



**Whitepaper**

# Using SAS with CgFX and FX file formats

Kevin Bjorke





# Using SAS with CgFX and FX file formats

---

## What, Why, How

This document covers the use of SAS (that is, “Standard Annotations and Semantics,” and usually pronounced “sass”) in effect files for either of the two popular text-based “\*.fx” formats – either .fx (for DirectX only, and based on HLSL) or .cgfx (for use with either DirectX or OpenGL, and based on Cg). Both formats are extremely similar, and the SAS details are identical between the two<sup>1</sup>.

To date, *NVIDIA FX Composer* is the shading tool with the most comprehensive support of SAS, and will be used as a baseline example. This document will also cover annotations and semantics that are unique to *FX Composer*.

*This is not a general shading tutorial. It assumes that you can be comfortable with C++ -like languages, and in general doesn't try to teach you about color and light, nor HLSL or Cg except in the context of SAS. For a general introduction to hardware shading languages, try “The Cg Tutorial,” which is available both at booksellers and on the NVIDIA developer web site, <http://developer.nvidia.com>*

Effect files (that is, shader files with the suffixes “.cgfx” or “.fx”) try to describe a complete setup for real-time rendering. That means they include not only the code for the specific vertex and pixel shader functions that will run on the GPU, but also definitions of the contexts for these shader functions: the required graphics states; the user parameters that control them (whose values may be controlled by a UI, or automatically provided by the rendering program); and in many cases, the multiple rendering passes that together define a complete, final image. In some advanced cases, as we'll describe below, these multiple passes may be executing in varying order – and in some cases there can be some limited interactions between multiple effect files.

Control over all of these interactions is provided by SAS.

---

<sup>1</sup> SAS is sometimes also called “DXSAS” to indicate its DirectX origins, but the “DX” is usually dropped since SAS is effective across graphics APIs.

## Semantics and Annotations

Just so that we are clear on what semantics and annotations are, consider this typical parameter declaration, drawn from the NVIDIA shader library's "plasticP.fx" effect<sup>2</sup>:

```
float3 gLamp0Pos : Position <
    string Object = "PointLight0";
    string UIName = "Lamp 0 Position";
    string Space = "World";
> = { -0.5f, 2.0f, 1.25f } ;
```

In this example, global variable `gLamp0Pos` has a **semantic**, three **annotations**, and a **value**. *All three are optional*. In general, parameter declarations are of the form:

```
type name : semantic <annotation_list> = value ;
```

which can cover one or many lines, up to the closing semicolon.

A bare minimum declaration might be something like:

```
float gCurrentScore;
```

Where the host program (e.g., *FX Composer*) will insert defaults for the **semantic** (none), **annotation\_list** (none) and **value** (in this case, probably a zero).

## Why Semantics *and* Annotations?

During the initial development of Cg and HLSL, semantics were used to describe specific GPU hardware registers. For example, this structure declaration describes the data that an effect expects in the vertex buffer before executing a vertex shader function:

```
struct appdata {
    float3 Position      : POSITION;
    float4 UV            : TEXCOORD0;
    float4 Normal        : NORMAL;
    float4 Tangent       : TANGENT0;
    float4 Binormal      : BINORMAL0;
};
```

---

<sup>2</sup> For OpenGL users, there is also a corresponding "plasticP.cgfx" in the library, which is the same in terms of semantics and annotations. In general we will use HLSL examples in this document, but the SAS usage applies identically to both languages, save for the use of the HLSL virtual machine as described in the Appendix on "texture shaders."

Each of the semantics in this structure declaration indicate a specific part of the hardware memory that's fed to vertex shaders that use input values of type "appdata." Here's the vertex shader from that same file (along with the declarations of its output data).

```
struct vertexOutput {
    float4 HPosition          : POSITION;
    float2 UV                 : TEXCOORD0;
    // The following values are passed in "World"
    // coordinates since it tends to be the most flexible
    // and easy for handling reflections, sky lighting,
    // and other "global" effects.
    float3 LightVec           : TEXCOORD1;
    float3 WorldNormal        : TEXCOORD2;
    float3 WorldTangent       : TEXCOORD3;
    float3 WorldBinormal      : TEXCOORD4;
    float3 WorldView          : TEXCOORD5;
};

vertexOutput std_VS(appdata IN,
    uniform float4x4 WorldITXf,
    uniform float4x4 WorldXf,
    uniform float4x4 ViewIXf,
    uniform float4x4 WvpXf,
    uniform float3 LampPos) {
    vertexOutput OUT = (vertexOutput)0;
    OUT.WorldNormal = mul(IN.Normal,WorldITXf).xyz;
    OUT.WorldTangent = mul(IN.Tangent,WorldITXf).xyz;
    OUT.WorldBinormal = mul(IN.Binormal,WorldITXf).xyz;
    float4 Po = float4(IN.Position.xyz,1);
    float4 Pw = mul(Po,WorldXf);
    OUT.LightVec = (LampPos - Pw.xyz);
#ifdef FLIP_TEXTURE_Y
    OUT.UV = float2(IN.UV.x,(1.0-IN.UV.y));
#else /* !FLIP_TEXTURE_Y */
    OUT.UV = IN.UV.xy;
#endif /* !FLIP_TEXTURE_Y */
    OUT.WorldView = normalize(ViewIXf[3].xyz - Pw.xyz);
    OUT.HPosition = mul(Po,WvpXf);
    return OUT;
}
```

The semantics specify hardware registers, and their usage is also defined by context. You may have noticed that both the input structure *AND* the output structure contain members with the **POSITION** semantic – the context (in this case, being compiled as a vertex shader) makes the difference clear to the compiler and host application. If you tried to compile this function as a pixel shader, it would fail.

Here is a pixel shader from the same effect file:

```

float4 plasticPS(vertexOutput IN,
                 uniform float3 SurfaceColor,
                 uniform float Kd,
                 uniform float Ks,
                 uniform float SpecExpon,
                 uniform float3 LampColor,
                 uniform float3 AmbiColor
) : COLOR {
    float3 diffContrib;
    float3 specContrib;
    plastic_shared(IN,Kd,Ks,SpecExpon,
                  LampColor,AmbiColor,
                  diffContrib,specContrib);
    float3 result = specContrib +
                  (SurfaceColor * diffContrib);
    return float4(result,1);
}

```

Here *the function itself* has been assigned a hardware semantic –for a pixel shader, that **COLOR** semantic (or the synonym **COLOR0**) means the final output color of the pixel as the image is delivered to the final render target (either to a render surface or the frame buffer).

There are a variety of syntactical ways to do this – some users may prefer to use “out” parameters rather than directly returning a value, or they may not like to use structs, or they may prefer global-scope variables as opposed to passing formal parameters. Here’s a version using an “out” parameter instead:

```

void plasticPS(vertexOutput IN,
               uniform float3 SurfaceColor,
               uniform float Kd,
               uniform float Ks,
               uniform float SpecExpon,
               uniform float3 LampColor,
               uniform float3 AmbiColor,
               out float4 FinalColor : COLOR0
) {
    float3 diffContrib;
    float3 specContrib;
    plastic_shared(IN,Kd,Ks,SpecExpon,
                  LampColor,AmbiColor,
                  diffContrib,specContrib);
    float3 result = specContrib +
                  (SurfaceColor * diffContrib);
    FinalColor = float4(result,1);
}

```

See the Appendix “Alternative Constructions” to see some possibilities. They are all, however, functionally identical.

