Collisions et physique créer un simulateur de vaisseau spatial

Vous commencez à connaître bon nombre des aspects du framework XNA et vous vous êtes sûrement déjà aventuré dans la création d'un vrai jeu. Vous avez alors certainement été confronté aux problèmes liés aux règles physiques de l'univers que vous avez créé, surtout si ce jeu était un clone d'*Asteroids*, ou tout du moins un jeu y ressemblant.

Culture Asteroids

Il s'agit d'un jeu de tir spatial en deux dimensions qui a fait sensation en 1979 (et qui continue sûrement à remporter du succès auprès de joueurs nostalgiques de ce temps). Dans ce jeu, vous contrôlez un vaisseau qui doit faire face, armé de son lance-missiles, à des champs d'astéroïdes et des vaisseaux ennemis. Le vaisseau que le joueur dirige connaît une inertie évoquant le milieu spatial, ce qui rend sa prise en main délicate.

Dans ce chapitre consacré à la création d'un petit jeu de simulation spatiale en 2D, notre but est d'étudier les collisions par rectangle ou par pixels ainsi qu'un moteur physique (le module d'un jeu chargé du déplacement de diverses entités), FarseerPhysics.

Comment détecter les collisions

Dans la première partie de ce chapitre, vous allez réaliser une première version squelette du projet, le but étant de s'intéresser tout particulièrement à la gestion des collisions entre

le vaisseau du joueur et les astéroïdes. Dans un premier temps, la gestion se basera sur la collision entre rectangles, puis nous descendrons au niveau des pixels.

Créer les bases du jeu

Commencez par créer un projet vierge. Avant toute chose, ajoutez deux variables statiques à la classe principale du projet. Ces dernières définiront la taille de la fenêtre, il faudra donc les appliquer à l'objet `graphics`. Les valeurs qui sont utilisées ici correspondent à une fenêtre étroite, mais assez haute.

```
public static int SCREEN_WIDTH = 512;
public static int SCREEN_HEIGHT = 748;

public ChapitreDix()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    graphics.PreferredBackBufferHeight = SCREEN_HEIGHT;
    graphics.PreferredBackBufferWidth = SCREEN_WIDTH;
}
```

Créer le vaisseau

Maintenant, occupez-vous de la création du vaisseau du joueur. En espérant que cela ne froissera pas les fans du jeu d'origine, dans l'exemple proposé dans ce livre, le joueur contrôlera un simple triangle blanc de 32 pixels par 32 pixels, visible à la figure 10-1.

Figure 10-1

Quel ennemi oserait se présenter face à pareil engin ?

Comme d'habitude, importez les fichiers suivants des précédents projets : `IKeyboardService.cs`, `KeyboardService.cs`, `ServiceHelper.cs` et `Sprite.cs`. Ajoutez une classe `Player` au projet ; vous la ferez dériver de la classe `Sprite`. Puisque la position du vaisseau du joueur ne peut pas être déterminée dès l'appel au constructeur (il faut connaître la taille de la texture pour pouvoir centrer le vaisseau), passez plutôt la valeur prédéfinie `Vector2.Zero` au constructeur de la classe parente.

```
public Player()
    : base(Vector2.Zero)
{
}
```

L'étape suivante est de compléter la méthode `Update()`. Celle-ci aura deux tâches à effectuer :

- La première est de déplacer le vaisseau en fonction des touches que le joueur presse : rien de plus facile en appelant le service dédié. Profitez-en pour ajouter un élément de gameplay déterminant : la vitesse du vaisseau lorsqu'il recule sera moins importante que lorsqu'il avance.

- La seconde tâche consiste à vérifier que le vaisseau du joueur ne quitte pas l'écran. Si cela se produit, il a perdu, il faut le replacer au centre de l'écran.

  Vous serez en mesure de répondre à ces exigences en utilisant les propriétés de taille de la texture utilisée par le vaisseau, ainsi que la taille de l'écran (les variables statiques `SCREEN_WIDTH` et `SCREEN_HEIGHT`). Comme d'habitude, n'oubliez pas que, par défaut, le point d'origine est placé en haut à gauche de la texture.

La méthode Update() complétée

```
public void Update(GameTime gameTime)
{
    if(ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Up))
        Position = new Vector2(Position.X, Position.Y - (float)(0.4 *
        ➥gameTime.ElapsedGameTime.Milliseconds));

    if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Down))
        Position = new Vector2(Position.X, Position.Y + (float)(0.2 *
        ➥gameTime.ElapsedGameTime.Milliseconds));

    if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Left))
        Position = new Vector2(Position.X - (float)(0.3 *
        ➥gameTime.ElapsedGameTime.Milliseconds), Position.Y);

    if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Right))
        Position = new Vector2(Position.X + (float)(0.3 *
        ➥gameTime.ElapsedGameTime.Milliseconds), Position.Y);

    if (Position.X < 0)
        Position = new Vector2(0, Position.Y);

    if (Position.X + Texture.Width > ChapitreDix.SCREEN_WIDTH)
        Position = new Vector2(ChapitreDix.SCREEN_WIDTH - Texture.Width,
        ➥Position.Y);

    if (Position.Y < 0)
        Position = new Vector2(Position.X, 0);

    if (Position.Y + Texture.Height > ChapitreDix.SCREEN_HEIGHT)
        Position = new Vector2(Position.X, ChapitreDix.SCREEN_HEIGHT -
        ➥Texture.Height);
}
```

À vos claviers

La structure qui a été retenue pour ce jeu se contente d'utiliser des classes dérivées de la classe `Sprite`. Cependant, vous pourriez très bien faire dériver la classe `Player` de la classe `DrawableGameComponent`, ou encore utiliser les interfaces `IGameComponent`, `IUpdatable` et `IDrawable`. N'hésitez pas à récrire ce mini-jeu exemple avec ces solutions, cela constitue un très bon entraînement.

Dernière méthode de cette classe, `ResetPosition ()` se chargera de placer le vaisseau du joueur au centre de la largeur de l'écran et en retrait de 10 % de sa hauteur. Cette méthode sera appelée en début de partie (après le chargement de la texture du vaisseau), ainsi qu'à chaque fois que le joueur aura perdu.

```
public void ResetPosition()
{
    Position = new Vector2(ChapitreDix.SCREEN_WIDTH / 2 - (Texture.Width / 2), 9 *
    ➥(ChapitreDix.SCREEN_HEIGHT / 10) - Texture.Height);
}
```

Créer les astéroïdes

À présent, vous allez vous occuper des astéroïdes. Tout d'abord, créez une texture ou récupérez-la sur Internet, par exemple sur `http://www.cgtextures.com/`. Comme vous le voyez à la figure 10-2, nous utilisons un simple disque marron.

