User's Guide

FX Composer 2
Beta

May 2007
DU-02761-001_v01

Table of Contents

List of Figures

List of Tables

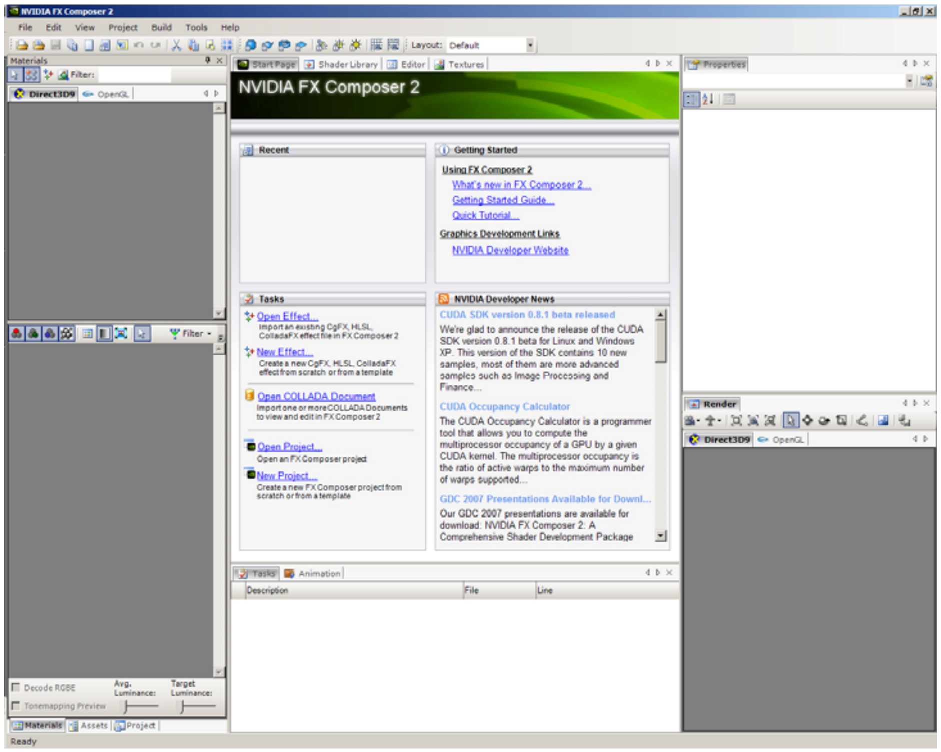
Quick Tutorial

Overview

This chapter presents a very short FX Composer 2 tutorial to quickly introduce you to several convenient and powerful new features. Even if you've used FX Composer 1.8, we highly recommend that you read through this tutorial because so many things have been improved in FX Composer 2.

Creating an Effect

When you first run FX Composer 2, you'll see its **Start Page**. This page gives you convenient access to several commonly-used commands and resources.



Import an existing CgFX, HLSL, ColladaFX effect file in FX Composer 2

Create a new CgFX, HLSL, ColladaFX effect from scratch or from a template

Import one or more COLLADA Documents to view and edit in FX Composer 2

Open an FX Composer project

Create a new FX Composer project from scratch or from a template

CUDA SDK version 0.8.1 beta released

We're glad to announce the release of the CUDA SDK version 0.8.1 beta for Linux and Windows XP. This version of the SDK contains 10 new samples, most of them are more advanced samples such as Image Processing and Finance...

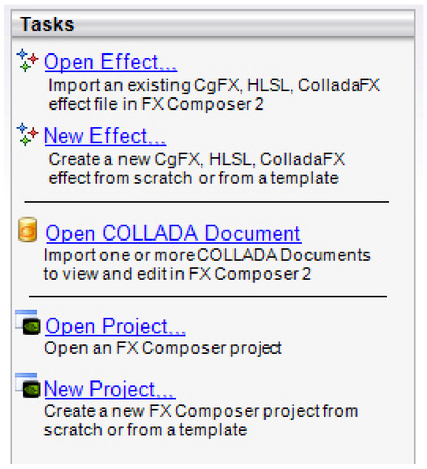
CUDA Occupancy Calculator

The CUDA Occupancy Calculator is a programmer tool that allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported...

GDC 2007 Presentations Available for Downl...

Our GDC 2007 presentations are available for download: NVIDIA FX Composer 2: A Comprehensive Shader Development Package

Let's create a new effect. Select **New Effect** under the Tasks portion of the Start Page.


Tasks

Open Effect...
Import an existing CgFX, HLSL, ColladaFX effect file in FX Composer 2

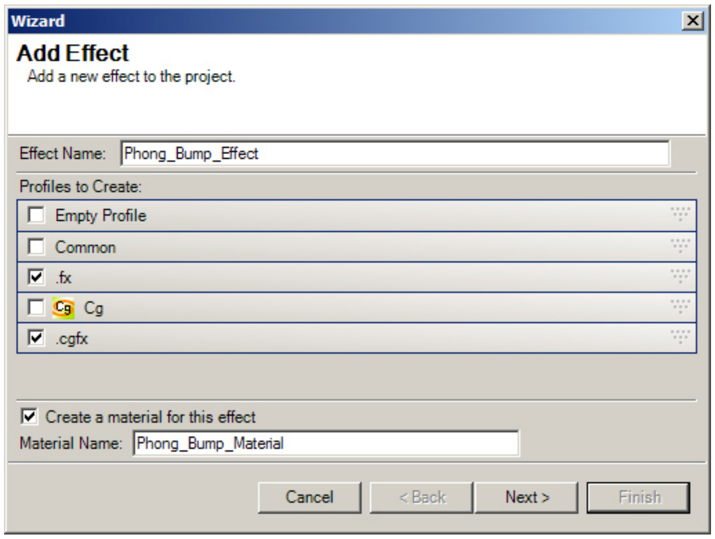
New Effect...
Create a new CgFX, HLSL, ColladaFX effect from scratch or from a template

Open COLLADA Document
Import one or more COLLADA Documents to view and edit in FX Composer 2

Open Project...
Open an FX Composer project

New Project...
Create a new FX Composer project from scratch or from a template

A short wizard will pop up, guiding you through the creation of your new effect.


Wizard

Add Effect
Add a new effect to the project.

Effect Name: Phong_Bump_Effect

Profiles to Create:

- [ ] Empty Profile
- [ ] Common
- [x] .fx
- [ ] Cg
- [x] .cgfx

[x] Create a material for this effect

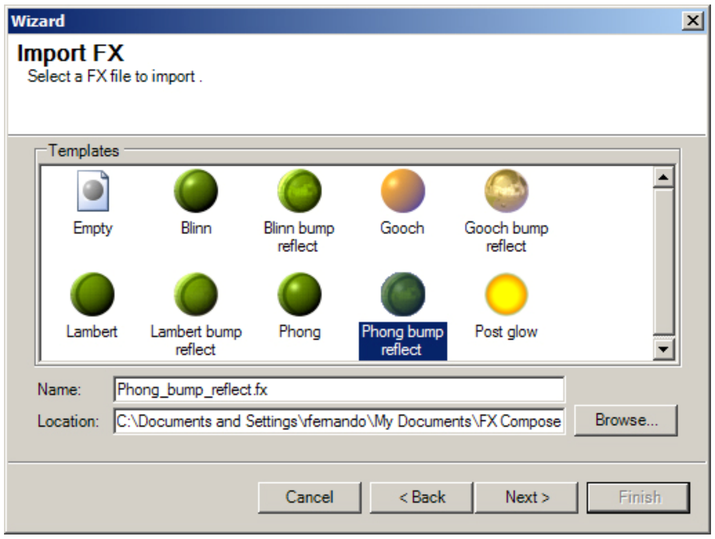
Material Name: Phong_Bump_Material

The wizard will prompt you for the types of shaders you want to add. Select .fx and .cgfx. Also, set the **Effect Name** as "Phong_Bump_Effect", and set the **Material Name** to "Phong_Bump_ Material".

You may be wondering what the difference is between a material and an effect. An "effect" is a shader—for example, marble. A "material" is an instance of an effect with specific properties settings—for example, green marble. Materials are what you actually apply to objects in your scene.

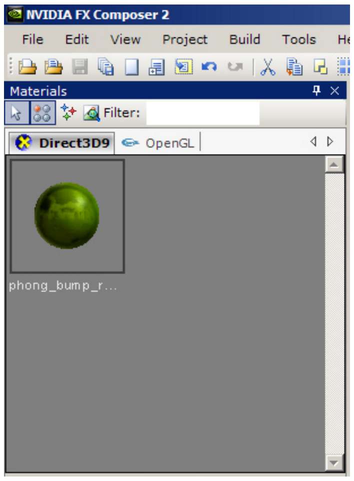
The advantage of having effects and materials is that you can modify the underlying shader code of several materials at once if they are based on the same effect, simply by modifying the effect. Without a materials system, you would have to create separate shaders for each material variant and modify all of these shaders individually to achieve the same result.

Click **Next**. You'll now get a chance to pick from a variety of shader templates for the .fx shader effect.



Choose **Phong Bump Reflect** and click **Next**. Then choose **Phong Bump Reflect** for the .cgfx file, and click **Finish**.

You'll now see a sphere in the Materials panel, shaded using your new effect.



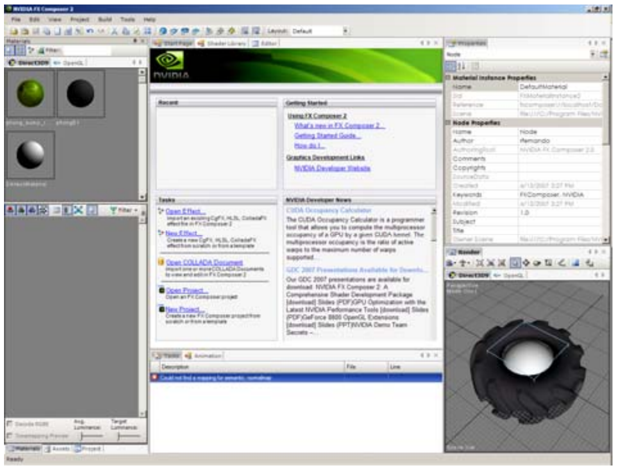
Importing Geometry

The next step is to create some geometry. On the Start Page, click on **Open COLLADA Document**. (Alternatively, you could go to the Project menu and choose **Import** to load other formats such as .fbx, .3ds, .obj, or .x.

In the file dialog box, choose:
FX Composer 2/MEDIA/COLLADA/tire_big.dae

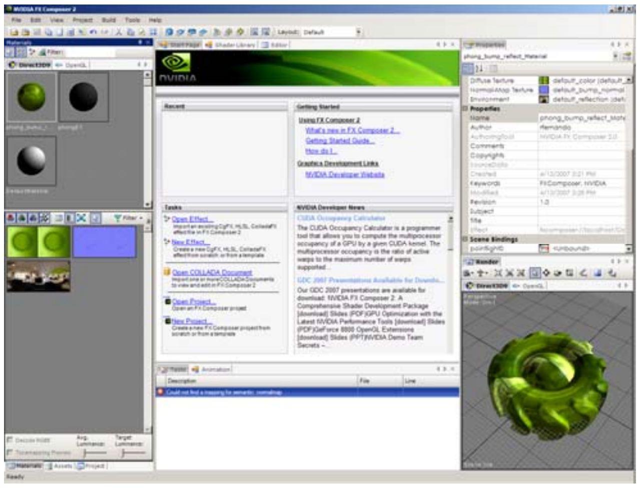
You'll now see a black tire in the Render panel. Use **Alt + Leftmouse** in the Render panel to get a better vantage point by rotating the view. **Shift + Leftmouse** zooms in and out, and **Ctrl + Leftmouse** pans.

Let's also create a sphere by clicking on the **Create Sphere** icon on the upper toolbar. The sphere will appear at the world's origin, so it happens to fit conveniently inside the tire. Make sure the Render panel's **Direct3D** tab is active.



Applying Materials to Geometry

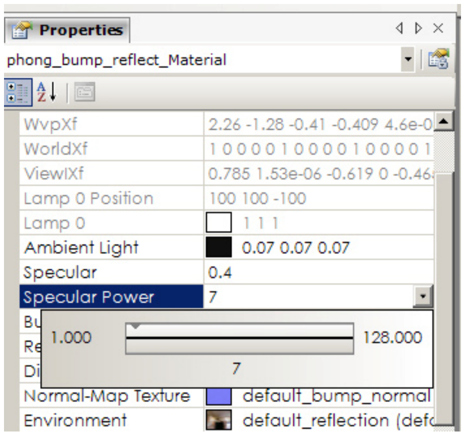
Now it's time to apply our material to our geometry. To do this, simply drag-and-drop the **Phong Bump Reflect material sphere** from the **Materials panel** onto the tire, and then repeat the process for the sphere.



Modifying Material Parameters

Click on the **Phong Bump Reflect** material sphere in the Materials panel. This will show the material's properties in the Properties panel.

Scroll down in the **Properties panel** until you see the **Specular Power** parameter. Click on its value and change it to 7 either by using a slider or by typing in the value directly. You should see the **Render panel** updating dynamically as you change the parameter.

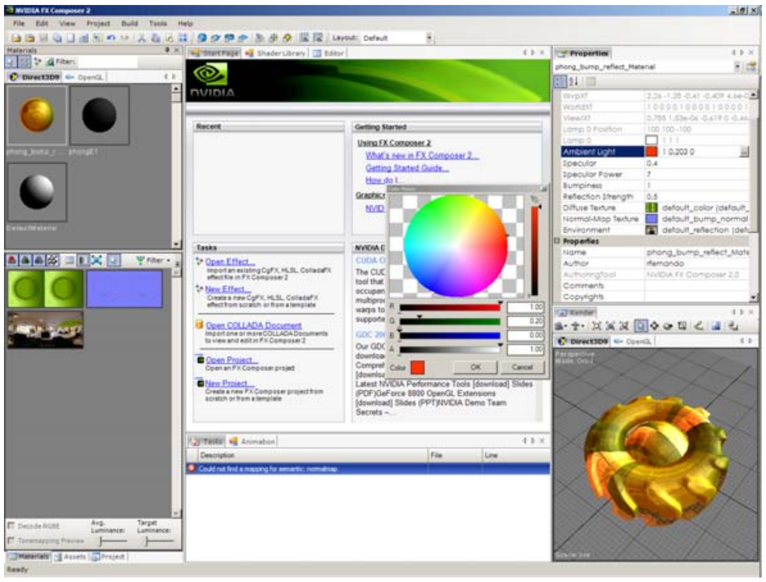


Properties

phong_bump_reflect_Material

WvpXf	2.26 -1.28 -0.41 -0.409 4.6e-0
WorldXf	1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
ViewIXf	0.785 1.53e-06 -0.619 0 -0.46
Lamp 0 Position	100 100 -100
Lamp 0	1 1 1
Ambient Light	0.07 0.07 0.07
Specular	0.4
Specular Power	7
Bu	1.000 — 128.000 (7)
Re	
Di	
Normal-Map Texture	default_bump_normal
Environment	default_reflection (defa

Do the same for the **Ambient Light** parameter. This is a color, so you'll use FX Composer's HDR color picker to pick a new color. In the color picker, dragging sliders with the left mouse button will change their base (mantissa) values. Dragging slides with the right mouse button changes their exponent. Make sure to use the left mouse button and drag the brightness slider (to the right of the color gamut) upwards. Again, the **Render panel** will show all your changes applied to the scene in real-time.

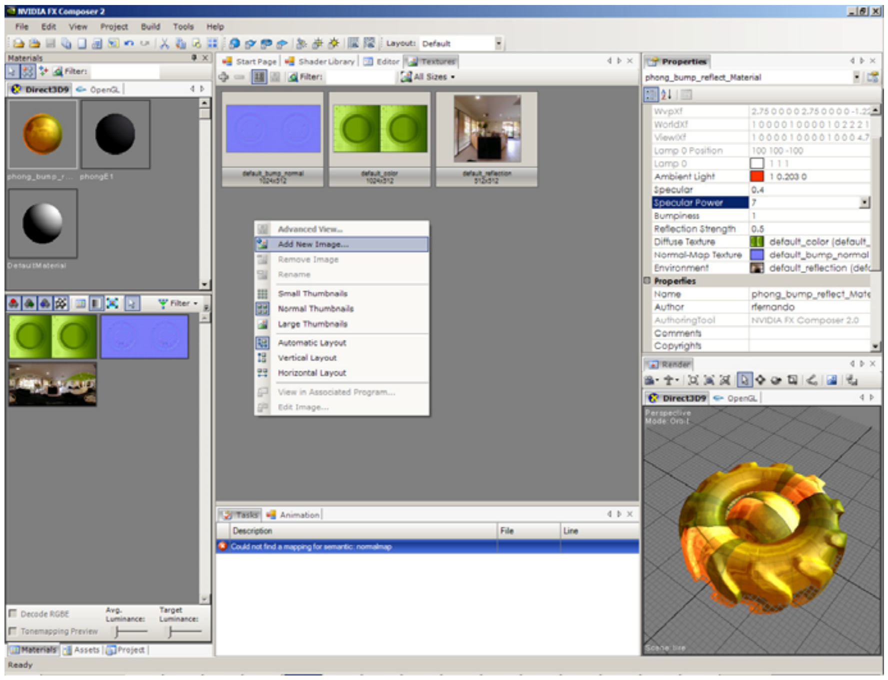


Assigning Textures

If you look through the list of parameters in the **Properties panel**, you'll notice several textures: Diffuse Texture, Normal-Map Texture, and Environment. Let's change the diffuse texture.

FX Composer has a **Textures panel** specifically for working with 2D, 3D, and cube map textures. View that panel by selecting **Textures** from the **View** menu.

Now click on the Texture Panel's toolbar's button ➕ to add new images. It is worth noting that you can also drag and drop image files from Windows Explorer directly into the Texture Panel).



Could not find a mapping for semantic: normalmap

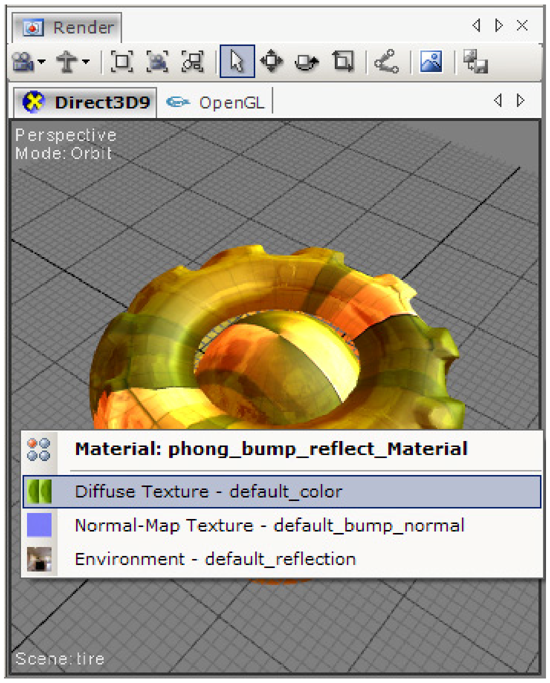
phong_bump_reflect_Material

WvpXf	2.75 0 0 0 0 2.75 0 0 0 0 -1.22
WorldXf	1 0 0 0 0 1 0 0 0 0 1 0 2 2 2 1
ViewXf	1 0 0 0 0 1 0 0 0 0 1 0 0 0 4.7
Lamp 0 Position	100 100 -100
Lamp 0	1 1 1
Ambient Light	1 0.203 0
Specular	0.4
Specular Power	7
Bumpiness	1
Reflection Strength	0.5
Diffuse Texture	default_color (default_
Normal-Map Texture	default_bump_normal
Environment	default_reflection (def
Properties	
Name	phong_bump_reflect_Mate
Author	rfernando
AuthoringTool	NVIDIA FX Composer 2.0
Comments	
Copyrights	

In the file dialog box, choose:
FX Composer 2/MEDIA/textures/2D/rockwall.jpg

You'll now see **rockwall.jpg** in the list of textures. Double-click on your texture to see detailed information about it.

Drag-and-drop the **rockwall.jpg** texture thumbnail onto the tire. You'll now be prompted for which of the Phong Bump Reflect material's textures to replace. Choose **Diffuse Texture**. Note that both the tire and the sphere change because they use the same material.


Render
Direct3D9
OpenGL
Perspective
Mode: Orbit
Material: phong_bump_reflect_Material
Diffuse Texture - default_color
Normal-Map Texture - default_bump_normal
Environment - default_reflection
Scene: tire

Binding a Light to a Material

Now let's bind a light to your material. This means that when you move the light, you'll see the material's shading change. (Where there are no lights in the scene, FX Composer searches through your material for the first light object it can find, and it uses the default positions specified there.)

Click on the **Add Spotlight** button on the main toolbar to add a spotlight to your scene. The spotlight is created at the world origin, so it's obscured by the sphere.

Click on the **Translate Object** icon in the Render panel. Now you'll see a set of axes at the origin for the light. Clicking and dragging on any individual axis will allow you to move the light along just that axis. For free movement, click on the grey circle at the intersection of the axes.

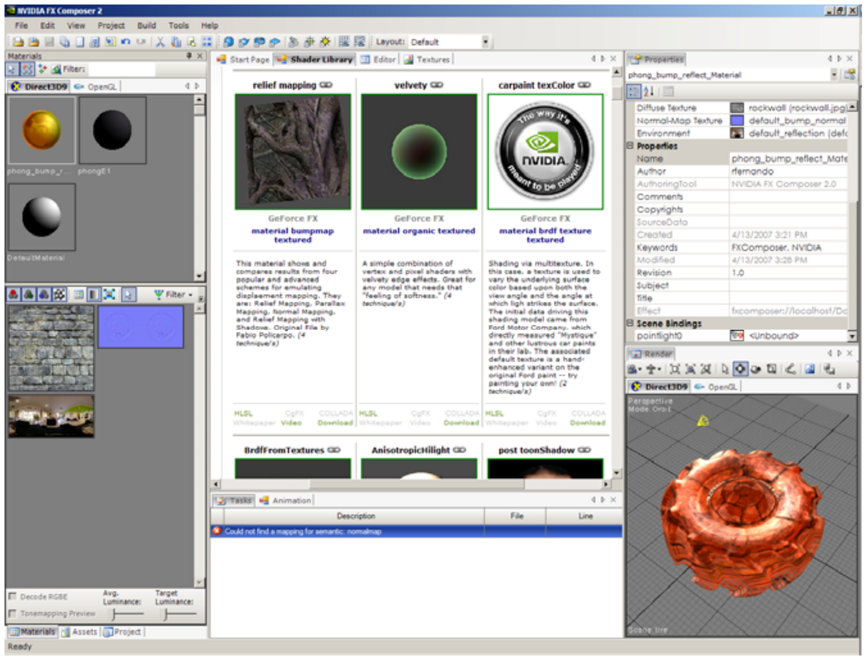
Move the light to a reasonable location above the tire and sphere.

Now click on the **Select Object** icon in the Render panel. Click on the light to select it. Click-and-drag the light onto the tire. This will automatically bind the light to the tire's material. (If a material has several light inputs, you will be prompted for which one to use.)

Your light is now bound to the material. If you switch to **Translate Object** again and move the light around, you'll see the lighting on the tire respond to the light position. (But notice that the sphere, which isn't bound to the light, doesn't change its appearance as the light moves.)

Shader Library

The NVIDIA Shader Library, which is tightly integrated with FX Composer, offers a vast collection of great shaders for both inspiration and extension. You can drag-and-drop shaders from the Shader Library onto objects in your scene.



relief mapping

GeForce FX

material bumpmap textured

This material shows and compares results from four popular and advanced schemes for emulating displaement mapping. They are: Relief Mapping, Parallax Mapping, Normal Mapping, and Relief Mapping with Shadows. Original File by Fabio Policarpo. *(4 technique/s)*

velvety

GeForce FX

material organic textured

A simple combination of vertex and pixel shaders with velvety edge effects. Great for any model that needs that "feeling of softness." *(4 technique/s)*

carpaint texColor

GeForce FX

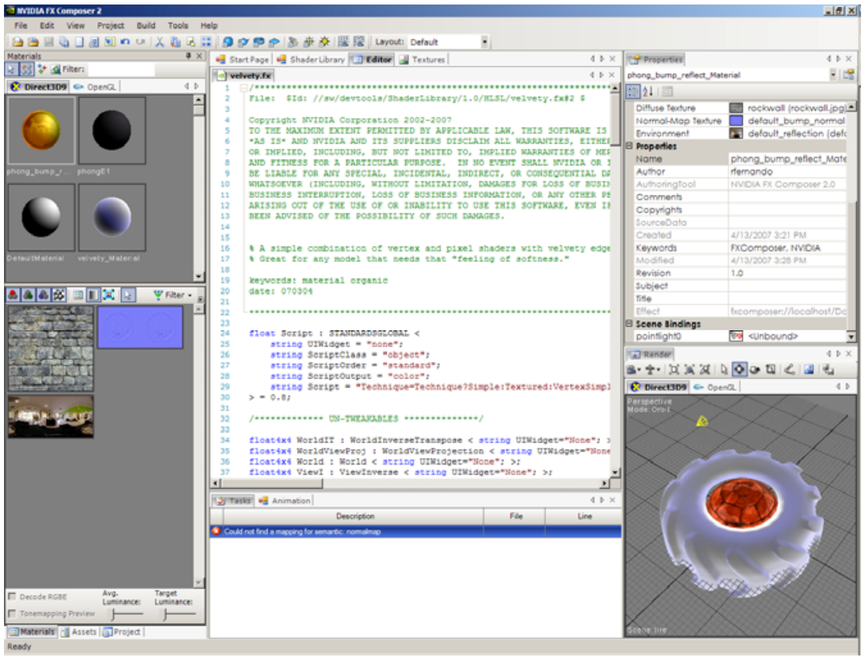
material brdf texture textured

Shading via multitexture. In this case, a texture is used to vary the underlying surface color based upon both the view angle and the angle at which ligh strikes the surface. The initial data driving this shading model came from Ford Motor Company, which directly measured "Mystique" and other lustrous car paints in their lab. The associated default texture is a hand-enhanced variant on the original Ford paint -- try painting your own! *(2 technique/s)*

Description	File	Line
Could not find a mapping for semantic: normalmap		

To do this:

Click on the **Shader Library** tab in FX Composer's central panel. Click-and-drag "velvety" onto the tire. (Optionally, you can re-associate the light with the tire by dragging-and-dropping it again.)



```
/***************************************************************
File:  $Id: //sw/devtools/ShaderLibrary/1.0/HLSL/velvety.fx#2 $

Copyright NVIDIA Corporation 2002-2007
TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THIS SOFTWARE IS
*AS IS* AND NVIDIA AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EITHE
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MER
AND FITNESS FOR A PARTICULAR PURPOSE.  IN NO EVENT SHALL NVIDIA OR I
BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DA
WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSIN
BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PE
ARISING OUT OF THE USE OF OR INABILITY TO USE THIS SOFTWARE, EVEN IF
BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


% A simple combination of vertex and pixel shaders with velvety edge
% Great for any model that needs that "feeling of softness."

keywords: material organic
date: 070304

****************************************************************

float Script : STANDARDSGLOBAL <
    string UIWidget = "none";
    string ScriptClass = "object";
    string ScriptOrder = "standard";
    string ScriptOutput = "color";
    string Script = "Technique=Technique?Simple:Textured:VertexSimpl
> = 0.8;

/************* UN-TWEAKABLES **************/

float4x4 WorldIT : WorldInverseTranspose < string UIWidget="None"; >
float4x4 WorldViewProj : WorldViewProjection < string UIWidget="None
float4x4 World : World < string UIWidget="None"; >;
float4x4 ViewI : ViewInverse < string UIWidget="None"; >;
```

Description	File	Line
Could not find a mapping for semantic: normalmap		

Editing Shaders

You can quickly access a material or effect shader source code by bringing its context menu in the Material Panel and selecting the View Code menu item.

At this point though, you have noticed that dragging-and-dropping velvety.fx from the Shader Library onto the scene has automatically opened velvety.fx in FX Composer's editor. (You can change this default behavior via the **Settings…** option of the Tools menu.)

Press **Ctrl+F** and search for the word "result".

Let's modify the shader by changing the result expression to:

```
half3 result = diffContrib – specContrib;
```

Press **Ctrl+F7** to recompile the shader. The Render panel will update to reflect the new shader as well.

Overview

What Is FX Composer 2?

The NVIDIA® FX Composer™ 2 software package is an integrated development environment for modern shader development in both OpenGL and DirectX with support for multiple shading languages. FX Composer empowers developers to create high-performance shaders with real-time preview and optimization features available only from NVIDIA.

Designed to make shader development and optimization easier for programmers, FX Composer offers an intuitive user interface for artists customizing shaders in a particular scene.

In addition, FX Composer is bundled with Mental Mill Artist Edition, a free visual shader authoring tool from mental images that allows developers to rapidly prototype shaders by connecting blocks.

Support for Standard Development Features

FX Composer supports all the standard features you would expect in an integrated development environment for high-performance shaders, such as the following:

- Start Page to guide users getting started with common tasks. This allows users to create new shaders and projects from a set of templates.
- Offers sophisticated text editing with syntax highlighting.
- Works directly with COLLADA FX, CgFX, and HLSL.FX files to create multiple techniques and passes. Lets you use the files you create with FX Composer 2 directly in your application.
- Supports the authoring of complex full scene effects such as shadow mapping or depth of field.
- Features convenient, artist-friendly graphical editing of shader properties and certain attributes, with undo/redo.
- Offers scene manipulation and object creation functionality, including undo/redo.
- Supports Microsoft DirectX 9.0 standard HLSL semantics and annotations.
- Allows the use of custom semantics and annotations and vertex stream packing to facilitate shader integration into real-world production pipelines.
- Provides a complete plug-in architecture that allows arbitrary extensibility, including important custom scene data, support for additional shading languages, and user interfaces.
- Supports Python scripting to automated common production tasks like material parameter binding to scene objects.
- Predefined and Custom layouts for more comfort dealing with certain authoring and editing tasks.

Visual Debugging and Performance Tuning

FX Composer 2 provides developers with visual debugging and advanced performance tuning features, such as:

- Practical SDK that enables custom importers, exporters, and scene modifiers.
- Visible preview of intermediate (generated) textures and render targets.
- Capture of pre-calculated functions to texture look-up tables and save render ports to disk.
- Interactive compiler that shows where the problems are—it jumps directly to problems in your source code.
- Handles minimal recompilation of dependencies with shader include files, per effect compilation options and directives, and custom build configurations.
- Simulated performance results for the entire family of NVIDIA GPUs through NVIDIA ShaderPerf 2 integration.
- Empirical performance metrics such as GPU cycle count, temporary register usage, and vertex and pixel throughput.
- Support for industry standard 3d file formats: COLLADA, OBJ, X, 3DS and FBX.

Mental Mill Artist Edition

Bundled with FX Composer 2, Mental Mill Artist Edition offers a convenient visual shader authoring system. In Mental Mill Artist Edition, you can quickly connect different modules to create new shaders – an ideal approach for artists and technical directors who may not want to work directly with shader code.

Shaders can then be exported to CgFX or HLSL, and then loaded in FX Composer's production-friendly development environment as shown in Figure 1.

For more information, please visit http://www.mentalimages.com/2_4_mentalmill/index.html.

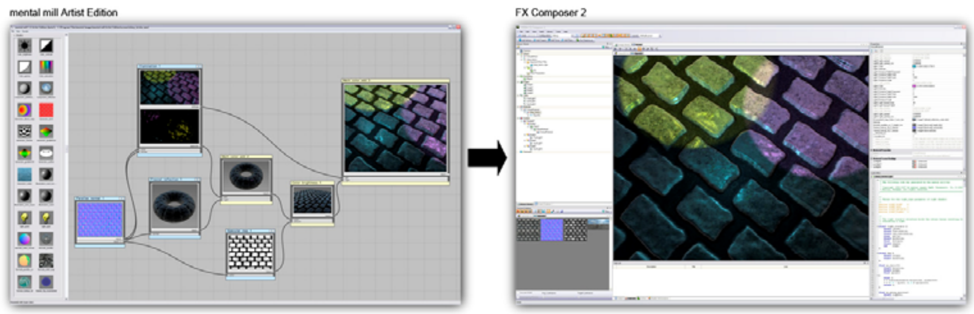
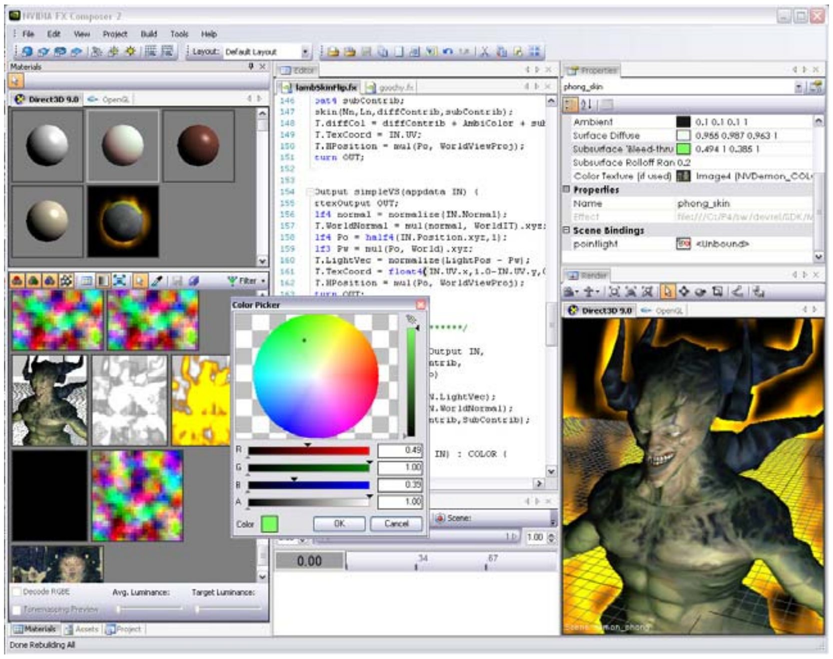


Figure 1. Exporting a Shader from Mental Mill Artist Edition to FX Composer 2

The Basics

Layout

FX Composer is made up of several panels that present a wide range of information, such as available materials, properties, shader code, rendered output, and more. Together, they create a comprehensive environment for shader authoring. A typical FX Composer 2 screenshot is shown in Figure 2.



```
oat4 subContrib;
skin(Nn,Ln,diffContrib,subContrib);
T.diffCol = diffContrib + AmbiColor + sub
T.TexCoord = IN.UV;
T.HPosition = mul(Po, WorldViewProj);
turn OUT;


Output simpleVS(appdata IN) {
rtexOutput OUT;
lf4 normal = normalize(IN.Normal);
T.WorldNormal = mul(normal, WorldIT).xyz;
lf4 Po = half4(IN.Position.xyz,1);
lf3 Pw = mul(Po, World).xyz;
T.LightVec = normalize(LightPos - Pw);
T.TexCoord = float4(IN.UV.x,1.0-IN.UV.y,(
T.HPosition = mul(Po, WorldViewProj);
turn OUT;
```

phong_skin

Ambient	■	0.1 0.1 0.1 1
Surface Diffuse	□	0.955 0.987 0.963 1
Subsurface 'Bleed-thru	🟩	0.494 1 0.385 1
Subsurface Rolloff Ran	0.2	
Color Texture (if used)	▨	Image4 (NVDemon_COL
Properties		
Name	phong_skin	
Effect	file:///C:/P4/sw/devrel/SDK/M	
Scene Bindings		
pointlight		<Unbound>

Figure 2. Layout of FX Composer 2

FX Composer 2 consists of several panels that can be docked in the main window or arranged outside the main window to more convenient locations. To move a panel to a different location, click and drag the panel's title bar.

Note: When you dock a panel, you will see an outline of where the panel would dock. This docking location is determined by the location of the mouse pointer.

If a panel contains several sub-panels, you can control docking by clicking and dragging a sub-panel's tab. While you're dragging the tab, you will see a docking layout control at the center of the panel (Figure 3) that shows possible docking locations. Move the mouse pointer over the control to preview what the docking would look like. When you are satisfied with the preview, release the mouse button to dock the window.

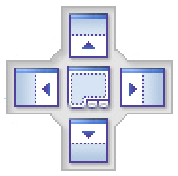


Figure 3. Docking Layout Control

Before you start working with FX Composer, it's important to understand what the various panels do. The following list briefly describes the panels and their functions:

- **Start Page**. Several shortcuts for common tasks.
- **Code Editor.** Edit shader source code.
- **Properties Panel.** View and modify shader properties.
- **Shader Library Panel.** Download shaders from the NVIDIA Shader Library.
- **Texture Viewer.** View and manage textures.
- **The Render Panel.** View a 3D rendering of your scene.
- **ShaderPerf Panel.** Analyze and tune shader performance.
- **Assets Panel.** See all the assets in your project, organized by asset.
- **Project Explorer.** Organize assets across multiple COLLADA documents.
- **Task List.** See compilation errors and warning messages.
- **Output Panel.** Displays any output from FX Composer.
- **Scripting.** Access powerful scripting features via a console.
- **Plug-in Explorer.** Manage FX Composer's plug-ins.