The semantic names describing standard hardware, such as **TEXCOORD0**, as understood to have specific mappings to specific hardware registers, as defined

by the API (DirectX or OpenGL). In 99.9% of the cases, these are the names you should use. For some very old game or CAD engines, you may need to specify *exactly* the hardware register you need. In this rare case, you can use the **register** semantic. The register semantic look like a function: e.g., **register(c1)** means “color register 1.” This semantic’s use is fading into oblivion.

## Other Hardware Semantic Names

The sample shown covers the most common hardware semantics. There are a couple of other rarely-used ones, such as the polygon-facing flag (**VFACE** in DirectX, **FACE** in OpenGL) and the **PSIZE** attribute of particles.

---

## Semantics from software API's

Semantics for hardware registers have connections defined by the overall graphics state. What about values that should be connected to the surrounding software environment? For example, how can we bind the location of the current object in the world to a value, or the color of the current light source?

DirectX and the CgFX runtime both accept several semantics relating to just these issues. They can help provide hints to the renderer about connecting colors and positions to existing parts of the scene, and they can tell the renderer to load data from the scene into specific parameters.

All of these semantics will be assigned to *\*fx parameters* – never to functions. When a parameter is bound to a scene element (either automatically or by the user), its values will then be automatically assigned on each frame – so the user will not be able to tweak those values (say, in the FX Composer properties pane) except by altering the underlying scene. Once bound, they are “untweakable.”

Some hints are stronger than others, depending on the renderer. In general, the semantics indicating transform matrices appear to bind automatically in all renderers. Semantics for light positions, surface colors, and so forth: unpredictable! Though well supported in *FX Composer*. Happily, only a few very old renderers appear to be actually upset if you provide a semantic that they don't support.

Typical semantics used on parameters will correspond to the values of the old DirectX “hardware transform and lighting” days – for example the **DIFFUSE** semantic is intended to correspond to the “Diffuse” value in a DirectX material structure and assigned in a DirectX9 program by **IDirect3DDevice9::SetMaterial()**. In general, these semantics relating to old-style, pre-pixel-shader-days material and light definitions (others include **SPECULAR**, or **AMBIENT** – if you look at the member names of the D3D Material structs they will look surprisingly familiar...) are being used less and less, as they are references to an antiquated material definition scheme (if you do want to

bind to such values, there is also a newer way, using annotations to bind to an object or light – in such cases, use the semantic **COLOR** and see the section “Light and Object Binding” – there are additional semantics associated with lights and cameras that will be described in that section).

Vectors indicating the position or direction of a lamp can be distinguished by the **POSITION** or **DIRECTION** semantics, which can also let the renderer know what sorts of lights can be successfully bound to which parameters. (Later on, see info on the special *FX Composer* “dirpos” use of the **POSITION** semantic for float4 vectors, which can dynamically flip between both point and directional lamp types).

## Transforms

Transforms never go out of style. Both fx and cgfx formats follow the DirectX definition of a matrix stack:

Object → World → View → Projection

Points in the vertex buffer are in “Object” space, which can be transformed to a shared “World” coordinate system. Transforming from “World” to “View” space sets the points to be located relative to the camera’s location and orientation, and the final matrix transforms them into projected screen space.

Those last three transforms can be used as semantics, and can also be (in the order shown) concatenated: That is, you can have semantics **WORLD**, **WORLDVIEW**, **VIEWPROJECTION**, or **WORLDVIEWPROJECTION**.

(Why no “OBJECT” semantic? Because it will always be an identity matrix!)

In addition, the optional suffixes INVERSE and TRANSPOSE can be added to the mix: so the ultimate automatic transform name could be the intimidating **WORLDVIEWPROJECTIONINVERSETRANSPOSE**.

Generally, a small number of transforms will get you what you need for any specific effect. Here are the global transform variables declared by the example “plasticP.fx” file:

```
float4x4 gWorldITXf : WorldInverseTranspose;
float4x4 gWvpXf : WorldViewProjection;
float4x4 gWorldXf : World;
float4x4 gViewIXf : ViewInverse;
```

In this case, we are defining the “View” as the primary scene camera. What about spotlight shadow cameras? These are specified by the “Object” annotation. This specialized usage will be covered in the section on shadowing.

## Viewport Semantics

The **VIEWPORTPIXELSIZE** semantic should be attached to a float2 parameter and provides pixel-count information about the size of the current scene render window. This is especially important for effects that use Render-To-Texture as part of a full-screen effect.

The “Quad.fxh” file used by many of the Shader Library FX samples defines the following variable, used widely by full-screen library shaders:

```
float2 QuadScreenSize : VIEWPORTPIXELSIZE <
    string UIName="Screen Size";
    string UIWidget="None";
>;
```

Individual pixels/texels will be of the size  $\text{float2}(1.0,1.0)/\text{QuadScreenSize}$  – and importantly (a fact already factored into the vertex shaders defined in the same header file), this calculation can be used to adjust for the half-texel origin offset in DirectX.

## Texture Semantics

Textures have a small number of semantics. The most important ones are those that designate a texture as a writable render target: **RENDERCOLORTARGET** or **RENDERDEPTHSTENCILTARGET**. Occasionally you will find textures marked as **DIFFUSE**, **DIFFUSEMAP**, **NORMAL**, **SPECULAR**, **SPECULARMAP**, **ENVMAP**, **ENVIRONMENTNORMAL**, or **ENVIRONMENT**. These provide hints to the API about automatic binding, but these usages are disappearing over time (again, they are mostly useful in mapping fixed-function values to shader values, and are now antiquated). See the appendix on Texture Annotations for more details.

## FX Composer-Specific Semantics

### Combining Directions and Positions

Some game and rendering engines like to use the nature of homogenous data to let the same shader accept light data from either point lights or directional lights using the same vector. That is, for a float4 vector, the *w* component indicates if the data is a point in space (nonzero *w*) or a vector through space (zero *w*). *FX Composer* supports this, and float4 vectors can be tagged with the semantic **POSITION**. *FX Composer* will know how to bind both point and directional lights to such values, and will set the *w* component appropriately<sup>3</sup>.

---

<sup>3</sup> Float3 data, which lacks a *w* component, can't be used this way. Float3 positions will only bind to regular position data in *FX Composer*.