Figure 10-2
Des sphères parfaites se cachent peut-être dans l'espace…

Créez une classe `Asteroid` et ajoutez-lui un attribut `speed` de type `Vector2` : il pourra y avoir plusieurs astéroïdes à l'écran, la position et la vitesse de chacun d'entre eux sera aléatoire. Vous fixez ces deux éléments dans une méthode privée `Initialize ()`. Pour la génération de nombres aléatoires, vous utiliserez un objet statique de la classe `Random` et vous passerez à son constructeur le nombre de millisecondes à cet instant.

```
static Random random = new Random(DateTime.Now.Millisecond);
```

Avancé. Les nombres aléatoires

Il faut savoir qu'en informatique, les nombres aléatoires ne le sont pas réellement. En fait, il est impossible de générer une suite de nombres réellement aléatoires, il faudrait plutôt les appeler nombres pseudo-aléatoires. La détermination de ces nombres se fait via l'utilisation d'une graine, c'est-à-dire d'une valeur de départ. Si la valeur de cette graine est toujours la même, les nombres générés par le programme seront toujours les mêmes. Pour éviter ce problème, vous pouvez faire varier la valeur de la graine en fonction du temps.

La méthode `Next()` de l'objet `random` attend comme paramètres une borne inférieure et une borne supérieure, puis retourne un entier. Initialement, le sprite devra se situer légèrement au-dessus de la fenêtre et n'importe où sur l'axe des abcisses. Pour la vitesse, en ce qui concerne l'axe X, l'astéroïde doit pouvoir aller vers la gauche comme vers la droite ; en ce qui concerne l'axe Y, il doit uniquement pouvoir aller vers le bas.

```
private void Initialize()
{
    Position = new Vector2(random.Next(0, ChapitreDix.SCREEN_WIDTH - Texture.Width),
    ➥-Texture.Height);
    speed = new Vector2((float)random.Next(-7, 7) / 10, (float)random.Next(1, 7)
    ➥/ 10);
}
```

Vous pouvez à présent écrire le constructeur. Dans celui-ci, vous appellerez la méthode `LoadContent ()` de la classe parente, ainsi que la méthode `Initialize ()`.

```
public Asteroid(ContentManager content)
    : base(Vector2.Zero)
{
    base.LoadContent(content, "asteroid");
    Initialize();
}
```

Dernière chose, la méthode `Update ()` où vous mettrez à jour la position de l'astéroïde et, s'il sort de l'écran, où vous appellerez sa méthode `Initialize()`.

```
public void Update(GameTime gameTime)
{
    Position = new Vector2(Position.X + (speed.X *
    ➥gameTime.ElapsedGameTime.Milliseconds), Position.Y + (speed.Y *
    ➥gameTime.ElapsedGameTime.Milliseconds));

    if (Position.X + Texture.Width < 0)
        Initialize();

    if (Position.X > ChapitreDix.SCREEN_WIDTH)
        Initialize();

    if (Position.Y > ChapitreDix.SCREEN_HEIGHT)
        Initialize();
}
```

Vous n'avez plus qu'à utiliser ces deux classes. Il n'y a rien de spécial à souligner en ce qui concerne la création du vaisseau du joueur. Vous stockerez les astéroïdes dans une collection `List<>`. Pour faire apparaître un astéroïde toutes les *x* secondes, utilisez un objet de type `TimeSpan` que vous remettrez à zéro à chaque ajout d'un nouvel objet à la liste.

La classe de base du jeu complétée

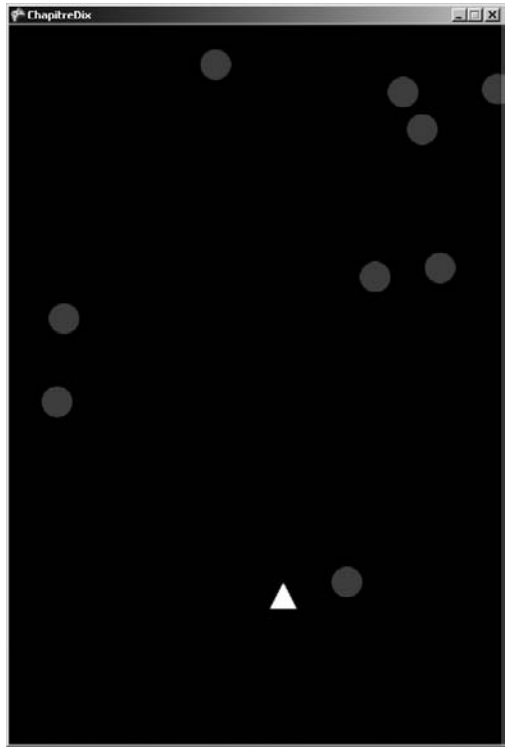
```
public class ChapitreDix : Microsoft.Xna.Framework.Game
{
    public static int SCREEN_WIDTH = 512;
    public static int SCREEN_HEIGHT = 748;
    static int NEW_METEOR_TIME = 5;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Player playerShip;
    List<Asteroid> asteroids = new List<Asteroid>();

    TimeSpan elapsedTimeSinceLastNewAsteroid = TimeSpan.Zero;

    public ChapitreDix()
    {
```

```
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        ServiceHelper.Game = this;
        Components.Add(new KeyboardService(this));
        graphics.PreferredBackBufferHeight = SCREEN_HEIGHT;
        graphics.PreferredBackBufferWidth = SCREEN_WIDTH;
    }

    protected override void Initialize()
    {
        playerShip = new Player();

        base.Initialize();
    }

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

        playerShip.LoadContent(Content, "ship");
        playerShip.ResetPosition();
    }

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

        playerShip.Update(gameTime);

        foreach (Asteroid asteroid in asteroids)
            asteroid.Update(gameTime);

        elapsedTimeSinceLastNewAsteroid += gameTime.ElapsedGameTime;

        if (elapsedTimeSinceLastNewAsteroid.Seconds >= NEW_METEOR_TIME)
        {
            asteroids.Add(new Asteroid(Content));
            elapsedTimeSinceLastNewAsteroid = TimeSpan.Zero;
        }

        base.Update(gameTime);
    }

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

        spriteBatch.Begin();
        playerShip.Draw(spriteBatch);
        foreach (Asteroid asteroid in asteroids)
```

```
            asteroid.Draw(spriteBatch);
        spriteBatch.End();

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

Vous pouvez à présent compiler et exécuter le jeu (voir figure 10-3). Cependant, sans les collisions, ce n'est pas très intéressant d'y jouer…

Figure 10-3
Les astéroïdes deviennent vite très nombreux



Établir une zone de collision autour des astéroïdes

La méthode la plus simple pour tester si le vaisseau du joueur entre en collision avec un astéroïde consiste à encadrer chaque élément par un rectangle, puis à tester si les rectangles se coupent ou non.

Commencez par ajouter une propriété à la classe `Sprite` afin que les classes `Player` et `Asteroid` puissent en bénéficier. Cette propriété devra créer un rectangle en fonction de la position du `Sprite` et de la taille de sa texture. Pour que la classe reste générique, si la variable `sourceRectangle` n'est pas nulle, utilisez-la plutôt que les dimensions de la texture.

