Cross Platform Game Development With MonoGame

Written By Michael Fleischauer
About This Book

This book, Cross Platform Game Development with MonoGame, began life as a tutorial series on GameFromScratch.com. Since the very beginning it has intended to be written in book form, and what you hold in your hands is the results of that effort. While not as in-depth as I originally intended, this book should present all you need to know to get up and running using the MonoGame framework, including graphics, input, 3D, audio and more.

MonoGame is the spiritual and literal successor to the Microsoft’s popular but now defunct XNA game framework. I won’t go into much detail as I cover the history quite thoroughly in Chapter One. Just know that it is an open source, C# based cross platform library for creating games for Windows, Mac, Linux, Android and iOS, as well as a couple of consoles. Several popular and successful commercial games have been written or ported using MonoGame.

While the journey is a short one, I hope you find the contents of this book useful in learning XNA/MonoGame. In addition to the text you see here, there is also a corresponding video series available on YouTube. The playlist for this series is http://bit.ly/1Fwb3DC or available on the GameFromScratch channel at https://www.youtube.com/user/gamefromscratch.

All of the source code used in this book is available on Github at https://github.com/serapth/MonoGameBook. It is provided as a single Visual Studio 2013 solution with separate projects for each code sample from each chapter. Although each example is provided for Windows only, the code should work on any supported platform.

If you enjoy this book or the tutorials on GameFromScratch.com, please consider supporting future development on Patreon at https://www.patreon.com/gamefromscratch.
About The Author

Michael Fleischauer has spent the last several years creating and running the popular game development site, GameFromScratch.com. Home to hundreds of tutorials and guides covering all the most popular game development technologies, Michael strives to make game development as accessible to as many people as possible.

In addition to game development, Michael has spent 15+ years as a professional programmer in a number of industries. He is also the author of PlayStation Mobile Development Cookbook, which thanks to the early demise of that technology, about 7 people have read.

Michael lives with his wife and daughter in Toronto, Canada. He is an 11th degree blackbelt, inventor of the Microwave oven and the teller of the occasional lie. He is also sick of talking about himself in the third person.
Table Of Contents

Chapter One
An Introduction and Brief History

Chapter Two
Getting Started with MonoGame on Windows

Chapter Three
Getting Started with MonoGame on MacOS

Chapter Four
Creating an Application

Chapter Five
2D Graphics

Chapter Six
Audio Programming

Chapter Seven
3D Graphics
Chapter One

An Introduction and Brief History

Welcome to a brand new series on GameFromScratch.com covering the MonoGame open source game framework. Over the next several chapters we will cover almost every factor of using MonoGame for game development. First we are going to start with a bit of a history lesson and introduction to MonoGame; we'll talk about where it came from and why you should care. Don't worry, we will get to the code and the technical bits very soon!

For the more visually inclined there is also an HD video version covering mostly the same material we will discuss here.

So what exactly is MonoGame? The simple answer is, it's a cross platform, open source implementation of the XNA game libraries. XNA in turn was Microsoft's indie focused 2D/3D game library. A more complex answer than that requires a bit of a history lesson.
A Brief History of XNA

XNA stands for XNA's Not Acronymed, a recursive algorithm and very much on vogue at the time (think GNU or WINE). Although the more likely reason for the name is X's were really just quite cool at the time, XBox, X Games, XXX the Movie... XNA was announced at the Game Developer Conference (GDC) in 2004 as a great and accessible way to bring game development to the masses. At the time however, we instead got (what eventually became) MS Build and they open sourced the source code and all assets for the game MechCommander 2. However two years later XNA Game Studio was finally released.

So what exactly was XNA Game Studio? First off, it was a special version of Visual Studio for C# with an integrated pipeline for importing game assets. On top of that, it was a collection of .NET libraries aimed at making games, covering facets like input handling, graphics, audio and networking. Finally it was a special .NET runtime capable of running on PC, (eventually) Windows Phone and perhaps most importantly, XBox 360. That was the big selling point... XNA enabled basically anybody to make games for a game console. Outside of the very limited edition (and much more expensive) Net Yaroze (a special version of the PlayStation 1), nobody had ever done this. You could argue that XNA more than any other single release, gave rise to the burgeoning indie development scene we have today.
What makes XNA even more impressive is realizing the world it was released in. Game engines certainly existed, Unreal was AAA only and several million dollars per title, Unity was I believe still a Mac only product and several hundred dollars per seat. Several of the prominent open source libraries of today simply didn't exist back then. Plus the IDE XNA itself was based on, Visual Studio, cost a few thousand dollars. XNA really did bring console game development to the masses and many of the "big" indie games, titles like Fez, Braid and Terraria were create using XNA.

It wasn't all roses though, and even though Microsoft basically invented the indie developer market, it quickly pissed it away too. XNA titles, without a publisher agreement, were confined to Xbox Indie Games channel as well as Windows Phone stores. The Xbox IG channel quickly became a dumping ground, discovery became a huge factor and very few developers made money. With the release of XBox One, it seems Sony stole Microsoft's crown as the place for indie developers to shine.

Worse, and this is a story that could fill a book on it's own, there was a shake up at Microsoft and several technologies ended up having their plug pulled. XNA was one of them. Sadly, in Oct 2011, with XNA Game Studio 4.0 refresh, XNA saw it's final release.

So then, a dead and unsupported technology... why the heck should we care? Simple... XNA was... is good. Very good. Plus "unsupported" isn't exactly the right word to used. Enter...
A Brief History of MonoGame

So XNA became very popular very quickly. There was a large and growing community, several books available on the subject and new versions being released. Hands down the biggest limitation were the supported platforms, XNA was very much tied to Microsoft platforms and a limited subset at that. This lead to two different projects, SilverSprite and XNA Touch. The first was an attempt to make a code compatible version of XNA (at least, the 2D bits) that could be run in the browser using Silverlight. The other was a port of XNA to run on an OpenGL back end on mobile devices. XNA Touch ended up using the 2D code from Silver Sprite and the two eventually emerged as MonoGame.

This was several years ago and since then MonoGame has grown into a nearly 100% complete implementation of XNA 4 that can be run on several platforms ( Windows DirectX, Windows OpenGL, Mac, Linux, Android, iOS, PlayStation Vita, PlayStation 4, OUYA and more ). The biggest hanging point has always been the content pipeline. Until recently you’ve needed to keep an old version ( and a PC ) Game Studio around to convert your assets into the binary XNB format. However, near the end of 2014 the MonoGame team provided that final missing piece with the release of their own cross platform content pipeline tool. It will be interesting to see what the future holds now that the MonoGame team have basically implemented 100% of XNA. The future of XNA is now theirs to determine.
Why Should I Use MonoGame?

So, that was a bit of XNA history for you. The biggest question you may have now is... "So what? Why should I use XNA/MonoGame today?"

Sometimes it is often easier to start with why you shouldn't use something. The following are the biggest reasons I can think of not to use MonoGame.

- You want a WYSIWYG environment or integrated game editor
- You don't like C#
- You want a higher level programming experience
- There is a price tag attached to support iOS and Android

Those are probably the biggest drawbacks of using XNA. There is no world editor. In fact, there are very few tools at all, except content importers. Even the libraries themselves are fairly low level, leaving you to roll your own sprite and animation classes for example. XNA is also not the easiest library to learn, especially compared to many scripting languages or all in one solutions like GameMaker or Construct. That coding is done in C# ( or other Mono supported languages like F#, but 99% of examples are in C# ), so if you aren't a fan of that language, you won't like MonoGame. Finally, and perhaps worst, targeting iOS or Android requires a Xamarin license, and that costs money.

So that's why you shouldn't use MonoGame, now why you should.

First off, because XNA was and is just a great library! It was well designed, very clean and works well. Simply put, if you like working in C# ( and I do, very much ), working in XNA is just a very pleasant coding experience. You work at a lower level of abstraction than many of today's game engines, but after being around for about a decade, there are a huge amount of code samples and libraries to fill those pieces. The core though, those pieces every game needs, they are all provided. If you are the type of person that likes to have maximum control, to dig in deep and get your finger nails dirty, you'll feel right at home!
Also, MonoGame provides access to pretty much every single platform you’d want to target today. One of the scary things about working with some open source game project is, you never know how well they will work in the real world. Fortunately, MonoGame has an pretty impressive resume of shipped titles, such as:

- Bastion
- Fez
- Infinite Flight
- Skulls of the Shogun
- Transistor

Some of the biggest selling indie game, from both the past and the present, prove that MonoGame is capable.

So basically MonoGame is a battle proven cross platform indie friendly code focused C# based game engine built on the bones of Microsoft's XNA. Now, let’s jump into the technical aspects and you can see why I am such a big fan of the XNA libraries in the first place. Along the way we will cover the way MonoGame does things differently, and look at some of the unique concerns that cross platform development bring. Next however, let’s look at getting MonoGame up and running on your platform of choice.
Chapter Two

Getting Started with MonoGame on Windows

This tutorial is a quick guide on getting MonoGame up and running on Windows OS. The process is incredibly straight forward, with two options on how to proceed.

First you are going to need a development environment, of which two options are available. Xamarin's Xamarin Studio or Microsoft's Visual Studio. If you wish to go the Xamarin Studio route, be sure to check the MacOS guide, as the process will be virtually identical. The rest of this tutorial assumes that you chose to install Visual Studio, the industry standard IDE for Windows development which is now thankfully available for free.

