Chapter 5. MonoGame Basics

In 2006, Microsoft released XNA Game Studio Express, a game development framework to make it easier for students and hobbyists to develop games. Microsoft developed and released a number of versions of the framework over the subsequent years; the final version, XNA Game Studio 4.0 Refresh, was released in 2011. In January 2013, Microsoft announced they would no longer be actively developing new versions of the XNA framework.

Starting in 2011, the MonoGame open source project has been developing support for the complete Microsoft XNA 4 framework. Almost all of that framework is implemented in MonoGame 3.4. As the MonoGame web site notes, Microsoft Studios actually released several Windows 8 and Windows 8 Phone games in 2013 using MonoGame!

The XNA 4 framework supported building games for Windows 8, Windows Phone 8, and the Xbox 360. In contrast, MonoGame 3.4 supports development for iOS, Android, MacOS, Linux, Windows, Windows Store, Windows Phone, Xbox 360, PlayStation 4, and Ouya (with support for PlayStation Mobile, Xbox One, and RaspberryPi in the works). Go here to see some of the games MonoGame developers have published. To make a long story short (too late), MonoGame 3.4 is a great industrial-strength cross-platform game development framework.

In this book, we'll be using MonoGame to develop Windows Games. If you do want to check out developing for other platforms, you should go here.

5.1. The Worst Game EVER

Just to drive you into a frenzy of game development madness, where all you want to do is drink Mountain Dew, eat Twinkies, and sling code until your fingers bleed[1], we'll start by building the worst game EVER.

Okay, start up the IDE, click New Project …, and click on the MonoGame subfolder in the Installed Templates area of the pane on the left. Click the MonoGame Windows Project icon in the Templates pane, change the Name to whatever name you want to call the project, set the location where you'd like the project to be saved, and click OK. Run the program by pressing F5; when you do, you'll get the amazing output shown in Figure 5.1. Just click the X in the upper right corner of the window to return to the IDE.

Mac Users: Start up Xamarin Studio, click New Solution …, and click on the Miscellaneous category in the Other section in the pane on the left. Click the MonoGame Mac Application (MonoMac) icon in the Templates pane and click Next. Change the Project Name to whatever name you want to call the project, set the location where you'd like the project to be saved, and click Create.

[1] While the author has never coded until his fingers bled, there is a vicious rumor that he once played the arcade version of Track and Field until one of his fingers bled. Sadly, the vicious rumor is in fact true.

Linux Users: Start up MonoDevelop, Click New... just below the Solutions heading, and click on the Miscellaneous category in the Other section in the pane on the left. Click the MonoGame Application (OpenGL) icon in the Templates pane and click Next. Change the Project Name to whatever name you want to call the project, set the location where you'd like the project to be saved, and click Create.

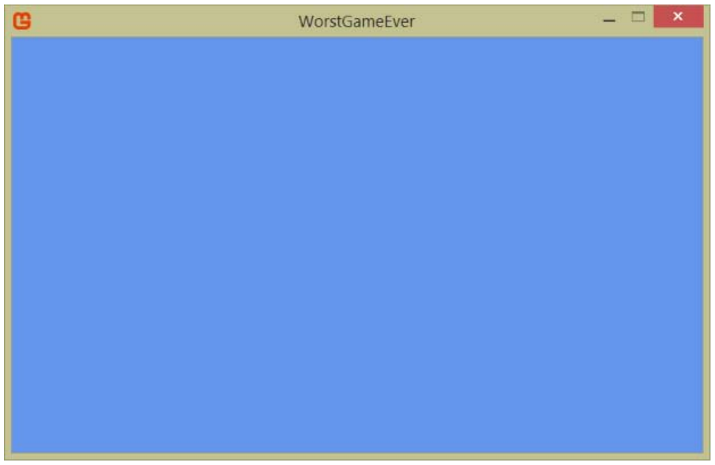


Figure 5.1. Worst Game Ever Output

If this "game" looks familiar, it's because we did the exact same thing in Appendix A when we were making sure that MonoGame installed properly. So why is this the worst game ever? Because there's no player interaction with the game world, and that's what games are all about!

Mac and Linux users will probably see the game open in full screen rather than in windowed mode. If you delete the line of code at or near line 25 that sets `graphics.IsFullScreen` to `true` your game will run in windowed mode.

The bad news is that to have player interaction we need to use MonoGame's input capabilities, which are covered in Chapter 8, not here. The good news is that we can start learning about how MonoGame works in this chapter by providing some interesting graphical output in our game.

5.2. A Walk Through the Game Class

Let's take a look at the `Game1` class that the IDE automatically generated for us when we selected the MonoGame Windows Project template[2]. Once we understand how that class works, we'll be able to move on to our next (slightly better) game.

The first thing you'll notice is that this class doesn't look anything like the application classes we've been working with so far; where's the `Main` method? That's because the `Game1` class isn't the application class, the `Program` class is. If you double-click the Program.cs entry in the Solution Explorer pane of the IDE, you'll see that the `Program` class has the `Main` method we

[2] As a default the IDE gives us a `Game1` class that extends the `Game` class. We'll learn more about inheritance later in the book.

were looking for. Of course, all the `Main` method does is create a `Game1` object and start running it, so let's go back to the `Game1` class to see how it works.

You should also notice that there are a number of `using` statements at the top of the class using code in XNA namespaces. Those namespaces include the top-level namespace, `Microsoft.Xna.Framework`, and several function-specific namespaces (`Microsoft.Xna.Framework.Input`, which lets us process player input in our game, for example). These `using` statements let us access XNA code to help us do our game development more easily.

You might be wondering at this point why we're talking about XNA rather than MonoGame. Because the MonoGame project was started to port the XNA Framework over to an open-source, cross-platform solution, the MonoGame developers kept all the namespace names the same. This is a big help (as we can personally attest) as developers port games they already developed in XNA over to MonoGame, and it also means that developers can just use all the XNA namespace documentation from Microsoft if they want to. Appendix A describes how you can integrate the XNA documentation into Visual Studio Community 2015.

Our approach throughout the book will be to use MonoGame when we're talking about the overall game framework and XNA when we're talking about classes in the top-level or function-specific namespaces.

Before looking at the specifics of the methods in the `Game1` class, you should understand how a *game loop* works. The general idea behind a game loop is that games essentially do two things: update the game world and draw (or render) the game world. That's exactly how MonoGame works; it updates, then draws, then updates, then draws, then … well, you get the idea. There are usually some start-up actions we need to take before we enter that update/draw loop, though, so we'll look at those first.

Our `Game1` class has two fields: `graphics` and `spriteBatch`. The `graphics` field is a `GraphicsDeviceManager` object, and it's used to configure and manage the graphics device being used by the game. In this book, we won't use the `graphics` field directly much at all, though it certainly does a lot of "under the hood" work for us. The `spriteBatch` field is a `SpriteBatch` object; we use this field to draw sprites to the screen. We'll talk much more about how we use the `spriteBatch` field later in this chapter.

The constructor for our `Game1` class just does some initial setup; we won't need to worry about those details at all.

The first method (after the constructor) we come to in our `Game1` class is the `Initialize` method. The `Initialize` method is called right after the constructor is called so we can initialize the game world. Initializing the game world could mean building a deck of cards and creating some players with initial money amounts for a Blackjack game, setting up a number of Non-Player Characters (NPCs) in an RPG, building the first puzzle for the player to solve in a puzzle game, and so on. Basically, you should do the things that are necessary before the player actually starts playing the game.

