Introduction to the XNA Framework

Getting Started

After installing XNA Game Studio, we will create a dual project solution. To keep things easy to manage, we are going to separate the components of our 3D Engine from the code used to run demos and applications on top of it so we can simply compile the engine into a library for use outside this solution. Start off by creating a Windows or Xbox Game project for the demos we will make to run our engine. Also, create a new Windows or Xbox 360 Game Library project. This will be the project we create all of the code for the game engine in. I named my projects 'HMDemo' and 'HMEngine'. Once the projects are created, delete the default Game.cs, Program.cs, and Class.cs files that were added so we can start from scratch. If you are creating an Xbox 360 version of the engine, you will want to go into the options and add an Xbox to send the compiled code to. Detailed instructions for doing so are located in the documentation.

Getting the Classes Ready to Render

To start, we need to create a class in the HMEngine project for our basic Game class. Mine will be called HMGame. To add the class, right click the project name and choose Add -> Class. Type the name of your class and click 'Add'. Also, just to be ready for running the demo, let's add a new class to the HMDemo project as well. I'll call this one 'HMDemo' like the project. We will also want to add the HMEngine as a Reference to the demo project, so right click 'References' and choose 'Add Reference'. Then, on the Projects tab, double click the HMEngine project. Finally, right click the demo project and set it as the startup project so when we click the Debug button it will look in that project for Main and not in our engine (since the engine is a class library and should have no Main). Now we are ready to start getting into some code!

For starters, let's set up our HMGame class to inherit from the XNA Game class to give us the ability of adding components to the game and further customizing it later on. To begin set up the code like this:

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

namespace HMEngine {
    public sealed class HMGame : Game {
    }
}
```

This used to set the file up to have a designer view opened when we double click it way back in the early XNA beta days, but since we are no longer able to use designers with XNA, we will have to start adding all of the necessary components by hand. Here is the first part of the game class, the graphics manager:

```csharp
public sealed class HMGame : Game {
    private readonly GraphicsDeviceManager myGraphics;
```

Tutorial 1 - Introduction to the XNA Framework

```
    public HMGame(int width, int height) {
        myGraphics = new GraphicsDeviceManager(this) {
            PreferredBackBufferWidth = width,
            PreferredBackBufferHeight = height
        };
    }

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

This will simply create a new graphics manager for us that will set up our game's width and height and call the Draw method of any components we add in later.

Most of the graphics manager functionality is done in the background with handling shader versioning and device enumeration, and now thanks to the updates in the XNA framework since version 1.0, we no longer have to explicitly check for our device or call to Present to the screen either. All of this is handled for us by the game class and the graphics device manager. To see this in action, we will need to create our Main function in the HMDemo class to run the game:

```
using HMEngine;

namespace HMDemo {
    internal static class HMDemo {
        private static readonly HMGame game = new HMGame(800, 600);

        public static void Main() { game.Run(); }
    }
}
```

Simple, no? If you set everything up correctly, you should be able to run the project now and see the blue background screen come up. Since that was so easy, we might as well add in some other functionalities while we are at it.

Resizing the Running Game

Simply allowing resizing of the window will not be enough to handle the actual changes we need to happen within our device when the aspect of the render target changes and the device is lost or reset. We are going to want to reset the graphics device with the new back buffer dimensions so our scene isn't simply being stretched to fit the window, but actually rendered at that size. To accomplish this, we simply need to add a custom event handler to the Window object that is already a part of our Game class. The code for this goes in the HMGame constructor:

```
// Need System for its EventArgs
using System;

public HMGame(int width, int height) {
    myGraphics = new GraphicsDeviceManager(this) {
        PreferredBackBufferWidth = width,
        PreferredBackBufferHeight = height
    };

    Window.AllowUserResizing = true;
    Window.ClientSizeChanged += Window_ClientSizeChanged;
}
```

Tutorial 1 - Introduction to the XNA Framework

```csharp
private void Window_ClientSizeChanged(object sender, EventArgs e) {
    myGraphics.PreferredBackBufferWidth = Window.ClientBounds.Width;
    myGraphics.PreferredBackBufferHeight = Window.ClientBounds.Height;
}
```

This will automatically resize the backbuffer for us to match the new window size whenever the user decides to change the window itself.

What About Going Fullscreen?

Glad you asked. The GraphicsDeviceManager class has a built in function called ToggleFullScreen that will handle the needed things for us. Calling this function can be done in many ways, but for now I am simply going to capture a keypress on the letter 'f' and map that to the toggling. The place to put this type of functionality in my opinion (at least until we get some real input handling) is inside the Update method of the Game class, so we will overwrite it and place our key handling there:

```csharp
protected override void Update(GameTime gameTime) {
    KeyboardState kState = Keyboard.GetState();
    if (kState.IsKeyDown(Keys.F)) {
        myGraphics.ToggleFullScreen();
    }

    base.Update(gameTime);
}
```

You'll need to add a reference to the Input classes provided by XNA as well to use the KeyboardState to check for key presses:

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

That's it! We now have a fully functioning XNA based playground to start building our engine in. I hope you liked this quick intro to the XNA framework.