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:

There are no downloads associated with this tutorial.

Tutorial II: Showing Images and Accepting Input

This part of the tutorial series is going to cover a couple big pieces. First, we're going to begin to discuss XNA's Content Pipeline, which allows us to easily load and manage content in our projects, such as levels, images, audio, and others. Later in the tutorial series, we are going to extend the content pipeline to load our game's maps.

After that, we're going to discuss creating an input helper for your game class, which will tell you when keys and buttons are pressed, what's being held, and give you easy methods to accept and manage game input.

We're going to cover a lot of ground in this tutorial, but I'll work to make sure that I do it in a clean, concise manner, and I will try to keep it as easy to understand as possible. So, without further ado, let's jump in!

Part 1: Displaying an Image on Screen

Content Manager

If you remember from the first tutorial, we did a small amount of messing with the "Content" project that was created with our project, when we created our sub-folders ("Textures", "Audio", "Effects", etc) to store our data. Now, we're going to go a bit more in detail about the content manager; what it's for, why it helps us, and how to load and display images on screen.

When we compile our project, whether we are developing it for Windows or for the XBOX, the images, audio, and other data stored in the Content project gets compiled as well. This provides a small amount of compression in some cases, but more importantly, it pre-formats our images, audio, and other data so that it can load and display our media easily and without having to convert it, which saves us time, processing power, and memory.

The content manager also gives us an easy way to load data in our program. For example, if we were to use a pixel shader (developed in High Level Shader Language, which will be covered in a later tutorial) to create an effect for when Link drinks a poisonous potion, we would load the shader through our content loader.

It's not an easy thing to explain in text, so we're going to get started using it, which will give you a better grasp on how to use it and what it does for us, the developers.

Playing with the Graphics Settings

Since we have to walk before we run, we're just going to load a static, 800x600 image, Zelda.jpg, and display it on screen. For now, we're going to force our screen to stay at an 800x600 screen resolution, but in a later tutorial, we're going to learn how to change that value. Before we do anything, let's go ahead and set the graphics options. After opening your code and the Engine.cs file, find the Initialize() function of your Engine class, and add the following code after "// TODO: Add Your Initialization Logic Here".

```
// Screen Width & Height
graphics.PreferredBackBufferWidth = 800;
graphics.PreferredBackBufferHeight = 600;

// Vertical Sync (Prevents tearing)
graphics.SynchronizeWithVerticalRetrace = true;

// Apply Graphics Settings
graphics.ApplyChanges();

// Set the window title (PC)
Window.Title = "Legend of Zelda Engine";
```

First, we are setting the screen Width and the screen Height to 800 and 600, respectively. If we wanted, we could set it at 400x300, 1280x1024, or anything we wanted. 800x600 is a reasonable basis on which we can create our game, so we're going to go with this. Next, we ensure that we enable VSync. This will prevent tearing, which is a flickering effect that you get when your game renders faster than your monitor refreshes. Afterwards, we apply the graphics settings, which makes our screen switch to the new resolution and enable VSync, and we then change the title of our Window for the PC version of our game. If we run it, we still see the same thing as we would before running the code (unless we changed the title), because 800x600 is the default resolution used by XNA. This is useful, however, because it gives us the chance to come in and change it if we so choose to later.

Figure 2-1: The image we are going to be displaying on-screen

Adding the Image to our Content Manager

Now, let's show our fancy image (see Figure 2-1, above) on the screen. First, we must add the image to our project. Right-click on your Textures folder, click "Add Existing Item," then browse to and add the Zelda.jpg file. You should now see a [+] sign next to your Textures folder. Press it, and click on the Zelda.jpg file shown under it. In your property window, which is right below the solution explorer (if it isn't there, go to "View->Properties Window"), we see four important options:

Asset Name: This is the name we use to call our asset within our project, which you will see shortly, when we load it in the game.
Build Action: This makes sure that the file is compiled and included with your project.
Content Importer: This should be "Texture – XNA Framework". We will cover this and the next item in the tutorial about expanding the content processor. For now, leave this at the default value.
Content Processor: We will explain this in a later tutorial. For now, leave this at the default value.

We are now ready to load this image in our code.

Loading the Image in Code

To load this image, we need to declare a variable to hold it. We are going to declare this variable as a member of the Engine class; so, go up to the top of the class definition, and under the line where we defined our sprite batch, add the following:

```
Texture2D zeldaTexture;
```

A Texture2D object stores a two-dimensional image (a texture), and allows us easy access to the data on that image. The use of this will be instrumental in showing the image, as it will essentially *be* the image. Now, we are going to load the image. Go into your LoadContent() method, and after "// TODO: use this.Content to load your game content here", add the following:

```
zeldaTexture = Content.Load<Texture2D>("Textures/zelda");
```

This actually loads our image. Content is a default object (of type ContentManager), that allows us to access media in the content pipeline. We use this class to load sounds, images, shader effects, and, later, our maps. Inside the greater-and-less-than signs, we put the type of object we are loading; this allows us to tell the program how to format the data. Next, we set the relative path from our content manager to the asset. Because it is in the Textures folder, we use "Textures/" to signify that folder. Remember our asset name? We use that to identify the file; there is no use of the file's extension. The asset name defaults to the file name, but doesn't have to have anything to do with that. We could have changed our asset name to "rabid_wombat", and would have used "Textures/rabid_wombat" in its place, and nothing about the file itself would have changed.

