

## 4

### Displaying Images

- Find out how the Content Manager lets us add pictures to XNA games.
- Discover how pictures are manipulated in games programs.
- Display our pictures on the screen.
- Make a better version of Color Nerve, and an even more groovy Mood Light.

### Introduction

Our understanding of computers and programs is coming along nicely. We are starting to get a grasp of classes, methods and data, as well as the C# constructions that let our programs make decisions based on the values in our variables. We also know how to read information from gamepad and keyboard, and use this to change what a game does when it runs.

In this chapter we are going to add the ability to use images in our programs, improve Color Nerve so that it lets us use our own pictures, and make an even more impressive Mood Light.

---

### Program Project: Picture Display

Pictures in games are always nice. The Xbox is extremely good at manipulating images on the screen. Many games make use of image resources which are then used to generate the view the player sees. In this project we are going to display a picture on the Xbox screen. Once we have got some of our images into the Xbox we can see about using them in games.

---

To do this very simple thing we are actually going to have to work quite hard:

1. We have to get the picture that we wish to draw into our game project, so that it becomes part of the program when it is loaded into the Xbox.
2. Then we have to add code to the program which will fetch the image into the program when it runs.

3. Next we have to tell XNA whereabouts on the screen the image is to be drawn.
4. Finally we can go ahead and actually draw the item.

The good news is that whilst we are learning how to do this we are actually finding out a lot about how games, C#, XNA and the Xbox actually work.

## Resources and Content

In the early days of computers a program simply read in numbers and printed out results. Things have moved on a bit since then, and now computer programs can work with images, video and sound. This is particularly useful where games are concerned, a large part of the enjoyment of a game can come from an attractive environment. And of course sometimes the graphics themselves form part of the game-play. If you want to become a game developer you will need to know how these resources are made part of your program. In fact many programs nowadays have significant graphical content in the form of splash screens, icons and the like. So the first thing we need to do is get some images and incorporate them into our project. Later on we will move on to other forms of resource, including fonts (for writing text) and sounds. Unfortunately I'm not going to be able to help you can create your graphics for use in computer games. I have no artistic abilities whatever, although I do know how to use a camera. My advice is if you need artistic resources, find someone who is good at art and commission them to do the drawings for you. The same goes for any music or sounds that you might need.

This means that you can concentrate on what you are supposed to be good at, which is create the game itself. This is what professional game developers do. They have a team of programmers who make the game work, and a team of artists who work on the look of the game. Having said that, you might be good at graphic design as well as programming, in which case you can do both. I'd still advise getting an artist involved though; it helps spread the work around and provides you with a useful sounding board for ideas. It also makes it more fun.

## Getting Some Pictures

At this point in the course you are going to need some pictures. There are a number of different formats for picture storage on computers. Your pictures should be in the PNG (Portable Network Graphics) format.

The Xbox screen is capable of showing very high resolution images. A high resolution image is made up of a large number of dots, or *pixels*. Modern cameras can create very high quality images which thousands of pixels in height and width. However, from a game point of view we want to make the images as small as we can. This reduces the amount of memory they consume and also reduces the work required to move them around the screen. For our games we don't always need very high resolution, so our pictures need be no more than 600 pixels in each direction.

If you have no pictures of your own (which I consider highly unlikely) you can use the ones that I have provided with the sample files for this chapter, but the games will work best if you use your own pictures. Figure 4-1 shows my picture of Jake, I will be using this for my first Xbox graphics programs. You can use another picture if you wish.



**Figure 4-1**  
*Jake*

I have saved the image in the PNG file format with a width of 600 pixels. If you need to convert into this format you can load an image using the Microsoft Paint program

and then save it in this form. The Paint program will also perform resizing for you if you want to reduce the number of pixels in the image.

## Content Management using XNA

As far as XNA is concerned things that make games more interesting (images, sounds, 3D models, video) are just *content*. XNA treats items of content in the same way as we create variables in our programs. It can import a content item of a particular type (for example my file containing a picture of Jake) and give it an identifier. When the game program is running XNA fetches the game content items as they are requested by name. These content items are sometimes referred to as *assets*. In the same way that a company has assets such as buildings, machinery and staff, a game has assets such as sounds and images.

