à appeler une fois que l'opération asynchrone est terminée.
`Object state`	Un objet créé par l'utilisateur pour identifier l'appel à la méthode.

Ci-dessous, vous retrouvez le code source d'une classe exemple utilisant ce type de boîte de dialogue.

Utiliser la fenêtre KeyboardInput

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

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

        this.Components.Add(new GamerServicesComponent(this));
    }

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

        Guide.BeginShowKeyboardInput(PlayerIndex.One, "Entrez un message", "Entrez
        ➥un message pour cet exemple", "", new AsyncCallback(EndShowKeyboardInput),
        ➥null);
    }

    private void EndShowKeyboardInput(IAsyncResult result)
    {
        string userInput = Guide.EndShowKeyboardInput(result);
    }

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

Figure 8-7

Le joueur peut aisément écrire un message dans cette boîte de dialogue

La GamerCard : la carte d'identité du joueur

Vous pouvez afficher les informations sur un joueur connecté grâce à la méthode `ShowGamerCard ()` de la classe `Guide`. Elle attend comme paramètre un `PlayerIndex`, ainsi qu'un objet de type `Gamer`. Vous pouvez récupérer la liste des joueurs connectés grâce à la collection `SignedInGamers` de la classe `SignedInGamer`.

La classe ci-dessous invite le joueur à se connecter puis, lorsque celui-ci pressera la touche A de son clavier, elle affichera des informations le concernant. N'oubliez pas d'ajouter un bloc `try … catch`, notamment au cas où aucun joueur ne serait connecté. Le test sur la propriété `IsVisible` permet de ne pas demander un nouvel affichage du Guide si celui-ci est déjà présent à l'écran.

Afficher la GamerCard du joueur connecté

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

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

        this.Components.Add(new GamerServicesComponent(this));
    }

    protected override void Initialize()
    {
        base.Initialize();
        Guide.ShowSignIn(1, true);
    }
```

```
    protected override void Update(GameTime gameTime)
    {
        if (Keyboard.GetState().IsKeyDown(Keys.A) && !Guide.IsVisible)
        {
            try
            {
                Guide.ShowGamerCard(PlayerIndex.One, SignedInGamer
                ➥.SignedInGamers[0]);
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
                ➥string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }

        base.Update(gameTime);
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }

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

Figure 8-8

À partir de ce panneau, le joueur peut administrer son compte

De la même manière, vous pouvez permettre au joueur d'écrire un message avec `ShowComposeMessage()`, d'afficher la liste de ses amis avec `ShowFriends()`, d'afficher la fenêtre d'ajout d'un ami avec `ShowFriendRequest()`, etc. La liste est encore longue, à vous de l'explorer et d'utiliser ces méthodes selon vos besoins.

Version démo

Si vous avez décidé de créer un jeu que vous vendrez ensuite à la communauté, vous pouvez lui ajouter un mode de démonstration (*trial mode*). Bien évidemment, dans ce mode, vous limiterez la liberté du joueur vis-à-vis des possibilités offertes par le jeu. Pour que vous, développeur, puissiez tester ce mode démonstration, vous devez définir le booléen `SimulateTrialMode` à `true`. Vous pouvez ensuite savoir si le jeu est exécuté en mode démonstration via le booléen `IsTrialMode`.

Essayez la classe suivante.

Vérifier s'il s'agit d'une version d'essai

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

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

        this.Components.Add(new GamerServicesComponent(this));
    }

    protected override void Update(GameTime gameTime)
    {
        if (Guide.IsTrialMode)
            Window.Title = "Trial Mode";
        else
            Window.Title = "Full Mode";

        base.Update(gameTime);
    }

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

Dans la barre de titre de la fenêtre, vous pouvez lire :

```
Full Mode
```

Maintenant, modifiez la classe de manière à ce que la propriété `SimulateTrialMode` soit définie à `true`.

Simuler une version d'essai

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

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

        this.Components.Add(new GamerServicesComponent(this));
    }

    protected override void Initialize()
    {
        base.Initialize();
        Guide.SimulateTrialMode = true;
    }

    protected override void Update(GameTime gameTime)
    {
        if (Guide.IsTrialMode)
            Window.Title = "Trial Mode";
        else
            Window.Title = "Full Mode";

        base.Update(gameTime);
    }

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

Vous pouvez à présent lire :

```
Trial Mode
```

La sauvegarde en pratique : réalisation d'un éditeur de cartes

Pour vous aider à bien comprendre comment mettre en pratique une solution de sauvegarde et de chargement de données dans le jeu, nous allons maintenant développer un petit éditeur de cartes en 2D.

Identifier les besoins

Commençons par définir les besoins auxquels devra répondre l'éditeur de cartes.

Lorsque l'utilisateur appuiera sur la touche C de son clavier, une carte vierge sera générée. L'utilisateur déplacera ensuite un curseur sur la carte grâce aux touches fléchées du clavier. Il pourra ensuite choisir la texture à utiliser sur chacune des cases de la carte grâce aux touches fonctions (F1, F2, etc.). Lorsque l'utilisateur pressera la touche S du clavier, la carte sera sauvegardée. Il pourra ensuite fermer le programme, le rouvrir et recharger sa carte en appuyant sur la touche L.

La première chose à faire est de créer un projet partagé de type Windows Game Library. Dans ce projet, créez la classe correspondant aux cases de la carte. La classe s'appellera `Tile`. Elle contiendra une texture, une chaîne de caractères correspondant au nom de fichier de la texture et enfin, une paire de coordonnées correspondant à sa position logique sur la carte.

```
public class Tile
{
    Texture2D texture;

    string assetName;

    public string AssetName
    {
        get
        {
            return assetName;
        }
        set
        {
            assetName = value;
        }
    }

    Vector2 position;

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

```
    public Tile()
    {
    }

    public Tile(string assetName, Vector2 position)
    {
        this.assetName = assetName;
        this.position = position;
    }

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

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, new Vector2(position.X * texture.Width, position.Y
        ➥* texture.Height), Color.White);
    }
}
```

Occupons-nous maintenant de la classe qui correspond à la carte. Cette classe contiendra simplement un tableau de tableau de `Tile.public class Map`.

```
{
    Tile[][] tiles;

    public Tile[][] Tiles
    {
        get
        {
            return tiles;
        }
        set
        {
            tiles = value;
        }
    }

    public Map()
    {
    }

    public Map(Vector2 size)
    {
        tiles = new Tile[(int)size.Y][];
        for(int i = 0; i < tiles.Length; i ++)
            tiles[i] = new Tile[(int)size.X];
    }
```

```
    public void LoadContent(ContentManager Content)
    {
        for (int y = 0; y < tiles.Length; y++)
        {
            for (int x = 0; x < tiles[0].Length; x++)
            {
                tiles[y][x].LoadContent(Content);
            }
        }
}

    public void Draw(SpriteBatch spriteBatch)
    {
        for (int y = 0; y < tiles.Length; y++)
        {
            for (int x = 0; x < tiles[0].Length; x++)
            {
                tiles[y][x].LoadContent(Content);
            }
        }
    }
}
```

La dernière classe à préparer est celle du curseur. Il s'agit d'un simple sprite qui devra se déplacer selon les entrées clavier de l'utilisateur.

```
public class Cursor
{
    Texture2D texture;

    Vector2 position;

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

    KeyboardState keyboardState;

    KeyboardState lastKeyboardState;

    public Cursor()
    {
        position = new Vector2(0, 0);
    }
```

```
    public void LoadContent(ContentManager Content)
    {
        texture = Content.Load<Texture2D>("cursor");
    }

    public void Update(GameTime gameTime, Vector2 mapSize)
    {
        lastKeyboardState = keyboardState;

        keyboardState = Keyboard.GetState();

        if (keyboardState.IsKeyDown(Keys.Left) && lastKeyboardState
        ➥.IsKeyUp(Keys.Left) && position.X > 0)
        {
            position.X--;
        }
        if (keyboardState.IsKeyDown(Keys.Right) && lastKeyboardState
        ➥.IsKeyUp(Keys.Right) && position.X < mapSize.X-1)
        {
            position.X++;
        }
        if (keyboardState.IsKeyDown(Keys.Up) && lastKeyboardState.IsKeyUp(Keys.Up)
        ➥&& position.Y > 0)
        {
            position.Y--;
        }
        if (keyboardState.IsKeyDown(Keys.Down) && lastKeyboardState
        ➥.IsKeyUp(Keys.Down) && position.Y < mapSize.Y-1)
        {
            position.Y++;
        }
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, new Vector2(position.X * texture.Width, position.Y
        ➥* texture.Height), Color.White);
    }
}
```

Pour terminer, référencez le projet de type Windows Game Library dans le projet principal. Pour ce faire, dans l'explorateur de solution faites un clic droit sur le conteneur de Références du projet principal, puis choisissez Ajouter une référence. Allez ensuite sur l'onglet Projet et choisissez le projet partagé.

Nous avons maintenant tous les outils en main pour mettre en place l'éditeur de cartes. Ajoutons des objets de type `Map` et `Cursor` à la classe principale du projet. Si l'utilisateur appuie sur la touche C du clavier, nous initialisons la carte et le curseur. Dans le cas de la carte, on charge la même texture pour toutes les cases de la carte.

Ensuite, si la carte et le curseur existent bien, nous vérifierons si une touche Fx est pressée et modifions la texture de la case concernée en conséquence.

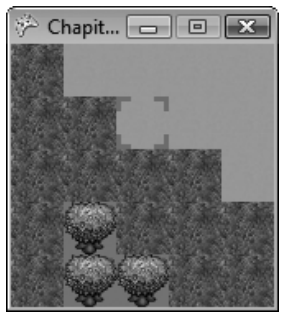
```
public class Chapitre_8 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    KeyboardState keyboardState;
    KeyboardState lastKeyboardState;

    Map map;
    Cursor cursor;

