How to Develop a Zelda Fan Game for Microsoft Windows and XBOX 360

Tutorial Series by Kenneth Murray (aka Minalien)

DISCLAIMER

This is a tutorial series focusing on developers who are able to grasp at least the primary basics of the C# programming language, or can, at the very least, make some sense of it. This series covers development of the core game engine, as well as the more intricate portions, such as creating and debugging components such as the ever-popular "Ocarina Engine" systems, text engines, and everything that many fan gamers have come to love about Zelda fan games.

However, this tutorial series focuses on the development, primarily programming, aspects of producing a Legend of Zelda fan game, and, although there will be tidbits of information and advice to steer you in the right direction, this tutorial is not focusing on actually designing aspects such as supporting characters.

That being said, I welcome you to this tutorial series, and hope that you can make the best of it; I hope to see many Legend of Zelda fan games springing up that are being developed using XNA, rather than the usual suspects (Game Maker, RPG Maker, and the sort). Not that there's anything wrong with them; it's all in the preferences of the developers and the needs of the project.

Downloads

The following are downloads relevant to this portion of the series:

Visual C# 2005 Express Edition: http://visual-c-2005-express-edition-sp1.en.malavida.com/d2433-free-download-windows

Microsoft XNA Game Studio 2.0:
http://www.microsoft.com/downloads/details.aspx?FamilyId=DF80D533-BA87-40B4-ABE2-1EF12EA506B7&displaylang=en

Part I: Setting up the Project

This portion is going to cover setting up the project in Visual Studio. Although in my tutorials, I am going to use Visual Studio 2005 Professional on my end, using Microsoft Visual C# 2005 Express Edition will work in very much the same way; the only difference will be in some of the screenshots that accompany the tutorial series. Everything else works essentially the same way.

Before we get started, you need to ensure that you have downloaded, updated, and installed Microsoft Visual C # 2005 Express Edition (see downloads section; above), or one of the Microsoft Visual Studio 2005 full studio editions, such as Professional, and installed Microsoft XNA 2.0. The reason we are not using the 3.0 Beta, which is the most recent version released, is because it is not yet supported on the XBOX 360, which is one of the main focuses on XBOX 360, and the reason that we are developing using XNA. Just follow the on-screen instructions to set everything up.

Step I: Setting up the Project

Alright, now it's time to get started, setting up the project, messing with our settings, and getting things running. Open your IDE (Visual Studio or Visual C# Express). We're going to create a new project, so you need to either click "Create: Project" on the start page that appears when Visual Studio starts, or "File->New->Project". A dialog similar (though without the "Project Types" selection on the left, if you're using Visual C# Express; If you're using Visual Studio, you want to select "Visual C# -> XNA Game Studio 2.0") to Figure 1-1 will open. Select "Windows Game (2.0)", regardless of whether you want to build for the XBOX or the PC primarily. The reason for this is because we are going to ensure that our engine is compatible with both systems, so it will be a trivial task to create a project that will deploy to the XBOX 360 in our next tutorial. I'm going to call the project "Zelda", but you can call it whatever you like.