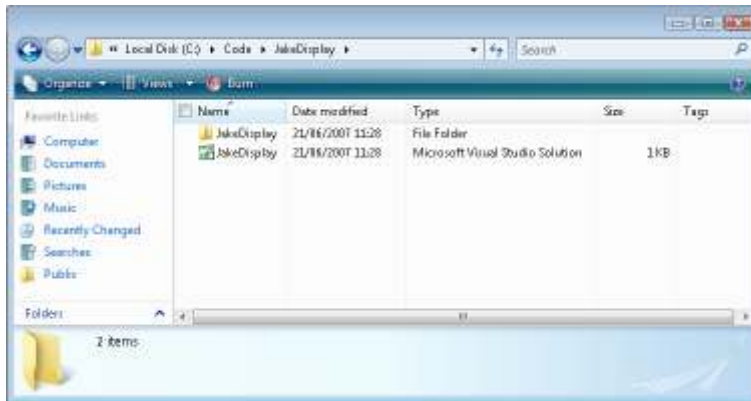
## Working with Content using Visual Studio

We put content into our game using Visual Studio. When the finished program is constructed Visual Studio will merge the assets are into a single file that is transferred into the Xbox to run our game. This file is called an *assembly*. The good news for us is that we don't need to worry about any of this, all we have to know is how to load resources into Visual Studio and get hold of them from within our game programs.

## Visual Studio Solutions and Projects

We start making a game by creating a brand new project. I called mine `JakeDisplay`. We create the project using the New Project dialog as we have done for all our previous projects. You can see this dialog in use in Figure 1-16 in chapter 1. Note that the "Create directory for solution" option is selected on this dialog. Whenever you create a project you should ensure that this option is checked. This will create a directory structure which contains the program and all the other items that are required to make the game work.

Figure 4-2 shows what is created when I make a new project called `JakeDisplay`.

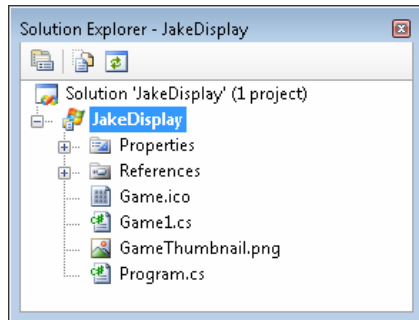
**Figure 4-2***JakeDisplay solution directory*

However, the file `JakeDisplay` that you can see in the directory is actually a *solution*. This is confusing. We've just used the "New Project" command in Visual Studio and we have ended up with a solution. What has actually happened is that Visual Studio has created a solution called `JakeDisplay` and then added a single project to that solution. The project is also called `JakeDisplay`. You can think of a solution as a "shopping list" of projects. Figure 4-3 shows how this works. The solution holds a list of the names of project files. Each of the project files holds a list of the names of the files used in that project. Each item on the list is often referred to as a *reference* to that item, in that it will tell Visual Studio how to get to it.

**Figure 4-3***JakeDisplay solution*

The solution file just holds the name the `JakeDisplay` project. The project file holds the names of the `C#` files in the project (`Game1.cs` and `Program1.cs`) and other resources used by the project. At present the only two resources present are `GameThumbnail.png`, which is an image used as a thumbnail on the Xbox display when the game is stored on the Xbox, and `Game.ico`, which is the icon used for the game program file. When we add our image of Jake to the project we will actually

just add the name of the file to the project file so that Visual Studio knows where to get the asset from. Visual Studio displays the contents of the solution and project files as a diagram in the Solution Explorer, as show in Figure 4-4. Note that the solution file and project files also contain other settings (the Properties and References) which we will be making use of later.



**Figure 4-4**  
*JakeDisplay in Visual Studio Solution Explorer*

Sometimes we may want to add additional projects to a solution. This is so that we can reuse code. For example we might make a project which we called `HighScoreManager`. This would be in charge of displaying high score tables for our game. The way that high scores work is the same across lots of games so it makes sense to only write the code once, and then use it in lots of games. The way that we would do this would be to create a library project to deal with the high scores and then add this project to the “shopping list” of those solutions. However, for now we are just going to create games which are single projects.