So first things first, download and install Visual Studio 2013 Community. Be sure that you select Community and not one of the 90 trial editions. The community install is a complete and full functioning version of Visual Studio, but with some limitations on the size of your company.

As of time of writing, this is the version you want.
If you want to talk a walk on the wild side, the release candidate of Visual Studio 2015 will also work. Of course, it's a release candidate... so buyer beware.

Installing either with the minimal recommendations or better will get you all that you need installed.

**Install using the Installer**

By far the easiest option, simply download and run the installer available here. Be sure to shut down Visual Studio before installing.

Click Next, then agree to the EULA... after you read it and submit it to your lawyer for approval of course... Next you will be prompted for the features you want installed. The defaults are pretty solid, click Install:
Install using NuGet
Visual Studio integrates a package manager called NuGet. This offers a few (potential) benefits over using the library's standalone installer.

- dependency resolution. If it depends on other libraries, NuGet can handle those dependencies
- package updating abilities. Keep yourself up to date easier

Actually, that's about it. Basically if you want to be kept up to date on updates, this is the route to go. The install process is certainly more complicated though, at least initially.
First of course you need the NuGet package manager installed. It's getting more and more common in use, so you will probably have it installed or need it installed shortly. It is available as a Visual Studio extension or command line utility.

To install with NuGet, Launch Visual Studio, on first run you may have to make some configuration choices, don't worry, most of these can be revisited later on. Once configured, select the menu Tools->NuGet Package Manager->Package Manager Console:
Now you simply install the packages you want. Unlike the installer you download the various packages independently. The list of packages are available here. Assuming you are going to develop initially on Windows, you probably want to start with the DirectX Windows version. To install that in the Package Manager Console type:

**Install-Package MonoGame.Framework.WindowsDX**

This will download and install all the required files and dependencies. For more details on the MonoGame NuGet packages please read this post on StackOverflow by the maintainer.
Your First Project

Now that you've got MonoGame installed let's create our first project. If not already done, load up Visual Studio.

Select File->New Project

In the resulting dialog, on the left hand side under installed templates, select Visual C#->MonoGame, then on the right hand side select MonoGame Windows Project, pick a location, a project name, if you want source control, then click OK.
As you can see, MonoGame ships out of the box with templates for a number of different targets and a few above may require a bit of an explanation. The MonoGame Windows Project targets Windows desktop using Direct X for the backend. The OpenGL project is another Windows target, but instead using OpenGL as the backend. As DirectX is Windows, XBox and WinPhone only, you may find using the GL backend the most consistent if targeting Mac, Linux, Android and/or iOS, as those all use OpenGL on the back end. A Windows 8.1 Universal project is an application that supports both Win 8 desktop and mobile targets with one code base, and if I am honest, with the announcement of Windows 10, is probably a complete dead end.

Ideally you will write most of your code as libraries, with only the platform specific portions in their own corresponding project file. We will look at this process closer down the road. For now we will KISS (Keep It Simple Stupid) and just target the platform we are developing on.

Once you click OK, the following simple project will be created:

```csharp
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
```
namespace Game1
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    public class Game1 : 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()
        {
            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
/// game-specific 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)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
        ButtonState.Pressed || Keyboard.GetState().IsKeyDown(
            Keys.Escape))
        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)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
Don’t concern yourself overly with the code, we will explain it all shortly. We just want to make sure that your MonoGame install is up and running. To run your game hit F5 or press:

Assuming all went well, you should see a window drawn in CornFlower blue:

Congratulations, you’ve just created your first ever MonoGame application!

Let’s talk briefly about the Content Pipeline. This is the way you get content into your MonoGame application. If you look in the Solution Explorer, you will see a file named Content.mgcb.
Double click it, and the MonoGame Content Pipeline tool will open:

Let’s quickly add a texture to our project. Select Edit->Add->Existing Item, navigate to an existing image file somewhere on your computer.

Next you will be prompted for how you want the file to be added, copied or linked, I chose copy to break the connection with the source image. This means updates to the original image will not be propagated, nor will
deleting it have any effect.

The image will be added to your content bundle, as as you can see with it selected, there are a number of properties exposed for how the image should be processed when building content:

There are several Import processors available for various different file types:
Each one exposes different parameters you can modify. For images some of the key properties exposed are the option to automatically resize the image to a power of 2 dimension, to change the texture compression settings (TextureFormat) or setting the color key value used for transparencies.

Now select the Content root node and you will see the property details have changed:

```
<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common</td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>C: Users\Mike\Documents\Visual Studio 2019\Projects\MyGame\Content</td>
</tr>
<tr>
<td>Name</td>
<td>Content</td>
</tr>
<tr>
<td>Misc</td>
<td>False</td>
</tr>
<tr>
<td>Settings</td>
<td></td>
</tr>
<tr>
<td>Compress</td>
<td>False</td>
</tr>
<tr>
<td>Config</td>
<td></td>
</tr>
<tr>
<td>Intermediate Folder</td>
<td>obj\Windows</td>
</tr>
<tr>
<td>Output Folder</td>
<td>bin\Windows</td>
</tr>
<tr>
<td>Platform</td>
<td>Windows</td>
</tr>
<tr>
<td>Profile</td>
<td>Reach</td>
</tr>
<tr>
<td>References</td>
<td>(Collection)</td>
</tr>
<tr>
<td>Statistics</td>
<td></td>
</tr>
<tr>
<td>Total Items</td>
<td>1</td>
</tr>
</tbody>
</table>
```

The key setting here is Platform. Each different platform can have slightly different requirements and the Content Pipeline takes care of that for you. In this case we already have Windows set for the pipeline, so there is nothing that needs to be changed.

Now Build your content using the Build->Build menu option, or by hitting F6.
Now back in Visual Studio, confirm that the build action on your Content.mgcb file is set correctly. Right click the file and select Properties:

Make sure that Build Action is set to MonoGameContentReference.

This will enable you to use the content as if it was installed locally, making switching between platform versions trivial.

Now actually using content in code is as simple as:

```csharp
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    var myImage = this.Content.Load<Texture2D>("SBO6h0Gk");
}
```

Don’t worry, we will cover this process in more detail later. Just note that the file extension is not used. Behind the scenes, you may notice that the Content Pipeline tool created the following xnb file in the /bin/
Windows subdirectory of Content:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBO6h0Gk.xnb</td>
<td>6/10/2015 2:44 PM</td>
<td>XNB File</td>
</tr>
</tbody>
</table>
Chapter Three

Getting Started with MonoGame on MacOS

In this tutorial we are going to look at getting a MonoGame development environment up and running on MacOS, create and run our first MonoGame application and finally take a quick look at the Content Pipeline tool.

For those that prefer video, there is a video version of this tutorial available here or embedded below.

The Installation Process

In order to get up and running there are a couple things you need to install, Mono itself then MonoGame. We will also be installing Xamarin Platform, which includes Xamarin's cross platform Mono based IDE Xamarin Studio. It is not strictly required, but will make your life a great deal simpler.

The process is as straight forward as installs can get, simple download an run each installer. There is not ambiguity to the install, however the install order is extremely important. Download the following in this order. In each case be sure to install the most recent version if given an option.

- Mono Developer Kit
- Xamarin Platform
- MonoGame

As I said earlier, there should be nothing complicated about any of those installs, except the order. If you install out of order the proper templates for MonoGame won't get added to Xamarin Studio.
Your First Project

Now let's fire up Xamarin Studio and create our first application. Once Xamarin loads, select New Solution...

In the resulting dialog, next select the project template to get started. Along the left hand side scroll down and locate Other->Miscellaneous. On the right hand side you should see a number of preconfigured MonoGame templates. For this example we are going to target only Mac OS using Xamarin's Mac libraries, so select MonoGame Mac Application (Xamarin.Mac), then click Next.
All that remains is to pick a location and name for your application as well as if you want Git control or not. When ready click Create.
This should create a simple application for us with the following code:

```csharp
#region Using Statements
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Storage;
using Microsoft.Xna.Framework.Input;

namespace HelloMonoGame
{
    /// <summary>
    /// This is the main type for your game.
    /// </summary>
    /// <summary>
    public class Game1 : Game
```
public Game1 ()
{
    graphics = new GraphicsDeviceManager (this);
    Content.RootDirectory = "Content";
    graphics.IsFullScreen = true;
}

/// <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>
/// Allows the game to run logic such as updating the world,
protected override void Update (GameTime gameTime)
{
    // For Mobile devices, this logic will close the Game when the
    Back button is pressed
    // Exit() is obsolete on iOS
#if !__IOS__
    if (GamePad.GetState (PlayerIndex.One).Buttons.Back ==
        ButtonState.Pressed ||
        Keyboard.GetState ().IsKeyDown (Keys.Escape)) {
        Exit ();
    }
#endif
    // TODO: Add your update logic here
    base.Update (gameTime);
}

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

    //TODO: Add your drawing code here

    base.Draw (gameTime);
}

We aren't going to go into any specifics of what this code does, not yet anyways, but we will make sure that it runs. Hit the Play icon in the top control bar:
You should now see a full screen cornflower blue window. If so, congratulations, you just successfully made your first Monogame application!
The Content Pipeline
The content pipeline was a major component missing from MonoGame, but they have recently implemented one. The Content Pipeline takes external assets for your game, be them videos, textures, sounds, fonts, etc... converts them for your platform of choice, generating a XNB file, a binary format that XNA used internally. Let's take a look at the process in MonoGame.

If you look under the Content folder of your Solution, you will see a file called Content.mgcb.