As a momentary shading aside, here is a typical vertex shader for such data:

```
vertexOutput std_dp_VS(appdata IN,
    uniform float4x4 WorldITXf,
    uniform float4x4 WorldXf,
    uniform float4x4 ViewIXf,
    uniform float4x4 WvpXf,
    uniform float4 LampDirPos // "dirpos" POSITION
) {
    vertexOutput OUT = (vertexOutput)0;
    OUT.WorldNormal = mul(IN.Normal,WorldITXf).xyz;
    OUT.WorldTangent = mul(IN.Tangent,WorldITXf).xyz;
    OUT.WorldBinormal = mul(IN.Binormal,WorldITXf).xyz;
    float4 Po = float4(IN.Position.xyz,1);
    float4 Pw = mul(Po,WorldXf);
    if (LampDirPos.w == 0) { // "dirpos" switch HERE
        OUT.LightVec = -normalize(LampDirPos.xyz);
    } else {
        // we are still passing a (non-normalized) vector
        OUT.LightVec = LampDirPos.xyz - Pw.xyz;
    }
#ifdef FLIP_TEXTURE_Y
    OUT.UV = float2(IN.UV.x,(1.0-IN.UV.y));
#else /* !FLIP_TEXTURE_Y */
    OUT.UV = IN.UV.xy;
#endif /* !FLIP_TEXTURE_Y */
    OUT.WorldView = normalize(ViewIXf[3].xyz - Pw.xyz);
    OUT.HPosition = mul(Po,WvpXf);
    return OUT;
}
```

Which correctly preserves the direction and magnitude for light calculations in the pixel shader, regardless of lightsource type.

## Timing

The **TIME** semantic provides *absolute system time*. In seconds. As this makes for a rather large unreadable number, such parameters are usually marked as hidden using a **UIWidget** annotation.

Absolute system time is best used for “self-animating” effects like fire, spinning fighter propellers, water, etc. Several NVIDIA Shader Library effects use it, such as “MrWiggle,” “Ocean,” or “post\_corona.”

To grab information from animated clips, consider FX Composer’s “playblast” Python module, or used the **ANIMATIONTIME** and **ANIMATIONTICK** semantics.

**TIME** is also often important when dealing with interactivity via the mouse.

## Mouse Position and State

*FX Composer* permits querying the position of the mouse, the states of the mousebutton, and the time when those buttons were last pressed.

**MOUSEPOSITION** parameters should be float2. They will report the current location relative to the render window.

**LEFTMOUSEDOWN** & **RIGHTMOUSEDOWN** provide four bits of info as a float4: if the button was pressed; where the mouse was when it was pressed as an XY pair just like **MOUSEPOSITION**; and *when* the button was pressed, in the same format as used by **TIME** parameters. For either button semantic, the format will be:

**XY**: Where the mouse was when pressed.

**Z**: Is the button pressed? One or zero.

**W**: The system time when the button was last pressed.

## Renderer Reset

When the RenderWindow is reset, either by being resized, a new scene has started, etc., then “bool” parameters with the **FXCOMPOSER\_RESETPULSE** semantic will be set to true (1) – *for exactly one frame*. At the end of that first new frame they'll flip back to false (0). This will be important later on when we discuss full-screen effects.

---

## Annotations

Annotations appeared because rendering applications needed a method to specify the UI for specific effect parameters. While an application could guess, either by name or by reference to a semantic, the results are unpredictable. It seems obvious that a float3 parameter with the **DIFFUSE** semantic is a color, but what about a float3 called, say, “Mixer”? Is it a location, a color, something else?

Annotations eliminate the guessing by providing a mechanism to specify UI details explicitly. Any type of metadata can be attached to any parameter, for whatever use you desire. If an application doesn't recognize the specific annotations, it can just ignore them (for *FX Composer* users, unrecognized attributes can still be queried from Python, if you have any custom scheme).

Here is another example from our sample effect:

```
float gKd <
    string UIWidget = "slider";
    float UIMin = 0.0;
    float UIMax = 1.0;
    float UIStep = 0.01;
    string UIName = "Diffuse Strength";
> = 0.9;
```

As you can see annotations require a type declaration as well as their own name. In this example, the annotations are purely related to the UI for the parameter “gKd” – they say that it should be displayed as a fine-grained slider with a range from zero to one, and that its *displayed* name should not be the cryptic-looking (to a non-shader-programmer) “gKd,” but the more human-readable “Diffuse Strength.”

Annotations can also be added to technique and pass definitions, which will be important later.

## Light and Object Binding

Often scales, colors, and transforms need to be bound to a specific lamp or other object in the current scene. Annotations and semantics, working together, provide a flexible mechanism for accomplishing this. A semantic can be used to specify the *kind* of data required – say, a **COLOR** – while an annotation (usually **Object**) is used to describe *where* to find that kind of information.

Depending on the kind of object, a semantic may make sense or not. In general, if the Object named includes the substring “light,” then the data should be appropriate for a light. If instead it includes the substring “camera,” then the data should be appropriate for a camera. Otherwise, it’s just a user-defined name & there’s little guarantee that the value can be bound sensibly.

## Coordinate Systems

For bound data that is spatial, getting the data in the appropriate coordinate system is important. Currently only **POSITION** and **DIRECTION** semantics support the **Space** annotation, which can be assigned as “World,” “View,” or “Object.”

## Light Names

Typical light names might be something like “Spotlight0” – FX Composer will keep track of all such names encountered, and permit the user to bind any light to the symbolic name “Spotlight0” – with correct propagation of values to any and all parameters that use the name.

Light names can be used with either **Object** or **Frustum** annotations, as we’ll see below.

## Light Semantics

The following semantics can be used sensibly with lights. Specify the light in the **Object** annotation and (for geometric values) be sure to specify a desired **Space** for coordinates.

**POSITION**

**DIRECTION**

**COLOR**

**DIFFUSE**

**SPECULAR**

**AMBIENT**

**ATTENUATION**

**CONSTANTATTENUATION**

**LINEARATTENUATION**

**QUADRATICATTENUATION**

**FALLOFFANGLE**

**FALLOFFEXPONENT**

As an example:

```
float gCone1 : FALLOFFANGLE <
    string Object = "Spotlight1";
>;
```

The values associated with lights within *FX Composer* can be seen by selecting the light and its properties. They can be edited there, if not already bound by animation (say, from a *3ds Max* clip).

Viewing transforms associated with shadow maps are sometimes bound using the **Frustum** annotation rather than **Object**. *FX Composer* supports both usages. See the section "Shadow Transforms" for an example.

## Camera Semantics

Only camera transform information is supported.

## Scaling Info

How big is an object? The shading of a 2-inch block of wood and a 20-foot one are likely to be different. The **UNITSCALE** semantic indicates to the rendering program that the indicated value can be mapped to some external or real-world measurement. Here's an example from the Shader Library's "cage.fx":

```
float gScale : UNITSSCALE <
    string units = "inches";
    string UIWidget = "slider";
    float uimin = 0.0;
    float uimax = 20.0;
    float uistep = 0.01;
    string UIName = "Size of Pattern";
> = 5.1;
```

The host render may define its own default value of for **units** if it is not supplied by the annotation.

## Annotations without End?

While annotations let you add arbitrary (meta)data, they also invite a Babel-like explosion of arbitrary duplications of the same purposes – say, five different definitions of “slider” – to avoid this hazard, *standard* annotations and semantics were defined<sup>4</sup>.

---

## Standard Annotations and Semantics

At its core, SAS simply defines a standard list. While any rendering application can use semantics and annotations as they see fit, following the standard list gives everyone the best chance of broad compatibility.

For most programs, the key annotations are few, and related to UI. The first is the **UIName** annotation, which we saw above. This allows sometimes cryptic variable names to be displayed to a user as something human-readable and more descriptive.

Next is the **UIWidget** annotation, which we also saw in the example. The **UIWidget** string has three especially useful values: “Slider,” “Color,” and “None.”