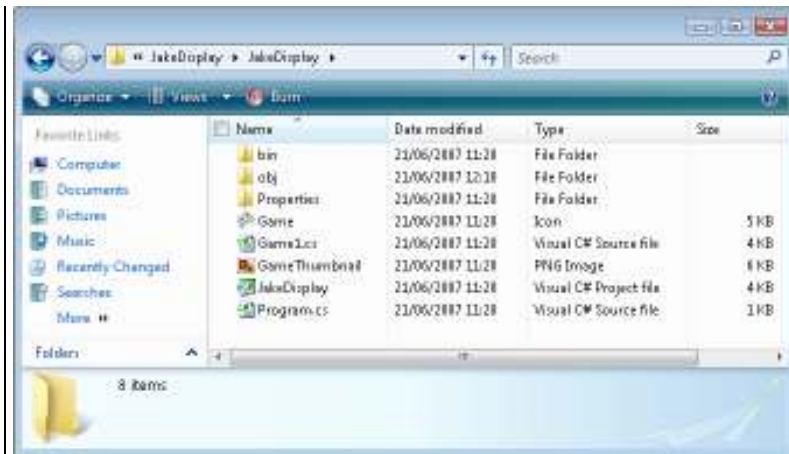
**The Great Programmer Speaks: Architecture is Important**

Our Great Programmer is very keen on using projects to re-use code. The way she sees it, that way she can get paid several times for writing the same piece of software. When she starts work on a new system she takes a lot of time to try and structure things into projects so that different parts of the system are in separate projects.

**Adding Resources into a Project**

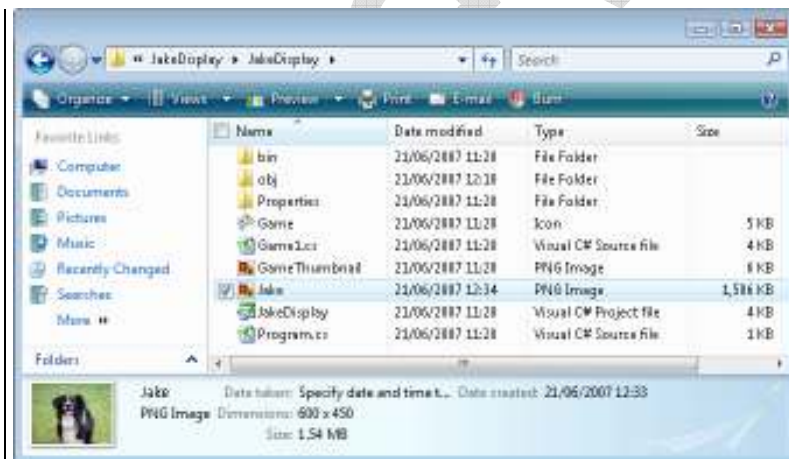
A Visual Studio project will contain references to all the things that it uses. To keep things simple we can keep all the things used by a project in a single file directory.

Figure 4-5 shows the content of the JakeDisplay project directory that Visual Studio created for us when we made the new project. You can see the C# source files and also some other resources.



**Figure 4-5**  
*The contents of the JakeDisplay project*

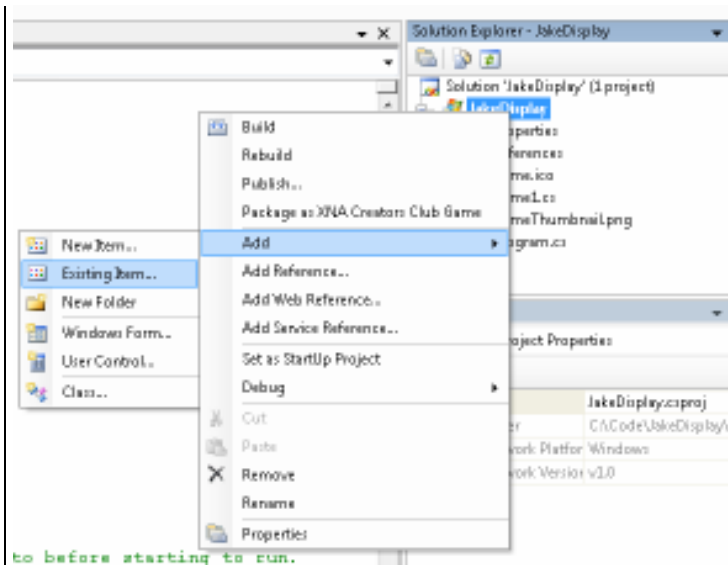
We can place our image file into this directory so that it can be picked up and used in our game. Figure 4-6 shows the directory once I have placed the image in the directory.



