Ancillary Chapter

Invitation to XNA

This ancillary chapter presents Microsoft's XNA game development framework. It discusses the .Net Framework, the C# programming language, the XNA framework implementation and the XNA game studio integrated development environment. The chapter closes with a straightforward XNA framework application.

In this chapter you will learn about:

- The .NET Framework and the Common Language Runtime
- The C# programming language
- The XNA Framework and Game Studio 2.0
- XNA game programming

A.1 Introduction to the .NET Framework

The Microsoft .NET Framework is a Windows development platform originally released in 2002 for Windows 98, Windows NT 4.0, Windows 2000 and Windows XP. It has since been upgraded four times; with the current version being the .NET Framework 3.5 (presently shipping with Visual Studio 2008 and available as separate download from http://www.microsoft.com/net/DownloadCurrent.aspx).

Microsoft's .NET Framework allows for the seamless integration of various programming languages (Managed C++/C#/VB.NET) and is used for numerous applications targeting the Windows platform. Applications written for the .NET framework are executed within a runtime environment called the *Common Language Runtime* (CLR). This runtime environment is comparable to a virtual machine as first popularised by the Java programming language. A *virtual machine*, as the name suggests, is a computer emulation layer responsible for the execution of programs. Virtual machines eliminate one's need to consider underlying system hardware. A virtual machine can thus be summarised as a software-based execution layer responsible for shielding executing programs from both system hardware and operating system specifics. The CLR is actually an implementation of the *Common Language Infrastructure* (CLI) – an open specification defining the software execution environment.

All .NET programs (written in C#, Managed C++, Visual Basic .NET and J#) are compiled to an intermediate language called the *Common Intermediate Language* (CIL) or *Microsoft Intermediate Language* (MSIL). The term "intermediate language" has its roots in compiler design where the compiler is tasked with translating human readable program source code to an intermediate form amenable to code-based optimisations. This intermediate language is in turn translated into a machine readable form for execution. Programs compiled to the CIL/MSIL are assembled into managed code for execution by the .NET Common Language Runtime (the .NET virtual machine). The term "managed code" simply refers to program code executed via a virtual machine as opposed to unmanaged code directly executed by the CPU itself. Figure A.1 shows the organisation of the CLI.

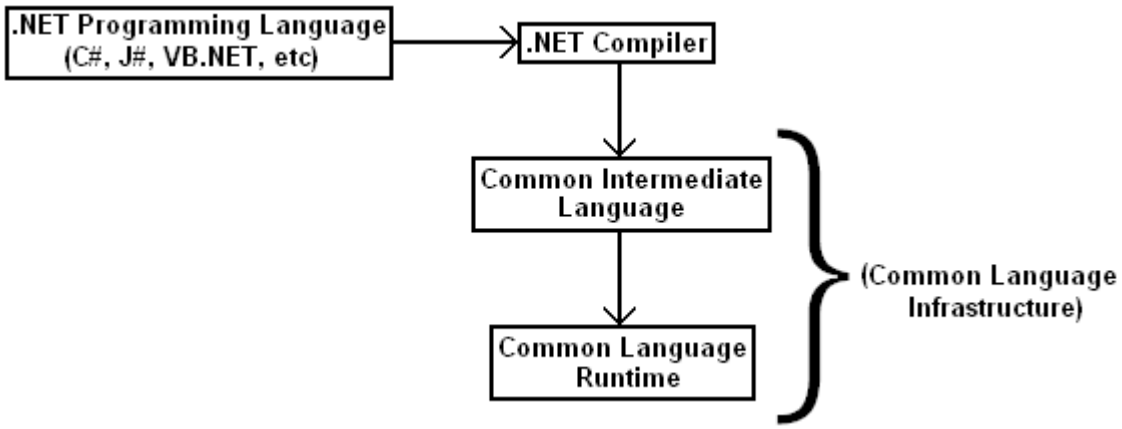

Figure A.1 Organisation of the Common Language Infrastructure.

The Microsoft .NET Framework is the standard and most widely used .NET implementation. The framework (excluding the .NET Base Class Library) is fully documented as an Ecma/ISO specification – resulting in numerous other .NET implementations. The *Base Class Library* is .NET's standard class library, providing .NET programmers with a vast array of common functions for tasks such as file manipulation, database connectivity and resource management to name but a few. Alternative .NET implementations include Novell's *Mono*, Microsoft's *Shared Source Common Language Infrastructure* and *Portable .NET*.

Mono is an Ecma International (European Computer Manufacturers Association International) compliant .NET implementation featuring a C# compiler, the Common Language Runtime and the base class libraries. This implementation currently supports Linux/UNIX/BSD-based operating systems (including Mac OS X), Solaris and Windows. The Shared Source Common Language (SSCLI) is Microsoft's version of an open source CLI implementation. The SSCLI is not open to commercial use and configured to run on Windows, Mac OS X and FreeBSD. SSCLI 2.0 is the most recent version and comparable, in functionality, to the .NET Framework 2.0; however, Windows XP Service Pack 2 is the only supported operating system at time of writing. Portable .NET is another open source implementation of the Common Language Infrastructure. It also features a C# compiler and portions of the .NET Base Class Library.

The .NET Framework simplifies the development process by offering a large collection of helper functions, a common security model, a high degree of language independence and true cross platform compatibility. Programs written for the framework are guaranteed to run, without modification, on each and every platform featuring Microsoft's official implementation (Windows, Windows CE and the Xbox 360) with unofficial framework implementations ensuring even further compatibility.

We will now briefly look at the C# programming language – a modern, general-purpose, object-orientated language that is core to both the .NET Framework and the XNA toolset.

