XNA Development: Tutorial 2

By Matthew Christian (Matt@InsideGamer.org)

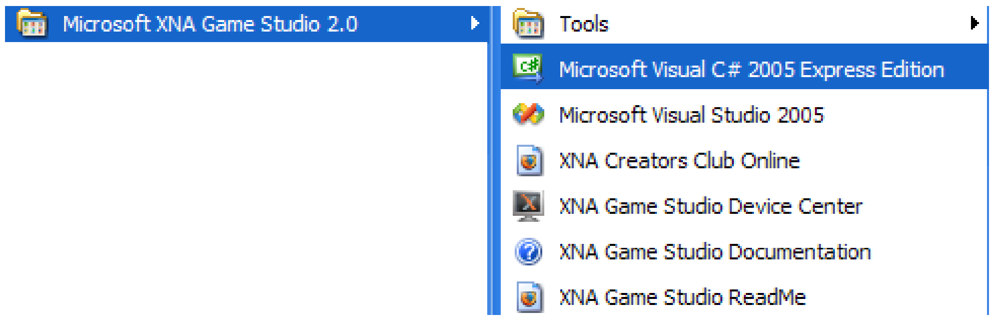
Code and Other Tutorials Found at http://www.insidegamer.org/xnatutorials.aspx

We've got Visual C# installed, XNA Game Studio Express, and have connected our Xbox 360 to our PC using the generated key feature provided with the XNA Game Launcher we got after joining the XNA Creator's Club. Now we can begin to work on creating games and small 'tech demos' we can play on the Xbox 360 system. Let's start by creating a new project in XNA Game Studio Express and deploying it to the Xbox 360. From there, we can begin to create objects and content to display within our game.

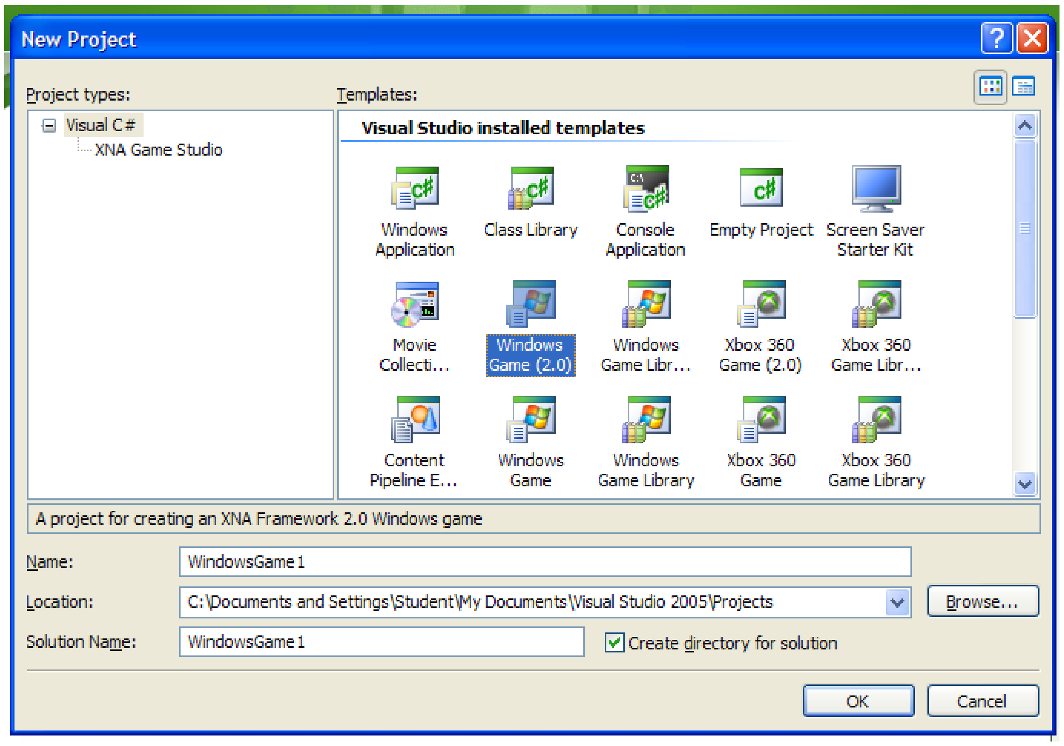
Note: If you don't have the above mentioned installed, view XNA Development: Tutorial 1 which you can find at the tutorial link at the top of this tutorial. I'm also assuming you have a bit of knowledge with programming but I go into enough detail so you can understand the basic idea of the project and its flow. This tutorial will follow creating a new project of the Windows Game (2.0) type which is included in the XNA 2.0 release of XNA Game Studio Express.