**Figure 4-6**  
*The contents of the JakeDisplay project with the Jake image added*

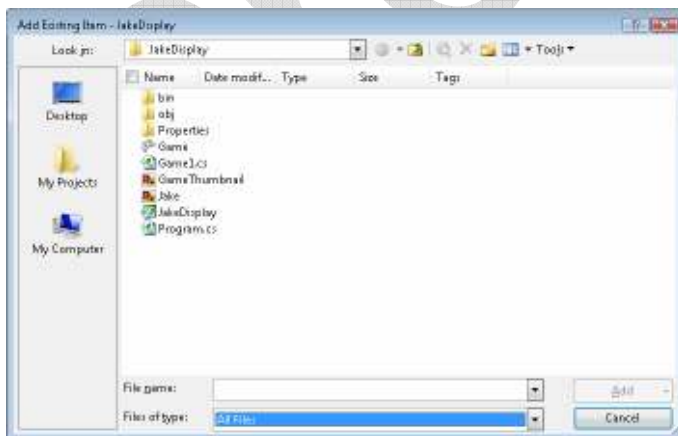
I created a PNG (Portable Network Graphics) image which I placed in the JakeDisplay project directory. You can either use the one of the graphics images that are available in the sample projects or you can create your own picture at this

point. Now we have our graphics resource we can actually tell Visual Studio to use it. To do this we need to add a reference into the project to tell it about a resource we want to use. Resource references are added by using Add Existing Item dialog which can be opened as shown in Figure 4-7. Start by right clicking the JakeDisplay project. From the menu which appears select Add, and then select Existing Item.



**Figure 4-7**  
*Opening the Add Existing Item dialog*

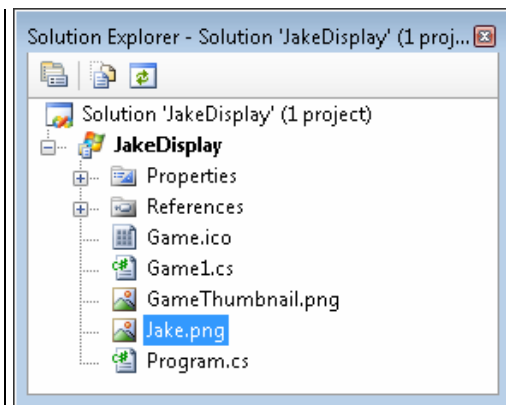
Figure 4-8 shows the dialog which we can use to select an item to add to the project.



**Figure 4-8**  
*The Add Existing Item dialog*

Note that I have changed the selector at the bottom of this dialog to show Files of type "All Files". This makes the graphics resources visible. Now we can select the

image file that we copied into the directory and press the Add button to add it. The project now contains a reference to the resource. Figure 4-9 shows the resource reference in the project once we have added it. Note that this does not make the resource “part” of the project as such, it just adds the resource to the “shopping list” of items to be found and placed in the output assembly when the project is built. You can follow exactly the same process to add other images to a game.



**Figure 4-9**  
*The JakeDisplay project containing the image resource*

If you want to add more than one image to a project you just repeat the process. Remember that each image is actually stored as part of the game program, so the more images that we add the larger our game becomes, and the longer it takes to be transferred into the Xbox when it runs.

### **The XNA Content Pipeline**

This process of feeding resource in at one end and getting a complete game assembly out of the other is a bit like a *pipeline*. In fact the XNA Framework refers to this part of the game building process as the Content Management Pipeline.

## **Using Resources in a Game**

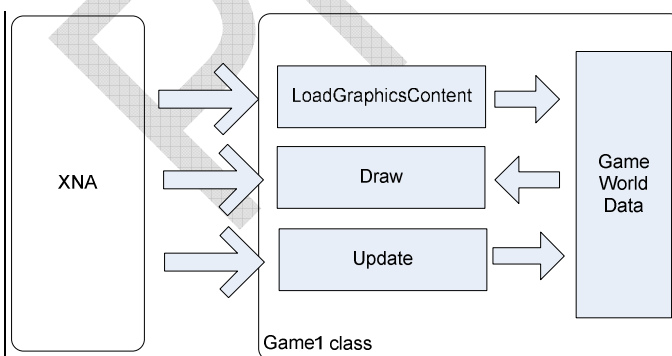
We have done a lot of hard work, but we still haven't got our Xbox to draw any pictures. If we run the solution that we have created we just get the familiar blue screen. Next we have to write some C# which fetches the image resource and draws it on the screen at a particular position.

## Loading XNA Textures

Within XNA images that you want to draw in your games are called *textures*. Textures can be drawn as flat pictures and they can also be wrapped around 3D models. We have already seen how to use the XNA `Color` type, which lets us manipulate color information. Now we are going to use another type of XNA data store allow us to work with our picture as a texture. XNA provides a range of types which are used to deal with textures. The type we are going to use is called `Texture2D`. This holds a texture that we are going to manipulate in two dimensions, i.e. it will be drawn on a flat surface.