A.2 Introduction to the C# Programming Language

The C# programming language was designed by Microsoft (with Anders Hejlsberg as lead architect) and first released in 2001. It is based on the C++ and Java programming languages with some concepts taken from Visual Basic and Borland Delphi (Hejlsberg was formerly involved with the design of both Turbo Pascal and Delphi). C# is, according to Microsoft's C# programming guide, an "object-oriented language that enables developers to build a wide range of secure and robust applications that run on the .NET Framework." The language is suited for the creation of "traditional Windows client applications, XML Web services, distributed

components, client-server applications, database applications, and much, much more."

C# code is compiled to an intermediate language, the previously discussed CIL, and executed on top of the .NET Framework. This compiled code, coupled with resource files such as sounds and images, is packaged as an executable called an *assembly* (commonly stored as an .exe or .dll). This assembly is loaded into the Common Language Runtime upon execution. The Common Language Runtime is in turn tasked with converting the intermediate language code to native machine language for execution.

A.2.1 A Very Basic C# Program

We will now look at the basic structure of a C# program by printing the string "Hello World!" to the system console.

Our program starts with the '**`using System`**' directive:

```
using System;
```

This directive allows one to use System classes and methods (fundamental classes and methods provided by the .NET Framework Class Library) without the need to fully qualify them. For example, by including the **`using System`** directive we can use '**`Console.WriteLine`**' instead of '**`System.Console.Writeline`**' to write text to the standard output stream.

Following the **`using System`** directive we need to specify a namespace, '**`namespace OurProgram`**':

```
namespace OurProgram
{
```

Namespaces are used to organise classes, enumerations, structs, other namespaces and so forth. A single namespace generally contains a number of classes and/or structs. In the call '**`System.Console.Writeline`**', **`System`** is a namespace with **`Console`** a class within said namespace and **`Writeline`** a method within **`Console`**. These are all of course part of the .Net Framework Class Library.

The entry point of this program is specified using the **`Main`** method. This method is the first method called upon execution of the application and it's used to call other methods and to create objects. The **`Main`** method resides within a class or struct, in this case within a class called **`WriteOutput`**:

```
    class WriteOutput
    {
```

```
        static void Main()
        {
            System.Console.WriteLine("Hello World!");
        }
    }
}
```

The **static** keyword is used to indicate a member (a class or class member) that's accessible without creation of an instance of the class or member (by means of the **new** keyword). For example, we can use the **new** operator to create an object of a class as follows (please refer to the ancillary C++ eBook for more information on objects and classes):

```
ClassName object = new ClassName();
```

The **void** type signals the return of nothing (as the case in the given program), where **int** indicates the return of an integer. As mentioned in the ancillary C++ eBook, programmers refer to **void** and **int** as return types. The compiler expects something to be returned whenever a return type is placed in front of a method name.

To compile the program (if saved as '**HelloWorld.cs**'); execute the following from within its file location:

```
csc HelloWorld.cs
```

To run the program type:

```
HelloWorld
```

It will output the following to the screen:

```
Hello World!
```

This is an extremely basic C# program, with real-world implementations consisting of numerous and significantly more complex C# source files. The given example does serve to highlight the general structure of a C# program and we will revisit C# when considering XNA's dependency on this modern programming language. We will now continue with an introduction to Microsoft's XNA Game Studio 2.0 by looking at the XNA framework and the XNA game studio integrated development environment.

A.3 Microsoft XNA Game Studio 2.0

XNA, short for "XNA's Not Acronymed", is a collection of tools developed specifically for hobbyists and novice game developers. XNA Game Studio currently caters for the development of both Xbox 360 and Microsoft Windows games. Microsoft first introduced the XNA toolset in 2004 with the latest release, XNA Game Studio 2.0, being available as of December 2007. The XNA Framework, as part of the XNA

Game Studio, uses version 2.0 of the .NET Compact Framework as foundation for Xbox 360 development and version 2.0 of the .NET Framework for Windows implementations. The .NET Compact Framework is similar to the .NET Framework for Windows, with the exception of being targeted at mobile and embedded devices. The compact framework shares many of the full .NET Framework's class libraries; it also includes a number of libraries designed specifically with embedded devices in mind.

The XNA Framework, offering a wide collection of game development class libraries, provides developers with a completely portable game execution platform. For example, an XNA-based game written on Windows Vista will run on any platform supporting the XNA Framework (some modification or porting, although rare, may be required depending on the complexity of the title).

The main goal of the XNA Framework is to provide developers with high-level game specific classes and libraries explicitly designed with ease of use and rapid game development in mind. Developers are thus free from low-level technological complexities, allowing them to focus more on content creation and the overall gaming experience than the technological implementation of the game.

Microsoft's XNA Game Studio 2.0 requires a pre-installed copy of Visual Studio 2005 to function. Both Windows XP with Service Pack 2 or later and Windows Vista are supported. XNA-based games require Shader Model 1.1/DirectX 9.0c hardware – not quite the Shader Model 4.0/DirectX10 technology as presented in the textbook, however, the idea behind XNA isn't next-generation titles – XNA's overall goal is to put the fun back in game development. It does this by focussing solely on hobbyists, students and homebrew game developers. Xbox 360 development requires a hard drive connection to an Xbox 360 console as well as membership in the XNA Creators Club (available from the Xbox LIVE Marketplace) including Xbox Live Membership for multiplayer by means of Xbox LIVE or via system link (using a LAN connection between consoles).

We will now look at the steps necessary for creating games using XNA Game Studio 2.0.