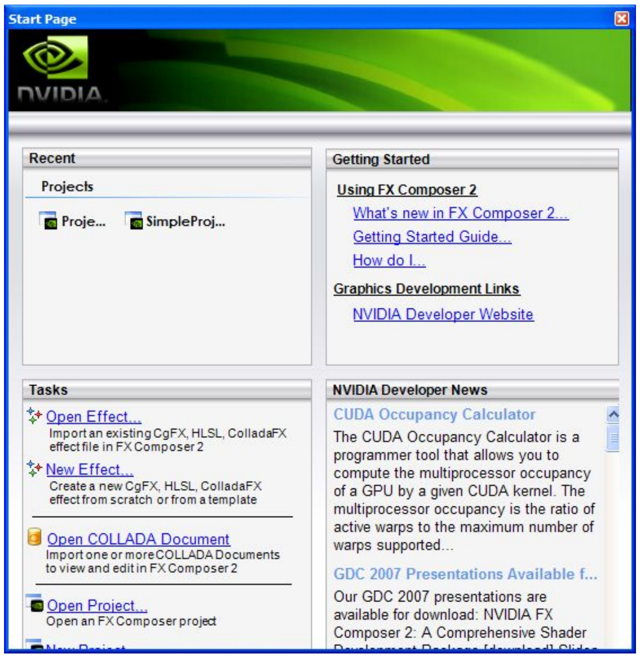
Start Page

The Start Page (see Figure 4) is the first panel you'll see when you start FX Composer. It will help you quickly start on several typical tasks, such as:

- Creating a new project
- Opening an existing project
- Creating a new effect
- Opening an existing effect
- Importing COLLADA documents
- Viewing this Getting Started Guide

- Visiting the NVIDIA Developer Web Site

In addition to this, the Start Page contains an RSS feed of the NVIDIA developer news which allows you to be kept up to date on the last developer technologies and materials at NVIDIA.



Import an existing CgFX, HLSL, ColladaFX effect file in FX Composer 2

Create a new CgFX, HLSL, ColladaFX effect from scratch or from a template

Import one or more COLLADA Documents to view and edit in FX Composer 2

Open an FX Composer project

CUDA Occupancy Calculator

The CUDA Occupancy Calculator is a programmer tool that allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported...

GDC 2007 Presentations Available f...

Our GDC 2007 presentations are available for download: NVIDIA FX Composer 2: A Comprehensive Shader

Figure 4. FX Composer's Start Page

Working with Shaders

Authoring Shaders

Authoring shaders is arguably FX Composer's primary function. FX Composer 2 provides many ways to do this depending on your needs and level of expertise:

- Drag-and-drop from Shader Library
- Shader Wizard
- Code Editor
- Mental Mill Artist Edition

Shader Library Panel

The fastest way to author a shader is to use the NVIDIA Shader Library, which is integrated into FX Composer (see Figure 5). Each shader in the library is complete with a screenshot, summary, category, and versions in various languages. To apply a shader to an object in your scene, simply drag-and-drop the shader onto the object in the Render panel. FX Composer will automatically create a new material for you using that shader. You can also drag-and-drop shaders onto the Materials panel if you don't want to immediately apply them to objects.

If you would like to share your own shaders with other developers, please submit it to the Shader Library by following the directions given here:

http://developer.nvidia.com/object/shader_submission_guidelines.html

Shader Library Preferences

You can set various options for the Shader Library via the **Settings…** option in the Tools menu. In the Settings dialog box, click on Environment -> Preferences -> Shader Library and you will see several options for the shader library:

- **Add to current project on download.** Any downloaded shader will automatically be added to the current project. Otherwise, the shader will only be stored on the hard disk.
- **Load in editor on download.** Any downloaded shader will automatically be opened in the code editor.
- **Show download dialog boxes.** Displays a confirmation dialog box (showing the location of the file on the hard disk) whenever a download completes.

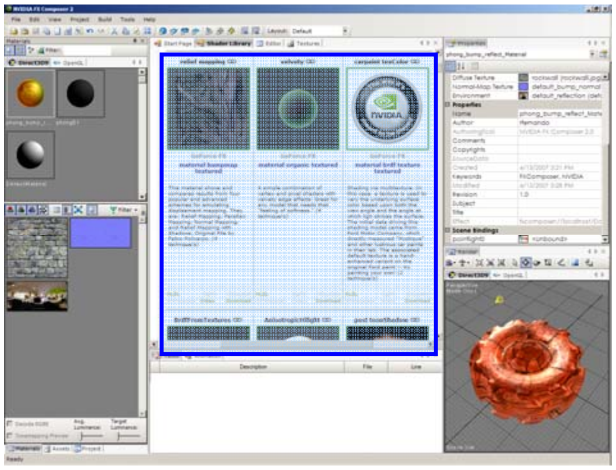


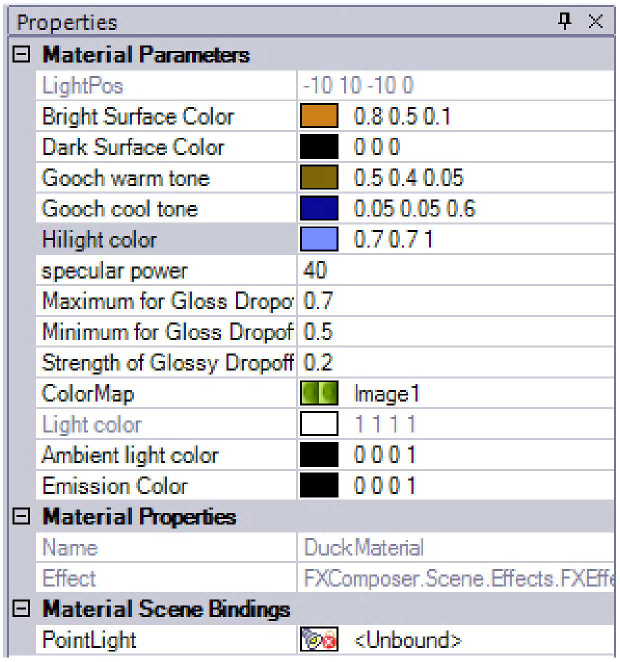
Figure 5. The NVIDIA Shader Library in FX Composer

Properties Panel

Use the Properties panel (Figure 6) to view and change object properties—primarily material properties. For example, you can use Color Picker (Figure 7) to adjust a Phong material's diffuse color, specular color, or specular exponent, or you can modify other values via sliders or keyboard entry. Pressing the left and right arrow keys while modifying a slider will increment or decrement the current value by one step. The step size is specified in the `UIStep` shader annotation for that parameter.

Note: Vector and matrix values are separated by spaces.

The Properties panel can also be used to view/change shapes (for procedural shapes such as teapots), textures, and other items in the scene graph such as lights (colors, positions, spotlight cutoff, and other properties).



Properties

Material Parameters	
LightPos	-10 10 -10 0
Bright Surface Color	🟧 0.8 0.5 0.1
Dark Surface Color	⬛ 0 0 0
Gooch warm tone	🟫 0.5 0.4 0.05
Gooch cool tone	🟦 0.05 0.05 0.6
Hilight color	🟦 0.7 0.7 1
specular power	40
Maximum for Gloss Dropo	0.7
Minimum for Gloss Dropof	0.5
Strength of Glossy Dropoff	0.2
ColorMap	🟩 Image1
Light color	⬜ 1 1 1 1
Ambient light color	⬛ 0 0 0 1
Emission Color	⬛ 0 0 0 1
Material Properties	
Name	DuckMaterial
Effect	FXComposer.Scene.Effects.FXEffe
Material Scene Bindings	
PointLight	<Unbound>

Figure 6. Properties Panel

Material Scene Bindings

In FX Composer, properties can be bound to elements of the scene—for example, the light color inputs of a Phong material could be tied to a light in the scene. This is called "material scene binding." In these cases, those properties will be inaccessible through the material because they are tied to the light. If you want to modify them, you must do so by editing the light's properties. Any bound objects will be listed in the Material Scene Bindings portion of the Properties panel.

You can also assign scene bindings at an even finer grain—the material instance level. For example, you can have two objects that share the same material, but you can bind each material to a different light.

To change scene bindings at the material instance level, expand the "Scenes" portion of the Assets Panel until you see the material instance you want to work on. You can then change the scene binding for that material instance in the Properties panel.

In particular, if you want to bind a light to a particular material, simply drag-and-drop the light onto the material in the Render panel.

Color Picker

FX Composer's Color Picker supports high-dynamic range and allows you access to the full range of floating point values. The Color Picker shows individual color channels, as well as an overall exponent (represented by the vertical slider on the right of the Color Picker).

Adjust the value of any slider by left-clicking or right-clicking on it. Left-clicking changes the mantissa value (indicated by a black arrow), while right-clicking changes the exponent value (indicated by a gray arrow).

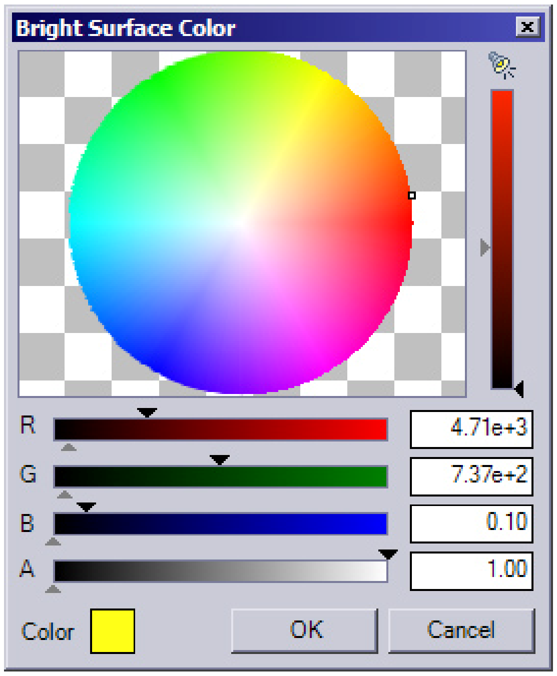
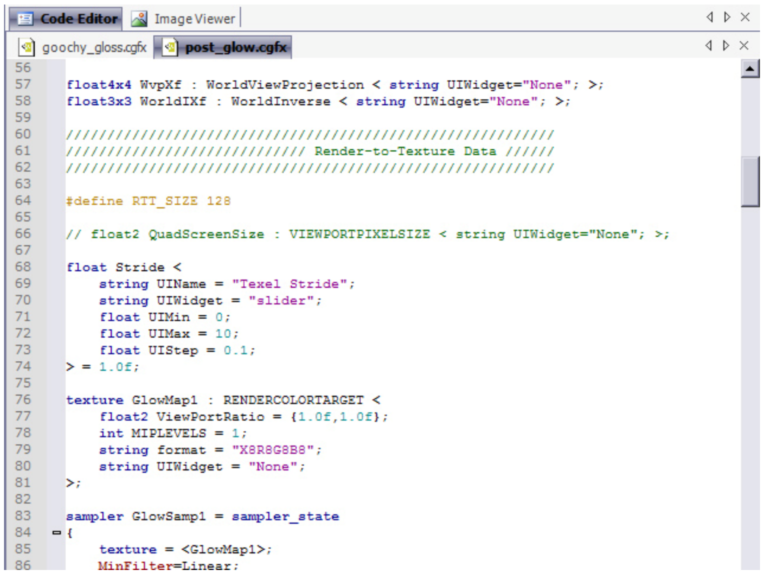


Figure 7. FX Composer's HDR Color Picker

Code Editor

The Code Editor (Figure 8) lets you view and modify shader source code. The editor features all the standard editing functionality you would expect, such as:

- Load/save files
- Cut/copy/paste text
- Find/replace in text and in files
- Undo/redo
- Code folding for functions, techniques, and passes
- Compile effects
- Display tab stops
- Line numbers
- Monitor external file modifications
- Highlight errors for source code and included source code
- Jump to errors by double-clicking



```

float4x4 WvpXf : WorldViewProjection < string UIWidget="None"; >;
float3x3 WorldIXf : WorldInverse < string UIWidget="None"; >;

///////////////////////////////////////////////////////////
///////////////////////////// Render-to-Texture Data //////
///////////////////////////////////////////////////////////

#define RTT_SIZE 128

// float2 QuadScreenSize : VIEWPORTPIXELSIZE < string UIWidget="None"; >;

float Stride <
    string UIName = "Texel Stride";
    string UIWidget = "slider";
    float UIMin = 0;
    float UIMax = 10;
    float UIStep = 0.1;
> = 1.0f;

texture GlowMap1 : RENDERCOLORTARGET <
    float2 ViewPortRatio = {1.0f,1.0f};
    int MIPLEVELS = 1;
    string format = "X8R8G8B8";
    string UIWidget = "None";
>;

sampler GlowSamp1 = sampler_state
{
    texture = <GlowMap1>;
    MinFilter=Linear;
```

Figure 8. The Code Editor

FX Composer allows you to right-click on files referenced by "#include" statements to jump directly to their source code according to the source include path configuration.

You can open multiple files simultaneously in Code Editor:

- Each file appears in a separate tab. You can toggle between the tabs with CTRL+TAB key combinations
- Read-only files are indicated by a lock icon.
- Modified and unsaved files have an asterisk suffixed to their names.

The Build menu is contextual, reflecting the current shader file, and it also includes a Rebuild All option to recompile all shaders in the current project.

In addition, the Tool menu includes an option to "Analyze Shader Performance," which runs NVIDIA ShaderPerf on the current shader. For more details, refer to "Shader Performance Panel" further on in this document.

Code Editor Settings

Code Editor Settings can be configured in the Settings dialog that can be accessed via the main menu->Tool->Settings. The Preference section has a sub-section dedicated to the Code Editor. Code folding can be toggled on/off by setting the EnableCodeFolding property accordingly.

Mental Mill Artist Edition

Bundled with FX Composer 2, Mental Mill Artist Edition offers a convenient visual shader authoring system. In Mental Mill Artist Edition, you can quickly connect different modules to create new shaders – an ideal approach for artists and technical directors who may not want to work directly with shader code.

Shaders can then be exported to CgFX or HLSL, and then loaded in FX Composer's production-friendly development environment as shown in Figure 1.

For more information, please visit http://www.mentalimages.com/2_4_mentalmill/index.html.

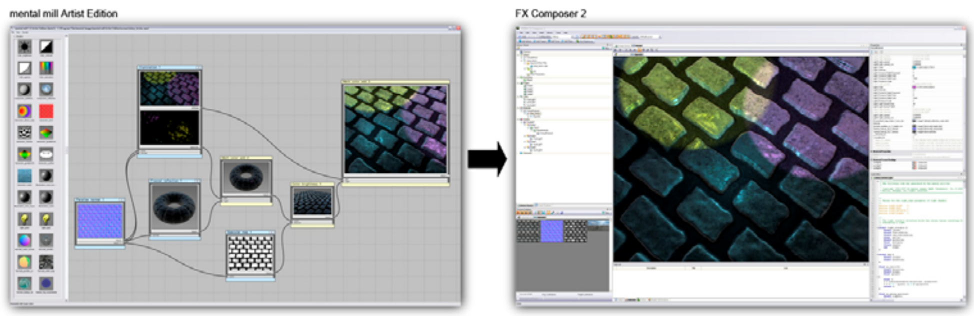
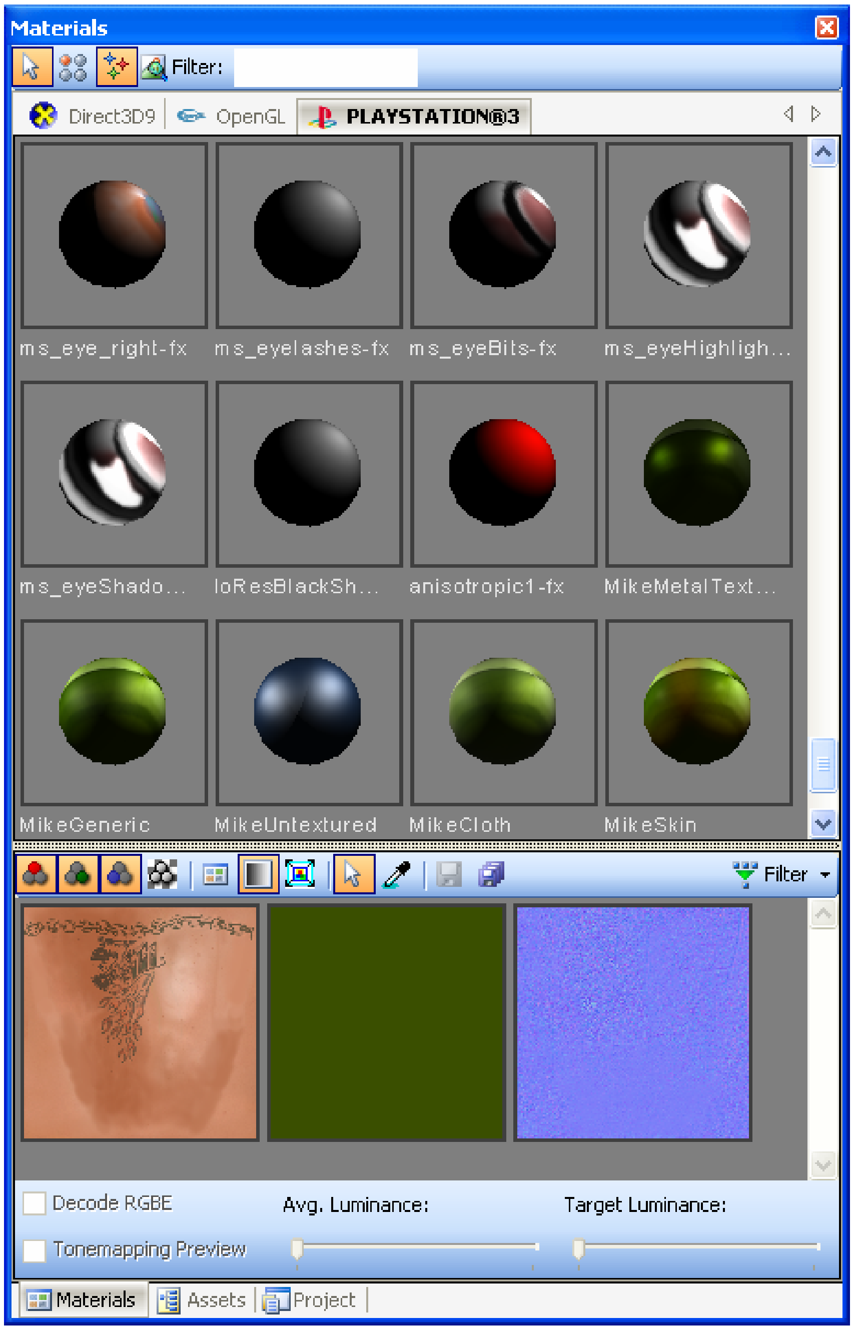


Figure 9. Exporting a Shader from Mental Mill Artist Edition to FX Composer 2

Material Panel

FX Composer provides the user with a Material Panel for visualizing them before they are applied to 3D objects.

The Material panel tab contains two sections. The top section lists all the materials and effects, and the bottom section lists all the Textures referenced by the materials.



Each Material swath lives in a device-specific tab that shows shader balls with the material applied to it and its name underneath it. The Panels can be detached or tiled to allow for a side by side comparison as to how certain device-specific implementations may differ.

The Material panel offers a set of controls that allows the user to interact with the swaths:

- Left Mouse click to select a Material/Effect
- Left Mouse + ALT (just like the scene panel) to rotate the shader balls.
- Mouse Wheel + CTRL on an item to vary the size of the swatches.

In addition to these controls, you can toggle the preview of Materials versus Effects by clicking Material and Effect buttons respectively, on the Pane's toolbar.

Each Material swath behaves just like a Material or an Effect node from the Assets Panel. This means that right-click on the selected swath will bring up the usual menu of operations (Apply To Selection, Assign Effect, rename, delete, clone, etc…). Furthermore, you can drag and drop the selection onto any geometric object in the 3D scene panel to assign the selected material or effect.

Finally, selecting a swath will bring its properties in the Properties panel as one would expect.

The Texture section displays all the textures used in the currently selected material, including procedurally generated textures and render targets. The Texture section also enables visualization of cube maps and normal maps. You can use the mousewheel to scroll, and CTRL + mousewheel to zoom in and out of textures. Left-clicking on a texture will display the texture's information in the Properties panel.

More advanced controls are available to allow shader writers to investigate shading problems that could be caused by texturing issues. All these controls are available on the Texture section's toolbar.

The Texture panel provides you with a color picker mode that allows you to inspect the texel value of the texture. If the texture is much bigger that the size of its rendering in the Texture Panel, you can use the "Show Items Actual Size" in order to force the rendering of the texture to their original size. This will effectively force a 1 to 1 mapping of pixels and texels.

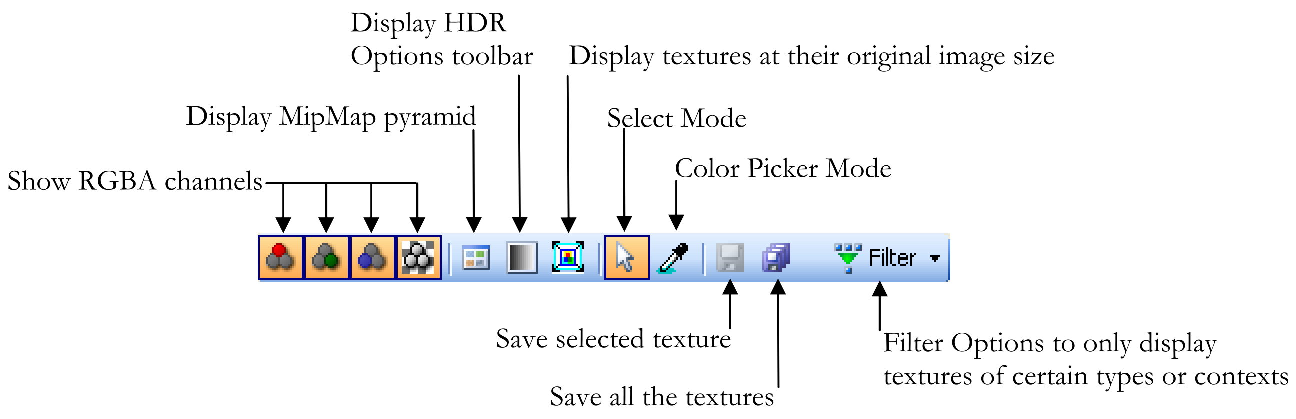

Display HDR Options toolbar
Display textures at their original image size
Display MipMap pyramid
Select Mode
Show RGBA channels
Color Picker Mode
Filter
Save selected texture
Filter Options to only display textures of certain types or contexts
Save all the textures

Figure 10: Toolbar of advanced display controls and options of the Texture section of the Material panel

You can toggle on and off each color channel. By default, the alpha channel is used to blend the texture onto a checker pattern. Toggling off the alpha channel button on the toolbar will draw the texture RGB values only. Compare the rendering of the textures in **Error! Reference source not found.** to see the effect of toggling alpha on or off.

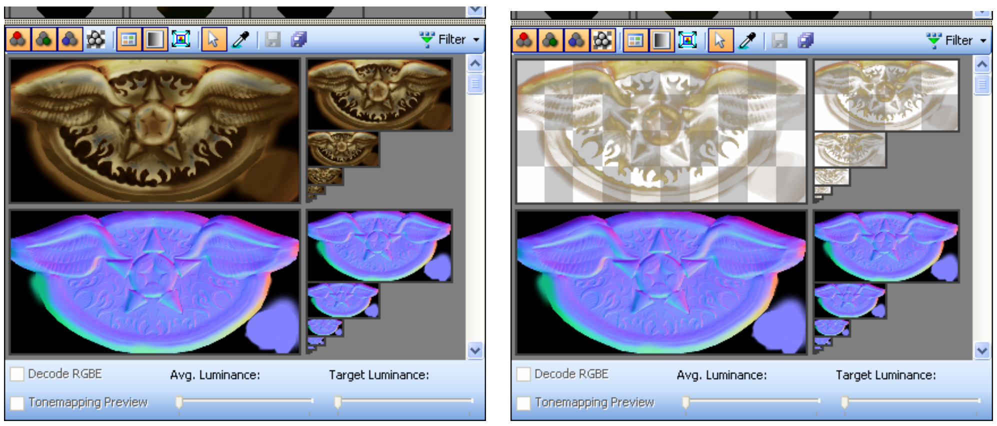


Figure 11: Texture Panel with the alpha channel turned off and on

The same controls are available for the red, green, and blue color channels. In addition to these toggles, you can display the mipmap hierarchy of the textures. This is useful to see how certain filtering modes or effects based texture LOD bias end up being affected by certain mipmap levels.

FX Composer supports high dynamic range image file formats like .hrd and .exr. In order to help you manipulate these images more comfortably, the Texture Panel has a set of advanced HDR controls located in the HDR toolbar.

When the HDR toolbar is displayed, the controls will be activated upon the selection of an HDR texture.

The figure below shows an example of an OpenEXR (.exr) image with a 16-bit float RGBA format. You will notice that when the Tonemapping Preview is enabled, you can tweak the exposure by moving the Target Luminance slider.

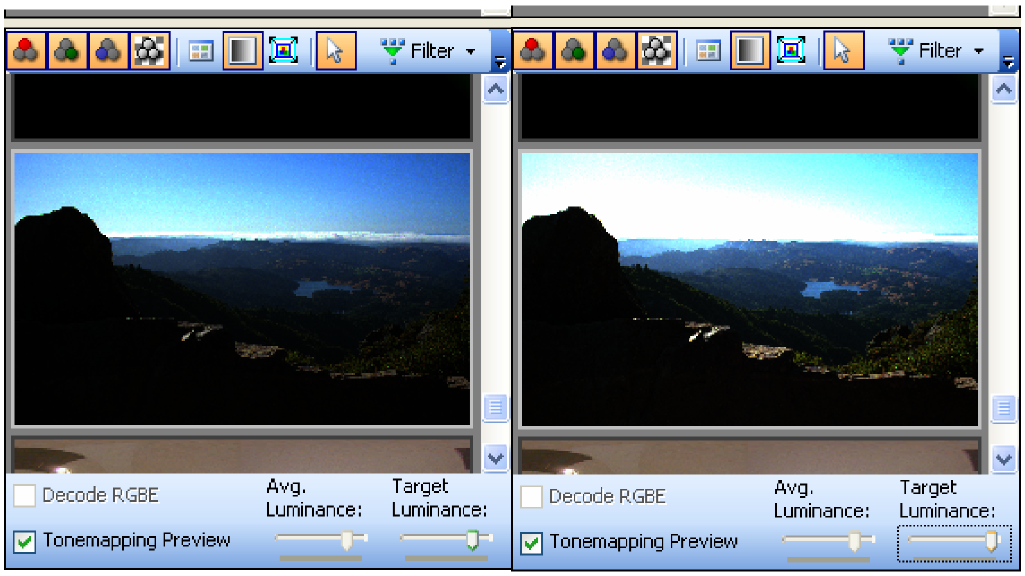


Figure 12. HDR Controls for an OpenEXR texture (Low exposure on the left versus high exposure on the right)

Another common format for HDR images is RGBE. This formats store a common exponent to the RGB values such that final HDR RGB values can be evaluated in the shader code. The HDR toolbar has a Decode RGBE toggle that will convert the RGBA textures to RGBE.

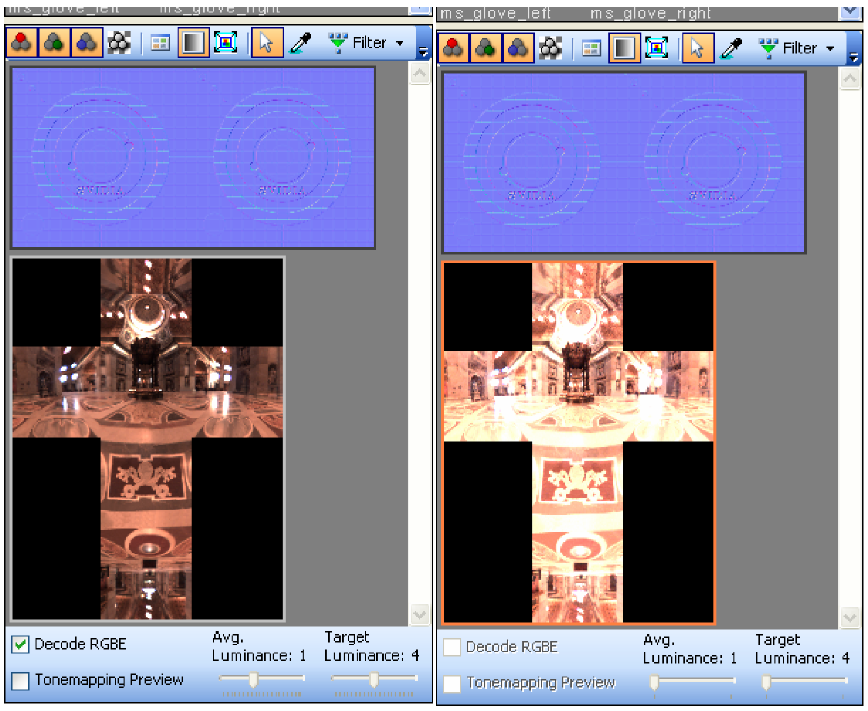
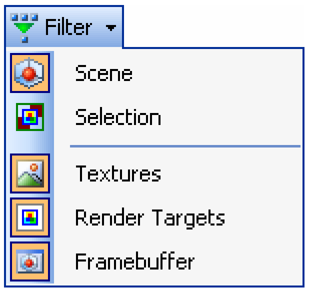


Figure 13. HDR Controls for a .HDR (RGBE) texture (Low exposure on the left versus high exposure on the right)



Finally, the Texture Panel allows you to restrict the list of textures being displayed. You can either specify that you want see all the textures within the current scene or only the textures within the current material selection. This allows the user to focus on the textures of interest. Furthermore, you can restrict the display of textures in the texture panel by enabling certain types of textures:

- Textures
- Render Targets
- Framebuffer

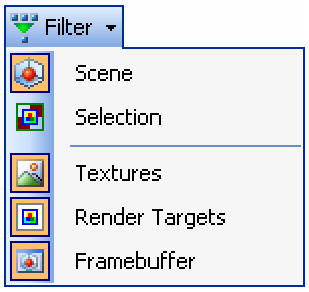


Figure 14: Filter menu for displaying certain types of textures or textures from the current scene or selection

Texture Viewer

FX Composer's handy Texture Viewer helps you navigate through the images in your project (Figure 15). By default, Texture Viewer arranges thumbnails of all your images neatly in rows. Each thumbnail is accompanied by the image's file name and resolution.

The Texture Viewer has a regular expression filter box that allows you to narrow down the textures displayed. This feature makes it easy to find specific textures of specific filename or sizes.

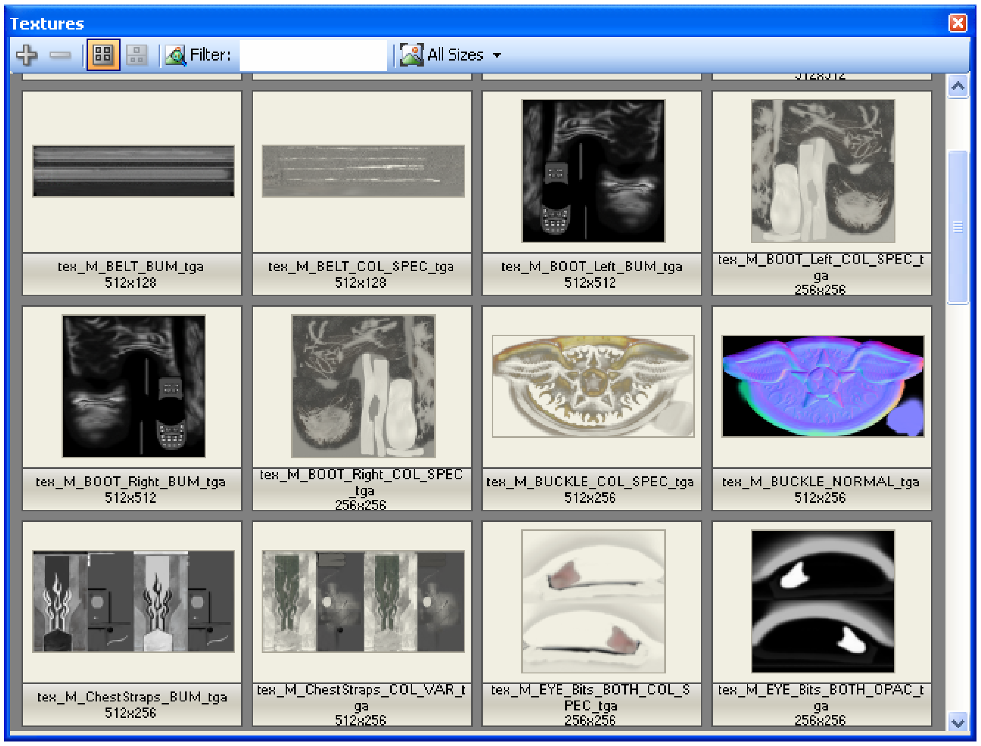

Textures
Filter:
All Sizes
tex_M_BELT_BUM_tga 512x128
tex_M_BELT_COL_SPEC_tga 512x128
tex_M_BOOT_Left_BUM_tga 512x512
tex_M_BOOT_Left_COL_SPEC_tga 256x256
tex_M_BOOT_Right_BUM_tga 512x512
tex_M_BOOT_Right_COL_SPEC_tga 256x256
tex_M_BUCKLE_COL_SPEC_tga 512x256
tex_M_BUCKLE_NORMAL_tga 512x256
tex_M_ChestStraps_BUM_tga 512x256
tex_M_ChestStraps_COL_VAR_tga 512x256
tex_M_EYE_Bits_BOTH_COL_SPEC_tga 256x256
tex_M_EYE_Bits_BOTH_OPAC_tga 256x256

Figure 15. Texture Viewer

You can drag-and-drop any image onto a texture reference in the Properties panel. While dragging you'll see a small preview of your texture, and the field that you are dropping to will be highlighted. You can also right-click on any image to bring up a context menu with several options for adding new images, adjusting thumbnail size, choosing among layouts, and viewing/editing images in external applications.

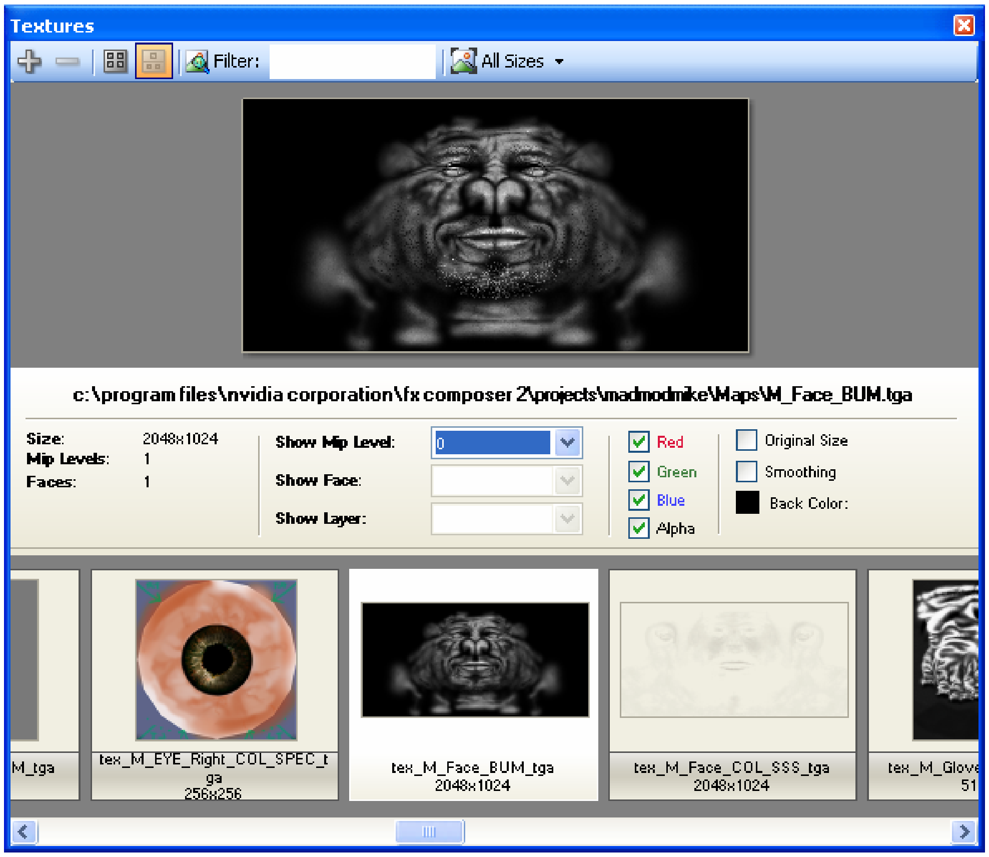
In addition, you can drag-and-drop any image onto an object in the Render panel. If that object's material can accept textures, you'll be prompted for the texture to replace.

The Texture Viewer toolbar allows you to add/remove textures to the project and toggle between the thumbnail and advanced views. The advanced view (Figure 16) gives an in-depth view of your image, including mipmap information, number of faces, and texture format. By checking the "Original Size" check box, you can see your image at its original resolution (and pan around the image if it is larger then the viewing area). You can also view color and alpha channels individually and set the background color for images with transparency.