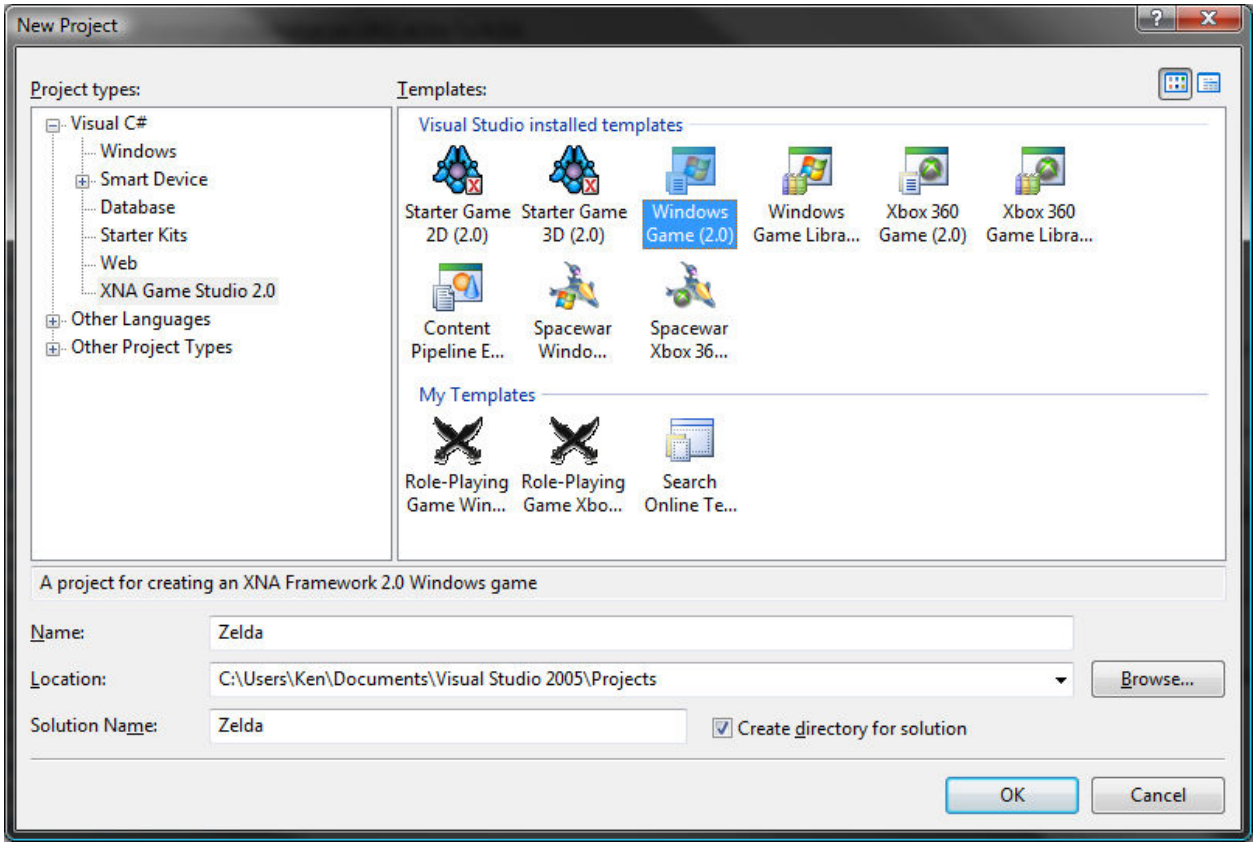


Figure 1-1: New Project Dialog

Step II: Changing our Settings

After everything looks good, and you press “OK”, your project files are generated, and we’re ready to change the settings and information about our game, and the file Game1.cs appears with the default content. Right now, if you run the program (by pressing F5, going to “Debug->Start Debugging”, or pressing the green Play button), a simple window should appear, cleared to a simple light blue color. It doesn’t look like much, but we’ve already got all that is needed to show a basic window that makes calls to the graphics system; we’re going to be ready to start programming soon.

We’re going to start by giving our Game1.cs file a better name; rename the file (right-click on it and select “Rename”, or select the file (in the Solution Explorer, on the right of your screen) and press F2) to Engine.cs. Click yes on the dialog (seen in Figure 1-2) that appears; this automatically changes all calls to “Game1” into calls to “Engine”, as well as renames our class defined in the file. This isn’t required, but it looks nicer.