A.3.1 Using XNA Game Studio 2.0

Following successful installation of the software, we can create a project by launching Visual Studio 2005 and selecting the "New Project" option under the "File" menu (Figure A.2).

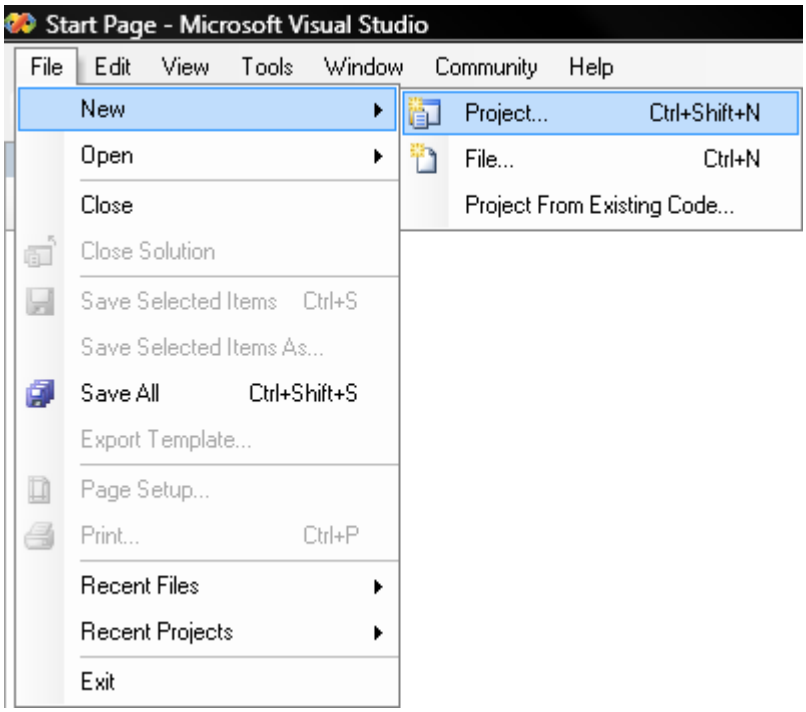

Figure A.2 Step 1: Creating a new project.

A dialog box as shown in Figure A.3 will now appear. Select “Windows Game (2.0)” under the “XNA Game Studio 2.0” tree node. There are also a number of other options such as “Xbox 360 Game (2.0)”, “Spacewar Xbox 360 Starter Kit (2.0)”, etc. Exploration of these options is left to the reader.

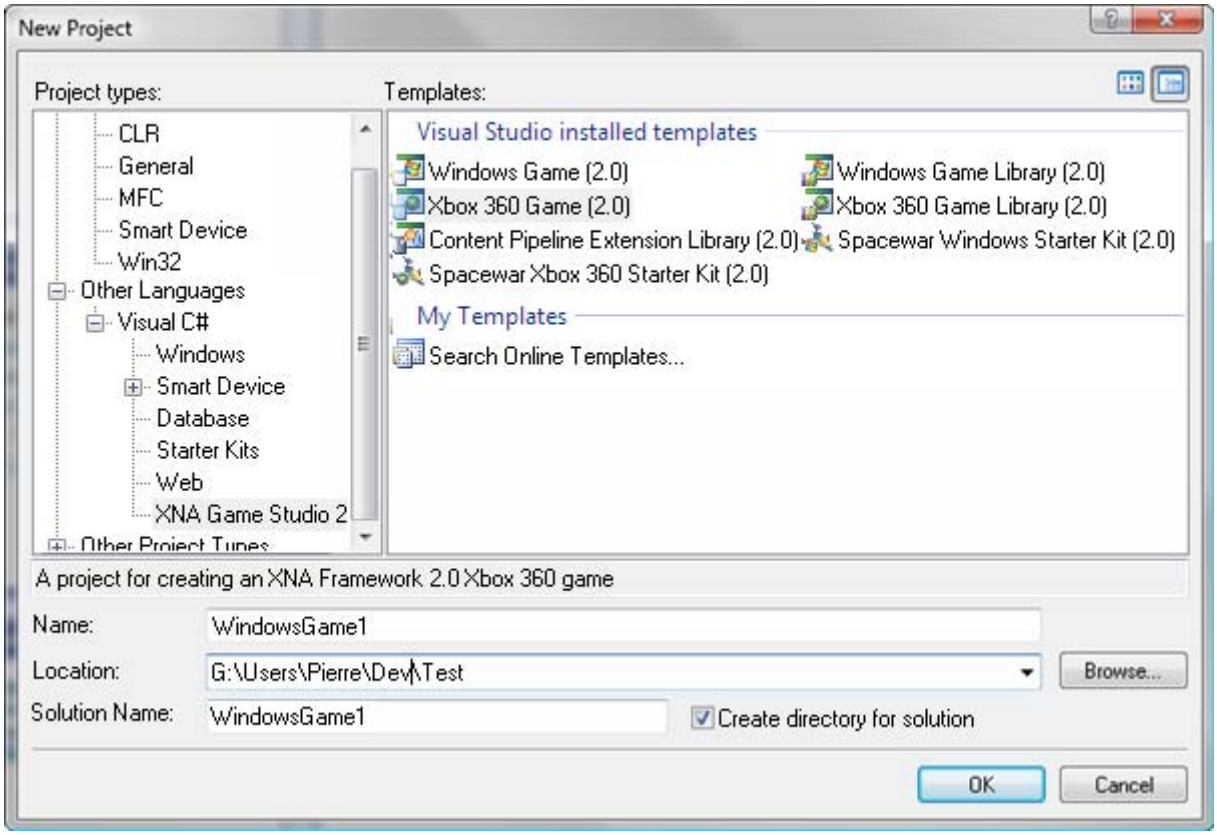

Figure A.3 Step 2: Creating a new XNA Game Studio 2.0 project.