Right-clicking on an image will show a context menu with the following self-explanatory options:

- **Reset Panning**
- **View in Associated Program…**
- **Edit Image…**

You can specify the external image viewer and editor via the **Settings…** option the Tools menu. The options are listed in the Utilities subsection under Environment.



c:\program files\nvidia corporation\fx composer 2\projects\madmodmike\Maps\M_Face_BUM.tga

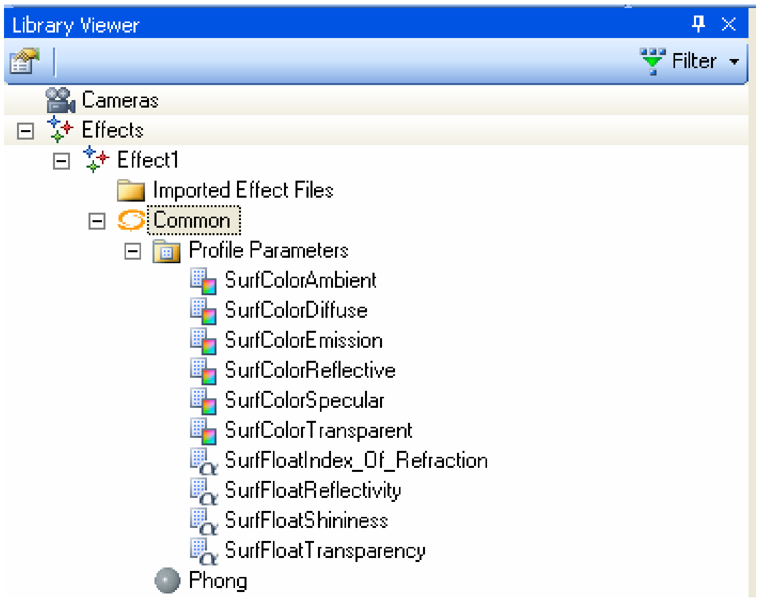
Size: 2048x1024
Mip Levels: 1
Faces: 1
Show Mip Level: 0
Red ✓ Green ✓ Blue ✓ Alpha ✓
Original Size ☐ Smoothing ☐

Figure 16. The Texture Viewer's Advanced View

Creating a COLLADA FX Common Profile

This section describes in great details how to create a new COLLADA FX Common profile effect in FX Composer. The authoring happens in the Assets Panel.

- **In the Effects section, right-click on the divider ->Create Effect…**
  This will create a new empty Effect called "Effect1"
- **Right-click on the new node named Effect1, navigate the context menu hierarchy to "Add Profile->Common->Phong" and select the Phong menu item.**
  This will create a COLLADA FX Common profile effect with a Phong lighting model. The Effect layout should look like Figure 17.

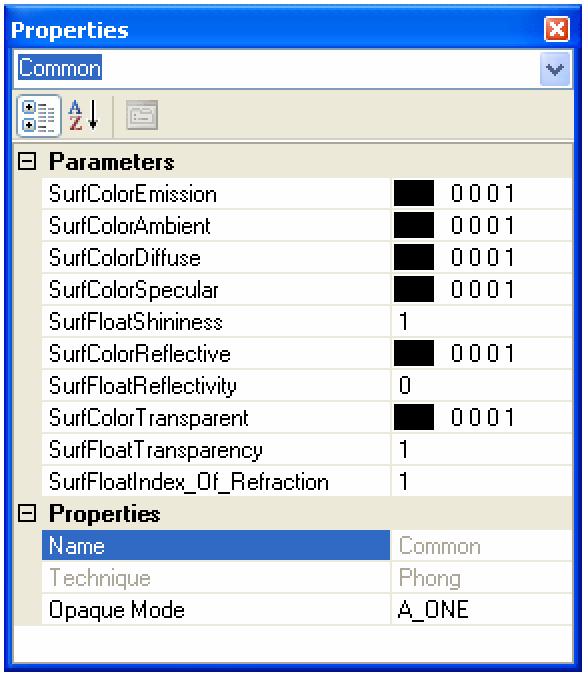


Library Viewer
Filter

```
Cameras
Effects
  Effect1
    Imported Effect Files
    Common
      Profile Parameters
        SurfColorAmbient
        SurfColorDiffuse
        SurfColorEmission
        SurfColorReflective
        SurfColorSpecular
        SurfColorTransparent
        SurfFloatIndex_Of_Refraction
        SurfFloatReflectivity
        SurfFloatShininess
        SurfFloatTransparency
      Phong
```

Figure 17: New COLLADA FX Common profile effect with Phong lighting model

- **Click on the effect profile node named "Common" to see its properties definition in the Properties Panel.**
  The Effect tree control in the Assets Panel shows the profile parameters under the "Profile Parameters" node. Selecting the Effect Profile node allows you to configure the type of the Common profile parameters.

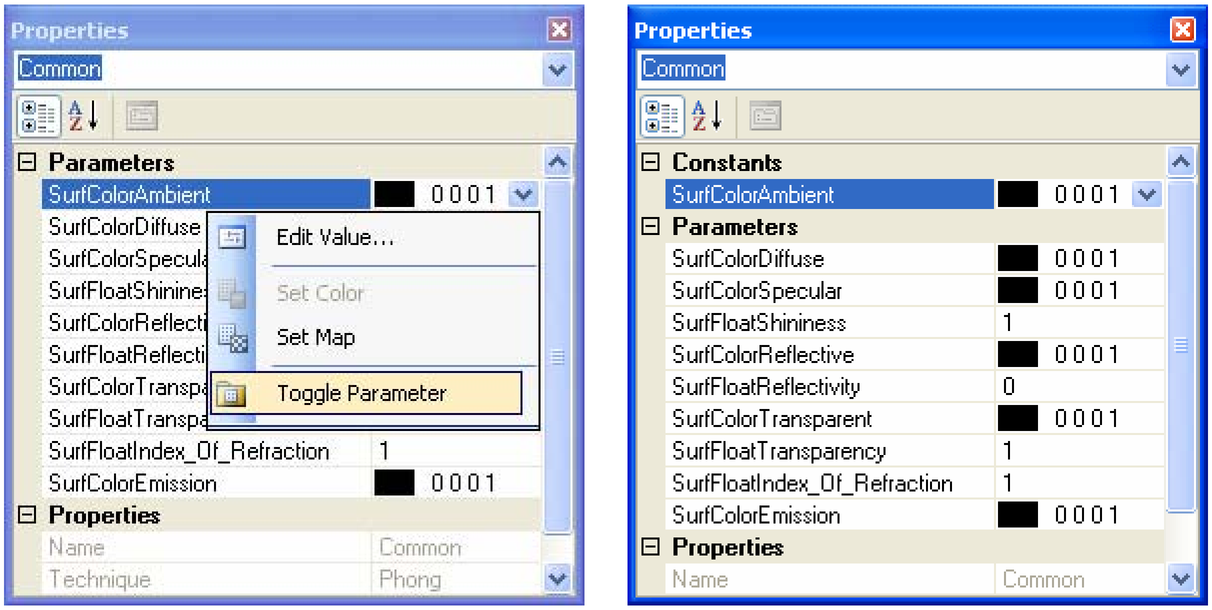


Parameters	
SurfColorEmission	■ 0 0 0 1
SurfColorAmbient	■ 0 0 0 1
SurfColorDiffuse	■ 0 0 0 1
SurfColorSpecular	■ 0 0 0 1
SurfFloatShininess	1
SurfColorReflective	■ 0 0 0 1
SurfFloatReflectivity	0
SurfColorTransparent	■ 0 0 0 1
SurfFloatTransparency	1
SurfFloatIndex_Of_Refraction	1
Properties	
Name	Common
Technique	Phong
Opaque Mode	A_ONE

The Common Profile properties of the effect are separated in two groups:

- **Constants**. These values are only visible at the effect level. Changing a constant value in an effect will affect all materials based on this effect, even those already created.
  These values are not visible at the material level.
- **Parameters.** These values are visible at both the effect and the material level. Changing an effect parameter will not affect already created materials. Only materials that will be created after the change will be affected.
  These values are visible at the material level.

Properties can be configured to be Constant or Parameter. They can be toggled by clicking on "Toggle Parameter" menu option from the property's drop down menu in the Properties Panel.



Parameters	
SurfColorAmbient	■ 0 0 0 1
SurfColorDiffuse	
SurfColorSpecul	
SurfFloatShinine	
SurfColorReflecti	
SurfFloatReflecti	
SurfColorTranspa	
SurfFloatTranspa	
SurfFloatIndex_Of_Refraction	1
SurfColorEmission	■ 0 0 0 1
Properties	
Name	Common
Technique	Phong

Constants	
SurfColorAmbient	■ 0 0 0 1
Parameters	
SurfColorDiffuse	■ 0 0 0 1
SurfColorSpecular	■ 0 0 0 1
SurfFloatShininess	1
SurfColorReflective	■ 0 0 0 1
SurfFloatReflectivity	0
SurfColorTransparent	■ 0 0 0 1
SurfFloatTransparency	1
SurfFloatIndex_Of_Refraction	1
SurfColorEmission	■ 0 0 0 1
Properties	
Name	Common

Figure 18. Configuration of a COMMON Profile parameter to be Constant

- **Changing the type of a COMMON Profile parameter**

Some common profile parameters can be configured to be either a texture or a color. To change their type, bring up the property's drop down menu in the Properties Panel and click on "Set Map" or "Set Color".

- **Editing a COMMON Profile parameter's default value**

To edit the Default value, click on "Edit Value" menu item of the property's drop down menu in the Properties Panel. This action will display an Image Selector window or a Color Picker, depending on the type of parameter.

- **Modifying a Material based on a Common Profile Effect.**

Once a material is assigned a common profile effect, it has a set of visible parameters based on the effect parameters. To change their values, click on the arrow next to each parameter.

Note: Note that it is **not** possible to change the type of a parameter at the material level.

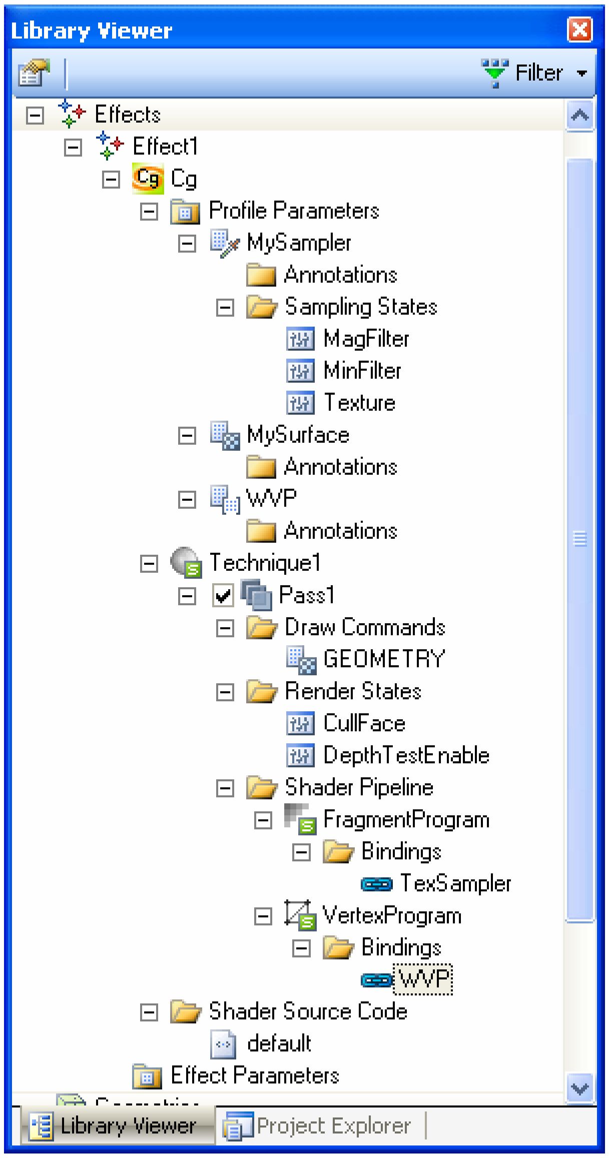
Creating Various Types of Shaders

This section describes how to create shaders in various programming languages.

Creating a COLLADA FX Cg Effect

This section describes how to create a new COLLADA FX Cg profile effect in FX Composer. The authoring process starts in the Assets Panel.

- In the Effects section, right-click on the divider ->Create Effect…
  This will create a new empty Effect called "Effect1"
- Right-click on the new node named Effect1, navigate the context menu hierarchy to "Add Profile->Cg" and select the "Cg" menu item.
  This will create a default COLLADA FX Cg profile effect that does vertex diffuse lighting modulated by a diffuse texture. The Effect layout should look like Figure 19.



```
Library Viewer
Filter
Effects
  Effect1
    Cg
      Profile Parameters
        MySampler
          Annotations
          Sampling States
            MagFilter
            MinFilter
            Texture
        MySurface
          Annotations
        WVP
          Annotations
      Technique1
        Pass1
          Draw Commands
            GEOMETRY
          Render States
            CullFace
            DepthTestEnable
          Shader Pipeline
            FragmentProgram
              Bindings
                TexSampler
            VertexProgram
              Bindings
                WVP
      Shader Source Code
        default
    Effect Parameters
Library Viewer | Project Explorer
```

Figure 19: Default COLLADA FX Cg profile effect

In this tutorial, we want to modify the default COLLADA FX Cg effect to turn it into a per-pixel Phong lit effect.

Before we get started, let's create a helper mesh object to apply our effect onto so that that we can visualize the different steps of writing our improved shader. Go to the Main menu -> Library -> Add Geometry->Add Teapot to create a new teapot. Next, drag and drop Effect1 on to the teapot in the 3D scene panel. Now we can start improving the default shader.

So the first step is to move the diffuse lighting from the vertex stage to the fragment stage:

- We need to pass the surface normal to the fragment shader stage. This requires modification of the vertexOutput connector to pass through the vertex normal to the fragment shader. Add the following member to the vertexOuput structure:

  ```
  float3 Normal : TEXCOORD1;
  ```

  Next, remove the diffCol member from the vertexOuput connector and add the following statement at the end of the MainVS function:

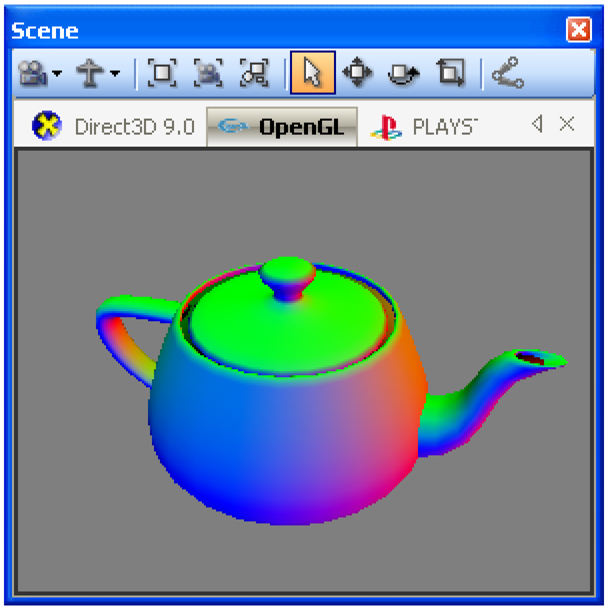
  ```
  OUT.Normal = IN.Normal;
  return OUT;
  ```

  Finally, the line of Cg code used to compute diffCol in MainVS needs to be taken out.

  Now the fragment shader is set to access the surface normal. You can visualize the vertex normals, interpolated across the triangles by making the MainFS function return the following:

  ```
  return float4(IN.Normal,0);
  ```

  Hit Ctrl+F7 and you should see something like this:



Now that we have the surface normal at the fragment stage, we can compute the diffuse lighting term per pixel.

Delete the following line of code in MainFS:

```
   Return IN.diffCol * tex2D(TexSampler, IN.Tex);
```

Or, if you have visualized the normals previously

```
   return float4(IN.Normal,0);
```

And type the following:

```
   Float4 diffCol = dot(IN.Normal, float3(0.0,0.0,1.0)).xxxx;
   Return diffCol * tex2D(TexSampler, IN.Tex);
```

This essentially computes an N dot L term where the light vector is fixed as +Z. (We probably should renormalize the normal per pixel to get a normal vector of unit 1…but we don't have to.)

Now, we need to add a specular term. In order to do so, we need to compute the reflection vector per pixel. Therefore, we need to pass the eye vector and the light vector in the same space. Instead of pass the normal vector in object space, we will transform it into World space and do all the diffuse and specular lighting computation in World space. So first, let's add a matrix to transform our normal into World Space, by adding the following statement at the top of the source code:

```
float4x4 WorldITXf;
```

Next, we need to transform the normal with this matrix in the vertex shader by replacing this statement

```
   OUT.Normal = IN.Normal;
```

by:

```
   OUT.Normal = mul(WorldITXf, float4(IN.Normal,0)).xyz;
```

If you compile the shader, you will get a black image…this is normal. The reason why is that you need define to COLLADA FX Cg what WorldITXf means. For this, you need to create a shader binding and a profile parameter with the correct semantic.

Go to the vertex program of the ShaderPipeline of Effect1/Technique1/Pass1, and right-click on the Bindings folder -> Add -> WorldITXf.

Next, create a profile parameter by right-click on the Profile Parameters under the Cg profile node and select Float4x4. This will create a new profile parameter of type float4x4 named MyFloat4x4. Select the node and press F2 to rename the node to: WorldITXf.

Lastly, you need to type in the semantic of the matrix – select the WorldITXf profile parameter and edit its Semantic in the Properties Panel by typing: WorldInverseTranspose. After you hit enter, you should see the teapot lit again.

Next we need to do a similar operation to transform the vertices into World space in order to compute a view direction vector in World space. We will add a WorldXf matrix and set its semantic to World. Once this is done, we can add the following code statement in MainVS:

```
float3 Pw = mul(WorldXf, float4(IN.Position.xyz,1)).xyz;
OUT.WorldView = normalize(ViewITXf[3].xyz – Pw);
```

Let's not forget to add the following new member to the vertexOutput connector:

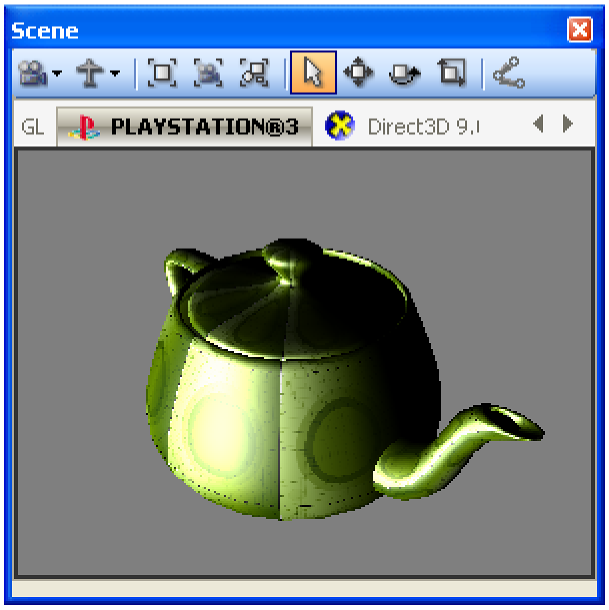
```
float3 WorldView :  TEXCOORD2;
```

We are now finally ready to tackle the phong lighting computation per-fragment.

Replace the code of the MainFS function with the following Cg code:

```
float3 Nn = normalize(IN.Normal);
float3 Ln = float3(0,0,1);
float ldn = dot(Ln,Nn);
float3 Vn = normalize(IN.WorldView);
float3 Hn = normalize(Vn + Ln);
float hdn = dot(Hn,Nn);
float4 litV = lit(ldn,hdn,12);//SpecExpon
ldn = litV.y;
hdn = ldn * litV.z;
float4 SurfColor = tex2D(TexSampler, IN.Tex);
float3 result = ((ldn * SurfColor.xyz ) + hdn);
return float4(result.xyz,1.0);
```

This code basically implements the basic Phong lighting model by normalizing the Normal (define in World space), assigning the light direction vector (Ln) to a constant value of +Z, doing a dot product of the light vector and the surface normal in WorldSpace to compute the diffuse term in ldn. It then normalizes the view direction vector (Vn), compute the half-angle vector hdn and use it with the diffuse term ldn to use the Lit function to store the result of the lighting. The diffuse term is then modulated with a surface color texture and the specular term is added. You should see the following image in FX composer 2:



As a final improvement to our shader, we are going to enable the light direction to be bind able to a real light object from a scene.

Let's add a new uniform variable to the shader at the top of the Cg code:

```
float3 LightDir;
```

And replace our +Z constant light direction computation with the following expression

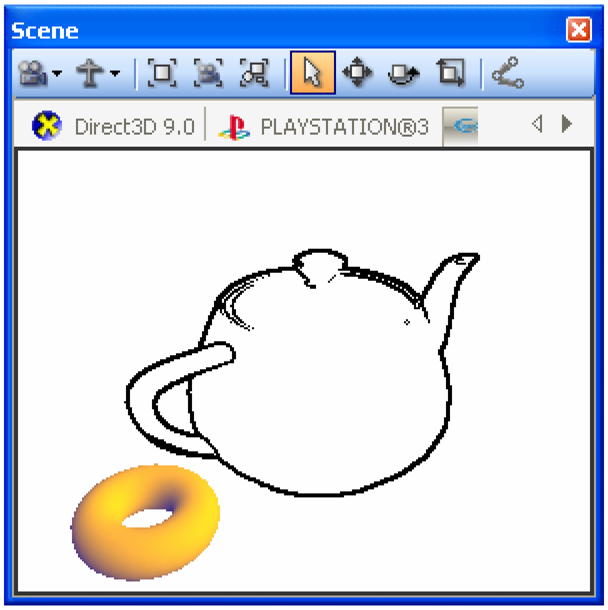
```
float3 Nn = -normalize(LightDir);
```

Next, we need to bind this variable to a profile parameter. Let's create the Fragment Program binding by right-clicking on the Bindings folder of the FragmentProgram node and select Add -> LightDir. This will create the LightDir binding. Next, you need to create the float3 profile parameter and rename it LightDir. In order to allow for FX Composer 2 to feed the proper data to the LightDir variable, we need to add a Direction semantic to let it know that it needs to compute Direction vector and a set of annotation to qualify what Cartesian space the direction vector has to be computed. In order to achieve this, you need to add a String annotation of name Space and set its value to World.

You can also add a UIName string annotation to display a more artist friendly text in the Properties panel when artist would be using the shader.

Creating a Fullscene COLLADA FX Effect

This section describes how to create a new Fullscene COLLADA FX Cg profile effect and apply it within a simple scene. The goal of our composition is to apply an EdgeDetect filter onto a Layer of the scene and composite it with another Layer that does have any fullscene effect.



Before we get started with the creation of the effect, we need to create a simple scene to visualize our work. For this, create a teapot and a torus using the Library -> Geometry menu after having created a new project.

Next, create a new Effect by importing a .CgFX like Goochy_gloss.cgfx. Apply this effect to the objects in the scene by drag and dropping the effect onto the 3d object in the scene panel. At this point, the test scene should look like Figure 20.

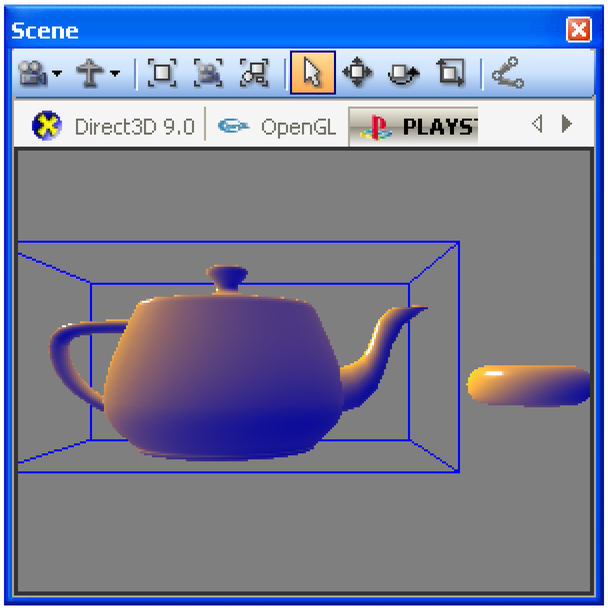
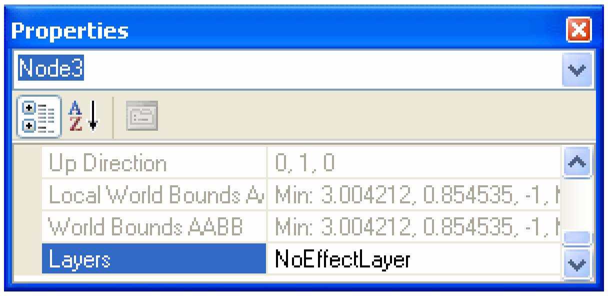


Figure 20: Test scene with Goochy_gloss.cgfx effect applied

Next, we want to leverage the Layer system to separate these two objects and allow us to apply the fullscene effect only to one object and not the other. For this, select the teapot and go the Assets Panel-> Scene control to select its node. Go to the Properties Panel and type in the Layers parameter: EffectLayer. Do the same with the torus node and type in NoEffectLayer.



Up Direction	0, 1, 0
Local World Bounds A	Min: 3.004212, 0.854535, -1,
World Bounds AABB	Min: 3.004212, 0.854535, -1,
Layers	NoEffectLayer

We now have a scene with two objects that belong to two different layers. We are ready to create the fullscene effect.

Create a new empty effect in the Assets Panel and add a COLLADA FX Cg profile. Lets rename the Effect: "ColladaFS Effect". Let's also create a new material in the Assets Panel, named "ColladaFS Material" and have it reference "ColladaFS Effect".

Now, we need to go back to the Effect and start adding the necessary pieces to create our Fullscene COLLADA FX Cg.

We have to add the following profile parameters:

- Surface for the render target
- Sampler for the sampling states of the render target
- Depth surface for the Z and Stencil buffer of the render target

- 2D vector to hold the dimensions of the rendertarget
- Scalar parameter for setting up the anti-aliasing of the outline of the edge detection effect
- Scalar parameter for setting up the threshold level to detect an edge.

The following section will detail how to create these parameters:

1. Rendertarget Surface
   - Right-click on the profile parameter folder and select create -> Surface.
   - Rename "MySurface" to "SceneTexture" (Press F2 on the node)
   - In the properties panel, set the SurfaceUsage to ColorTarget, IsViewportRatio to TRUE and Format to A8R8G8B8.
   These settings tell the engine that the surface has to be treated as a rendertarget with a texture format of A8R8G8B8 and should inherit its viewport ratio.
2. Sampler of the Rendertarget Surface
   - Right-click on the profile parameter folder and select create -> Sampler.
   - Rename "MySampler" to "SceneTextureSampler"
   - In the properties panel, set the SamplerType to Sampler2D and Texture to "SceneTexture".
3. Depth surface of the rendertarget
   - Right-click on the profile parameter and select create -> Surface.
   - Rename "MySurface" to "DepthSurface"
   - In the properties Panel, set the Surface Usage to DepthTarget, the DepthFormat to D24S8, the SurfaceType to Surface2D and the IsViewportRatio to TRUE.
4. 2D vector for holding viewport size
   - Right-click on the profile parameter and select create -> float2x1.
   - Rename MyFloat2x1 by "QuadScreenSize"
   - In the Properties Panel, set the Semantic to VIEWPORTPIXELSIZE
   This will basically allow FX Composer engine to feed the float 2 vector with the dimensions of the viewport.
5. Scalar control for Anti-aliasing
   - Right-click on the profile parameter and select create -> float
   - Rename the "MyFloat" to "NPixels".
   - In the properties panel, set its default value to 1.2
6. Scalar Control for Edgedetect threshold
   - Right-click on the profile parameter and select create -> float
   - Rename the "MyFloat" to "Threshold".
   - In the properties panel, set its default value to .5

Now that we have all the profile parameters in place, we can implement our 2-pass setup. Our first pass will clear the color and z buffers and draw the scene onto it.

The second pass will source the color buffer and apply an edge detect filter that will be used to draw a full scene quad.

Lets start to implement the edge detect code by filling the Default Shader code node with the following code:

```
float NPixels = 1.5f;
float Threshhold = .2f;
float2 QuadScreenSize;

sampler SceneTextureSampler = sampler_state {
   MagFilter = Nearest;
   MinFilter = Nearest;
};

float getGray(float4 c)
{
   return(dot(c.rgb,((0.33333).xxx)));
}

struct QuadVertexOutput {
   float4 Position       : POSITION;
   float2 UV             : TEXCOORD0;
};


QuadVertexOutput ScreenQuadVS(
          float3 Position : POSITION,
          float3 TexCoord : TEXCOORD0)
{
   QuadVertexOutput OUT;
   OUT.Position = float4(Position, 1);
   OUT.UV = TexCoord.xy;
   return OUT;
}


float4 edgeDetectPS(QuadVertexOutput IN) : COLOR {
   float2 ox = float2(NPixels/QuadScreenSize.x,0.0);
   float2 oy = float2(0.0,NPixels/QuadScreenSize.y);
   float2 PP = IN.UV.xy - oy;
   float4 CC = tex2D(SceneTextureSampler,PP-ox);
   float g00 = getGray(CC);
   CC = tex2D(SceneTextureSampler,PP);
   float g01 = getGray(CC);
   CC = tex2D(SceneTextureSampler,PP+ox);
   float g02 = getGray(CC);
   PP = IN.UV.xy;
   CC = tex2D(SceneTextureSampler,PP-ox);
   float g10 = getGray(CC);
   CC = tex2D(SceneTextureSampler,PP);
   float g11 = getGray(CC);
   CC = tex2D(SceneTextureSampler,PP+ox);
   float g12 = getGray(CC);
   PP = IN.UV.xy + oy;
   CC = tex2D(SceneTextureSampler,PP-ox);
   float g20 = getGray(CC);
   CC = tex2D(SceneTextureSampler,PP);
   float g21 = getGray(CC);
   CC = tex2D(SceneTextureSampler,PP+ox);
   float g22 = getGray(CC);
   float K00 = -1;
   float K01 = -2;
   float K02 = -1;
   float K10 = 0;
   float K11 = 0;
   float K12 = 0;
   float K20 = 1;
   float K21 = 2;
   float K22 = 1;
   float sx = 0;
   float sy = 0;
   sx += g00 * K00;
   sx += g01 * K01;
   sx += g02 * K02;
   sx += g10 * K10;
   sx += g11 * K11;
   sx += g12 * K12;
   sx += g20 * K20;
   sx += g21 * K21;
   sx += g22 * K22;
   sy += g00 * K00;
   sy += g01 * K10;
   sy += g02 * K20;
   sy += g10 * K01;
   sy += g11 * K11;
   sy += g12 * K21;
   sy += g20 * K02;
   sy += g21 * K12;
   sy += g22 * K22;
   float dist = sqrt(sx*sx+sy*sy);
   float result = 1;
   if (dist>Threshhold) { result = 0; }
   return result.xxxx;
}
```

Next, we need to set the Draw Commands of Pass1.

- Right-Click on the Draw Commands folder and a ClearColor, ClearDepth, ColorTarget and DepthTarget. The ColorTarget and DepthTarget's surface parameters need to be set to SceneTexture and DepthSurface respectively.

- Right-click on the Draw Type node of the Drawing Command and set it to SCENE_IMAGE

This pass is all set to render the scene in the rendertarget.

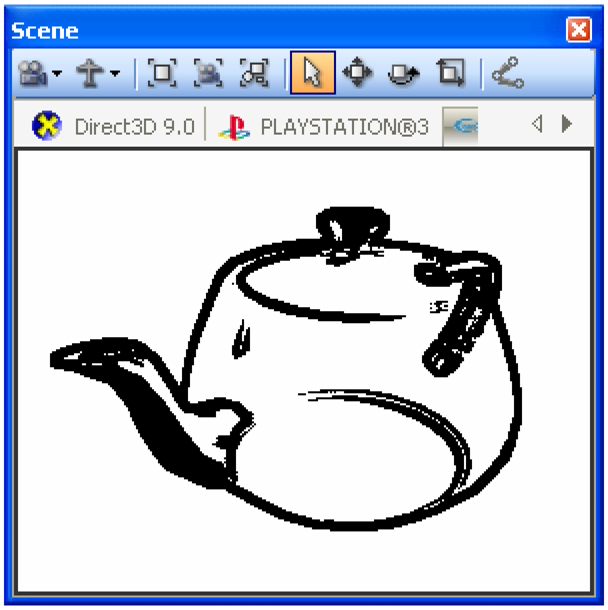
Now, we need to set up the second pass that will draw the fullscene quad. Right-click on technique1 and select Add Pass. Right-click on the Draw Type node of Draw Commands folder and set it to SCENE_SCREEN_QUAD.

We then need to rig the shader entry points to the vertex and fragment program. Select the vertexProgram folder and go to the Properties Panel to set the EntryPoint to "ScreenQuadVS", the Compiler Target to "arbvp1" and the Source to "default". Similarly, select the FragmentProgram folder and go to the Properties Panel to set the EntryPoint to "edgeDetectPS", the Compiler Target to "arbfp1" and the Source to "default".

At last, the FragmentProgram necessary fragment program bindings need to be done. Right-click on the Bindings folder of the FragmentProgram node and Add NPixels, QuadScreenSize, SceneTextureSize and Threshhold to be bound to NPixels, QuadScreenSize, SceneTextureSize and Threshold, respectively, in the properties panel.

The work is done. We just need to set the scene to be rendered with this fullscene material. Go to the scene section of the Assets Panel and select the scene root node. Right-click and select Add Evaluate Scene. Next, select the new EvaluateScene node and right-click to select Add Render. Right Click on the Render node and select Add Layer. Rename the Layer to "EffectLayer". Finally, right-click on the Render node and select Assign Material ->FullScene_Material.

You should see the following.



Toggling on and off the check box off of the the EvaluateScene node will disable the fullscene effect or not.

COLLADA FX Authoring

Collada FX authoring can be achieved in the Assets Panel panel of FX Composer. Under the effects section of the tree, it's possible to author Collada FX using several different 'profiles'. Additionally, the material section enables connection of the effects to objects in the scene, and the scene section allows creation of full screen effects via 'evaluate scene' tree nodes.

Here we'll discuss the various elements in the effect tree, and how to create/edit them. Later we'll cover using the evaluate scene elements.

Assets Panel

To save repetition, it should be noted that most elements in the Assets Panel can be created by right-clicking the folder above the desired location, and choosing 'Create->', followed by the list of options that are possible. Additionally, most elements can be removed by right-clicking them and choosing 'Delete'. If an element can be renamed, there will also be a right-click menu, or the standard shortcut, function key F2, or an additional click with the mouse on the tree node will achieve the same thing

All operations in the Assets Panel (as in the rest of FX Composer) can be undone, by the use of the CTRL+Z keyboard shortcut, or redone using CTRL+Y. This option is also available under the edit menu.

Where this document refers to adding, removing or renaming elements, the above description should be kept in mind.

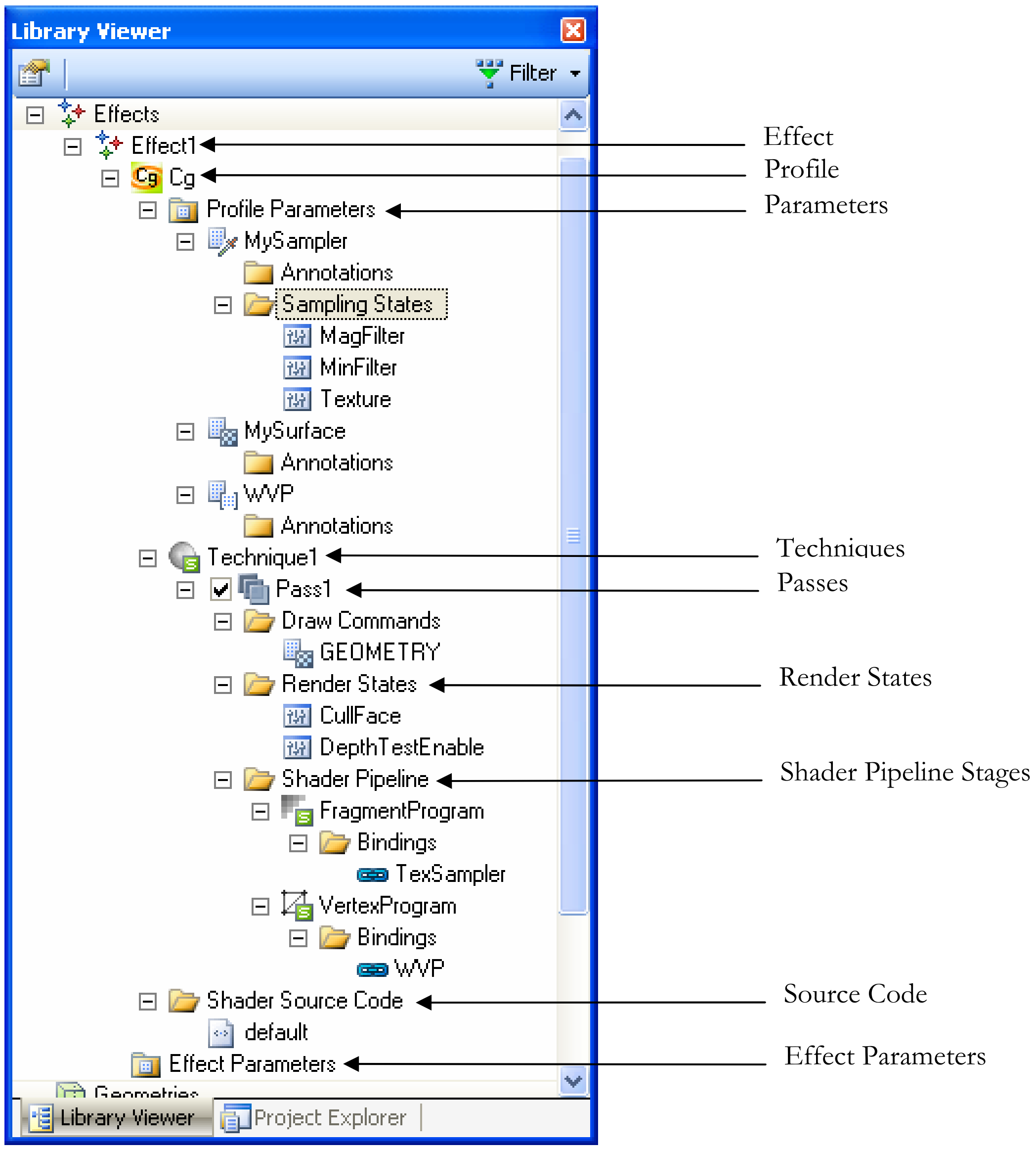


Figure 21: Layout of a COLLADA FX Cg profile effect

Effects

The Figure 21 shows the layout of a default Collada FX, using a Cg profile. The first thing to notice is that an effect named 'Effect1' has been added to the tree. Effects represent the collection of state required to achieve a particular visual effect on screen. They are the 'template' for how the graphics chip should be set up. An effect might be designed to make something look like metal, or wood, or fur, etc. Effects are added to the current scene via materials, which reference an effect, and apply modifications to the effect setup to change its behavior. For example, materials might be created for 'red metal' and 'blue metal', but both materials use the same 'metal' effect; they just modify the colors it uses. There is usually a many-to-one relationship between materials and effects for a given visual look. Effect authors determine how effects will behave, and material authors determine how they will appear in a given scene.

