Cg Toolkit

CgFX 1.2
Overview

CgFX 1.2 Overview

What is CgFX?

CgFX is a powerful and versatile shader specification and interchange format. The CgFX Runtime, like Cg, supports OpenGL as well as DirectX 8 and DirectX 9. For artists and developers of real-time graphics, this format provides several key benefits:

- Cross-API, cross-platform compatibility and portability.
- Encapsulation of multiple techniques, enabling fallbacks for level-of-detail, functionality, or performance.
- Support for Cg, assembly language, and fixed-function shaders.
- Editable parameters and GUI descriptions embedded in the file.
- Multipass shaders.
- Render state and texture state specification.

In practical terms, by wrapping both Cg vertex programs and Cg fragment programs together with render state, texture state, and pass information, developers can describe a complete rendering effect in a CgFX file. Although individual Cg programs may contain the core rendering algorithms necessary for an effect, only when combined with this additional environmental information does the shader become complete and self-contained. The addition of artist-friendly GUI descriptions and fallbacks enables CgFX files to integrate well with the production workflow used by artists and programmers.

CgFX File Format Overview

The Cg language lets you easily express how an object should be rendered. Although current Cg profiles describe only a single rendering pass, many shading techniques, such as shadow volumes or shadow maps, require more than one rendering pass.

Many applications need to target a wide range of graphics hardware functionality and performance. Thus, versions of shaders that run on older hardware, and versions that aid performance for distant objects are important.

Each Cg program typically targets a single profile, and doesn't specify how to fall back to other profiles, to assembly-language shaders, or to fixed-function vertex or fragment processing.

To generate images with Cg programs, some information about their environment is needed. For instance, some programs might require alpha blending to be turned on and depth writes to be disabled. Others may need a certain texture format to work correctly. This information is not present in standard Cg source files.

CgFX addresses these kinds of issues through a text-based file containing Cg, assembly, and fixed-function shaders, along with the render states and environment information needed to render the effect. This text file syntax is similar to the Microsoft .fx 2.0 format (the DirectX 9.0 Effect format).

CgFX encapsulates, in a single text file, everything needed to apply a rendering effect. This feature lets a third-party tool or another 3D application use a CgFX text file as is, with no external information other than the necessary geometry and texture data. In this sense, CgFX acts as an interchange format. CgFX allows shaders to be exchanged without the associated C++ code that is normally necessary to make a Cg program work with OpenGL or Direct3D.

Techniques

Each CgFX file usually presents a certain effect that the shader author is trying to achieve—such as bump mapping, environment mapping, or anisotropic lighting. The CgFX file contains one or more *techniques*, each of which describes a way to achieve the effect. Each technique usually targets a certain level of GPU functionality, so a CgFX file may contain one technique for an advanced GPU with powerful fragment programmability, and another technique for older graphics hardware supporting fixed-function texture blending. CgFX techniques can also be used for functionality, level-of-detail, or performance fallbacks. For example:

```
effect myEffectName
{
  technique PixelShaderVersion
  {…};

  technique FixedFunctionVersion
  {…};

  technique LowDetailVersion
  {…};
};
```

An application can make queries about which techniques are present in an effect and can choose an appropriate one at runtime, based on whatever criteria are appropriate.

Passes

Each technique contains one or more *passes*. Each pass represents a set of render states and shaders to apply for a single rendering pass within a technique. For instance, the first pass might lay down depth only, so that subsequent passes can apply an additive alpha-blending technique without requiring polygon sorting.

Each pass may contain a vertex program, a fragment program, or both, and each pass may use fixed-function vertex or pixel processing, or both. For example, a first pass might use fixed-function pixel processing to output the ambient color. The next pass could use a **`ps_1_1`** fragment program, and pass 2 might use a **`ps_2_0`** fragment

program. In practice, all passes within a technique typically use fixed-function processing, or all use Cg or assembly programs. This method prevents depth-fighting artifacts that can occur when the fixed-function and programmable parts of some GPUs process the same data in different ways.

Render States

Each pass also contains render states such as alpha blending, depth writes, and texture filtering modes, to name a few. For example:

```
pass firstPass
{
   DepthWriteEnable = true;
   AlphaBlendEnable = false;
   MinFilter[ 0 ] = Linear;
   MagFilter[ 0 ] = Linear;
   MipFilter[ 0 ] = Linear;

   // Pixel shader written in assembly
   PixelShader = asm
   {
      ps.1.1
      tex t0;
      mov r0, t0;
   };
};
```

Notice that CgFX, in addition to embedding Cg programs, allows you to encode assembly-language vertex and fragment programs with the **asm** keyword.