After entering a project name and clicking on the “OK” button, Visual Studio will generate the following code (see the file called “**Game1.cs**”):

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace XNA_Game
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

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

        /// <summary>
        /// Allows the game to perform any initialization it
        /// needs to before starting to run.
        /// This is where it can query for any required
        /// services and load any non-graphic related content.
        /// Calling base.Initialize will enumerate through any
        /// components and initialize them as well.
        /// </summary>
        protected override void Initialize()
        {
            // TODO: Add your initialization logic here

            base.Initialize();
        }
```

```
/// <summary>
/// LoadContent will be called once per game and is
/// the place to load all of your content.
/// </summary>
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to
    // draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: use this.Content to load your game
    // content here
}

/// <summary>
/// UnloadContent will be called once per game and is
/// the place to unload all content.
/// </summary>
protected override void UnloadContent()
{
    // TODO: Unload any non ContentManager content
    // here
}

/// <summary>
/// Allows the game to run logic such as updating the
/// world, checking for collisions, gathering input,
/// and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of
/// timing values.</param>
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back
                                == ButtonState.Pressed)
        this.Exit();

    // TODO: Add your update logic here

    base.Update(gameTime);
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
```

```
        /// <param name="gameTime">Provides a snapshot of
        /// timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

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

The code sample starts with the inclusion of several namespaces. The game's namespace, '**XNA_Game**', is subsequently specified followed by the main class of the game, **Game1**. This class inherits from the XNA Framework class **Game**. C# inheritance is similar to that of C++ (see Chapter 5 of the ancillary C++ ebook) – it reduces the complexity of a program by avoiding the redefinition of classes and member functions (called methods in C# and Java). In essence, one class shares code already defined in another. The class where the code is shared from is known as the *base class* with the class inheriting this code and attributes called the *derived class*. Inheritance is accomplished by placing a colon after the declared class name with the name of the class being inherited from following this colon. The XNA **Game** class (**Microsoft.Xna.Framework.Game**) provides methods for graphics device initialisation, game logic and rendering. Table A.1 lists all of the XNA Framework namespaces as well as a description of each. The detailed study of these namespaces and their classes is, however, beyond the scope of this introductory chapter.

Namespaces	Description
Microsoft.Xna.Framework	Game-specific classes for device initialisation, game logic, pre-calculated mathematical values, etc.
Microsoft.Xna.Framework.Audio	Low-level API calls for the playback of sounds.
Microsoft.Xna.Framework.Content	A number of run-time components such as a content reader used for the loading of game assets, etc
Microsoft.Xna.Framework.Design	A number of classes for type conversions.
Microsoft.Xna.Framework.GamerServices	A collection of "gamer" specific classes, such as classes detailing a player's default settings, profile, etc.
Microsoft.Xna.Framework.Graphics	Low-level API calls for the display of 3D objects.
Microsoft.Xna.Framework.Graphics.PackedVector	A number of structures for the

	representation of vector types.
Microsoft.Xna.Framework.Input	Several input control classes (the Xbox 360 controller, keyboard and mouse are supported).
Microsoft.Xna.Framework.Net	Network specific classes featuring support for Xbox LIVE and multiplayer gaming.
Microsoft.Xna.Framework.Storage	Classes dealing with file storage (the reading and writing of files).

Table A.1 XNA Framework namespaces.

The first line of code in the **Game1** class, "**GraphicsDeviceManager graphics;**", defines a **GraphicsDeviceManager** object for the management and configuration of the game's graphics device. The **GraphicsDeviceManager** object is created and registered in the public method, **Game1** (the constructor of our game). We use this object to render a blank screen within the **Draw** method. A constructor, as discussed in the ancillary C++ ebook, is a type of member function or method, more specifically; it is a method with the same name as the class. A constructor is invoked each time the program creates an object of the class.

The next line of code, "**Content.RootDirectory = "Content";**", sets the base path of the content build process. This "Content" folder is basically the 'root' folder where all the game assets (like textures and models) will be loaded from. For example, the generated code available from the book's website uses the folder "**...XNA Game\Content**" for this purpose. We can also specify the Content folder manually as shown in the following line of code:

```
gameTexture = Content.Load<Texture2D>("Content\texture1");
```

This code snippet loads a texture with the filename "**texture1**". Thus, setting the content base path via the **RootDirectory** property will allow us to load this texture without specification of the Content folder:

```
gameTexture = Content.Load<Texture2D>("texture1");
```

Following this we have a method called "**Initialize**". This method is responsible for all game-specific initialisations and content loading as needed prior to the game's start-up. The **protected** access modifier specifies that the **Initialize** method can only be accessed within its own class, **Game1**. The **Initialize** method can also be accessed by derived classes. Going back to the **Game1** class, we can see that it was declared using the **public** access modifier. This modifier doesn't impose any restrictions on accessibility. The **Initialize** method is also declared using the **override** modifier. This modifier allows us to extend and/or modify abstract or virtual implementations of inherited methods. An abstract or virtual implementation is basically a method declared using the **virtual** keyword. This virtual or abstract

method can subsequently be overridden by classes inheriting it. For example, say we have the following method declaration:

```
public virtual double Damage()
{
     return playerDamage;
}
```

We can now, using the **override** keyword, change or extend this virtual implementation in the class inheriting our **Damage** method. Methods in derived classes can thus now have the same name as those inherited from base classes.