    Vector2 mapSize = new Vector2(5, 5);

    public Chapitre_8()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";

        graphics.PreferredBackBufferHeight = 160;
        graphics.PreferredBackBufferWidth = 160;
    }

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

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

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        lastKeyboardState = keyboardState;
        keyboardState = Keyboard.GetState();

        if(keyboardState.IsKeyDown(Keys.C) && lastKeyboardState.IsKeyUp(Keys.C))
        {
            map = new Map();

            map = new Map(mapSize);
```

```
            for (int y = 0; y < map.Tiles.Length; y++)
            {
                for (int x = 0; x < map.Tiles[0].Length; x++)
                {
                    map.Tiles[y][x] = new Tile("grass", new Vector2(x, y));
                }
            }

            map.LoadContent(Content);

            cursor = new Cursor();
            cursor.LoadContent(Content);
        }
        else if (keyboardState.IsKeyDown(Keys.S) && lastKeyboardState
        ➥.IsKeyUp(Keys.S) && map != null)
        {

        }
        else if (keyboardState.IsKeyDown(Keys.L) && lastKeyboardState
        ➥.IsKeyUp(Keys.L))
        {

        }

        if (cursor != null && map != null)
        {
            cursor.Update(gameTime, mapSize);

            if (keyboardState.IsKeyDown(Keys.F1) && lastKeyboardState
            ➥.IsKeyUp(Keys.F1))
            {
                map.Tiles[(int)cursor.Position.Y][(int)cursor.Position.X].AssetName
                ➥= "grass";
                map.Tiles[(int)cursor.Position.Y][(int)cursor.Position.X]
                ➥.LoadContent(Content);
            }
            else if (keyboardState.IsKeyDown(Keys.F2) && lastKeyboardState
            ➥.IsKeyUp(Keys.F2))
            {
                map.Tiles[(int)cursor.Position.Y][(int)cursor.Position.X].AssetName
                ➥= "tree";
                map.Tiles[(int)cursor.Position.Y][(int)cursor.Position.X]
                ➥.LoadContent(Content);
            }
            else if (keyboardState.IsKeyDown(Keys.F3) && lastKeyboardState
            ➥.IsKeyUp(Keys.F3))
            {
                map.Tiles[(int)cursor.Position.Y][(int)cursor.Position.X].AssetName
                ➥= "sand";
                map.Tiles[(int)cursor.Position.Y][(int)cursor.Position.X]
                ➥.LoadContent(Content);
            }
        }
    }
```

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

        spriteBatch.Begin();

        if (map != null)
            map.Draw(spriteBatch);

        if (cursor != null)
            cursor.Draw(spriteBatch);

        spriteBatch.End();
    }
```

Vous pouvez à présent exécuter l'éditeur de cartes et vous amuser un peu avec (figure 8-9).

Figure 8-9
L'éditeur de cartes



Chemin du dossier de jeu

Vous pouvez récupérer le chemin du dossier du jeu grâce à la propriété statique `TitleLocation` de l'objet `StorageContainer`. L'exemple suivant affiche ce chemin à la place du titre du jeu.

Récupérer le chemin du dossier de jeu

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

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

    protected override void Initialize()
    {
        Window.Title = StorageContainer.TitleLocation;
        base.Initialize();
    }
```

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

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();

        base.Update(gameTime);
    }

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

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

Figure 8-10
Le chemin complet vers le dossier du jeu

Gérer les dossiers

C'est l'espace de noms `System.IO` qui contient les classes permettant de gérer fichiers et répertoires. Le tableau ci-dessous présente quelques-unes des méthodes statiques mises à disposition par la classe `Directory`. Il en existe beaucoup d'autres, mais seules celles-ci seront utilisées dans le cas présent.

Tableau 8-3 Méthodes de la classe Directory

Méthode	Description
`Bool Exists(string path)`	Renvoie vrai si le répertoire correspondant au chemin passé en argument existe.
`DirectoryInfo CreateDirectory(string path)`	Crée tous les répertoires et sous-répertoires correspondant au chemin passé en argument.
`Void Delete(string path)`	Supprime le répertoire correspondant au chemin passé en argument.

L'exemple suivant utilise ces trois méthodes. Si le joueur presse la touche C et que le répertoire `test` n'existe pas dans le dossier de l'utilisateur (on récupère le chemin via la propriété `Path` de l'objet de type `StorageContainer`), on le crée. Si le joueur presse la touche D et que le répertoire existe, on le supprime.

Gérer les dossiers dans le répertoire de l'utilisateur

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

    StorageDevice device;
    StorageContainer container;

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

        this.Components.Add(new GamerServicesComponent(this));
    }

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

        Guide.BeginShowStorageDeviceSelector(new AsyncCallback(GetDevice), null);
    }

    protected override void Update(GameTime gameTime)
    {
        if (Keyboard.GetState().IsKeyDown(Keys.C))
        {
            try
            {
                if(!Directory.Exists(Path.Combine(container.Path, "test")))
                    Directory.CreateDirectory(Path.Combine(container.Path, "test"));
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
                ➥string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }

        else if (Keyboard.GetState().IsKeyDown(Keys.D))
        {
            try
            {
```

```
                if (Directory.Exists(Path.Combine(container.Path, "test")))
                    Directory.Delete(Path.Combine(container.Path, "test"));
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
                ➥string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }

        base.Update(gameTime);
    }

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

        base.Draw(gameTime);
    }

    private void GetDevice(IAsyncResult result)
    {
        if (result.IsCompleted)
        {
            try
            {
                device = Guide.EndShowStorageDeviceSelector(result);
                container = device.OpenContainer("ChapitreHuit");
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
                ➥string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }
}
```

Dans cet exemple, vous remarquez que la méthode `Combine` de la classe `Path` est utilisée pour créer le chemin du répertoire. Vous n'avez donc pas à vous soucier des séparateurs `/` ou `\`.

La classe `Directory` possède d'autres méthodes qui pourront vous servir dans le développement des jeux. Ainsi, la méthode `GetFiles()` retournera un tableau de chaînes de caractères correspondant à la liste des fichiers présents dans un répertoire. L'une de ses surcharges vous permet également d'ajouter un filtre sur les noms de fichiers à récupérer.

Manipuler les fichiers

La manipulation de fichiers se fera grâce à la classe `File`. Elle possède un très grand nombre de méthodes, mais nous ne les présenterons pas toutes ici. Pour commencer, cette classe dispose, comme la classe `Directory`, d'une méthode `Exists()` fonctionnant exactement de la même manière.

Pour créer un fichier, vous pouvez utiliser la méthode `Create ()`. Elle attend comme seul paramètre le chemin du fichier à créer. Cependant, elle dispose de surcharges vous permettant de définir la taille du buffer qui sera utilisé pour la lecture et l'écriture, la méthode de création du fichier ou encore les options de sécurité à appliquer au fichier. La méthode retourne un objet de type `FileStream`. Si vous ne souhaitez pas modifier tout de suite le contenu du fichier, fermez-le directement avec la fonction `Close()`.

Bonne pratique

Une bonne habitude à prendre en programmation est de toujours fermer les fichiers que vous avez créés ou ouverts lorsque vous n'en avez plus besoin.

Vous pouvez copier un fichier grâce à la méthode `Copy ()`. Elle prend en paramètre le chemin du fichier à copier et le chemin du fichier de destination. Elle possède une surcharge qui vous propose de définir si le fichier de destination doit être écrasé lorsqu'il existe déjà.

Pour supprimer un fichier, utilisez simplement la méthode `Delete ()` qui attend comme argument le chemin du fichier à supprimer.

La classe ci-dessous utilise toutes ces méthodes. Si le joueur appuie sur la touche C et que le fichier `test.sav` n'existe pas dans le dossier de l'utilisateur, il est créé ; s'il existe, il est copié en `test_copy.sav` en écrasant le fichier de destination. Si le joueur appuie sur D et que `test.sav` existe, il est supprimé.

Gérer les fichiers dans le répertoire de l'utilisateur

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

    StorageDevice device;
    StorageContainer container;
```

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

    this.Components.Add(new GamerServicesComponent(this));
}

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

    Guide.BeginShowStorageDeviceSelector(new AsyncCallback(GetDevice), null);
}

protected override void Update(GameTime gameTime)
{
    if (Keyboard.GetState().IsKeyDown(Keys.C))
    {
        try
        {
            if(!File.Exists(Path.Combine(container.Path, "test.sav")))
                File.Create(Path.Combine(container.Path, "test.sav")).Close();
            if(File.Exists(Path.Combine(container.Path, "test.sav")))
                File.Copy(Path.Combine(container.Path, "test.sav")
                ➥,Path.Combine(container.Path, "test_copy.sav"), true);
        }
        catch (Exception e)
        {
            Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message,
            ➥new string[] { "Ok" }, 0, MessageBoxIcon.Error, new
            ➥AsyncCallback(EndShowMessageBox), null);
        }
    }

    else if (Keyboard.GetState().IsKeyDown(Keys.D))
    {
        try
        {
            if (File.Exists(Path.Combine(container.Path, "test.sav")))
                File.Delete(Path.Combine(container.Path, "test.sav"));
        }
        catch (Exception e)
        {
            Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message,
            ➥new string[] { "Ok" }, 0, MessageBoxIcon.Error, new
            ➥AsyncCallback(EndShowMessageBox), null);
        }
    }

    base.Update(gameTime);
}
```