Double click this and the Content Pipeline tool will load:
This is the tool you use to assemble the assets your game is going to use. Let's take a quick example of importing a texture.

First select Edit->Add->Existing Item...
Browse to the image you wish to import as a texture then select OK.

Now you will be asked how you wish to import the file. I want a copy to be made so the original is unaffected.
Your texture should now be added:

You now have a few options about how the content will be processed on import. With the image selected, notice the options available below it:
Each different importer has a different set of parameters you can modify. Here you can set the automatic generation of mip maps, resize to power of 2 dimensions, change the compression settings, etc.

By selecting the Importer drop down you can get an idea of the various type of assets supported by the Content Pipeline:
Now select the Content node and the properties available will change again. This time we simply want to change the target platform to MacOS:
You can have the Content Pipeline automatically process content for a variety of different platforms. Now that this task is done let's actually build our content. Simply select Build->Build or hit F6:
In the directory [Content]/bin/MacOSX there will now be an XNB file available for use. Let's add it to our project.

In Xamarin Studio, right click the Content folder and select Add->Add Files...

Navigate to the XNB file in Content/bin/MacOSX that you just created and select it. Then select Override Build action and choose 'Content'
This time when prompted how to add the file, choose the Link option:
Congratulations, you have now added an asset to your game. In future tutorials we will look at using them in more detail.
Creating an Application

In this chapter we are going to look closely at the structure of a typical XNA game. By the end you should have a better idea of the life cycle of a typical MonoGame application, from program creation, loading content, the game loop, unloading content and exiting.

If you prefer videos to text, you can watch this content in HD video right here.

Let's start by looking at the code for an automatically generated application, stripped of comments. There are two code files in the generated project, Program.cs and [YourProjectName].cs. Let's start with Program.cs

```csharp
using System;

namespace Example1 {
    
#if WINDOWS || LINUX
    public static class Program {
    
        [STAThread]
        static void Main() {
            using (var game = new Game1())
                game.Run();
        }
    }
#endif

    }
}
```
The heart of this code is the creation of our Game object, then calling it's Run() method, which starts our game executing, kicking off the game loop until the Game execution finishes. We will talk about the game loop in a bit more detail later on. Otherwise this is a standard C# style application, with Main() being the applications entry point. This is true for Windows and Linux applications at least, which is the reason for the #if preprocessor directive. We will discuss the entry point of various other platforms later on, so don't worry about this too much. Also note, if you didn't select Windows as your platform when creating this project, your own Program.cs file contents may look slightly different. Again, don't worry about this right now, the most important thing is to realize that Program creates and runs your Game derived class.

Predefined Platform Values

One of the major features of MonoGame over XNA is the addition of several other supported platforms. In addition to WINDOWS and LINUX symbols, the following platforms have been defined:

- ANDROID
- IOS
- LINUX
- MONOMAC
- OUYA
- PSM
- WINDOWS
- WINDOWS_PHONE
- WINDOWS_PHONE81
- WINDOWSRT
- WEB

Of course, new platforms are being added all of the time, so this list may not still be current. You can look up the definitions in the MonoGame sources in the file /Build/Projects/MonoGame.Framework.definition in the MonoGame GitHub repository.

Please note, there are plenty of other defines per platform, for example iOS, Android, MacOS, Ouya and WindowsGL all also define OPENGL. You can use these predefined symbols for implementing platform or library specific code.

Now let's move on to our Game class

```csharp
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
```
namespace Example1
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

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

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

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

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
                Exit();
            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.CornflowerBlue);
            base.Draw(gameTime);
        }
    }
}
Our game is derived from the class `Microsoft.Xna.Framework.Game` and is the heart of our application. The Game class is responsible for initializing the graphics device, loading content and most importantly, running the application game loop. The majority of our code is implemented by overriding several of Game's protected methods.

Let's take a look at the code, starting from the top. We create two member variables, graphics and spriteBatch. GraphicsDeviceManager requires a reference to a Game instance. We will cover the GraphicsDeviceManager and SpriteBatch classes shortly in the graphics chapter, so please ignore them for now.

Next we override the Initialize(), LoadContent and UnLoadContent methods. The most immediate question you've probably got is, why have an Initialize() method at all, why not just do initialization in the constructor. First, behavior of calling a virtual function from a constructor can lead to all kinds of hard to find bugs in a C# application. Second, you don't generally want to do any heavy lifting in a constructor. The existence of LoadContent() however, will often leave you will an empty Initialize() method. As a general rule, perform initializations that are required (like GraphicsDeviceManager's allocation) for the object to be valid in the constructor, perform long running initializations (such as procedural generation of terrain) in Initialize() and load all game content in LoadContent().

Next we override Update() and Draw(), which is essentially the heart of your application's game loop. Update() is responsible for updating the state of your game world, things like polling input or moving entities, while Draw is responsible for drawing your game world. In our default Update() call, we check for the player hitting the back button or escape key and exit if they do. Don't worry about the details, we will cover Input shortly. In Draw() we simply clear the screen to CornFlower Blue (an XNA tradition). You will notice in both examples we call the base class as well.

What's a game loop?

A game loop is essentially the heart of a game, what causes the game to actually run. The following is a fairly typical game loop:

```csharp
void gameLoop()
{
    while (game != DONE)
    {
        getInput();
        physicsEngine.stepForward();
        updateWorld();
        render();
    }
    cleanup();
}
```

As you can see, it's quite literally a loop that calls the various functions that make your game a game. This is
obviously a rather primitive example but really 90% of game loops end up looking very similar to this.

However, once you are using a game engine or framework, things behave slightly different. All this stuff still happens, it's just no longer your code's responsibility to create the loop. Instead the game engine performs the loop and each iteration it then calls back to your game code. This is where the various overridden function such as update() and draw() are called. Looking at our sample loop above though, you might notice a physics engine call. XNA doesn't have a built in physics engine, so instead of the game loop updating the physics, you will have to do it yourself in your games update() call.

When you run this code you should see:

![Sample window](image)

Please note, depending on what platform you are running on, this window may or may not be created full screen. On Windows it defaults to windowed, while on MacOS it defaults to full screen. Hit the Escape key, or Back button if you have a controller installed, to exit the application.

Let's take a quick look at a program's lifecycle with this handy graphic.
In a nutshell the game is created, initialized, content is loaded, the game loop runs until Exit() is called, then the game cleans up and exits. There are actually a few more methods behind the scenes, such as BeginDraw() and EndDraw(), but for most games, this is sufficient detail.

Our current example isn't exactly exciting because absolutely nothing happens. Let's create a slightly more interesting example, one that draws a rectangle on screen and rolls it across the screen. Don't worry about the specifics, we will cover graphics in more detail shortly.

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

// This example simply adds a red rectangle to the screen
// then updates it's position along the X axis each frame.
namespace Example2
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
    }
}
Texture2D texture;
Vector2 position;

public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    position = new Vector2(0, 0);
}

protected override void Initialize()
{
    texture = new Texture2D(this.GraphicsDevice, 100, 100);
    Color[] colorData = new Color[100 * 100];
    for (int i = 0; i < 10000; i++)
        colorData[i] = Color.Red;

    texture.SetData<Color>(colorData);
    base.Initialize();
}

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

protected override void UnloadContent()
{
}

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

    position.X += 1;
    if (position.X > this.GraphicsDevice.Viewport.Width)
        position.X = 0;
    base.Update(gameTime);
}
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    spriteBatch.Draw(texture, position);
    spriteBatch.End();
    base.Draw(gameTime);
}

When we run this we will see:

![Example2](image)

Nothing exciting, but at least our game does something now. Again, don't worry overly about the details of how, we will cover this all later. What we want to do is look at a few key topics when dealing about dealing with the game loop.
Pausing Your Game

Pausing your game is a pretty common task, especially when an application loses focus. If you take the above example, minimize the application, then restore it and you will notice the animation continued to occur, even when the game didn't have focus. Implementing pause functionality is pretty simple though, let's take a look at how:

```csharp
protected override void Update(GameTime gameTime)
{
    if (IsActive)
    {
            Exit();
        position.X += 1;
        if (position.X > this.GraphicsDevice.Viewport.Width)
            position.X = 0;
        base.Update(gameTime);
    }
}
```

Well that was simple enough. There is a flag set, IsActive, when your game is active or not. The definition of IsActive depends on the platform it's running. On a desktop platform, an app is active if it's not minimized AND has input focus. On console it's active if an overlay like the Xbox guide is not being shown, while on phones it's active if it's the running foreground application and not showing a system dialog of some kind.

As you can see, the can pause the game by simply not performing Game::Update() calls.