Back to the overridden **Initialize** method. According to the XNA Game Studio 2.0 documentation we must override this method (defined in Microsoft.Xna.Framework.Game assembly) to "query for any required services and load any non-graphics resources".

The next method, **LoadContent**, is similar in nature – it's also overridden and declared using the **protected** access modifier. This method allows us to load game content such as graphics resources like textures and billboards/sprites. The **LoadContent** method is originally defined in the Microsoft.Xna.Framework assembly and overridden here to provide us with the required resource-loading functionality. We also have an **UnloadContent** method responsible for the unloading of all game-specific graphics resources. This method is also defined in the Microsoft.Xna.Framework assembly and overridden here.

The second last method, **Update**, updates all of the game logic (collision detection and response, input processing, audio playback, etc). It is also a **protected override** method with the original defined in the Microsoft.Xna.Framework.Game assembly. This method takes a single **GameTime** parameter, namely **gameTime**. The **GameTime** class type, defined in the Microsoft.Xna.Framework.Game assembly, allows us to take a snapshot of the "game timing state expressed in values that can be used by variable-step (real time) or fixed-step (game time) games". The first portion of code in the **Update** method allows us to exit the game by pressing the back button on a connected Xbox 360 controller. The second portion of code, "**base.Update(gameTime)**", is simply added to ensure that the game logic is processed as determined by the game.

The final method, **Draw**, is called when the "game determines that it is time to draw a frame". This method is overridden here to ensure the execution of game-specific rendering routines. A snapshot of timing values is once again received as method parameter. The first line of code, "**graphics.GraphicsDevice.Clear(Color.CornflowerBlue)**", sets the window's background colour to light blue with the second line, "**base.Draw(gameTime)**", called to ensure enumeration through and initialisation of all graphics components.

There is also a generated C# source file called "**Program.cs**". This file, serving as entry point for the application, creates an object, **game**, of **Game1** type. This object is used to call the **Run** method (defined in the Microsoft.Xna.Framework.Game assembly) which will lead to the game being initialised (in turn resulting in the initialisation of the game loop and the processing of game events):

```
using System;

namespace XNA_Game
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
    }
}
```

Executing the code opens a window with a light blue background as shown in Figure A.4.

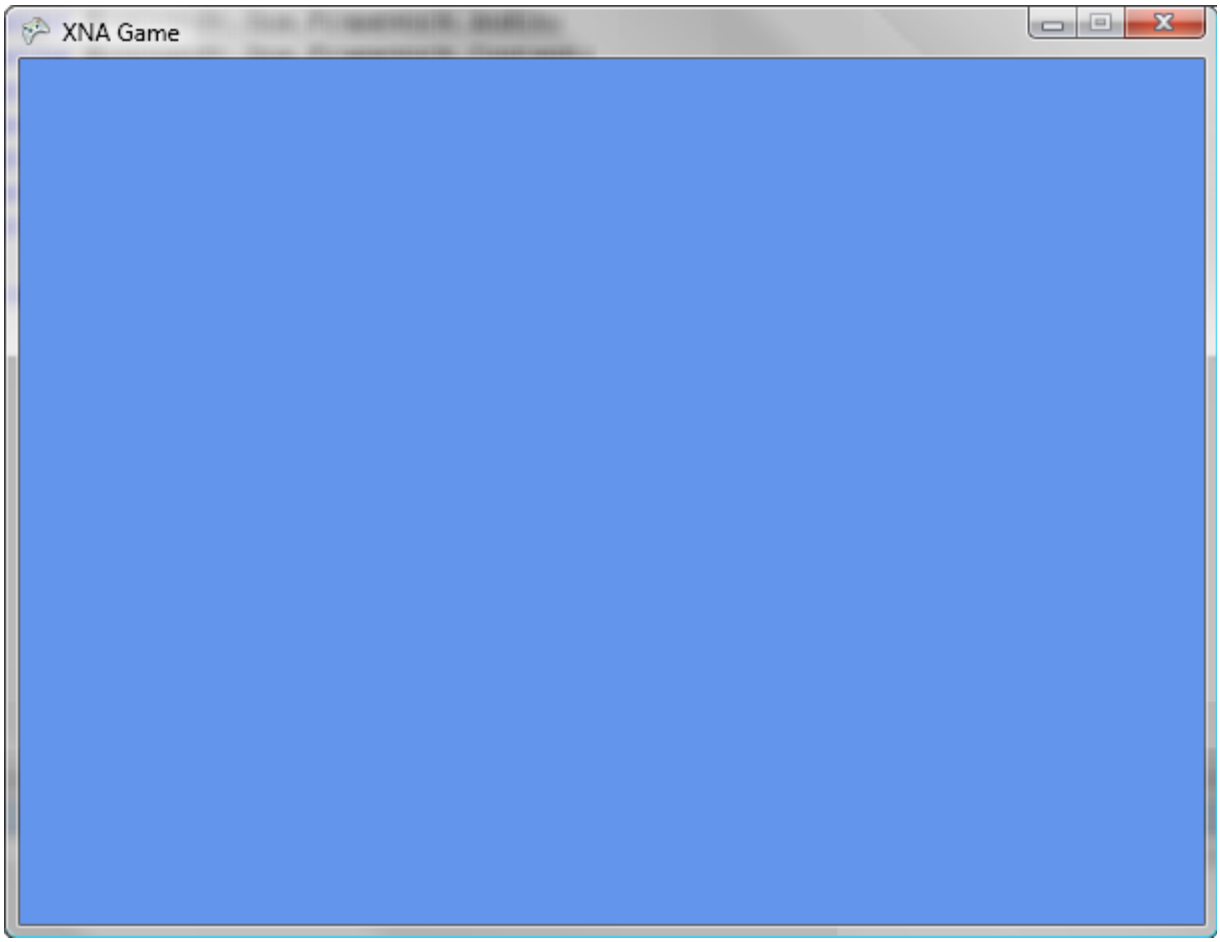

Figure A.4 Our XNA application window.