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

        base.Draw(gameTime);
    }

    private void GetDevice(IAsyncResult result)
    {
        if (result.IsCompleted)
        {
            try
            {
                device = Guide.EndShowStorageDeviceSelector(result);
                container = device.OpenContainer("ChapitreHuit");
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message,
                ➥new string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }
}
```

Écrire dans un fichier

Pour écrire dans un fichier, commencez par récupérer un flux en écriture vers le fichier désiré. Cette opération se fera grâce à la méthode `Open ()` de la classe `File`. Le tableau 8-4 détaille les paramètres attendus par la méthode.

Tableau 8-4 Paramètres de la méthode Open

Paramètre	Description
`String path`	Chemin du fichier à ouvrir.
`FileMode mode`	Méthode d'ouverture à utiliser sur le fichier. `FileMode` est une énumération qui peut prendre plusieurs valeurs (ajout à la fin du fichier, vider le fichier, ouverture classique, etc.).
`FileAccess access`	Définit les opérations qui pourront être effectuées sur le fichier (lecture, écriture ou les deux).
`FileShare share`	Définit le type d'accès que les autres threads ont sur le fichier.

Utilisez ensuite un objet de type `StreamWriter` pour écrire dans le flux récupéré précédemment. Comme pour le flux de sortie standard (souvenez-vous de vos premiers programmes en mode console), l'écriture se fait grâce aux méthodes `Write ()` et `WriteLine ()`. N'oubliez pas ensuite de vider le buffer grâce à la méthode `Flush ()` et enfin de fermer le flux par la méthode `Close ()`.

La classe suivante met ces actions en pratique. Lorsque le joueur appuie sur la touche O, on vérifie si le fichier `test.sav` existe dans le répertoire de l'utilisateur. Si c'est le cas, on se place à la fin du fichier, sinon il est créé. Il ne reste plus qu'à ajouter une ligne au flux, puis à fermer le fichier proprement. Après avoir testé le code, vous lirez dans le fichier la phrase suivante :

```
XNA's Not Acronymed
```

Ouvrir un fichier et écrire dans un fichier

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

    StorageDevice device;
    StorageContainer container;

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

        this.Components.Add(new GamerServicesComponent(this));
    }

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

        Guide.BeginShowStorageDeviceSelector(new AsyncCallback(GetDevice), null);
    }

    protected override void Update(GameTime gameTime)
    {
        if (Keyboard.GetState().IsKeyDown(Keys.O))
        {
            try
            {
                FileStream file = File.Open(Path.Combine(container.Path,
                ➥"test.sav"), FileMode.Append);
                StreamWriter fileWriter = new StreamWriter(file);
                fileWriter.WriteLine("XNA's Not Acronymed");
                fileWriter.Flush();
                fileWriter.Close();
            }
```

```
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message, new
                ➥string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }

        base.Update(gameTime);
    }

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

        base.Draw(gameTime);
    }

    private void GetDevice(IAsyncResult result)
    {
        if (result.IsCompleted)
        {
            try
            {
                device = Guide.EndShowStorageDeviceSelector(result);
                container = device.OpenContainer("ChapitreHuit");
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message,
                ➥new string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }
}
```

Lire un fichier

C'est une bonne chose de savoir écrire, c'est encore mieux de pouvoir lire ce qui a été écrit. Ainsi, pour lire des données écrites en clair, commencez par récupérer un flux `FileStream` que vous exploiterez grâce aux méthodes `Read()`, `ReadLine()`, `ReadToEnd()`, etc., d'un objet `StreamReader`.

Lire le contenu d'un fichier

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

    StorageDevice device;
    StorageContainer container;

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

        this.Components.Add(new GamerServicesComponent(this));
    }

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

        Guide.BeginShowStorageDeviceSelector(new AsyncCallback(GetDevice), null);
    }

    protected override void Update(GameTime gameTime)
    {
        if (Keyboard.GetState().IsKeyDown(Keys.L) && File.Exists
        ➥(Path.Combine(container.Path, "test.sav")))
        {
            try
            {
                FileStream file = File.Open(Path.Combine(container.Path,
                ➥"test.sav"), FileMode.Open);
                StreamReader fileReader = new StreamReader(file);
                Window.Title = fileReader.r();
                fileReader.Close();
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message,
                ➥new string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }

        base.Update(gameTime);
    }

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

        base.Draw(gameTime);
    }
```

```
    private void GetDevice(IAsyncResult result)
    {
        if (result.IsCompleted)
        {
            try
            {
                device = Guide.EndShowStorageDeviceSelector(result);
                container = device.OpenContainer("ChapitreHuit");
            }
            catch (Exception e)
            {
                Guide.BeginShowMessageBox(PlayerIndex.One, "Erreur", e.Message,
                ➥new string[] { "Ok" }, 0, MessageBoxIcon.Error, new
                ➥AsyncCallback(EndShowMessageBox), null);
            }
        }
    }

    private void EndShowMessageBox(IAsyncResult result)
    {
        if (result.IsCompleted)
            Guide.EndShowMessageBox(result);
    }
}
```

Sérialiser des données

Reprenons maintenant l'éditeur de cartes que nous avons commencé à réaliser plus tôt dans ce chapitre. Nous allons à présent voir comment sérialiser les données : dans un premier temps en binaire, puis en XML.

Pour qu'une classe puisse être sérialisée, il faut lui ajouter l'attribut `[Serializable]`. Ajoutons-le aux classes `Map` et `Tile`.

```
[Serializable]
public class Map
{ … }

[Serializable]
public class Tile
{ … }
```

Nous ne devons pas sérialiser la texture de chaque `Tile`, mais seulement son nom. Pour qu'un attribut ne soit pas sérialisé, il faut le marquer comme `[NonSerialized]`.

```
[NonSerialized]
Texture2D texture;
```

Pour sérialiser les données en binaire, utilisez l'espace de noms `System.Runtime.Serialization.Formatters.Binary`. Ensuite, si le joueur presse la touche S, vérifiez l'existence du fichier de destination et créez-le si nécessaire, sinon ouvrez-le (dans l'exemple, il est tronqué à l'ouverture). Utilisez ensuite un objet de type `BinaryFormatter` et sa méthode `Serialize()` à laquelle vous devrez passer le flux où écrire et l'objet à sérialiser. Pour terminer, fermez le flux.

```
else if (keyboardState.IsKeyDown(Keys.S) && lastKeyboardState.IsKeyUp(Keys.S) && map
➥!= null)
{
    FileStream file;

    if (!File.Exists(Path.Combine(container.Path, "test.sav")))
        file = File.Create(Path.Combine(container.Path, "test.sav"));
    else
        file = File.Open(Path.Combine(container.Path, "test.sav"),
        ➥FileMode.Truncate);

    BinaryFormatter serializer = new BinaryFormatter();
    serializer.Serialize(file, map);
    file.Close();
}
```

Si vous vous rendez dans le répertoire de sauvegarde du jeu, vous verrez le fichier `test.sav`. Vous pourrez ensuite l'ouvrir et constater que les données ne sont pas du tout intelligibles (figure-8-11).

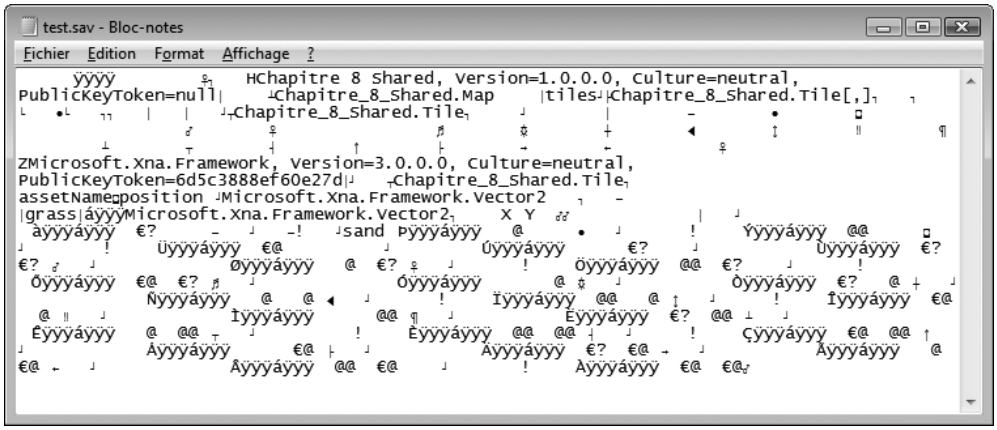


Figure 8-11

Les deux fichiers créés par la classe exemple

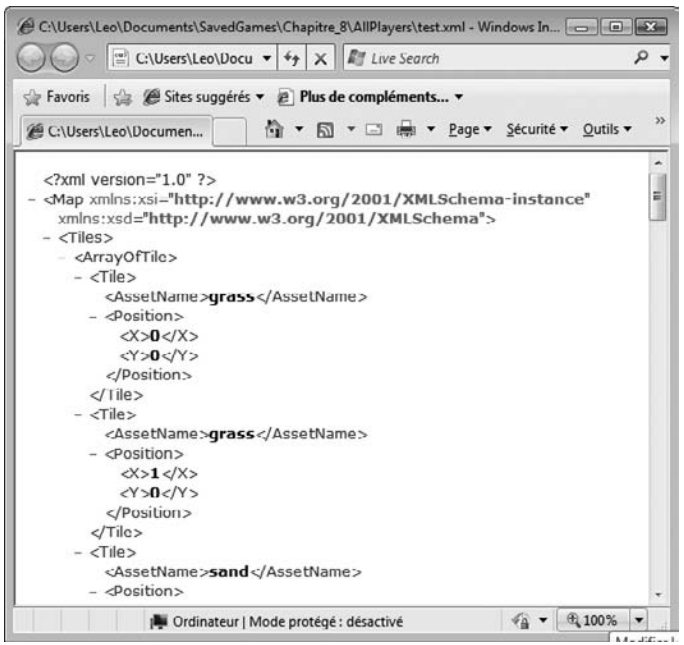
Vous pouvez également sérialiser les données sous forme de fichier XML. L'espace de noms à utiliser est `System.XML.Serialization`. Une classe peut être sérialisée en XML uniquement si elle possède un constructeur sans paramètres. Il faut également que tous les attributs à sérialiser possèdent un accesseur en lecture et un autre en écriture. S'il y a des paramètres que la sérialisation doit ignorer, utilisez l'attribut `[XmlIgnore]`.

Du côté de la classe `Chapitre_8`, instanciez un objet `XmlSerializer` auquel vous devrez passer le type d'objet à sérialiser. Servez-vous pour cela de l'opérateur `typeof()`. Enfin, comme pour le `BinaryFormatter`, utilisez la méthode `Serialize()` qui attend les mêmes arguments. N'oubliez pas de fermer le flux après utilisation.