The next method we see is the `LoadContent` method. Like the `Initialize` method, the `LoadContent` method is called before the game loop is started. You probably won't be surprised to learn that the point of this method is to load the content that your game needs. So what's content? The graphical and sound assets your game needs to work properly. For the games we develop in this book, we'll load some or all the game content from within the `LoadContent` method, but you should realize that for larger games this wouldn't be the best approach. Instead, it would be better to load the content for each level as we're transitioning to that level; that way, we only hold the content that's currently required in memory rather than all of the game content. Of course, loading content takes time, so games will typically show some sort of dynamic loading icon or series of screens while the content is loading for the next level.

The `Game1` class also provides an `UnloadContent` method that does exactly what you'd suspect. In larger games, we'd use this method to unload the content for a level we just finished before loading the content for the next level, but for the games in this book we won't need to use this method.

Now we get to the `Update` method. The `Update` method gets called every time through the game loop, so we use this method to update the state of our game world. Chances are, this includes updating the states of all the objects in our game world (and perhaps doing some extra processing as well). What kinds of things might we do here? Changing the locations of moving game entities, detecting any collisions and processing them appropriately, having NPCs do their planning for what they want to do next, and so on.

So how often does the `Update` method actually get called? The default in MonoGame is that the game runs on a fixed interval, where `Update` (and `Draw`) get called every 1/60th of a second. If you just use the default settings, your game will run at 60 frames per second (fps)[3]. This will work fine for our games, but if you want the game to go as fast as possible – in other words, faster than 60 fps when it can – you can change the `IsFixedTimeStep` property in the `Game1` class to `false`.

Of course, updating the state of the game world without showing those changes to the players isn't a very good idea. That where the `Draw` method comes in. The `Draw` method also gets called every time through the game loop, and its job is to draw the visible game entities.

5.3. A Better Not A Game (NAG)

Now that we know a little more about how MonoGame works, let's build a slightly better Not A Game (NAG) than our worst game ever. For this NAG (though we'll lie and call it a game throughout the section), we'll focus on loading content and drawing a few stationary teddy bears to the screen.

[3] There are certain cases where the game might slow down and you actually get a worse frame rate. We won't worry about that here, but MonoGame does provide a way to tell if your game is running slowly and take appropriate action.

Create a new MonoGame Windows Project in the IDE, naming it BetterNag. The first thing we're going to do is add a few more fields near the top of the class, just below the line that says `SpriteBatch spriteBatch`; here's what you should add:

```
// sprites for teddy bears
Texture2D teddyBearSprite0;
Texture2D teddyBearSprite1;
Texture2D teddyBearSprite2;
```

Remember a field is a variable, which is a place in memory we use to store information; we already talked about what actually gets stored in memory in Chapters 3 and 4. For now, just think of these variables as storing `Texture2D` objects. You should recognize from our discussion in the previous chapter that `Texture2D` is a class, but what is it for?

The `Texture2D` class, which is found in the `Microsoft.XNA.Framework.Graphics` namespace, holds a two-dimensional grid of texels; think of these as pixels in a 2D image. To simplify quite a bit, a `Texture2D` object holds a two-dimensional image that we can manipulate and display. We'll be using the variables you declared above to hold the graphical assets we use in this game.

So why did we start numbering the sprites with 0 instead of 1? You'll find that computer programmers regularly start counting at 0. This is partly for historical reasons, but it's also because many of the structures we look at later in the book (like arrays and `List`s) hold their first element in the 0th location. We know it seems weird, but that's how we'll number things in our programs. Don't worry, this will seem perfectly normal to other programmers.

Having `Texture2D` objects for the sprites we want to draw in our game is great, but we also need to know where we want to draw them in the game window. You should add the following fields right below your new sprite fields:

```
// draw rectangles for teddy bears
Rectangle drawRectangle0;
Rectangle drawRectangle1;
Rectangle drawRectangle2;
```

XNA provides the `Rectangle` structure to let us easily represent and process rectangles. Structures are a lot like classes, with differences that aren’t important in this book; we essentially use them the same way as classes, so those differences aren't pertinent at this point. In addition to useful properties and methods we'll use in later games, the `Rectangle` structure also has four public fields we can access: `X`, `Y`, `Width`, and `Height`. `X` and `Y` provide the screen x and y coordinates of the upper left-hand corner of the rectangle, and `Width` and `Height` are (duh) the width and height of the rectangle.

We'll discuss our `drawRectangle` fields more when we create the `Rectangle` objects for them below, but for now you can think of stretching the sprite to fit onto a rectangle with a particular `Width` and `Height`, then moving that rectangle so its upper left corner is at `X`, `Y` in the game window.

The next place we need to add code is in the `Game1` constructor. MonoGame actually defaults to an 800 by 480 screen resolution because that's the resolution on Windows Phone 7s. That's a 5:3 aspect ratio, which doesn't really make sense in Windows. The most common Windows aspect ratios are 4:3, 16:9, and 16:10. We're going to use a 604 by 453 resolution (which is a 4:3 aspect ratio) throughout this book.

What the heck? Why aren't we using at least 800 by 600, a much more common 4:3 resolution? Well, it turns out that we need our images to be no wider than 622 pixels to avoid scaling on some e-readers. Rather than reducing all our game screen shots and potentially making them slightly blurry, we'll use a smaller resolution than we'd typically use in a game. You can, of course, use whatever resolution makes sense to you in your own games; in fact, you'll probably even let the player select their preferred resolution if you build a commercial Windows game.

To change the resolution, we need to add the following two lines to the end of the `Game1` constructor:

```
graphics.PreferredBackBufferWidth = 604;
graphics.PreferredBackBufferHeight = 453;
```

Next, we need to load the graphical assets for each of the three teddy bears, and we'll do that in the `LoadContent` method. Before we do, we'll need to actually add the content to our solution in the IDE; for now, think of the content as all the art assets you need for your game.

The first thing you should do is right click on the Content folder under the BetterNag project in the pane on the right and select Add > New Folder; name the new folder graphics.

Next, we need to use the MonoGame content pipeline to build the content we need to include in the game. It doesn't really matter where you save your content pipeline project, but if you download the code from the Burning Teddy web site you'll see we always create a content pipeline folder in the same folder as the .sln file for our Visual Studio project and put our content pipeline project and all our game assets in that folder. We do it that way so we can easily move the entire game to a different location on our computer instead of having to move multiple pieces or changing folder paths within our Visual Studio solution.