We saw “Slider,” as well as the related **UIMin**, **UIMax**, and **UIStep** annotations that are unique to sliders. Here is a “Color”:

```
float3 gSurfaceColor : DIFFUSE <
    string UIName = "Surface";
    string UIWidget = "Color";
> = {1,1,1};
```

---

<sup>4</sup> Some damage is already done! For example, effect files for some DCC applications like XSI and 3ds Max are already using their own semantics for camera location. Fortunately simple workarounds are available.

The parameter will be displayed with the name "Surface" (which, since it's being displayed as a color, is obviously "Surface Color").

The "None" value for **UIWidget** completely suppresses display of the parameter. It's principally useful for parameters that are automatically-bound to an element from the scene, such as a transform. There's no need to clutter-up the properties pane with a lot of changing but un-tweakable numbers ("number noise"). The full matrix declarations from "plasticP.fx" actually include this semantic for just that reason:

```
float4x4 gWorldITxf : WorldInverseTranspose <
    string UIWidget="None"; >;
float4x4 gWvpXf : WorldViewProjection <
    string UIWidget="None"; >;
float4x4 gWorldXf : World <
    string UIWidget="None"; >;
float4x4 gViewIXf : ViewInverse <
    string UIWidget="None"; >;
```

Any rendering application may have its own special cases – for example, the standard Maya Cg plugin only allows float4 parameters to be displayed as colors. Some applications may have special **UIWidget** values that are not widely recognized (say, an enumerated pulldown). But the standard **UIWidget** name should still be used.

## The Two SAS's

Two different versions of SAS exist. The version 1.0 spec, which is officially supported by Microsoft, and the 0.86 pre-release, which is the version used by FX Composer and several other applications. The principal difference between the two? *Scripting*.

---

## SAS Scripting

SAS 0.86 includes support for the scripting of techniques and passes.

Why would you want to do that?

Rendering occurs in a series of discrete passes. Often, only one pass is needed to render an object (or entire scene). Sometimes, several passes are accumulated or otherwise combined. By default, the passes are simply executed in order, one after another. If that's all you need, then scripting isn't required. But it often is! For example, you may have an iterative technique that requires looping on a single pass. Or you may want to use a shadow map, which needs to be rendered to an offscreen buffer with a different render size and transform from the regular scene camera.

Scripting also lets you set the desired priority of effects that contain multiple techniques, and pass data to the rendering program to let it know when to execute your effect in relationship to other effects in the scene.

## The Script : STANDARDSGLOBAL Parameter

In “plastic.fx,” you may notice this block:

```
float Script : STANDARDSGLOBAL <
    string UIWidget = "none";
    string ScriptClass = "object";
    string ScriptOrder = "standard";
    string ScriptOutput = "color";
    string Script =
    "Technique=Technique?SimplePS:TexturedPS:SimpleQuadraticPS:
    TexturedQuadraticPS;";
    > = 0.8;
```

This special “parameter” **Script**, when semantically tapped as **STANDARDSGLOBAL**, is expected by the renderer to contain annotations to describe the overall effect.

The value **0.8** specifies the SAS version.

The **UIWidget** hides this variable (it isn’t ever actually called by the shader functions).

The **ScriptClass**, **ScriptOrder**, and **ScriptOutput** annotations indicate that this is a surface material (“object”) effect, that it follows standard use and it outputs colored pixels. So far so good and the expected default state -- in fact we could completely ignore and delete the **Script** parameter, except for the contents of the last **Script** annotation (yes, the **Script** variable has a **Script** annotation).

## Script <Script> and Technique Selection

By default, \*.fx files will by default select techniques according to the order they appear in the file. If a technique is unavailable, it will be skipped and the next one checked, working down the list until a working one is found. Here, the **Script** annotation is letting the shader author *explicitly* name which techniques should be available, and the order they should appear when used in renderers that permit on-the-fly switching of techniques.

In this case, the **Script** specifies that the “SimplePS” technique is the correct default. In programs like FX Composer or Maya that permit technique selection, the preferred order of the displayed lists in those programs will be

```
SimplePS
TexturedPS
SimpleQuadraticPS
TexturedQuadraticPS
```

If there are other techniques in the file say, a vertex-shaded alternative, or a version specific to a particular game platform), they will not, by default, be displayed or available to the user of this effect.

*(The shader library includes **Script** parameters and other SAS details in most shaders, even though they are often redundant. This is because shader library samples are intended to provide templates for further exploration. They're a bit deliberately "over-engineered" – we do the typing so you don't have to.)*

## ScriptOutput

Let's get back to those other annotations: **ScriptClass**, **ScriptOrder**, and **ScriptOutput**. The last is the easiest – **ScriptOutput** should *always* be "color" and nothing else. This is a practical assertion – while values like "depth" were intended to be supported, they've never been used.

## ScriptClass

**ScriptClass** defines the purpose of this effect. Is it a material to be applied to objects? Then it should be "Object," which is the default expectation. Some effects, however, are complex, and may require rendering to multiple render targets. They may in fact render no objects directly at all. Such effects should designate themselves as being in the "Scene" **ScriptClass**.

There is also a third choice for **ScriptClass**, "SceneOrObject." Its purpose has been to provide a more complete choice of techniques. By default, it's expected that "Object" effects don't need to write to offscreen buffers, while "Scene" effects can. "Object" effects can be bound to geometry, while "Scene" effects are typically bound only to the entire frame. But sometimes you really do need both – for example, a material for terrain may have a simple version, but also a second, more-complicated version that renders shadow maps.

## ScriptOrder

The **ScriptOrder** is meaningful for "Scene" and "SceneOrObject" effects. It has three possible values: "standard," "preprocess," and "postprocess." You should normally not assign "preprocess" or "postprocess" to "Object" effects – these two **ScriptOrder** choices are specifically intended for image processing.

The three choices allow some broad-scale control over effect ordering in the overall scene. Generally, the user may have little control over the order in which effects are executed – they just get fired-off as their objects appear in the buffer. The **ScriptOrder** lets this amorphous mass be split into three smaller amorphous masses. The preprocess phase, which will always be rendered first and can include the creation of background images and (for COLLADA) shared textures; the standard phase, which will render next (and unless over-ridden, will include all object rendering); and the closing postprocess phase, which renders

after all others to add screen-buffer effects like motion blur, color styling, and bloom.

There may be multiple preprocess and/or postprocess effects in a single scene, and their specific ordering in relation to one another within those groups is not always well-defined (generally, they will render in the same order as they were added to the scene). But they *are* guaranteed to be rendered within their group.

This is important for image-processing-based rendering. For example, a scene may include a preprocess effect to render a background, then the objects are rendered on top of that, and then a postprocess effect adds contrast control and color tinting.

## Pass Ordering and Screen Clears: Technique Scripting

As mentioned, we can attach annotations to individual techniques and individual passes within those techniques. If we just want to execute those passes in a straight-ahead, all-pixels-to-the-framebuffer fashion, then we don't need to do anything special. But if we want additional control, then the **Script** annotation is useful here, too.

We can also manipulate when and how the screen is to be cleared while executing our effect (as you can imagine, it's very dangerous to let object materials clear the screen! Other objects may disappear).