```
else if (keyboardState.IsKeyDown(Keys.S) && lastKeyboardState.IsKeyUp(Keys.S) && map
➥!= null)
{
    FileStream file;

    if (!File.Exists(Path.Combine(container.Path, "test.xml")))
        file = File.Create(Path.Combine(container.Path, "test.xml"));
    else
        file = File.Open(Path.Combine(container.Path, "test.xml"),
        ➥FileMode.Truncate);

    XmlSerializer serializer = new XmlSerializer(typeof(Map));
    serializer.Serialize(file, map);
    file.Close();
}
```

Vous pouvez tester l'éditeur en créant une carte et en la sauvegardant : un fichier XML est bel et bien généré (figure-8-12).

Figure 8-12
Le fichier XML représentant la carte



Désérialiser des données

Vous savez maintenant comment sauvegarder des données, il ne vous reste plus qu'à apprendre à les charger ! Ce processus se fera très simplement en utilisant la méthode `Deserialize ()` de l'objet `BinaryFormatter`. Il faut ensuite effectuer un *cast* (c'est-à-dire une modification de type) de l'objet retourné par la fonction pour récupérer un objet `Map`.

Après avoir récupéré l'objet `Map`, n'oubliez pas d'appeler la méthode `LoadContent()` pour charger les textures.

```
else if (keyboardState.IsKeyDown(Keys.L) && lastKeyboardState.IsKeyUp(Keys.L))
{
    FileStream file = File.Open(Path.Combine(container.Path, "test.sav"),
    ➥FileMode.Open);
    BinaryFormatter serializer = new BinaryFormatter();
    map = (Map)serializer.Deserialize(file);
    file.Close();

    map.LoadContent(Content);

    cursor = new Cursor();
    cursor.LoadContent(Content);
}
```

Voyons à présent la désérialisation XML. Dans la classe `Chapitre_8`, vous utiliserez la méthode `Deserialize()` à laquelle vous passerez le flux à traiter. Transformez ensuite l'objet retourné par la méthode en un objet de type `Map` avec la méthode du casting.

Attention, lorsque vous utilisez la sérialisation XML : tout n'est pas sérialisable. C'est pour cette raison que nous avons utilisé un tableau de tableau de `Tile` plutôt qu'un tableau à deux dimensions.

```
else if (keyboardState.IsKeyDown(Keys.L) && lastKeyboardState.IsKeyUp(Keys.L))
{
    FileStream file = File.Open(Path.Combine(container.Path, "test.xml"),
    ➥FileMode.OpenOrCreate);
    XmlSerializer deserializer = new XmlSerializer(typeof(Map));
    map = (Map)deserializer.Deserialize(file);
    file.Close();

    map.LoadContent(Content);

    cursor = new Cursor();
    cursor.LoadContent(Content);
}
```

L'éditeur de cartes est maintenant prêt : vous pouvez créer, sauvegarder et charger des cartes en utilisant soit la sérialisation binaire, soit la sérialisation XML.

Les Content Importers, une solution compatible avec la Xbox 360

Si vous essayez de désérialiser une carte comme vous venez de le voir dans un projet exécuté sur Xbox 360, vous vous rendrez compte que cela ne fonctionne pas. Sur cette plate-forme, vous ne pouvez charger que des fichiers .xnb et, pour générer de tels fichiers à partir des types personnalisés (comme la classe `Map`), vous devrez créer un `ContentImporter`.

1. Ajoutez un projet de type `ContentPipelineExtension` à la solution.
2. Ajoutez à ce nouveau projet une référence vers le projet partagé qui contient les classes `Map`, `Tile` et `Cursor`.
3. Ajoutez ensuite au projet un nouvel élément de type `Content Type Writer` que vous nommerez `TileWriter`.
4. Renseignez le type de données concernées par le `Content Type Writer`.

```
using TWrite = Chapitre_8_Shared.Tile;
```

5. Il faut ensuite compléter la méthode `Write()`. Le principe est simple : vous disposez de l'objet (`value`) et vous écrivez le contenu de ses différents paramètres sur l'objet `output`.
6. Écrivez le nom de sa texture en utilisant la méthode `Write()` de l'objet `output`. De la même manière, occupez-vous de l'attribut position.

```
protected override void Write(ContentWriter output, TWrite value)
{
    output.Write(value.AssetName);

    output.Write(value.Position);
}
```

À présent, les données de type `Tile` peuvent être sérialisées vers un fichier .xnb.

Intéressons-nous maintenant à leur lecture.

1. Dans le projet de bibliothèque de classes, ajoutez un nouvel élément de type `Content Type Reader`.
2. Comme pour le `Writer`, commencez par renseigner le type de données concernées.

```
Using Tread = Chapitre_8_Shared.Tile;
```

3. À présent, c'est la méthode `Read()` que vous allez devoir compléter. Commencez par créer une nouvelle instance de `Tile`. Ensuite, récupérez la valeur des deux attributs grâce aux méthodes `ReadString()` et `ReadVector2()`. Enfin, n'oubliez pas de retourner l'objet recréé.

```
protected override TRead Read(ContentReader input, TRead existingInstance)
{
    existingInstance = new TRead();

    existingInstance.AssetName = input.ReadString();

    existingInstance.Position = input.ReadVector2();

    return existingInstance;
}
```

4. Il reste une dernière chose à faire pour la classe `Tile`. Retournez sur le `Writer` et complétez la méthode `GetRuntimeReader()`.

```
public override string GetRuntimeReader(TargetPlatform targetPlatform)
{
    return typeof(Chapitre_8_Shared.TileReader).AssemblyQualifiedName;
}
```

C'est fini pour la classe `Tile`.

5. Répétez ces opérations pour la classe `Map`. Il y a cependant un petite différence : vous ne pouvez pas utiliser la méthode `Write()` puisque vous avez défini le type de l'attribut `Tiles` (un tableau de tableau de `Tile`). Vous devrez utiliser la méthode générique `WriteObject()`.

```
protected override void Write(ContentWriter output, TWrite value)
{
    output.WriteObject<Chapitre_8_Shared.Tile[][]>(value.Tiles);
}
```

6. Le raisonnement est le même que pour le `Reader` : vous devez utiliser la méthode `ReadObject()`.

```
protected override TRead Read(ContentReader input, TRead existingInstance)
{
    existingInstance = new Map();

    existingInstance.Tiles = input.ReadObject<Tile[][]>();

    return existingInstance;
}
```

7. N'oubliez pas la méthode `GetRuntimeReader()`.

```
public override string GetRuntimeReader(TargetPlatform targetPlatform)
{
    return typeof(Chapitre_8_Shared.MapReader).AssemblyQualifiedName;
}
```

8. Le `ContentImporter` est fin prêt. Cependant, vous devez lui fournir un fichier XML en entrée. Pour créer ce fichier, commencez par ajouter une nouvelle référence au projet principal vers l'assembly `Microsoft.XNA.Framework.Content.Pipeline`. Ajoutez ensuite une directive `using` à la classe `Chapitre_8`.

```
using Microsoft.Xna.Framework.Content.Pipeline.Serialization.Intermediate;
```

9. Pour générer un fichier XML correspondant à la carte, vous utiliserez un objet de type `XmlWritter` qui correspondra au flux de sortie. Vous sérialisez ensuite les données grâce à la méthode statique `Serialize` de la classe `IntermediateSerializer()`.

```
XmlWriterSettings xmlSettings = new XmlWriterSettings();
xmlSettings.Indent = true;

using (XmlWriter xmlWritter = XmlWriter.Create("map.xml", xmlSettings))
{
    IntermediateSerializer.Serialize(xmlWritter, map, null);
}
```

Vous pouvez exécuter l'éditeur, créer une carte et la sauvegarder.

10. Rendez-vous dans le répertoire de sortie du projet et vérifiez l'existence du fichier `map.xml`.
11. À présent, ajoutez une référence vers le projet d'extension du `Content Pipeline` au projet de contenu. Ajoutez ensuite le fichier `map.xml` au projet de contenu comme vous ajouteriez n'importe quel autre type de ressource.

À ce stade, si vous compilez le projet, vous remarquez que la génération d'un fichier .xnb correspondant à la carte. Il ne vous reste plus qu'à le charger comme vous le feriez avec n'importe quel type de ressource de base de XNA.