Microsoft's official XNA Game Studio 2.0 documentation details extension of the above generated code to include a sprite object. The first step in this process involves right-clicking on the "**`Content`**" node in the projects solutions explorer window (see Figure A.5).

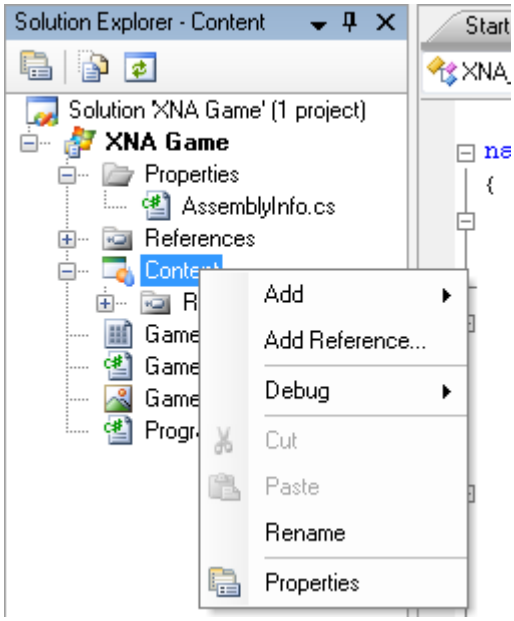

Figure A.5 Step 1: Adding new content.

We can now click on “Add” followed by the selection of “Existing Item” (shown in Figure A.6):

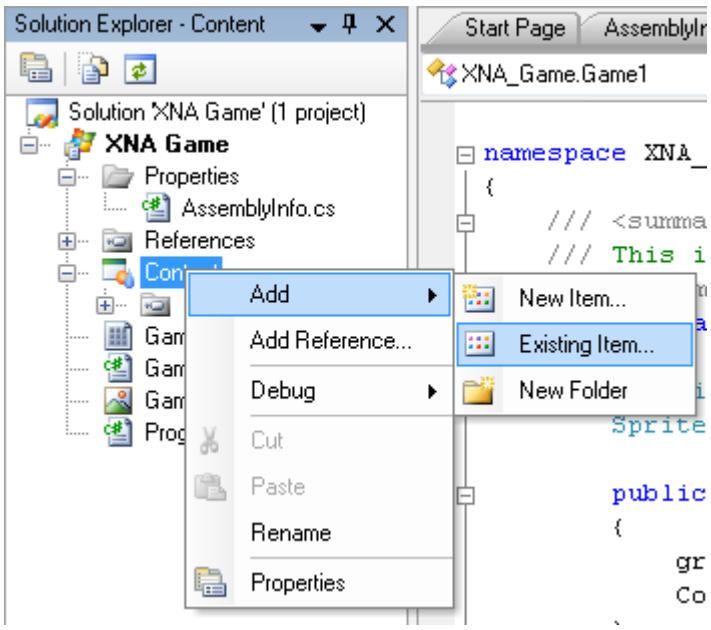

Figure A.6 Step 2: Adding new content.

This opens the dialog shown in Figure A.7 – we’ll be opening the TGA graphics/texture file, “sprite.tga”, located in our project’s “Content” folder.