There may be times when you wish to perform some activity when you gain/lose active status. This can be done with a pair of event handlers:

```csharp
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
}
```
position = new Vector2(0, 0);

this.Activated += (sender, args) => {
    this.Window.Title = "Active Application";
};
this.Deactivated += (sender, args) => {
    this.Window.Title = "InActive Application";
};

Or by overriding the functions OnActivated and OnDeactivated, which is the recommended approach:

protected override void OnActivated(object sender, System.EventArgs args)
{
    this.Window.Title = "Active Application";
    base.OnActivated(sender, args);
}

protected override void OnDeactivated(object sender, System.EventArgs args)
{
    this.Window.Title = "InActive Application";
    base.OnActivated(sender, args);
}
Controlling the Game Loop

Another challenge games face is controlling the speed games run at across a variety of different devices. In our relatively simple example there is no problem for two reasons. First, it's a very simple application and not particularly taxing, meaning any machine should be able to run it extremely quickly. Second, the speed is actually being capped by two factors, we are running at a fixed time step (more on that later) and we have vsync enabled, which on many modern monitors, refresh at a 59 or 60hz rate. If we turn both of these features off and let our game run at maximum speed, then suddenly the speed of the computer it's running on becomes incredibly important:

```csharp
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    position = new Vector2(0, 0);
    this.IsFixedTimeStep = false;
    this.graphics.SynchronizeWithVerticalRetrace = false;
}
```

By setting IsFixedTimeStep to false and Graphics.SynchronizeWithVerticalRetrace to false, our application will now run as fast as the game is capable. The problem is, this will result in our rectangle being drawn extremely fast and at different speeds on different machines. We obviously don't want this, but fortunately there is an easy fix. Take a look at Update() and you will notice it is being passed in a GameTime parameter. This value contains the amount of time that occurred since the last time Update() was called. You may also notice Draw() also has this parameter. This value can be used to smooth out motion so it runs at a consistent speed across machines. Let's change our update call so position instead of being ++'ed each frame, we now move at a fixed rate, say 200 pixels per second. That can be accomplished with this change to your Update() position code:

```csharp
position.X += 200.0f * (float)gameTime.ElapsedGameTime.TotalSeconds;
```

TotalSeconds will contain the fraction of a second that have elapsed since Update() was last called, assuming of course your game is running at at least 1 frame a second! For example, if your game is updating a 60hz (second times per second), then TotalSeconds will have a value of 0.016666 (1/60). Assuming your stays
pretty steady at 60fps, this results in your Update code updating the position by 3.2 pixels per frame \((200 \times 0.016)\). However, on a computer running at 30fps, this same logic would then update position by 6.4 pixels per frame \((2000 \times (1/30))\). The end result is the behavior of the game on both machines is identical even though one draws twice as fast as the other.

The `GameTime` class contains a couple useful pieces of information:

<table>
<thead>
<tr>
<th><code>ElapsedGameTime</code></th>
<th>{00:00:00.0026065}</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ticks</td>
<td>26065</td>
</tr>
<tr>
<td>Days</td>
<td>0</td>
</tr>
<tr>
<td>Hours</td>
<td>0</td>
</tr>
<tr>
<td>Milliseconds</td>
<td>2</td>
</tr>
<tr>
<td>Minutes</td>
<td>0</td>
</tr>
<tr>
<td>Seconds</td>
<td>0</td>
</tr>
<tr>
<td>Ticks</td>
<td>26065</td>
</tr>
<tr>
<td>TotalDays</td>
<td>0.00000003016782407407</td>
</tr>
<tr>
<td>TotalHours</td>
<td>0.000007240277777777</td>
</tr>
<tr>
<td>TotalMilliseconds</td>
<td>2.6065</td>
</tr>
<tr>
<td>TotalMinutes</td>
<td>0.0004344166666666666</td>
</tr>
<tr>
<td>TotalSeconds</td>
<td>0.0026065</td>
</tr>
</tbody>
</table>

`ElapsedGameTime` holds information on how much time has happened since the last call to `Update` (or `Draw`, completely separate values by the way). As you just saw, this value can be used to normalize behavior regardless to how fast the underlying machine actually performs. The value `TotalGameTime` on the other hand is the amount of time that elapsed since game started, including time spend paused. Finally there is the `IsRunningSlowly` flag, which is set if the game isn't hitting it's target elapsed time, something we will discuss momentarily. `Game` also has a property called `InactiveSleepTime`, which along with `TotalGameTime` can be used to calculate the amount of time a game spent running.
Fixed Vs. Variable TimeStep
Finally let's discuss using a FixedStep game loop instead. Instead of messing with all of this frame normalization stuff you can instead say "Hey, Monogame, I want you to run my game at this speed" and it will do it's best. Let's look at this process:

```c#
public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";
    position = new Vector2(0, 0);
    this.IsFixedTimeStep = true;
    this.graphics.SynchronizeWithVerticalRetrace = true;
    this.TargetElapsedTime = new System.TimeSpan(0, 0, 0, 0, 33); // 33ms = 30fps
}
```

This will then try to call the Update() exactly 30 times per second. If it can't maintain this speed, it will set the IsRunningSlowly flag to true. At the same time, the engine will try to call Draw() as much as possible, but if the machine isn't fast enough to hit the TargetElapsedTime it will start skipping Draw() frames, calling only Update() until it's "caught up". If your You can control the maximum amount of draw calls that will be skipped per Update() by setting MaxElapsedTime. This value is a MonoGame specific extension and is not in the original XNA. If you don't specify a TargetElapsedTime, the default is 60fps (1/60).

Ultimately the decision between using a Fixed Step game loop or not is ultimately up to you. A fixed step game loop has the advantage of being easier to implement and provides a more consistent experience. A variable step loop is a bit more complicated, but can result in smoother graphics on higher end machines. The result is more pronounced on a 3D game than a 2D one, resulting in smoother controls and slightly cleaner visuals. In a 2D game the difference is much less pronounced.

In the next chapter we will move on to the much more exciting topic of 2D graphics.
Chapter Five

Textures and SpriteBatch

Now we move on to a topic that people always seem to love, graphics! In the past few chapters/videos I've said over and over "don't worry, we will cover this later", well... welcome to later. We are primarily going to focus on loading and displaying textures using a SpriteBatch. As you will quickly discover, this is a more complex subject than it sounds.

Before we can proceed too far we need a texture to draw. A texture can generally be thought of as a 2D image stored in memory. The source image of a texture can be in bmp, dds, dib, hdr, jpg, pfm, png, ppm or tga formats. In the "real world" that generally means bmp, jpg or png formats and there is something to be aware of right away. Of those three formats, only png and some jogs have an alpha channel, meaning it supports transparency out of the box. There are however ways to represent transparency in these other formats, as we will see shortly. If you've got no idea which format to pick, or why, pick png.

Using the Content Pipeline

If you've been reading since the beginning you've already seen a bit of the content pipeline, but now we are going to actually see it in action with a real world example.

Do we have to use the content pipeline for images?

I should make it clear, you can load images that haven't been converted into xnb format. As of XNA 4, a simpler image loading api was added that allowed you to load gif, jpg and png files directly with the ability to
crop, scale and save. The content pipeline does a lot for you though, including massaging your texture into a platform friendly format, potentially compressing your image, generation of mip maps or power of two textures, premultiplied alpha (explained shortly), optimized loading and more. MonoGame included a number of methods for directly loading content to make up for it's lack of a working cross platform pipeline. With the release of the content pipeline tool, these methods are deprecated. Simply put, for game assets (aka, not screen shots, dynamic images, etc), you should use the content pipeline.

Create a new project, then in the Contents folder, double click the file `Content.mgcb`.

This will open the MonoGame Content Pipeline tool. Let's add our texture file, simple select Edit->Add->Existing Item...

Navigate to a select a compatible image file. When prompted chose the mode that makes the most sense. I want the original to be untouched, so I am choosing Copy the file to the directory.
Your content project should now look like:

![Image of AddFileDialog]

The default import settings for our image are fine, but we need to set the Content build platform. Select Content in the dialog pictured above, then under Platform select the platform you need to build for.
Note the two options for Windows, Windows and WindowsGL. The Windows platform uses a DirectX backend for rendering, while WindowsGL uses OpenGL. This does have an effect on how content is processed so the difference is important.

Now select Build->Build, saving when prompted:

You should get a message that your content was built.

Build started 6/19/2015 9:00:45 AM
C:/Users/Mike/Dropbox/Projects/MonoGame/Code/MonoGameCodeSamples/Chapter5/Example1/Content/logo.png
Build 1 succeeded, 0 failed.
Time elapsed 00:00:00.72.

We are now finished importing, return to your IDE.
Important Platform Specific Information

One Windows the .mgcb file is all that you need. When the IDE encounters it, it will basically treat it as a symlink and instead refer to the contents it contains. Currently when building on MacOS using Xamarin, you have to manually copy the generated XNB contents into your project and set their build type as Content. The generated files are located in the Output Folder as configured in the Content Pipeline. I have been notified that a fix for this is currently underway, so hopefully the Mac and Windows development experience will be identical soon.

Alright, we now have an image to work with, let's jump into some code.

Loading and displaying a Texture2D

So now we are going to load the texture we just added to the content project, and display it on screen. Let's just jump straight into the code.