```
map = Content.Load<Map>("map");
```

En résumé

Dans ce chapitre vous avez découvert :

- les différents types d'espace de stockage disponibles avec XNA ;
- ce qu'est un fichier XML et comment en créer un ;
- comment manipuler dossiers et fichiers en C# ;
- ce qu'est la sérialisation, qu'elle soit binaire ou XML, et comment l'utiliser dans le cadre d'un projet sous XNA.

9

Pathfinding : programmer les déplacements des personnages

Comment le héros d'un jeu vidéo fait-il pour trouver rapidement la sortie d'un labyrinthe alors que le joueur a simplement cliqué sur la sortie de celui-ci ? Après une introduction sur l'intelligence artificielle et les algorithmes de recherche de chemin, vous découvrirez en détail l'algorithme A*, puis vous apprendrez à le mettre en œuvre. À la fin de chapitre vous serez donc capable de répondre à cette question, et même mieux, vous apprendrez à programmer le comportement du héros qui doit sortir du labyrinthe.

Les enjeux de l'intelligence artificielle

Pour rendre un jeu intéressant pour le joueur, vous allez par exemple devoir y introduire des mécanismes liés à l'intelligence artificielle.

Le terme intelligence artificielle est difficilement définissable, puisque même les experts du domaine ne s'accordent pas sur sa signification. Certains diront qu'une machine est intelligente dès lors qu'elle sait reproduire le comportement d'un être humain, par exemple en accomplissant des tâches que l'homme sait lui aussi accomplir grâce à son intelligence. À cela, d'autres répondront qu'on ne peut pas parler ici d'intelligence, mais de simple copie du comportement. Il y a de très nombreuses choses à dire sur ce débat, puisque celui-ci relève même pour certaines personnes du domaine de l'éthique. Mais là n'est pas l'objectif de ce chapitre !

La recherche sur l'intelligence artificielle évolue rapidement. De nos jours, un champion du monde d'échecs se fait battre par un ordinateur. Peut-être que dans un avenir proche, les choses vont prendre une dimension encore plus grande. Des scientifiques travaillent sur un système capable de représenter des pensées en images, d'autres ont créé un robot qui fonctionne avec un cerveau contrôlé par des neurones de rat. Le futur présenté dans les films de science-fiction de ces 30 dernières années semble arriver bien plus vite que n'importe qui aurait pu l'imaginer.

XNA ne propose pas de classes ou de fonctions prêtes à l'emploi en rapport avec n'importe quel domaine de l'intelligence artificielle. C'est à vous, développeur, de programmer pour les jeux les algorithmes qui vous intéressent.

Comprendre le pathfinding

En programmation de jeu vidéo, on appelle *pathfinding* (recherche de chemin, en français) le processus de détermination du chemin entre un point de départ et un point d'arrivée. Le tableau 9-1 présente quelques algorithmes de recherche de chemin.

Tableau 9-1 Algorithmes de recherche de chemin

Nom	Description
Dijkstra	Il retourne le meilleur chemin. Il est utilisé par exemple dans certains protocoles de routage réseau.
Viterbi	Il permet de corriger les erreurs survenues lors d'une transmission via un canal bruité.
A* (prononcez « A Star »)	Il ne retourne pas forcément la meilleure solution, mais c'est un bon compromis entre pertinence du résultat et coût du calcul.

Les domaines d'application du *pathfinding* sont nombreux et variés : GPS, robotique, réseaux informatiques, jeux vidéo etc. Dans tous ces domaines, le processus de détermination du chemin à emprunter est essentiel.

Ainsi, si on considère les jeux de stratégie, des milliers de personnages peuvent se déplacer en même temps : l'ordinateur ou la console effectue sans relâche des calculs pour permettre aux différentes entités de se déplacer. Il faut donc trouver un moyen rapide d'effectuer ces calculs tout en conservant des résultats pertinents. Soyez cependant vigilant : si la solution que vous mettez en place ne retourne pas des résultats assez rapidement, le jeu sera saccadé. En vous reportant au tableau précédent, vous trouverez facilement quel algorithme est adapté aux jeux vidéo.

Effectivement, il s'agit de A*. Dans la suite de ce chapitre, nous nous intéresserons à son principe de fonctionnement, puis à sa mise en œuvre en C# avec XNA. Attention, il a été question de performances quelques lignes plus tôt : l'algorithme tel qu'il est présenté ici est loin d'être vraiment utilisable dans une grosse production. Il y a beaucoup de choses à améliorer pour réduire le temps nécessaire à son exécution. Cependant, ce chapitre n'est qu'une introduction à la recherche de chemin. Si vous vous intéressez aux bonnes

pratiques en matière d'optimisation d'algorithmes, l'Internet fourmille d'articles à ce sujet, consultez-les.

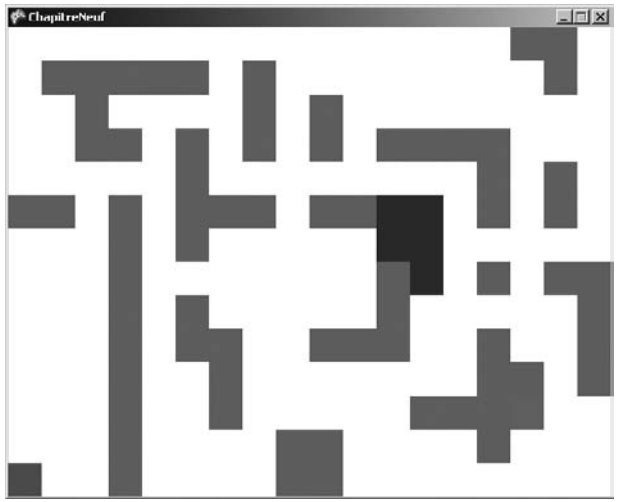
L'algorithme A* : compromis entre performance et pertinence

Dans cette section, nous allons nous pencher sur le fonctionnement de l'algorithme et verrons un exemple d'utilisation. Vous serez ainsi capable de l'utiliser dans vos jeux.

Principe de l'algorithme

Nous allons à présent travailler sur une carte représentant le niveau d'un jeu. Chaque case qui constitue la carte est appelée nœud. Le point de départ sera symbolisé par une case verte et le point d'arrivée, par une case rouge. Les cases blanches symbolisent des cases classiques, franchissables sans effort particulier. Les cases bleues symbolisent l'eau et sont plus difficiles à franchir que les cases classiques. Enfin, les cases grises symbolisent des murs parfaitement infranchissables. La figure 9-1 donne un exemple de ce type de carte.

Figure 9-1
Le type de carte qui sera utilisé



Le travail de recherche de chemin s'effectue grâce à deux listes de nœuds :

- une liste ouverte, qui contient les nœuds susceptibles de conduire le joueur au nœud de destination, c'est-à-dire les nœuds à vérifier ;
- et une liste fermée, contenant les nœuds déjà traités qui composeront le chemin final.

Intéressons-nous d'abord à la première phase de l'algorithme.

Commencez par ajouter le point de départ à la liste fermée, puisque la solution passera forcément par lui. Intéressez-vous ensuite à tous les points voisins de ce point de départ et ajoutez-les également à la liste ouverte, tout en ignorant ceux qui sont infranchissables (les murs dans notre exemple).

Déplacement oblique

Ici, les déplacements se font uniquement à la verticale et à l'horizontale. Cependant, vous pouvez bien sûr utiliser l'algorithme avec des déplacements obliques.

À présent, il nous faut définir leur parent, c'est-à-dire le nœud qui permet d'y arriver. Pour l'instant, dans notre cas, il s'agit du nœud de départ. Ensuite, choisissez le nœud que vous devrez inspecter après le nœud de départ. Pour cela, déterminez lequel de ces nœuds est le plus proche du nœud de destination.

Pour déterminer cette distance, il faut calculer ce que l'on appelle la distance de Manhattan. Cette distance correspond au nombre de déplacement horizontaux et verticaux qui devront être effectués pour aller d'un point à un autre. Si nous reprenons l'exemple des figures précédentes, cela donne dans les deux cas 16 déplacements à effectuer. Pour le nœud au-dessus du nœud de départ, il y a 15 déplacements horizontaux et un vertical. Pour le nœud à droite du nœud de départ, il y a 14 déplacements horizontaux et 2 verticaux.

Nous pouvons donc utiliser n'importe lequel des deux nœuds pour continuer notre recherche. Choisissez-en un, retirez-le de la liste ouverte et ajoutez-le à la liste fermée. Puis répétez les opérations d'analyse et de détermination du meilleur nœud voisin, jusqu'à arriver au nœud de destination. Une fois à destination, remontez de nœud en nœud grâce au nœud parent que vous avez défini pour chacun d'entre eux, jusqu'à arriver au nœud de départ qui n'aura pas de parent.

Attention

Si, à un moment donné, la liste ouverte ne contient plus de nœud, mais que vous n'êtes pas encore arrivé au nœud de destination, c'est qu'il n'existe pas de chemin possible vers ce nœud.

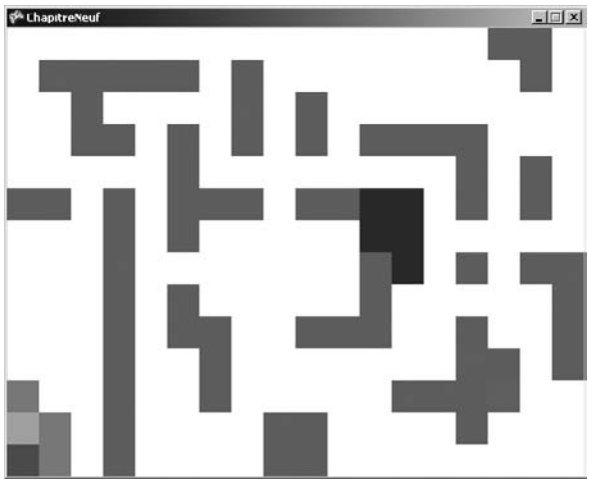


Figure 9-2
Analyse du second nœud et de ses voisins…

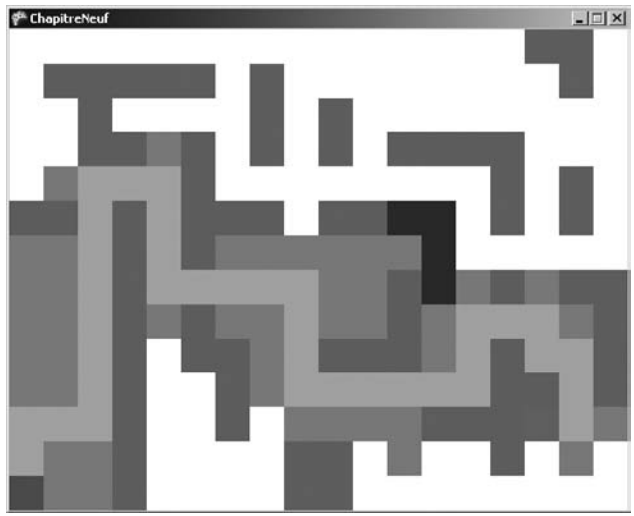


Figure 9-3
… jusqu'au nœud de destination

Dans le calcul de l'estimation de la distance vers le nœud de destination, il est possible d'ajouter la notion de coût du passage sur un nœud. Le chemin déterminé, visible sur les figures précédentes, utilise cette notion. Le passage sur une case classique n'a pas de coût particulier, alors que le passage sur une case eau (bleue) a un coût de deux. Si vous supprimez la zone d'eau ou que vous diminuez son coût, l'algorithme préférera passer par cette zone (figure 9-4). On peut également imaginer la présence d'un pont au-dessus du cours d'eau : ainsi, en ajustant bien les coûts, l'algorithme préférera emprunter le pont plutôt que d'envoyer le personnage dans l'eau.

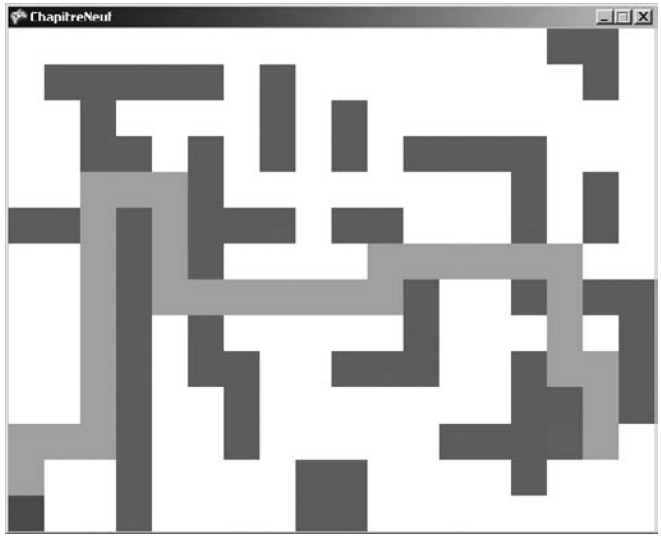


Figure 9-4
Sans la zone d'eau, l'algorithme détermine un autre chemin

Implanter l'algorithme dans un jeu de type STR

Maintenant que vous connaissez les bases du fonctionnement de l'algorithme, cette section explique comment l'appliquer à un début de jeu de stratégie en temps réel (STR).

La classe Tile et la carte

Créez un nouveau projet baptisé « ChapitreNeuf » et renommez la classe `Game1` en `ChapitreNeuf`. Ajoutez la classe `Sprite` telle qu'elle était à la fin du chapitre 6, puis créez une classe dérivée de Sprite que vous appellerez Tile.

Une carte est composée à l'écran de plusieurs objets de type `Tile`, il s'agit donc de « cases ».

Il en existe trois types :

- une case normale ;
- une case représentant de l'eau ;
- une case représentant un mur.

Créez une énumération chargée de représenter ces trois cas.