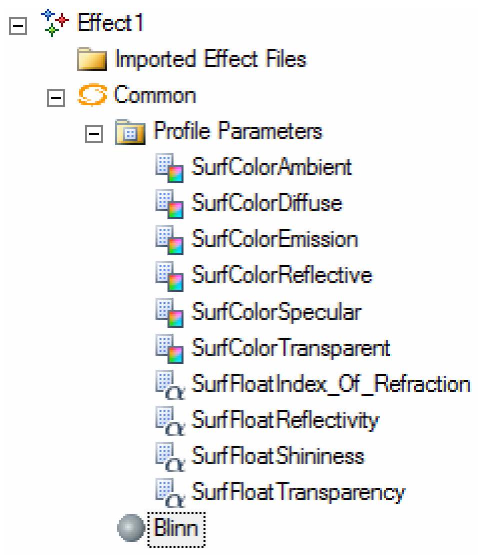
Effects typically contain shader code and render states for the GPU. They are usually authored by a programmer or a technical director with intimate knowledge of a given 3D API. Materials typically contain parameters for control of the shader, and are thus usually tweaked by the artist. The tree approach to authoring Collada FX helps to blur this boundary somewhat, since it is much easier to author effects when selecting from options; but knowledge of shaders and rendering state are still required to author effects well.

Profiles

Profiles describe different ways to realize the same effect. When a device back-end is rendering an effect, it may support one or more profiles with which to draw that effect. There are broadly 2 types of profile – API specific profiles, and the 'Common' profile.

Common Profile

The 'Common' profile is designed to be the most basic shading effect, capable of being rendered on most GPU/API combinations. As such, supplied back-end devices in FX Composer all support the common profile. DCC applications traditionally support this sort of profile as their default shading model. It can be a good idea to add a common profile to an effect, to ensure that even back-ends without profile support in the effect can render an approximation to the required visual. The Common profile supports basic texturing and lighting, and comes in 4 flavors: Blinn, Phong, Lambertian, Constant. Only one common profile is allowed in the effect, so only one of these shading models can be used in a given effect. Any basic reference material on graphics will give details about these different shading models. To add a common profile to an effect, right-click the effect icon and choose 'Add Profile->Common->…'. Once a common profile has been added, it can be selected, and its parameters adjusted just like any other effect. Parameters are covered later. The default parameters for a Blinn-based common profile are shown below. Note that the common profile is indicated by the orange Collada logo. Note also that the default common profile has used color values, but these can be changed to be textures using the properties panel.



```
Effect1
    Imported Effect Files
    Common
        Profile Parameters
            SurfColorAmbient
            SurfColorDiffuse
            SurfColorEmission
            SurfColorReflective
            SurfColorSpecular
            SurfColorTransparent
            SurfFloatIndex_Of_Refraction
            SurfFloatReflectivity
            SurfFloatShininess
            SurfFloatTransparency
        Blinn
```

API specific profiles

Since there are many shading language options, and no easy way to create a common ground between them, Collada supports API specific profiles. These profiles are specified to closely match a given API/Shading language.

In Figure 21, a single profile has been added to 'Effect1' – it is called 'Cg', and represents a Cg language-based shader. An effect can of course contain multiple profiles, enabling it to run with a variety of shader effects. For example, if supported, the user might add an OpenGL-ES specific profile.

For the purposes of explanation, this document describes Cg profile authoring, but the procedure is similar for any given profile – though not necessarily identical. FX Composer supports additional profiles through its plug-in architecture. Specific profiles may add different user interface controls to enable more effective authoring according to the profiles characteristics.

To create a Cg profile, simply right-click the effect and choose 'Add Profile->Cg'. FX Composer will fill in a typical set of state in order to get you started authoring a Cg profile. The default state is sufficient to draw an object in the scene with a texture.

If you need a different default effect, you can create your own by editing the <shaders> section of the \Plugins\Scenes\Profiles\profiles.fxplug xml file.

Profile Parameters

Once a profile has been created, it is possible to add parameters to the profile using the menu on the profile parameters folder. Only parameters that the profile supports can be added, since they are profile specific. Note that an additional 'global' parameters location exists at the effects level, right under the effect; here a common subset of parameters can be added and enable values to be shared across different profiles. Wherever the parameters are declared, they are created in the same way, via the menu. Parameters also have annotations. In the Assets Panel diagram, the WVP matrix parameter can be seen to have a 'UIWidget' annotation (which in this case is set to 'none' in the property panel to ensure that this matrix is not visible to the user). Annotations are authored by right-clicking the annotation folder and choosing 'Create->' in the same way as parameters. The subset of annotations supported is determined by the profile, and the type of parameter.

One parameter type that is slightly different is the Sampler. Samplers have an additional sub-folder called 'Sampling States', and this contains sampler-specific settings, such as the texture filtering mode. The context menu on the sampling states enables additional states to be added.

Adjusting parameters

To adjust any parameter, select it in the Assets Panel and the properties panel will enable it to be changed. For example, if we select the WVP parameter, the properties panel will show the current matrix value, and a push button will enable the matrix editor to adjust it. If the parameter supports a semantic, this will also be editable in the properties panel.

Parameter editors will vary depending on type. For example, a color parameter will enable the color picker, but a matrix will show the matrix editor.

Techniques

A technique in an effect profile represents a chosen implementation of a given effect. Techniques might be split by level of detail, or by implementation cost, or by visual look – for example, a metal effect's Cg profile might contain 'distorted', 'polished', and 'scratched' as technique options. Techniques are typically selected by the runtime engine (for example – a game might choose a low detail technique when objects are far away), or from the material; since in FX Composer you can choose the technique at the material level.

Any number of techniques can be added to an effect. Techniques can also be given a more meaningful name.

Passes

Under each technique lives a list of passes. Passes represent the rendering steps to execute for evaluating the technique. Like techniques, any number of passes can be applied to build up the effect. Passes are made up of Drawing commands, Rendering state, and a shader pipeline.

Drawing Commands

Drawing commands tell the graphics engine how to draw a particular pass. The default draw command is the 'Draw Type', and it is usually set to 'GEOMETRY', indicating that this pass will simply draw the geometry that is assigned to it, using the current pass state. The draw command is always present and can be changed by right-clicking it and choosing a different type. The options are typically used with full screen effects, and are:

GEOMETRY – Just draw geometry

SCENE_GEOMETRY – Draw all of the scene geometry, using this pass state.

SCENE_IMAGE –Draw the whole scene in to the current rendertarget.

SCENE_SCREEN_QUAD – Draw a quad over the whole viewport.

SCENE_SCREEN_QUAD_PLUS_HALF_PIXEL – Draw a quad over the whole viewport, aligned at ½ pixel boundaries.

SCENE_GEOMETRY_NOOVERRIDE – Draw all the scene geometry, but don't override with the current pass state.

The remaining drawing commands can be added using the menu, and enable setting of a different rendertarget surface, and causing a clear to occur before the pass is drawn.

Rendertargets are editable in the properties panel, and the user sets the target index (for MRT), and the surface ID, which is the name of the surface parameter in the profile parameters.

Clear commands are also editable in the properties panel and clear color/depth values can be adjusted, as well as the target index.

Render States

Rendering state represents GPU states which are fixed during the pass. Options such as back face culling, or depth testing are rendering states. Right-clicking the rendering states folder will enable creation of rendering states, and they are grouped into similar sections, such as 'Framebuffer', 'Transform', etc. Rendering state options are different depending on the current device.

Shader Pipeline

The shader pipeline is the last part of the pass information, and describes the shading stages present in the API of choice. In Cg, this is equivalent to Vertex & Fragment programs; and in this case, you may only add these 2 types of stage. Other API's might use fixed function stages, or none at all – these are again specific to the profile, and will show up differently.

Clicking on the fragment program will show a list of settings for that program. In the case of Cg, they are:

- **Entry Point.** The function to call in the shader code.
- **Compiler Target.** The compiler target, such as arbfp1.
- **Compiler Options.** Set the compiler options to be passed in to the command line. It is combined with the compiler options from the technique and the global compiler options of the settings window.
- **SID.** The ID of the shader code. This is shown in the "Shader Source Code" folder, see later.

Once the source code references have been setup, the remaining item is to connect parameters in the shader source code to values in the profile or effect parameters. This is done using bindings. The bindings folder under the pipeline stage shows a list of active bindings. Bindings can be added using the right-click context menu. The name of the binding is the same as the name of the shader source code parameter. The reference value (settable in the properties panel via a drop-down list), refers to the given profile or effect parameter. If a binding is incorrect, this will result in a compile error in the task list.

Shader Source Code

The pipeline stages, in the case of Cg at least, refer to source code that has been written by the developer. The source code folder enables creation of new source code elements, either from files on the disk, or as embedded shader files inside the Collada file. The menu lets you reference existing files, create new ones, or create embedded code. Embedded code elements are given a fixed name which cannot be changed (and this is referenced in the pipeline stage SID). File based shaders gain the name of their associated file.

The menu on a given shader can be used to switch it between an embedded reference and a file based reference, for maximum flexibility.

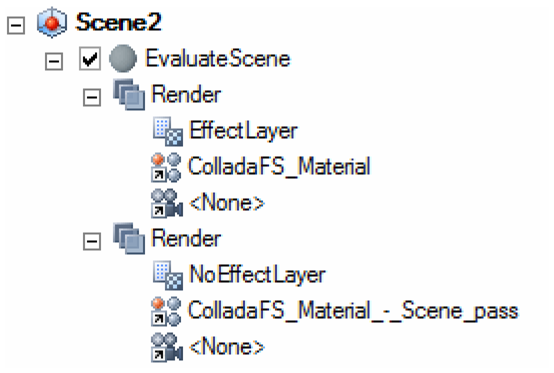
Full Screen Collada FX

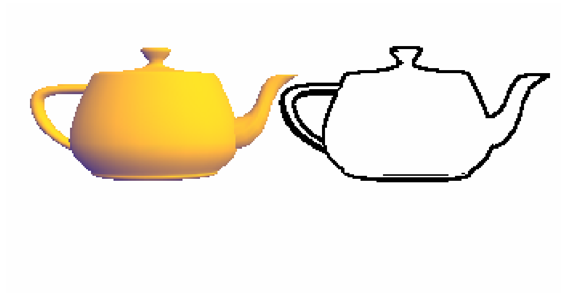
FX Composer enables authoring of effects that change the way the scene is drawn at quite a high level. Examples of full screen effects are things such as 'glow', 'blur', etc… Such visual effects typically modify the whole scene, or a part of it, by building up their visual using offscreen buffers, and compositing techniques.

Full screen effects are authored using the effect system, and then hooked into a given scene using 'Evaluate Scene' nodes. An example is shown below. In this example, the scene is evaluated using 2 rendering steps. Each step consists of a list of layers, a selected material, and a chosen camera. Any number of evaluate scene nodes can be added to a scene, and each can be made up of any number of rendering steps. The example evaluate scene is built using a script, and can be created using the following script commands in the python window:

```
import test_fscolladafx
test_fscolladafx.test()
```

The effect result is to draw 2 teapots in a scene, one of which is edge detected, the other is not. This is achieved using a layer mask to select which objects are drawn using the full screen effect. The screenshots below show the results.


```
Scene2
  EvaluateScene
    Render
      EffectLayer
      ColladaFS_Material
      <None>
    Render
      NoEffectLayer
      ColladaFS_Material_-_Scene_pass
      <None>
```



Evaluate Scene

Each evaluate scene is enabled using a checkbox, as shown in the screenshot. Only evaluate scenes that are selected will be enabled. It is possible to stack up multiple evaluate scenes to composite scenes together.

Render

Each render call causes the engine to draw something. The render menu enables layers to be added, and the material and camera nodes enable different assignments of materials and cameras in the scene.

- **Layers.** Layers are simply selectors for geometry in the scene. If a layer is specified, then the engine will only draw nodes that are present on that given layer. Layers are identified simply by their names, which can be changed. Any node can be assigned to the default layer by setting its layer string to empty, or added to any number of layers by setting a list of comma separated layer names to its node property (which is accessible from the properties page when a node in the scene is selected). The first rendering pass here uses the layer called 'EffectLayer', and the teapot on the right has been added to this layer in the scene. The second pass uses the 'NoEffectLayer' and the teapot on the right is in this layer.
- **Material.** The material selects which material to draw this rendering pass with. This in turn references a given effect, which contains additional Drawing commands in the pass (see Drawing Commands, page 47). The material menu gives a list of available materials to use.
- **Camera.** The camera is used to specify which scene camera point of view is used. This camera will be used to look at the scene for all rendering in this pass. If no camera is specified, then the current scene camera will be used.

CgFX and .fx Authoring

To author a CgFX or HLSL .fx file, right-click on a new or existing effect in the "Effects" section of Project Explorer. The file's context menu will include an option for "New Effect File," which will allow you to choose either an HLSL or a CgFX file. FX Composer will prompt you for the file name and location of the file and open the new file in Code Editor.

To help you get started, FX Composer populates this new file with a simple Phong shader template. You can customize the template by modifying the default.cgfx and default.fx files located in the data folder of the FX Composer installation.

Vertex Stream Remapper

Because different shaders expect common vertex attributes to appear in specific vertex attribute slots, FX Composer 2 gives you full control over this mapping.

There are two levels of control where this can be done: at the Environment level and at the material level.

Global Remapper:

The global Vertex Stream Remapper allows custom vertex attribute configurations to be easily set without having to micro manage each shader assignment.
Since it has no information on the types and semantics, it is string based and matches semantics and streams by name with a non case sensitive search.

You can set a global remapping for the vertex streams in the Development Environment (Tools → Settings… → Environment → Vertex Stream Renapper) that will take effect as the default mapping.

Two types of mapping are possible in these remappers:

- **Stream remapping**
  Remaps an entire stream, for example, to feed a shader with an input semantic of Position with data coming from the Normal stream in the geometry.
- **Vertex Component Remapping**
  Remaps individual components of a stream allowing swizzling on the input streams fed to the shader.

  Example:

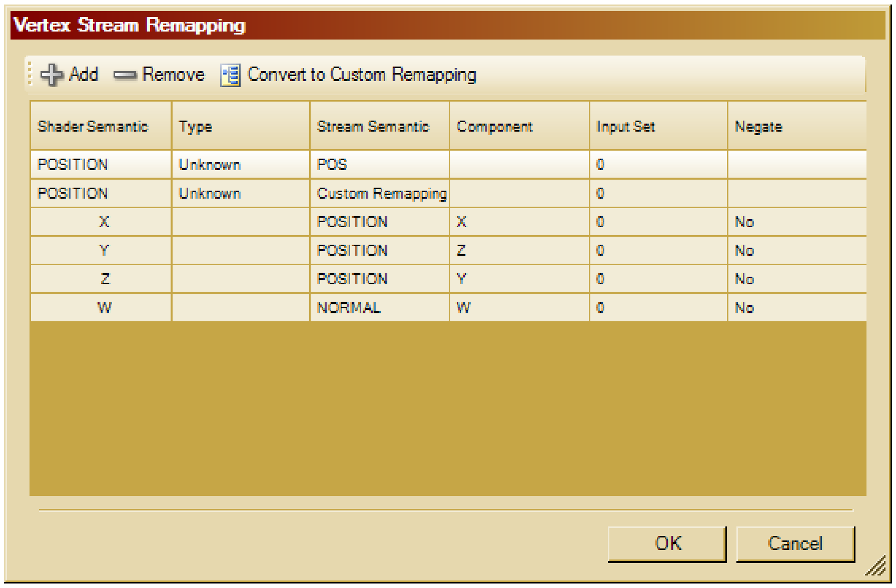
Shader Input	**Geometry Stream**
Position.X	Position.X
Position.Y	Position.Z
Position.Z	Position.Y
Position.W	Normal.W

To modify the global remappings, click on the "…" button on the VertexStreamRemapping row.

It will display a window with a table with 6 columns:

- **Shader Semantic:** The shader semantic to remap.
- **Type:** The type of the data. Usually unknown for this mode since the global remapper has no knowledge of the stream types.
- **Stream Semantic:** The type of data that the shader semantic will be remapped to
- **Component:** The component of the stream. This field is not modifiable when remapping entire streams.
- **Inputset:** The input set of the stream
- **Negate:** An optional negate option to the input component

The figure below shows two examples of remappings:



Shader Semantic	Type	Stream Semantic	Component	Input Set	Negate
POSITION	Unknown	POS		0	
POSITION	Unknown	Custom Remapping		0	
X		POSITION	X	0	No
Y		POSITION	Z	0	No
Z		POSITION	Y	0	No
W		NORMAL	W	0	No

Figure 22. The Vertex Stream Remapper

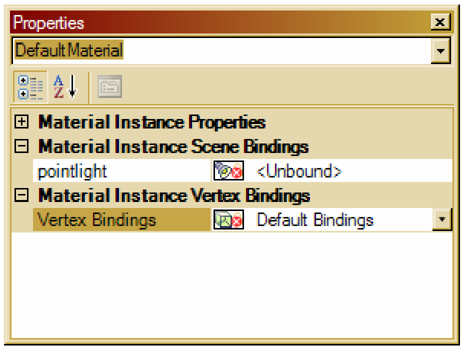
The first line is a stream remapping that will set the "Position" shader semantic to a stream named "Pos" if it exists in the geometry.

The second line and 4 subsequent lines are the representation of the previous example that swizzles individual components of a stream before it sets it to the shader input.

Conversion from a Stream Remapping to Custom Remapping and vice versa can be done by selecting a row and clicking on the "Convert to" button above the table.

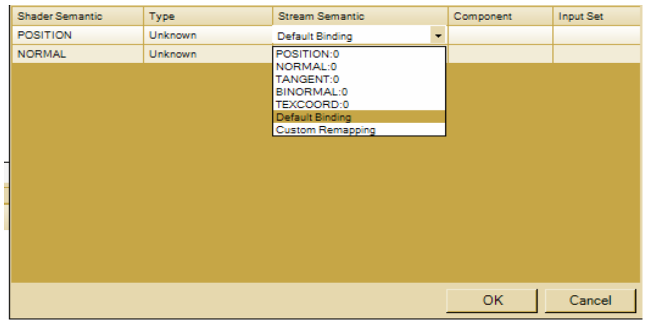
Finer Remapping :

FX Composer provides the user with even finer control over the remapping to enable easy handling of special cases. In order to support shaders that are not sharing the same vertex stream configuration, you can override the default mapping at the material level by left-clicking on an instanced material (that is, a material attached to a piece of scene geometry) in the Assets Panel. The Properties panel will then show you the current mappings in the Material Instance Vertex Binding section.



Material Instance Properties
Material Instance Scene Bindings
pointlight <Unbound>
Material Instance Vertex Bindings
Vertex Bindings Default Bindings

Clicking on the arrow will show a dialog similar to the one used in the global vertex remapper covered in the previous paragraph. Since the streams are known from the program, the Shader and Stream Semantics can be selected in drop downs instead of hand typed.



Shader Semantic	Type	Stream Semantic	Component	Input Set
POSITION	Unknown	Default Binding		
NORMAL	Unknown			

The first column contains the vertex streams that the shader expects (FX Composer derives these by parsing the material's source code). The second column shows the type of the stream when it can be obtained from the material. The third column contains the geometry streams that can be fed to the shader.

Similarly to the global remapper, a custom remapping of fields is possible by selecting "Custom Remapping" in the Stream Semantic drop down of a row. The line with expand in three sublines where a finer control of the input can be set.

By default, FX Composer uses the following vertex stream mapping:

Table 1. Vertex Stream Mapping Used by FX Composer

Register Name	Type of Data
POSITION	Vertex position
NORMAL	Normal
TEXCOORD0	u-v texture coordinates
TEXCOORD5	Tangent
TEXCOORD6	Binormal

Converting CgFX Effects to COLLADA FX Effect

You can convert a CgFX effect to a COLLADA FX effect simply by right-clicking on the corresponding CgFX profile in Project Explorer, and selecting "Convert to COLLADA FX" from the context menu. (Figure 23). Once converted, the CgFX profile will be replaced by a COLLADA FX profile.

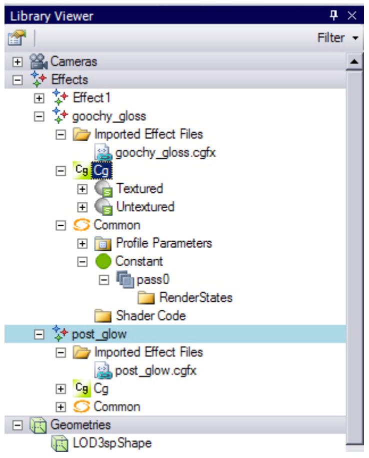
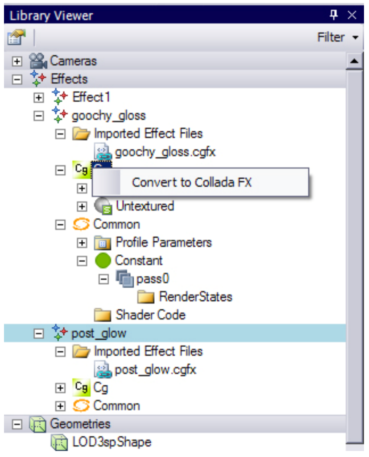
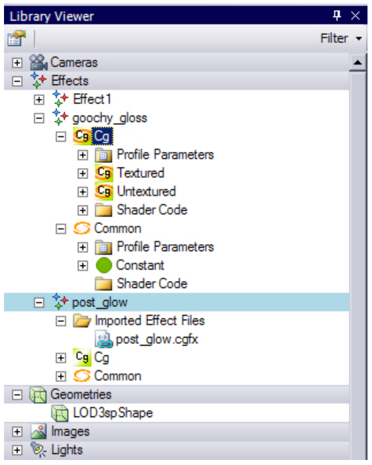
Step 1. Select the CgFX effect.	**Step 2. Right-click and select Convert to Collada FX.**	**Result: Effect is successfully converted.**
		

Figure 23. Converting a CgFX Effect to a COLLADA FX Effect

FX Composer offers various customization options when you convert CgFX files to COLLADA FX. These options are available through the Tools menu (Tools → Settings → Environment → COLLADA Conversion) and are explained below:

- **PreprocessorCommand.** Specifies the preprocessor that FX Composer will use. By default, this is set to "cgc," which is the Cg compiler.
- **RunPreprocessor.** When set to "True" (the default), FX Composer runs the preprocessor on CgFX file. (The preprocessor is necessary for shaders that contain "#include" and "#define" statements.)
- **RunStripper.** When set to "True" (the default), FX Composer will run the code stripper utility specified by "StripperCommand" (explained below) when converting a CgFX file to COLLADA FX.
- **StripperCommand.** Specifies the utility that will be used to process CgFX files. By default, this is set to "fxclean," which removes comments, techniques, passes, and most annotations, leaving just Cg code.

Note: If a CgFX file contains parameters that are evaluated by the Cg runtime virtual machine, that file cannot be converted to COLLADA FX.

Editing COLLADA FX Cg Shaders

To modify a Cg shader in a COLLADA FX effect, double-click the corresponding Cg file name or the inlined code node listed in Project Explorer's Included Shader Files folder, "Converting CgFX to COLLADA FX."

Converting .fx to CgFX

FX Composer 2 comes with a Perl script that automatically converts HLSL .fx files to CgFX. It is called "convert_fx.pl" and you can find it in the Utilities sub-folder of the FX Composer 2 installed location.

convert_fx.pl outputs a copy of the input effect with Microsoft FX/CgFX 1.2 specific state assignments converted to the new OpenGL specific state assignments in CgFX 1.5.

Perl must be installed for the script to run. (You can download a free Perl distribution from www.activeperl.com.) To run the script, execute the following command:

```
perl convert_fx.pl input.fx output.cgfx
```

Limitations

Multiple state assignments on a single line are not supported. If the input file contains any lines similar to the following:

```
SrcBlend = One; DestBlend = Zero;
```

you must move these state assignments to their own line before running this script, as shown below:

```
SrcBlend = One;

DestBlend = Zero;
```

State that has no direct OpenGL equivalent will not be changed; instead, this script will add a comment to the line indicating that the state will have to be manually updated. Search for "FIXME" in the output file to locate these problem states.

See the declaration of the unsupportedState array below for a list of state that is not supported.

- Sampler state inside a pass block is not supported by CgFX 1.5 or this script and will need to be manually moved into a sampler_state block.
- This script only updates state assignments inside a pass or sampler_state block; it will not modify any shaders, annotations, or semantics.

The Render Panel

The Render Panel

The Render panel displays the current scene and has several controls for navigating and manipulating scenes. A tab for each device is installed on your system; typically, you'll see tabs for Direct3D and OpenGL devices. You can dock or undock these to view multiple devices simultaneously or to enlarge a particular rendering.

Toolbar

The toolbar helps you work with cameras and objects in the scene and contains the following buttons:

- **Cameras**. Lists all available cameras in the active scene.
- **Navigation Mode**. Allows you to choose between orbit, trackball, and flythrough.
  - **Orbit**. The orbit mode is a mode of rotation with 2 degrees of freedom. The horizontal displacement of the mouse determines the rotation of the camera in the horizontal plane of the object space. The vertical mouse displacement determines the vertical screen space rotation of the camera. The center point of rotation is always at the center of the screen, while the depth of the center point is adapted according to events such as "zoom extents."
  - **Trackball**. The trackball rotation mode offers a full 3 degrees of rotation. The axis of rotation is orthogonal to the mouse displacement and is parallel to the screen. The angle of rotation is determined by the magnitude of the mouse displacement. The shorter of the width and the height of the screen corresponds to a 360 degree rotation. The center point of rotation is set, as in the orbit mode.
  - **Flythrough**. The flythrough camera mode offers a FPS-style navigation mode for large scenes. The vertical displacement of the mouse corresponds to camera pitch, whereas the horizontal displacement is factored in for the camera yaw. The camera can be moved forward or backward using the mouse wheel or with Shift + left mouse button.
- **Object Selection (Q).** Allows you to select objects. If you want to translate, rotate, or scale a specific object in the scene, you must first select it. Selecting an object also brings up its properties in the Properties panel. You can use the left and right arrow keys to move the selection from object to object. This can be very helpful in complex scenes.
- **Object Translation (W)**. Translates the selected object. A set of axes will appear, and you must click and drag on the axis that you want to translate along.

- **Object Rotation (E).** Rotates the selected object. A set of circles will appear to represent rotation along various axes. You must click and drag on a circle to rotate around that axis.
- **Object Scaling (R).** Scales the selected object. A set of axes will appear to represent scaling along various axes. You must click and drag on an axis to scale in that direction.
- **Zoom Selected Object Extents**. Zooms the camera so that the extents of the selected object are visible.
- **Zoom Scene Extents**. Zooms the camera so that the extents of the entire scene are visible.
- **Reset Camera Rotation**. The camera has a rotation component and a translation component. These cumulative rotations and translations are stored independently. "Reset Camera Rotation" resets the rotation component of the camera only and is helpful, for example, when you have rotated your camera to a point where you are confused about its current orientation.
- **Display Skeleton Bones**. Show bones (if your scene has any).
- **Set Background Color.** Changes the rendering's background color.
- **Save Viewport to File**. This saves the current view to disk as a BMP, JPG, or PNG file.

Manipulating the Camera

To manipulate the camera (that is, the viewpoint), use:

- Alt + Left Mouse Button for rotations
- Ctrl + Left Mouse Button for translation
- Mouse wheel for zooming in and out

Applying Materials

To apply a material to an object, drag-and-drop the material onto the object—the object's appearance will change to reflect the new shader. To apply a full-scene material to your scene, drag-and-drop the material onto the Scene Window's background (which is gray by default).

You can also drag shader files from the Windows Explorer to achieve the same results. If you try to drag a shader onto an object that doesn't have a material assigned to it, FX Composer will automatically create a material for you and assign the effect to that material.

If you try to drag a shader onto an object that has a material, but not the specific profile you're adding, FX Composer will add that new profile to the existing effect. And, if an identical profile already exists, you will be prompted to either replace it or leave it.

You can also change the effect associated with a particular material by dragging-and-dropping a new effect onto that material in the Assets Panel. Any objects in the scene that use that material will automatically be updated to reflect the new shader.

Context Menu

Right-click on any part of the Render panel to view the context menu. The Context menu has the following options:

- **Set Current Scene** (allows you to choose between available scenes)
- **Scene Properties** (displays the current scene's properties in the Properties panel)
- **Show Lights**
- **Show Grid**
- **Show Cameras**
- **Show HUD** (the HUD displays scene name, camera, and camera mode)
- **New Window** – Creates a new viewport tab for the same Device type.
- **Material/Effects**
  - <Shows names of material>
  - <Shows names of effects>

Viewports

The Render Panel contains a set of tabs where each tab holds a viewport. Depending on the configuration, you may see a variety of tabs that reflects the various plug-ins and their respective device. By default, FX Composer 2.0 ships with a Direct3D 9 and an OpenGL device.

Additional plug-ins maybe available from other partners as well.

In addition to supporting multiple devices, you can create multiple viewports from the same device type. Bring up the viewport context menu and select "New window" to create a new tab with its own viewport. This is useful to have different views of the same scene using different cameras or view a different scene with the same type of device.

Scene Options

You can modify the following Render panel parameters via the **Settings…** option the Tools menu. The options are listed in the Scene Options subsection under Environment.

- **InverseMouseWheelZoom.** Changes the direction of zooming when the mousewheel is used.

Tips for Working with Complex Scenes

Here are a few tips to help you navigate complex scenes in FX Composer:

- Zoom Scene Extents and Zoom Selected Object Extents are very powerful, and are a great way to get a good view of your overall scene or the selected object.
- There are two ways to select an object:
  - Click on the object in the Render panel. If the object is within the bounds of a bigger object, simply click again without moving the mouse and FX Composer will cycle through all objects that lie in that pixel.

- If you have an extremely complex scene and navigating through the scene panel is difficult, you can navigate through the Scenes node of Project Explorer to find the object you're looking for. Once you've found it, you can right-click and choose "Select" from the context menu. This will then select the object in the Render panel. (Zooming extents on the object may be useful at that point.)

- If you've selected an object in a complex scene and want to know what material is assigned to it, you can right-click on it and the Material sub-menu in the resulting context menu will contain a reference to the material's technique. Clicking on the material or technique in the context menu will automatically expand Project Explorer to show that material or technique.

Animation Panel

The Animation Slider should be used to control the playback of animated scenes. It allows you to scrub the timeline to position the animation at a current time. You can also play, pause, skip, stop, and loop the animation.

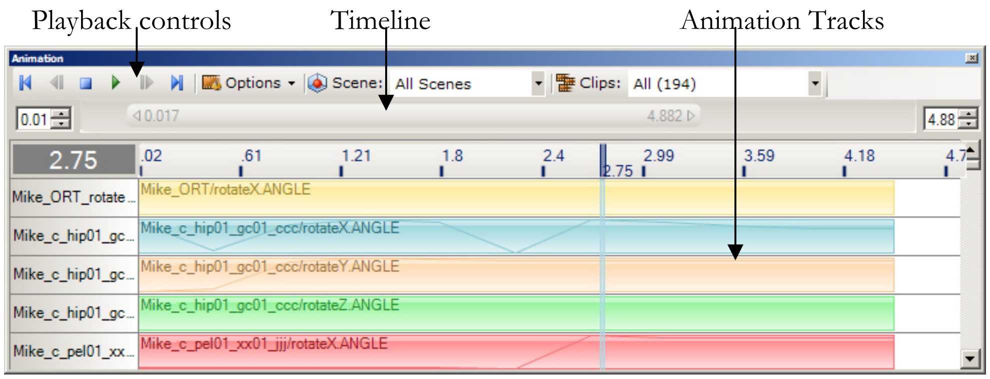


Figure 24. The Animation Panel

In addition to these typical controls, you can constrain the timeline to a sub-range. By either typing the start and end times or resizing the Time Range bar directly, you can enforce a certain range of time within which you can operate.

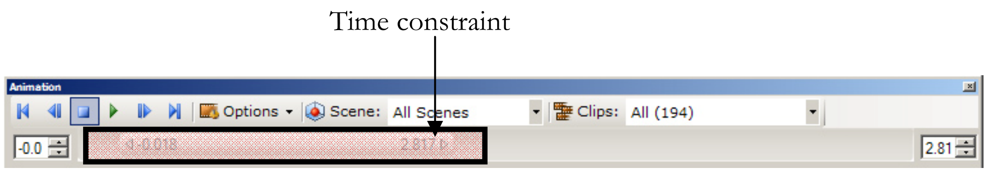


Figure 25: Animation Slider with a sub-range selection

If you play the animation by pressing the play button in the Playback Controls, the animation will be constrained to the time sub-range. The animation toolbar also allows you to pick which scene and clips to display, the playback speed, and whether to sync animation with the active scene.

In addition, the Animation panel displays each individual animation track in your scene. Against the background of each track, you can see a line that shows how that particular track's value changes over time.

Analyzing Shader Performance

The ShaderPerf Panel

FX Composer integrates NVIDIA's ShaderPerf tool to analyze shader performance, complete with informative graphs and tables (Figure 26). To analyze a shader, right-click on its source file in the Assets Panel and select "Analyze Shader Performance" from the context menu (Figure 27). (In FX Composer 2 Beta, Shader performance analysis is only supported on COLLADA FX files.)

By default, the ShaderPerf panel is located in the bottom center of the application.

When no experimentation is active, a startup form is shown. This form allows you to:

- View the currently loaded effects
- Load new effects
- Access the ShaderPerf Panel configuration window

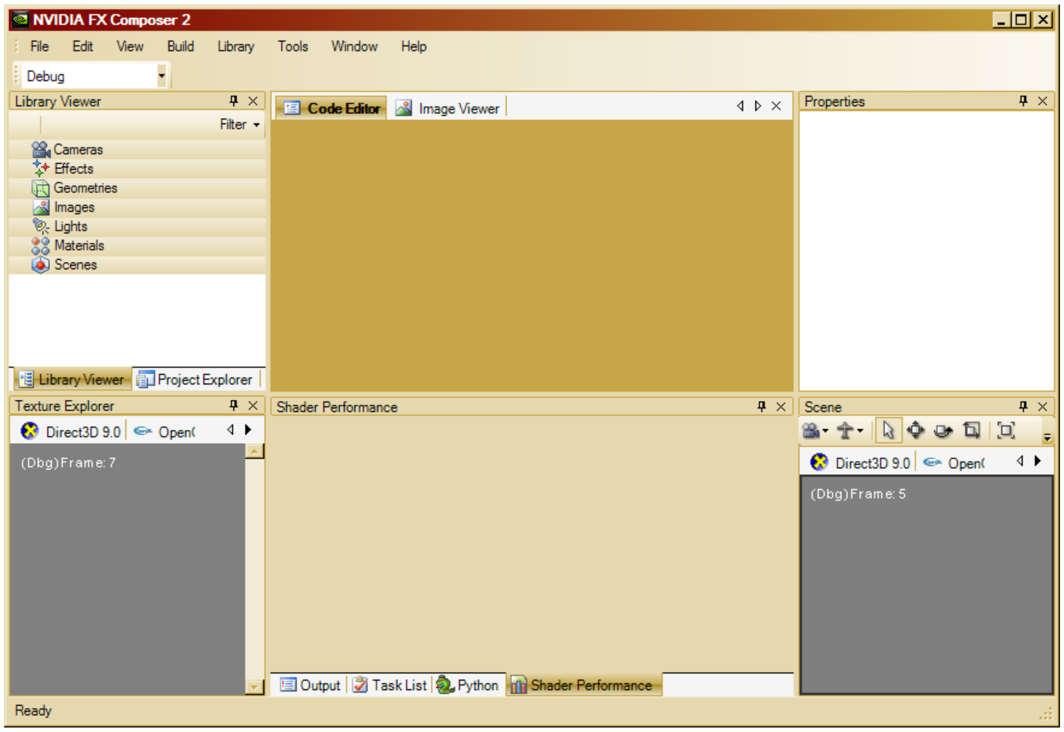


Figure 26. The ShaderPerf Panel

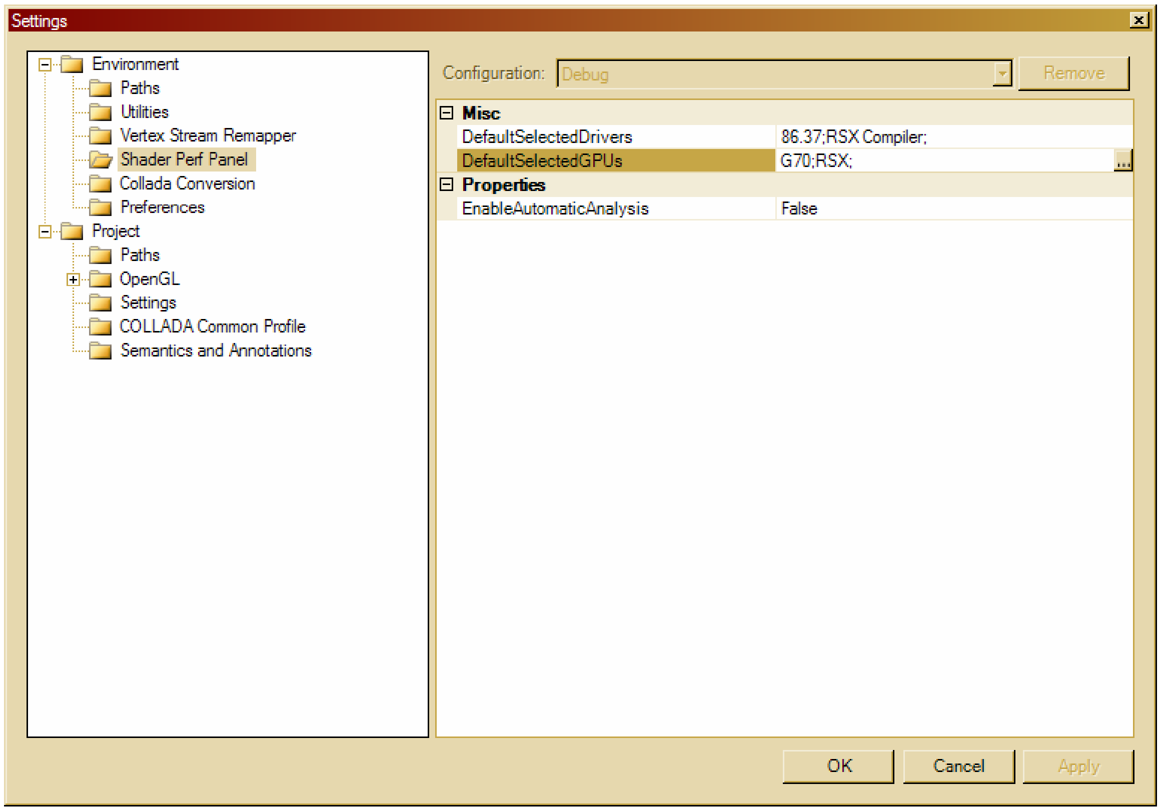
Configuring the ShaderPerf Panel:

Three configuration options are available for the ShaderPerf Panel:

- **Default Selected Drivers:** This list contains all the drivers to include in the experimentations. Disable those that you don't need for better performance during the effect analysis.
- **Default Selected GPUs:** This list contains all the GPUs to include in the experimentations. Since the list can be quite large, it is recommended that you uncheck all unnecessary GPUs from this list.
- **Enable Automatic Analysis:** This option enables the automatic performance analysis of an effect file when it is compiled.