Drawing the Image to the Screen

Now, we are going to draw our loaded image on the screen. Go to your Draw() method and add the following code after "// TODO: Add your drawing code here":

```
spriteBatch.Begin();
spriteBatch.Draw(zeldaTexture, new Vector2(0, 0), Color.White);
spriteBatch.End();
```

This actually draws the image through the use of our Sprite Batch. The first parameter is the Texture2D object we created earlier, which tells the engine what, specifically, to draw. Next, we define the point, which is a two –dimensional vector (Vector2). This takes an x and a y value as parameters, with the top-left corner of the screen being (0, 0). As you move right and down, x and y increase in value, while as you move left and up, they decrease, respectively. The third, and final, parameter tells the engine how to tint the image. If we had specified red instead of white, the image would appear to be tinted red; white is the neutral, no-tint color.

The first and last functions of the sprite batch, Begin() and End(), start and end the sprite batch's processing, respectively. When you call Begin(), the sprite batch starts accepting Draw() methods. When you call End(), the sprite batch sends all of the data from our Draw() methods to the screen. Run the program, and you should have the following image:

Figure2-2: Running and Displaying our Image.

Congratulations, you have now shown your first image on screen in this Zelda tutorial. Now let's move on to input management.

Part 2: Accepting and Handling Game Input with Components

Now, we're going to learn about another great feature of XNA – Game Components. Game components run alongside the rest of the program, and we create them to do specific functions, such as managing enemy AI, managing the player, and handling input. We're going to create a new file in the project. Right-click on your project, and select "Add->New Item", and select the Game Component template, and call our file InputManager.cs.

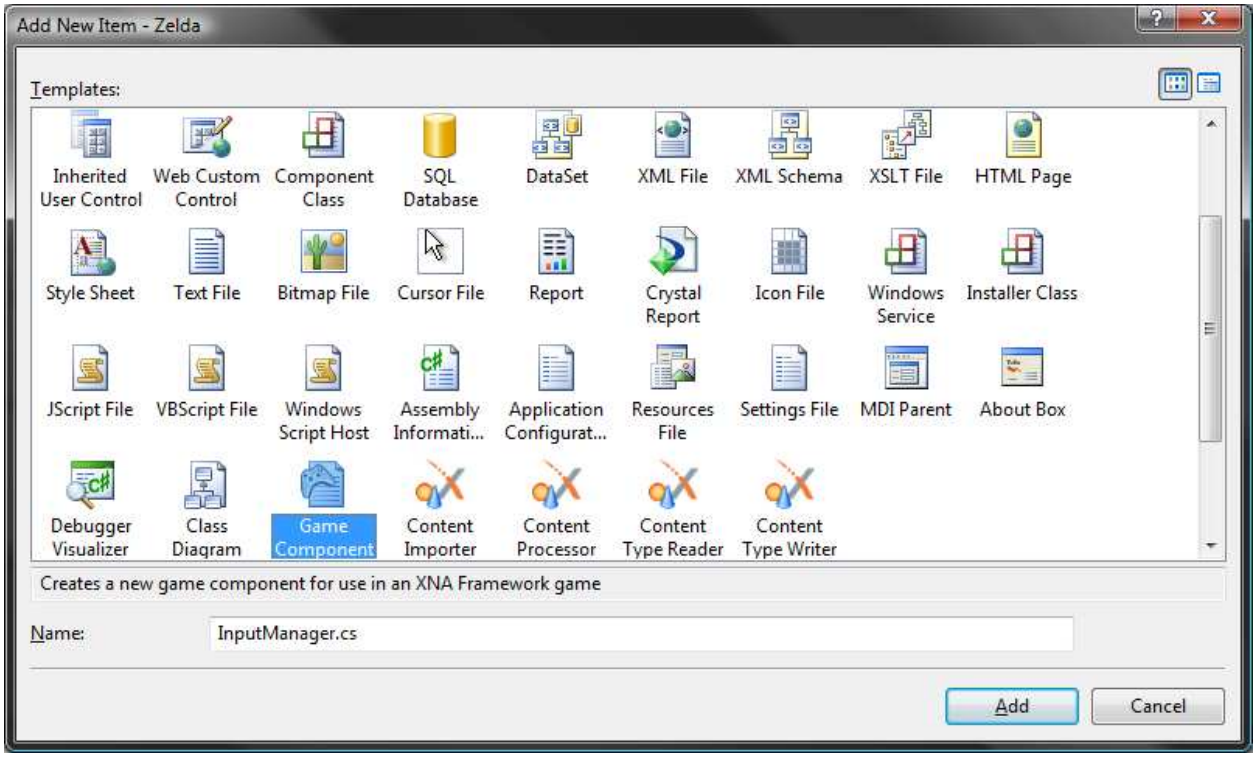

Figure 2-3: Selecting the GameComponent template.