We are going to use exactly the same program structure as we used for previous games, we are going to have members of our game class which will represent the “Game World”. These will be updated by the `Update` method and used by the `Draw` method to draw the output.

However, we are going to also make use of another method which lets the program get control when the graphics need to be loaded. Figure 4-10 shows how this works. It is a more detailed version of Figure 2-5 which we saw in Chapter 2, which showed how XNA calls the `Draw` and `Update` methods as a game runs. It shows that there is also a `LoadGraphicsContent` method which is called by XNA when a game starts running.



**Figure 4-10**  
*The Game1 class with the LoadGraphicsContent method*

We can think of `LoadGraphicsContent` as another person in the `Game1` office. They have their own telephone and when it rings they are told whether or not they need to load all the content and make it ready for use.

```
protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent)
    {
        // TODO: Load any ResourceManagementMode.Automatic content
    }

    // TODO: Load any ResourceManagementMode.Manual content
}
```

The method is provided with a *parameter* which tells it whether or not to load all the graphics content. A parameter is something which is fed into a method to tell it something, we have seen these several times.

In this case the parameter is called `loadAllContent` and is of type `bool`. This means it can be either true or false. We can use a `bool` value directly within a condition and use it to control the execution of a block of statements. We have even been given a comment to tell us where to place the code that loads our texture.

```
// The game world
Texture2D gameTexture;

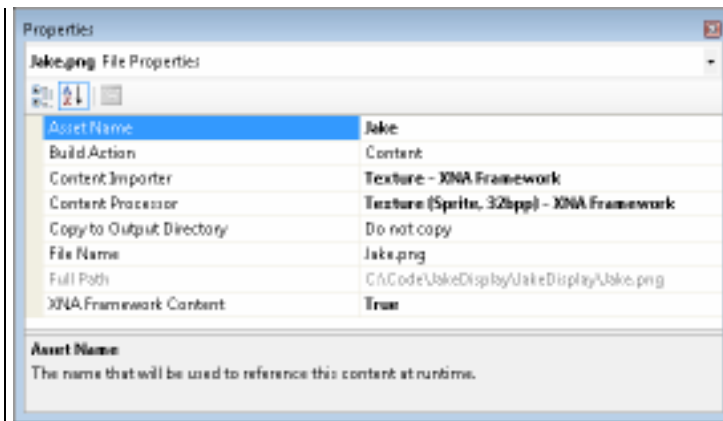
protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent)
    {
        gameTexture = content.Load<Texture2D>("Jake");
    }

    // TODO: Load any ResourceManagementMode.Manual content
}
```

The above code shows how the texture is loaded. At the moment our game world is just the single image. When the game starts XNA will call the `LoadGraphicsContent` method with this parameter set to `true`. The method will then obey the statement that loads the texture content.

```
gameTexture = content.Load<Texture2D>("Jake");
```

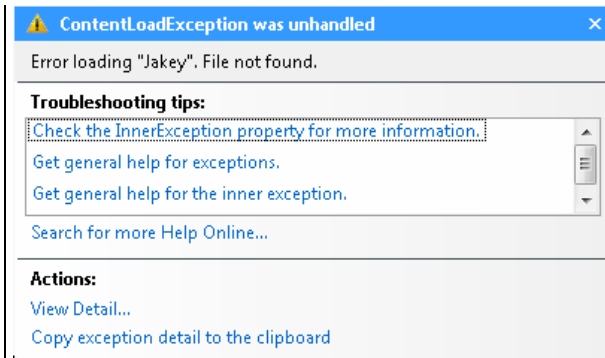
The `Load` method is a kind of multi-purpose tool. It can be used to fetch any kind of item, from textures to audio files to 3D models. We get `Load` to fetch us a `Texture2D` by placing the name of the type we want after the method name. We then give the method the of the asset that we want it to fetch. If you select the `Jake.png` item in Solution Explorer as shown in Figure 4-9 and then look in the Visual Studio Properties pane (it should be in the lower right of the Visual Studio window) you can see that the asset name has been taken from the filename of the resource. Figure 4-11 below shows the property information for the Jake image resource.



**Figure 4-11**  
*Jake image file properties*

This property information tells Visual Studio where the image file is located, what to do with the file when the project is built and also the name to use in the program. So, once the `Load` method has completed we have a copy of the image in the texture in our game assembly. If the game had lots of different images we would declare additional `Texture2D` items in our game world and load them in this method as well.