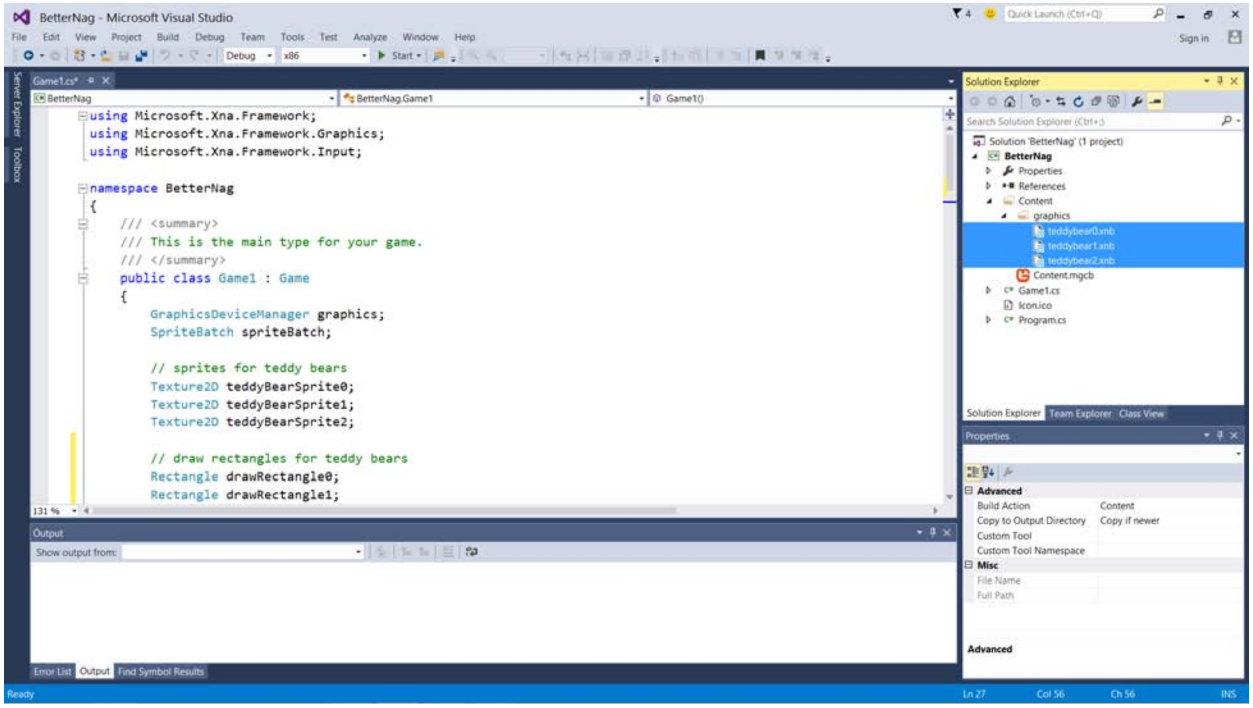
Appendix B discusses the content pipeline, and Section B.1. describes how to build the content for our current example. You should go follow the instructions provided there, then come back here to actually add the content to our Visual Studio project.

Once you've built the content using the Pipeline tool, it's time to add it to our Visual Studio project. Right click on the graphics folder in the pane on the right and select Add > Existing Item ... (Mac and Linux users: select Add > Add Files ...) Navigate to the folder where the xnb files the Pipeline tool generated are located (Section B.1. tells you where they get put) and select all three xnb files. The file dialog won't initially show any files, but if you click the down arrow next to "Visual C# Files (*.cs, *.resx," near the lower right corner of the file dialog and change that to "All Files (*.*)" you'll see the xnb files. Click the down arrow next to "Add" near the lower right corner of the file dialog and select Add As Link. At this point, the xnb files should have been added to the graphics folder in your Visual Studio project.

Why did we select Add As Link instead of Add? Because if the artists go back later to change the art assets for the xnb files and rebuild the content, the new content will automatically be added to the game. This is great, because otherwise we'd need to go back and manually re-import the art assets into our Visual Studio project every time they changed. Adding the content as links is the way we've done it on all our commercial game projects because it makes content changes much less traumatic!

We're almost done getting our content integrated into our project (Mac and Linux users, see paragraph below). Select all three xnb files in the graphics folder. Click the None box next to Build Action in the pane on the lower right, click the arrow that appears on the right, and select Content. Click the Do not copy box next to Copy to Output Directory in the pane on the lower right, click the arrow that appears on the right, and select Copy if newer (Figure 5.2. shows what your project should look like at this point). Making these changes ensures that your content files get copied to the appropriate location when you build your game so they're available when you run your game. Skipping this step guarantees your game will crash when you try to run it, so don't forget to do this every time you add content!

Mac and Linux Users: Select all three xnb files in the graphics folder. Right click and select Build Action > Content. Right click again and select Quick Properties > Copy to Output Directory. Making these changes ensures that your content files get copied to the appropriate location when you build your game so they're available when you run your game. Skipping this step guarantees your game will crash when you try to run it, so don't forget to do this every time you add content!


BetterNag - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug x86 Start
Quick Launch (Ctrl+Q)
Sign in
Game1.cs*
BetterNag
BetterNag.Game1
Game1()
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'BetterNag' (1 project)
BetterNag
Properties
References
Content
graphics
teddybear0.xnb
teddybear1.xnb
teddybear2.xnb
Content.mgcb
Game1.cs
Icon.ico
Program.cs
Solution Explorer Team Explorer Class View
Properties
Output
Show output from:
Error List Output Find Symbol Results
131 %
Ready Ln 27 Col 56 Ch 56 INS

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

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

        // sprites for teddy bears
        Texture2D teddyBearSprite0;
        Texture2D teddyBearSprite1;
        Texture2D teddyBearSprite2;

        // draw rectangles for teddy bears
        Rectangle drawRectangle0;
        Rectangle drawRectangle1;
```

Advanced	
Build Action	Content
Copy to Output Directory	Copy if newer
Custom Tool	
Custom Tool Namespace	
Misc	
File Name	
Full Path	

Advanced

Figure 5.2. Project With Content Added

Now that we have our assets added to the project, we can add the code we need in the `LoadContent` method to actually load the content when we run the game. Replace the comment that says `// TODO: use this.Content to load your game content here` with the following code:

```
// load teddy bear sprites
teddyBearSprite0 = Content.Load<Texture2D>(@"graphics\teddybear0");
teddyBearSprite1 = Content.Load<Texture2D>(@"graphics\teddybear1");
teddyBearSprite2 = Content.Load<Texture2D>(@"graphics\teddybear2");
```

Let's look at what these lines of code are actually doing. In the first statement, the code on the right-hand side of the equals sign is a method call that creates a new `Texture2D` object from the teddybear0 graphical asset we added to the project; the name of our graphical asset is simply the name of our xnb file without the .xnb extension, though we need to include the graphics folder as part of the path to our asset. After we execute the first line of code, the `teddyBearSprite0` variable refers to ("holds", if you prefer) that new object. So after we execute all of the lines of code above, our three teddy bear assets have been loaded into our three variables.

We learned in the previous chapter that the backslash (\) is an escape character, so we need to make sure the `\t` in our strings above isn't interpreted as a tab in our asset names. When we add the `@` character before a string literal, we're saying the \ shouldn't be treated as an escape character in the string literal. We prefer to use this approach when we're dealing with folder paths, but you can also just include a \\ in your string literals to include a single \ in your folder path if your prefer.

We also need to build the draw rectangles for each of our teddy bears; that's what this code does:

```
// build draw rectangles
drawRectangle0 = new Rectangle(graphics.PreferredBackBufferWidth / 4,
    graphics.PreferredBackBufferHeight / 4,
    teddyBearSprite0.Width, teddyBearSprite0.Height);
drawRectangle1 = new Rectangle(graphics.PreferredBackBufferWidth / 2,
    graphics.PreferredBackBufferHeight / 2,
    teddyBearSprite1.Width, teddyBearSprite1.Height);
drawRectangle2 = new Rectangle(graphics.PreferredBackBufferWidth * 3 / 4,
    graphics.PreferredBackBufferHeight * 3 / 4,
    teddyBearSprite2.Width, teddyBearSprite2.Height);
```

We create a new `Rectangle` object using the `new` keyword just as we do for classes, and we're using the (overloaded) `Rectangle` constructor that lets us specify all four fields. Because `Texture2D` objects have properties called `Width` and `Height` to access the width and height of the image, we're using those values for the `Width` and `Height` of the rectangle we're drawing the image in on the screen. That means we're drawing the image in its actual size; we can stretch or shrink the actual image by changing the width and height of the draw rectangle. Go ahead, try it once we start drawing the sprites – you know you want to!

You might be wondering why we did all that math with `graphics.PreferredBackBufferWidth` and `graphics.PreferredBackBufferHeight` instead of just hard-coding numbers (like 146) for our teddy bear locations. We do it this way because if we decide to change our resolution later the code we write here won't have to be changed. Even more importantly, if a player were changing their resolution as part of their game settings, this code would work fine no matter what resolution they picked.

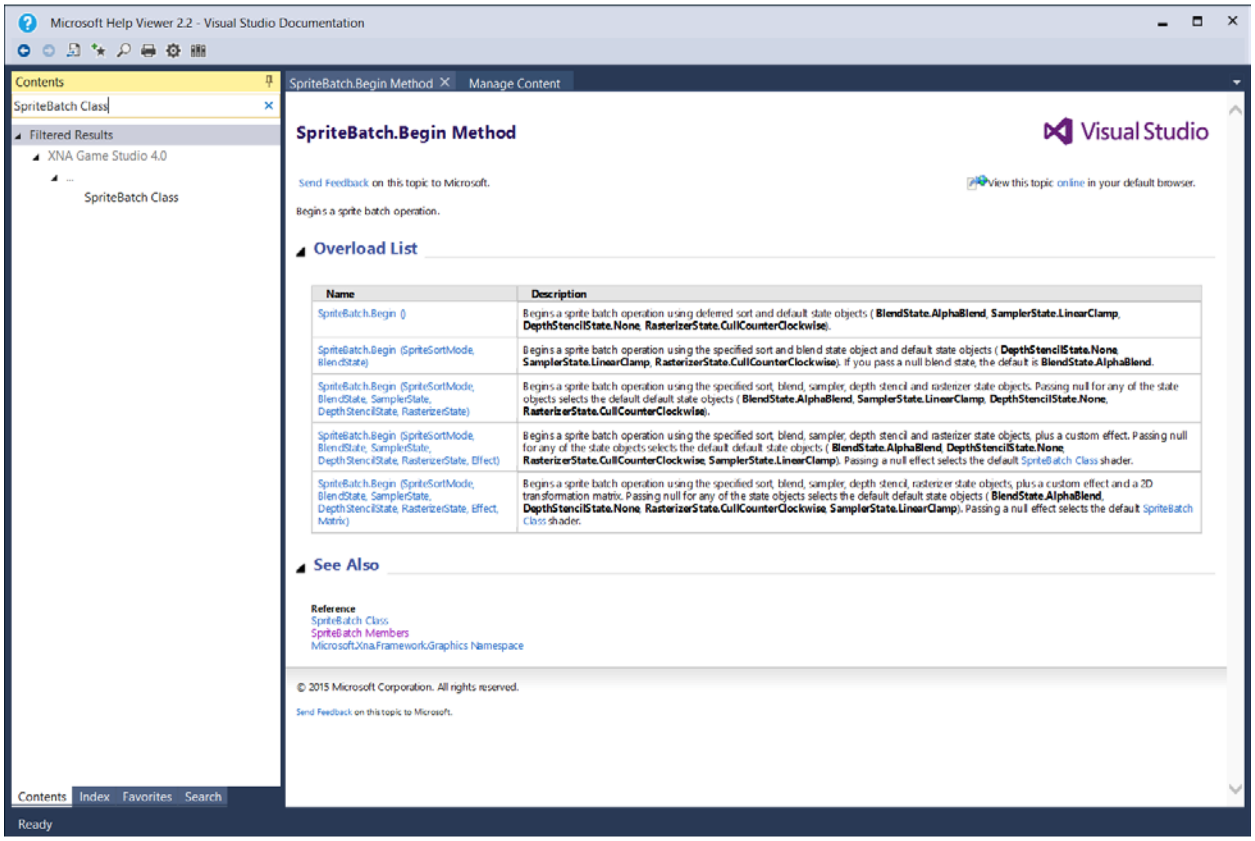
In a typical game – like the one we'll write in the next section – we'd now add code to the `Update` method to update the state of the game world each time through the game loop. For this game, though, we're not changing the game world, so we'll skip that step.

The final thing we need to add to our game is the code we need to actually draw each of the teddy bears. As you know from the discussion in the previous section, this code goes in our `Draw` method. Replace the line that says `// TODO: Add your drawing code here` with the following code:

```
// draw the teddy bears
spriteBatch.Begin();
spriteBatch.Draw(teddyBearSprite0, drawRectangle0, Color.White);
spriteBatch.Draw(teddyBearSprite1, drawRectangle1, Color.White);
spriteBatch.Draw(teddyBearSprite2, drawRectangle2, Color.White);
spriteBatch.End();
```

As you type in the code, you'll notice that the IDE pops up little windows to help you figure out how to call the `Begin` and `Draw` methods of the `spriteBatch` object correctly; Microsoft calls this IntelliSense, and believe us, you'll quickly get used to having it!

There are actually five `Begin` methods contained in the `SpriteBatch` class (`spriteBatch` is an object of that class). One way to see the difference between them is to use help to navigate to the documentation of that method; see Figure 5.3.



Microsoft Help Viewer 2.2 - Visual Studio Documentation

Contents

SpriteBatch Class

Filtered Results

XNA Game Studio 4.0

...

SpriteBatch Class

SpriteBatch.Begin Method

Manage Content

SpriteBatch.Begin Method

Visual Studio

Send Feedback on this topic to Microsoft.

View this topic online in your default browser.

Begins a sprite batch operation.

Overload List

Name	Description
SpriteBatch.Begin ()	Begins a sprite batch operation using deferred sort and default state objects (**BlendState.AlphaBlend**, **SamplerState.LinearClamp**, **DepthStencilState.None**, **RasterizerState.CullCounterClockwise**).
SpriteBatch.Begin (SpriteSortMode, BlendState)	Begins a sprite batch operation using the specified sort and blend state object and default state objects (**DepthStencilState.None**, **SamplerState.LinearClamp**, **RasterizerState.CullCounterClockwise**). If you pass a null blend state, the default is **BlendState.AlphaBlend**.
SpriteBatch.Begin (SpriteSortMode, BlendState, SamplerState, DepthStencilState, RasterizerState)	Begins a sprite batch operation using the specified sort, blend, sampler, depth stencil and rasterizer state objects. Passing null for any of the state objects selects the default default state objects (**BlendState.AlphaBlend**, **SamplerState.LinearClamp**, **DepthStencilState.None**, **RasterizerState.CullCounterClockwise**).
SpriteBatch.Begin (SpriteSortMode, BlendState, SamplerState, DepthStencilState, RasterizerState, Effect)	Begins a sprite batch operation using the specified sort, blend, sampler, depth stencil and rasterizer state objects, plus a custom effect. Passing null for any of the state objects selects the default default state objects (**BlendState.AlphaBlend**, **DepthStencilState.None**, **RasterizerState.CullCounterClockwise**, **SamplerState.LinearClamp**). Passing a null effect selects the default SpriteBatch Class shader.
SpriteBatch.Begin (SpriteSortMode, BlendState, SamplerState, DepthStencilState, RasterizerState, Effect, Matrix)	Begins a sprite batch operation using the specified sort, blend, sampler, depth stencil, rasterizer state objects, plus a custom effect and a 2D transformation matrix. Passing null for any of the state objects selects the default default state objects (**BlendState.AlphaBlend**, **DepthStencilState.None**, **RasterizerState.CullCounterClockwise**, **SamplerState.LinearClamp**). Passing a null effect selects the default SpriteBatch Class shader.

See Also

Reference
SpriteBatch Class
SpriteBatch Members
Microsoft.Xna.Framework.Graphics Namespace

© 2015 Microsoft Corporation. All rights reserved.

Send Feedback on this topic to Microsoft.

Figure 5.3. Documentation for SpriteBatch Begin Method

First, notice that the heading for the methods is called Overload List. Whenever a class has methods with the same name but different parameters, that method is said to be *overloaded*. That's why the list of `Begin` methods is called the Overload List in the documentation. We've seen this before with the documentation for the `Console WriteLine` method, but it's worth some further discussion here.

For an overloaded method like `Begin`, how do we indicate which version of the method we want to call? By the arguments we provide between the parentheses in the method call. In the code you typed in, we said we wanted to use the first version on the list, which doesn't require any arguments.

You should know that one of the characteristics of this version of the `Begin` method is that it uses `BlendState.AlphaBlend`. To understand why we want to use that blend mode, open up one of the .png files using a program like Paint .NET or Paint Shop Pro. When you do, you'll see that the sprite actually has areas that are fully transparent. We store four different values for each pixel in a .png image: R, G, B, and A. R, G, and B are the Red, Green, and Blue components of the pixel, and A is the alpha value of the pixel. The alpha value specifies the opacity of the pixel, with a value of 255 indicating fully opaque and a value of 0 indicating fully transparent. So when we use a `Begin` method that uses the `BlendState.AlphaBlend` blend state, we're saying that transparency information should be used when we draw graphical assets.

The next things you added were the calls to the `Draw` method to add each of the teddy bear sprites to the sprite batch. The `Draw` method has 7 different overloads, so you have lots of flexibility deciding how you want to draw! We chose to use the overload that takes three parameters: a `Texture2D`, a `Rectangle`, and a `Color`. In the first call to the `Draw` method, we provide the first teddy bear sprite as the `Texture2D` to be drawn and the first draw rectangle to tell where to draw that sprite (and what size to draw it). We provide `Color.White` as the color to use because we want the image to be the same color as the .png file image; we can "tint" the image we're drawing by using a color other than white (and there are LOTS of options!).

We actually admit that we initially tried to simplify this example by creating our draw rectangles each time we draw our sprites inside the `Draw` method. That's a bad idea, though, because creating three new `Rectangle` objects on every call to the `Draw` method is very wasteful. The problem is that we're creating these new objects 60 times a second (running at a fixed time step), so we're using much more memory than we really need to; we should always be reasonably efficient with our memory usage. We won't bother changing the code for this example so we can stay focused on how the drawing actually works, but for later programs we'll make sure we don't do anything silly like this. Take a moment to think about how you'd solve this problem for this program; you'll see a reasonable way to do this later in the chapter.

The last line you added, the call to the `End` method, flushes the sprite batch. You might have thought it a little strange that we said calls to the `Draw` method add the sprites to the sprite batch rather than saying that the calls actually draw the sprites. That's because all the sprites we add to the sprite batch actually don't get drawn until we call the `End` method! There are ways to override this behavior in MonoGame if you really want to, making each sprite get drawn immediately rather than waiting until the call to the `End` method, but since you have to call the `End` method anyway when you're done drawing, we'll just use the default behavior.

Although we don't need it for this game, you can actually have multiple blocks that have the `spriteBatch` `Begin`, `Draw`, and `End` method calls. One reason you might do this is if you want to use different blend modes for different things you're drawing.

Now that we've added all the code for this project, go ahead and run it. You should get output that looks like 5.4. Okay, so it's better than the worst game ever, but it's still awfully boring. Let's make the bears at least move around!

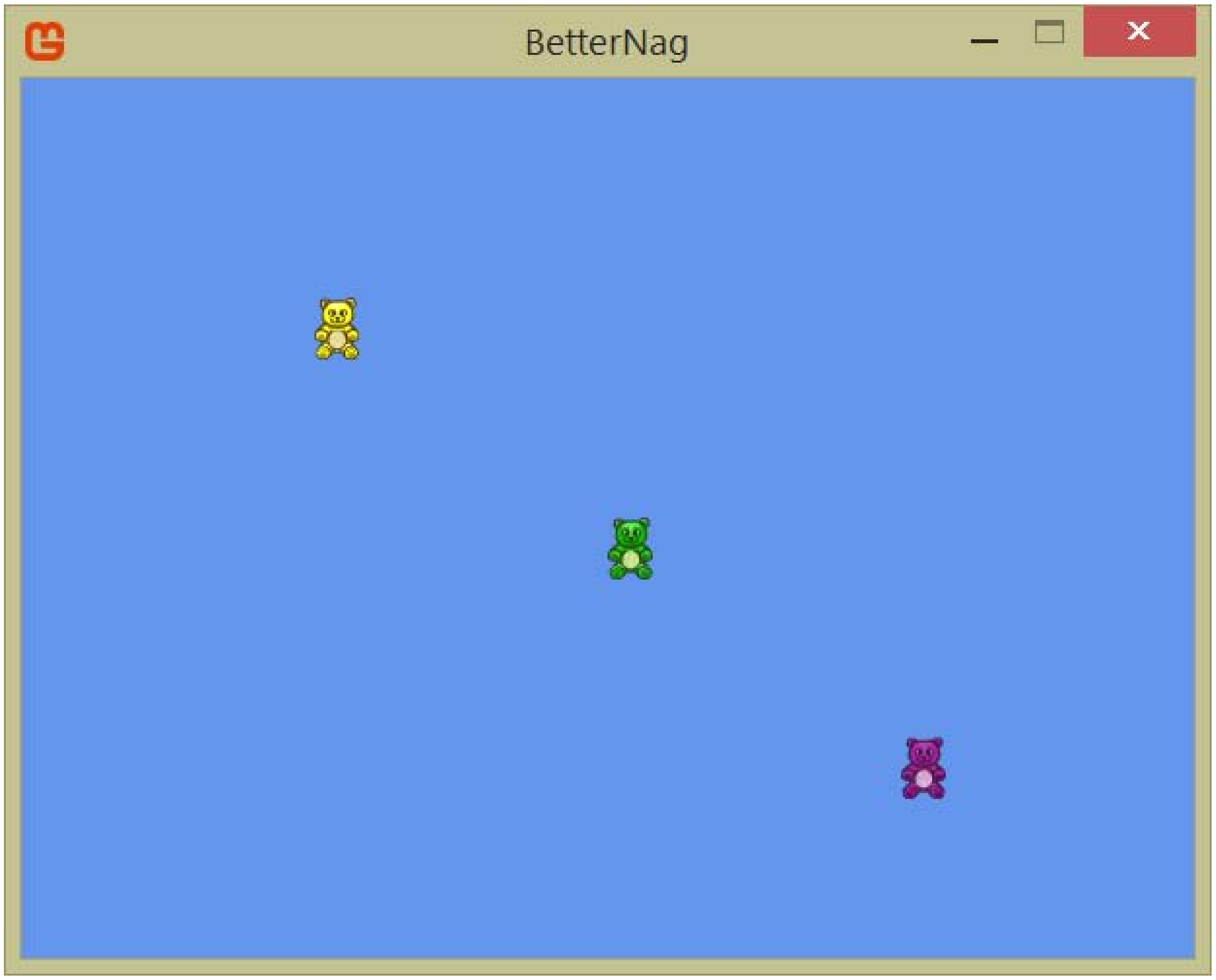


Figure 5.4. Better NAG Output

5.4. Putting It All Together

Let's make our NAG more interesting; here’s the problem description:

Move three teddy bears around the screen until the player quits the game.

Understand the Problem

This seems pretty straightforward, but you should have at least one question. What should we do when a teddy bear reaches the edge of the screen? There are lots of possibilities here – we could have the teddy bear bounce off the edge of the screen, warp to the other side of the screen, explode and then re-spawn on the screen, and so on.

Since the problem description doesn't tell us which method is preferred, we'd like to bounce the bears off the edges of the screen. Because we're going to use our problem description for our testing, though, we need to revise the problem description to capture this detail. Here's the revised problem description:

Move three teddy bears around the screen, bouncing them off the edges, until the player quits the game.

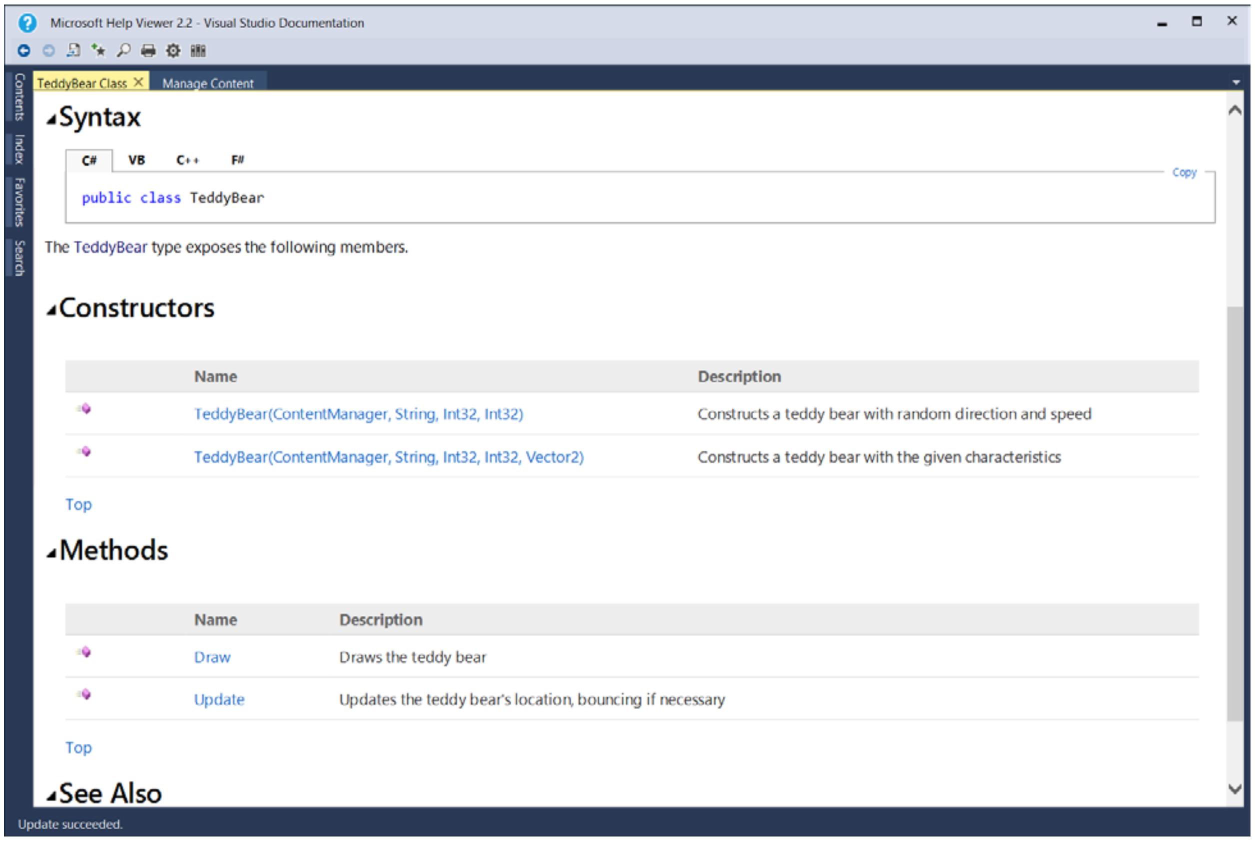
By the way, you as the programmer will almost NEVER get to change the problem description at your own discretion! Instead, the typical process would be to discuss your recommended changes with whoever "owns" that problem description (typically, the person who gave you the problem to solve) and negotiate the appropriate changes to the description. Since the author owns the problem description above, though, we negotiated with ourselves[4] and agreed to the revised wording.

Design a Solution

We could solve this problem by adding lots of teddy bear-specific code to our `Game1` class, but that's not the kind of object-oriented approach we want to use for the games we develop. Instead, we'll use a `TeddyBear` class to build teddy bear objects that behave in the appropriate way. Our `Game1` class will then create those objects and call their methods as appropriate. Of course, we had to develop the `TeddyBear` class to use in our solution; it’s not something that comes with MonoGame.

Although in the previous chapter we waited until we wrote the code to look at how to use the `Card` class, our Design a Solution step really should be looking at the classes we're going to use to build our solution. For that reason, we'll look at the details of the `TeddyBear` class here and then simply use it when we get to the Write the Code step. Let's look at the documentation of the `TeddyBear` members first; see Figure 5.5.

[4] Talking to yourself is a perfectly normal part of programming, so don't be alarmed if you find yourself doing so. It's also normal to talk to your computer in either a pleading or threatening tone depending on your current mood.



Syntax