To modify these settings, select the row and click on the button on the right of the row.

Certain NVShaderPerf modules like the RSX dll, allow the compiler they use to be changed. The preferences window contains a setting to change the path of the compiler in "Environment\PLAYSTATION 3\PS3 Cg Compiler".



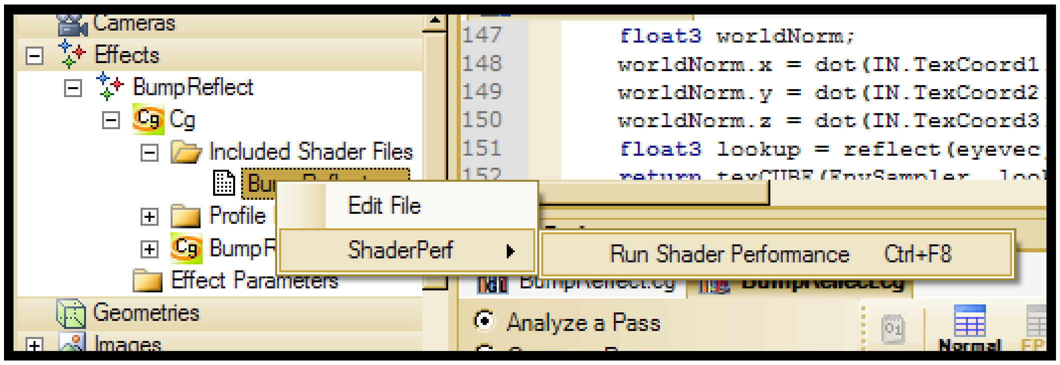
Misc	
DefaultSelectedDrivers	86.37;RSX Compiler;
DefaultSelectedGPUs	G70;RSX;
Properties	
EnableAutomaticAnalysis	False

Running tests

There are two ways to run analyze the performance of an effect file:

- By right clicking on an included effect/shader file in the Assets Panel and selecting Analyze Shader Performance (Figure 13)

- By clicking on a effect file name, in the loaded effect list of the ShaderPerf Panel



```
float3 worldNorm;
worldNorm.x = dot(IN.TexCoord1
worldNorm.y = dot(IN.TexCoord2
worldNorm.z = dot(IN.TexCoord3
float3 lookup = reflect(eyevec
return texCUBE(EnvSampler, lool
```

Figure 27. Analyzing a Shader

Once the Shader performance test is complete, a group of tabs will be created (one per device effect). Each tab contains the result of a device-specific performance test.

ShaderPerf Panel Interface

The ShaderPerf panel's interface shows you various analysis options on the left and results on the right. The options on the left allow you to select which passes to analyze, the type of data to show, and the list of drivers/GPUs to use.

On the right, you'll see the results of the shader analysis experiments, based on the selected options. The type of representation can be modified by using the toolbar buttons in the upper part of the right pane. It can be a table or a graph.

The toolbar on top of the results pane contains can perform the following actions:

- Run: Runs tests for the current panel.
- ASM: Shows the assembly code of the selected passes
- Table: Shows results data as a table
- Graph: Shows the results as a histogram
- Log: Shows the log of the experiments with more details on the results.
- Export: Exports the data as Comma Separated Values (CSV) for external use

The left side of the panel shows all the available options for the current experiment such as:

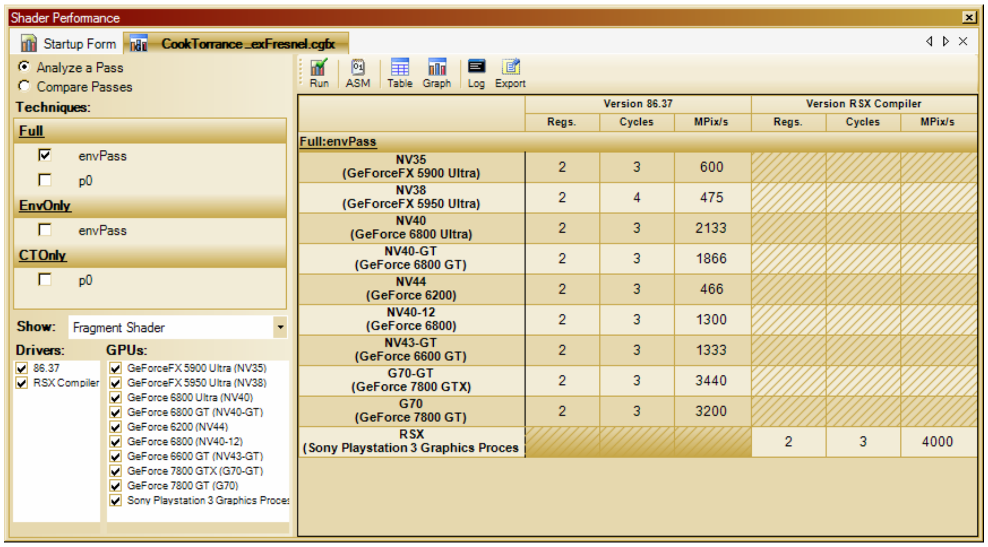
- The passes
- A selection between the Fragment Shader or Vertex Shader
- The driver and GPU selection lists

This side bar allows you to precisely select the data you wish to display in the tables and graphs.

Table/Graph Modes

There are two different ways to display the results of an experiment. The Table and Graph icons on the toolbar allow you to change the display mode.

- The table mode shows the data in columns and rows. Columns represent drivers and rows represent the GPUs. Each cell has three inner cells containing the results of the experiment. (The number of R Registers (128 bits registers) used, the number of cycles used, and the pixel/vertex throughput)
- The graph mode shows a histogram of the vertex/pixel throughput. The displayed data can be exported to an image file by right clicking on the graphical component and selecting "Save image…"


Shader Performance

Startup Form | CookTorrance_exFresnel.cgfx

(●) Analyze a Pass
( ) Compare Passes

Techniques:

Full
- [✓] envPass
- [ ] p0

EnvOnly
- [ ] envPass

CTOnly
- [ ] p0

Show: Fragment Shader

Drivers:
- [✓] 86.37
- [✓] RSX Compiler

GPUs:
- [✓] GeForceFX 5900 Ultra (NV35)
- [✓] GeForceFX 5950 Ultra (NV38)
- [✓] GeForce 6800 Ultra (NV40)
- [✓] GeForce 6800 GT (NV40-GT)
- [✓] GeForce 6200 (NV44)
- [✓] GeForce 6800 (NV40-12)
- [✓] GeForce 6600 GT (NV43-GT)
- [✓] GeForce 7800 GTX (G70-GT)
- [✓] GeForce 7800 GT (G70)
- [✓] Sony Playstation 3 Graphics Proce

Run | ASM | Table | Graph | Log | Export

	Version 86.37			Version RSX Compiler		
	Regs.	Cycles	MPix/s	Regs.	Cycles	MPix/s
Full:envPass						
NV35 (GeForceFX 5900 Ultra)	2	3	600	▨	▨	▨
NV38 (GeForceFX 5950 Ultra)	2	4	475	▨	▨	▨
NV40 (GeForce 6800 Ultra)	2	3	2133	▨	▨	▨
NV40-GT (GeForce 6800 GT)	2	3	1866	▨	▨	▨
NV44 (GeForce 6200)	2	3	466	▨	▨	▨
NV40-12 (GeForce 6800)	2	3	1300	▨	▨	▨
NV43-GT (GeForce 6600 GT)	2	3	1333	▨	▨	▨
G70-GT (GeForce 7800 GTX)	2	3	3440	▨	▨	▨
G70 (GeForce 7800 GT)	2	3	3200	▨	▨	▨
RSX (Sony Playstation 3 Graphics Proces	▨	▨	▨	2	3	4000

Figure 28. The ShaderPerf Panel's Table View

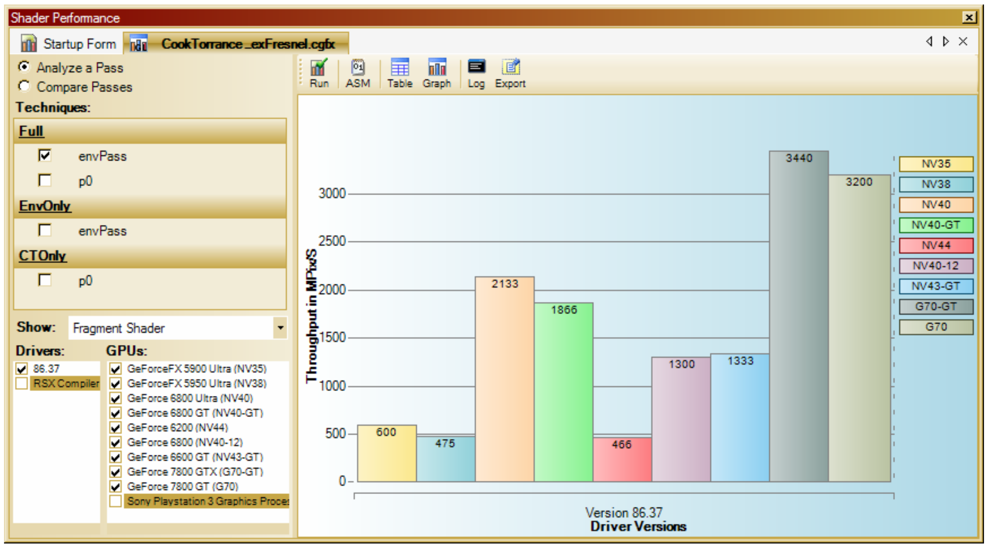
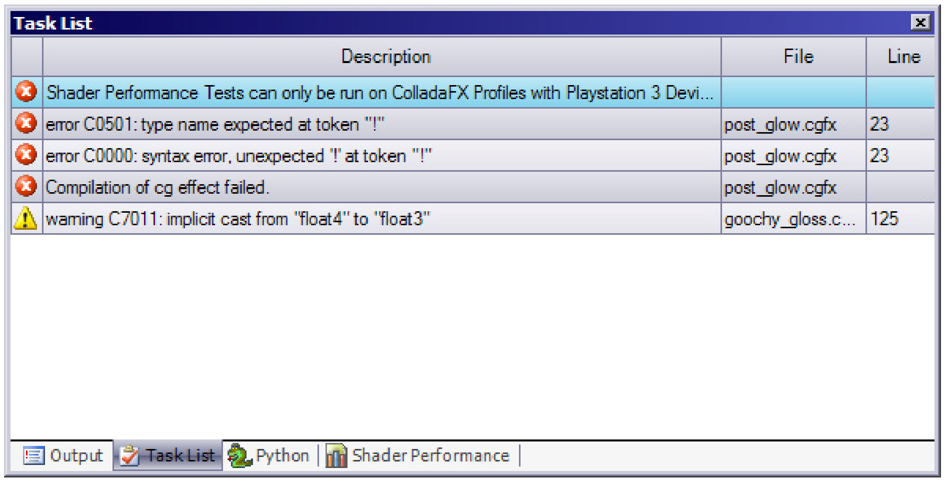


Figure 29. The ShaderPerf Panel's Graph View

Task List

Task List (Figure 30) shows any errors that are found when compiling an effect. The offending lines are highlighted in Code Editor, and the corresponding compiler error is displayed. Double-clicking on an error will take you to the corresponding line of code in Code Editor.

When loading a COLLADA file, FX Composer Beta reports any errors found while validating the file against version 1.4.0 of the COLLADA schema. Even if any errors are found, FX Composer will try to load the file as best as it can. We highly recommend that you repair any COLLADA files that contain errors when possible, even if they load successfully.



	Description	File	Line
⊗	Shader Performance Tests can only be run on ColladaFX Profiles with Playstation 3 Devi...		
⊗	error C0501: type name expected at token "!"	post_glow.cgfx	23
⊗	error C0000: syntax error, unexpected '!' at token "!"	post_glow.cgfx	23
⊗	Compilation of cg effect failed.	post_glow.cgfx	
⚠	warning C7011: implicit cast from "float4" to "float3"	goochy_gloss.c...	125

Figure 30. Task List

ShaderPerf Panel Settings

You can modify the following ShaderPerf panel parameters via the **Settings…** option the Tools menu. The options are listed in the ShaderPerf subsection under Environment.

❑ **Default Selected Drivers.** Specifies which driver(s) to use by default.

❑ **Default Selected GPUs.** Specifies which GPU(s) to use by default.

❑ **Enable Automatic Analysis.** Automatically analyzes your shader when it is opened.

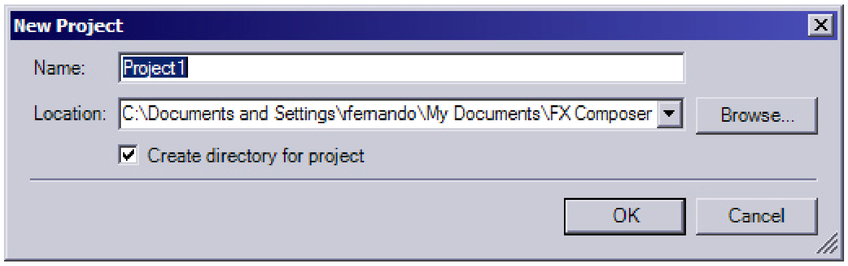
Working with Projects

This section explains how FX Composer projects work and how they interact with COLLADA files.

Each FX Composer project is saved with the .fxcproj extension. These project files can contain reference scene data and shader data in multiple COLLADA files. (Keep in mind that COLLADA files can contain shaders, textures, and geometry, as well as pointers to other COLLADA files.)

Project Structure

Select "New Project" from the File menu to create a new blank project in FX Composer. When you save it, you'll see a dialog box that prompts you for a name and a location for your project (Figure 31). By default, the dialog box will start in My Documents\FX Composer 2 Projects.



Name: Project1

Location: C:\Documents and Settings\rfernando\My Documents\FX Composer

☑ Create directory for project

Figure 31. New Project Dialog

At the root of the project directory is a file called "<project name>.fxcproj." This file contains a list of references to the assets files used in the project, as well as project settings such as the locations of textures, scripts, models, include files, and shaders.

FX Composer allows you to specify a per-project path and a global path for your various resources (textures, models, and so on.) This allows you to easily exchange assets with others. When opening a COLLADA file, FX Composer will first look for resources in your project paths, and then search in the global paths. If no paths are specified, FX Composer will use the URI paths specified in the COLLADA file.

Project Explorer

Project Explorer (Figure 32) is a file-based representation of your project in the form of an expandable scene graph. The main nodes of the scene graph show the various documents in your project.

Documents and Assets

A "document" is a container for textures, shaders, cameras, lights, geometry, and so on. A document's assets are listed as sub-nodes under each document node. For example, in Figure 32, "Duck" is the name of the project, and its assets are "goochy_gloss," "post_glow," "LOD3spShape," and so on.

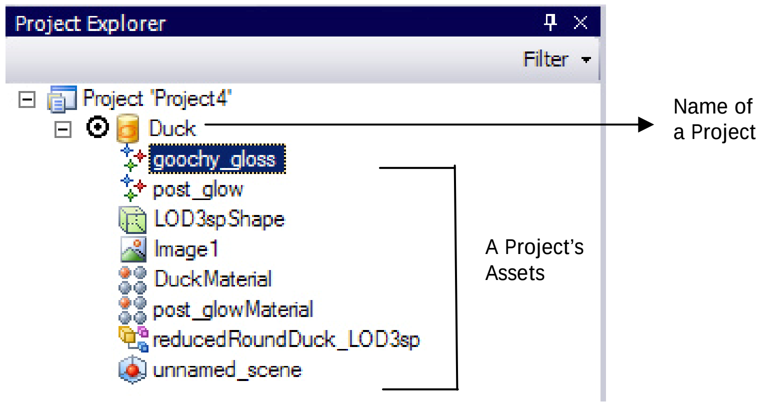


Figure 32. Project Explorer

FX Composer support 3d file formats other than COLLADA. When importing geometric assets within FX Composer, everything gets converted internally to COLLADA. Therefore, when you import 3ds, .x, OBJ or FBX files, they get turned into COLLADA geometry and materials.

Active Documents

Although you can have a number of documents in a project, only one can be active at any time. If you create additional geometry, lights, shaders, textures, or cameras, they will be added to the active document. The active document is indicated by a filled dot in the scene graph.

Documents can exist physically on disk (for example, as a COLLADA .dae file), or virtually (in memory).

Physical Documents

To load a physical document, right-click on the project node and select "Add Document From File" from the resulting context menu. A standard "Open File" dialog box will open, allowing you to load a COLLADA .dae file. (Future versions of FX Composer will support other formats such as .3ds .and x.)

Virtual Documents

Virtual documents are useful for organizing and grouping assets into a logical collection. You can create a new virtual document by right-clicking on the project node and selecting "New Document." This will create a new empty node in Project Explorer, where you can place additional assets.

Once you are satisfied with a virtual document's contents, save it to disk by selecting it and choosing "File → Save <Document Name> As…." At this point, it becomes a physical document.

COLLADA Documents

You can organize the assets within your documents by dragging-and-dropping them across documents. For example, you could use this feature to create a document that contains all your meshes.

When you save a project, all effects in your project will be saved into their respective COLLADA files. You can create a copy of a COLLADA document by right-clicking on its asset file within Project Explorer and choosing "Save As…" from the resulting context menu.

Keep in mind that any new assets you create or import are added to the active document. The active document is indicated by a filled radio button. You can make a different document active by either clicking on the radio button next to it or right-clicking on it and selecting "Set as Active Document."

Sample COLLADA Files

FX Composer ships with several examples of COLLADA files. You can find them in the MEDIA/COLLADA sub-folder at FX Composer's installed location. Typically this will be:

C:\Program Files\NVIDIA Corporation\NVIDIA FX Composer 2\MEDIA\COLLADA

Project Configurations

Similar to other development environments, FX Composer supports the notion of project configurations. This allows you to specify different paths and compiler options for each configuration. For example, your Debug configuration might include visualization techniques that are useful during development but not meant to ship in the final product.

By default, a new project contains both a Debug and a Release configuration, but you can create custom configurations by selecting any project folder, clicking on the "Configuration" drop-down, and choosing "Add new configuration…" You will then be prompted for the configuration name, and you can also choose whether you want project settings copied from an existing project.

You can switch configurations using the "Configuration" drop-down just below the File menu.

Assets Panel

Assets Panel (shown in Figure 33**Error! Reference source not found.**) displays all the "assets" in a project, grouped by asset type. Use the drop-down Filter menu to select the asset types you want to see, such as:

- Animations
- Animation Clips
- Cameras
- Effects
- Geometries
- Lights
- Materials
- Scenes
- Deformers
- Textures

The Asset panel provides a detailed view of all the art assets that are composing the project.

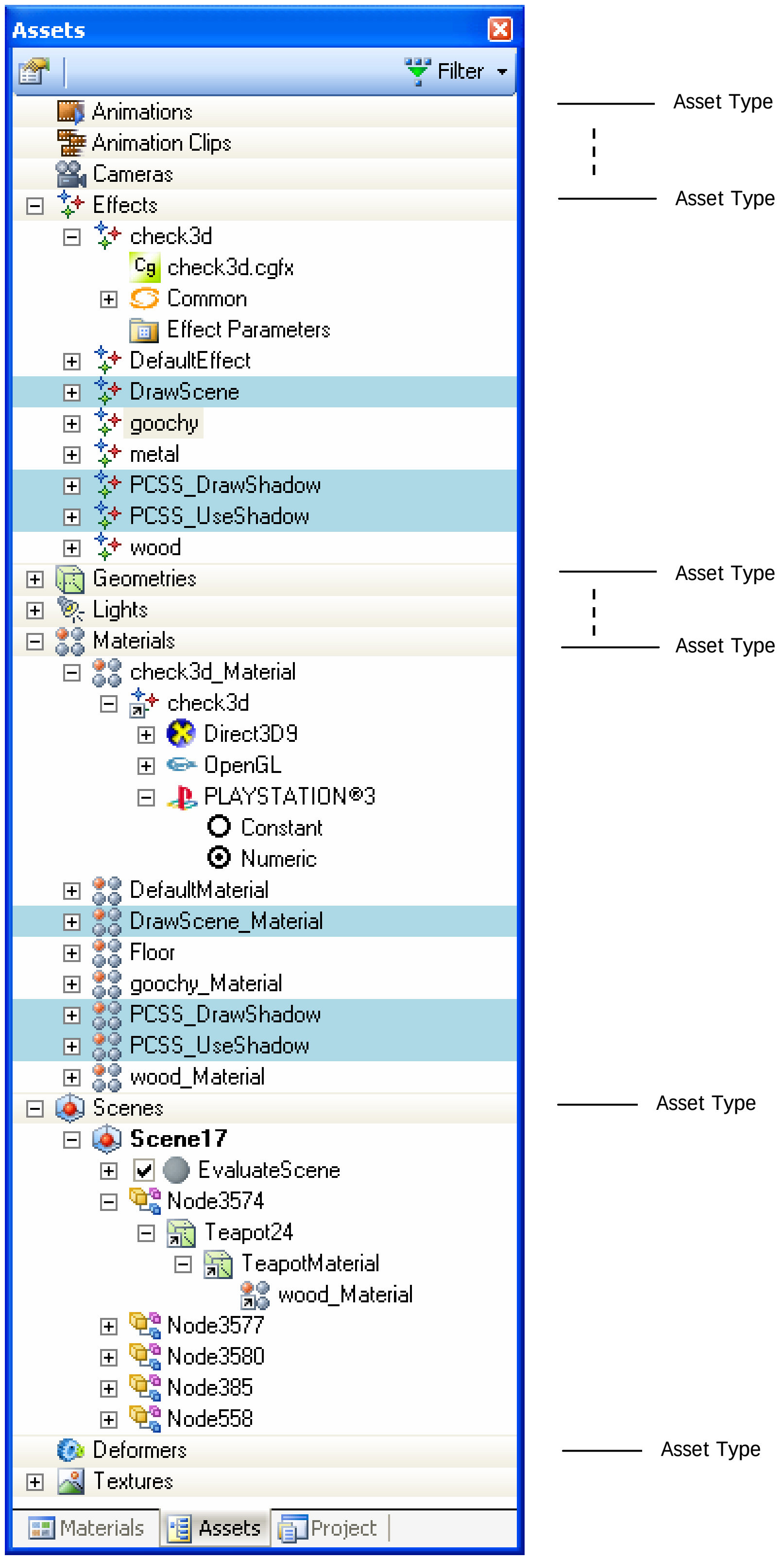


Figure 33. Assets Panel

Common Options

Right-click on any divider or node to bring up a context-sensitive menu of common options. For example, right-click on the "Lights" divider to add the various light types—directional, point, or spot—as shown in **Error! Reference source not found.**.

To rename an asset, select it and press F2.

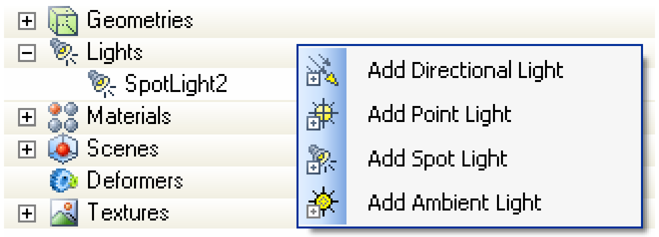


Figure 34. Right-Clicking on a Divider

Types of Assets

Animation

This section of the Assets Panel groups all the animation streams in your project. Animation are referenced by the Animation Clips that bind the streams to objects parameters.

Animation Clips

This section groups all the Animation Clips in your project. Animation Clips are entities that bind Animation data to objects parameters. The binding allows to reference a sub-section of the animation stream in order to crop certain portion of the animation stream.

Scenes

A "scene" is a composition of geometric objects, lights, cameras, and so on. A project can have many scenes. You can use different scenes for a variety of purposes—common examples are different levels or different test scenarios. Right-click on a scene node and select "Apply to All Viewports" to make a scene active to all the Device tabs. You will notice that it is possible to view different scenes through different viewports. If you select a specific viewport in the 3D Render Panel and then a specific Scene in the Assets Panel, the scene object context menu allows you visualize the scene if you select "Apply to current viewport". (See section Viewports, page 59 - to learn how to manage additional viewports).

In each scene you can reference the various assets in your project. For example, you could have two scenes, each of which references your main character's geometry (which would show up in the "Geometries" section of the Assets Panel).

The Scene section is, in essence, a scene graph, where each node contains a reference to an asset. (Each asset icon includes a "shortcut" picture to remind you of this.) Double-clicking on any node with a shortcut icon automatically selects the referenced asset.

Selecting a node displays its properties in FX Composer's Properties panel and show the bounding box selection in the Render panel if the node has a 3D representation. Right-clicking on a node in the Assets Panel brings up a context-sensitive menu with relevant actions.

Effects

It is important to understand the relationship between effects and materials. An "effect" is a shader—for example, marble. A "material" is an instance of an effect with specific properties settings—for example, green marble. Materials are what you actually apply to objects in your scene.

The advantage of having effects and materials is that you can modify the underlying shader code of several materials at once if they are based on the same effect, simply by modifying the effect. Without a materials system, you would have to create separate shaders for each material variant and modify all of these shaders individually to achieve the same result.

Sub-Nodes

Each effect node in the Effect section is represented by the icon and contains several sub-nodes. Full-scene effects are indicated with a blue bar that spans the width of the control.

Effect Parameters. Double-clicking on this displays all the effect's parameters in the Properties panel. Note, however, that none of these parameters can be modified because you can only interact with tweakables for **materials**. Effects parameters are useful to drive multiple profile parameters altogether.

All other effect sub-nodes are Effect Profiles. These vary according to the needs of the user or the project. FX Composer 2 supports the following Effect profiles:

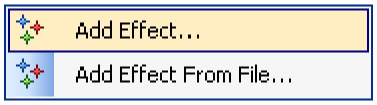
HLSL **Microsoft Effect FX**. This node lists the vertex and fragment programs that are referenced in the COLLADA FX shader nodes. Double-clicking on a shader program (for example "goochVS.cg") opens it in Code Editor.

Cg **NVIDIA Effect CgFX**. This node lists external effects that have been imported. For example, Microsoft HLSL .fx files or CgFX files can be authored and applied in FX Composer through this mechanism. Double-clicking on an effect opens it in Code Editor. You can convert an imported effect file to a COLLADA FX by right-clicking on the file and selecting "Convert to COLLADA FX" from the resulting context menu.

COLLADA FX Common. This node lists native COLLADA FX common profiles such as Constant, Blinn and Phong effects along with all the parameters to configure them.

Cg**COLLADA FX Cg**. This node contains a sub-tree which represents the structure on a COLLADA FX Cg effect. The profile sub-nodes are used to create and edit these profile. To learn more about COLLADA FX Cg authoring, See section (???)

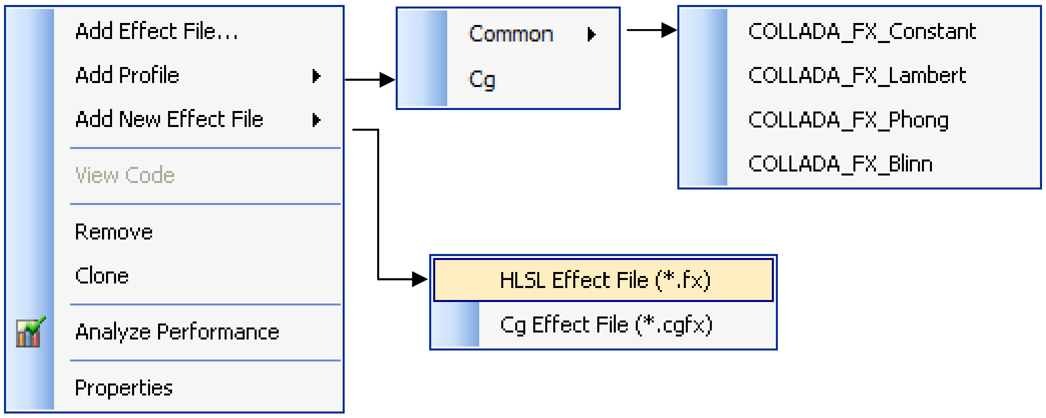
To add new individual effects, right-click on the Effects section of the Assets Panel, and the following context menu will appear:



- **Add Effect…** Allows you to create a new Effect via the Effect Wizard. To learn more about the Effect Wizard, See section XXX.
- **Add Effect From File…** Allows you to create a new Effect from an existing shader container file like a Microsoft .FX or an NVIDIA .CgFX file..

Context Menu

Right-click on any effect to display a context menu with several choices:



- **Add Effect File…** Allows you to load an existing shader container file like a Microsoft .FX or an NVIDIA .CgFX file. This action merges the file's techniques with the current effect's techniques.
- **Add Profile…** Allows you to create a new profile from the supported profiles.
- **Add New Effect File…** Creates a new empty shader container file.
- **View Code**. Loads any shader code related to any profiles in the Code Editor.
- **Remove**. Removes the current effect from Project Explorer.
- **Clone**. Create a copy of the Effect. The copy is directly added to the list of Effects under a default name starting by "Copy_of_".
- **Analyze Performance**. Activates the Shader Performance panel to run the GPU and Driver simulators to analyze the selected shaders (See section Shader Performance to learn more about this feature) .
- **Properties**. Displays an effect's properties in the Properties panel.

Materials

Each material node in the material section contains several sub-nodes. Full-scene materials are indicated by a blue highlight and have a check box to enable or disable them.

Click on a material name to display all of the referenced effect's parameters in the Properties panel. You can then modify the parameters to customize the material's appearance.

Sub-Nodes

Effect Reference. Effect Reference tells you what effect the current material is based on. For example, in **Error! Reference source not found.**, you can see that the material named "Check3d_material" is based on the "check3d" effect. Under the effect reference is a list of rendering devices supported by the material:

- Sony PLAYSTATION 3
- PC (Direct3D 9.0)
- PC (OpenGL)

Under each rendering device, you will see the available techniques, accompanied by radio buttons for choosing the technique. The list of techniques also features icons to indicate their profile:

- DirectX HLSL .fx
- OpenGL CgFX
- COLLADA FX Cg

Context Menu

Right-click on any material to display a context menu with several choices:

- **Assign Effect**. Allows you to assign any available effect to this material. Note that only effects listed in Project Explorer are available in this sub-menu.
- **Apply to Selection**. Applies the material to the currently selected geometry
- **View Code**. Loads the shader code of all the profiles in the Code Editor
- **Rename**. Renames the current effect.
- **Remove**. Removes the current material from Project Explorer.
- **Properties**. Displays material properties in the Properties panel.

For a *material*, you will see the same properties regardless of which backend renderer you're using (provided that the properties are identical across profiles). This means that modifying a material's properties will be reflected in all the different renderers that the material supports.

On the other hand, when you view an *effect's* parameters, you will see a full set of parameters for each backend renderer that the effect supports. These parameters will all be grayed out because you can only modify parameters for materials.

Lights

This section of the Assets Panel groups all the lights in your project. FX Composer supports spot lights, directional lights, ambient lights and point lights.

Cameras

This section of the Assets Panel groups all the cameras in your project. Cameras can be either orthographic or perspective.

Textures

This section of the Assets Panel groups all the textures in your project. FX Composer supports the following file formats: .dds, .jpg, .png, .tga, .bmp, .exr, and

.hdr. Furthermore, it supports a variety of texture types (cube map, 1D, 2D, 3D) and pixel formats (RGBA, RGB, DXT, 32-bit float, 16-bit float).

Double-click on a texture node to open the Texture Viewer, which makes it easy to browse a collection of images (see the Texture Viewer section for more information).

In addition to static textures, FX Composer supports project-wide Render Targets that can be shared across multiple scene renderings. The Render Targets are useful for efficient full-scene effects like Shadow Mapping or Glow effects.

Left-click on a Render Target node to bring its properties in the Properties Panel. Render Targets properties are editable as opposed to static textures.

Geometries

This section of the Assets Panel groups all 3D models in your project. FX Composer also has built-in geometric primitives: teapots, planes, tori and spheres.

Deformers

This section of the Assets Panel groups all mesh Deformers in your project. FX Composer supports Skin Deformers for Skinned meshes.

Asset Location Resolution

FX Composer 2 uses a typical priority-based search path behavior that helps users interchange their projects and COLLADA documents across different machines.

Three different search levels are available for resolving asset locations, which occur in the following order:

1. Use the relative path of the referenced asset.
2. Use the project path.
3. Use the environment path.

If the asset is still not found, FX Composer uses the asset filename, project path, and then the environment paths to try to find the asset.

Here is an example of how FX Composer would go about finding the texture file foobar.dds referenced by imagelib.dae under the relative path .\Images\2d\foobar.dds.

Project Path

C:\temp

Environment Path

D:\temp

ImageLib.dae is located in "My Documents."

FX Composer 2 will attempt to find "foobar.dds" in the following locations in the following order:

4. My Documents\.\Images\2d\foobar.dds
5. C:\Temp\.\Images\2d\foobar.dds

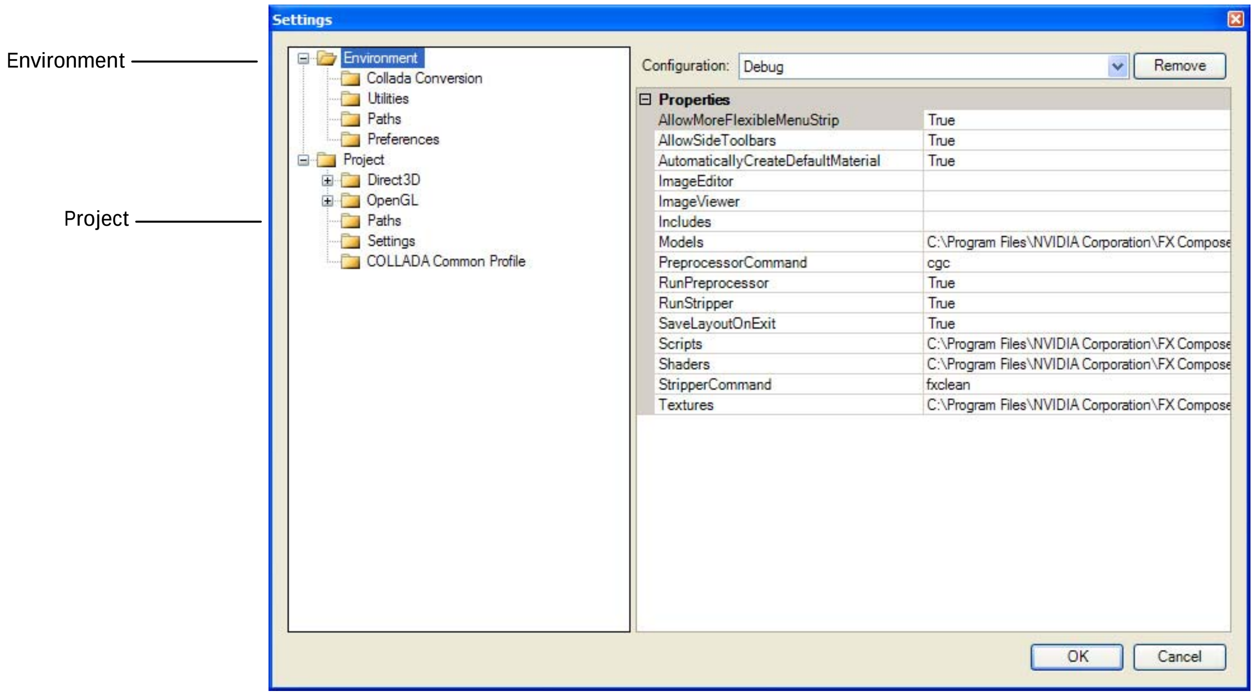
6. D:\Temp\.\Images\2d\foobar.dds
7. My Documents\foobar.dds
8. C:\Temp\foobar.dds
9. D:\Temp\foobar.dds
10. Display an Error – File Not Found!