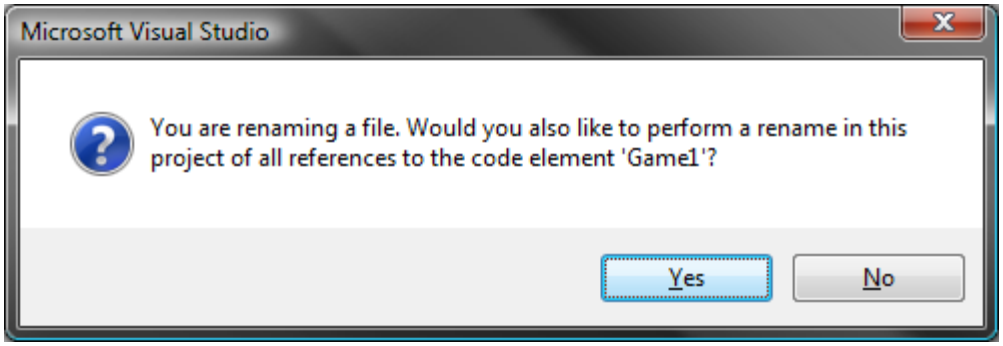


Figure 1-2: Content Replace Dialog

Now, we're going to change our settings; specifically, we're going to edit what our game gets called when we deploy to the XBOX 360, and when users right-click our executable and select click Properties. This information relates to the product name, copyright information, company name, etc. To open this information, double click on "Properties" under your project name in the solution explorer. When the properties window (see Figure 1-3) appears, click the "Assembly Information" button. In here, we can set the following settings:

Title: This sets the title for our project, as would be seen in the Games Library. Only the first twenty-five characters of this are shown on the XBOX 360, due to size limitations, so it's best to just keep it short and sweet.
Description: This is a brief, three-hundred-character description of your game (it can be more, but only the first three-hundred characters will be shown on the XBOX) that will be displayed in the game library.
Company: This is the company name, as used in the Assembly Manifest, which isn't all that important to us. Go ahead and set it to what you like; I'm going to just leave it blank.
Product: Like Company, this is only used in the Assembly Manifest. It's essentially the same as the Title property.
Copyright: This is a copyright notice that can be stored in the Assembly Manifest. Most of these properties related to the manifest aren't very important, and just show up in the executable file's properties window on Windows builds of the project.
Trademark: This works just the same as Copyright.
Assembly Version: Version of your product, used in the Assembly Manifest.
File Version: Because this option isn't compatible with XBOX 360 builds, don't even bother to include it.
GUID: Globally Unique Identifier. The default property is just fine.
Neutral Language: Language of the build (probably English)
Make assembly COM-visible: This is, once again, not available on the XBOX due to how things are handled. This is more important in software development than it is in game development, so we're not going to worry about it.