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

namespace Example1
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D texture;

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

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

        protected override void LoadContent()
        {
```
spriteBatch = new SpriteBatch(GraphicsDevice);
texture = this.Content.Load<Texture2D>("logo");
}

protected override void UnloadContent()
{
    //texture.Dispose(); <-- Only directly loaded
    Content.Unload();
}

protected override void Update(GameTime gameTime)
{
        Exit();
    base.Update(gameTime);
}

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

    spriteBatch.Begin();
    spriteBatch.Draw(texture, Vector2.Zero);
    spriteBatch.End();

    base.Draw(gameTime);
}

When we run this code we see:
Obviously your image will vary from mine, but our texture is drawn on screen at the position (0,0).

There are a few key things to notice here. First we added a Texture2D to our class, which is essentially the in memory container for our texture image. In LoadContent() we then load our image into our texture using the call:

```csharp
texture = this.Content.Load<Texture2D>("logo");
```

You notice we use our Game's Content member here. This is an instance of Microsoft.Xna.Framework.ContentManager and it is ultimately responsible for loading binary assets from the content pipeline. The primary method is the Load() generic method which takes a single parameter, the name of the asset to load **minus the extension**. Notice the bold there? That's because this is a very common tripping point. In addition to Texture2D, Load() supports the following types:

- Effect
- Model
- SpriteFont
- Texture
- Texture2D
- TextureCube

It is possible to extend the processor to support additional types, but it is beyond the scope of what we are covering here today.

Next we get to the UnloadContent method, where we simply call Content.Unload(); The ContentManager
"owns" all of the content it loads, so this cleans up all of the memory for all of the objects loaded through the ContentManager. Notice I left a commented out example calling Dispose(). It's important to know if you load a texture outside of the ContentManager or create one dynamically, it's is your responsibility to dispose of it or you may leak memory. You may say, hey, this will all get cleaned up on program exit anyways. Honestly this isn't technically wrong, although cleaning up after yourself is certainly a good habit to get into.

Memory Leaks in C#?

Many new to C# developers think because it's managed you can't leak memory. This simply isn't true. While compared to languages like C++, memory management is much simpler in C#, it is still quite possible to have memory leaks. In C# the easiest way is to not Dispose() of classes that implement IDisposable. An object that implements IDisposable owns an unmanaged resource (such as a Texture) and that memory will be leaked if someone doesn't call the Dispose() method. Wrapping the allocation in a using statement will result in Dispose() being called at the end of scope. As a point of trivia, other common C# memory leaks are caused by not removing event listeners and of course, calling leaky native code (pInvoke).

Now that we have our texture loaded, it's time to display it on screen. This is done with the following code:

```csharp
spriteBatch.Begin();
spriteBatch.Draw(texture, Vector2.Zero);
spriteBatch.End();
```

I will explain the SpriteBatch in a few moments, so let's instead focus on the Draw() call. This needs to be called within a Begin()/End() pair. Let's just say SpriteBatch.Draw() has A LOT of overloads, that we will look at now. In this example we simply Draw the passed in texture at the passed in position (0,0). Next let's look at a few of the options we have when calling Draw().

Where is 0,0?

Different libraries, frameworks and engines have different coordinate systems. In XNA, like most windowing or UI libraries, the position (0,0) refers to the top left corner of the screen. For sprites, (0,0) refers to the top left corner as well, although this can be changed in code. In many OpenGL based game engines, (0,0) is located at the bottom left corner of the screen. This distinction becomes especially important when you start working with 3rd party libraries like Box2D, which may have a different coordinate system. Using a top left origin system has advantages when dealing with UI, as your existing OS mouse and pixel coordinates are the same as your game's. However the OpenGL approach is more consistent with mathematics, where positive X and Y coordinate values refer to the top right quadrant on a Cartesian plane. Both are valid options, work equally well, just require some brain power to convert between.

Translation and Scaling
Cross Platform Game Development using MonoGame 63

```csharp
spriteBatch.Draw(texture, destinationRectangle: new Rectangle(50, 50, 300, 300));
```

This will draw our sprite at the position (50,50) and scaled to a width of 300 and a height of 300.

[Image of a sprite drawn at (50,50) with a width and height of 300]

**Rotated**

```csharp
spriteBatch.Draw(texture, destinationRectangle: new Rectangle(50, 50, 300, 300), rotation:-45f
);
```

This will rotate the image -45 degrees about its origin.

[Image of a sprite rotated -45 degrees]

Notice that the rotation was performed relative to the top left corner of the texture. Quite commonly when rotating and scaling you would rather do it about the sprite's mid point. This is where the origin value comes in.

**Rotated about the Origin**

```csharp
spriteBatch.Draw(texture, destinationRectangle: new Rectangle(150 + 50, 150 + 50, 300, 300),
```

Ok, this one may require a bit of explanation. The origin is now the midpoint of our texture, however we are going to be translating and scaling relative to our midpoint as well, not the top left. This means the coordinates passed into our Rectangle need to take this into account if we wish to remained centered. Also you need to keep in mind that you are resizing the texture as part of the draw call. This code results in:

```
origin: new Vector2(texture.Width/2, texture.Height/2),
rotation: -45f
```

For a bit of clarity, if we hadn't translated(moved) the above, instead used this code:

```
spriteBatch.Draw(texture, destinationRectangle: new Rectangle(0, 0, 300, 300),
                  origin: new Vector2(texture.Width/2, texture.Height/2),
                  rotation: -45f
```

We would rotate centered to our sprite, but at the origin of our screen

So it’s important to consider how the various parameters passed to draw interact with each other!
Tinted

spriteBatch.Begin();
spriteBatch.Draw(texture,
    destinationRectangle: new Rectangle(50, 50, 300, 300),
    color:Color.Red);
spriteBatch.End();

The Color passed in (in this case Red) was then added to every pixel in the texture. Notice how it only effects the texture, the Cornflower Blue background is unaffected. The additive nature of adding red to blue resulted in a black-ish colour, while white pixels simply became red.

Flippped

spriteBatch.Draw(texture,
    destinationRectangle: new Rectangle(50, 50, 300, 300),
    effects:SpriteEffects.FlipHorizontally|SpriteEffects.FlipVertically);

That’s about it for draw, now let’s look a bit closer at SpriteBatch.
**SpriteBatch**

In order to understand exactly what SpriteBatch does, it's important to understand how XNA does 2D. At the end of the day, with modern GPUs, 2D game renderers no longer really exist. Instead the renderer is actually still working in 3D and faking 2D. This is done by using an orthographic camera (explained later, don't worry) and drawing to a texture that is plastered on a 2D quad that is parallel to the camera. SpriteBatch however takes care of this process for you, making it feel like you are still working in 2 dimensions.

That isn't it however, SpriteBatch is also a key optimization trick. Consider if your scene consisted of hundreds of small block shape sprites each consisting of a small 32x32 texture, plus all of the active characters in your scene, each with their own texture being drawn to the screen. This would result in hundreds or thousands of Direct3D or OpenGL draw calls, which would really hurt performance. This is where the "Batch" part of sprite batch comes in. In its default operating mode (deferred), a simply queues up all of the drawing calls, they aren't executed until End() is called. It then tries to "batch" them all together into a single draw call, thus rendering as fast as possible.

There are settings attached to a SpriteBatch called, specified in the Begin() that we will see shortly. These are the same for every single Draw call within the batch. Additionally you should try to keep every single Draw call within the batch in the same texture, or within as few different textures as possible. Each different texture within a batch incurs a performance penalty. You can also call multiple Begin()/End() pairs in a single render pass, just be aware that the Begin() process is rather expensive and this can quickly hurt performance if you do it too many times. Don't worry though, there are ways to easily organize multiple sprites within a single texture. If by chance you actually want to perform each Draw call as it occurs you can instead run the sprite batch in immediate mode, although since XNA 4 (which MonoGame is based on), there is little reason to use Immediate mode, and the performance penalty is harsh.

One other major function of the SpriteBatch is handling blending, which is how overlapping sprites interact.
Sprite Blending

Up until now we've used a single sprite with no transparency, so that's been relatively simple. Let's instead look at an example that isn't entirely opaque.

Let's go ahead an add a transparent sprite to our content project. Myself I am going to use this one:

... I'm sorry, I simply couldn't resist the pun. The key part is that your sprite supports transparency, so if you draw it over itself you should see:
Now let's change our code to draw two sprites in XNA.

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

namespace Example2
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D texture;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            graphics.PreferredBackBufferWidth = 400;
            graphics.PreferredBackBufferHeight = 400;
            Content.RootDirectory = "Content";
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
            texture = this.Content.Load<Texture2D>("transparentSprite");
        }

        protected override void Update(GameTime gameTime)
        {
            {
                Exit();
            }
            base.Update(gameTime);
        }

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

            spriteBatch.Begin();
            spriteBatch.Draw(texture, Vector2.Zero);
            spriteBatch.Draw(texture, new Vector2(100, 0));
        }
    }
}
```
spriteBatch.End();

base.Draw(gameTime);
}
}
}

... and run:

Pretty cool.

This example worked right out of the box for a couple reasons. First, our sprite was transparent and identical, so draw order didn't matter. Also when we ran the content pipeline, the default importer (and the default sprite batch blend mode) is transparency friendly.
This setting creates a special transparency channel for your image upon import, which is used by the SpriteBatch when calculating transparency between images.

Let's look at a less trivial example, with a transparent and opaque image instead.