```
enum TileType
{
    Wall = -1,
    Normal = 0,
    Water = 2,
};
```

Il est également nécessaire d'ajouter une nouvelle paire de coordonnées (x, y) à cette classe. En effet, les coordonnées fournies par la classe `Sprite` représentent les coordonnées à l'écran or, ici, il faudra travailler avec les coordonnées sur une carte de jeu. Ces nouvelles coordonnées représenteront la position de l'objet `Tile` dans un tableau à deux dimensions, la valeur `y` étant la position sur la première dimension et la valeur `x`, la position sur la deuxième dimension.

Chaque case sera représentée à l'écran par un carré de 32×32 pixels qui sera coloré selon son type : gris si c'est un mur, bleu pour l'eau, vert pour le point de départ, rouge pour le point d'arrivée et enfin, orange pour les points composant le chemin retourné par l'algorithme.

La classe Tile

```
class Tile : Sprite
{
    int x;

    public int X
    {
        get { return this.x; }
        set { this.x = value ; }
    }

    int y;
```

```
    public int Y
    {
        get { return this.y; }
        set { this.x = value ; }
    }

    TileType type;

    public TileType Type
    {
        get { return type; }
    }

    public Tile(int y, int x, byte type)
        : base(new Vector2(x * 32, y * 32))
    {
        this.x = x;
        this.y = y;

        switch (type)
        {
            case 1:
                Color = Color.Gray;
                this.type = TileType.Wall;
                break;
            case 2:
                Color = Color.Blue;
                this.type = TileType.Water;
                break;
            default:
                this.type = TileType.Normal;
                break;
        }
    }
}
```

Ajoutez ensuite une classe `Map`. Elle contiendra la liste des `Tile` à afficher dans un tableau à deux dimensions. Ce tableau sera initialisé et rempli dans le constructeur de la classe via un tableau de `byte` (0 pour une case normale, 1 pour un mur et 2 pour de l'eau). Bien sûr, la classe dispose d'une méthode `Draw()` qui va parcourir le tableau et appeler la méthode du même nom de chaque `Tile`. Pour terminer, la méthode `ValidCoordinates ()` permettra de déterminer si une paire de coordonnées (x, y) correspond à un élément valide du tableau.

La classe Map

```
class Map
{
    Tile[,] tileList;

    public Tile[,] TileList
    {
        get { return tileList; }
        set { tileList = value; }
    }
```

```
    public Map(byte[,] table)
    {
        tileList = new Tile[table.GetLength(0),table.GetLength(1)];

        for (int y = 0; y < table.GetLength(0); y++)
        {
            for (int x = 0; x < table.GetLength(1); x++)
            {
                tileList[y, x] = new Tile(y, x, table[y, x]);
            }
        }
    }

    public void Draw(SpriteBatch spriteBatch)
    {
        foreach (Tile tile in tileList)
        {
            tile.Draw(spriteBatch);
        }
    }

    public bool ValidCoordinates(int x, int y)
    {
        if (x < 0)
            return false;

        if (y < 0)
            return false;

        if (x >= tileList.GetLength(1))
            return false;

        if (y >= tileList.GetLength(0))
            return false;

        return true;
    }
}
```

Lier les nœuds aux cases : la distance de Manhattan

La prochaine classe à ajouter est celle des nœuds, appelez-la `Node`. Chaque nœud doit être lié à une case `Tile`, doit pouvoir avoir un nœud parent et enfin, et doit connaître l'estimation de la distance vers le nœud de destination. Nous utiliserons ici la distance de Manhattan. Dans ce calcul, n'oubliez pas d'ajouter le coût de la case (les valeurs étant fixées dans l'énumération `TileType`). La méthode `GetPossibleNode()` va chercher les nœuds voisins (uniquement sur les axes horizontaux et verticaux) qui sont des cases valides et ne sont pas des murs. Elle renvoie ensuite une collection `List<Node>` contenant les nœuds ainsi trouvés.

La classe Node

```
class Node
{
    Tile tile;
```

```
    public Tile Tile
    {
        get { return tile; }
    }

    Node parent;

    public Node Parent
    {
        get { return parent; }
        set { parent = value; }
    }

    int estimatedMovement;

    public int EstimatedMovement
    {
        get { return estimatedMovement; }
    }

    public Node(Tile tile, Node parent, Tile destination)
    {
        this.tile = tile;
        this.parent = parent;
        this.estimatedMovement = Math.Abs(tile.X - destination.X) + Math.Abs(tile.Y
        ➥- destination.Y) + (int)tile.Type;
    }

    public List<Node> GetPossibleNode(Map map, Tile destination)
    {
        List<Node> result = new List<Node>();

        // Bottom
        if (map.ValidCoordinates(tile.X, tile.Y + 1) && map.TileList[tile.Y + 1,
        ➥tile.X].Type != TileType.Wall)
            result.Add(new Node(map.TileList[tile.Y + 1, tile.X], this,
            ➥destination));

        // Right
        if (map.ValidCoordinates(tile.X + 1, tile.Y) && map.TileList[tile.Y,
        ➥tile.X + 1].Type != TileType.Wall)
            result.Add(new Node(map.TileList[tile.Y, tile.X + 1], this,
            ➥destination));

        // Top
        if (map.ValidCoordinates(tile.X, tile.Y - 1) && map.TileList[tile.Y - 1,
        ➥tile.X].Type != TileType.Wall)
            result.Add(new Node(map.TileList[tile.Y - 1, tile.X], this,
            ➥destination));

        // Left
        if (map.ValidCoordinates(tile.X - 1, tile.Y) && map.TileList[tile.Y,
        ➥tile.X - 1].Type != TileType.Wall)
            result.Add(new Node(map.TileList[tile.Y, tile.X - 1], this,
            ➥destination));

        return result;
    }
}
```

Listes ouverte et fermée : la collection List

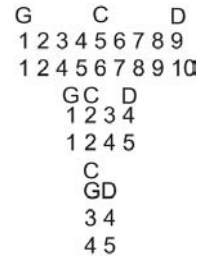
Il ne reste plus qu'une chose à préparer : une collection qui servira aux listes ouverte et fermée. Il s'agit ici de créer une collection dérivée de la collection `List<T>`, l'intérêt étant de lui donner la possibilité de retourner un objet sans connaître son index, de savoir si un nœud est déjà présent dans la liste et enfin, d'effectuer une insertion dichotomique dans la liste.

Qu'est-ce qu'une insertion dichotomique ? Dans le principe de l'algorithme, vous avez pu lire qu'à chaque itération, il faut choisir le nœud dont la distance avec le nœud de destination est la plus faible. Vous pouvez donc parcourir à chaque fois la liste ouverte à la recherche du nœud ayant la distance la plus faible ou alors entretenir une liste triée dans laquelle vous savez que le premier élément de la liste est toujours celui qui a la distance la plus faible. L'insertion dichotomique vous permet donc de placer cet élément au bon endroit. Voici l'algorithme en pseudo-code :

```
Gauche <- 0 // Indice du premier élément
Droite <- Taille - 1 // Indice du dernier élément

TantQue Gauche <= Right
Faire

Centre <- (Gauche + Droite) / 2

Si distance du nœud à insérer < distance du nœud liste[centre]
    // On peut se passer de toute la partie à droite de la liste
    Droite = Centre - 1