Starting a New Project in Microsoft XNA Game Studio Express

To start out, open Microsoft XNA Game Studio Express, which (assuming you've installed shortcuts to your Start menu) can be found by clicking Start, All Programs, Microsoft XNA Game Studio 2.0, and Microsoft Visual C# 2005 Express Edition.


Microsoft XNA Game Studio 2.0
Tools
Microsoft Visual C# 2005 Express Edition
Microsoft Visual Studio 2005
XNA Creators Club Online
XNA Game Studio Device Center
XNA Game Studio Documentation
XNA Game Studio ReadMe

Visual C# Express Edition should load up and bring you to the default Start Page. Since we're going to start a new project go to the menu bar across the top and click File, New Project… A dialog will open displaying templates you can use to base your projects on (Figure 1). If you wanted to work with the Xbox 360, you'd use the 'Xbox 360 Game (2.0)' template under 'Visual Studio installed templates'. Since it's easier to compile and work with a Windows Project, select the 'Windows Game (2.0)' installed template. You can change the project name, the location the project is stored at, and the name of the solution at this point if necessary. Also, there is a checkbox to create a new folder for the project which is typically used and is the default. Once you have these properties set (leaving them as their defaults is fine), click the OK button and Visual Studio will create your project files and load the project into the development environment.



A project for creating an XNA Framework 2.0 Windows game

Name: WindowsGame1

Location: C:\Documents and Settings\Student\My Documents\Visual Studio 2005\Projects

Solution Name: WindowsGame1

[✓] Create directory for solution

Figure 1 - New Project Dialog

After loading, the environment should open to the Game1.cs file. Game1.cs is the primary, default C# page that will contain your program code. Let's look at the other files contained in our new Xbox 360 game project. Locate the Solution Explorer pane; it should be located on the left or right side of your development environment. If you don't see the pane, go to the main menu and click View, and Solution Explorer and it should appear. This window contains all the files related to your project. During the creation of a new Windows Game project, Visual C# creates a solution, a project under that solution, 2 folders named Properties and References, 2 C# pages titled Game1.cs and Program.cs, and an image named GameThumbnail.png.

Understanding the Project Structure

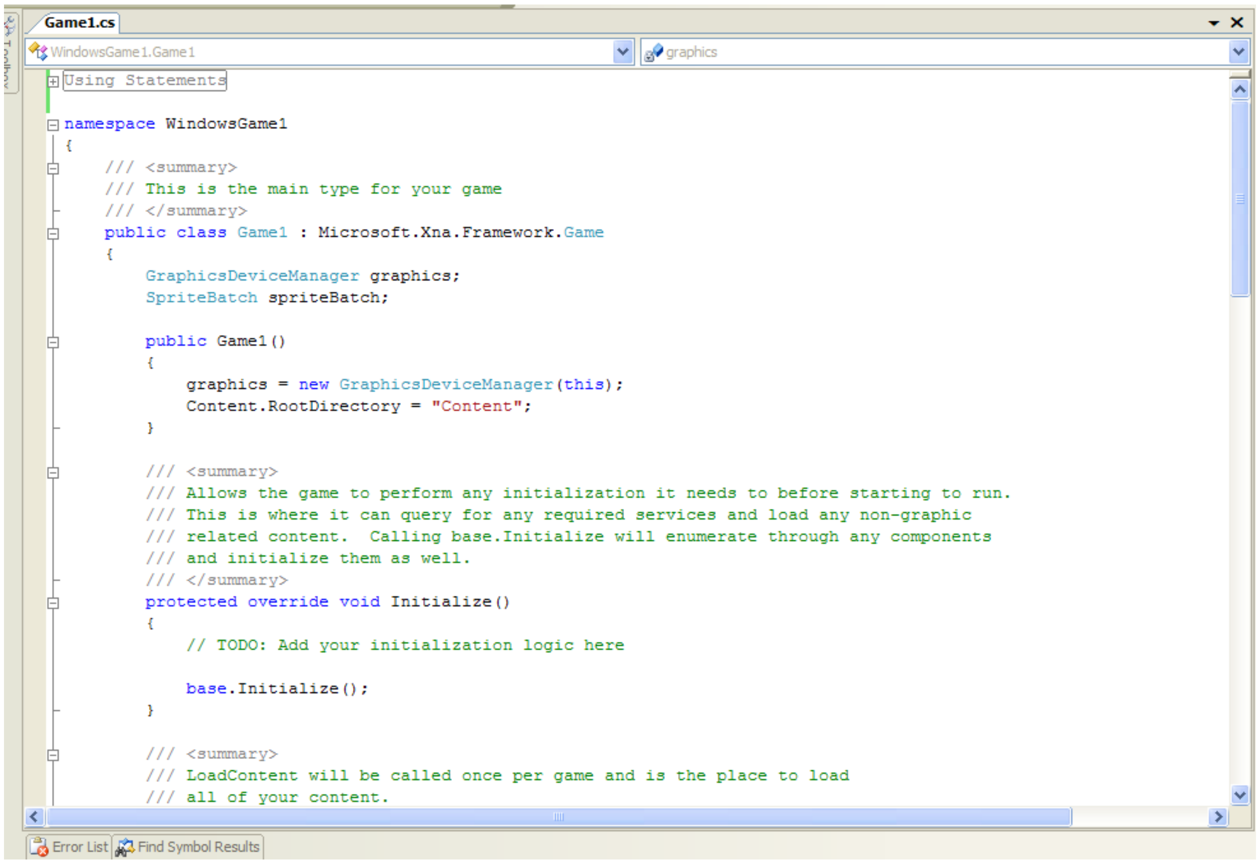
Note: Write down (or remember) whatever your Solution Explorer has named your solution and project file (default is WindowsGame1 for both). We will be changing these values to understand the folder structure and will need to change them back.