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

namespace Example2
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Texture2D texture;
       Texture2D texture2;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            graphics.PreferredBackBufferWidth = 400;
            graphics.PreferredBackBufferHeight = 400;
            Content.RootDirectory = "Content";
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
            texture = this.Content.Load<Texture2D>("logo");
```
texture2 = this.Content.Load<Texture2D>("transparentSprite");

protected override void Update(GameTime gameTime)
{
        Exit();
    base.Update(gameTime);
}

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

    spriteBatch.Begin();
    spriteBatch.Draw(texture, Vector2.Zero);
    spriteBatch.Draw(texture2, Vector2.Zero);
    spriteBatch.End();

    base.Draw(gameTime);
}
So far, so good. Now let's mix up the draw order a bit...

spriteBatch.Begin();
spriteBatch.Draw(texture2, Vector2.Zero);
spriteBatch.Draw(texture, Vector2.Zero);
spriteBatch.End();

... and run:
Oh...

As you can see, the order we make Draw calls is by default, the order the sprites are drawn. As in the second Draw() call will draw over the results of the first Draw() call and so on.

There is a way to explicitly set the drawing order:

```csharp
spriteBatch.Begin(sortMode: SpriteSortMode.FrontToBack);
spriteBatch.Draw(texture2, Vector2.Zero, layerDepth: 0.0f);
spriteBatch.Draw(texture, Vector2.Zero, layerDepth: 1.0f);
spriteBatch.End();
```

Here you are setting the SpriteBatch sort order to be front to back, then manually setting the draw layer in each draw call. If you are guessing, there is also a BackToFront setting. SpriteSortMode is also what determines if drawing is immediate (SpriteSortMode.Immediate) or deferred (SpriteSortMode.Differed).

**Blend States**

We mentioned earlier that textures imported using the Content Pipeline by default has a special pre-calculated transparency channel created. This corresponds with SpriteBatches default BlendState, AlphaBlend. This uses the magic value created by the pipeline to determine how overlapping transparent
sprites are renderers. If you don’t have a really good reason otherwise, and are using the Content Pipeline to import your textures, you should stick to the default. I should point out, this behavior only became the default in XNA4, so older tutorials may have much different behavior.

The old default used to be interpolative blending, which used the RGBA values of the texture to determine transparency. This could lead to some strange rendering artifacts (discussed here: https://en.wikipedia.org/wiki/Alpha_compositing). The advantage is, all you need to blend images is an alpha channel, there was no requirement to create a special pre-multiplied channel. This means you didn’t have to run these images through the content pipeline. If you wish to do things the "old" way, when importing your assets (if not simply loaded directly from file) select false for PreMultiplied alpha in the Texture Importer Processor settings of the Content Pipeline. Then in your SpriteBatch, do the following:

```csharp
spriteBatch.Begin(blendState:BlendState.NonPremultiplied);
```

There are additional BlendState options including Additive (colors are simply added together) and Opaque (subsequent draw calls simply overwrite the earlier calls). You can have a great deal of control over the BlendState, but most projects simply will not require it. One other thing I ignored is Chromekeying. This is another option for supporting transparency basically you dedicate a single color to be transparent, then specify that color in the Content Pipeline. Essentially you are forming a 1bit alpha channel and are essentially "green screening" like in movies. Obviously you cannot use the color in your image however. In exchange for ugly source sprites and extra labor, you save in file size as you don’t need to encode the alpha channel.

There is some additional functionality built into SpriteBatch, including texture sampling, stencil buffers, matrix transforms and even special effects. These are well beyond the basics though, so we will have to cover them at a later stage.
Chapter Six

Audio

In this chapter we are going to look at using audio in XNA. Originally XNA supported one way of playing audio, using XACT (Cross Platform Audio Creation Tool). Since the initial release they added a much simplified API. We will be taking a look at both processes.

There is an HD video of this chapter [available here](#).

When playing audio there is always the challenge of what formats are supported, especially when you are dealing with multiple different platforms, all of which have different requirements. Fortunately the content pipeline takes care of a great deal of the complications for us. Simply add your audio files (mp3, mp4, wma, wav, ogg) to the content pipeline and it will do the rest of the work for you. As you will see shortly though, it is also possible to load audio files outside of the content pipeline. In this situation, be aware that certain platforms do not support certain formats (for example, no wma support on Android or iOS, while iOS doesn't support ogg but does support mp3). Unless you have a good reason, I would recommend you stick to the content pipeline for audio whenever possible.

The Perils of MP3

Although MP3 is supported by MonoGame, you probably want to stay away from using it. Why? Patents. If your game has over 5,000 users you could be legally required to purchase a license. From a legal perspective, Ogg Vorbis is superior in every single way. Unfortunately Ogg support is not as ubiquitous as we'd like it to be.
Adding Audio Content using the Content Pipeline

This process is virtually identical to adding a graphic file in your content file.

Simply add the content like you did using right click->Add Existing Items or the Edit menu:

If it is a supported format you will see the Processor field is filled (otherwise it will display Unknown). The
only option here is to configure the mp3 audio quality, a trade off between size and fidelity.

Playing a Song

Now let's look at the code involved in playing the song we just added to our game.

// This example shows playing a song using the simplified audio api

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace Example1
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        Song song;

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

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

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

            this.song = Content.Load<Song>("prepare");
            MediaPlayer.Play(song);
            // Uncomment the following line will also loop the song
            // MediaPlayer.IsRepeating = true;
            MediaPlayer.MediaStateChanged += MediaPlayer_MediaStateChanged;
        }
    }
}
Notice that we added the using statement Microsoft.Xna.Framework.Media. We depend on this for the MediaPlayer and Song classes. Our Song is loaded using the ContentManager just like we did earlier with Texture, this time with the type Song. Once again the content loader does not use the file's extension. Our Song can then be played with a call to MediaPlayer.Play(). In this example we wire up a MediaStateChanged event handler that will be called when the song completes, decreasing the volume and playing the song again.
Playing Sound Effects

This example shows playing sound effects. Unlike a Song, SoundEffects are designed to support multiple instances being played at once. Let's take a look at playing SoundEffect in MonoGame:

```
// Example showing playing sound effects using the simplified audio api
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Audio;
using System.Collections.Generic;

namespace Example2
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        List<SoundEffect> soundEffects;

        public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
            soundEffects = new List<SoundEffect>();
        }

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

        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);
            soundEffects.Add(Content.Load<SoundEffect>("airlockclose"));
        }
    }
}
```
soundEffects.Add(Content.Load<SoundEffect>("ak47");
soundEffects.Add(Content.Load<SoundEffect>("icecream");
soundEffects.Add(Content.Load<SoundEffect>("sneeze");

// Fire and forget play
soundEffects[0].Play();

// Play that can be manipulated after the fact
var instance = soundEffects[0].CreateInstance();
instance.IsLooped = true;
instance.Play();

protected override void Update(GameTime gameTime)
{
        Exit();

    if (Keyboard.GetState().IsKeyDown(Keys.D1))
        soundEffects[0].CreateInstance().Play();
    if (Keyboard.GetState().IsKeyDown(Keys.D2))
        soundEffects[1].CreateInstance().Play();
    if (Keyboard.GetState().IsKeyDown(Keys.D3))
        soundEffects[2].CreateInstance().Play();
    if (Keyboard.GetState().IsKeyDown(Keys.D4))
        soundEffects[3].CreateInstance().Play();

    if (Keyboard.GetState().IsKeyDown(Keys.Space))
    {
        if (SoundEffect.MasterVolume == 0.0f)
            SoundEffect.MasterVolume = 1.0f;
        else
            SoundEffect.MasterVolume = 0.0f;
    }
    base.Update(gameTime);
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    base.Draw(gameTime);
}

Note the using Microsoft.Xna.Framework.Audio statement at the beginning. Once again we added our audio files using the Content Pipeline, in this case I added several WAV files. They are loaded using Content.Load() this time with the type SoundEffect. Next it is important to note the two different ways the SoundEffects are played. You can either call Play() directly on the SoundEffect class. This creates a fire and forget instance of the class with minimal options for controlling it. If you have need for greater control (such as changing the volume, looping or applying effects) you should instead create a SoundEffectInstance using the SoundEffect.CreateInstance() call. You should also create a separate instance if you want to have multiple concurrent instances of the same sound effect playing. It is important to realize that all instances of the same SoundEffect share resources, so memory will not increase massively for each instance created. The number of simultaneous supported sounds varies from platform to platform, with 64 being the limit on Windows Phone 8, while the Xbox 360 limits it to 300 instances. There is no hard limit on the PC, although you will obviously hit device limitations quickly enough.

In the above example, we create a single looping sound effect right away. Then each frame we check to see if the user presses 1, 2, 3 or 4 and play an instance of the corresponding sound effect. If the user hits the spacebar we either mute or set to full volume the global MasterVolume of the SoundEffect class. This will affect all playing sound effects.

**Positional Audio Playback**

Sound effects can also be positioned in 3D space easily in XNA.

// Display positional audio

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Audio;

namespace Example3
{

}
public class Game1 : Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;
    SoundEffect soundEffect;
    SoundEffectInstance instance;
    AudioListener listener;
    AudioEmitter emitter;

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

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

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
        soundEffect = this.Content.Load<SoundEffect>("circus");
        instance = soundEffect.CreateInstance();
        instance.IsLooped = true;

        listener = new AudioListener();
        emitter = new AudioEmitter();

        // WARNING!!!! Apply3D requires sound effect be Mono!
        // Stereo will throw exception
        instance.Apply3D(listener, emitter);
        instance.Play();
    }

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back ==
            ButtonState.Pressed || Keyboard.GetState().IsKeyDown(}
if (Keyboard.GetState().IsKeyDown(Keys.Left))
{
    instance.Apply3D(listener, emitter);
}
if (Keyboard.GetState().IsKeyDown(Keys.Right))
{
    instance.Apply3D(listener, emitter);
}
if (Keyboard.GetState().IsKeyDown(Keys.Up))
{
    instance.Apply3D(listener, emitter);
}
if (Keyboard.GetState().IsKeyDown(Keys.Down))
{
    instance.Apply3D(listener, emitter);
}
base.Update(gameTime);

protected override void Draw(GameTime gameTime)
{
   GraphicsDevice.Clear(Color.CornflowerBlue);
    base.Draw(gameTime);
}
In this example, we load a single SoundEffect and start it looping infinitely. We then create an 
AudioListener and AudioEmitter instance. The AudioListener represents the location of your ear within the 
virtual world, while the AudioEmitter represents the position of the sound effect. The default location of 
both is a Vector3 at (0,0,0). You set the position of a SoundEffect by calling Apply3D(). In our Update() 
call, if the user hits an arrow key we updated the Position of the AudioListener accordingly. After changing 
the position of a sound you have to call Apply3D again. As you hit the arrow keys you will notice the audio 
pans and changes volume to correspond with the updated position. It is very important that your source 
audio file is in Mono (as opposed to Stereo) format if you use Apply3D, or an exception will be thrown.

Using XACT

As mentioned earlier, XACT used to be the only option when it came to audio programming in XNA. 
XACT is still available and it enables your audio designer to have advanced control over the music and 
sound effects that appear in your game, while the programmer uses a simple programmatic interface. One 
big caveat is XACT is part of the XNA installer or part of the Direct X SDK as is not available on Mac OS 
or Linux. If you wish to install it but do not have an old version of Visual Studio installed, instructions can 
be found here (http://www.gamefromscratch.com/post/2015/07/23/Installing-XNA-Tools-Like-XACT-
without-Visual-Studio-2010.aspx). If you are on MacOS or Linux, you want to stick to the simplified audio 
API that we demonstrated earlier.

Xact is installed as part of the XNA Studio install, on 64bit Windows by default the Xact executable will be 
located in C:\Program Files (x86)\Microsoft XNA\XNA Game Studio\v4.0\Tools. Start by running 
AudConsole3.exe:
The XACT Auditioning Tool needs to be running when you run the Xact tool.
Then launch Xact3.exe
First create a new project:

Next right click Wave Banks and select New Wave Bank

Drag and drop your source audio files into the Wave Bank window:
Now create a new Sound Bank by right clicking Sound Bank and selecting New Wave Bank.

Now drag the Wave you wish to use from the Wave Bank to the Sound Bank.

Now create a Cue by dragging and dropping the Sound Bank to the Cue window. Multiple files can be added to a cue if desired.
You can rename the Cue, set the probability to play if you set several Sounds in the Cue and change the instance properties of the Cue in the properties window to your left:
Now Build the results:

<table>
<thead>
<tr>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ak47</td>
<td>...</td>
</tr>
</tbody>
</table>

### Variation Playlist

- Playlists Type: Fandom (no immediate repeats)

### Variations

<table>
<thead>
<tr>
<th>Sound</th>
<th>Weight</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>ak47</td>
<td>255</td>
<td>100%</td>
</tr>
</tbody>
</table>

### Instance Limiting

- Limit Instances: 
- Maximum: 1
- Fade In (sec): 0.000
- Fade Out (sec): 0.000
- Crossfade Type:
  - Linear
  - Logarithmic
  - Equal Power

- Behavior At Max:
  - Fail To Play
  - Queue
  - Replace

- Replace Instance:
  - Lowest Priority
  - Oldest
  - Quietest

- Crossfader Type:
  - Linear
  - Logarithmic
  - Equal Power
This will then create two directories in the folder you created your project in:

These files need to be added directly to your project, **you do not use the content pipeline tool!** Simply copy all three files to the content folder and set it's build action to Copy.

Now let's look at the code required to use these generated files:

```csharp
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Audio;

namespace Example4
{
    public class Game1 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        AudioEngine audioEngine;
        SoundBank soundBank;
        WaveBank waveBank;

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

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

        protected override void LoadContent()
        {
        }
    }
}
```
First you create an AudioEngine using the xgs file, then a SoundBank using the xsb and a WaveBank unsing the xwb file. We then play the Cue we created earlier with a call to SoundBank.GetQue().Play(). This process allows the audio details to be configured outside of the game while the programmer simply uses the created Que.
Finally it is possible to play audio files that weren't added using the content pipeline or using Xact using a Uri.

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

    // URL MUST be relative in MonoGame
    System.Uri uri = new System.Uri("content/background.mp3",
                                       System.UriKind.Relative);
    Song song = Song.FromUri("mySong", uri);
    MediaPlayer.Play(song);
    MediaPlayer.ActiveSongChanged += (s, e) => {
        song.Dispose();
        System.Diagnostics.Debug.WriteLine("Song ended and disposed");
    };
}
```

First you create a Uri that locates the audio file you want to load. We then load it using the method FromUri, passing in a name as well as the uri. One very important thing to be aware of here, on XNA you could use any URI. In MonoGame it needs to be a relative path.
Chapter Seven

3D Graphics

In this chapter we start looking at 3D game development using MonoGame. Previously I called XNA a low level code focused engine and you are about to understand why. If you come from a higher level game engine like Unity or even LibGDX you are about to be in for a shock. Things you may take for granted in other engines/libraries, like cameras, are your responsibility in Monogame. Don't worry though, it's not all that difficult.

This information is also available in HD Video.

This chapter is going to require some prior math experience, such as an understanding of Matrix mathematics. Unfortunately teaching such concepts if far beyond the scope of what we can cover here without adding a few hundred more pages! If you need to brush up on the underlying math, the Khan Academy is a very good place to start. There are also a few books dedicated to teaching gamedev related math including 3D Math Primer for Graphics and Game Development and Mathematics for 3D Game Programming and Computer Graphics. Don't worry, Monogame/XNA provide the Matrix and Vector classes for you, but it's good to understand when to use them and why.

Our First 3D Application
This might be one of those topics that’s easier explained by seeing. So let’s jump right in with an example and follow it up with explanation. This example creates then displays a simple triangle about the origin, then creates a user controlled camera that can orbit and zoom in/out on said triangle.

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

namespace Test3D
{
    public class Test3DDemo : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        // Camera
        Vector3 camTarget;
        Vector3 camPosition;
        Matrix projectionMatrix;
        Matrix viewMatrix;
        Matrix worldMatrix;

        // BasicEffect for rendering
        BasicEffect basicEffect;

        // Geometric info
        VertexPositionColor[] triangleVertices;
        VertexBuffer vertexBuffer;

        // Orbit
        bool orbit = false;