After selecting "Add", we get the default code for our component, which looks very similar to our game engine's base code. We have a constructor, an Initialize() function, and an Update() function. The Initialize() method here is called after our game engine's Initialize() function. Now, we're going to start fleshing this out. We need to add two objects, one to handle keyboard input, and one to handle controller input. Go into your component's code, and at the top of the class, before our constructor, we are going to insert four member variables:

```
private KeyboardState prevKeyboard;
private KeyboardState curKeyboard;
private GamePadState prevGamePad;
private GamePadState curGamePad;
```

We declared two KeyboardState objects, which stores the state of the keyboard. We added one variable (prevKeyboard) to hold the state of the variable on the previous frame, and curKeyboard to store what it is on this frame. We also declared two GamePadState objectsin the same manager

Now, we are going to have our component update these variables. Inside the Update() function, after "// TODO: Add your update code here", add the following code:

```
            // Keyboard State
            prevKeyboard = curKeyboard;
            curKeyboard = Keyboard.GetState();

            // Gamepad
            prevGamePad = curGamePad;
            curGamePad = GamePad.GetState(PlayerIndex.One);
```

Because we are only supporting a single player in our game, all we need to worry about is the first player's controller. This updates both of our member variables, so that we now have access to our input devices, the Keyboard and the Game Pad. Now we are going to create two functions: one to check if a key is being pressed and held, and one to tell us if the button was just now pressed.

Under the Update() function, add the following to method:

```
        public bool IsHeld(Keys Key, Buttons Button)
        {
            // Keyboard
            if (curKeyboard.IsKeyDown(Key) && prevKeyboard.IsKeyDown(Key))
                return true;

            // Gamepad
            if (curGamePad.IsButtonDown(Button) &&
prevGamePad.IsButtonDown(Button))
                return true;

            // Neither is being held
            return false;
        }
```

This checks to see if a key or button was held. Because we are going to map our keyboard and our gamepad input to equivalent keys to ensure cross-platform compatibility, we are going to make our calls to both a key and gamepad button each time. First, we check if the keyboard button has been held, if so, we return true. Then, we check if it's being held on the controller, if neither is being held, we return false, signifying that no button is being held. Now, we are going to check if it was just now pressed.

```
        public bool IsPressed(Keys Key, Buttons Button)
        {
            // Keyboard
            if (curKeyboard.IsKeyDown(Key) && !prevKeyboard.IsKeyDown(Key))
                return true;

            // Gamepad
            if (curGamePad.IsButtonDown(Button) &&
!prevGamePad.IsButtonDown(Button))
                return true;

            // Neither was pressed
            return false;
        }
```

This is essentially the same as the IsHeld() function, except now we check to make sure that it is now pressed, whereas previously, it was not. Everything else is the same. Our input manager, for the time being, is done. So now, we're going to add it to our game.

Switch back to your Engine.cs file, and add a new member variable under our Texture definition.

```
        InputManager inputManager;
```

Now, jump to our constructor, and we're going to create the object, and add it to our game's list of components. Add the following after our "content.RootDirectory = "Content"" line.

```
            inputManager = new InputManager(this);
            Components.Add(inputManager);
```

And now, it's time to test it. We're going to make some small changes to our Update function. First, we are going to replace the current check for the Back button with the following:

```
            if (inputManager.IsPressed(Keys.Escape, Buttons.Back))
                this.Exit();
```

This now allows us to check for not only the Back button on the keyboard, but we also check the Escape key. If either of these is pressed, the game will exit. Now, after "// TODO: Add your update logic here", we are going to add the following, which will change our window title.

```
            if (inputManager.IsPressed(Keys.Enter, Buttons.Start))
                Window.Title = "Pressed Enter or Start";
```

If we press Enter, we will get the following effect:

Figure 2-4: When we press enter; testing our input manager.

After-Action Report

We've got all we need to start managing game input, now. We also learned how to show an image on the screen, how to set screen settings, how to exit the game, and how to change the window title on a Windows game project. Although we haven't gone very in-depth, we can actually use what we have right now to create an image-swapping test with our input manager. In the next tutorial, we are going to cover sprites, animations, and character movement.

Challenges

1) Use the input manager and your knowledge of texture objects to create something that swaps between two different textures when a button is pressed on the keyboard or on the gamepad.
2) Preempt the next tutorial and create your own moving image by changing the X & Y used to render an image based on input from our input manager.