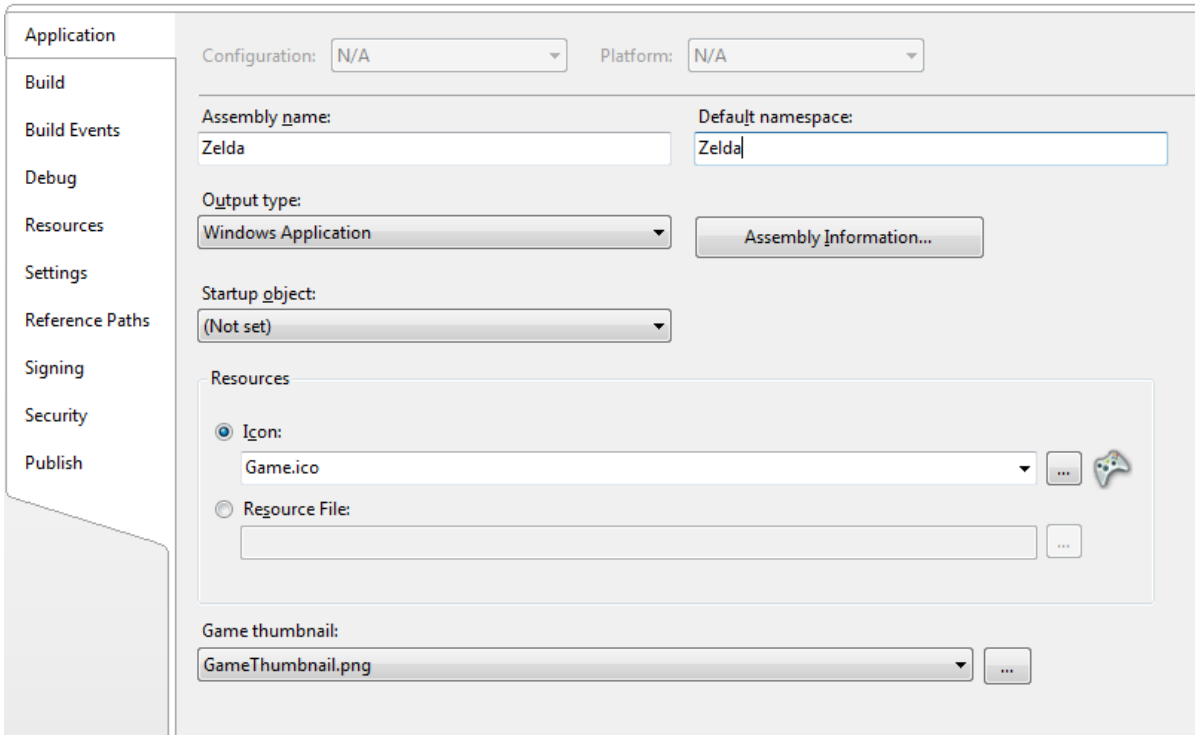


Figure 1-3: Project Properties Dialog

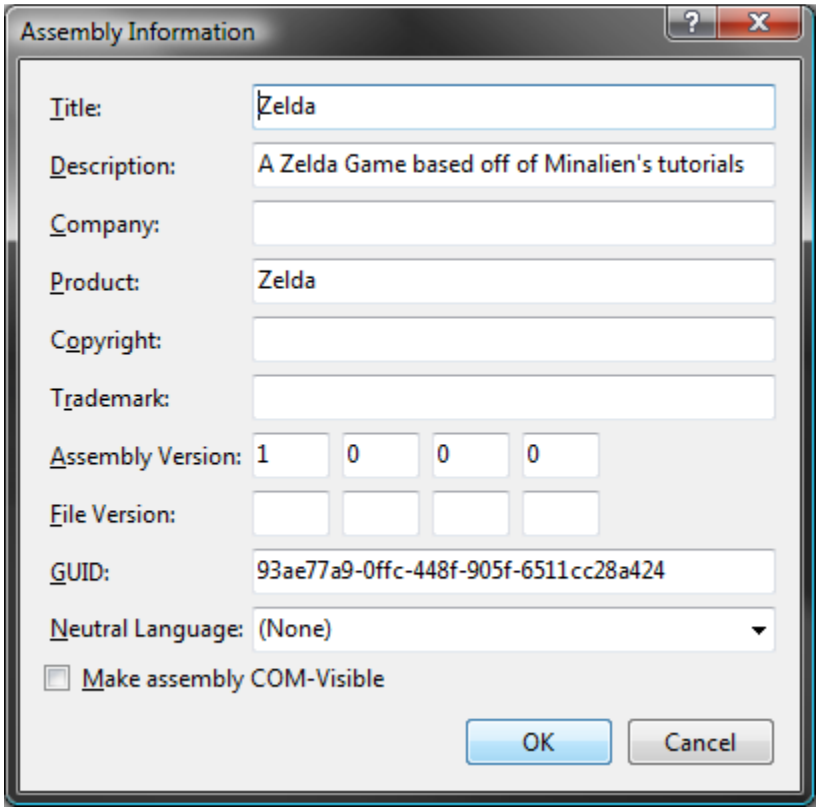


Figure 1-4: Assembly Information Example

Step III: Understanding what we have

We now have our settings defined, and I think it's safe for us to look at our project's default code, piece-by-piece.

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

Anybody even remotely familiar with C#, or any C-based language, should understand what's going on here. We're using the System namespace, Collections namespace (which gives us access to generic type lists, dictionaries, etc, which we will use later in the project), and gives us access to most anything XNA has to offer. These statements make it so that we don't have to type the entire namespace in order to use functionality provided by these namespaces.

```
namespace Zelda
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Engine : Microsoft.Xna.Framework.Game
    {
```

This portion simply defines our Zelda namespace, and creates our Engine class as a Microsoft XNA Game. This gives us our overloaded functions, and gives us access to various pieces of information that we need to get the game to run smoothly.