If we get the name of the texture wrong the game program will fail in this method, as it will be looking for an asset that is not there. The program fails by throwing an exception. Figure 4-12 shows the error that is produced if the asset name of a content item is incorrect.

**Figure 4-12***Texture file not found exception*

Later we shall find out how to get control when things go wrong like this, for now you should make sure that the asset name you use in the call of `Load` should match the name of the content item.

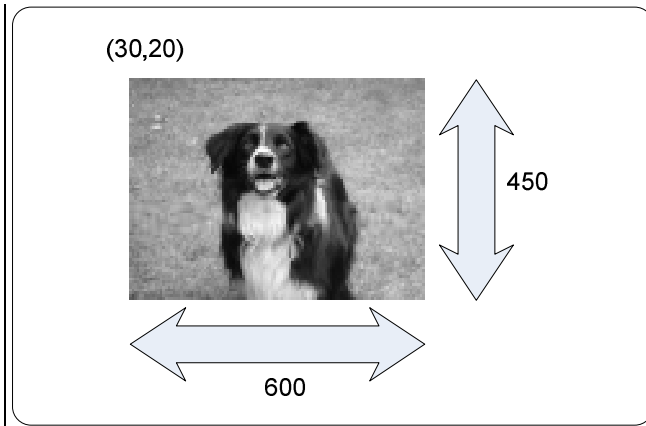
### **The Great Programmer Speaks: Always Worry About Things Going Wrong**

Our Great Programmer spends a lot of time worrying about things which might go wrong. She reckons that in a commercial application, for example one which might be used in a bank, she has to write at least as much program code to deal with all the errors as she writes to do the actual job. Games programs are probably not as critical as bank code, in that if they go wrong nobody will actually lose any money, but if a game constantly crashes it will never become popular. We will see later how we can make sure that our program fails as little as possible.

## **Positioning our Game Sprite on the Screen**

In computer gaming terms our image of Jake is actually a *sprite*. A sprite is a flat, pre-loaded, image which is used as part of a computer game. Sprites can be large, for example the background sky, or smaller, for example the space ships and missiles in a space shooter game. From the point of view of XNA a sprite is an image resource along with location information which tells XNA where to draw the image. This means that we a way to need to tell XNA where on the screen that we want to put our sprite. We do this using yet another XNA type, the `Rectangle`. This holds information about the position and size of a rectangle. We don't need to worry how a rectangle works at the moment, we just need to know how to create one and set the

size and position of it. Figure 4-13 shows how we use a rectangle to express where on the screen we want Jake to be drawn.



**Figure 4-13**

*Placing a draw Rectangle on the screen*

The position of the rectangle is given by the coordinates of the top left hand corner of the screen. You can regard the screen as a piece of graph paper. We express a position on the screen by giving an X coordinate value (the distance across the screen from the left) and a Y coordinate (the distance **down** the screen from the top). This means that the position with the coordinate of (0,0) is the **top** left corner. Note that this is not quite the same as graphs that we may have drawn in the past. In a conventional graph the Y coordinate increases as you go up the page. In computer graphics, when the Y value is increased this moves you down the page.

In Figure 4-13 you can see that the top left hand corner of our Jake sprite is at position (30, 20). This means 30 steps across and 20 down. The units are *pixels*. Pixel is an abbreviation of “picture element” and means the smallest dot that can be drawn on the screen. The Xbox can drive displays with a range of different sizes, and so the pixel at position (30,20) may be a different distance across the screen, depending on the actual screen being used. Later we will find out how to write games which automatically scale themselves to fit any screen.

A rectangle is also used to give the width and height of the sprite, in Figure 4-13 above I am drawing the sprite in an area which is 600 pixels wide and 450 high. The good thing about this is that I don’t have worry about the original size of the image,

XNA will just scale the image to fit in a rectangle that size. Later on we will have some fun modifying the size. We can create a rectangle using `new`:

```
Rectangle spriteRect;  
  
spriteRect = new Rectangle(30, 20, 600, 450);
```

The code above declares a `Rectangle` variable and then sets it to one with the size and dimensions that we need. I've given the `Rectangle` variable the identifier `spriteRect`. This variable will be part of the game world. When the rectangle is created it is passed the X, Y, width and height values so that these can be held within the rectangle structure. Note that this means if we ever want to move the image, or change its size on the screen, we just have to change one of the values that is held in the rectangle. These values are members of the `Rectangle` structure. In C# members which hold values are called *fields*.