        public Test3DDemo()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        protected override void Initialize()
        {
            base.Initialize();
        }
    }
}
```
//Setup Camera
float camTarget = new Vector3(0f, 0f, 0f);
float camPosition = new Vector3(0f, 0f, -100f);
float projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(45f),
GraphicsDevice.DisplayMode.AspectRatio, 1f, 1000f);
float viewMatrix = Matrix.CreateLookAt(camPosition, camTarget,
new Vector3(0f, 1f, 0f)); // Y up
float worldMatrix = Matrix.CreateWorld(camTarget, Vector3.Forward, Vector3.Up);

//BasicEffect
float basicEffect = new BasicEffect(GraphicsDevice);
basicEffect.Alpha = 1f;

// Want to see the colors of the vertices, this needs to be on
basicEffect.VertexColorEnabled = true;

// Lighting requires normal information which VertexPositionColor does not have
// If you want to use lighting and VPC you need to create a custom def
basicEffect.LightingEnabled = false;

// Geometry - a simple triangle about the origin
float triangleVertices = new VertexPositionColor[3];
triangleVertices[0] = new VertexPositionColor(new Vector3(0, 20, 0), Color.Red);
triangleVertices[1] = new VertexPositionColor(new Vector3(-20, -20, 0), Color.Green);
triangleVertices[2] = new VertexPositionColor(new Vector3(20, -20, 0), Color.Blue);

// Vert buffer
float vertexBuffer = new VertexBuffer(GraphicsDevice, typeof(VertexPositionColor), 3, BufferUsage.WriteOnly);
vertexBuffer.SetData<VertexPositionColor>(triangleVertices);
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
}

protected override void UnloadContent()
{
}

protected override void Update(GameTime gameTime)
{
        Exit();

    if (Keyboard.GetState().IsKeyDown(Keys.Left))
    {
        camPosition.X -= 1f;
        camTarget.X -= 1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Right))
    {
        camPosition.X += 1f;
        camTarget.X += 1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Up))
    {
        camPosition.Y -= 1f;
        camTarget.Y -= 1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Down))
    {
        camPosition.Y += 1f;
        camTarget.Y += 1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.OemPlus))
    {
        camPosition.Z += 1f;
    }
}
if (Keyboard.GetState().IsKeyDown(Keys.OemMinus))
{
    camPosition.Z -= 1f;
}
if (Keyboard.GetState().IsKeyDown(Keys.Space))
{
    orbit = !orbit;
}

if (orbit)
{
    Matrix rotationMatrix = Matrix.CreateRotationY(
        MathHelper.ToRadians(1f));
    camPosition = Vector3.Transform(camPosition,
        rotationMatrix);
}
viewMatrix = Matrix.CreateLookAt(camPosition, camTarget,
        Vector3.Up);
base.Update(gameTime);

protected override void Draw(GameTime gameTime)
{
    basicEffect.Projection = projectionMatrix;
    basicEffect.View = viewMatrix;
    basicEffect.World = worldMatrix;

    GraphicsDevice.Clear(Color.CornflowerBlue);
    GraphicsDevice.SetVertexBuffer(vertexBuffer);

    // Turn off culling so we see both sides of our rendered
    // triangle
    RasterizerState rasterizerState = new RasterizerState();
    rasterizerState.CullMode = CullMode.None;
    GraphicsDevice.RasterizerState = rasterizerState;

    foreach (EffectPass pass in basicEffect.CurrentTechnique.
        Passes)
    {
        pass.Apply();
        GraphicsDevice.DrawPrimitives(PrimitiveType.
            TriangleList, 0, 3);
Alright... that's a large code sample, but don't worry, it's not all that complicated. At a top level what we do here is create a triangle oriented about the origin. We then create a camera, offset –100 units along the z-axis but looking at the origin. We then respond to keyboard, panning the camera in response to the arrow keys, zooming in and out in response to the plus and minus key and toggling orbit using the space bar. Now let's take a look at how we accomplish all of this.

First, when I said we create a camera, that is a misnomer, in fact we are creating three different Matrices (singular – Matrix), the View, Projection and World matrix. These three matrices are combined to help position elements in your game world. Let's take a quick look at the function of each.

**View Matrix** The View Matrix is used to transform coordinates from World to View space. A much easier way to envision the View matrix is it represents the position and orientation of the camera. It is created by passing in the camera location, where the camera is pointing and by specifying which axis represents "Up" in the universe. XNA uses a Y-up orientation, which is important to be aware of when creating 3D models. Blender by default treats Z as the up/down axis, while 3D Studio MAX uses the Y-axis as "Up".

**Projection Matrix** The Projection Matrix is used to convert 3D view space to 2D. In a nutshell, this is your actual camera lens and is created by specifying calling `CreatePerspectiveFieldOfView()` or `CreateOrthographicFieldOfView()`. With Orthographic projection, the size of things remain the same regardless to their "depth" within the scene. For Perspective rendering it simulates the way an eye works, by rendering things smaller as they get further away. As a general rule, for a 2D game you use Orthographic, while in 3D you use Perspective projection. When creating a Perspective view we specify the field of view (think of this as the degrees of visibility from the center of your eye view), the aspect ratio (the proportions between width and height of the display), near and far plane (minimum and maximum depth to render with camera... basically the range of the camera). These values all go together to calculate something called the view frustum, which can be thought of as a pyramid in 3D space representing what is currently available.

**World Matrix** The World matrix is used to position your entity within the scene. Essentially this is your position in the 3D world. In addition to positional information, the World matrix can also represent an objects orientation.

So nutshell way to think of it:
Cross Platform Game Development using MonoGame

**View Matrix** → Camera Location  
**Projection Matrix** → Camera Lens  
**World Matrix** → Object Position/Orientation in 3D Scene

By multiplying these three Matrices together we get the WorldProjView matrix, or a magic calculation that can turn a 3D object into pixels.

**What value should I use for Field of View?**

You may notice in this example I used a relatively small value of 45 degrees in this example. What you may ask is the ideal setting for field of view? Well, there isn't one, although there are some commonly accepted values. Human beings generally have a field of view of about 180 degrees, but this includes peripheral vision. This means if you hold your hands straight out you should be able to just see them out of the edge of your vision. Basically if its in front of you, you can see it.

However video games, at least not taking into account VR headset games, don't really use the peripherals of your visual space. Console games generally set of a field of view of about 60 degrees, while PC games often set the field of view higher, in the 80-100 degree range. The difference is generally due to the size of the screen viewed and the distance from it. The higher the field of view, the more of the scene that will be rendered on screen.

Next up we have the BasicEffect. Remember how earlier we used a SpriteBatch to draw sprites on screen? Well the BasicEffect is the 3D equivalent. In reality it's a wrapper over a HLSL shader responsible for rendering things to the screen. Now HLSL coverage is way beyond the scope of what we can cover here, but basically it's the instructions to the shader units on your graphic card telling how to render things. Although I can't go into a lot of details about how HLSL work, you are in luck, as Microsoft actually released the shader code used to create BasicEffect in the Stock Effect sample available at [http://xbox.create.msdn.com/en-US/education/catalog/sample/stock_effects](http://xbox.create.msdn.com/en-US/education/catalog/sample/stock_effects). In order for BasicEffect to work it needs the View, Projection and Matrix matrixes specified, thankfully we just calculated all three of these.

Finally at the end of Initialize() we create an array of VertexPositionColor, which you can guess is a Vertex with positional and color data. We then copy the triangle data to a VertexBuffer using a call to SetData(). You may be thinking to yourself... WOW, doesn't XNA have simple primitives like this built in? No, it doesn't, although there are easy community examples you can download such as this one: [http://xbox.create.msdn.com/en-US/education/catalog/sample/primitives_3d](http://xbox.create.msdn.com/en-US/education/catalog/sample/primitives_3d).

The logic in Update() is quite simple. We check for input from the user and respond accordingly. In the event of arrow keys being pressed, or +/- keys, we change the cameraPosition. At the end of the update we then recalculate the View matrix using our new Camera position. Also in response to the space bar, we
toggle orbiting the camera and if we are orbiting, we rotate the camera by another 1 degree relative to the origin. Basically this shows how easy it is to update the camera by changing the viewMatrix. Note the Projection Matrix generally isn't updated after creation, unless the resolution changes.

Finally we come to our Draw() call. Here we set the view, projection and world matrix of the BasicEffect, clear the screen, load our VertexBuffer into the GraphicsDevice calling SetVertexBuffer(). Next we create a RasterState object and set culling off. We do this so we don't cull back faces, which would result in our triangle from being invisible when we rotate behind it. Often you actually want to cull back faces, no sense drawing vertices that aren't visible! Finally we load through each of the Techniques in the BasicEffect ( look at the BasicEffect.fx HLSL file and this will make a great deal more sense. Otherwise stay tuned for when we cover custom shaders later on ), finally we draw our triangle data to screen by calling DrawPrimitives, in this case it's a TriangleList. There are other options such as lines and triangle strips, you are basically telling it what kind of data is in the VertexBuffer.

I'll admit, compared to many other engines, that's a heck of a lot of code to just draw a triangle on screen! Reality is though, you generally write this code once and that's it. Or you work at a higher level, such as with 3D models imported using the content pipeline.

### Loading and Displaying 3D Models

Next we take a look at the process of bringing a 3D model in from a 3D application, in this case Blender. The process of creating such a model is well beyond the scope of this tutorial, although I have created a video showing the entire process available right here. Or you can simply download the created COLLADA file and texture.

### Which File Format works Best?

The MonoGame pipeline tool relies on an underlying library named Assimp for loading 3D models. You may wonder which of the many model formats supported should you use if exporting from Blender? FBX and COLLADA(dae) are the two most commonly used formats, while X and OBJ can often be used reliably with very simple non-animated meshes. That said, exporting from Blender is always a tricky prospect, and it's a very good idea to use a viewer like the one included in the FBX Converter package to verify your exported model looks correct.

The above video also illustrates adding the model and texture using the content pipeline. I won't cover the process here as it works identically to when we used the content pipeline earlier. Let's jump right in to the code instead:

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

namespace Test3D
{

    public class Test3DDemo2 : Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        //Camera
        Vector3 camTarget;
        Vector3 camPosition;
        Matrix projectionMatrix;
        Matrix viewMatrix;
        Matrix worldMatrix;

        //Geometric info
        Model model;

        //Orbit
        bool orbit = false;

        public Test3DDemo2()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

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

            //Setup Camera
            camTarget = new Vector3(0f, 0f, 0f);
            camPosition = new Vector3(0f, 0f, -5);
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(
                MathHelper.ToRadians(45f), graphics.GraphicsDevice.Viewport.AspectRatio, 1f, 1000f);
            viewMatrix = Matrix.CreateLookAt(camPosition, camTarget, CamTarget, CamPosition, MousePosition);
new Vector3(0f, 1f, 0f)); // Y up
worldMatrix = Matrix.CreateWorld(camTarget, Vector3.Forward, Vector3.Up);

model = Content.Load<Model>("MonoCube");
}

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

protected override void UnloadContent()
{
}

protected override void Update(GameTime gameTime)
{
        Exit();

    if (Keyboard.GetState().IsKeyDown(Keys.Left))
    {
        camPosition.X -= 0.1f;
        camTarget.X -= 0.1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Right))
    {
        camPosition.X += 0.1f;
        camTarget.X += 0.1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Up))
    {
        camPosition.Y -= 0.1f;
        camTarget.Y -= 0.1f;
    }
    if (Keyboard.GetState().IsKeyDown(Keys.Down))
    {
        camPosition.Y += 0.1f;
        camTarget.Y += 0.1f;
} if (Keyboard.GetState().IsKeyDown(Keys.OemPlus))
{
    camPosition.Z += 0.1f;
}
if (Keyboard.GetState().IsKeyDown(Keys.OemMinus))
{
    camPosition.Z -= 0.1f;
}
if (Keyboard.GetState().IsKeyDown(Keys.Space))
{
    orbit = !orbit;
}

if (orbit)
{
    Matrix rotationMatrix = Matrix.CreateRotationY(
        MathHelper.ToRadians(1f));
    camPosition = Vector3.Transform(camPosition,
        rotationMatrix);
}
viewMatrix = Matrix.CreateLookAt(camPosition, camTarget,
    Vector3.Up);
base.Update(gameTime);
}

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

    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            //effect.EnableDefaultLighting();
            effect.AmbientLightColor = new Vector3(1f, 0, 0);
            effect.View = viewMatrix;
            effect.World = worldMatrix;
            effect.Projection = projectionMatrix;
        }
        mesh.Draw();
    }
base.Draw(gameTime);

It operates almost identically to when we created the triangle by hand, except that model is loaded using a call to Content.Load<Model>(). The other major difference is you no longer have to create a BasicEffect, one is automatically created for you as part of the import process and is stored in the Mesh’s Effects property. Simply loop through each effect, setting up the View, Projection and World matrix values, then call Draw(). If you have a custom effect you wish to use instead of the generated Effects, you can follow the process documented here: https://msdn.microsoft.com/en-us/library/bb975391(v=xnagamestudio.31).aspx.