Pass *ordering* is controlled by the technique's **Script** annotation. Here's one of the longest in the library, from "scene\_uvds\_skin.fx":

```
technique Main <
    string Script =
        "Pass=MakeShadow;"
        "Pass=HBlur;"
        "Pass=VBlur;"
        "RenderColorTarget0=;"
        "RenderDepthStencilTarget=;"
        "ClearSetColor=gClearColor;"
        "ClearSetDepth=gClearDepth;"
        "Clear=Color;"
        "Clear=Depth;"
        "Pass=useBakedLighting;";
    > {
        // ...rest of technique and pass declarations...
```

As we can see, the **Script** contains a string with multiple commands (the per-line strings are automatically concatenated up until the final semicolon). Each of the commands itself has a semicolon (forgetting these is a common mistake), and each command is essentially a leftside=rightside assignment. In two cases, the rightside is empty. In others, the rightside contains the names of scene global parameters, and in others the names of passes, and in others keywords telling

the renderer what to do. It's admittedly a strange language-within-a-language. Fortunately, you've already seen 90% of it!

Each line represents a specific action to take. First, the passes named "MakeShadow," "HBlur," and "VBlur" are executed, in that order.

Then, we assign the RenderColorTarget (for DirectX9, this is analogous to calling **IDirect3DDevice9::SetRenderTarget()**). Having an empty rightside means "set to the current enclosing-scope default" (e.g., the frame buffer). Passes can also assign their own RenderColorTarget (and these ones do!), so this redirects the renderer back to the main scene.

In the same way, we next redirect the RenderDepthStencilTarget to "" (that is, to the framebuffer).

Next, we assign the Clear color and the Clear depth to the values of the indicated global parameters, which had been declared earlier in the effect file. We're not actually clearing any buffers here, we're just setting some "state" values to be used when we do call "Clear."

Next, we clear, twice. "Clear=Color" clears the RenderColorTarget with our values from **gClearColor**, while "Clear=Depth" clears the RenderDepthStencilTarget with the values from **gClearDepth**. So after these four commands, we've drawn some buffers, come back, selected the main frame buffer, and cleared it.

Finally, we execute one more pass, named "useBakedLighting." We're done!

If there are other passes in the technique, they're ignored (this means the technique **Script** is a good tool for trying different combinations while creating and debugging, by the way).

In this sample we've already seen that we can assign render and depth targets, clear the frame (which constitutes a sort of "hidden pass"), and control the frame order. Technique **Script** annotations can perform one extra bit of magic: *Looping*.

## Looping

Here's the technique **Script** from the Shader Library "paint\_brush.fx" effect (which turns FX Composer into a very minimal paint program):

```
< string Script =
    "RenderColorTarget0=;"
    "RenderDepthStencilTarget=;"
    "LoopByCount=bReset;"
    "Pass=revert;"
    "LoopEnd=;"
    "Pass=splat;"; >
```

What it does, and how it does it:

The parameter **bReset** is a global parameter that's controlled by the application-specific semantic **FXCOMPOSER\_RESETPULSE** (which we'll also describe later on). Here is its declaration:

```
bool bReset : FXCOMPOSER_RESETPULSE <
    string UIName="Clear Canvas"; >;
```

A parameter bound to **FXCOMPOSER\_RESETPULSE** is set to one (true) if either the user selects "Clear Canvas" or if the canvas (render window) is reset – say, by a view-size change. It will be true for *that one frame*– on all subsequent frames, it will be zero (false).

Okay, so we have a value in **bReset** that is one or zero. Now we can use the **Script** commands "LoopByCount" and "LoopEnd." To give us a more elegant frame reset than just a simple "clear the screen" (in this case, our "revert" pass redraws the screen with a user-assigned background texture, which they can then paint over).

"LoopByCount" looks at the value of the designated parameter and executes the commands up to "LoopEnd" exactly that many times. In this case, the count is either one or zero. If it's zero, the enclosed commands are executed zero times – that is, the "revert" pass doesn't execute at all.

This combination of **FXCOMPOSER\_RESETPULSE** (or really, any bool parameter) and the "LoopByCount" **Script** command provides us with a somewhat kludgy but effective "if { }" mechanism.

Oh, and by the way, the indentation of the sub-strings in a **Script** means *nothing* – it just makes it easier for a person to read.

Let's look at a more traditional use of looping, this time from "scene\_reaction\_diffusion.fx." As before, we declare a parameter **bReset**, but also one called **gIterationsPerFrame**. Its declaration is (note the deliberately-clunky **UIStep**):

```
float gIterationsPerFrame <
    string UIName = "Iterations Per Frame";
    string UIWidget = "slider";
    float UIMin = 0;
    float UIMax = 100;
    float UIStep = 1;
> = 10;
```

...and the technique **Script** is:

```
< string Script =
    "LoopByCount=bReset;"
    "Pass=revert;"
    "LoopEnd=;"
    "LoopByCount=gIterationsPerFrame;"
    "Pass=simulate01;"
    "Pass=simulate10;"
    "LoopEnd=;"
    "pass=exterior01;"
    "Pass=paint;"
    "Pass=seed;"
    "Pass=clearSeed;"
    "Pass=display;"; >
```

Here we still keep our revert pass (which initializes an offscreen buffer). We also let the user actively choose how many iterations of reaction and diffusion to perform each frame.

Our third and final looping example is from "scene\_EasyBake.fx" and demonstrates the ability to push the loop count back into the shader function (in this case, to place instances of a model in varying locations). We declare a global value that will define the number of iterations, and also a global value into which we will be passing-back each current loop's count:

```
float gThisInstance <
    string UIWidget = "none"; >; // loop counter, hidden
```

then in the technique **Script**:

```
< string Script =
    "Pass=bake;"
    "RenderColorTarget0=;"
    "RenderDepthStencilTarget=;"
    "ClearSetColor=ClearColor;"
    "ClearSetDepth=ClearDepth;"
    "Clear=Color;"
    "Clear=Depth;"
    "LoopByCount=gNInstances;"
    "LoopGetIndex=gThisInstance;"
    "Pass=useBakedLighting;"
    "LoopEnd;"; >
```

Here we see the loop iteration count set by **gNInstances**, and a new **Script** command, "LoopGetIndex" is used to set the value of **gThisInstance**. That parameter can be used in any way by any passes (in this case, "useBakedLighting") within the loop.

Here is the vertex shader declared by the "useBakedLighting" pass declaration itself:

```
VertexShader = compile vs_2_0 instancesVS(gXvpXf,
                                           gThisInstance,
                                           gNInstance,
                                           gSpacing);
```

The value `gThisInstance` is passed along like any other.

## Pass Scripting and Rendering to Texture

Pass scripting, like technique scripting, can assign render targets and execute clear commands. It can be tough to decide if you should assign the targets in the technique Script or in the pass Script – in general, the method chosen for the NVIDIA Shader library has been: passes that use offscreen targets assign their own targets, while handling of the global-scope framebuffer is done by the technique. There is no hard and fast rule, but in general that has been the easiest to manage and debug.

RenderTargets are a piece of the rendering state, and *there is no state stack* – that is, if a pass assigns a RenderColorTarget, that target won't "pop" when the pass is completed – its assignment will remain until explicitly reset by itself, the technique, or another pass. Got a black frame? Make sure you're rendering where you think!