You can think of a field as a variable that has been declared inside something. In the case of our `Game1.cs` class, the game world data that we created (for example the color intensity values for our mood light) are fields of that class. We will see later how we actually get hold of individual fields inside the `Rectangle` so that we can change its size and position.

## Sprite Drawing with SpriteBatch

We now have all the information about our sprite and are ready to draw it. Next we need to take control in the `Draw` method and put our image onto the screen. But before we can do the drawing we have to take a little time out and discover just how games consoles actually work.

A modern game console is not one powerful computer, it is in fact several. Some of these run the game itself while other special graphics processors drive the display. The graphics processor unit (GPU) contains optimized hardware to allow it to update the screen as fast as possible. When the `Draw` method runs it actually assembles a bunch of instructions for the GPU and sends them into it. The GPU then follows those instructions to put a picture on the screen. Complex games will contain many images which may be drawn at several different positions on the screen. It is important that

the transfer of the position information and associated images is organized to as efficiently as possible. XNA provides a special class called `SpriteBatch` to batch up a set of sprite drawing instructions. To do the drawing we have to create a `SpriteBatch` of our own. We could do this in the `LoadGraphicsContent` method but XNA provides another place where it is more sensible, and that is the `Initialize` method. This is called when the game starts up. If all these methods are confusing, think about what happens when you organize a party. This takes a number of steps:

5. Set up the tables and chairs.
6. Fetch the food and drink.
7. Repeatedly play music and dance.
8. Tidy up afterwards.

When an XNA game runs it goes through the same process:

1. Set things up: `Initialize`
2. Load game content: `LoadGraphicsContent`
3. Repeatedly update the game and draw the display: `Draw and Update`
4. Free up all the content: `UnloadGraphicsContent`

When the game ends the XNA system will call the `UnloadGraphicsContent` method. We can add statements to that method to explicitly release resources that our game has used, but for now we can leave this out.

In fact we don't have to provide code for all these methods, they are just there so that we can get control at that point in the life cycle of the game. The code that we need to put in the `Initialize` method needs to create a `SpriteBatch` instance which will manage the drawing of all our sprites:

```
SpriteBatch gameSpriteBatch;  
Rectangle spriteRect;  
  
protected override void Initialize()
```

```

{
    gameSpriteBatch = new SpriteBatch(graphics.GraphicsDevice);
    spriteRect = new Rectangle(30, 20, 600, 450);
    base.Initialize();
}

```

The `Initialize` method runs at the start of the game. We have created a variable in our `Game1` class called `gameSpriteBatch` which will do our drawing. When a new `SpriteBatch` is constructed it is given a reference to the graphics device that it must use. Note that we have also created the `Rectangle` which will be used to position the sprite on the screen.

Now that we have our `SpriteBatch` we can use it to draw the sprite. The `SpriteBatch` needs to be told when we start drawing sprites, and when we have finished.

```

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

    gameSpriteBatch.Begin();
    gameSpriteBatch.Draw(gameTexture, spriteRect, Color.White);
    gameSpriteBatch.End();
    base.Draw(gameTime);
}

```

We call methods on the our `SpriteBatch` variable to begin the draw process, draw the sprite and then end the drawing. The `Draw` method is part of the `SpriteBatch` class and is given parameters which identify the image to be drawn, the rectangle to place it in and the color of the light to “shine” on the texture. If we put a program together with the methods as described above we can finally run a program which will display an image on the screen.

#### **Sample Code: Jake Display**

The sample project in the directory “01 JakeDisplay” in the resources for this chapter will draw a picture of Jake.

Figure 4-14 shows the output that we get when run our program to display Jake on the screen.



**Figure 4-14**  
*Displaying Jake on a PC screen*

If you change the content of the file `Jake.jpg` you can make this program display other pictures.