```
public Rectangle Rectangle
{
    get
    {
        if (sourceRectangle == null)
            return new Rectangle((int)position.X, (int)position.Y, texture.Width,
            ➥texture.Height);
        else
            return new Rectangle((int)position.X, (int)position.Y,
            ➥sourceRectangle.Value.Width, sourceRectangle.Value.Height);
    }
}
```

Cette portion de code aurait aussi pu s'écrire de la manière suivante :

```
public Rectangle Rectangle
{
    get { return new Rectangle((int)Position.X, (int)Position.Y,
        (sourceRectangle == null) ? Texture.Width : sourceRectangle.Value.Width,
        ➥(sourceRectangle == null) ? Texture.Height : sourceRectangle.Value.Height);
    }
}
```

Avancé Opérateur ternaire

L'exemple précédent utilise un élément du langage que vous n'avez encore jamais vu : l'opérateur ternaire. Il permet de résumer des blocs if en une seule ligne. Sa syntaxe est la suivante :

```
(test) ? valeur si vrai : valeur si faux
```

Modifiez ensuite la méthode `Update ()` de la classe `ChapitreDix` pour qu'elle effectue le test entre les deux rectangles. Utilisez la méthode `Intersects()` d'un rectangle et passez l'autre rectangle en argument.

Si les deux rectangles se superposent, le joueur a perdu : il faut réinitialiser sa position et vider la liste des astéroïdes. La liste se vide via la méthode `Clear ()`. Attention, si vous supprimez un ou plusieurs éléments de la collection, vous devez sortir de la boucle qui est en train de l'énumérer.

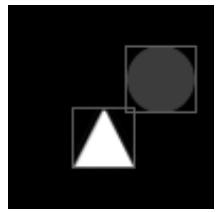
```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    playerShip.Update(gameTime);

    foreach (Asteroid asteroid in asteroids)
    {
        asteroid.Update(gameTime);
        if (playerShip.Rectangle.Intersects(asteroid.Rectangle))
        {
            playerShip.ResetPosition();
```

```
            asteroids.Clear();
            break;
        }
    }

    elapsedTimeSinceLastNewAsteroid += gameTime.ElapsedGameTime;

    if (elapsedTimeSinceLastNewAsteroid.Seconds >= NEW_METEOR_TIME)
    {
        asteroids.Add(new Asteroid(Content));
        elapsedTimeSinceLastNewAsteroid = TimeSpan.Zero;
    }

    base.Update(gameTime);
}
```

Vous pouvez tester le jeu à présent : la détection des collisions fonctionne et le jeu reprendra à zéro dès que le joueur percutera un astéroïde. En réalité, cette dernière phrase n'est pas juste : le jeu reprend à zéro lorsque le rectangle qui entoure le vaisseau du joueur entre en collision avec celui qui entoure un astéroïde, pourtant il n'y a pas forcément collision entre les deux éléments (figure 10-4).

Figure 10-4

Il y a collision entre les rectangles mais pas entre les deux éléments



Comment savoir s'il y a vraiment collision entre deux éléments ? La réponse se situe au niveau des pixels… Si vous détectez une collision potentielle grâce à la méthode d'intersection entre les rectangles, vous devrez zoomer sur la zone de chevauchement entre les deux sprites et analyser les pixels de cette zone. Si, à un point (x, y), les pixels des deux sprites sont totalement opaques, il y a collision.

Vous pouvez récupérer des informations sur la couleur de chaque pixel d'une texture grâce à la méthode `GetData ()` de la classe `Texture2D`. Commencez donc par ajouter une nouvelle propriété à la classe `Sprite` qui permettra de récupérer ces informations. Passez en paramètre à la méthode `GetData()` un tableau de `Color` qu'elle devra compléter. Le nombre d'éléments du tableau sera le même que le nombre de pixels dans la texture.

```
public Color[] TextureData
{
    get
    {
        Color[] textureData = new Color[texture.Width * texture.Height];
        texture.GetData(textureData);
        return textureData;
    }
}
```

Passons maintenant à l'écriture de la méthode `CollisionPerPixel` (dans le projet exemple, elle est rattachée à la classe `ChapitreDix`). La première chose à faire est de déterminer le rectangle contenant tous les pixels concernés par la collision entre les deux sprites. Une fois la position des quatre côtés du rectangle déterminée, utilisez-les pour parcourir le tableau de pixels. Attention, tous les pixels sont classés de manière linéaire dans le tableau puisque celui-ci ne comporte qu'une dimension.

Pour chacun des pixels de cette zone, vérifiez la composante alpha (la transparence). Si elle n'est pas nulle pour les deux sprites, alors les deux sprites ne sont pas totalement transparents et il y a collision.