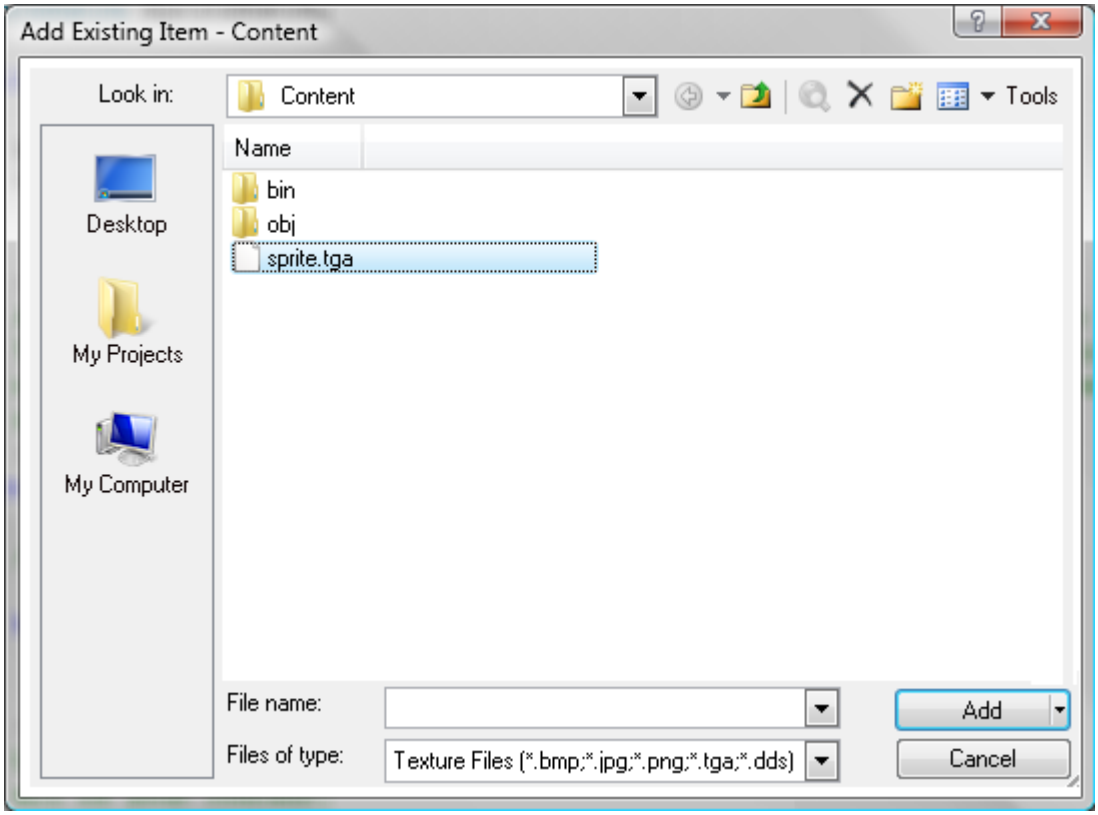

Figure A.7 Loading the texture file.

The file, “sprite.tga”, will now be listed in the project’s solutions explorer window (Figure A.8).

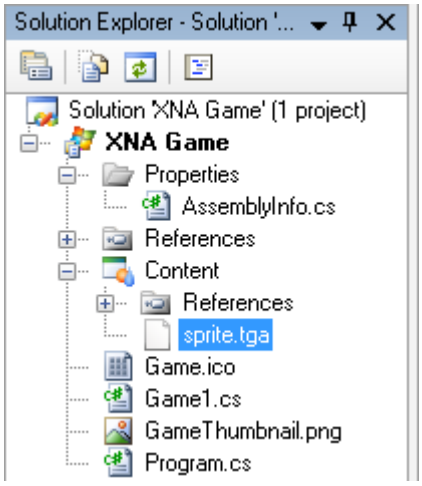

Figure A.8 The loaded image content.

The next step is to set the file’s “Asset Name” property (our asset name value has been set to “sprite”). Asset name properties are used to load files into XNA-based games. The “Asset Name” is accessible via the “Properties Window” (Figure A.9).

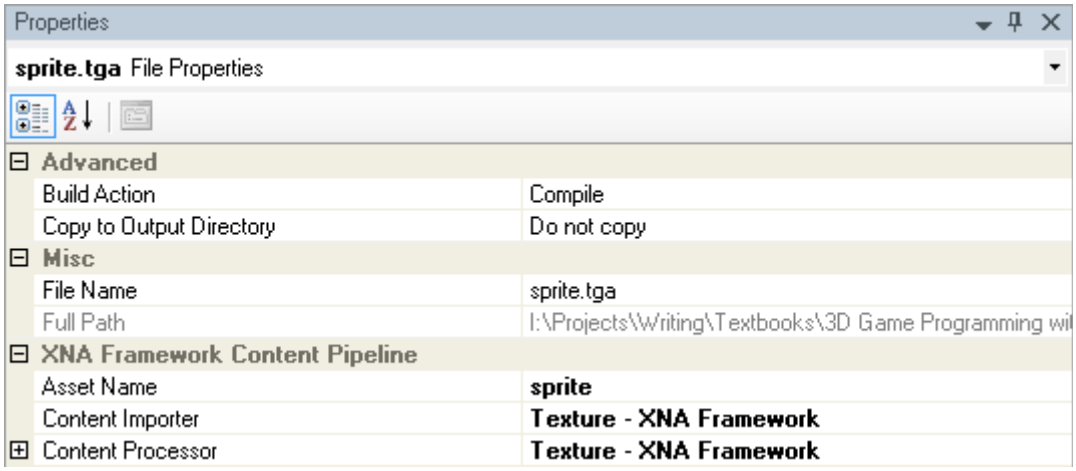

Figure A.9 The Properties Window listing the “Asset Name”.

Similar to the example given in the XNA Game Studio 2.0 documentation, we add the following to load and display the image “sprite.tga” on screen:

```
//the sprite texture
Texture2D spriteTexture;

//the rendering position
Vector2 spritePosition = Vector2.Zero;

//the modified LoadContent method
protected override void LoadContent()
{
```

```
        /* Create a new SpriteBatch, which can be used to draw
           textures.*/
        spriteBatch = new SpriteBatch(GraphicsDevice);

        spriteTexture = Content.Load<Texture2D>("sprite");
}
```

The **Texture2D** class is used to define an uninitialised texture resource – in turn initialised using the local texture asset name property (via the **Content.Load** method). We also declare and initialise a position vector, **spritePosition**, which will be used to position the image during rendering. The **Vector2.Zero** property sets the x- and y-component of our vector to zero. The first new line in our **LoadContent** method declares a new **SpriteBatch** object which can be used to draw numerous sprite images to our screen – all using the same settings.

The final step is to draw the image on the screen. This basically requires modification of the given **Draw** method:

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

        // Draw the sprite.
        spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
        spriteBatch.Draw(sprite, spritePosition, Color.White);
        spriteBatch.End();

        base.Draw(gameTime);
}
```

Using the previously declared **spriteBatch** object, we start by enabling alpha blending (using the **Begin** method with "**SpriteBlendMode.AlphaBlend**" as parameter). Following this, we use the **Draw** method to add the "sprite.tga" image to the batch of images to render – we simply specify the texture, the screen position and colour tint of the image as parameters (this tint simply specifies colour effects that can be applied to the texture – "**Color.White**" is used to draw a texture without any effects). The final method call, "**spriteBatch.End()**", simply flushes the image batch, restoring the device state as it was before the **Begin** method call.

Executing our modified code example opens a window as shown in Figure A.9.

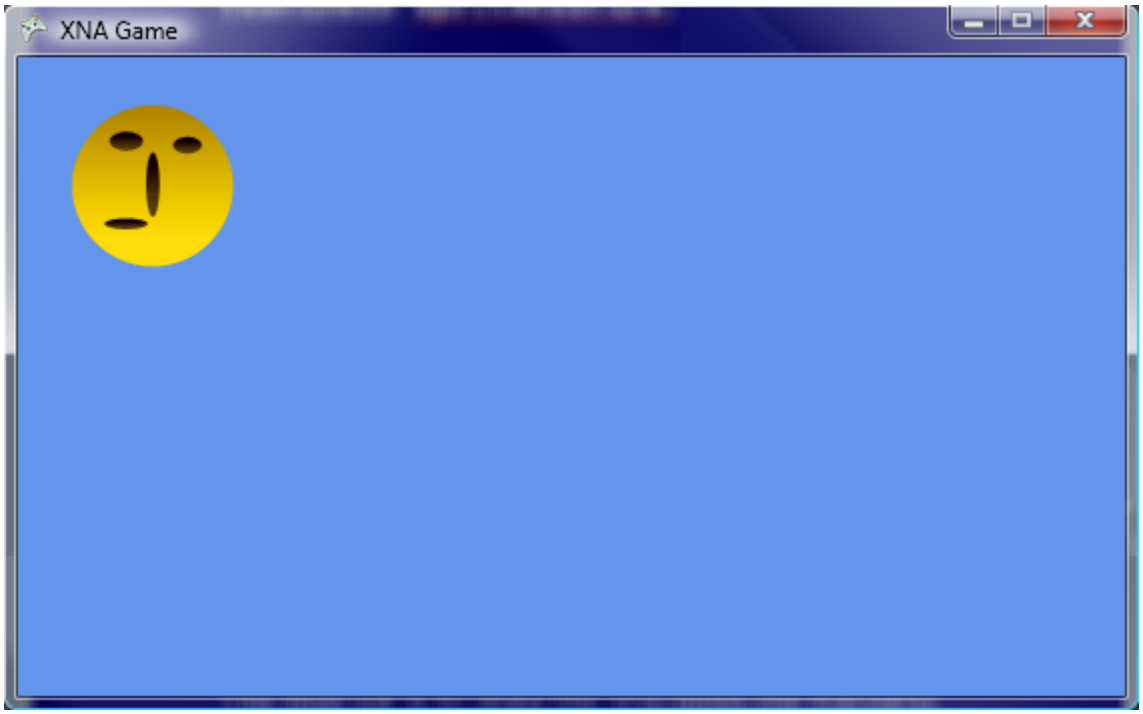

Figure A.9 The image loaded into our XNA application window.

We can now proceed with the animation of our sprite. More specifically, the official Microsoft XNA documentation adds the method "**UpdateSprite(gameTime)**" to the generated **Update** method:

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

     // Move the sprite around.
     UpdateSprite(gameTime);

     base.Update(gameTime);
}
```

This **UpdateSprite** method (taken from the official XNA Game Studio 2.0 documentation) uses the window boundaries and elapsed game time (**gameTime**) to increment or decrement the object's x- and y-position based on the object's current position in relation to the window edges (just resulting in our sprite being "bounced" from the window edges):

```
void UpdateSprite(GameTime gameTime)
{
    // Move the sprite by speed, scaled by elapsed time.
    spritePosition += spriteSpeed *
```

```
                (float)gameTime.ElapsedGameTime.TotalSeconds;

    int MaxX = graphics.GraphicsDevice.Viewport.Width –
               myTexture.Width;
    int MinX = 0;

    int MaxY = graphics.GraphicsDevice.Viewport.Height -
               myTexture.Height;
    int MinY = 0;

    // Check for bounce.
    if (spritePosition.X > MaxX)
    {
        spriteSpeed.X *= -1;
        spritePosition.X = MaxX;
    }

    else if (spritePosition.X < MinX)
    {
        spriteSpeed.X *= -1;
        spritePosition.X = MinX;
    }

    if (spritePosition.Y > MaxY)
    {
        spriteSpeed.Y *= -1;
        spritePosition.Y = MaxY;
    }

    else if (spritePosition.Y < MinY)
    {
        spriteSpeed.Y *= -1;
        spritePosition.Y = MinY;
    }
}
```

This concludes our invitation to Microsoft's XNA. We've only briefly scratched the surface of the C# programming language and XNA Game Studio 2.0, however, the information given here should be more than sufficient as a starting point for XNA game programming.

A.4 Further Reading

There are several great books detailing Microsoft's C# programming language. One such a book is *C# 3.0 in a Nutshell: A Desktop Quick Reference by Joseph and Bend*

Albahari [ISBN: 0596527578]. *Beginning Microsoft Visual C# 2008 by Karli Watson et al* [ISBN: 047019135X] and *Pro C# 2008 and the .NET 3.5 Platform by Andrew Troelsen* [ISBN: 1590598849] also deal with C# in detail. The C# programming language is officially documented in Microsoft's MSDN libraries complementing the Visual Studio IDE. This documentation can alternatively be accessed from Microsoft's msdn.microsoft.com website (http://msdn.microsoft.com/en-gb/vcsharp/default.aspx).

The official XNA documentation and numerous example applications and tutorials are provided with the XNA 2.0 SDK distribution (http://creators.xna.com/en-US/downloads). The official XNA Game Studio 2.0 product documentation is also available online at Microsoft's XNA Developer Centre (http://msdn.microsoft.com/en-us/library/bb200104.aspx).