If FX Composer finds foobar.dds in steps 4 though 6—a warning will appear in the log Window because it is not considered an ideal asset resolution.

Environment and Project Settings

FX Composer offers numerous settings for customizing its development environment and project configuration (Figure 35). You can access these through the Tools menu (Tools → Settings).

Items in this menu are grouped under two main categories: Environment and Project. Environment settings are global, whereas Project settings are specific to the current project. Project settings override Environment settings.



Configuration: Debug

Properties	
AllowMoreFlexibleMenuStrip	True
AllowSideToolbars	True
AutomaticallyCreateDefaultMaterial	True
ImageEditor	
ImageViewer	
Includes	
Models	C:\Program Files\NVIDIA Corporation\FX Compose
PreprocessorCommand	cgc
RunPreprocessor	True
RunStripper	True
SaveLayoutOnExit	True
Scripts	C:\Program Files\NVIDIA Corporation\FX Compose
Shaders	C:\Program Files\NVIDIA Corporation\FX Compose
StripperCommand	fxclean
Textures	C:\Program Files\NVIDIA Corporation\FX Compose

Figure 35. Environment and Project Settings Dialog

The dialog box includes settings such as project paths, utility locations, ShaderPerf panel settings, Cg compiler settings, and semantics and annotations. Each of these is explained in detail in the relevant section of this document.

Sample Project

FX Composer's sample project demonstrates many powerful features. Loading and exploring this project is a great way to become more familiar with FX Composer and its capabilities.

You can find the MadModMike sample project under the FX Composer 2.0\Project folder.

It comes with a Python script located in MikeMap.py that demonstrates how to automatically assign shaders to a character freshly exported from Autodesk Maya. Bring the Scripting Panel and type import MikeMap, MikeMap.Run()…and the script will look for object names and materials to automatically apply new CgFX shaders and normal maps.

This is a good example as to how a studio can automate shader assignements.

Working with Layouts

Because FX Composer is highly customizable, it provides you with the ability to load and save layouts. You can access this capability via "Layouts" in the View menu (see Figure 36).

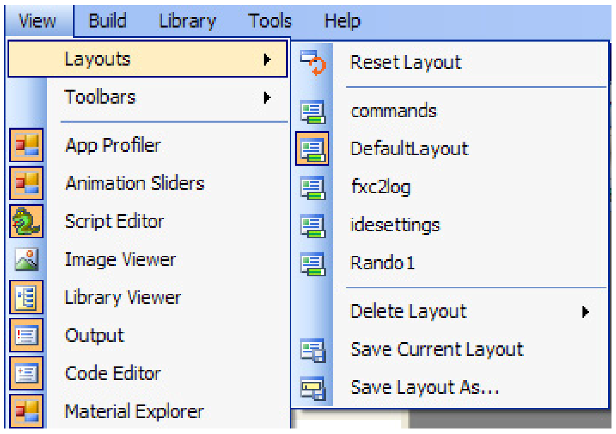


Figure 36. The Layouts Sub-Menu

This menu offers several options:

- **Reset Layout.** This reverts to FX Composer's default layout.
- **Existing Layouts.** This is the list of existing layouts. Any new layouts you create will be included in this list.
- **Delete Layout.** Allows you to delete one of your existing layouts.
- **Save Current Layout.** Saves the current layout, overwriting the old layout with the same name. Note that if you use this option without first creating your own layout, you will replace FX Composer's default layout (not recommended).
- **Save Layout As…** Saves the current layout with a new name. This new layout will then be added to the list of existing layouts.

Changing Layouts

You can choose between existing layouts by using the layout toolbar (see Figure 37), or by selecting a layout from the Layouts submenu in the View menu.

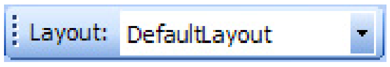
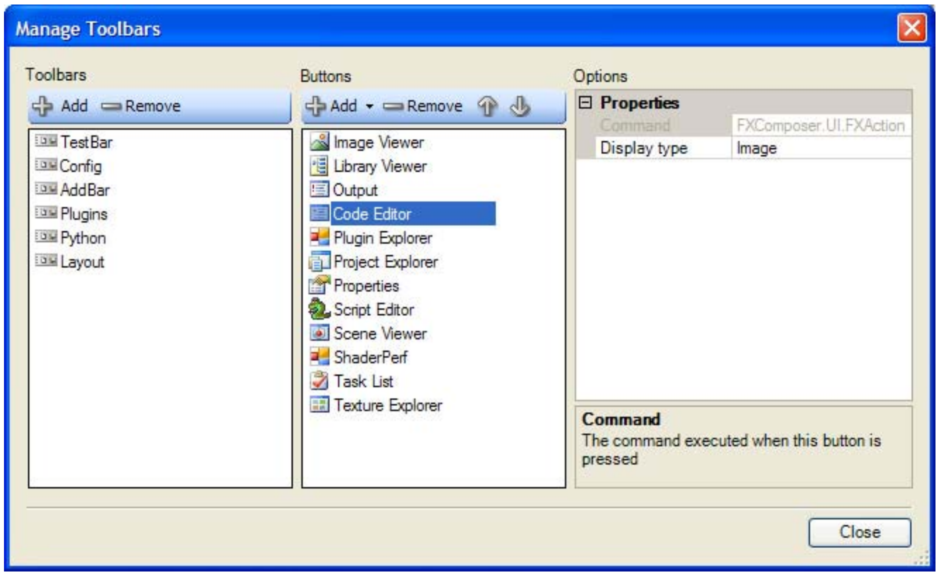


Figure 37. The Layout Toolbar

Customizing Toolbars

One of FX Composer's key production-friendly features is the ability to customize toolbars. To do this, click on the View menu, and choose "Toolbars → Customize…." This will bring up a Manage Toolbars dialog box (see Figure 38). Alternatively, you can right-click on any toolbar and select "Customize…" from the resulting context menu.



Command	FXComposer.UI.FXAction
Display type	Image

Command
The command executed when this button is pressed

Figure 38. The Manage Toolbars Dialog Box

The Manage Toolbars dialog box allows you to create and modify toolbars. In the first column you'll see the existing list of toolbars. You can create a new toolbar by pressing the + icon. (And you can remove toolbars with the – icon.) Selecting a toolbar from the list shows you the buttons that belong to it. You can create additional buttons by pressing the + icon above the column. (And you can remove buttons with the – icon.) When adding buttons, you can choose between Commands, Separators, the Layout Selector, Labels, Python Scripts, and the Configuration Selector.

Selecting a particular button will show you the metadata associated with it (for example, icon, script filename, tooltip, display text, etc…) You can then modify any of these to customize the button.

Please note that if you're assigning a Python function, its name must be carefully specified with the full namespace, function name, and parentheses. A typical function example would be "test_toolbars.AddSphereFunction()," with the script filename as "test_toolbars.py."

General Preferences

You can set a variety of preferences for FX Composer via the Settings… selection in the Tools menu. General preferences are in the Preferences subsection of the Environment section.

- **Add To Current Project On Download.** (Applies to Shader Library downloads.)
- **Allow More Flexible Menu Strip.**
- **Allow Side Toolbars.** Enables toolbars to dock on the side of panels.
- **Automatically Create Default Material.** Automatically creates a default material when you create an object.
- **Load in Editor on Download.** (Applies to Shader Library downloads.)
- **Save Layout on Exit.**
- **Show Download Options Box.** (Applies to Shader Library downloads.)
- **Show Finished Downloading Message Box.** (Applies to Shader Library downloads.)

FX Composer SDK

Geometry File Importer

FX Composer provides support for the geometry file formats commonly used in the graphics industry. FX Composer 2 Beta supports Wavefront OBJ, Microsoft X, Autodesk FBX and Autodesk 3DS file formats.

The source code of the geometry importer plug-in is given as part of the FX Composer SDK. They are useful for allowing developers who are using other file formats to very quickly write their own importers and enable them to load their geometry into FX Composer.

The source code for all the geometry importers is located in:

C:\Program Files\NVIDIA Corporation\FX Composer 2\SDK\

The OBJ file importer is located below this directory under the WavefrontObj directory.

It is worth noting that all geometry imported in FX Composer project files gets translated to COLLADA and stored as such. COLLADA represents a good way for storing more advanced information about shaders, materials and their parameters, and as to how they can be bound to scene geometries and objects such as lights and cameras.

Image File Importer

FX Composer provides support for the most common image file formats. In addition to these formats, FX Composer 2 Beta provides support for Softimage 3D PIC image file format. This is a non common file format proprietary to Softimage (You can find an Adobe PhotoShop Plug-in for the PIC format on: http://www.fxshare.com/)

The FX Composer PIC plug-in is part of the SDK, and therefore, the complete source code is given. The FX Composer 2 SDK is located in:

C:\Program Files\NVIDIA Corporation\FX Composer 2\SDK\

The PIC image file loader is located below this directory under the PIC Format directory.

Scripting Commands

Talk about the scripts in Plugins/etc…

Semantic and Annotation Remapper Plug-ins

FX Composer allows the user to fully remap the set of Semantic and Annotation used to feed the correct data to shaders. The section entitled Semantic and Annotation
Remapping (page 103) gives a detailed explanation as to how to use the system.

FX Composer goes one step further to support custom Semantic and Annotation by exposing custom node operators via plug-ins.

A code example of a Semantic and Annotation Remapper plug-in is located in the Remapper folder under the SDK folder (C:\Program Files\NVIDIA Corporation\FX Composer 2\SDK\)

The sample shows how to implement a Matrix Inversion operator and how to implement a custom way of setting a value of a vector element.

Advanced Asset Creation

Materials

A material in FX Composer always references a particular effect. Selecting a material in the Assets Panel brings up all the material's properties in the Properties panel. You can then modify the properties as you wish, including dragging textures from the Texture Viewer for properties that require textures. (See "Texture Viewer" for more information.)

Material parameters can also be bound in groups to objects in the scene (for example, the light position in the Phong model can be tied to a spot light's position). This is known as a material scene binding, and is explained in more detail in the "Material Scene Bindings."

You can create a new material by right-clicking on the "Materials" divider in the Assets Panel. You can also rename a material by pressing F2 when it is selected.

Dragging-and-dropping a material onto an object in the Render panel will assign that material to the object, replacing any existing material assignment. If you want to keep the material the same, but change its associated effect, you can do so by dragging and dropping the effect onto the object.

Images

This section of the Assets Panel groups all the images in your project. FX Composer supports the following file formats: .dds, .jpg, .png, .tga, .bmp, .exr, and .hdr. Furthermore, it supports a variety of texture types (cube map, 1D, 2D, 3D) and pixel formats (RGBA, RGB, DXT, 32-bit float, 16-bit float). Clicking on an image in the Images list will highlight it in the Texture Viewer (if the Texture Viewer is visible). Double-clicking on an image node will open the Texture Viewer, which makes it easy to browse a collection of images (refer to the Texture Viewer section for more information).

You can add images by right-clicking on the Images divider in the Assets Panel. You can also rename an image by pressing F2 when it is selected.

When working with images, please note that OpenGL assumes the texture coordinate origin is at the bottom-left, whereas Direct3D assumes the origin is at the top-left. FX Composer's image loader automatically flips images when necessary for each API (depending on the image format and its origin convention). This approach means that decals are applied properly in either API without having to modify the texture coordinates, but look-up tables have to either be API-specific, or the shader needs to be API-specific.

The following table lists supported image formats and their origin conventions:

Table 2. Supported Image Formats and Origin Conventions

Format	Image Origin Specification
.dds	Top-Left
.jpg	Top-Left
.png	Top-Left
.tga	Top-Left or Bottom-Left
.bmp	Bottom-Left
.exr	Top-Left
.hdr	Top-Left

Geometry

FX Composer supports the creation of internal procedural primitives (currently spheres, planes, tori and teapots) by accessing the Library Menu -> Add Geometry off of the applications' main menu or by right-clicking on the Geometries divider in the Assets Panel. You can also rename a mesh by pressing F2 when it is selected.

In addition to these procedural primitives, FX Composer supports import of OBJ files. (See 3D file importers, on page 88 for more information on how to import external geometry in to FX Composer)

The Geometries section of the Assets Panel lists all mesh definitions that are available. You can instantiate any mesh definition into the current scene by dragging-and-dropping it onto the Render panel. The mesh will then appear at the scene origin, and FX Composer will do a zoom extents operation to ensure that the entire scene is in view. The drag-and-drop operation will also create a new scene node (in the Scenes section of the Assets Panel) with the geometry as a child node. Alternatively, you can achieve the same result by dragging-and-dropping a mesh from the Geometries section of the Assets Panel onto a scene node (in the Scenes section).

Deformers

FX Composer supports rigging of geometry through skeletons. The rigging data is imported from COLLADA files that have "controller" data. You can also rename a mesh by pressing F2 when it is selected.

The Deformers section of the Assets Panel lists all skinning data that is loaded. Currently, dragging and dropping of deformers is not supported. If a COLLADA file instantiates the controllers, the Render panel will display the instanced controllers. Selecting an instanced controller object displays its properties such as the material and the skeleton data in the Properties panel. You can zoom extents to a controller just as you zoom extents to a piece of geometry. Currently FX Composer does not support the "Save As" feature for deformers, but they can be saved using the "Save All" feature.

Lights

FX Composer supports four types of lights: ambient lights, spot lights, directional lights, and point lights. You can add a light by right-clicking on the Lights divider in the Assets Panel. You can also rename a light by pressing F2 when it is selected. Finally, left-clicking a light in the Assets Panel shows its properties in the Properties panel.

The Lights section of the Assets Panel lists all lights in the current project. You can instantiate any light into the current scene by dragging-and-dropping it onto the Render panel. The light will then appear at the scene origin, and FX Composer will do a zoom extents operation to ensure that the entire scene is in view. The drag-and-drop operation will also create a new scene node (in the Scenes section of the Assets Panel) with the light as a child node.

Alternatively, you can achieve the same result by dragging-and-dropping a light from the "Lights" section of the Assets Panel onto a scene node (in the "Scenes" section).

Cameras

FX Composer supports two types of cameras: orthographic and perspective. You can add a camera by right-clicking on the "Cameras divider in the Assets Panel. You can also rename a camera by pressing F2 when it is selected. Finally, left-clicking a camera in the Assets Panel shows its properties in the Properties panel.

The Cameras section of the Assets Panel lists all cameras in the current project. You can instantiate any camera into the current scene by dragging-and-dropping it onto the Render panel. The camera will then appear at the scene origin, and FX Composer will do a zoom extents operation to ensure that the entire scene is in view. The drag-and-drop operation will also create a new scene node (in the Scenes section of the Assets Panel) with the camera as a child node. Alternatively, you can achieve the same result by dragging-and-dropping a mesh from the Cameras section of the Assets Panel onto a scene node (in the Scenes section).

Working with Materials and Effects

Creating Asset Libraries

FX Composer's flexibility allows you to create and organize assets any way you choose. FX Composer provides two ways of creating asset libraries:

1. Create several new assets (effects, materials, images, lights, cameras, or geometry) in a new empty document. (The document must be active before creating the assets.)
2. Load a collection of COLLADA documents (each with its own assets). Then create a new empty document and drag-and-drop your chosen assets into the new document.

Scene Object Binding

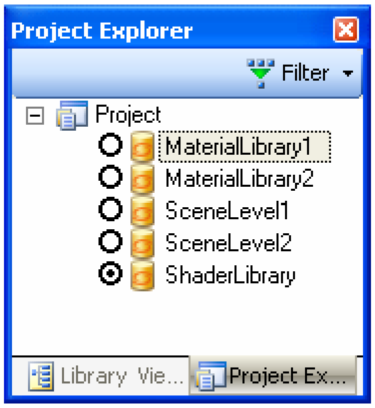
It is sometimes convenient to tie shader parameters to a scene object. For example, a lighting shader might be tied to a particular light source, meaning that the light position is input to the shader as a parameter. This means that the rendering will be updated correctly when you move the light.

This association between shader parameters and scene objects is called "scene object binding." In practice, you may want to associate more than just a position with a shader— in the case of a light, you may want to associate its diffuse and specular properties, specular exponent, and attenuation with a material. Instead of having to associate each of these individually, FX Composer allows you to simply select the object that should drive all the parameters simultaneously. You can specify this in the Properties panel when a material is selected

Asset Management

FX Composer uses COLLADA as its main asset storage format. This means that an FX Composer project file is made of a collection of COLLADA Document. Typical COLLADA databases are decomposed into COLLADA documents that group certain types of assets or collection of assets that belong to some segmentation of the content.

As an example, a typical production project may be organized as shown in Figure 39, where Effects would be in a ShaderLibrary and materials would be organized on a per-level basis. The materials would still reference the unique Effects in the shader library. Similarly, levels would have their objects and scenes in their own documents (one could dissociate geometries and scenes but this is just an example).



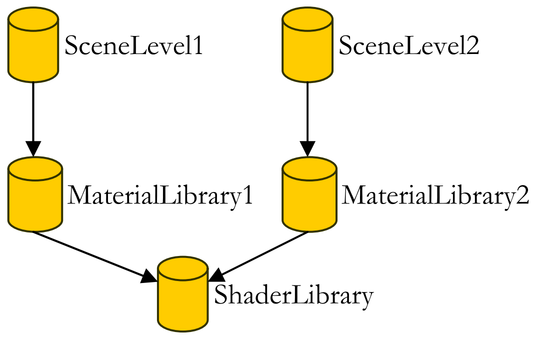


Figure 39: A typical COLLADA database organization

The obvious advantage of breaking down your asset database in such a way is to allow the collaboration of multiple team members to operate on distinct asset types and asset documents without stomping on each other. In our example, a TD should easily work on the Shader Library while artists could create materials that would be using the Effect Library. Additionally, game level creators could load the various material libraries and instantiate geometry in separate levels concurrently.

This setup can be nicely integrated into a source control system like Perforce, where each asset document can be checked in and out based on who is working on what.

In order to ease the integration with such file revision systems, FX Composer tracks the status of the Document file to lock or unlock the edition of assets contained in these documents. In our example, if you want to create materials in the MaterialLibrary1 document, you would check out the MaterialLibrary1 DAE file from Perforce. All other documents should be left locked and therefore preventing the user from modifying them.

Locked assets are shown as grayed out in the Properties Panel and rendered with a "lock" icon in the Assets Panel and the Project Explorer.

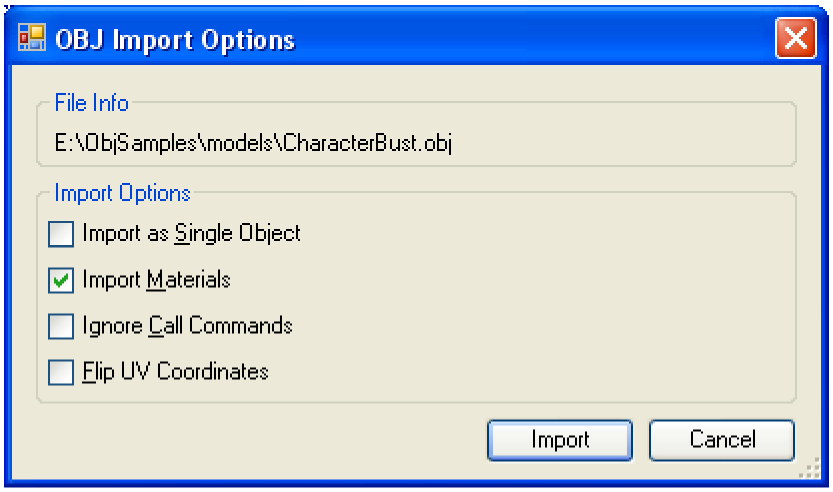
To unlock these assets, change the property of the Document that contains it by either using your file revision system or Windows Explorer.

File Importers

3D file importers

Wavefront OBJ file format

FX Composer supports the Wavefront OBJ file format. To import an OBJ file, go to File->Add->Import Existing Files… and select an OBJ file.



E:\ObjSamples\models\CharacterBust.obj

- [ ] Import as Single Object
- [✓] Import Materials
- [ ] Ignore Call Commands
- [ ] Flip UV Coordinates

Figure 40: OBJ file format import options

The OBJ file importer supports the following import options (See Figure 40):

- Import as Single Object: Consolidate all the mesh objects into one mesh
- Import Materials: Load the Materials (.mtl) and create COLLADA FX Common materials
- Ignore Call Commands: Do not import other .obj files that may be referenced within this obj file
- Flip UV Coordinates: applies the following uv transforms (which preserves the u coordinate and mirrors v):

$$\begin{cases} u' = u \\ v' = 1 - v \end{cases}$$

Importing an OBJ file will result in an instantiation of the geometry in the current scene and adding the geometry definition to the Library of Geometries. Saving a project that has imported OBJ geometry will be saved out as COLLADA geometry.

The source code of the OBJ importer is part of the FX Composer 2.0 SDK. (See section for finding out more information).

The OBJ file format usually comes with a sidekick file with the .mtl extension. This file contains the OBJ material definition. The following section explains how mtl materials are supported in FX Composer.

Wavefront MTL Support in FXC

For a full specification of .mtl files please see http://www.fileformat.info/format/material/

ALL STATEMENTS MENTIONED BELOW ARE SUPPORTED BY THE .MTL/.OBJ IMPORTER IN FXComposer2.0 UNLESS OTHERWISE STATED.

- **MTL Illumination Models**

The mtl file format has a predefined collection of illumination Models **reference by the** "illum_#" token where # can be a number from 0 to 10. The illumination models are summarized below:

MTL Illumination Models	Description
illum_0	Color on and Ambient off
illum_1	Color on and Ambient on
illum_2	Collada Blinn common profile
illum_3	Highlight on
illum_4	Transparency: Glass on, Reflection: Ray trace on
illum_5	Reflection: Fresnel on and Ray trace on
illum_6	Transparency: Refraction on, Reflection: Fresnel off and Ray trace on
illum_7	Transparency: Refraction on, Reflection: Fresnel on and Ray trace on
illum_8	Reflection on and Ray trace off
illum_9	Transparency: Glass on, Reflection: Ray trace off
illum_10	Casts shadows onto invisible surfaces

FX Composer translates these illumination models with COLLADA FX Common profiles as such:

MTL Illumination Models	FX Composer 2 COLLADA FX Common Profile
illum_0	Collada Constant common profile
illum_1	Collada Lambert common profile
illum_2	Collada Blinn common profile
illum_3	Not supported, default to Collada Phong common profile
illum_4	Not supported, default to Collada Phong common profile
illum_5	Not supported, default to Collada Phong common profile
illum_6	Not supported, default to Collada Phong common profile
illum_7	Not supported, default to Collada Phong common profile

illum_8	Not supported, default to Collada Phong common profile
illum_9	Collada Phong common profile
illum_10	Not supported, default to Collada Phong common profile

- **Colors and Float Modifiers**

Note: For all colors the use of CIEXYZ and Spectral files are not supported

e.g., Ka spectral file.rfl factor

Ka xyz x y z

Ambient Color (Ka r g b)
The Ka statement specifies the ambient reflectivity using RGB values.
"r g b" are the values for the red, green, and blue components of the color. The g and b arguments are optional. If only r is specified, then g, and b are assumed to be equal to r. The r g b values are normally in the range of 0.0 to 1.0. Values outside this range increase or decrease the reflectivity accordingly.

Importing to FX Composer is trivial; this simply becomes the ambient color of the material.

Diffuse Color (Kd r g b)
The Kd statement specifies the diffuse reflectivity using RGB values.
"r g b" are the values for the red, green, and blue components of the atmosphere. The g and b arguments are optional. If only r is specified, then g, and b are assumed to be equal to r. The r g b values are normally in the range of 0.0 to 1.0. Values outside this range increase or decrease the reflectivity accordingly.

Importing to FX Composer is trivial; this simply becomes the diffuse color of the material.

Specular Color (Ks r g b)
The Ks statement specifies the specular reflectivity using RGB values.
"r g b" are the values for the red, green, and blue components of the atmosphere. The g and b arguments are optional. If only r is specified, then g, and b are assumed to be equal to r. The r g b values are normally in the range of 0.0 to 1.0. Values outside this range increase or decrease the reflectivity accordingly.

Importing to FX Composer is trivial; this simply becomes the specular color of the material.

Note: Transmission Filter (Tf r g b) is not currently supported in FX Composer 2.

Transparency (d factor)
Specifies the dissolve for the current material.
"factor" is the amount this material dissolves into the background. A factor of 1.0 is fully opaque. This is the default when a new material is created. A factor of 0.0 is fully dissolved (completely transparent). Unlike a real transparent material, the dissolve does not depend upon material thickness nor does it have any spectral character. Dissolve works on all illumination models.

This is simply translated as the transparency of the material in FXC. The –halo option of this is not currently supported.

Specular Exponent (Ns exponent)
Specifies the specular exponent for the current material. This defines the focus of the specular highlight.
"exponent" is the value for the specular exponent. A high exponent results in a tight, concentrated highlight. Ns values normally range from 0 to 1000.

This value will be brought into the range between 0 and 128 via a simple ratio; e.g., a value of 1000 in the .mtl file will become 128 in FXC.

Note: Mapping of Sharpness (Sharpness value) is not currently supported in FX Composer 2.

Index of Refraction (Ni optical_density)
Specifies the optical density for the surface. This is also known as index of refraction.
"optical_density" is the value for the optical density. The values can range from 0.001 to 10. A value of 1.0 means that light does not bend as it passes through an object. Increasing the optical_density increases the amount of bending. Glass has an index of refraction of about 1.5. Values of less than 1.0 produce bizarre results and are not recommended.

This is translated directly into the index of refraction in the COLLADA common profile by FX Composer 2.

❑ Texture Maps

Ambient Texture Map (map_Ka -options args filename)
Specifies that a color texture file or a color procedural texture file is applied to the ambient reflectivity of the material. During rendering, the "map_Ka" value is multiplied by the "Ka" value.
"filename" is the name of an FX Composer supported image file.

FX Composer does not yet support the –options and arguments and will ignore them if they are present. The texture map for the ambient component will replace the ambient color in the collada common profile.

Diffuse Texture Map (map_Kd -options args filename)
Specifies that a color texture file or color procedural texture file is linked to the diffuse reflectivity of the material. During rendering, the map_Kd value is multiplied by the Kd value.
"filename" is the name of an FX Composer supported image file.

FX Composer does not yet support the –options and arguments and will ignore them if they are present. The texture map for the diffuse component will replace the diffuse color in the COLLADA Common profile; use this if you want a typical texture on your model's surface.

Specular Texture Map (map_Ks -options args filename)
Specifies that a color texture file or color procedural texture file is linked to the specular reflectivity of the material. During rendering, the map_Ks value is multiplied by the Ks value.
"filename" is the name of an FX Composer supported image file.
FX Composer does not yet support the –options and arguments and will ignore them if they are present. The texture map for the specular component will replace the specular color in the collada common profile.

Note: Specular exponent texture map (map_Ns -options args filename) is not currently supported by FX Composer.

Note: Disolve texture map (map_d -options args filename) is not currently supported by FX Composer.

Note: Texture map anti-aliasing (map_aat on) is not currently supported by FX Composer.

Note: Decal Texture mapping (decal -options args filename) is not currently supported by FX Composer.

Note: Displacement Texture mapping (disp -options args filename) is not currently supported by FX Composer.

Note: Bump Texture mapping (bump -options args filename) is not currently supported by FX Composer.

Note: Reflection mapping (refl …) is not currently supported by FX Composer.

Autodesk 3DS file format

FX Composer 2 supports the import of geometry from .3ds Autodesk files. Animation, Light and Camera object are ignored.

Microsoft X file format

FX Composer 2 supports the import of geometry from .X Microsoft files. Animation is completely ignored.

Autodesk FBX file format

FX Composer 2 supports the import of geometry from .FBX Autodesk files. Any additional asset type contained is ignored.

Image File Importer

Softimage® 3D Picture file format (PIC)

FX Composer supports the Softimage PIC file format. You can import PIC image files directly into the Textures Panel via the Assets Panel context menu from the Image section like any other file.

The source code of the PIC image loader is part of the FX Composer 2.0 SDK. (See section for finding out more information)

Scripting

This document describes using the scripting system in FX Composer 2 Beta. Please note that some of the details of the API and its usage may change before release.

Introduction

Scripting in FX Composer is typically accomplished using the Python scripting language. The implementation of Python used is IronPython 1 from Microsoft. IronPython supports all of the standard python syntax & library, with the additional advantage that it is implemented in a .NET language. This means that python scripting in FX Composer can call the rest of the FX Composer engine without additional work. It is important to note that none of the core functionality of FX Composer requires Python. The scripting is implemented entirely using a plugin (FXComposer.UI.Python), and in theory any scripting language that could communicate with the FX Composer engine could be used.

Scripting can be used for various purposes. It can be used to build and modify custom scenes, generate shaders, and geometry, or load and save collada documents, for example. Scripting examples come with the FX Composer install and demonstrate how to talk to the FX Composer engine.

The approach taken to FX Composer scripting is to allow the scripting to talk to all aspects of the engine. This means that a script author needs to be careful not to make calls into the engine that will cause instability. To assist with this, a script-friendly API layer has been provided – this layer ensures that scripting is easier to do and has no side effects, with the added advantage that all such commands are undoable. Whenever a script modifies the state of FX Composer, it should use one of the Fxc* API's.

Fxc* APIs

Most scripts make extensive use of the simple API's. These all begin with 'Fxc' followed by a meaningful name for the group of things they effect. For example, to work with an FX Composer scene (FXScene), the API commands are mainly in 'FxcScene'. To work with nodes, the commands are in 'FxcNode'. These classes are mainly made up of static methods that can be called, and usually take the object as their first parameter.

For example, to create & destroy a scene:

```
>>> FXScene scene = FxcScene.Create(“MyScene”);
>>> FxcScene.Destroy(scene);
```

To create & destroy a node:

```
>>> FXNode node = FxcNode.Create("MyNode");
>>> FxcNode.Destroy(node)
```

It is important to note that the FXNode & FXScene types are engine classes, which should not be modified directly. Calling Fxc* API's has the advantage that the call can be undone. Directly modifying the returned structures will cause complications and should be avoided. In the above cases, the return node and scene objects are passed into the Destroy API to remove them. The rest of this document will demonstrate many of the Fxc API's, and the reference help file also contains a complete list of the current methods.

Namespaces

FX Composer is split up into namespaces, as in any typical .NET project. FXComposer.Maths is the namespace for maths operations, FXComposer.Scene.Commands is the namespace for Fxc* API's that control the scene, FXComposer.IO is the namespace for IO modules, etc. In order to access classes in any of these namespaces, they must be imported. The Python scripting host automatically imports the key namespaces at start of day. The default python script that is run is called fxcapi.py, and here are the imports that are automatically run:

```
# Core python libraries
import sys
import System
import System.IO
import clr

# Core FX Composer assemblies
clr.AddReferenceByName("FXComposer.Core")
clr.AddReferenceByName("FXComposer.Maths")
clr.AddReferenceByName("FXComposer.Utilities")

# Import core
from FXComposer.Core import *

# Load all plugin assemblies
assemblies = FXRuntime.Instance.Plugins.PluginAssemblies
for assembly in assemblies:
            clr.AddReferenceByName(assembly.FullName)

# Common imports
from FXComposer.Core import *
from FXComposer.Core.Commands import *
from FXComposer.Maths import *
from FXComposer.Scene import *
from FXComposer.Scene.Commands import *
from FXComposer.Scene.Services import *
from FXComposer.Scene.Effects import *
from FXComposer.Scene.Control.Services import *
from FXComposer.Scene.Control import *
from FXComposer.Services import *
from FXComposer.Commands import *
```

Iron Python can be downloaded free from Microsoft, at the following address, and contains documentation for the CLR methods seen above. The release is useful for users wishing to see what's possible when scripting .NET applications, and for examples and tutorials on using IronPython.

http://www.codeplex.com/IronPython

Properties

Most objects in the FX Composer engine have a list of properties associated with them. The property list can be enumerated, or more commonly, properties can be directly accessed. To get the value of the current translation of a Node, we would do this:

```
>>> translation = node.Translation.PropertyValue
```

The 'PropertyValue' is the value held in the property. If we wanted the name of the property we would do this:

```
>>> name = node.Translation.Name
```

The result would be "Translation". Note that as mentioned above, we never directly modify this value, instead we would do:

```
>>> FxcNode.SetTranslation(node,node.Translation.PropertyValue +
FXVector3(1,0,0))
```

Here we have added a 1,1,1 translation value to the current one in the node, resulting in moving any objects in the scene attached to this node 1 unit to the right.

Services

Sometimes during scripting, a service in the engine will be called. Services are responsible for a particular task in FX Composer – for example, the task service manages a list of current tasks, renderport service manages a list of current renderports, etc.

An example of a service that might be called is:

```
>>> from FXComposer.IDE.Services import *
>>> FXProjectService.Instance.AddDocument(FXUri("c:/mydae.dae"))
```

Here we are first importing the service from its namespace (FXComposer.IDE.Services), then we are calling it directly to add a Collada document to the current project. Services are singletons, so have an 'Instance' static member which can be used to access them. To load a project, we use the same service:

```
>>> FXProjectService.Instance.OpenProject(FXUri("c:/myproj.fxcproj"))
```

Using the Scripting – Example Walkthrough

In this section we are going to walk through the basic steps involved in building a scene and drawing it in a viewport. We start by creating a scene…

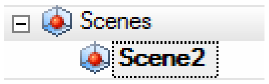
Create a Scene

Firstly we will create a new scene. FX Composer supports any number of scenes, and can even draw different scenes in different 'renderports'. In FX Composer, a renderport is simply a view of a scene. The Render Panel contains renderports for each backend device.

Firstly, we bring up the python panel in FX Composer, and type the following to create a scene:

```
>>> scene = FxcScene.Create()
```

If we go to the Assets Panel, we will see that a scene is now visible:



1. Testing Undo/Redo

Let's try the undo functionality. If we type:

```
Undo()
```

We will see the scene we created disappear. If we type:

```
Redo()
```

It will return. In this way we should be able to undo/redo any operations we perform using the Fxc API's.

There is an interesting catch here though.... The value of 'scene' in the python scripting window is now a reference to a disposed object. We can check this by typing:

```
>>> scene.IsDisposed
True
```

The response tells us that we do in fact have an old reference. We can fix it by getting the current active scene (since there is only one in this case):

```
>>> scene = FXSceneService.Instance.ActiveScene
>>> scene.IsDisposed
False
```

When we are looking at geometry later, we'll see another way of finding the scene by its Uri.

Manipulating the RenderPorts

Since we are going to build an OpenGL scene, we will ensure that the OpenGL window is visible, and get a reference to it. We can do this from the user interface by clicking the tab of course, but let's do it from script, which can be useful for automation:

```
port = FxcRender.FindRenderPort("OpenGL")
```

To activate the port, we use the command:

```
FxcRender.SetActiveRenderPort(port)
```

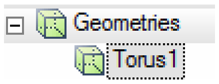
If you didn't have this renderport selected, you should find that it is now brought to the front in the scene panel.

2. Creating Some Geometry

To make our scene more interesting, lets add some geometry to the scene. Firstly, we will create a torus:

```
torus = FxcGeometry.CreateTorus("MyTorus")
```

If we again look in the Assets Panel we will see our torus geometry:



The torus is a complex object made up of a 'geopipe' and various ' geopipe objects'. The geopipe is just a container for the geometry, and the geopipe objects are the stages inside the geometry pipeine. We will see how to modify the torus later, but it is a good exercise to click it and view the properties in the property panel.

Note that we didn't get any geometry in the scene. This is because we need to instance the geometry into the scene we created. Using the user interface is slightly different – clicking on the "Add Torus" button in the UI will create the torus geometry *and* add it to the current scene. Try it to see this in action. Notice the geometry is created, and added to a scene. Don't forget to hit CTRL+Z to remove it before going to the next step.

What if we accidentally forgot to assign the 'torus' variable? How to we 'find' the created torus and reference it? Here are the steps:

```
>>> FxcGeometry.CreateTorus("Torus1")
<FXComposer.Scene.FXGeoPipe object at 0x000000000000002F
[fxcomposer://localhost/Document1#Torus1]>
>>> torus =
FXRuntime.Instance.UriManager.FindObject(FXUri("fxcomposer://localhost/
Document1#Torus1"))
```

The Uri we make here has to be the same Uri of the object we created. The previous command actually output the Uri for us in the script window. Another way to find out the Uri would be to click on the torus in the Assets Panel and look for the Uri property. This same method works for all items that have a Uri.

3. Adding Geometry to the Scene

To add our newly created geometry to the scene, we need to first create a node for it in the current scene, like so:

```
>>> node = FxcNode.Create()
```

And then we add this node to the root of the scene.

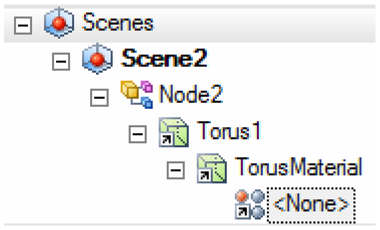
```
>>> FxcNode.SetParent(node, scene)
```

The SetParent call here is being used to set the parent of the node to be the scene; this makes the node a top-level node in the scene. Set Parent can also be called with another node to build a hierarchy of nodes.