```
protected bool CollisionPerPixels(Sprite spriteA, Sprite spriteB)
{
    int top = Math.Max(spriteA.Rectangle.Top, spriteB.Rectangle.Top);
    int bottom = Math.Min(spriteA.Rectangle.Bottom, spriteB.Rectangle.Bottom);
    int left = Math.Max(spriteA.Rectangle.Left, spriteB.Rectangle.Left);
    int right = Math.Min(spriteA.Rectangle.Right, spriteB.Rectangle.Right);

    for (int y = top; y < bottom; y++)
    {
        for (int x = left; x < right; x++)
        {
            Color colorA = spriteA.TextureData[(x - spriteA.Rectangle.Left) +
                        (y - spriteA.Rectangle.Top) * spriteA.Rectangle.Width];
            Color colorB = spriteB.TextureData[(x - spriteB.Rectangle.Left) +
                        (y - spriteB.Rectangle.Top) * spriteB.Rectangle.Width];

            if (colorA.A != 0 && colorB.A != 0)
                return true;
        }
    }

    return false;
}
```

Reprenez la méthode et ajoutez l'appel à la fonction `CollisionPerPixels()` en plus de la méthode de détection qui utilise les rectangles.

Langage C# Opérateur &&

Lorsque vous utilisez l'opérateur && de la manière suivante :

`test_A && test_B`

Si le résultat de `test_A` est faux, `test_B` ne sera même pas exécuté. Ainsi, le test qui descend au niveau de détails des pixels ne sera exécuté que si le test plus large avec les rectangles a détecté une collision potentielle.

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

```
        playerShip.Update(gameTime);

        foreach (Asteroid asteroid in asteroids)
        {
            asteroid.Update(gameTime);
            if (playerShip.Rectangle.Intersects(asteroid.Rectangle) &&
            ➥CollisionPerPixels(playerShip, asteroid))
            {
                playerShip.ResetPosition();
                asteroids.Clear();
                break;
            }
        }

        elapsedTimeSinceLastNewAsteroid += gameTime.ElapsedGameTime;

        if (elapsedTimeSinceLastNewAsteroid.Seconds >= NEW_METEOR_TIME)
        {
            asteroids.Add(new Asteroid(Content));
            elapsedTimeSinceLastNewAsteroid = TimeSpan.Zero;
        }