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

Now, we define two member variables in our Engine class: a Graphics Device Manager called graphics, and a Sprite Batch called spriteBatch. The Graphics Device Manager is an important component that gives us access to our graphics device, which allows us to do all kinds of fancy things, like set screen settings and displaying graphics on screen. A Sprite Batch is a very useful tool that greatly improves performance, and is, in my opinion, a great tool provided to us by Microsoft. Rather than sending each individual image to our graphics card, one at a time, our Sprite Batch allows us to instead send them all to the sprite batch, which, as you will see later on, will send all of the graphics information required by the graphics card at one time, reducing the load put on our graphics card.

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

Now, we have the Engine class's constructor. In here, we declare the game components to use (such as the graphics device manager), and set the root directory of our game media. Leave this to the default "Content" value, as that is the name of not only our content directory (which was created for us by XNA when we created the project), but also our Content sub-project.

```
        /// <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();
        }
```

This is the first of several important overridden functions of the Microsoft XNA Game class. Initialize is where we will call all of our basic, one-time initialization code. This function is called when the project is first started, and then doesn't get used again.

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

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

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

The LoadContent() method is where we will load static game content. This function is only going to be used to load content during the first few tutorials; later, we are going to have content load on a per-level basis. Content that is required throughout the entire game, such as a watermark that you may want to show during a demo version or trial version of your game, is the only thing that should be loaded here in a final version of our project. In here, we create the instance to our sprite batch, by passing it the graphics device. GraphicsDevice, which is passed in as a parameter, is not only a type, but is defined by the XNA Game class as a quick-and-easy method for us to access our game's graphics device, without going through our device manager. UnloadContent() will not be used in our project; its purpose is to unload content that we don't load through our content pipeline. Due to the fact that we will be loading all of our media through that, for the sake of our XBOX 360 project, we're going to just leave this here and ignore it.

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

            // TODO: Add your update logic here

            base.Update(gameTime);
        }
```

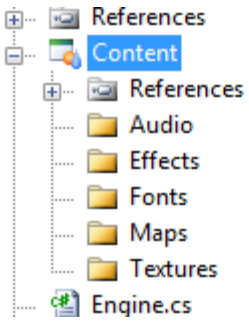
This function is where our game works its magic. From here, we will call all game logic; moving enemies, checking for keyboard or gamepad input, and checking for collision will all (though it will be farther down the object hierarchy) originate from this point. There is a conditional in here that checks if the Back button has been pressed on the first player's controller; if it has, the game exits. We will cover input more in-depth in a later tutorial. For now, keep this here, so that we have a way to exit the project without going through our XBOX Guide menu to exit it if we deploy our project to the XBOX 360.

```
        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing
values.</param>
        protected override void Draw(GameTime gameTime)
        {
            graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

            // TODO: Add your drawing code here

            base.Draw(gameTime);
        }
```

This is to drawing everything to the screen what the Update function was to accepting input and updating game data. First, we clear the graphics device to a Cornflower blue (light blue) color. You can change this color and see what happens when you run the project again. We will populate this with calls to our sprite batch later.

Step IV: Setting up our Content folder structure

We're going to go ahead and define our folder structure right now, as it will give us a base to work with later on. Click the [+] next to Content in the solution explorer, and you'll see a References area, just as with our game project. Right-click on Content, and select New Folder. We're going to create the following folder structure:



We'll expand on this later, but for now, know that it sets up an organized, basic, expandable system for storing and categorizing our game content.

After-Action Report

That about covers everything for this portion of the series; in the next few tutorials we're going to actually get started developing our engine, and getting things to show up. It's not going at a high rate of speed, but because we're also covering creating our engine, and getting everything from the background (input management, audio management, etc.), it's a semi-decent step into the door.

In this portion, we created the XNA game project, edited our settings, set the title for our game, overviewed the default game code, and defined a structure for our game media. We have now set the base on which the rest of the tutorials in the series will expand, bringing us up the chain.

Challenges:

There aren't really many challenges that we can go through with this project, but here are some things you can do:

1) Change the color that was clearing the background. This isn't much, but it gets you familiar with the Color structure, and will help you be more comfortable with accessing the graphics device in the future.
2) See what small tweaks you can do in terms of organization of the game code. See if there are parts that you can remove, and parts that you can optimize. I'm going to assume in the rest of the tutorials that nothing has changed, however, so don't clear things if you're not sure you'll remember what you did.