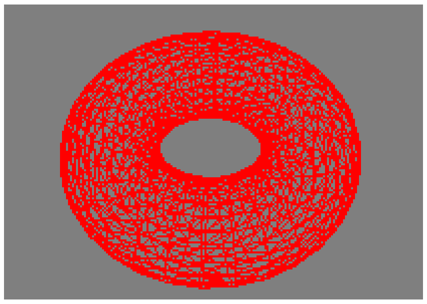
All we have to do to get our torus in the scene is to add it as a geometry instance under the node:

```
>>> torusInstance = FxcNode.CreateGeometryInstance(node, torus)
```

If we look in the Assets Panel, we can now see the node and the torus assigned to the scene. If we expand out the torus, we can see that there is currently no assigned material.



Checking the geometry in the viewport confirms this:



We can make the geometry fill the viewport by doing:

```
>>> scene.ZoomExtentsAll(1.0, 1)
```

The second parameter is a Boolean, which in python is passed as 1 or 0. The first parameter is the ratio of the current viewport. We could explicitly calculate this if we wished:

```
>>> port.Width.PropertyValue / port.Height.PropertyValue
```

4. Creating a material

In FX Composer, effects represent shaders designed to be run on one or more backend devices. They contain profiles, techniques, passes, parameters, etc. Materials, on the other hand, are instances of effects that are applied to geometry in the scene.

Typically the programmer builds the effect, and the artist manipulates the material by setting parameters. An effect might be 'metal', but an artist might tweak its parameters to make 2 materials, one that is shiny metal, and the other that is dull metal.

We need both an effect and a material for our torus.

Firstly, we create a material. That's easy:

```
>>> material = FxcMaterial.Create("BumpyShiny")
```

Note that this time we've decided to name the material asset. Most of the previous commands also have this option, to give assets a name.

There are many ways to create an effect. We are going to use the most simple. We will load our effect from a .cgfx effect file using an FxcEffect API:

```
>>> effect  =
FxcEffect.OpenEffectFile(FindShaderFile("bumpreflect.cgfx"),
"BumpyShinyEffect")
```

The API we call just requires the path to the effect file, and the name of the created effect. We've used a helper API, defined in fxcapi.py which makes this easy for us. It searches FX Composer's project and global paths for the shader filename we requested. Once we have the effect, we can tie the two together by referencing the effect from the material:

```
>> FxcMaterial.SetEffect(material, effect)
```

Finally, we need to add the material to the torus:

```
>> FxcGeometry.SetInstanceMaterial(torusInstance, material, "")
```

We use "" to indicate an empty material symbol. We could equally well have used "TorusMaterial", which is the default material symbol for the torus mesh. The empty symbol has the effect of applying our material to all material symbols on the torus. Symbols are applied to polygon groups; each model may have any number of material symbols which an instance material can be assigned to. The default shapes typically have one.

At this point we have created a scene, added geometry, and assigned a working material.

Semantic and Annotation Remapping

FX Composer includes a semantic remapper to allow developers to fine-tune the inputs that FX Composer will feed their effects.

In previous versions only a name remapping was possible, but in this latest release of FX Composer, a complete type remapping is now possible. For example, if your effects have special need of a forged matrix or vector, FX Composer now allows you to manipulate the default fixed semantics to make the input you want.

Numerous operators like Matrix/Vector operations are shipped with the product. This collection of operators can be extended by users using the FX Composer plugin system, meaning you can code the operator that suits your needs.

Syntax

FX Composer remappings are located in a file named "mappings.xml" located in <application root>\Plugins\scene\render\Data\.

If you open this file, you will notice that it contains several predefined remappings. We've included as many mappings so you don't have to create them yourself.

The system is based on a graph of what we call "operator nodes." These nodes are connected to form a graph that will have inputs from FX Composer and that will have an output calculated from the traversed nodes.

Here is a simple example of a Matrix Multiplication:

```
<Remapping name="Name">
    <identifiers>
        <parametername value="WorldView"/>
    </identifiers>
    <expression>
        <MatrixMultiply description="World * View">
            <input type="internalsemantic" value="world"/>
            <input type="internalsemantic" value="view"/>
        </MatrixMultiply>
    </expression>
</Remapping>
```

The Remapping Tag (line 1):

This tag specifies a block of remapping. The name parameter can be set to anything helpful to document what the remapping does.

The identifiers: (Lines 2 to 4)

This tag defines what needs to be matched in the shader file in order for this remapping to be used. In the previous Matrix Multiplication example, the only thing that we need in order to remap this parameter is that its name be ViewInv.

Other identifiers can be used such as:

<semantic name="LightPosition"/>

This option specifies that in order to be remapped, the name of the parameter semantic has to match.

<parametername value="LightPos"/>

This option specifies that the parameter has to match the name of the parameter itself.

<annotation name="object" value="PointLight0"/>

This option specifies that the parameter has to have the specific annotation/value pair in order to match.

All of these identifiers can be used at the same time, but the remapping will be active only if all the identifiers match.

An Expression Node (Line 5):

This node specifies the beginning of an expression block containing remapping operators.

An Operator node (line 6):

In this case, "MatrixMultiply" represents a node that will have two inputs and one output. The inputs are the two child nodes of MatrixMultiply.

Other types of operators include MatrixTranspose, MatrixInverse, DotProduct, CrossProduct, Cosine, Sine, Log, Ceiling, and MatrixSelect…

All operator nodes have a Description attribute that can be used for debugging. This description attribute is used when an error or warning is displayed by FX Composer. This allows you to debug and locate any remapping problems.

Two Input Nodes (line 7 and 8):

This operator node is created at runtime and two of its inputs come from FX Composer Internals. These internals are referenced using the input tag with a type set to "internalsemantic," telling FX Composer that this node will in fact be fed with an internal value. In our example, the world and view matrix.

The output of the operator is then used by its parent if it is a node, or is directly fed to the shader if it is the last node.

Operator nodes can be linked to make even more complex remappings. A list of complete examples is featured at the end of this user reference.

List of Operators

MatrixMultiply

Description

Used to multiply two matrices.

Input/Output

Direction	Type	Remarks
Input 1	Matrix	Left matrix
Input 2	Matrix	Right matrix
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixMultiply description=”World * View”>
        <input type="internalsemantic" name="world"/>
        <input type="internalsemantic" name="view"/>
</ MatrixMultiply >

<MatrixMultiply description="(World * View) * Projection”/>
        <MatrixMultiply description="World * View">
                <input type="internalsemantic" name="world"/>
                <input type="internalsemantic" name="view"/>
        </MatrixMultiply>
        <input type="internalsemantic" name="projection"/>
</MatrixMultiply>
```

MatrixAdd

Description

Used to add two matrices.

Input/Output

Direction	Type	Remarks
Input 1	Matrix	Left Matrix
Input 2	Matrix	Right Right
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixAdd description="World + View">
        <input type="internalsemantic" name="world"/>
        <input type="internalsemantic" name="view"/>
</ MatrixAdd >
```

MatrixSubstract

Description

Used to substract two matrices.

Input/Output

Direction	Type	Remarks
Input 1	Matrix	Left Matrix
Input 2	Matrix	Right Right
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixSubstract description=”World - View”>
        <input type="internalsemantic" name="world"/>
        <input type="internalsemantic" name="view"/>
</ MatrixSubstract >
```

MatrixInverse

Description

Used to get the inverse of a matrix.

Input/Output

Direction	Type	Remarks
Input	Matrix	
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixInverse>
        <input type="internalsemantic" name="view"/>
</MatrixInverse>
```

MatrixTranspose

Description

Used to get the transpose of a matrix.

Input/Output

Direction	Type	Remarks
Input	Matrix	
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixTranspose>
        <input type="internalsemantic" name="view"/>
</MatrixTranspose>
```

MatrixTransposeConditional

Description

Used to get the transpose of a matrix only of Input 2 is true.

Input/Output

Direction	Type	Remarks
Input 1	Matrix	
Input 2	Boolean	The transpose will only be returned if this input is true
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixTransposeConditional>
        <input type="internalsemantic" name="view"/>
        <input type=”internalsemantic” name=”isopengl”/>
</ MatrixTransposeConditional >
```

MatrixScale

Description

Used to multiply a matrix by a scalar factor.

Input/Output

Direction	Type	Remarks
Input 1	Matrix	
Input 2	float	
Output	Matrix	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixScale description="Matrix Scale">
        <input type="internalsemantic" value="world"/>
        <input type="float" value="0.1"/>
</MatrixScale>
```

MatrixOrthonormalize

Description

Used to orthonormalize a matrix.

Input/Output

Direction	Type	Remarks
Input	Matrix	
Output	Matrix	

P

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<MatrixOrthonormalize>
        <input type="internalsemantic" name="view"/>
</MatrixOrthonormalize >
```

MatrixSelect

Description

Used to select a row/column of a matrix.

Input/Output

Direction	Type	Remarks
Input	Matrix	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling
Mode	Row1-4, Column1-4	Options are not case sensitive

Remarks

The output vector will the size of the column/row selected.

Example

```
<MatrixSelect mode="Row4">
	<input type="internalsemantic" name="view"/>
</MatrixSelect>
```

VectorMultiply

Description

Used to multiply a vector by a scalar.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	float	
Output	Vector	

Parameters:

Name	Value	Remarks
Description		Used for error handling

Example

```
<RemappedSemantic name="myVectorMultiplication">
        <VectorMultiply>
                <input type="float3" value="1, 0, 0"/>
                <input type="float" value="2.0"/>
        </VectorMultiply>
</RemappedSemantic>
```

VectorAdd

Description

Used to add two vectors.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	Vector	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<RemappedSemantic name="myfloat3Addition">
        <VectorAdd>
                <input type="float3" value="1, 0, 0"/>
                <input type="float3" value="1, 0, 0"/>
        </VectorAdd>
</RemappedSemantic>
```

VectorSubtract

Description

Used to subtract two vectors.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	Vector	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<VectorSubtract>
        <input type="float3" value="1, 0, 0"/>
        <input type="float3" value="1, 0, 0"/>
</ VectorSubtract >
```

VectorLength/VectorLengthSq

Description

Used to get the length of a vector.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<VectorLength>
        <input type="float3" value="1, 0, 0"/>
</ VectorLength >


<VectorLengthSq>
        <input type="float4" value="1, 0, 0, 0.25"/>
</ VectorLengthSq >
```

VectorNornalize

Description

Used to normalize a vector.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<VectorNormalize>
        <input type="float3" value="1, 0, 0"/>
</VectorNormalize >
```

CrossProduct

Description

Used to calculate the cross product of two vectors.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	Vector	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<CrossProduct>
        <input type="float3" value="1, 0, 0"/>
        <input type="float3" value="0, 1, 0"/>
</CrossProduct >
```

DotProduct

Description

Used to calculate the dot product of two vectors.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	Vector	
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<DotProduct>
        <input type="float3" value="1, 0, 0"/>
        <input type="float3" value="0, 1, 0"/>
</DotProduct>
```

Cast

Description

Used to get the cast from a type to another.

Input/Output

Direction	Type	Remarks
Input	Vector	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling
To	Float2, float3, float4	

Example

```
<Cast to="float3">
        <MatrixSelect mode="Row4">
                <MatrixInverse>
                        <input type="internalsemantic" name="view"/>
                </MatrixInverse>
        </MatrixSelect>
</Cast>
```

TransformCoordinate

Description

Used to transform a coordinate value by a matrix.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	Matrix	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<TransformCoordinate>
        <input type="float3" value="0, 1, 0"/>
        <input type="internalsemantic" value="world"/>
</TransformCoordinate>
```

TransformNormal

Description

Used to transform a normal value by a matrix.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Input 2	Matrix	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<TransformNormal>
        <input type="float3" value="0, 1, 0"/>
        <input type="internalsemantic" value="world"/>
</ TransformNormal >
```

VectorSetValue

Description

Used to set the value of an individual component of a vector.

Input/Output

Direction	Type	Remarks
Input 1	Vector	The input vector to modify
Input 2	Float	The value to assign to the selected component
Output	Vector	The modified vector

Parameters

Name	Value	Remarks
Description		Used for error handling
Operation	Char	Examples: "x" "y" "a" …

Remarks

This operator creates a vector of the length of operation parameter. It then fills the vector, component by component based on the order the parameters.

Example

```
<VectorSetValue component="z">
        <input type="float3" value="1, 2, 3"/>
        <input type="internalsemantic" value="currenttime"/>
</VectorSetValue>
```

This will result in an output of: "1, 2, 1233.021321" (last number being the current time).

Swizzle

Description

Used to transform a normal value by a matrix.

Input/Output

Direction	Type	Remarks
Input 1	Vector	
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling
Operation	String	Examples: xyz yxwz argb rgba ...

Remarks

This operator creates a vector of the length of operation parameter. It then fills the vector; component by component based on the order of the parameters.

Examples

```
<Swizzle operation="yxz">
            <input type="float3" value="1, 2, 3"/>
</Swizzle>
```

This will result in an output of: "2, 1, 3."

```
<Swizzle operation="wwyz">
            <input type="float4" value="1, 2, 3, 4"/>
</Swizzle>
```

This will result in an output of: "4, 4, 2, 3."

Demux

Description

Used to separate a vector into individual outputs.

Input/Output

Direction	Type	Remarks
Input	Vector	
Output	*n* individual pins	*n* being the size of the vector

Parameters

Name	Value	Remarks
Description		Used for error handling
Ooutputsize	Number (1, 2, 3, 4)	Determines the number of output pins to create from the input vector. It usually is the dimension of the vector.

Example

```
<DeMux outputsize="2">
        <MatrixSelect mode="Column1">
                <input type="internalsemantic" value="world"/>
        </MatrixSelect>
</DeMux>
```

Mux

Description

Used to concatenate output pins into a single vector of a given dimension.

Input/Output

Direction	Type	Remarks
Input	*n* individual pins	*n* being the size of the vector
Output	Vector	

Parameters

Name	Value	Remarks
Description		Used for error handling
Inputsize	Number (1, 2, 3, 4)	Determines the number of inputs that the parser will have to look for. This will also be the size of the output vector.

Example

Creating a float3 with currenttime, lasttime and a constant:

```
<Mux inputsize="3">
	<input type="internalsemantic" value="currenttime"/>
	<input type="internalsemantic" value="lasttime"/>
	<input type="float" value="0.025"/>
</Mux>
```

Cosine/ACosine/Sine/ASine/Tangent/ATangent

Description

This operator is used to calculate the sine, cosine and tangent of the input value.

Input/Output

Direction	Type	Remarks
Input	Float, double	
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<Cosine>
        <input type="internalsemantic" value="currenttime"/>
</Cosine>

<Tangent>
        <input type="internalsemantic" value="currenttime "/>
</Tangent>
```

Log

Description

Returns the logarithm of a number.

Input/Output

Direction	Type	Remarks
Input	Float	
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<Log>
        <input type="internalsemantic" value="currenttime"/>
</Log>
```

Pow

Description

Returns x raised to the power of y.

Input/Output

Direction	Type	Remarks
Input1	Float	
Input2	Float	Power
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<Pow>
        <input type="internalsemantic" value="currenttime"/>
        <input type="float" value="2.0"/>
</Pow>
```

Add/Substract/Multiply/Divide/Modulus

Description

Returns performed operation.

Input/Output

Direction	Type	Remarks
Input1	Float	
Input2	Float	
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
<Add>
        <input type="internalsemantic" value="currenttime"/>
        <input type="float" value="2.0"/>
</Add>

<Substract>
        <input type="internalsemantic" value="currenttime"/>
        <input type="float" value="2.0"/>
</Substract>
```

Floor

Description

Returns the largest integer less than or equal to the specified decimal number.

Input/Output

Direction	Type	Remarks
Input	Float	
Output	Float	

Parameters

Name	Value	Remarks
Description		Used for error handling

Example

```
< Floor >
        <input type="internalsemantic" value="currenttime"/>
</ Floor >
```

Ceiling

Description

Returns the smallest integer greater than or equal to the specified decimal.

Input/Output

Direction	**Type**	**Remarks**
Input	Float	
Output	Float	

Parameters

Name	**Value**	**Remarks**
Description		Used for error handling

Example

```
< Ceiling >
        <input type="internalsemantic" value="currenttime"/>
</ Ceiling >
```

Input (input)

Description

This node is used to reference an internal FX Composer 2 Semantic.

Input/Output

Direction	Type	Remarks

Parameters

Name	Value	Remarks
Type	Internalsemantic	
Value	World, view, projection, isopengl, mouseposition, time, lasttime, elapsedtime, random	

Name	Value	Remarks
Type	Float	
Value	Floating point value, pi	

Name	Value	Remarks
Type	Float2, float3, float4	
Value	Floating point values separated by comas	

Name	Value	Remarks
Type	Light, camera, node	The type of object you want to bind this semantic to
Name	String	Any group name that will represent the light, camera or object node you want to link
Value	Position, direction, color	The parameter you want to link
Space	World, object, view	Default is world

Examples

```
<MatrixTranspose>
        <input type="internalsemantic" name="view"/>
</MatrixTranspose>


<input type="float3" value="1, 2, 3"/>
```

Programming Your Own Operator Nodes

FX Composer 2 gives you the opportunity to add operator nodes that suit your needs. The concept used is the same as for regular FX Composer plugins.

The remapping is based on a graph composed of Remapping Operator Nodes.

- Each node has inputs and outputs that can be of various types. (FXMatrix, double, float…)
- Each input and output of these operators are connected to other operators.
- Every operator node has to inherit from FXSemanticMappingNode and implement the IFXPlugin Interface.

Integration into FX Composer

The operator nodes are loaded into FX Composer 2 using an fxplug file. Refer to main documentation for more information.

Here is an example of a declaration:

```
<virtualdirectory path="FX Composer/remappingoperators">
        <class name=" DummyNamespace.VectorLength"/>
</virtualdirectory>
```

Note that the virtual directory of the operator nodes has to be : "FX Composer/remappingoperators".

FX Composer will look for operators inside this virtual directory.

Naming Convention

FX Composer semantics remapping relies on string comparisons to figure out which operator to create from the xml nodes inside the file. Therefore a naming convention has been established. An operator node must have a name ending by "Node".

The name inside the xml is used without the "Node" suffix.

For the case below, the xml file would look like:

```
<VectorLength>
…
</VectorLength>
```

FX Composer then adds the suffix and looks for the type of the operator in its database.

Simple Example: VectorLength

In this case, this operator only takes one input and outputs one value.

```
public class VectorLengthNode : FXSemanticMappingNode
{
	private FXProperty<float> _Output;
	public override IFXProperty Output
	{
		get { return _Output; }
	}

	private FXProperty<FXMatrix> _Input;
	public override IFXProperty Input
	{
		get { return _Input; }
	}

	public VectorLengthNode()
	{
		_Output = new FXProperty<float>(this.Properties, "Length", "");
		_Input = new FXProperty<FXMatrix>(this.Properties, "Input Vector", "");
		_Input.PropertyValue = new FXMatrix(1, 3);
		_Output.DependsOn(_Input);
	}

	public override void EvaluateProperties()
	{
		FXMatrix mat = _Input.PropertyValue;
		FXVector3 vector = new FXVector3((float)mat[0, 0], (float)mat[0, 1],
(float)mat[0, 2]);
		_Output.PropertyValue = vector.Length();
	}
}
```

Remarks:

The constructor creates the properties and assigns a default value to the input.

The Evaluate properties method is called whenever this operator node has to be used.

Input and Output properties are used to access the internal fields.

Case of Multiple Inputs/Output

In the following case, a Matrix Scale, the operator needs two inputs. For that case, a property called InputPins is overridden from the base class. The getter of this property returns an array of IFXProperty containing the inputs in the order that they should appear in the xml file.

In the case below, it creates an array containing the internal input matrix, and the scalar value. They should appear in the same order in the mapping.xml file.

Example:

```
<MatrixScale description="Matrix Scale">
        <input type="internalsemantic" value="world"/>
        <input type="float" value="0.1"/>
</MatrixScale>
```

Here is the code for a MatrixScaleNode:

```
public class MatrixScaleNode : FXSemanticMappingNode, IFXPlugin
{
        private FXProperty<FXMatrix> _Output;
        public override IFXProperty Output
        {
                get { return _Output; }
        }

        private FXProperty<double> _Scalar;
        public FXProperty<double> Scalar
        {
                get { return _Scalar; }
                set { _Scalar = value; }
        }

        private FXProperty<FXMatrix> _Input;

        public override IFXProperty[] InputPins
        {
                get
                {
                        return new IFXProperty[] { _Input, _Scalar};
                }
        }

        public MatrixScaleNode()
        {
                _Output = new FXProperty<FXMatrix>(this.Properties, "Output Vector",
"");
                _Input = new FXProperty<FXMatrix>(this.Properties, "Input Matrix", "");
                _Input.PropertyValue = FXMatrix.Identity4x4;
                _Scalar = new FXProperty<double>(this.Properties, "Scalar", "");
                _Output.DependsOn(_Input);
        }

        public override void EvaluateProperties()
        {
                FXDebug.CheckArgument(_Input.PropertyValue, "Input pin cannot be
null");
```

```
            double scalar = _Scalar.PropertyValue;
            FXMatrix mat = _Input.PropertyValue * FXMatrix.Scaling((float)scalar,
                                                                   (float)scalar,
                                                                   (float)scalar);
            _Output.PropertyValue = mat;
        }

        #region IFXPlugin Members
        public bool Build(IFXPluginType pluginType)
        {
            return true;
        }
        #endregion

}
```

Complete Examples

Retrieving the Camera Position from the View Matrix

```
<Remapping name="">
        <identifiers>
                <semantic value="myCameraPosition"/>
        </identifiers>
        <expression>
                <Cast to="float3">
                        <MatrixSelect mode="Row4">
                                <MatrixInverse>
                                        <input type="internalsemantic" value="view"/>
                                </MatrixInverse>
                        </MatrixSelect>
                </Cast>
        </expression>
</Remapping>
```

Calculating the World x View x Projection Matrix

```
<Remapping name="">
        <identifiers>
                <semantic value="myWorldViewProjection"/>
        </identifiers>
        <expression>
                <MatrixTransposeConditional description="">
                        <MatrixMultiply description="(World * View) * Projection">
                                <MatrixMultiply description="World * View">
                                        <input type="internalsemantic" value="world"/>
                                        <input type="internalsemantic" value="view"/>
                                </MatrixMultiply>
                                <input type="internalsemantic" value="projection"/>
                        </MatrixMultiply>
                        <input type="internalsemantic" value="isopengl"/>
                </MatrixTransposeConditional>
        </expression>
</Remapping>
```

Calculating the World Inverse Transpose

```
<Remapping name="">
        <identifiers>
                <semantic value="myWorldInverseTranspose"/>
        </identifiers>
        <expression>
                <MatrixTransposeConditional description="OGL Render devices">
                        <MatrixTranspose description=" ">
                                <MatrixInverse description=" ">
                                        <input type="internalsemantic" value="world"/>
                                </MatrixInverse>
                        </MatrixTranspose>
                        <input type="internalsemantic" value="isopengl"/>
                </MatrixTransposeConditional>
        </expression>
</Remapping>
```

Computing the Angle Between Two Vectors in Degrees

```
<Remapping name="">
        <identifiers>
                <semantic value="myVectorAngleDegrees"/>
        </identifiers>
        <expression>
                <Divide>
                        <Multiply>
                                <ACosine>
                                        <DotProduct>
                                                <VectorNormalize>
                                                        <input type="float3" value="1, 1, 0"/>
                                                </VectorNormalize>
                                                <input type="float3" value="0, 1, 0"/>
                                        </DotProduct>
                                </ACosine>
                                <input type="float" value="180"/>
                        </Multiply>
                        <input type="float" value="pi"/>
                </Divide>
        </expression>
</Remapping>
```

Computing the Angle Between Two Vectors in Radians

```
<Remapping name="">
        <identifiers>
                <semantic value="myVectorAngleRadians"/>
        </identifiers>
        <expression>
                <ACosine>
                        <DotProduct>
                                <VectorNormalize>
                                        <input type="float3" value="1, 1, 0"/>
                                </VectorNormalize>
                                <input type="float3" value="0, 1, 0"/>
                        </DotProduct>
                </ACosine>
        </expression>
</Remapping>
```

Scripting

FX Composer supports scripting through any compliant .NET language using plugins. FX Composer contains a Python scripting window that can be used to control the engine using Python. This feature is not complete yet due to pending updates to the Python runtime, which will happen after this alpha. A sample command list might be:

```
from fxcapi import *
Reset()
Teapot()
Translate(1,1,1)
Teapot();
Undo()
Undo();
Undo()
Redo();
import FXComposer.Scene
dir(FXComposer.Scene)
```

Scripting can be used for task automation, interprocess communication, changes and additions to the UI, and more. For information on the "IronPython" version of Python used in FX Composer 2, see the IronPython community Web page at http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython

List of Commands

Command list goes here.

Scripting Toolbars

FX Composer allows you to create custom toolbars with buttons that execute Python scripts. For more information, please see the section on Working with Layouts.

Sample Scripts

Various use cases:

- Convert CgFX/.fx to COLLADA FX
- Binding light objects to parameters
- Assigning shaders based on name

FX Composer 2 in Your Production Pipeline

This chapter discusses the various ways you can integrate FX Composer 2 into your pipeline.

FX Composer 2–Centric

In this scenario you would use FX Composer to read in shader effects, COLLADA files, and geometry from DCC applications. You would then create and modify scenes in FX Composer (material assignment, parameter binding, and tweaking) and export the scenes through COLLADA and COLLADA FX. This cross-platform output can then be conditioned for multiple platforms, such as PCs, Macs, and consoles.

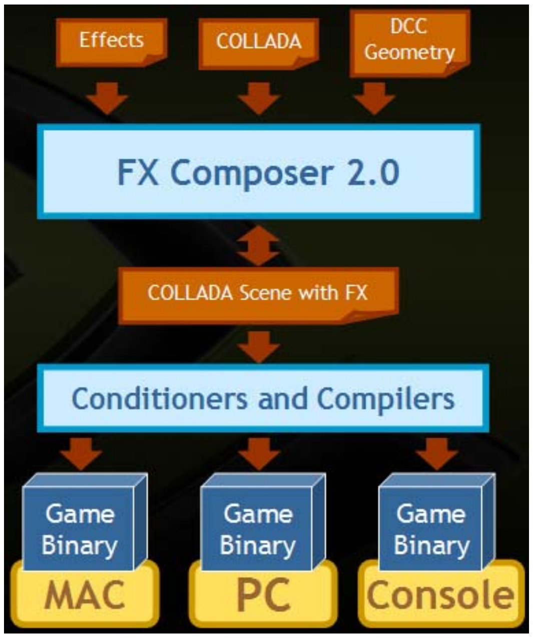


Figure 41. FX Composer-Centric Workflow

Effect Library Creator

FX Composer can be used to group numerous shader effects into a COLLADA document library. The DCC application then loads the COLLADA library, allows you to manipulate the scene, and then exports it to a conditioner, which prepares it for a specific platform such as a console.

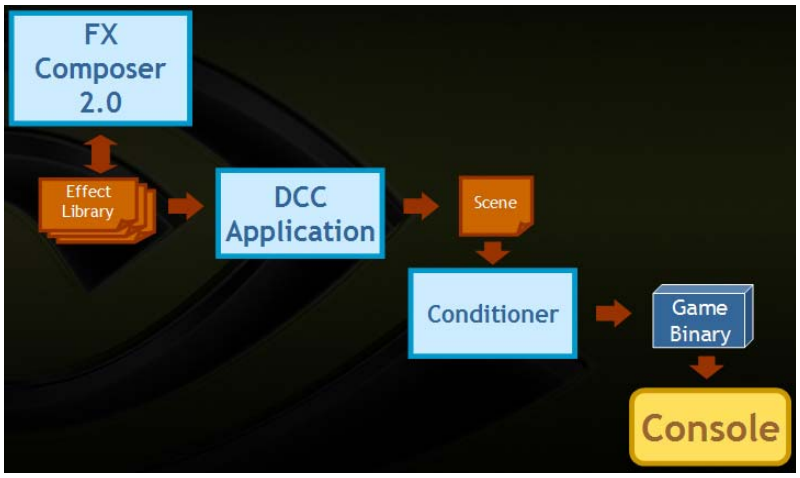


Figure 42. Workflow for Effect Library Creation

Engine

You can also use FX Composer as a complement to your game engine. In this case, FX Composer reads in shader files and imports game geometry so you can author, assign, and tweak shaders in FX Composer. This data would then be visible directly in the game engine via an interprocess communications interface (for example, a rendering game adapter).

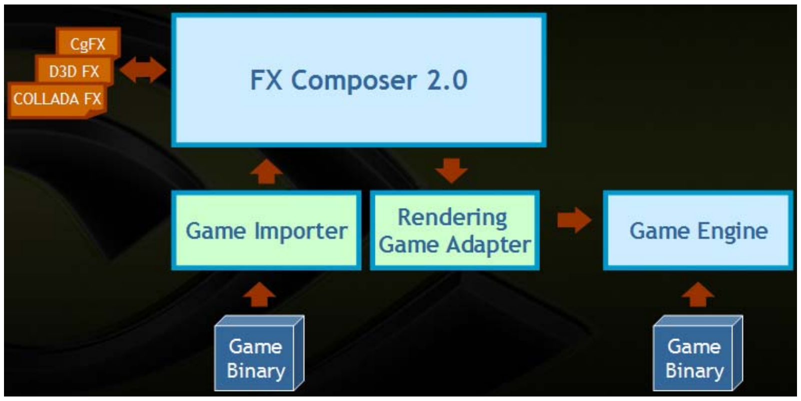


Figure 43. Workflow for Engine Integration

Shader Library

By creating an empty COLLADA file, importing a series of CgFX effects, converting them to COLLADA FX effects, and saving the file, you can conveniently create a library of COLLADA FX Cg effects. Shader libraries are useful for organizing and logically grouping various shader effects.

Release Notes

Please read the following release notes before using FX Composer 2.

1. Make sure to use a Release 80 or newer driver for full OpenGL and Frame Buffer Object functionality. We recommend the latest public driver available at http://www.nvidia.com/content/drivers/drivers.asp.
2. The following limitations exist in this alpha release of FX Composer 2:
    a. Per-object SAS scripts are not yet supported. In practice, all non-scene effects are by definition per-object, but these will only break if they do SAS scripting inside the pass, and a small number of our effects actually do this (less than 5). An example is "fur.fx."
    b. COLLADA FX supports a new layer syntax for expressing full-screen effects. The alpha does not yet read this information; thus, full-screen effects will only work with .cgfx/.fx shaders on this build.
3. You may see a message from FX Composer saying, "FX Composer has detected that an application or user has altered some Cg components in a way that is incompatible with FX Composer. In order to fix this problem, please re-install FX Composer. While this incompatibility exists, FX Composer will automatically disable the OpenGL renderer plugin."

    This message will be displayed if the Cg DLLs in the FX Composer directory are modified. Re-installing FX Composer will replace the DLLs with their original versions, and will not affect any other applications on your system that use Cg.
4. When loading a COLLADA file, FX Composer Beta now reports any errors found while validating the file against version 1.4.1 of the COLLADA schema. Even if any errors are found, FX Composer will try to load the file as best as it can. We highly recommend that you repair any COLLADA files that contain errors when possible, even if they load successfully.

Detailed Tutorial

Creating new shaders in FX Composer is a straightforward procedure. The easiest thing to do is to copy an existing shader and start modifying—this will save you a lot of effort compared to building shaders completely from scratch. With this in mind, a number of effects files are provided with the package.

Coding Conventions

Whether starting from a template, or writing a new shader from scratch, you should keep basic elements and coding styles consistent between shaders and their usages. This section describes the standards generally used in the example shaders. While not required, they are offered as suggestions to help maintain consistency and readability. Whatever the preference of an individual shader writer and their workmates, a consistent approach to your shaders is a Best Practice.

Naming

The names below are loosely based on common computer graphic mathematical practice reaching back to the late 1970s, and partly on names made common by early offline shading languages such as the RenderMan SL. In general, the two prime motivations are that names be ***descriptive*** and at the same time ***easy to type.*** Easy typing minimizes mistakes. These naming conventions are therefore not as strict as systems like "Hungarian naming."

Different languages imply different naming conventions; for example, GLSL contains predefined names like "gl_FragColor." In general, the naming used here is based, where applicable, on the names used in standard Cg/HLSL annotations and semantics. This makes shaders easier to understand across platforms.

In the following list, the portion of a name in [brackets] is optional (not an array index).

`Pos[ition]`
A point location passed into a shader, usually directly from the vertex buffer.

`HPos[ition]`
A point transformed into [xyzw] homogeneous clip-space coordinates.

`P[x]`
A transformed point in coordinate system x; for example, Po: object coordinates, Pw: world coordinates, Pl: light-source coordinates, Pt: texturing coordinates, Pn: noise-calculation coordinates, Ph: homogeneous clipspace coordinates. With no suffix, just "P" is the same as "Pos".

E[x]
Mostly present in vertex shaders, E indicates the location of the eye point. Ew: Eye point in world coordinates

L[x]
A light-direction vector. Without [x], the vector is in "raw" unprocessed form (for example, in a fragment shader receiving an unnormalized light vector from the vertex unit). An "n" on vector names usually indicates that the vector has been normalized; for example, Ln: normalized "L."

V[x]
A view vector, similar in usage to L[x]. Most-commonly seen normalized as Vn.

*Name*Xf
Any 4×4 or 4×3 transform matrix (other matrix types, say for color conversions, would not be marked "Xf.") Typically "Name" is the name of the coordinate system, as in"WorldXf."

WVPXf/wvpXf
A special case where ease-of-typing dominates descriptiveness, "wvp" is much easier to type than the concatenation "WorldViewProjection."

Name[I][T]Xf
Inverse and/or **T**ransposed transformation matrices. Again, ease-of-typing wins over fully descriptive but finger-fumbling names like "WorldViewProjectionInverseTransposeXf."

*Name*Color
Values that are colors should be clearly named to distinguish them from similar scalar values. As a bad example, it is not uncommon to see shaders that use the name "diffuse" to variously describe a scalar, a surface color, and even a texture sampler. Be sure to use an adequately descriptive name.

Ka
Scalar multiplier for any ambient color.

Kd
Scalar multiplier for any diffuse surface color.

Ks
Scalar multiplier for any specular surface color.

SpecExp/PhongExp
The exponent used in calculating phong specular nodes. Higher exponents (above 8 or 12) will create a smaller, more-focused highlight.

Kr
Scalar multiplier for any reflective (for example, from a cube map) surface color.

`KrMin`

Scalar multiplier for reflectivity when accounting for Fresnel attenuation. If KrMin is specified, the reflectivity strength will vary from KrMin (viewed directly along the surface normal) to Kr (along the contour edges).

`FresExp`

The exponent used when calculating the Schlick approximation of Fresnel's equation. In Schlick's paper, he used the value of 5. Recent writers have suggested 4 as a more-useful default, and lower values like 3 will create a more deep-gloss "lacquered" appearance.

`Name``Tex[ture]/`***`Name`***`Samp[ler]`

In general, it is a good idea to declare both the file texture and the associated hardware sampler together, colocated within the file and both sharing the sample root name. Resist the urge to use less-clear labels like "Map," which may potentially apply to either a texture or a sampler. Be sure to distinguish between Bump and Normal maps.

`UV[#]`

TEXCOORD register variables that are expressly being used for decal texture lookup (or as image processing coordinates). "UV" is clear but less typo-prone than "TEXCOORD"

Technique Names

If only one technique is provided, it is usually called "main." Otherwise, the name should be distinctive and descriptive to a nontechnical reader (for example, an artist using the shader). Longer names like "fragmentShadedTextured" are encouraged. Names like "textured," "untextured," or "debug_UVs" might be common.

Pass Names

Pass names should either be descriptive of function ("blurHorizontal") or at least their order of operation ("p0" and "p1" and so on.

Other Coding Conventions

"Tweakables" and "Untweakables"

Dynamic global parameters that are automatically tracked by the application, such as matrices with the semantic WORLDVIEWPROJECTION, are generically called "untweakables" because they cannot be directly tweaked by the artist/user. Such untweakables are usually collected in a group declaration within an effect file, and have their display widgets set to "None" so that they do not appear in the FX Composer Properties panel, where their uneditable presence would only create confusion (and slow the frame rate).

All other global parameters, which can be directly tweaked and *do* appear in the properties panel, are therefore called "tweakables."

Connection Structs

Most example shaders use a struct to collect their special-register inputs and outputs, where each member of the struct has the appropriate register semantic (for example, ":TANGENT0"). This is not a language requirement, but makes debugging much easier.

Unless multiple render targets (MRTs) are involved, fragment programs are usually declared with a :COLOR output semantic on the function, rather than using an "out" declaration on one of the program's formal parameters. The uses are equivalent, but using the "float4 function_name(): COLOR" semantic is clearer.

In general, functions are declared after global parameters, so that those parameters can be used without declaration. In some cases, the author may prefer to pass values as direct formal parameters to the shader. Most often this will occur when a single function may be repeated on multiple passes—say once for each light, where the light positions and colors are passed as function parameters.

"Ambient" versus "Incandescent" Light

Some shader writers add the scene's ambient color to the shaded surface color. Instead, the ambience should be multiplied by the surface color (and textures) so that contrast is not lost. If a color should be explicitly added to the surface after all other shading, that color should be labeled as "Incandescence" rather than "Ambience."

"World" Coordinates

Many of the sample shaders are written to use "World" coordinates. Many shaders are written to calculate shading in eye coordinates, tangent space, and so on. These are all valid options. Most of the shaders written to light in "World" space do so because it's a naturally good coordinate system for using and/or adding reflection maps.