        base.Update(gameTime);
    }
```

Cette fois-ci, si vous testez le jeu, vous remarquez qu'il est possible d'effleurer les astéroïdes sans problèmes de collisions (figure 10-5). Il ne vous reste plus qu'à compléter le jeu en ajoutant par exemple un système de gestion du score.

Figure 10-5
La détection de collision est maintenant plus précise

Simuler un environnement spatial : la gestion de la physique

Votre simulation spatiale n'est pas encore parfaite : en effet, les déplacements des vaisseaux ne semblent pas vraiment naturels et les astéroïdes n'entrent pas en collision entre eux… Dans la deuxième partie de ce chapitre, vous allez apprendre à utiliser un moteur physique pour simuler un environnement ressemblant à l'espace.

Choisir un moteur physique

En fait, le moteur physique est le module du jeu qui s'occupe du mouvement des objets, de la manière dont ils interagissent les uns avec les autres (par exemple les collisions), ou encore des comportements spéciaux qu'ils peuvent avoir (rebonds, frictions, déformations, etc.).

Certains jeux portent un intérêt énorme à la physique. Par exemple, dans Half Life 2, ou encore dans Portal (ces deux jeux utilisent le même moteur physique Havok), le joueur

est très souvent confronté à des énigmes qu'il devra résoudre en utilisant les lois de la physique. Bien sûr un moteur physique peut être beaucoup plus modeste, comme ceux qui sont utiles dans les petits jeux de plates-formes Mario-like, où vous devez simplement vous contenter d'appliquer la gravité sur vos personnages.

Le projet Phun (*http://www.phunland.com/wiki/Home*), à mi-chemin entre le jeu et l'utilitaire, est un formidable simulateur physique issu des recherches d'une université suédoise. Vous y dessinez des formes auxquelles vous pouvez attribuer densité, poids, etc. Même les liquides y sont gérés (figure 10-6) !

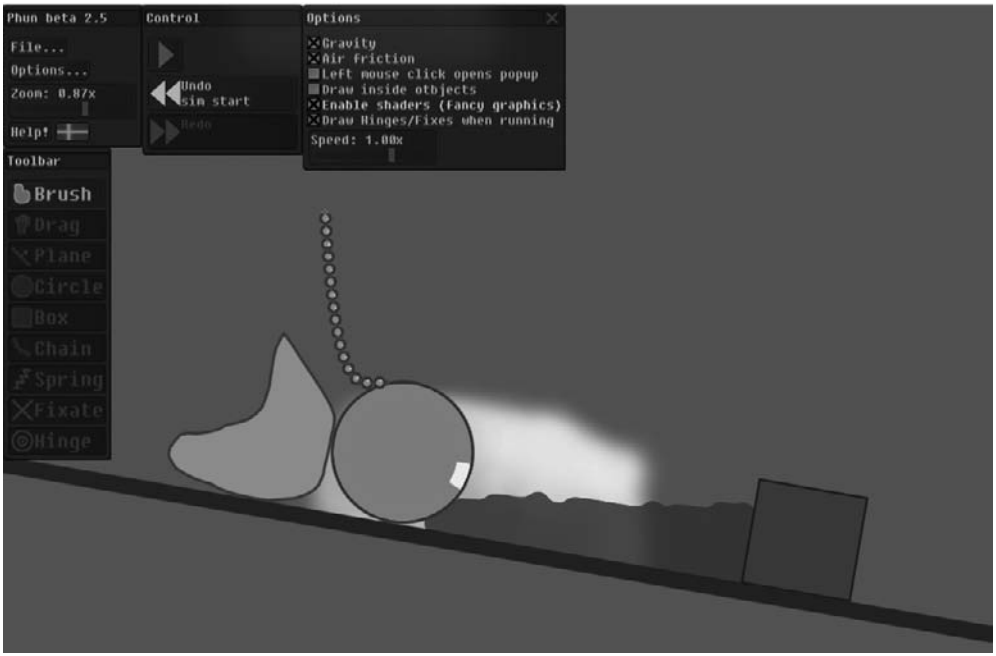


Figure 10-6

Les possibilités de Phun vous occuperont pendant des heures

Jusqu'à présent, toutes les briques nécessaires à la création d'un jeu vidéo (gestion de l'affichage à l'écran, des périphériques, du son, etc.) vous étaient fournies par XNA. Cependant, XNA ne possède nativement aucun système pour la gestion de la physique, vous allez devoir en créer un vous-même… Ou, plus modestement, en utiliser un fourni par un autre développeur.

Il existe un grand nombre de moteurs physiques sur Internet. La première chose à faire est d'en choisir un utilisable en C#, ensuite il faut s'intéresser à sa licence d'utilisation, par exemple :

- gratuit et open source (vous pouvez accéder à leur code source et y apporter des modifications) dans tous les contextes ;

- gratuit et open source uniquement pour les jeux non commerciaux ;
- gratuit et closed source dans tous les contextes ;
- payant et closed source.

> **Licences**
>
> Il existe un tas d'autres licences (elles portent d'ailleurs toutes un nom), les nuances entre elles étant parfois très subtiles : identifiez donc bien votre besoin lors du choix d'un moteur.

Dans ce chapitre, vous allez faire vos premiers pas avec le moteur FarseerPhysics. Ce moteur est gratuit, open source et directement compatible avec XNA.

Télécharger et installer FarseerPhysics

Le moteur physique FarseerPhysics est, au moment de l'écriture de ce livre, disponible en version 2.0.1. Le projet a été initialement lancé par Jeff Weber, mais il est à présent maintenu par une équipe de trois personnes. Il est disponible pour XNA et Silverlight (la technologie Microsoft concurrente d'Adobe Flash), mais propose aussi des classes indépendantes de toute plate-forme graphique.

1. Rendez-vous sur la page du projet sur CodePlex, la plate-forme de Microsoft pour les projets open source : *http://www.codeplex.com/FarseerPhysics*. Accédez à la page de téléchargement en cliquant sur l'onglet Releases et choisissez le projet pour XNA (figure 10-7).

Figure 10-7
La version du moteur adaptée à une utilisation avec XNA

2. Les développeurs proposent aussi une version du moteur avec des exemples d'utilisation simples ou plus avancés. Vous pouvez choisir de télécharger ces versions pour tester les capacités du moteur (figures 10-8 et 10-9).

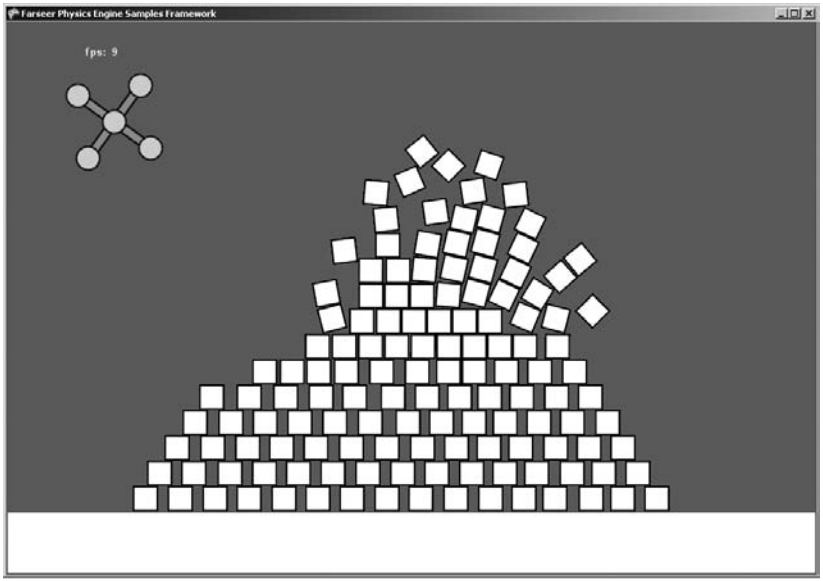


Figure 10-8

Une grande quantité de cubes soumis à la gravité

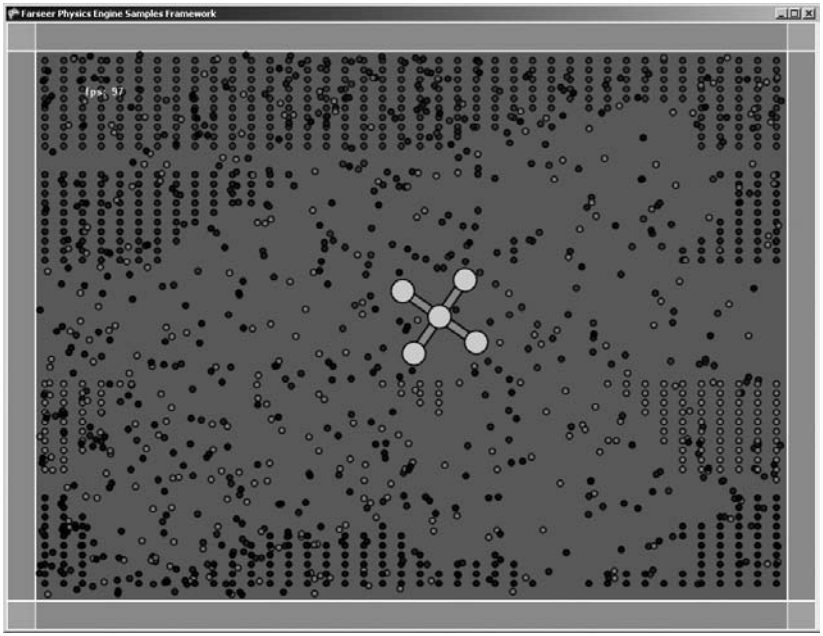


Figure 10-9

Le moteur sait gérer une multitude de collisions sans que les performances en pâtissent trop

3. Que vous ayez choisi de télécharger le moteur avec ou sans exemples, l'archive contiendra un projet `FarseerPhysics.csproj`. Ajoutez ce projet à la solution (figure 10-10) et générez-le (figure 10-11).

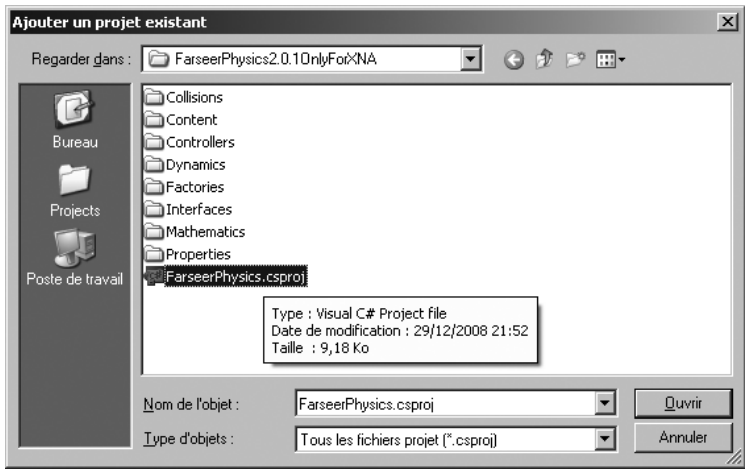


Figure 10-10

Ajout du projet de FarseerPhysics à la solution

Figure 10-11

Génération du projet FarseerPhysics

4. Créez ensuite un projet nommé `ChapitreDix_2`. Pour pouvoir utiliser le moteur dans ce projet, vous devez ajouter une référence vers le projet `FarseerPhysics`. Cliquez droit sur la section référence du projet dans l'explorateur de solution, puis cliquez sur Ajouter une référence… Dans la fenêtre qui s'ouvre, cliquez sur l'onglet Projets et choisissez dans la liste celui nommé FarseerPhysics (figure 10-12).

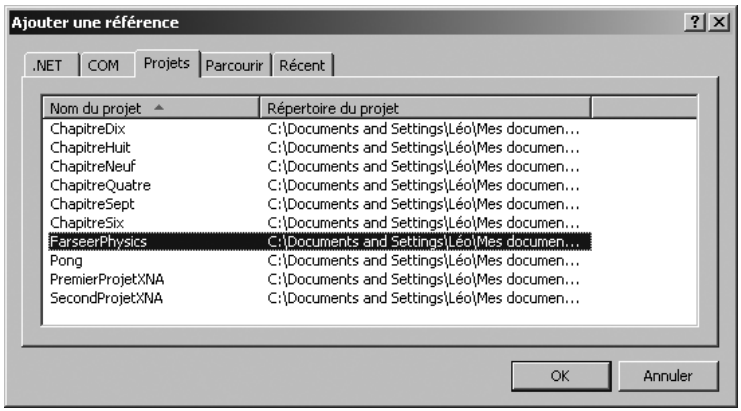


Figure 10-12

Ajout d'une référence vers un autre projet

5. La dernière chose à faire est d'ajouter une directive `using`.