Passes can also specify what to draw in each pass – either the geometry from the scene, or a screen-aligned quadrilateral that will exactly fit the render window. We use the pass **Script** "Draw" command to choose: either "Draw=Geometry;" for (you guessed it) geometric models, or "Draw=Buffer;" for a full-screen quad. If neither is specified, a "Draw=Geometry;" will be implied at the end of your pass Script.

(Note too that if you want your **Script** to reassign the RenderTarget *after* the Geometry has drawn, that's also okay).

Here is a sample from the "scene\_uvds\_skin.fx" file:

```
pass HBlur < string Script =
    "RenderColorTarget0=gBlur1Tex;"
    "RenderDepthStencilTarget=DepthBuffer;"
    "Draw=Buffer;"
> { //...
```

Both the color and depth Render Targets are assigned, to textures that have been defined using **RENDERCOLORTARGET** or **RENDERDEPTHSTENCILTARGET**. On the next pass, any sampler that has been bound to the texture gBlur1Tex will be available for reading – as long as you change the "RenderColorTarget0" first!

After the target assignment, the full-screen quad is drawn.

Examples from the Shader Library that draw full-screen quads usually include the NVIDIA "Quad" header file (either "include/Quad.fxh" or "include/Quad.cgh," depending on the language). These files include sample vertex shaders and pixel shaders to handle the "Buffer" quadrilateral and map previously-rendered

textures exactly to the screen coordinates (which can be tricky, because of a half-pixel offset under DirectX).

## Persistence

When a reset event occurs (the same events that would trigger **FXCOMPOSER\_RESETPULSE**): all RenderTargets may be potentially reset, resized, and flushed from memory. *You must always do your own clearing of render targets*, or they will be filled with either random junk or the results of the previous render.

This isn't always a bad thing! Shader library effects like "post\_corona" (which displays an animated "flaming" halo around objects) or "post\_trail" deliberately use this feature.

## Shadow Mapping

Is a kind of RTT that will introduce yet another **Script** command, "RenderPort." We will also need to get some special matrices that describe our shadowing lamp's relationship to the rest of the scene.

### Shadow Transforms

We can use the regular automatic transform semantics, with one additional feature: the Object annotation. We assign Object to the "semantic name" of the light (see also the section above on Light and Object Binding):

```
float4x4 gLampView : View <
    string Object ="SpotLight0"; >;
float4x4 gLampProj : Projection <
    string Object = "SpotLight0"; >;
float4x4 gLampViewProj : ViewProjection <
    string Object = "SpotLight0"; >;
```

Instead of basing their values on the default camera, they will base their values on a square camera controlled by the "SpotLight0" assignment (by default, the scene's first SpotLight object).

### Shadow Bias Matrices

These you need to construct yourself. The methods are different between HLSL and Cg – the Shader Library provides and uses a header called either "include/shadowMap.fxh" or "include/shadowMap.cgh" that can manage the details of shadow bias-matrix construction in an almost invisible manner.

## RenderPort

To complete our pass, we need to let the renderer know to switch cameras and formats. We can use the "Renderport" **Script** command to do just that.

"Renderport" can be set to the name of a lamp e.g., "RenderPort=SpotLight0;" and can be reset by setting back to "" – "Renderport=" alone.

Remember to reset before your final pass!

## MRTs

Using multiple render targets is straightforward for renderers that support it.

Besides RenderColorTarget0, you may also assign RenderColorTarget1, RenderColorTarget2, etc – up to the limits of your system.

Remember to reset *all* the targets after rendering!

In the pixel shader, use “out” values with semantics **COLOR0**, **COLOR1**, etc, as in this sample with three RenderTargets:

```
void prepMRTPS(vertexOutput IN,
               uniform float3 SurfaceColor,
               uniform sampler2D ColorSampler,
               out float4 ColorOutput : COLOR0,
               out float4 NormalOutput : COLOR1,
               out float4 ViewptOutput : COLOR2)
{
    float3 Nn = normalize(IN.WorldNormal);
    NormalOutput = float4(Nn,0);
    float3 Vn = normalize(IN.WorldView);
    ViewptOutput = float4(Vn,0);
    float3 texC = SurfaceColor *
                 tex2D(ColorSampler,IN.UV).rgb;
    ColorOutput = float4(texC,1);
}
```

## Preprocess and Postprocess effects

Preprocess effects are simple. They have no geometry, and thus they should just “Draw=Buffer;”

Postprocess files are somewhat trickier as they introduce our final **Script** command, “ScriptExternal=Color;” which is only used for postprocess effects.

Here is the entire technique from the shader library “post\_negative” effect:

```

technique Main < string Script =
    "RenderColorTarget0=SceneTexture;"
    "RenderDepthStencilTarget=DepthBuffer;"
    "ClearSetColor=gClearColor;"
    "ClearSetDepth=gClearDepth;"
    "Clear=Color;"
    "Clear=Depth;"
    "ScriptExternal=Color;"
    "Pass=PostP0;";
> {
    pass PostP0 < string Script =
        "RenderColorTarget0=;"
        "RenderDepthStencilTarget=;"
        "Draw=Buffer;";
    > {
        VertexShader = compile vs_2_0
            ScreenQuadVS2(QuadTexelOffsets);
        ZEnable = false;
        ZWriteEnable = false;
        AlphaBlendEnable = false;
        CullMode = None;
        PixelShader = compile ps_2_a
            negativePS(gSceneSampler);
    }
}

```

At this point you should be able to read the entire **Script**, save for the "ScriptExternal=Color." This technique has a single pass, but before that pass it assigns and clears color and depth RenderTargets, calls "ScriptExternal, then calls the Pass – which immediately resets the RenderTargets to "" and draws a fullscreen quad using the Sampler associated with the RenderTarget that was used by the technique when it called "ScriptExternal."

Just what did "ScriptExternal" render? *Everything*.

That is, everything in the pipeline that have come before this postprocess effect: first preprocess effects, then all geometries with their object effects.

"ScriptExternal" redirects the output of these effects to its own RenderTarget, rather than the frame buffer (which should make no difference at all to the function of those underlying effects).

That's it! For a complete list of commands to use in technique and pass **Script** annotations, see the very end of this document.

# Appendices

---

## Alternate Constructions

The choices made for the formatting of NVIDIA Shader Library were made in the interest of general clarity. They're not the only way to do it!

**Shader parameter names** have been chosen to be as consistent as possible between shaders, for example, so that it's easy to copy-and-paste between different samples when creating your own new variations.

**Structs** have been used to indicate hardware-register and hardware-connector inputs and outputs. These structs are common to most of the shaders, which eliminates a lot of typing and therefore a lot of potential for error.

Similarly, except for MRTs pixel shaders usually return a float4 and the function is marked with the **COLOR0** semantic. Some users may prefer to use *void* functions and enumerate the outputs explicitly using "out" variables. Both are valid choices.

In general, as in this document, semantics are CAPITALIZED to make them easy to spot.

**Global scope:** in many older shaders, globally-scoped parameters were used. This made the code clean in some ways, but made it difficult to copy and paste such shader code into other effect files or into game engines. Newer shaders, and revisions of many old ones, have switched to using formal parameter passing, and avoiding the direct global use of the shader parameters within functions. This makes for a bulkier-looking function, but also makes it much easier to drop such a function into other non-fx-based engines.