A Sample Shader

Now that the language is clearer, let's look at and modify a simple sample shader effect. Let's make a copy of "simple_Cg_example.cgfx" in Windows Explorer, and call it "bluePlastic.cgfx." Make sure that the file is writeable.

We'll apply this shader to a COLLADA dae model, and save it. You can use any model file; we'll be using "Widgie2.dae," which is a simple one-node model. Make a copy of "Widgie2.dae" that's write-enabled, just like the shader.

Next, open FX Composer 2. Create an empty Project ("File→New→Project…") called "Tutorial."

Now drag your copy of "Widgie2.dae" from Windows Explorer onto the Render panel—the model will appear in red wireframe.

Now drag the shader "bluePlastic.cgfx" from Windows Explorer and drop it onto that red wireframe model. Wait a moment for the shader to compile and you'll see something like this (Figure 44):

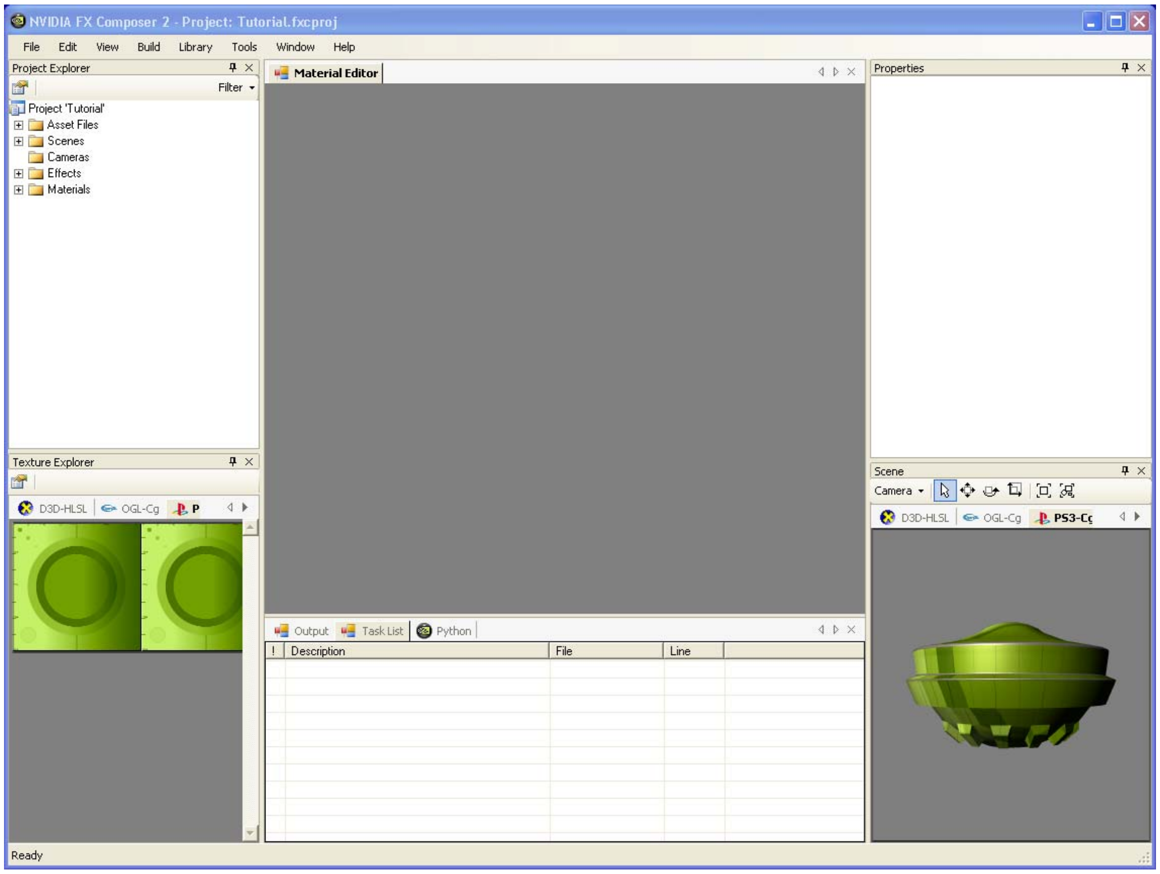
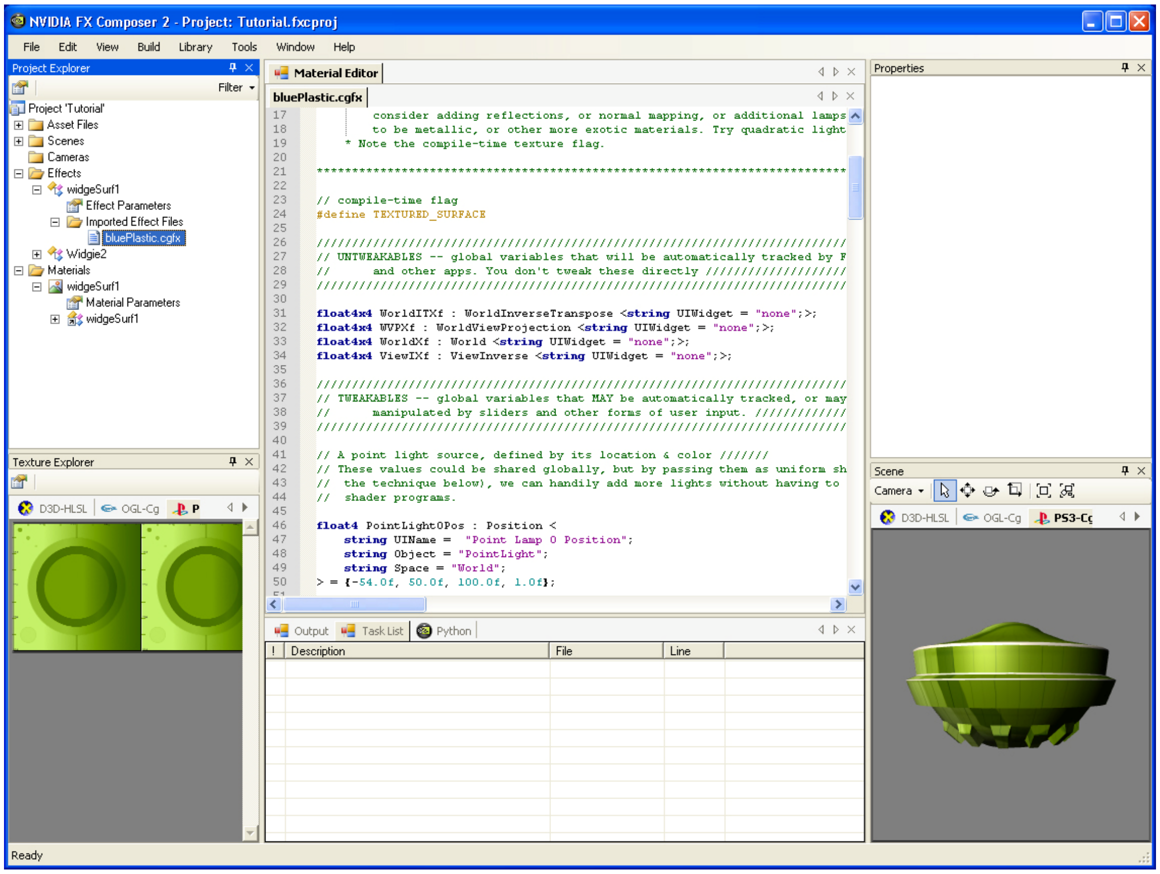


Figure 44. Unedited Sample Shader

This looks good but it's hardly blue, so we'll need to touch up the shader a bit.

FX Composer has automatically connected the shader to the preexisting material (from the DCC application that created the original "Widgie2"). In FX Composer Project Explorer, open the "Effects" folder, and then the effect "widgeSurf1." Inside that effect you'll find the folder "Import Effect Files" and inside that "bluePlastic.cgfx." Double-click "bluePlastic.cgfx" to load the source code directly into Code Editor.



```
        consider adding reflections, or normal mapping, or additional lamps
        to be metallic, or other more exotic materials. Try quadratic light
    * Note the compile-time texture flag.

****************************************************************************

// compile-time flag
#define TEXTURED_SURFACE

////////////////////////////////////////////////////////////////////////////
// UNTWEAKABLES -- global variables that will be automatically tracked by F
//      and other apps. You don't tweak these directly /////////////////////
////////////////////////////////////////////////////////////////////////////

float4x4 WorldITXf : WorldInverseTranspose <string UIWidget = "none";>;
float4x4 WVPXf : WorldViewProjection <string UIWidget = "none";>;
float4x4 WorldXf : World <string UIWidget = "none";>;
float4x4 ViewIXf : ViewInverse <string UIWidget = "none";>;

////////////////////////////////////////////////////////////////////////////
// TWEAKABLES -- global variables that MAY be automatically tracked, or may
//      manipulated by sliders and other forms of user input. //////////////
////////////////////////////////////////////////////////////////////////////

// A point light source, defined by its location & color ///////
// These values could be shared globally, but by passing them as uniform sh
//  the technique below), we can handily add more lights without having to
//  shader programs.

float4 PointLight0Pos : Position <
    string UIName =  "Point Lamp 0 Position";
    string Object = "PointLight";
    string Space = "World";
> = {-54.0f, 50.0f, 100.0f, 1.0f};
```

Figure 45. Source Code Loaded in Editor

Let's look at that source code (skipping a few comments). There's a fair chunk of text, but mostly it's all stock parts, and already typed-in for use, so all we need to do is look it over broadly. We'll see that there are three basic sections: declarations of variables, actual shader functions, and a "technique" block that collects the entire effect together:

```
#define TEXTURED_SURFACE

//////////////////////////////////////////////////////////////////////
/////////////////
// UNTWEAKABLES -- global variables that will be automatically tracked
by FX Composer//
//      and other apps. You don't tweak these directly
////////////////////////////////
//////////////////////////////////////////////////////////////////////
/////////////////

float4x4 WorldITXf : WorldInverseTranspose <string UIWidget =
"none";>;
float4x4 WVPXf : WorldViewProjection <string UIWidget = "none";>;
float4x4 WorldXf : World <string UIWidget = "none";>;
float4x4 ViewIXf : ViewInverse <string UIWidget = "none";>;

//////////////////////////////////////////////////////////////////////
/////////////////
// TWEAKABLES -- global variables that MAY be automatically tracked,
or may be ////////
//      manipulated by sliders and other forms of user input.
/////////////////////////
//////////////////////////////////////////////////////////////////////
/////////////////

float4 PointLight0Pos : Position <
   string UIName =  "Point Lamp 0 Position";
   string Object = "PointLight";
   string Space = "World";
> = {-54.0f, 50.0f, 100.0f, 1.0f};

float3 PointLight0Col : Diffuse <
   string UIName =  "Point Lamp 0";
   string UIWidget = "Color";
   string Object = "PointLight";
> = {1.0f, 0.95f, 0.85f};

// Ambient light everywhere in the scene
//////////////////////////////////

float3 AmbiColor : Ambient
<
   string UIName =  "Ambient Light";
   string UIWidget = "Color";
> = {0.07f, 0.07f, 0.07f};

// Surface attributes //////////////////////////

float3 SurfColor : Diffuse
<
   string UIName =  "Surface";
   string UIWidget = "Color";
> = {1.0f, 1.0f, 1.0f};

float SpecExpon : SpecularPower
<
   string UIName =  "Specular Power";
   string UIWidget = "Slider";
   float UIMin = 1.0;
   float UIMax = 128.0;
   float UIStep = 1.0;
> = 25.0;

float Kd
<
   string UIName =  "Diffuse Brightness";
   string UIWidget = "Slider";
   float UIMin = 0.0;
   float UIMax = 1.0;
   float UIStep = 0.05;
> = 1.0;

float Ks
<
   string UIName =  "Specular Brightness";
   string UIWidget = "Slider";
   float UIMin = 0.0;
   float UIMax = 1.0;
   float UIStep = 0.05;
> = 0.6;

// Surface texture //////

#ifdef TEXTURED_SURFACE
texture ColorTexture : Diffuse
<
   string ResourceName = "default_color.dds";
   string ResourceType = "2D";
   string UIName =  "Surface Texture (if used)";
>;

sampler2D ColorSampler = sampler_state
{
   Texture = <ColorTexture>;
   MinFilter = LinearMipMapLinear;
   MagFilter = Linear;
};
#endif /* TEXTURED_SURFACE */

//////////////////////////////////////////////////////////////////////
/////////////
// "CONNECTOR" STRUCTS -- how vertex buffer, vertex and fragment
shaders combine //
//////////////////////////////////////////////////////////////////////
/////////////

//
// This is the data fed to the effect's vertex shaders from the vertex
buffer.
//
struct appdata
{
   float3 Position       : POSITION;
   float4 UV             : TEXCOORD0;
   float4 Normal  : NORMAL0;
};

//
// This is the data that the vertex program will deliver to the
rasterizer and pixel shader,
//        for subsequent per-fragment shading.
//
struct vertexOutput
{
   float4 HPosition             : POSITION;
   float4 TexCoord                      : TEXCOORD0;
   float3 LightVec                      : TEXCOORD1;
   float3 WorldNormal           : TEXCOORD2; // why world space? see
"Why World?" above
   float3 WorldView             : TEXCOORD5;
};
```

```
/////////////////////////////////////////////////////////////////////
// SHADERS - VERTEX PROGRAMS ////////////////////////////////////////
/////////////////////////////////////////////////////////////////////

//
// Vertex setup for fragment-shaded point lighting
//
vertexOutput pointlightVS(appdata IN,
                                                uniform float4
LightPos)
{
    vertexOutput OUT = (vertexOutput)0;
    float4 normal = normalize(IN.Normal);
    OUT.WorldNormal = mul(WorldITXf, normal).xyz;
    float4 Po = float4(IN.Position.xyz,1);
    float3 Pw = mul(WorldXf, Po).xyz;
    OUT.LightVec = normalize(LightPos.xyz - Pw);
    OUT.TexCoord = IN.UV;
    OUT.WorldView = normalize(float3(ViewIXf[0].w, ViewIXf[1].w,
ViewIXf[2].w) - Pw);
    OUT.HPosition = mul(WVPXf, Po);
    return OUT;
}

/////////////////////////////////////////////////////////////////////
// SHADERS - FRAGMENT (PIXEL) PROGRAMS //////////////////////////////
/////////////////////////////////////////////////////////////////////

//
// unified fragment shading
//
float4 plasticPS(vertexOutput IN,
                          uniform float3 LightColor) : COLOR
{
   // surface info values
   float3 diffColor = SurfColor;
#ifdef TEXTURED_SURFACE
   diffColor *= tex2D(ColorSampler,IN.TexCoord.xy).xyz;
#endif /* TEXTURED_SURFACE */
   float3 Nn = normalize(IN.WorldNormal);
   float3 Vn = normalize(IN.WorldView);
   float3 Ln = normalize(IN.LightVec);   // from the point source
   float3 Hn = normalize(Vn + Ln);
   float4 litV = lit(dot(Ln,Nn),dot(Hn,Nn),SpecExpon); // built-in
lighting function
   float3 diffContrib = (litV.y * Kd) * LightColor * diffColor;
   float3 specContrib = (Ks * litV.z) * LightColor;
   // Now join spec and diffuse together
   float3 result = (diffContrib + specContrib);
   // finally, we add-in ambient light
   result += (AmbiColor * diffColor);
   return float4(result,1);
}

//////////////////////////////////////////////////////////////////////
///////////////
// TECHNIQUES -- Putting the shaders together into a complete effect
////////////////
//////////////////////////////////////////////////////////////////////
///////////////

//
// Fragment shading combined into a single pass
//
technique main
{
   pass pass0
   {
          VertexProgram = compile arbvp1
pointlightVS(PointLight0Pos);
```

```
            DepthTestEnable = true;
            DepthMask = true;
            CullFaceEnable = false;
            BlendEnable = false;
            FragmentProgram = compile arbfp1 plasticPS(PointLight0Col);
    }
}
```

Each section is clearly marked and pretty simple within itself. In Code Editor each line is numbered for your convenience during editing.

The image we have onscreen is textured, green plastic, not blue plastic. How can we make it blue?

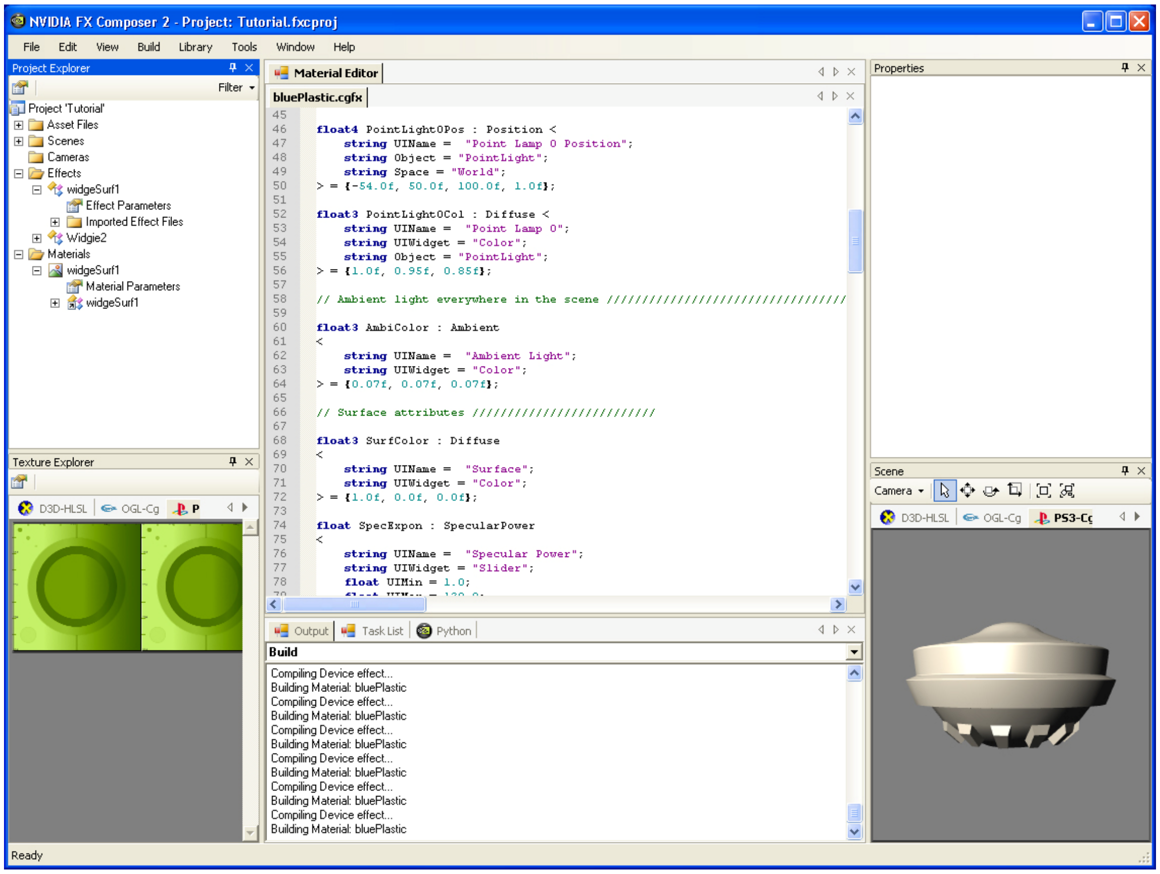
One way would be to simple choose a different texture; the texture currently loaded (and visible in the Texture Explorer panel) is green. In fact, what we really have is not green plastic but a *green texture* overlaid on *white* plastic. If the effect is loaded in the Properties panel, you'll see a white color chip. Changing this color will alter the surface.

Let's alter this shader to eliminate the texture entirely and set the default color to blue.

First, the elimination of the texture is easy—just comment-out or delete the line **`"#define TEXTURED_SURFACE"`** right at the top of the shader code (line 24).

Then, scroll down to the declaration of the surface color (line 72) and change the default color, originally {1,1,1} to pure blue {0,0,1}.

After making these two edits, from the Materials Editor menu bar select "Build→Compile bluePlastic.cgfx" to view the changes:



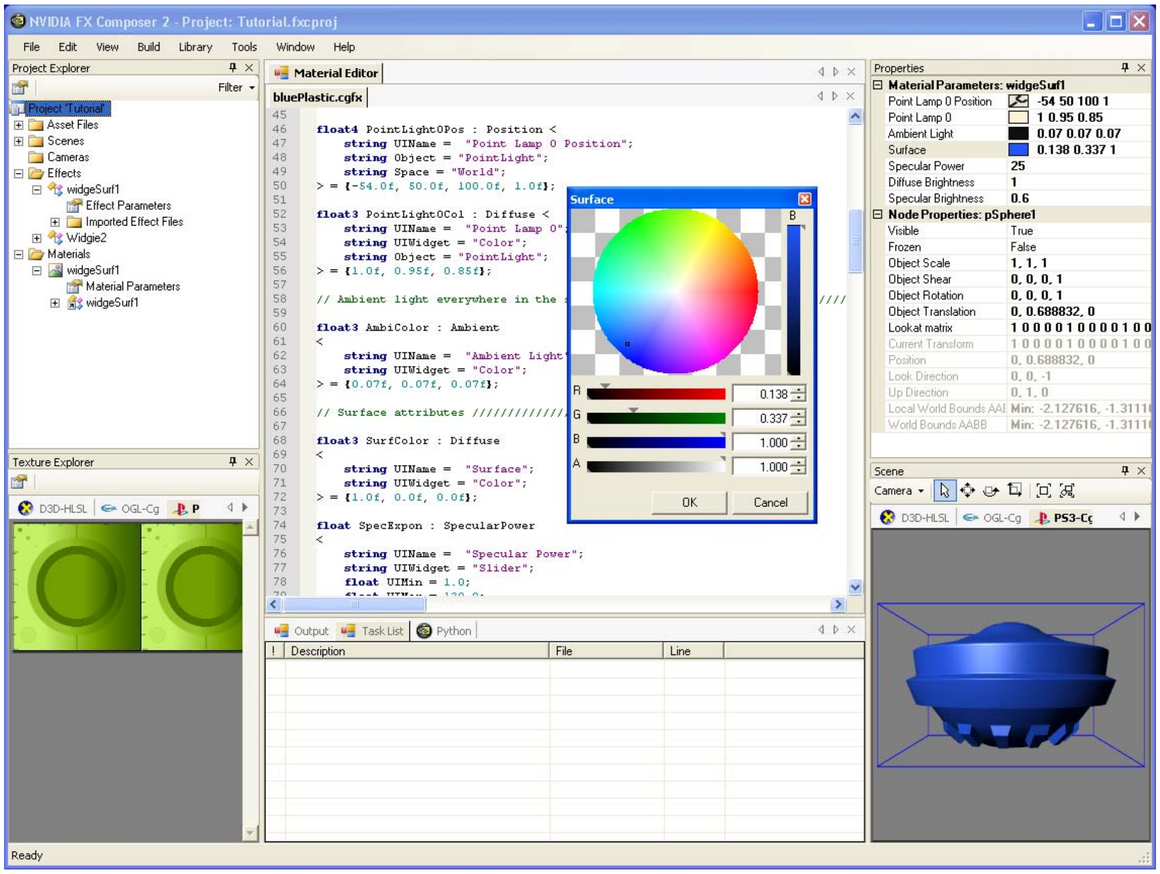
```

float4 PointLight0Pos : Position <
    string UIName =  "Point Lamp 0 Position";
    string Object = "PointLight";
    string Space = "World";
> = {-54.0f, 50.0f, 100.0f, 1.0f};

float3 PointLight0Col : Diffuse <
    string UIName =  "Point Lamp 0";
    string UIWidget = "Color";
    string Object = "PointLight";
> = {1.0f, 0.95f, 0.85f};

// Ambient light everywhere in the scene //////////////////////////////////

float3 AmbiColor : Ambient
<
    string UIName =  "Ambient Light";
    string UIWidget = "Color";
> = {0.07f, 0.07f, 0.07f};

// Surface attributes /////////////////////////

float3 SurfColor : Diffuse
<
    string UIName =  "Surface";
    string UIWidget = "Color";
> = {1.0f, 0.0f, 0.0f};

float SpecExpon : SpecularPower
<
    string UIName =  "Specular Power";
    string UIWidget = "Slider";
    float UIMin = 1.0;
```

```
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
```

Figure 46. Still Not Blue

The texture is gone, but: why isn't the model blue?

The reason is that we changed the *default value* for the variable "SurfColor," but the shader is already loaded as part of a material ("widSurf1" in the Materials folder). The value for the current material is still white. The next time we load the shader from scratch, "SurfColor" will initialize to blue. But because we altered the default after loading the effect, we need to alter the color in the properties panel if we want to look at it in the current scene. Open the material in Project Explorer and select Material Parameters, or just click on the model and the material properties will appear in the Properties pane, ready for editing.



```
45
46    float4 PointLight0Pos : Position <
47        string UIName =  "Point Lamp 0 Position";
48        string Object = "PointLight";
49        string Space = "World";
50    > = {-54.0f, 50.0f, 100.0f, 1.0f};
51
52    float3 PointLight0Col : Diffuse <
53        string UIName =  "Point Lamp 0"
54        string UIWidget = "Color";
55        string Object = "PointLight";
56    > = {1.0f, 0.95f, 0.85f};
57
58    // Ambient light everywhere in the                                    ////
59
60    float3 AmbiColor : Ambient
61    <
62        string UIName =  "Ambient Light
63        string UIWidget = "Color";
64    > = {0.07f, 0.07f, 0.07f};
65
66    // Surface attributes //////////////
67
68    float3 SurfColor : Diffuse
69    <
70        string UIName =  "Surface";
71        string UIWidget = "Color";
72    > = {1.0f, 0.0f, 0.0f};
73
74    float SpecExpon : SpecularPower
75    <
76        string UIName =  "Specular Power";
77        string UIWidget = "Slider";
78        float UIMin = 1.0;
```

R 0.138
G 0.337
B 1.000
A 1.000

Properties

Material Parameters: widgeSurf1	
Point Lamp 0 Position	-54 50 100 1
Point Lamp 0	1 0.95 0.85
Ambient Light	0.07 0.07 0.07
Surface	0.138 0.337 1
Specular Power	25
Diffuse Brightness	1
Specular Brightness	0.6
Node Properties: pSphere1	
Visible	True
Frozen	False
Object Scale	1, 1, 1
Object Shear	0, 0, 0, 1
Object Rotation	0, 0, 0, 1
Object Translation	0, 0.688832, 0
Lookat matrix	1 0 0 0 0 1 0 0 0 0 1 0 0
Current Transform	1 0 0 0 0 1 0 0 0 0 1 0 0
Position	0, 0.688832, 0
Look Direction	0, 0, -1
Up Direction	0, 1, 0
Local World Bounds AAE	Min: -2.127616, -1.3111
World Bounds AABB	Min: -2.127616, -1.3111

Figure 47. After Making the Material Blue

Here we go with the material edited to blue.

Try tweaking all the parameters to get a feeling for how they operate. Add a point light source from the "Primitives" menu, and then explore attaching it to the "Point Lamp 0 Position" property.

Altering the appearance more can just continue iterating on the same ideas. Try adding an extra color for the specular highlight: scroll down in Code Editor to line 68 or so, and copy the lines that define "SurfColor." Then paste a copy of the lines immediately below SurfColor's declaration. Rename the new variable "SpecColor" (both the name and the "UIName") and change the default value to something different, say {1,0,0}, which is red. Hit "Build File."

Your new property will now appear in the Properties panel, but the image is unchanged—we haven't altered the shading, just the parameter list.

Note: Shader authors: Avoid leaving in parameters that are "dead;" they drive artists crazy! Make sure you remove these parameters before passing shaders on to other users.

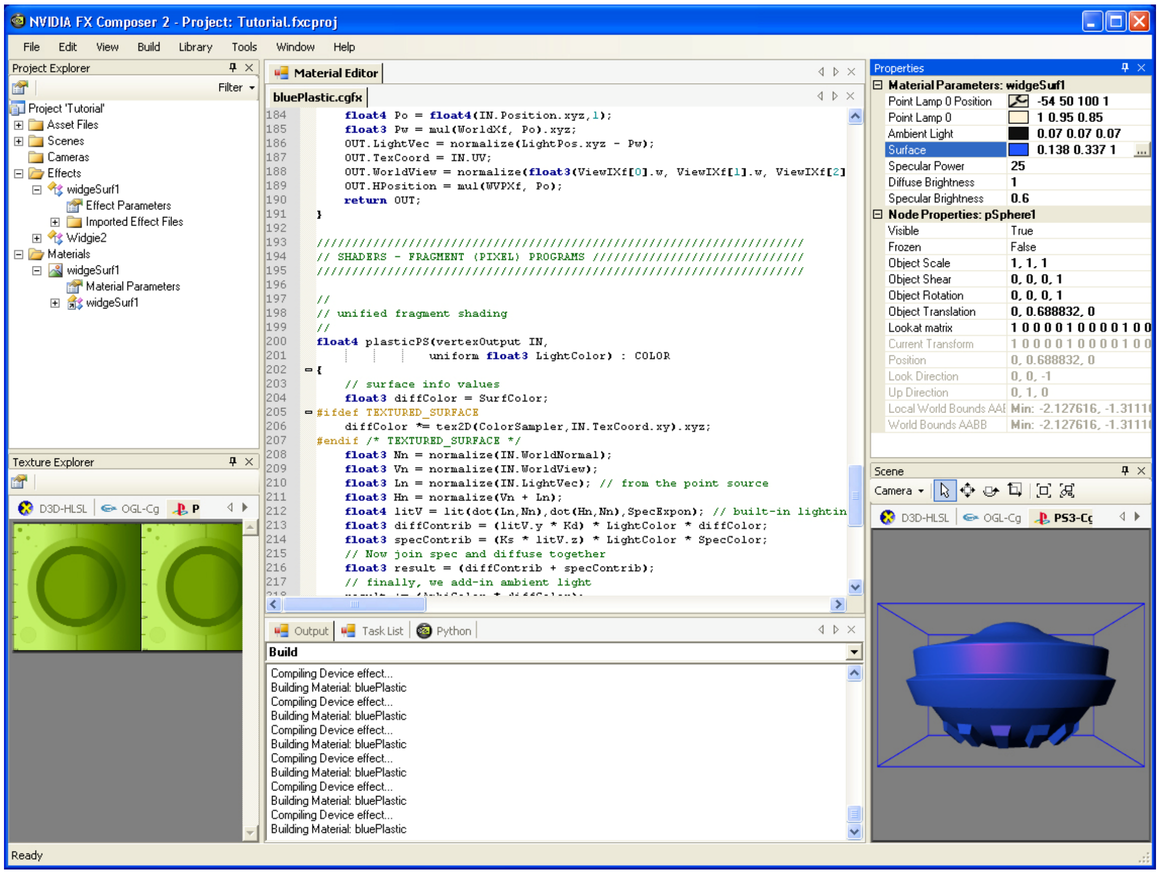
To add this color into our shading, let's track down the actual line that defines the specular highlight (somewhere around line 214 in the edited file):

```
float3 specContrib = (Ks * litV.z) * LightColor;
```

Let's change this to use our new SpecColor:

```
float3 specContrib = (Ks * litV.z) * LightColor * SpecColor;
```

Click again on "**Build→Compile bluePlastic.cgfx**"…



```
184     float4 Po = float4(IN.Position.xyz,1);
185     float3 Pw = mul(WorldXf, Po).xyz;
186     OUT.LightVec = normalize(LightPos.xyz - Pw);
187     OUT.TexCoord = IN.UV;
188     OUT.WorldView = normalize(float3(ViewIXf[0].w, ViewIXf[1].w, ViewIXf[2]
189     OUT.HPosition = mul(WVPXf, Po);
190     return OUT;
191 }
192 
193 ////////////////////////////////////////////////////////////////////
194 // SHADERS - FRAGMENT (PIXEL) PROGRAMS ////////////////////////////
195 ////////////////////////////////////////////////////////////////////
196 
197 //
198 // unified fragment shading
199 //
200 float4 plasticPS(vertexOutput IN,
201                 uniform float3 LightColor) : COLOR
202 {
203     // surface info values
204     float3 diffColor = SurfColor;
205 #ifdef TEXTURED_SURFACE
206     diffColor *= tex2D(ColorSampler,IN.TexCoord.xy).xyz;
207 #endif /* TEXTURED_SURFACE */
208     float3 Nn = normalize(IN.WorldNormal);
209     float3 Vn = normalize(IN.WorldView);
210     float3 Ln = normalize(IN.LightVec); // from the point source
211     float3 Hn = normalize(Vn + Ln);
212     float4 litV = lit(dot(Ln,Nn),dot(Hn,Nn),SpecExpon); // built-in lightin
213     float3 diffContrib = (litV.y * Kd) * LightColor * diffColor;
214     float3 specContrib = (Ks * litV.z) * LightColor * SpecColor;
215     // Now join spec and diffuse together
216     float3 result = (diffContrib + specContrib);
217     // finally, we add-in ambient light
```

Properties

Material Parameters: widgeSurf1	
Point Lamp 0 Position	-54 50 100 1
Point Lamp 0	1 0.95 0.85
Ambient Light	0.07 0.07 0.07
Surface	0.138 0.337 1
Specular Power	25
Diffuse Brightness	1
Specular Brightness	0.6
Node Properties: pSphere1	
Visible	True
Frozen	False
Object Scale	1, 1, 1
Object Shear	0, 0, 0, 1
Object Rotation	0, 0, 0, 1
Object Translation	0, 0.688832, 0
Lookat matrix	1 0 0 0 0 1 0 0 0 0 1 0 0
Current Transform	1 0 0 0 0 1 0 0 0 0 1 0 0
Position	0, 0.688832, 0
Look Direction	0, 0, -1
Up Direction	0, 1, 0
Local World Bounds AAE	Min: -2.127616, -1.3111
World Bounds AABB	Min: -2.127616, -1.3111

```
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
Compiling Device effect...
Building Material: bluePlastic
```

Figure 48. Adding a Colored Highlight

…and now we have a colored highlight.

Feel free to experiment with many different approaches, and don't be bashful about mashing together code from lots of different shader effect to get the look that's best for your project.

Finally, we'll save an updated version of the model and material, either through "Save All" or just exit FX Composer and answer "Yes" to the dialog box.

Appendix

Glossary of Terms

active document.

The document currently open. Although you can have a number of documents can exist for a project, only one can be active at any time. The active document is indicated by a filled dot in the scene graph.

assets

Sub-notes that makes up a document—such as scenes, material, lights, images, and geometries.

Asset Library

Section that provides two ways of creating asset libraries: 1) Create several new assets (effects, materials, images, lights, cameras, or geometry) in a new empty document, or 2) load a collection of COLLADA documents (each with its own assets) and then create a new empty document and drag-and-drop your chosen assets into the new document.

Cameras

The section of Assets Panel that groups all the cameras (orthographic cameras or perspective cameras) in a project.

Code Editor

The section that lets you view and modify shader source code. Code Editor features all the standard editing functionality.

Color Picker

Supports high-dynamic range and allows you access to the full range of floating point values. Color Picker shows individual color channels, as well as an overall exponent (represented by the vertical slider on the right of the Color Picker).

Deformers

The section of the Assets Panel that lists all skinning data that is loaded.

document

A container for textures, shaders, cameras, lights, geometry, and so on. A document's assets are listed as sub-nodes under each document.

effects

A generic shader—for example, marble.

Geometries

This section of the Assets Panel that groups all 3D models in a project

Texture Viewer

A viewer that helps you navigate through the images in your project. By default, Texture Viewer arranges thumbnails of all your images neatly in rows. Each thumbnail is accompanied by the image's file name and resolution.

Images

The section of Assets Panel that groups all the images in a project in certain formats (dds, .jpg, .png, .tga, .bmp, .exr, and .hdr). or texture types (cube map, 1D, 2D, 3D) and pixel formats (RGBA, RGB, DXT, 32-bit float, 16-bit float).

Assets Panel.

displays all the "assets" in a project, grouped by asset type. such as geometries, images, lights, materials, and scenes.

Lights

The section of Assets Panel that groups all the lights (spot lights, directional lights, and point lights) in a project.

material

An instance of an effect with specific properties settings—for example, green marble. Materials are what you actually apply to objects in a scene.

Output Panel

Displays various messages from FX Composer related to builds.

physical document

A document that exists on disk (for example, as a COLLADA .dae file). Compare to "virtual document."

Project Explorer

A file-based representation of your project in the form of an expandable scene graph. The main nodes of the scene graph show the various documents in your project.

Properties Panel

Allow you to view and change object properties—primarily material properties. For example, you can use Color Picker to adjust a Phong materials' diffuse color, specular color, or specular exponent, or you can type in color values.

Python Panel

Panel that gives you a console from which you can access FX Composer's powerful scripting features.

scene

A composition of geometric objects, lights, cameras, and so on. You can use different scenes for a variety of purposes—common examples are different levels or different test scenarios.

Render Panel

Panel that displays the current scene and has several controls for navigating and manipulating scenes. A tab for each device is installed on your system; typically, you'll see tabs for Direct3D and OpenGL devices. You can dock or undock these to view multiple devices simultaneously or to enlarge a particular rendering.

ShaderPerf

Tool that helps you analyze shader performance, complete with informative graphs and tables.

Texture Explorer

Displays all the textures used in the current scene, including procedurally generated textures and render targets. Texture Explorer panel also enables visualization of cube maps and normal maps.

Toolbar

Helps you work with cameras and objects in a scene and contains many buttons for this purpose, such as Cameras, Navigation Mode, and Object Selection.

virtual document

Documents not on disk that are useful for organizing and grouping assets into a logical collection.. Once you are satisfied with a virtual document's contents, you can save it to disk; at this point, it becomes a physical document. Compare to "physical document."

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com