```
using FarseerGames.FarseerPhysics;
```

Prise en main du moteur physique

Une nouvelle fois, ajoutez au projet les fichiers `IKeyboardService.cs`, `KeyboardService.cs`, `ServiceHelper.cs` et `Sprite.cs`. Créez les classes `Player` et `Asteroid` qui dériveront de la classe `Sprite`.

L'utilisation du moteur FarseerPhysics repose sur la classe `PhysicsSimulator` qui dispose de deux constructeurs : le premier n'attend aucun paramètre, alors que le second attend un objet de type `Vector2`. Cet objet correspond à la gravité à appliquer sur les axes X et Y. Donc, si vous utilisez le constructeur sans arguments, il n'y aura pas de gravité !

Vous devrez appeler régulièrement la méthode `Update()` du simulateur pour que celui-ci mette à jour tous les éléments qu'il gère. Cette méthode attend comme paramètre un intervalle de temps. Pour se rapprocher des calculs physiques habituels, passez-lui un temps en secondes. Ce temps est récupérable en utilisant la propriété `ElapsedGameTime.Milliseconds` que vous diviserez par mille (la propriété `ElapsedGameTime.Seconds` étant un entier, elle serait imprécise).

Enfin, passez l'objet de type `PhysicsSimulator` au constructeur des classes `Player` et `Asteroid`. Ci-dessous, vous retrouvez le code complet de la classe `ChapitreDix_2`.

Première classe de test du moteur physique

```
public class ChapitreDix_2 : Microsoft.Xna.Framework.Game
{
    public static int SCREEN_WIDTH = 512;
    public static int SCREEN_HEIGHT = 748;
    static int NEW_ASTEROID_TIME = 5;

    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    PhysicsSimulator physicsSimulator;

    Player ship;
    List<Asteroid> asteroids = new List<Asteroid>();

    TimeSpan elapsedTimeSinceLastNewAsteroid = TimeSpan.Zero;

    public ChapitreDix_2()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = "Content";
        ServiceHelper.Game = this;
        Components.Add(new KeyboardService(this));
        graphics.PreferredBackBufferHeight = SCREEN_HEIGHT;
        graphics.PreferredBackBufferWidth = SCREEN_WIDTH;
    }
```

```
    protected override void Initialize()
    {
        physicsSimulator = new PhysicsSimulator();
        ship = new Player(physicsSimulator);
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        ship.LoadContent(Content, "ship2");
        ship.ResetPosition();
    }

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

        physicsSimulator.Update(gameTime.ElapsedGameTime.Seconds * 0.001f);

        ship.Update(gameTime);

        foreach (Asteroid asteroid in asteroids)
        {
            asteroid.Update(gameTime);
        }

        elapsedTimeSinceLastNewAsteroid += gameTime.ElapsedGameTime;

        if (elapsedTimeSinceLastNewAsteroid.Seconds >= NEW_ASTEROID_TIME)
        {
            asteroids.Add(new Asteroid(physicsSimulator, Content));
            elapsedTimeSinceLastNewAsteroid = TimeSpan.Zero;
        }


        base.Update(gameTime);
    }

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

        spriteBatch.Begin();
        ship.Draw(spriteBatch);
        foreach (Asteroid asteroid in asteroids)
            asteroid.Draw(spriteBatch);
        spriteBatch.End();

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

À présent, vous devez adapter la classe `Player` du début de ce chapitre à l'utilisation du moteur physique. Celui-ci ne traitera pas directement une texture ni un sprite, il travaillera avec des objets de type `Body` (des corps). Vous devez donc créer un corps pour chaque élément qui devra être traité par FarseerPhysics. La classe `Body` se situe dans l'espace de noms `FarseerGames.FarseerPhysics.Dynamics`.

La création d'un corps se fait en utilisant la classe `BodyFactory` (espace de noms `FarseerGames.FarseerPhysics.Factories`). La méthode à utiliser dépend de la forme que vous désirez donner à votre corps. Dans le cas du vaisseau du joueur, utilisez une forme carrée grâce à la méthode `CreateRectangleBody()`.

Tableau 10-1 Paramètres de la méthode CreateRectangleBody

Paramètre	Description
`PhysicsSimulator physicsSimulator`	Ce paramètre est optionnel. Si vous l'utilisez, le corps sera automatiquement ajouté au simulateur physique.
`Float width`	Largeur du corps.
`Float height`	Hauteur du corps.
`Float mass`	Masse du corps.

Un corps possède ses propres propriétés position et rotation. Le sprite est la représentation graphique du vaisseau et le corps, sa représentation physique. Pour ne pas avoir de décalage entre les deux, vous mettez à jour la position et l'angle de rotation du sprite en fonction des propriétés du corps.

En ce qui concerne la rotation du corps, le point d'origine est pris au milieu du corps, pensez donc à faire de même en ce qui concerne le sprite.

