## Stock Effects

Provides source code for the five effects that come built-in to the XNA Framework (BasicEffect, SkinnedEffect, EnvironmentMapEffect, DualTextureEffect, and AlphaTestEffect), plus the default shader used by SpriteBatch (SpriteEffect).

Also includes a commandline utility (CompileEffect) which uses the Content Pipeline to compile a .fx source file into a binary blob which can be passed directly to the XNA Framework Effect class constructor.

## Overview

This code is provided for educational purposes. It may be a useful starting point when you create more advanced shaders of your own.

Note that although the built-in effect classes are simple and easy to use, the shader code behind them is actually quite complicated! This is because these effects support many rendering options within a single shader. For instance BasicEffect allows you to toggle texturing, vertex colors, and lighting on or off, and to select per-vertex or per-pixel lighting. To support all the possible permutations of these options, BasicEffect ends up needing no less than 20 different vertex shaders and 10 pixel shaders, all of which are combined within a single effect file.

If you are looking for an easy starting point to learn shader programming, you would probably be better off with the Shader Series samples, which are simpler than these effects because they do not include so many adjustable options.

## Using the Effects

To use these effects instead of the built-in versions, first you must compile the appropriate solution (StockEffectsWindows.sln or StockEffectsXbox.sln, depending on which platform you are developing for). Then you can right-click on your project, choose Add Reference, switch to the Browse tab, and select StockEffects\bin\<platform>\Debug\StockEffects.dll.

This assembly provides six classes:

* BasicEffect
* SkinnedEffect
* EnvironmentMapEffect
* DualTextureEffect
* AlphaTestEffect
* SpriteEffect

But these new versions are found in the StockEffects namespace, as opposed to Microsoft.Xna.Framework.Graphics. To use these versions instead of the built-in classes, simply change:

effect = new BasicEffect(GraphicsDevice);

to:

effect = new StockEffects.BasicEffect(GraphicsDevice);

## Changing the Effects

To tweak the shader code, you can simply edit the .fx files, then recompile the StockEffects.dll assembly. But if you want to make bigger changes or reuse these shaders in your own projects, you may need to understand how they are compiled.

The CompileEffect project is a commandline utility which uses the Content Pipeline to compile a .fx file into a binary blob. It does this by calling directly into the EffectImporter and EffectProcessor classes, then saves out the resulting compiled effect code, bypassing the part of the Content Pipeline that would normally embed this data into an .xnb format file.

The StockEffects project has a custom build rule which uses the CompileEffect utility to compile all six effects. To edit this rule, right-click on the StockEffects project in Solution Explorer, choose Properties, switch to the Build Events tab, and click the Edit Pre-build button (you may need to increase the size of this dialog to get rid of the confusing word-wrapping).

You will see that this build rule calls CompileEffect six times, building each .fx file into a .bin file, which is stored in the obj folder.

If you edit Resources.resx, you will see that it includes all six of these .bin files from the obj folder, so the compiled shader code is embedded directly as an assembly resource.

This allows the effect wrapper classes (eg. BasicEffect.cs) to access the compiled effect code via Resources.BasicEffect.

If you want to add or remove shaders, you must follow these steps:

* Add or remove the line that compiles the shader from the custom build rule
* Add or remove the shader output .bin file from Resources.resx

## Effect Include Files

Many of the built-in effects have similar behaviors. For instance they all use the same fog computations, while BasicEffect, SkinnedEffect, and EnvironmentMapEffect all share the same Blinn/Phong lighting model.

To reduce code duplication, this shared functionality is implemented in header files (Common.fxh, Lighting.fxh, and Structures.fxh), which are included by all the effects.

Macros.fxh serves a different purpose. This defines some helper macros which allow the same shader source code to be compiled for DX9 shader model 2.0 (as used by the XNA Framework Content Pipeline) or alternatively for shader model 4.0 (which is used by DX10/11, and has a slightly different syntax).

## Preshaders

A common tension in shader programming is that when you design effect parameters to provide a nice clean API, the resulting parameter formats are not always the most efficient for HLSL optimization.

D3D tries to correct any such mismatches through a feature called "preshaders". The HLSL compiler looks for computations that are the same for all vertices or all pixels, and moves these out of the main shader into a special setup pass which runs on the CPU before drawing begins. This is a great feature, but has a couple of fatal flaws:

* The HLSL compiler does not always spot every optimization possibility
* The virtual machine that evaluates preshaders is not especially efficient
* Preshaders are not supported on Xbox or Windows Phone

Instead, Game Studio implements preshader computations in C#, by overloading the Effect.OnApply method, which is called immediately before EffectPass.Apply sets parameter values onto the graphics device.

This allows our C# effect wrapper classes to expose whatever properties the API requires, without needing these to match the underlying HLSL shader parameters. When the programmer changes a managed property, we just set a dirty flag, then recompute derived HLSL parameter values during OnApply. We used this ability to precompute many things:

* Collapse the World, View, and Projection matrices into a single WorldViewProj matrix.
* When lighting is enabled, compute the WorldInverseTranspose matrix. This is necessary for correct normal transforms when using non-uniform scales, but something we never bothered to do right in previous versions.
* Extract the EyePosition vector from the View matrix.
* Combine FogStart, FogEnd, World, and View, generating a vector that can compute fog amount with a single dot product.
* Merge the DiffuseColor, EmissiveColor, AmbientLightColor, and Alpha properties into a more efficient set of combined parameters.