## Filling the screen

It would be nice if the image that we display could exactly fill the screen. We've used values which will let us see the picture, but it does not completely cover the display, and if you run the program on differently configured Xbox systems you will notice that the picture takes up a different amount of the screen. It turns out that this is quite easy to do. Our program can ask the XNA environment the width and height of the screen, and use this to set the size of the display rectangle:

```
spriteRect = new Rectangle(
    0,    // X position of top left hand corner
    0,    // Y position of top left hand corner
    graphics.GraphicsDevice.Viewport.Width, // rectangle width
    graphics.GraphicsDevice.Viewport.Height); // rectangle height
```

I've changed the layout of my call to construct the `spriteRect` variable. Rather than put everything on one line, I've spread the call out a bit and added some comments. This makes it easier to see what is happening. The code is constructing a `Rectangle` instance. When you do this you can feed information into the construction process to set up the value. This particular call is feeding in the position of the top left hand

corner, in the form of X and Y, and the width and height of the rectangle that is required. I can get the value width of the screen by using:

```
graphics.GraphicsDevice.Viewport.Width
```

This looks a bit scary, but is actually quite easy to understand. It is rather like the way that we explain where things are. My office is on the third floor of the Robert Blackburn building on the Hull campus of the University of Hull. So you could express it as:

```
HullCampus.RobertBlackburn.ThirdFloor.RobMiles
```

The Hull campus contains a number of buildings, the Robert Blackburn building contains a number of floors, and so on. You can now find your way to my office by starting at the Hull campus, looking for the Robert Blackburn building, going to the third floor and then finding the office with “Rob Miles” written on the door. The identifier:

```
graphics.GraphicsDevice.Viewport.Width
```

means “Start at the graphics variable, go to the GraphicsDevice, get the Viewport and then get the width field from it”. The graphics variable is created by XNA and contains methods and data that we can use in our program (we have already used the Clear method to clear the screen). It contains a GraphicsDevice, which contains a Viewport and so on. Part of the skill of using XNA is knowing whereabouts these data items are. We will pick up the various bits as we go along.

**Sample Code: Jake Full Screen**

The sample project in the directory “02 Jake Full Screen” in the resources for this chapter will draw a picture of Jake which completely fills the screen.

If you are using an Xbox which is connected to a TV you might notice that not all of the picture is visible. This is because TV’s use an “overscanned” display, where only the middle part of the picture is actually displayed. You will also find that if the shape of your picture does not exactly match that of the screen the image will appear stretched. We will look at these problems of “aspect ratio” later on.

---

## Game Idea: Color Nerve with a Picture

Now that we can display pictures, we can improve our Color Nerve game and display a picture rather than a blank background. This makes the game much more fun, particularly if a familiar picture is used.

---

The key to this is the way that we can select the color we are going to use to “light” any sprite that we draw:

```
gameSpriteBatch.Draw(gameTexture, spriteRect, Color.White);
```

When we draw our image I used a white light, so that the colors look natural. You can use light of any color, and XNA will process the image accordingly. If you want the image to be drawn more dimly, you can draw with the color grey, if you want to tint the image you can just change the color. We can just use the color that has been created to tint our sprite:

```
protected override void Draw(GameTime gameTime)
{
    Color backgroundColor;
    backgroundColor = new Color(redIntensity, greenIntensity, blueIntensity);

    gameSpriteBatch.Begin();
    gameSpriteBatch.Draw(gameTexture, spriteRect, backgroundColor);
    gameSpriteBatch.End();

    base.Draw(gameTime);
}
```

Rather than using white as the drawing color, the above version of Draw uses the color it creates based on the red, green and blue intensity values.

### **Sample Code: Jake Color Nerve**

The sample project in the directory “03 Image Color Nerve” in the resources for this chapter is a version of Color Nerve which uses the picture of Jake.

You can also use the same principle to make a picture mood light, this works particularly well if you use a black and white image or one with really strong colors in it.

### **Sample Code: Image MoodLight**

The sample project in the directory “04 Image MoodLight” in the resources for this chapter is a version of the ultimate mood light which uses an image background. The image contains a pattern of blocks of different colors. One interesting challenge is to try and work out which of the blocks is white (only one of them is).

## **Conclusion**

We have learnt a lot in this chapter. We have seen how we can add graphical resources to XNA projects and use them in our game programs. We have also found out how images are positioned and drawn on the screen in XNA.

## **Pop Quiz**

Just in case you thought you were having too much fun, here is a pop quiz to bring you back down to earth.

1. Images are managed by the C# compiler.
2. In an XNA program an image can be held in a texture.
3. The `LoadGraphicsContent` method is used by XNA to load the graphics images onto the display.
4. A sprite is a small, pixie like creature who lives with the fairies.
5. The `SpriteBatch` class is used to batch up sprites before they are drawn.
6. There is no need for us to add any code to the `Initialize` method to make our XNA game work.
7. The `Rectangle` has a `Width` field which specifies how many pixels across the screen it is.
8. The XNA system can only store one image at a time.