Vous pouvez appliquer une force sur un corps simplement grâce à la méthode `ApplyForce()`. Elle attend comme paramètre un objet `Vector2` qui contient les valeurs à appliquer sur les axes X et Y. Pour appliquer une force sur le vaisseau et que celui-ci se dirige dans la bonne direction, utilisez les fonctions mathématiques `Sin()` et `Cos()` à partir de son angle de rotation en radian (rappelez-vous du cercle trigonométrique !).

Les quatre derniers tests de la méthode `Update()` permettent au vaisseau de disparaître d'un côté de l'écran pour réapparaître de l'autre.

Le vaisseau du joueur utilisant le moteur physique

```
class Player : Sprite
{
    Body body;

    public Player(PhysicsSimulator physicsSimulator)
        : base(Vector2.Zero)
    {
        body = BodyFactory.Instance.CreateRectangleBody(physicsSimulator, 32, 32, 1);
    }
```

```
    public void Update(GameTime gameTime)
    {
        Position = body.Position;
        Rotation = body.Rotation;

        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Left))
            body.Rotation -= 0.05f;

        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Right))
            body.Rotation += 0.05f;

        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Up))
            body.ApplyForce(new Vector2((float)Math.Sin(body.Rotation) * 100,
            ➥(float)Math.Cos(body.Rotation) * -100));

        if (ServiceHelper.Get<IKeyboardService>().IsKeyDown(Keys.Down))
            body.ApplyForce(new Vector2((float)Math.Sin(body.Rotation) * -100,
            ➥(float)Math.Cos(body.Rotation) * 100));

        if (body.Position.X > ChapitreDix_2.SCREEN_WIDTH + (2 * Texture.Width))
            body.Position = new Vector2(-Texture.Width, body.Position.Y);

        if (body.Position.X < 0 - (2 * Texture.Width))
            body.Position = new Vector2(ChapitreDix_2.SCREEN_WIDTH + Texture.Width,
            ➥body.Position.Y);

        if (body.Position.Y > ChapitreDix_2.SCREEN_HEIGHT + (2 * Texture.Height))
            body.Position = new Vector2(body.Position.X, -Texture.Height);

        if (body.Position.Y < 0 - (2 * Texture.Height))
            body.Position = new Vector2(body.Position.X, ChapitreDix_2.SCREEN_HEIGHT
            ➥+ Texture.Height);
    }

    public void ResetPosition()
    {
        body.Position = new Vector2(ChapitreDix_2.SCREEN_WIDTH / 2 - (Texture.Width
        ➥/ 2), 9 * (ChapitreDix_2.SCREEN_HEIGHT / 10) - Texture.Height);
        Position = body.Position;
        body.Rotation = 0;
        Rotation = body.Rotation;
        Origin = new Vector2(Texture.Width / 2, Texture.Height / 2);
    }
}
```

En ce qui concerne la classe `Asteroid`, il faut appliquer deux forces. La première utilise la méthode `ApplyForce()` que nous venons de voir (la direction sera aléatoire). La seconde utilise la méthode `ApplyForceAtLocalPoint()` qui, comme son nom l'indique, vous permet d'appliquer une force à un point précis de votre corps. Ce point, déterminé par un objet

de type `Vector2` est donc le deuxième paramètre attendu par la méthode. Dans le cas présent, il est choisi aléatoirement de manière à ce que la rotation ainsi subie par le corps ne soit pas la même pour tous les astéroïdes.

Cette fois encore, n'oubliez pas de toujours coordonner position et rotation entre le sprite et le corps !

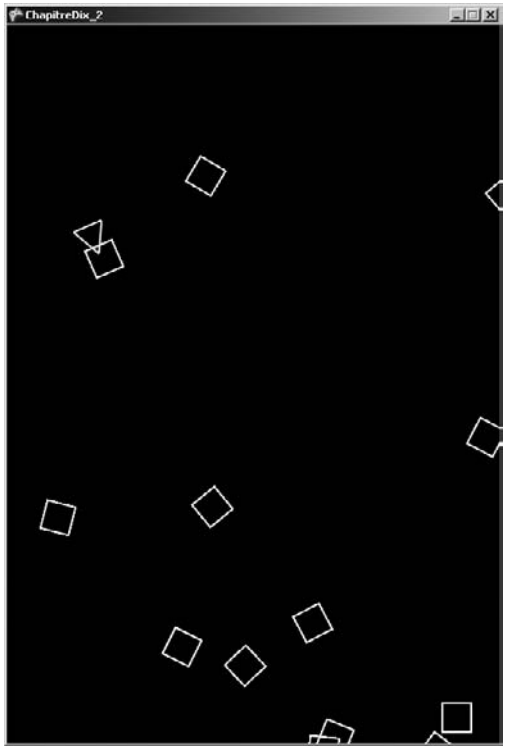
Les astéroïdes utilisent eux aussi le moteur physique

```
class Asteroid : Sprite
{
    static Random random = new Random(DateTime.Now.Millisecond);

    Body body;

    public Asteroid(PhysicsSimulator physicsSimulator, ContentManager content)
        : base(Vector2.Zero)
    {
        base.LoadContent(content, "asteroid2");
        body = BodyFactory.Instance.CreateRectangleBody(physicsSimulator, 32, 32, 1);
        Origin = new Vector2(Texture.Width / 2, Texture.Height / 2);
        Initialize();
    }

    private void Initialize()
    {
        body.Position = new Vector2(random.Next(0, ChapitreDix_2.SCREEN_WIDTH -
        ➥Texture.Width), -Texture.Height);
        body.ApplyForce(new Vector2(random.Next(-3, 3) * 1000, random.Next(-3, 3) *
        ➥1000));
        body.ApplyForceAtLocalPoint(new Vector2(random.Next(-3, 3) * 100,
        ➥random.Next(-3, 3) * 100), new Vector2(random.Next(0, Texture.Width),
        ➥random.Next(0, Texture.Height)));
        Position = body.Position;
        Rotation = body.Rotation;
    }

    public void Update(GameTime gameTime)
    {
        Position = body.Position;
        Rotation = body.Rotation;

        if (body.Position.X > ChapitreDix_2.SCREEN_WIDTH + (2 * Texture.Width))
            body.Position = new Vector2(-Texture.Width, body.Position.Y);

        if (body.Position.X < 0 - (2 * Texture.Width))
            body.Position = new Vector2(ChapitreDix_2.SCREEN_WIDTH + Texture.Width,
            ➥body.Position.Y);

        if (body.Position.Y > ChapitreDix_2.SCREEN_HEIGHT + (2 * Texture.Height))
            body.Position = new Vector2(body.Position.X, -Texture.Height);

        if (body.Position.Y < 0 - (2 * Texture.Height))
```

```
                body.Position = new Vector2(body.Position.X, ChapitreDix_2.SCREEN_HEIGHT
                ➥+ Texture.Height);
        }
    }
```

Maintenant, testez le jeu. L'inertie du vaisseau et les astéroïdes qui dérivent sont vraiment bien rendus. Au passage, vous n'avez qu'à améliorer l'image du vaisseau et celle des astéroïdes pour avoir un rendu plus *old school* (figure 10-13).