To make global parameters easier to spot, we have adopted the "Hungarian" convention of prepending a small letter "g" before parameter names, and passing the values to shader functions by using the same name without the "g": e.g., the float4x4 parameter "**gWorldViewXf**" would be passed to the appropriated vertex shader as "uniform float4x4 **WorldViewXf**"

**HLSL virtual machine:** some HLSL effects use the virtual machine to generate textures (see "Texture Shaders" in the appendix on Sampler States, below). Some HLSL effects also include uses of the HLSL "static" construct. Their corresponding CgFX versions (if any) obviously cannot! So the calculations are usually moved to the vertex and/or fragment shaders in the Cg versions. If you intend to use Shader Library code as a starting point for your game engine, be aware that sometimes simple calculations like calculations using **TIME** variables

(which only need to be calculated once for the entire frame) can be effectively moved to the CPU for a fractional GPU boost.

We're always interested in improving NVIDIA's products. If you have ideas on improving these constructions, that may increase their power and usefulness, let us know at the NVIDIA Developer Forums, on the web at

<http://developer.nvidia.com/forums/>

---

## Sampler States (are not annotations)

FX sampler states look like annotations, but they are not. Sampler states are enclosed in { braces }, while annotations use < angle brackets > – and more tellingly, sampler states are predefined, and therefore their declarations never require a data type.

Textures, however, may have annotations. Look at the following pair:

```
texture gEnvTexture : ENVIRONMENT <
    // These are annotations
    string ResourceName = "default_reflection.dds";
    string UIName = "Environment";
    string ResourceType = "Cube";
>;

samplerCUBE gEnvSampler = sampler_state {
    // These are not annotations
    Texture = <gEnvTexture>;
    MagFilter = Linear;
    MinFilter = Linear;
    MipFilter = Linear;
    AddressU = Clamp;
    AddressV = Clamp;
    AddressW = Clamp;
};
```

The following annotations are supported for textures. Some may not be meaningful depending upon the context: **ViewportRatio** has no influence on disk-based textures, for example (while conversely **ResourceName** means nothing to a RenderTarget):

- **ViewportRatio** (float2): The dimension of a RenderTarget texture expressed as ratios relative to the viewport. “float2 **ViewportRatio** = {1,1};” results in the same dimensions as the screen. “float2 **ViewportRatio** = {0.5, 1.0};” would result in a texture ½ as wide, but just as tall. Mutually exclusive with **Dimensions**, **Width**, **Height**, and **Depth** annotations.

- **Dimensions** (int 2-3): The expected width, height, & optional depth for a procedural or RenderTarget texture. Mutually exclusive with **ViewportRatio**, **Width**, **Height**, and **Depth** annotations.
- **Width** (component of dimensions) (int): Assigns the width of a procedural or RenderTarget texture. Mutually exclusive with **Dimensions** and **ViewportRatio** annotations.
- **Height** (component of dimensions) (int): Assigns the width of a procedural or RenderTarget texture. Mutually exclusive with **Dimensions** and **ViewportRatio** annotations.
- **Depth** (component of dimensions) (int): Assigns the depth of a procedural texture. Mutually exclusive with **Dimensions** and **ViewportRatio** annotations.
- **Format** (string): Pixel format. Ignored for disk-based textures.
- **Miplevels** (int): The number of mip levels expected.
- **Levels** (int): A synonym for **MipLevels**.
- **ResourceName** (string): The name of a file that should be read by default if not overwritten by material. Meaningless for RenderTargets.
- **Function** (string): The name of the virtual-machine function used to procedurally generate this texture (*HLSL only*). See the section immediately below for details.

## Texture Shaders in the HLSL Virtual Machine

DirectX9 includes the notion of “texture shaders” – functions run on the CPU once, when the effect is loaded, to procedurally create texture data. The parameters of these shaders have their very own special usages of the **POSITION**, **PSIZE**, and **COLOR** semantics.

To use a texture shader, you write an HLSL function that will be iteratively executed on each of the texture MIP levels. It is much like a pixel shader function, and should return a value with the **COLOR** semantic. Automatic inputs to this function will have two special semantics: **POSITION** will contain XY data varying from 0 to 1, while **PSIZE** is used indicate the pixel (texel) size (and thus the function can alter its calculations according to the MIP level).

Here is a sample used by the Shader Library “stripe\_tex.fxh” header file:

```

#define STRIPE_TEX_SIZE 128

float4 make_stripe_tex(
    float2 Pos : POSITION,
    float ps : PSIZE
) : COLOR
{
    float v = 0;
    float nx = Pos.x+ps; // last column = 1 for all MIPs
    v = nx > Pos.y;
    return float4(v.xxxx);
}

texture gStripeTexture <
    string function = "make_stripe_tex"; // our function
    string UIWidget = "None";
    float2 Dimensions = {
        STRIPE_TEX_SIZE,
        STRIPE_TEX_SIZE };
>;

sampler2D gStripeSampler = sampler_state {
    Texture = <gStripeTexture>;
    MinFilter = Linear;
    MipFilter = Linear;
    MagFilter = Linear;
    AddressU = Wrap;
    AddressV = Clamp;
};

```

A cautionary note: Because they are run at effect-load time, the inputs to a texture shader must be *constants* – that is, user parameters will have no control over the texture shader (since they haven't been displayed yet by the time the texture shader runs). If you want to change a texture shader input, you must edit the code and recompile (build) the effect.

---

## Things you cannot currently do with SAS Scripts

**Loop per-object or per-light** – that is, you can't cause a loop that goes "once per object" or "for each light, do..." – Loops only function by numeric count, though you could, Loop by a count and then extract light data from arrays by using "LoopGetIndex."

The **Script** directive "LoopByType," described in some versions of the DXSAS Spec, is not implemented (nor have we seen it implemented in any other tool, so far).

**Control object rendering order** (if there is more than one object) – this is defined by the rendering application, not SAS. Order-dependant algorithms (such as transparency) need to have the data ordered properly in advance.

**Specify single objects to draw** (when there is more than one object). Object hiding is up to the rendering application. This means there no non-hacky way to use, say, shadow proxy objects.

**Call passes from other passes.**

**Persistent non-texture values** – while textures can persist from frame to frame, numeric parameter values do not. If you *really* need them, you could write them to a texture and retrieve them each frame. This has been done in FX Composer 1.8. For FX Composer 2+, why not consider a Python script? Python namespaces persist across frames. Look at the “playblast” module, distributed with FX Composer. Using “playblast()” can control the frame count and play animated clips while executing your own Python per-frame callback functions, which can load the effect parameters with the values you need – more like a regular game engine.

**Share texture buffers, explicitly under SAS control** – that is, SAS provides no mechanism for sharing image buffers between effects – say a shadow map that would be used by multiple material effects, dynamic refelction maps, or full-scene deferred-render buffers. There are some possibilities, however...

## How to Share Textures: Two Methods

**COLLADA method:** FX Composer supports, for COLLADA-FX only the ability to write to a shared texture surface. Other COLLADA-FX effects (and CgFX effects) can access this texture from their properties pane. This is how the NVIDIA “Ninja Scene” was created, where the environment and the running Ninja can shared shadows. See the NVIDIA Developer Forums post at <http://developer.nvidia.com/forums/index.php?showtopic=267> for technical details.