Sinon Si distance du nœud à insérer > distance du nœud liste[centre]
    // On peut se passer de toute la partie à gauche de la liste
    Gauche = Centre + 1

Sinon
    // On a trouvé le bon endroit pour l'insertion
    Gauche = Centre
    Arrêter l'algorithme

Fin Si

FinTantQue

Insérer le nœud à la position gauche
```

L'exemple de la figure 9-5 illustre l'insertion du chiffre 3 dans une liste triée contenant les 8 autres premiers chiffres et le nombre 10. Il faut jouer avec les curseurs gauche et droite jusqu'à en faire coïncider 1 avec le curseur centre. Les lignes contenant des lettres correspondent à la position des curseurs, la ligne qui contient des chiffres en ordre croissant correspond aux indices de la collection. Enfin, la ligne qui contient des chiffres en ordre croissant, mais où il manque un chiffre, correspond aux valeurs de la collection.

Figure 9-5
Insertion dichotomique du chiffre 3 dans une liste



L'extrait de code suivant correspond à l'implémentation de ce que nous venons de voir.

La classe NodeList, liste générique personnalisée

```
class NodeList<T> : List<T> where T : Node
{
    public new bool Contains(T node)
    {
        return this[node] != null;
    }

    public T this[T node]
    {
        get
        {
            int count = this.Count;
            for (int i = 0; i < count; i++)
            {
                if (this[i].Tile == node.Tile)
                    return this[i];
            }
            return default(T);
        }
    }

    public void DichotomicInsertion(T node)
    {
        int left = 0;
        int right = this.Count - 1;
        int center = 0;

        while (left <= right)
        {
            center = (left + right) / 2;
            if (node.EstimatedMovement < this[center].EstimatedMovement)
                right = center - 1;
            else if (node.EstimatedMovement > this[center].EstimatedMovement)
                left = center + 1;
            else
            {
                left = center;
                break;
            }
        }
        this.Insert(left, node);
    }
}
```

La classe Pathfinding : implémentation de l'algorithme

Il est temps de passer aux choses sérieuses ! Ajoutez une classe `PathFinding` au projet et ajoutez-y une fonction statique qui retourne une liste de `Tile`. Elle devra recevoir en argument la carte sur laquelle elle travaillera, ainsi que la case de départ et celle d'arrivée.

La première chose à faire est de déclarer tout ce qui sera utile dans la fonction. Tout d'abord les collections : il en faut une qui contient la liste de cases pour la sortie de la fonction, une pour la liste ouverte, une pour la liste fermée et une pour les nœuds voisins à analyser.

Notez enfin qu'une variable contenant le nombre d'éléments de cette dernière liste est aussi déclarée. Il s'agit là d'une optimisation : l'utilisation d'une boucle `for` plutôt qu'une boucle `foreach` ne requiert pas la création d'un objet pour l'énumération. Ensuite, au lieu d'appeler à chaque fois la propriété `Count`, il est préférable de stocker sa valeur dans une variable. L'optimisation peut être encore plus poussée en employant des tableaux plutôt que des listes mais, pour ne pas compliquer plus les choses, ce n'est pas le cas ici.

Ensuite, il faut générer le nœud de départ (n'oubliez pas qu'il n'a pas de nœud parent) et l'ajouter à la liste ouverte.

Le reste de la fonction se contente d'appliquer l'algorithme : retirez le nœud de la liste ouverte et ajoutez-le à la liste fermée ; si la case du nœud est celle d'arrivée, remontez la liste fermée et remplissez la liste de sortie de la fonction, sinon inspectez les nœuds voisins. Si vous sortez de la boucle `while`, c'est qu'il n'y a plus d'éléments dans la liste ouverte et qu'il n'existe donc aucune solution ; dans ce cas, retournez `null`.

La méthode de recherche du plus court chemin

```
class Pathfinding
{
    public static List<Tile> CalculatePathWithAStar(Map map, Tile startTile, Tile
    ➥endTile)
    {
        List<Tile> result = new List<Tile>();
        NodeList<Node> openList = new NodeList<Node>();
        NodeList<Node> closedList = new NodeList<Node>();
        List<Node> possibleNodes;
        int possibleNodesCount;

        Node startNode = new Node(startTile, null, endTile);

        openList.Add(startNode);

        while (openList.Count > 0)
        {
            Node current = openList[0];
            openList.RemoveAt(0);
            closedList.Add(current);

            if (current.Tile == endTile)
            {
```

```
                List<Tile> sol = new List<Tile>();
                while (current.Parent != null)
                {
                    sol.Add(current.Tile);
                    current = current.Parent;
                }
                return sol;
            }

            possibleNodes = current.GetPossibleNode(map, endTile);
            possibleNodesCount = possibleNodes.Count;
            for (int i = 0; i < possibleNodesCount; i++)
            {
                if (!closedList.Contains(possibleNodes[i]))
                {
                    if (openList.Contains(possibleNodes[i]))
                    {
                        if (possibleNodes[i].EstimatedMovement < openList
                        ➥[possibleNodes[i]].EstimatedMovement)
                            openList[possibleNodes[i]].Parent = current;
                    }
                    else
                        openList.DichotomicInsertion(possibleNodes[i]);
                }
            }
        }
        return null;
    }
}
```

Phase de test

Il ne reste plus qu'à utiliser tout cela dans la classe `ChapitreNeuf`. Créez un objet de type `Map` et initialisez-le via un tableau de `byte`. Dans la classe ci-dessous, la taille de la fenêtre s'adapte automatiquement à la taille de la carte. Pour cela, il suffit de multiplier la taille de chaque dimension de la carte par 32 (taille d'une case).

Dans la méthode `Initialize()`, vous allez créer un objet `Tile` pour le point de départ et un autre pour le point d'arrivée. Modifiez la couleur du point de départ afin qu'il soit plus facilement reconnaissable, stockez le résultat de `CalculatePathWithAStar()` dans une liste de cases et parcourez-la de manière à coloriser toutes les cases qui composent le chemin. Le premier élément de cette liste est le point d'arrivée, vous pouvez donc employer une couleur différente. Enfin, il n'est pas nécessaire de revenir sur le contenu des méthodes `LoadContent()` et `Draw()`...

Une classe pour tester l'algorithme

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

```
Map map;

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

    map = new Map(new byte[,] {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0},
        {0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0},
        {0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0},
        {1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 2, 2, 0, 1, 0, 1, 0},
        {0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 1, 1},
        {0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
        {0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1},
        {0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1},
        {0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}
        });

    graphics.PreferredBackBufferWidth = map.TileList.GetLength(1) * 32;
    graphics.PreferredBackBufferHeight = map.TileList.GetLength(0) * 32;
}

protected override void Initialize()
{
    Tile start = map.TileList[13, 0];
    Tile end = map.TileList[11, 15];
    start.Color = Color.Green;
    end.Color = Color.Red;

    List<Tile> liste = Pathfinding.CalculatePathWithAStar(map, start, end);

    if (liste != null)
    {
        liste[0].Color = Color.Red;
        for (int i = 1; i < liste.Count; i++)
            liste[i].Color = Color.Orange;
    }

    base.Initialize();
}

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

    foreach (Tile tile in map.TileList)
    {
        tile.LoadContent(Content, "tile");
    }
}

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

```
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

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

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

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

Vous pouvez à présent tester l'algorithme sous toutes ses coutures. Commencez par bloquer le passage pour constater qu'il ne retourne bien aucun chemin.

Vous pouvez ajouter autant de types de case que votre imagination vous le permet. Par exemple, sur la figure 9-6, vous remarquez un bosquet en vert clair (le coût de chacune de ces cases est de un), ce qui amène l'algorithme à passer de préférence par la rivière.

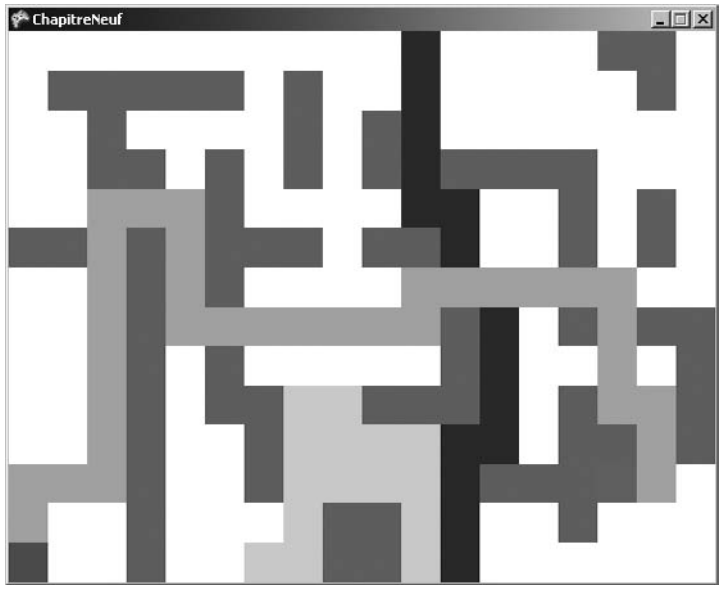


Figure 9-6
Test de l'algorithme avec un nouveau type de case

Enfin, vous pouvez aussi essayer une carte sur laquelle A* n'est pas performant (figure 9-7). L'algorithme a su trouver la bonne solution, toutefois il est tombé dans le piège tendu par la carte et a dû analyser toutes les cases de celle-ci.

Figure 9-7
Test de l'algorithme avec une carte piège

Certes, l'algorithme A* est rapide, mais il reste encore de nombreuses choses à ajouter, notamment :

- Sur des grandes cartes, diviser le parcours en *check points* et calculer seulement le temps pour aller du point de départ au premier *check point*, puis du premier *check point* au second, etc.
- Gérer les collisions avec d'autres entités différentes d'un mur ou d'un relief, ce principe étant applicable grâce à la notion de coût des nœuds.

Bien évidemment, toutes ces petites idées induiront d'autres problèmes, notamment de performances. Le code source que nous venons d'étudier n'est pas parfait : c'est à vous de développer et d'optimiser sans cesse le vôtre en fonction des besoins.

Cas pratique : implémenter le déplacement d'un personnage sur une carte

La dernière partie de ce chapitre est consacrée à la réalisation d'une application exemple plus poussée qui utilise le *pathfinding*. Le but est de déplacer un personnage (représenté par un carré) là où l'utilisateur cliquera sur la carte.

Préparation : identifier et traduire les actions du joueur

Tout d'abord, vous allez devoir vous occuper de la souris. Par défaut, elle est masquée dans une application sous XNA. Cependant, vous pouvez l'afficher simplement grâce à la propriété `IsMouseVisible`.