Figure 10-13
La nouvelle version du jeu… sans les collisions !



Les collisions avec FarseerPhysics

Cette nouvelle version du jeu doit faire face à un nouveau problème de taille : il n'y a plus de gestion des collisions ! Ce n'est pas grave, vous allez maintenant apprendre à les gérer avec le moteur physique.

Les collisions entre astéroïdes

Pour gérer les collisions, FarseerPhysics utilise encore un autre type d'objet : les formes géométriques, `Geom`. Ces objets sont créés grâce à la classe `GeomFactory` et la méthode correspondant à la forme que vous voulez créer. Dans le cas des astéroïdes, de forme carrée, vous utiliserez la méthode `CreateRectangleGeom ()`.

Tableau 10-2 Paramètres de la méthode CreateRectangleGeom

Paramètre	Description
`PhysicsSimulator physicsSimulator`	Ce paramètre est optionnel. Si vous l'utilisez, la forme géométrique sera automatiquement ajoutée au simulateur physique.
`Body body`	Corps concerné.
`Float width`	Largeur de la forme.
`Float height`	Hauteur de la forme.

Pour activer les collisions, définissez la propriété `CollisionResponseEnabled` de la forme géométrique à `true`. Vous pouvez indiquer le coefficient de réponse au choc (la force qui sera appliquée au corps qui entre en collision) grâce à la propriété `ResitutionCoefficient`.

La portion de code ci-dessous présente le constructeur de la classe `Asteroid` qui crée maintenant une forme géométrique et active les collisions.

```
public Asteroid(PhysicsSimulator physicsSimulator, ContentManager content)
    : base(Vector2.Zero)
{
    base.LoadContent(content, "asteroid2");
    body = BodyFactory.Instance.CreateRectangleBody(physicsSimulator, 32, 32, 1);
    Origin = new Vector2(Texture.Width / 2, Texture.Height / 2);
    GeomFactory.Instance.CreateRectangleGeom(physicsSimulator, body, 32,
    ➥32).CollisionResponseEnabled = true;
    Initialize();
}
```

Vous pouvez relancer le jeu : à présent, les astéroïdes dérivent et entrent doucement en collision.

Les collisions avec le joueur

Les choses vont être un peu plus complexes pour la classe `Player`. En effet, dès que le vaisseau du joueur entrera en collision avec un autre élément, vous ne devrez pas le faire rebondir, mais lui faire recommencer le jeu à zéro. Pour cela, la classe `Geom` met à votre disposition l'événement `OnCollision`.

Un événement se produit quand quelque chose se passe dans un programme : le clic d'un utilisateur sur un bouton, une collision entre deux entités, etc. Un événement doit être traité avec une fonction particulière appelée *event handler*, qui doit avoir une signature bien précise. Vous reliez l'événement à cette fonction en utilisant un délégué (vous les avez déjà rencontrés lors de l'étude des méthodes asynchrones).

L'abonnement à l'événement se fait donc de la manière suivante :

```
public Player(PhysicsSimulator physicsSimulator)
    : base(Vector2.Zero)
{
    body = BodyFactory.Instance.CreateRectangleBody(physicsSimulator, 32, 32, 1);
```

```
        GeomFactory.Instance.CreateRectangleGeom(physicsSimulator, body, 32,
        ➥32).OnCollision += new Geom.CollisionEventHandler(CollisionOccurs);
    }
```

Il ne vous reste plus qu'à ajouter la fonction `CollisionOccurs()`, qui est appelée dès qu'un événement a lieu. Nous avons vu sa signature avec IntelliSense au moment de la création du délégué (figure 10-14).

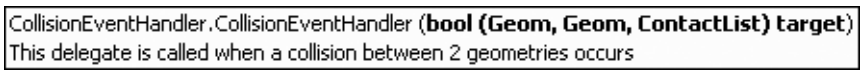


Figure 10-14

Signature de la fonction à créer

```
private bool CollisionOccurs(Geom geomA, Geom geomB, ContactList contactList)
{
    return true;
}
```

La fonction est encore incomplète, le temps de réfléchir un peu aux traitements à effectuer… Si le vaisseau entre en collision avec un astéroïde, il a perdu : il faut le remettre à sa position de départ et supprimer tous les astéroïdes. Il n'y a aucun problème pour appeler la fonction `ResetPosition()` de la classe `Player`. Cependant, vous n'avez pas accès à la liste des astéroïdes. Vous allez donc devoir vous-même faire parvenir un événement à la classe `ChapitreDix_2` pour qu'elle s'occupe de vider la liste.

Commencez par écrire le délégué et créer l'événement.

```
public delegate void ShipHasExplodedEventHandler();
public event ShipHasExplodedEventHandler ShipHasExploded;
```

Puis, appelez l'événement dans la fonction `CollisionOccurs()`.

```
private bool CollisionOccurs(Geom geomA, Geom geomB, ContactList contactList)
{
    ResetPosition();
    ShipHasExploded();
    return true;
}
```

Ensuite, dans la classe `ChapitreDix_2`, abonnez-vous à l'événement et écrivez la fonction qui videra la liste.

```
protected override void Initialize()
{
    physicsSimulator = new PhysicsSimulator();
    ship = new Player(physicsSimulator);
    ship.ShipDestroyed += new Player.ShipDestroyedEventHandler(ship_
    ➥ShipHasExploded);
    base.Initialize();
}
```

```
void ship_ShipHasExploded()
{
    asteroids.Clear();
}
```

Vous pouvez maintenant tester le jeu et vous amuser avec !

En résumé

Dans ce chapitre, vous avez découvert :

- comment gérer les collisions en 2D avec des rectangles ;
- comment gérer les collisions en 2D au niveau des pixels ;
- ce qu'est un moteur physique, et une liste de moteurs utilisables en C# ;
- comment utiliser le moteur FarseerPhysics.