```
public class TeddyBear
```

The TeddyBear type exposes the following members.

Constructors

	Name	Description
	TeddyBear(ContentManager, String, Int32, Int32)	Constructs a teddy bear with random direction and speed
	TeddyBear(ContentManager, String, Int32, Int32, Vector2)	Constructs a teddy bear with the given characteristics

Methods

	Name	Description
	Draw	Draws the teddy bear
	Update	Updates the teddy bear's location, bouncing if necessary

See Also

Figure 5.5. TeddyBear Members Documentation

We can see from the documentation that the `TeddyBear` class *exposes* a number of useful members. Here's a brief side note to help us easily discuss interactions between classes. Classes are designed to be used by entities external to the class; for ease of reference, we'll call those external entities *consumers* of the class. The members that a class provides to consumers of the class are said to be *exposed* because the consumers can "see" them.

The documentation indicates that we have multiple constructors to choose from as well as `Draw` and `Update` methods. Without needing to understand the implementation details within the `TeddyBear` class, we can read the documentation and understand how we should use the class in our problem solution.

Of course, before we can call any of the `TeddyBear` methods, we need to know how to create `TeddyBear` objects. As you know, we use constructors to do that, but we still need more details about the (overloaded) `TeddyBear` constructors.

We can see from the documentation that we have two ways to create a new `TeddyBear` object. We can either create a teddy bear that moves with a random velocity (random direction and speed), or we can explicitly set the velocity when we create the `TeddyBear` object. We'll use the first constructor, but we still don't know what we should provide for the String, Int32, and Int32 parameters. That's really a detail for when we actually write the code, though, so we'll look at that when we Write the Code.

You might be wondering why we need to pass a `ContentManager` object into the constructors, and you might also be wondering why the `TeddyBear` class doesn't expose a `LoadContent` method. To be honest, when we first wrote the `TeddyBear` class, we actually had a separate `LoadContent` method that we called in the `Game1` `LoadContent` method for each `TeddyBear` object in our game.

There are several problems with this approach. The most critical problem is that you should follow a general philosophy when creating objects that as soon as the object is created, it's ready to be used. For our `TeddyBear` objects, that means we should be ready to `Update` and `Draw` them as soon as they're instantiated using the constructor. We can't do that if we haven't loaded their content yet (there's nothing to draw until we have the sprite loaded), so we should load the content in the constructor.

The other problem is that the `Game1` `LoadContent` method is called once before the game enters the game loop. That's fine for objects that are available when the game is starting up, but doesn't work for objects that we dynamically spawn during game play. For those objects, we'd need to call both the constructor and the `LoadContent` methods before they were ready for use, which is of course more awkward for the consumer of the class. For these reasons, throughout the rest of this book (and in the commercial games we build), we make sure the constructor loads the content for the object.

Now we know how we can use the `TeddyBear` class to help us solve the problem using our `Game1` class. We'll create three teddy bears in the `Game1` `LoadContent` method and then update and draw them within the game loop. We'll still need to know a few more details about the `TeddyBear` class before we can actually write the code, but we'll get those details when we need them.

Write Test Cases

This program doesn't have any user input, so we'll just run it and make sure the teddy bears move around the screen and bounce when they're supposed to.

<u>Test Case 1: Checking Bear Behavior</u>
Step 1. Input: None.
Expected Result: Bears move around screen and bounce off edges

You might be wondering about the randomness when we run the program. Because we're going to be using the constructor that makes a teddy bear move in a random direction at a random speed, don't we have to make sure that's actually random? The answer is no for two reasons.

First, and most importantly, the problem description didn't say how the bears were supposed to move. We decided as part of our problem-solving process that we'd have them move randomly, but we could just as easily have chosen some other way to make them move. Functional tests compare the program's behavior against the program description (often called the *requirements*), so our functional tests don't have to test behavior that's not specified in the problem description.

Second, and also important, the randomness of the direction and speed is part of the behavior included in the `TeddyBear` class. Because we're simply a consumer of that class for our problem solution, we can reasonably assume that the `TeddyBear` class was thoroughly unit tested before it was provided to us. That means we don't have to test it again here.

Write the Code

It's time to get to the actual code. Create a new MonoGame Windows Project (call it MovingTeddyBears) and add the teddy bear content to the project like we did before. You'll also need to include the `TeddyBear` class in your project. There are a variety of ways to do this, including copying the TeddyBear.cs file from somewhere (if it's available) and adding it to the project just like we added content. You should just do it the easiest way of all, though – open up the project file provided on the web site and you'll see that everything you need is already included in the project.

The first thing we'll do is add fields to the `Game1` class to hold our three teddy bears. Add the following code immediately following the line that says `SpriteBatch spriteBatch;`