Typically your solution file and project file will generally stay the same while content will often be added and deleted from within. From the Solution Explorer, you can rename files and folders as well as open them in Visual Studio. If the file is not supported by Visual Studio (for example, our GameThumbnail.png), Visual Studio will automatically detect the appropriate program and open that file using that program. Try right click on the bolded project name (default is WindowsGame1) and click Rename towards the bottom of the drop down list. Type a new project name, something like 'MyNewXNAGame'. Notice the solution name does not change and still reads Solution 'WindowsGame1' (1 project). For a better understanding of what the difference is, navigate to wherever you created your XNA project. If left defaulted, you can get there by clicking Start, My Documents, and open the XNA folder. The subfolder within is titled WindowsGame1, it hasn't changed even if you've changed the project name. Open this folder; you'll notice 2 files and a folder (WindowsGame1.sln, WindowsGame1.suo, and WindowsGame1) which all haven't changed their name either.

Navigating into the WindowsGame1folder within will open up revealing plenty of folders and files, including those mentioned within your project. Also notice the MyNewXNAGame.cs project file, this is what we changed. This folder is where all your game content will be stored including your project files. Click the up folder button to go back to the WindowsGame1 directory. Go back to Visual C# Express and click on the Solution 'WindowsGame1' (1 project) entry in the Solution Explorer. We're going to change this and see what happens in the folder. There are two ways to edit the name of the solution, click on the item, wait for a second, and then click on it again, or open the Properties window and change the name property (you can open the Properties window by going to the main menu and selecting View, Properties Window). Change the name to match your project file (MyNewXNAGame). Return to the XNA folder and notice the subfolder is still titled WindowsGame1. Open this folder and notice the two files within have changed but the folder hasn't. We've changed the solution name and the project name but our folder structure hasn't changed.

Note: Change the solution and project file names back to their initial values (WindowsGame1). At the conclusion you'll see why we're doing this.

Next, we'll change the Game1.cs file and see what happens within the code. If it isn't opened in the environment already, double click on the Game1.cs file in the Solution Explorer. You should see a window like the figure below (Figure 2). The item labeled 'Using Statements' will be different in your solution at this point.



```
Using Statements

namespace WindowsGame1
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

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

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

            base.Initialize();
        }

        /// <summary>
        /// LoadContent will be called once per game and is the place to load
        /// all of your content.
```

Figure 2 - Game1.cs

We'll go through some of the code later but for now, notice the lines

```
public class Game1 : Microsoft.Xna.Framework.Game
  ...
```

```
        public Game1()
```

Now, change the name of Game1 (click, wait, click a second time) to something and you should see a message pop up, click Yes! Look at those lines again, they've changed! In fact, every file in your project that uses that file has changed to represent what the new name is. You're changing much more than the file though, you're changing the class name within the file (something we'll go into deeper much later). Note, if you wanted to change it and accidentally clicked you, you would have to go into the file you've renamed and change it as well as the Program.cs and change it (there are 2 spots in each file that need to be changed).