```
IsMouseVisible = true;
```

Ensuite, vous devez être en mesure de savoir si l'utilisateur vient de presser le bouton gauche de la souris et de récupérer les coordonnées du pointeur à ce moment-là. Commencez par ajouter la classe `ServiceHelper` que vous avez développée dans un chapitre précédent. Comme pour le clavier, commencez par créer une interface `IMouseService`.

```
interface IMouseService
{
    bool LeftButtonHasBeenPressed();
    Vector2 GetCoordinates();
}
```

Il vous reste ensuite à créer la classe qui implantera cette interface. N'oubliez pas d'ajouter dans le constructeur l'instance de la classe à la collection du `ServiceHelper`. Vous pouvez déterminer si le joueur a pressé un bouton ou non en regardant l'état de ce dernier aux instants `t` et `t-1`. À chaque appel de `Update()`, vous devrez stocker l'état de la souris et archiver son ancien état.

Le service qui met à disposition l'état de la souris

```
public class MouseService : GameComponent, IMouseService
{
    MouseState mouseState = Mouse.GetState();
    MouseState lastMouseState;

    public MouseService(Game game)
        : base(game)
    {
        ServiceHelper.Add<IMouseService>(this);
    }

    public bool LeftButtonHasBeenPressed()
    {
        return mouseState.LeftButton == ButtonState.Released &&
        ➥lastMouseState.LeftButton == ButtonState.Pressed;
    }

    public Vector2 GetCoordinates()
    {
        return new Vector2(mouseState.X, mouseState.Y);
    }

    public override void Update(GameTime gameTime)
    {
        lastMouseState = mouseState;
        mouseState = Mouse.GetState();
    }
}
```

Créer le personnage

Modifiez la classe `Tile` de manière à ajouter un nouveau type correspondant à un personnage. Dans l'exemple suivant, les personnages seront affichés en noir.

La classe Tile améliorée

```
enum TileType
{
    Wall = -1,
    Normal = 0,
```

```
    Tree = 1,
    Water = 2,
    Human = -1,
};

class Tile : Sprite
{
    int x;

    public int X
    {
        get { return this.x; }
        set { this.x = value; }
    }

    int y;

    public int Y
    {
        get { return this.y; }
        set { this.y = value; }
    }

    TileType type;

    public TileType Type
    {
        get { return type; }
    }

    public Tile(int y, int x, byte type)
        : base(new Vector2(x * 32, y * 32))
    {
        this.x = x;
        this.y = y;

        switch (type)
        {
            case 1:
                Color = Color.Gray;
                this.type = TileType.Wall;
                break;
            case 3:
                Color = Color.LightGreen;
                this.type = TileType.Tree;
                break;
            case 2:
                Color = Color.Blue;
                this.type = TileType.Water;
                break;
```

```
            case 4:
                Color = Color.Black;
                this.type = TileType.Human;
                break;
            default:
                this.type = TileType.Normal;
                break;
        }
    }
}
```

Enfin, créez une classe dérivée de `Tile` que vous appelerez `Hero`. Celle-ci contiendra un champ `msElapsed` qui permettra de réguler la vitesse de déplacement du personnage. L'autre champ est une liste de `Tile` ; une propriété permet de le définir et vérifie au passage que la liste de cases transmise n'est pas nulle. Dans la méthode `Update()`, vous déplacerez la position du `Hero` à l'écran, ainsi que sa position sur la carte en fonction de celles du dernier élément de la liste, puis vous supprimerez ce dernier élément.

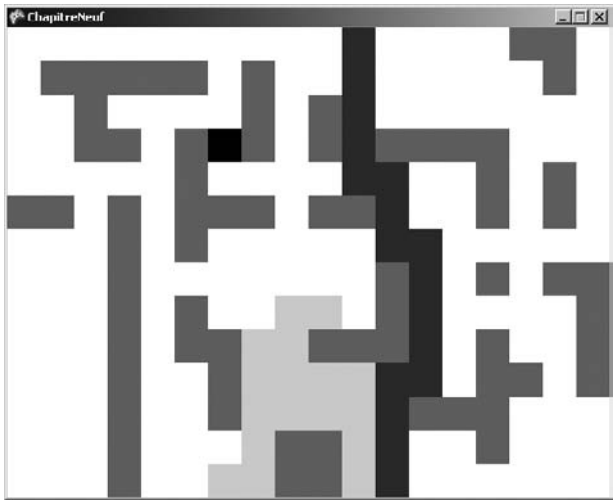
La classe Hero représente un élément particulier de la carte

```
class Hero : Tile
{
    int msElapsed = 0;

    List<Tile> walkingList = new List<Tile>();

    public List<Tile> WalkingList
    {
        set { if (value != null) walkingList = value; }
}

    public Hero(int y, int x, byte type)
        : base(y, x, type)
    {
    }

    public void Update(GameTime gameTime)
    {
        msElapsed += gameTime.ElapsedGameTime.Milliseconds;

        if (walkingList.Count != 0)
        {
            if (msElapsed >= 100)
            {
                msElapsed = 0;
                X = walkingList[walkingList.Count - 1].X;
                Y = walkingList[walkingList.Count - 1].Y;
                Position = walkingList[walkingList.Count - 1].Position;
                walkingList.RemoveAt(walkingList.Count - 1);
            }
        }
    }
}
```

Implémenter l'algorithme

Il ne vous reste plus qu'à utiliser tous ces nouveaux éléments. Instanciez un objet de type `Hero` et, lorsque l'utilisateur clique quelque part sur l'écran, passez à cet objet le résultat du calcul du chemin. Si le joueur clique sur le personnage, il n'est pas nécessaire d'effectuer le calcul du chemin !

Figure 9-8
Le sprite noir se déplace à la perfection



La nouvelle classe de test de l'algorithme

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

    Map map;
    Hero heros;

    public ChapitreNeuf()
    {
        graphics = new GraphicsDeviceManager(this);
        ServiceHelper.Game = this;
        Components.Add(new MouseService(this));
        Content.RootDirectory = "Content";

        map = new Map(new byte[,] {
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 1, 0},
            {0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 2, 0, 0, 0, 0, 0, 1, 0},
            {0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 2, 1, 1, 1, 1, 0, 0, 0},
            {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 2, 2, 0, 0, 1, 0, 1, 0},
            {1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 2, 0, 0, 1, 0, 1, 0},
```

```
            {0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0},
            {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 1, 1},
            {0, 0, 0, 1, 0, 1, 0, 0, 3, 3, 0, 1, 2, 0, 0, 0, 0, 1},
            {0, 0, 0, 1, 0, 1, 1, 3, 3, 1, 1, 1, 2, 0, 1, 0, 0, 1},
            {0, 0, 0, 1, 0, 0, 1, 3, 3, 3, 3, 2, 2, 0, 1, 1, 0, 1},
            {0, 0, 0, 1, 0, 0, 1, 3, 3, 3, 3, 2, 1, 1, 1, 0, 0, 0},
            {0, 0, 0, 1, 0, 0, 0, 3, 1, 1, 3, 2, 0, 0, 1, 0, 0, 0},
            {0, 0, 0, 1, 0, 0, 3, 3, 1, 1, 3, 2, 0, 0, 0, 0, 0, 0}
            });

        graphics.PreferredBackBufferWidth = map.TileList.GetLength(1) * 32;
        graphics.PreferredBackBufferHeight = map.TileList.GetLength(0) * 32;
        IsMouseVisible = true;
    }

    protected override void Initialize()
    {
        heros = new Hero(13, 0, 4);

        base.Initialize();
    }

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

        foreach (Tile tile in map.TileList)
        {
            tile.LoadContent(Content, "tile");
        }
        heros.LoadContent(Content, "tile");
    }

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();

        if (ServiceHelper.Get<IMouseService>().LeftButtonHasBeenPressed())
        {
            if (heros.X != ((int)ServiceHelper.Get<IMouseService>()
            ➥.GetCoordinates().X / 32) || heros.Y != ((int)ServiceHelper
            ➥.Get<IMouseService>().GetCoordinates().Y / 32))
            heros.WalkingList = Pathfinding.CalculatePathWithAStar(map, heros, map
            ➥.TileList[(int)ServiceHelper.Get<IMouseService>().GetCoordinates().Y
            ➥/ 32,(int)ServiceHelper.Get<IMouseService>().GetCoordinates().X / 32]);
        }

        heros.Update(gameTime);
        base.Update(gameTime);
    }
```

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

        spriteBatch.Begin();
        map.Draw(spriteBatch);
        heros.Draw(spriteBatch);
        spriteBatch.End();

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

Dans l'extrait de code précédent, nous créons une carte et nous y ajoutons un personnage. Si vous exécutez maintenant le projet, vous constaterez que le personnage se déplace là où vous cliquez.

En résumé

Ce chapitre vous a présenté rapidement les techniques propres à l'intelligence artificielle. Vous avez découvert l'algorithme de recherche du plus court chemin : A*. Vous avez appris à développer cet algorithme en C# et à l'appliquer à un exemple de jeu vidéo.