```
// instance variables for teddy bears
TeddyBear bear0;
TeddyBear bear1;
TeddyBear bear2;
```

Remember in the previous chapter that we said we can create our objects (by calling a constructor) when we declare the variables for them or we can create the objects later. In the games we develop, we'll create our game objects in a variety of places, including the `Initialize`, `LoadContent`, or `Update` methods of our `Game1` class. Because constructors give the objects their initial state (e.g., initializing those objects), doing it this way makes sense.

You should run the code now to make sure it compiles and that nothing breaks when you run it. You should still get the cornflower blue screen when you do so. Why should you do this now when nothing obvious changed in the program's output? Because you should definitely get in the habit of running your code every time you add a few lines of code. Don't wait until the end – check out your code regularly so that if (er, maybe even when) it does break you have a very good idea of what broke it.

Next, we’ll change the constructor so we have a 604 by 453 resolution for the game. In the Better NAG game, we simply used the literals 604 and 453 in our code to do this. Unfortunately, using that approach in this game will cause problems for us. Why?

The problem is that the `TeddyBear` class needs to know the window dimensions to be able to bounce teddy bears off the edges of the window. That class doesn’t have access to the `graphics` field in the `Game1` class, so it can’t use its `PreferredBackBufferWidth` and `PreferredBackBufferHeight` properties. We’ll solve this by declaring two constants in the `Game1` class just below where you declared your teddy bear variables:

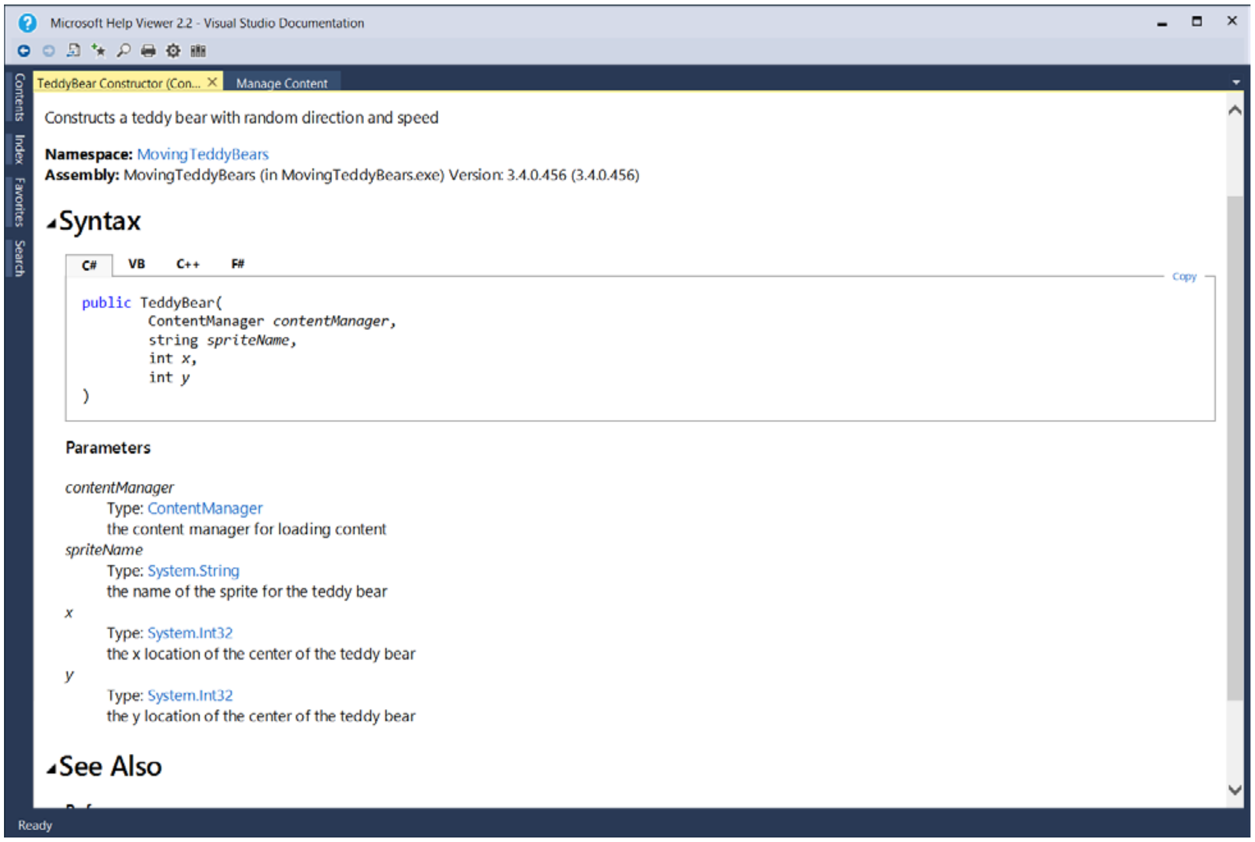
```
public const int WindowWidth = 604;
public const int WindowHeight = 453;
```

You've seen most of the syntax above before. We add `public` to our constant declaration to make it so other classes (like the `TeddyBear` class) can access those constants. The other change we need to make to use this new approach is in the `Game1` constructor, where we'll use