Note: Change the file back to Game1.cs and click Yes again. While we aren't working with a brand new project, we want it to look like it's a new project by being as close to what it was when it was created as possible until the next section.

This small exercise is simply a type of forewarning into creating projects. Visual Studio has done a great job in that, if you change the name of the file, the code relating to that will change as well but that only reaches to a certain extent. The bottom line with projects is that you want to name them once (in the New Project dialog) and leave them named that. Sure, you can go ahead and change the solution name and the project name, but your folders won't change (without forcefully changing the path name in numerous places) and eventually will begin to look messy with people looking for project files in folders that are named differently.

New Project Program Flow

We've talked a bit about Game1.cs but very little about Program.cs which is an incredibly important part of the project. In fact, Project.cs contains the biggest most important part of a game; the main() function. As all programmers should know, main() is the first function your program calls which in turn starts the program up.

Open up Program.cs. The first thing you may notice is how short it is, it's only about 18 lines (including comments). Within there's a basic namespace defaulted to the name of the project and a class named Program which includes a single method,

```
        static void Main(string[] args)
        {
            using (Game1 game = new Game1())
            {
                game.Run();
            }
        }
```

Basically, this method creates an object for us to use based on the type Game1 and invokes the Run() method of the Game1 object. Game1 is a class created when we made the new project and is contained in Game1.cs (big surprise!). It's obvious why this is the most important method of all, without this the system wouldn't know what to start with.

Note: I'll be using function and method interchangibly. Usually I'll use the word 'function' when it's more of a global function (i.e. something defined by Windows or one of it's libraries like Math) while I'll use 'method' as more of a local function (i.e. something we've written or something generated at creation time that isn't necessarily stored if this project wasn't ever created).

Understanding the Game Class

Our system knows it wants to run an object based on the Game1 class so let's take a peek at what is in included in that class. Open the Game1.cs file. Game1.cs is much larger than Program.cs but includes many more comments, without the comments the file really isn't too big at all. Just like the Program.cs, we have a namespace (notice the namespace is named the same as with Program.cs and is the same as the project name), a class, and some methods within that class. The biggest difference with the Game1 class, is that it's derived from the Microsoft.XNA.Framework.Game class. This class provided by Microsoft really helps simplify the code by initializing the graphics device and knowing when to draw to the screen.

The class starts by declaring two class-wide variables,

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
```

The graphics variable manages the graphics device (hence GraphicsDeviceManager) and the spriteBatch variable manages 2D objects loaded through the content pipeline (more on this in Tutorial 5).

Next we find the constructor for our Game1 class which intializes our graphics variable and sets the directory our game content will be stored in. The GraphicsDeviceManager constructor takes one argument which is the handle for your game (or what the device manager can reference as what called it) and is defaulted to 'this' (reference to an instance of our current class; basically a reference to our instance of Game1). Until XNA 2.0, the ContentManager was built with a new project. Now, it is integrated right into the project directly to provide better usability for anyone wanting to make their own content processing class. It contains a property called RootDirectory which is set to "Content", a subfolder type created to store where the content is stored in our project.

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

From here on out all the methods of our Game1 class are pretty well documented and should give you a general idea on what each one does. One item you may notice is being called over and over is 'base' and each is being defined as an 'override'. Base is simply refering to the class our Game1 class is derived from and the methods contained within it. I mentioned earlier the Microsoft.XNA.Framework.Game has all the initializing and drawing we need, and it does but we need to call those methods. You'll see this is done throughout using base.methodName() (where methodName is whichever method you want to call which is contained in the Game class). Override, is our way of telling the Game class we want to use what's defined in the base class but want to specify our own methods for certain methods. For example, they have a prebuilt Initialize() method, but we want to make our own and customize it for our own game so we override the Initialize method in Game1.

There are two final lines I'd like to point out. The first is in the Update method and looks like the following:

```
// Allows the game to exit
if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
    this.Exit();
```

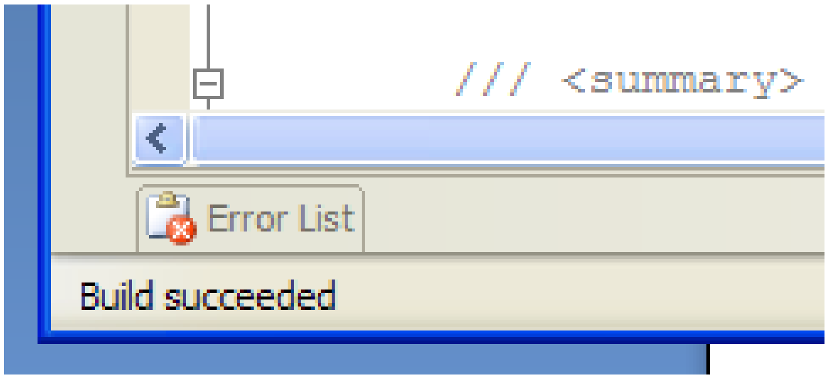
This is pretty self explanatory because of the included comment but if you aren't sure, it checks each time this update is called to see if the Back button is pressed on the Xbox 360 controller on the first player controller. If so, the game exits. The other line is found in our Draw method and resembles:

```
graphics.GraphicsDevice.Clear(Color.CornflowerBlue);
```

Here we use our newly made graphics variable to clear the screen and make it a specified color (default is CornflowerBlue).

Testing Our Game

Our code is looking good so it's time to run it. Before we can do this though, we need to compile the code and build the project. You can do this by clicking Build in the main menu, followed by Build Solution, or you can simply press CTRL+SHIFT+B. If all goes well, you should see 'Build Succeeded' in the lower right hand corner of the development environment. If not your error list should open up and show you what is wrong. I won't go into detail about debugging (hopefully you've programmed a bit and know how to debug errors well enough).



Once you've built your solution you can deploy it. Using an Xbox 360, deployment is slightly different. For Xbox users, go to your Xbox 360, load the XNA Game Launcher and select the Connect to Computer option. You should see a screen while the Xbox 360 searches for a connection to your computer. Remember, make sure you've set up your Xbox 360 to work with your compiler and are on the same network. For both Xbox Games and Windows Games, you're able to run the game from Visual C# by pressing F5 or clicking the green, 'Start Debugging' arrow (Figure 3).

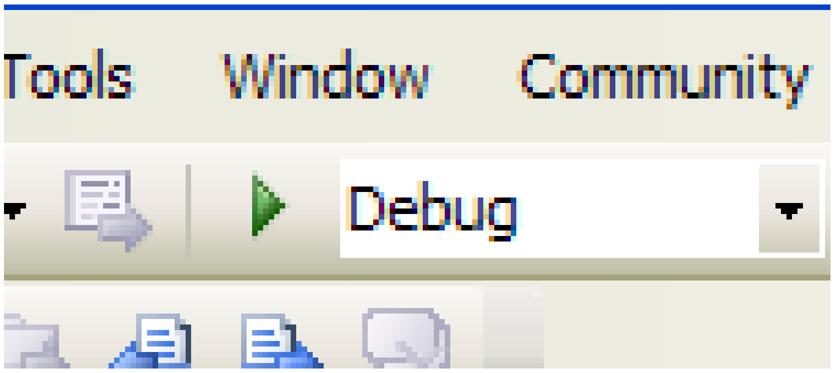


Figure 3 - Start Debugging Button

Your TV/Monitor will display a solid blue color (CornflowerBlue to be exact).   We've successfully built and deployed our first ever application using XNA! To exit, return to Visual C# and press the 'Stop Debugging' button (Xbox users can press Back on the first player Xbox 360 controller to exit also).

My Games (Xbox 360 Users)

After you deploy your game on the Xbox 360, it is stored on the console itself. From now on, if you want to play this 'game', you won't need to connect to your PC. Simply go to the My Games section of the XNA Game Launcher and select your game. Notice the image is the same image found in our project named GameThumbnail.png. This comes at a price though. Every time you run a game on the Xbox it will store it on the harddrive and list it under My Games. You will need to delete the game from your harddrive to remove it from the console. The more you test and debug, the more you'll need to delete so you'll need to find a balance between running on the console and building with just knowing if it will work or not if you want to lower the number of times you have to delete useless debugs off your Xbox harddrive.

Conclusion

This tutorial seemed very simplistic in that all we did was start a new project and look at what was created. In order to program some simple graphics and whatnot, you need to know what your game is built on and that was our goal here. At this point, everything should be set up properly and working between XNA and your PC (and Xbox 360, optionally) and we can get right into programming our very first game.

Suggested Exercises

1) Change the program so that the color displayed on your TV/Monitor is something other than blue.
2) Try changing which controller button exits the game or try making more than one button exit the game.
3) If you have another controller, add code to support exiting the game using a button pressed on that controller.
4) Create a method in the Game1 class, clear the screen to green, and call the method in the Draw method.
5) Follow Exercise 4, checking the Xbox controller for an exit press and place this method in the Update method.