**Autodesk 3DS Max method:** while not exactly controlled by SAS, 3DS Max has a slightly-different mechanism for defining preprocess, standard, and postprocess effects. The result is that Max scenes can indeed render themselves and use shared shadows, reflections, and so forth. For details, check the web for the Autodesk Presentation “Integrating High-Level Shading Effects into Autodesk 3ds Max and Maya,” which has been updated repeatedly by Autodesk and available from multiple sources.

*(A “secret” third method? The HLSL specification provides an explicit “shared” storage classifier for global parameters – the intent was to provide a mechanism for sharing both texture buffers and parameter values between effects – e.g., “shared float AA;” would provide the same numeric values to any loaded effects that contained a reference to that shared “AA” parameter. Alas, this keyword has so far never appeared in a running HLSL implementation)*

---

## Semantics and Python

Here is a quick IronPython snippet that can be used within *FX Composer* to view all the semantics recognized by your current edition of the program:

```
Import FXComposer
[s for s in dir(FXComposer.Core.FXSemanticID) if \
    not callable(getattr(FXComposer.Core.FXSemanticID.s)) \
    and not s.__contains__('_')]
```

---

## List of Common Semantics

### Hardware Semantics

#### Register Names in vertex buffers and vertex-output connections

**COLOR0**

**COLOR1**

**TEXCOORD0**

**TEXCOORD1**

**TEXCOORD2**

**TEXCOORD3**

**TEXCOORD4**

**TEXCOORD5**

**TEXCOORD6**

**TEXCOORD7**

**TEXCOORD8** (synonym for **COLOR0**)

**TEXCOORD9** (synonym for **COLOR1**)

**ATTR0** (synonym for **TEXCOORD0** sometimes seen in Cg)

**ATTR1** (synonym for **TEXCOORD1**)

**ATTR2** (synonym for **TEXCOORD2**)

**ATTR3** (synonym for **TEXCOORD3**)

**ATTR4** (synonym for **TEXCOORD4**)

**ATTR5** (synonym for **TEXCOORD5**)

**ATTR6** (synonym for **TEXCOORD6**)

**ATTR7** (synonym for **TEXCOORD7**)

**register(hardware\_register\_specifier)**

**POSITION**  
**NORMAL**  
**NORMAL0**  
**TANGENT**  
**TANGENT0**  
**BINORMAL**  
**BINORMAL0**

#### **Particle Size and Polygon Facing Flag Inputs for Pixel Shaders**

**PSIZE**  
**FACE** (OpenGL equivalent to **VFACE**)  
**VFACE** (DirectX equivalent to **FACE**)

#### **Pixel Shader Output Values**

**COLOR** (synonym for **COLOR0**)  
**COLOR0**  
**COLOR1**  
**COLOR2**  
**COLOR3**

## Standard Software Semantics

#### **Common Material and Light Characteristics**

**POSITION**  
**DIRECTION**  
**DIFFUSE**  
**SPECULAR**  
**AMBIENT**  
**POWER**  
**SPECULARPOWER**  
**ATTENUATION**  
**CONSTANTATTENUATION**  
**LINEARATTENUATION**  
**QUADRATICATTENUATION**  
**FALLOFFANGLE**  
**FALLOFFEXPONENT**

EMISSION  
EMISSIVE  
OPACITY  
REFRACTION

#### Texture-Related

RENDERDEPTHSTENCILTARGET  
RENDERCOLORTARGET  
VIEWPORTPIXELSIZE  
DIFFUSEMAP  
SPECULARMAP  
NORMAL  
ENVIRONMENT  
ENVMAP  
ENVIRONMENTNORMAL

#### Transforms & Locations

WORLD  
VIEW  
PROJECTION  
WORLDVIEW  
VIEW PROJECTION  
WORLDVIEWPROJECTION  
WORLDINVERSE  
VIEWINVERSE  
PROJECTIONINVERSE  
WORLDVIEWINVERSE  
VIEW PROJECTIONINVERSE  
WORLDVIEWPROJECTIONINVERSE  
WORLDTRANPOSE  
VIEWTRANPOSE  
PROJECTIONTRANPOSE  
WORLDVIEWTRANPOSE  
VIEW PROJECTIONTRANPOSE  
WORLDVIEWPROJECTIONTRANPOSE  
WORLDINVERSETRANPOSE

**VIEWINVERSETRANSPOSE**  
**PROJECTIONINVERSETRANSPOSE**  
**WORLDVIEWINVERSETRANSPOSE**  
**VIEW PROJECTIONINVERSETRANSPOSE**  
**WORLDVIEWPROJECTIONINVERSETRANSPOSE**  
**TRANSFORM**  
**LIGHTPOSITION**  
**CAMERAPOSITION**

#### **Others**

**STANDARDGLOBAL**  
**HEIGHT**  
**UNITSCALE**  
**UNKNOWN**  
**USER**  
**NONE**

## FXComposer Specific Semantics

#### **Resetting Views**

**FXCOMPOSER\_RESETPULSE**

#### **Mouse Interactions**

**MOUSEPOS**  
**LEFTMOUSEDOWN**  
**RIGHTMOUSEDOWN**

#### **Timing**

**TIME**  
**ANIMATIONTICK**  
**ANIMATIONTIME**

## Other Semantics Sometimes Encountered

**SI\_EYEPOS** (use **CAMERAPOSITION** or **VIEWINVERSE** instead)  
**WORLD\_CAMERA\_POSITION** (likewise)  
**COS\_TIME** (just call cos())

If you encounter any new semantics, please send them to us on the NVIDIA forums: <http://developer.nvidia.com/forums/>

---

## List of Scripting Commands

Pass=*pass\_name*  
 RenderColorTarget0=*render\_target\_or\_null*  
 RenderColorTarget1=*render\_target\_or\_null*  
 RenderColorTarget2=*render\_target\_or\_null*  
 RenderColorTarget3=*render\_target\_or\_null*  
 RenderDepthStencilTarget=*render\_target\_or\_null*  
 ClearSetColor=*color\_parameter\_name*  
 ClearSetDepth=*depth\_parameter\_name*  
 Clear=Color  
 Clear=Depth  
 Draw=Geometry  
 Draw=Buffer  
 LoopByCount=*parameter\_name*  
 LoopGetIndex=*parameter\_name*  
 LoopEnd=  
 RenderPort=*light\_identifier\_or\_null*  
 ScriptExternal=Color

⋮

### **Notice**

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent or patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. NVIDIA Corporation products are not authorized for use as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

### **Trademarks**

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

### **Macrovision Compliance Statement**

NVIDIA Products that are Macrovision enabled can only be sold or distributed to buyers with a valid and existing authorization from Macrovision to purchase and incorporate the device into buyer's products.

Macrovision copy protection technology is protected by U.S. patent numbers 4,631,603, 4,577,216 and 4,819,098 and other intellectual property rights. The use of Macrovision's copy protection technology in the device must be authorized by Macrovision and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by Macrovision. Reverse engineering or disassembly is prohibited

### **Copyright**

© 2008 NVIDIA Corporation. All rights reserved.



**NVIDIA.**