```
graphics.PreferredBackBufferWidth = WindowWidth;
graphics.PreferredBackBufferHeight = WindowHeight;
```

This approach lets us change the constants if we decide to change the window dimensions for our game and also gives the `TeddyBear` class access to those dimensions. Unfortunately, we lose the player control over the resolution that you'd expect to have in a commercial Windows game. In our commercial games, we use a separate class to handle all the resolution stuff to provide that functionality, but using constants like we're doing here will be fine for the games in this book.

Next we need to create our teddy bears in the `LoadContent` method. Recall from the Design a Solution step that the `TeddyBear` class exposes two constructors. Although we've already decided which constructor to use, we still don't know what information we need to provide to that constructor). How can we find out? You guessed it – from the `TeddyBear` documentation. If we click on the link for the constructor we want to use from the documentation shown in Figure 5.5, we're taken to Figure 5.6 (we scrolled down a little).


Microsoft Help Viewer 2.2 - Visual Studio Documentation
Contents
Index
Favorites
Search
TeddyBear Constructor (Con...
Manage Content
C#
VB
C++
F#
Copy
Ready

Constructs a teddy bear with random direction and speed

Namespace: MovingTeddyBears
Assembly: MovingTeddyBears (in MovingTeddyBears.exe) Version: 3.4.0.456 (3.4.0.456)

Syntax

```
public TeddyBear(
        ContentManager contentManager,
        string spriteName,
        int x,
        int y
)
```

Parameters

contentManager
Type: ContentManager
the content manager for loading content
spriteName
Type: System.String
the name of the sprite for the teddy bear
x
Type: System.Int32
the x location of the center of the teddy bear
y
Type: System.Int32
the y location of the center of the teddy bear

See Also

Figure 5.6. TeddyBear Random Velocity Constructor Documentation

We can easily see from the documentation in the Parameters section that we need to provide four pieces of information to the constructor we selected: the content manager so we can load the sprite for the teddy bear, the name of the sprite, and the x and y locations for the center of the teddy bear. Given that information, we change the `LoadContent` method to the following:

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // create the teddy bear objects
    bear0 = new TeddyBear(Content, @"graphics\teddybear0",
        graphics.PreferredBackBufferWidth / 4,
        graphics.PreferredBackBufferHeight / 4);
    bear1 = new TeddyBear(Content, @"graphics\teddybear1",
        graphics.PreferredBackBufferWidth / 2,
        graphics.PreferredBackBufferHeight / 2);
    bear2 = new TeddyBear(Content, @"graphics\teddybear2",
        graphics.PreferredBackBufferWidth * 3 / 4,
        graphics.PreferredBackBufferHeight * 3 / 4);
}
```

The `Content` argument (the first argument in each method call) is a default `ContentManager` object that the MonoGame framework automatically gives to us when we create a game project. The code above will start the three teddy bears in approximately the same locations they occupied in the Better NAG code we wrote.

If we were only going to have a single sprite for all the teddy bears we create, we wouldn't need to pass the sprite name into the constructor because the `TeddyBear` class could be coded to always load the same sprite. In our current problem, though, we need to be able to create `TeddyBear` objects using different sprites, so we need to pass the sprite name into the constructor.

The final two arguments are for the x and y locations of the center of the teddy bear. This is one of those areas where artists and programmers tend to disagree about what the location of a game object should mean. Many, though not all, artists think of the location as the upper left corner of the sprite, in large part because that's how lots of art development tools work. In contrast, most programmers think of the location as the center of the object, in large part because most game objects that can rotate do so around their center, not their upper left corner. We've found that interpreting the center of a game object as that object's location is the most intuitive and efficient approach in our commercial games, so that's the approach we'll take here.

You should notice a key difference between the code we're writing here and the code we used in the Better NAG solution. In the Better NAG code, we directly loaded each of the three sprites we needed. In an object-oriented solution, though, it makes more sense to have each object load its own content instead of having the `Game1` class do that for every object in the game; that's another good reason for loading the content in the `TeddyBear` constructor. By doing it this way, we have each object responsible for its own state, including the sprite they should display; that’s one of the key ideas behind object-oriented design.

Run your code again to make sure it still runs without breaking. You should STILL just have the cornflower blue screen! We know you're getting bored of doing this, but it really does help. For example, if you got one of the sprite names wrong in the calls to the teddy bear constructor, or you got the sprite name correct but forget to add the actual xnb file to the project, this is where your code would blow up. Don't you want to know that now rather than later?

The next thing we need to do is change the `Draw` method to tell each teddy bear to draw itself. You might think that we'd handle the update stuff next because `Update` gets called before `Draw` in our `Game1` class, but if we take care of the drawing now we can actually see something other than the cornflower blue screen. That's got to be a win! Change the `Draw` method to the following:

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

    // draw the teddy bears
    spriteBatch.Begin();
    bear0.Draw(spriteBatch);
    bear1.Draw(spriteBatch);
    bear2.Draw(spriteBatch);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Run the code, and now you'll see a screen that looks like our final output in the Better NAG; three different teddy bears spread across the screen.

Finally, we have each teddy bear update itself each time through the game loop. It should come as no surprise that we do this by changing the `Update` method to:

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed
        || Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // update the teddy bears
    bear0.Update();
    bear1.Update();
    bear2.Update();

    base.Update(gameTime);
}
```

That's all the code we need to write to solve this problem. You could run the code now to watch the teddy bears bounce around, but since we're going to do that in the next step you'll see that momentarily anyway.

Test the Code

Now we simply run our program to make sure we get the results we expected. Figure 5.7 shows a screen snap of the output when we run Test Case 1 (though of course it's hard to show in the book that the bears are actually moving!)

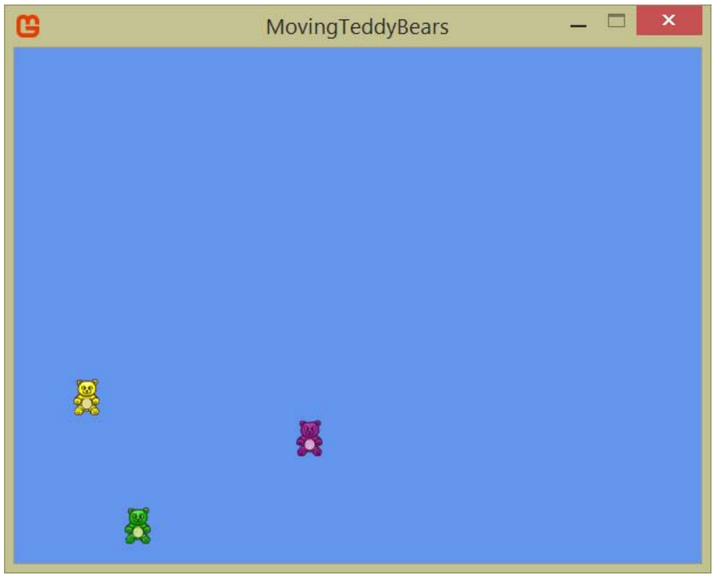


Figure 5.7. Test Case 1: Checking Bear Behavior

5.5. Common Mistakes

Trying to Load Content That's Not in the Project
If your `LoadContent` method, or any other method, tries to use a `ContentManager` to load content (like an image) that hasn't been added to your content project, your program will crash when you try to run your game. If this does happen you should carefully compare the content you're trying to load with the content that's been added to the project to try to find the problem.

Not Setting Copy to Output Directory to Copy if newer
If you forget to change the Copy to Output Directory property for any of your content to Copy if newer, you'll get a "Could not load <asset name> asset as a non-content file!" error when you try to run your game. It's easy to fix this; make sure all your content has Copy to Output Directory set to Copy if newer.

Calling the Draw Method Outside a Begin/End Block
You're required to call the `Draw` method of a `SpriteBatch` object between a call to the `Begin` method and a call to the `End` method for that `SpriteBatch` object. If you try to break this rule, your program will crash.