Variables and Semantics

Finally, the CgFX file contains global and per-technique Cg-style variables. These variables are usually passed as uniform parameters to Cg functions, or as the value for a render or texture state setting. For instance, a **bool** variable might be used as a uniform parameter to a Cg function, or as a value enabling or disabling the alpha blend render state:

```
bool AlphaBlending = false;
float bumpHeight = 0.5f;
```

These variables can contain a user-defined semantic, which helps applications provide the correct data to the shader without having to decipher the variable names:

```
float4x4 myViewMatrix : ViewMatrix;
texture2D someTexture : DiffuseMap;
```

A CgFX-enabled application can then query the CgFX file for its variables and their semantics.

Annotations

Additionally, each variable can have an optional *annotation*. The annotation is a per-variable-instance structure that contains data that the effect author wants to communicate to a CgFX-aware application, such as an artist tool. The application can

then allow the variable to be manipulated, based on a GUI element that is appropriate for the type of annotation.

An annotation can be used to describe a user interface element for manipulating uniform variables, or to describe the type of render target a rendering pass is expecting.

```
float bumpHeight
<
   string gui  = “slider”;
   float  uimin  = 0.0f;
   float  uimax  = 1.0f;
   float  uistep = 0.1f;
> = 0.5f;
```

The annotation appears after the optional semantic, and before variable initialization. Applications can query for annotations, and use them to expose certain parameters to artists in a CgFX-aware tool, such as Discreet’s 3ds max 5 or Alias|Wavefront’s Maya 4.5.

A Sample CgFX File

The example below shows a sample CgFX file that calculates basic diffuse and specular lighting.

```
struct VS_INPUT
{
   float4 vPosition  : POSITION;
   float4 vNormal    : NORMAL;
   float4 vTexCoords : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4 vTexCoord0 : TEXCOORD0;
   float4 vDiffuse   : COLOR0;
   float4 vPosition  : POSITION;
   float4 vSpecular  : COLOR1;
};

VS_OUTPUT myvs(uniform float4x4 ModelViewProj,
               uniform float4x4 ModelView,
               uniform float4x4 ModelViewIT,
               uniform float4x4 ViewIT,
               uniform float4x4 View,
               const VS_INPUT vin,
               uniform float4 lightPos,
               uniform float4 diffuse,
               uniform float4 specular,
               uniform float4 ambient)
{
  VS_OUTPUT vout;
  float4 position = mul(ModelView, vin.vPosition);
  float4 normal =   mul(ModelViewIT, vin.vNormal);

  float4 viewLightPos = mul(View, lightPos);
  float4 lightvec = normalize(viewLightPos - position);
  float4 eyevec =   normalize(ViewIT[3]);
  float self_shadow = max(dot(normal, lightvec), 0);

  float4 halfangle = normalize(lightvec + eyevec);
  float spec_term =  max(dot(normal, halfangle), 0);

  float4 diff_term = ambient + diffuse * self_shadow +
                     self_shadow * spec_term * specular;
  vout.vDiffuse = diff_term;
  vout.vPosition = mul(ModelViewProj, vin.vPosition);
  return vout;
}


float4x4 vit     : ViewIT;
float4x4 viewmat : View;
float4x4 mv      : WorldView;
float4x4 mvit    : WorldViewIT;
float4x4 mvp     : WorldViewProjection;
float4 diffuse   : DIFFUSE = { 0.1f, 0.1f, 0.5f, 1.0f };
float4 specular  : SPECULAR = { 1.0f, 1.0f, 1.0f, 1.0f };
float4 ambient   : AMBIENT = { 0.1f, 0.1f, 0.1f, 1.0f };

float4 lightPos : Position
<
   string Object = "PointLight";
   string Space = "World";
> = { 100.0f, 100.0f, 100.0f, 0.0f };

technique t0
{
  pass p0
```

```
  {
    Zenable = true;
    ZWriteEnable = true;
    CullMode = None;
    VertexShader = compile vs_1_1 myvs( mvp, mv, mvit, vit,
                                        viewmat, lightPos,
                                        diffuse, specular,
                                        ambient);
  }
}
```

A Sample CgFX File

CgFX Runtime API Overview

The CgFX Runtime API provides functions and interfaces for creating and using effects. One of the key ideas in the API is the difference between an effect and an effect compiler. Given a CgFX input file, either an effect object or an effect compiler object can be created to represent the effect inside the file. The difference between the two is that an effect represents a specialized version of the effect for use in a particular API (OpenGL or Direct3D) and specialized for use in a particular rendering device. An effect compiler represents the effect independently of any particular API. The effect compiler therefore supports fewer operations than effects, though it does have a method to create the effect that corresponds to it.

Effect Creation Functions

There are four creation methods, two for effects and two for effect compilers. For each type, one takes a string holding the effect, and the other takes the path to a filename holding the effect.

- **CgFXCreateEffect**(LPCSTR pSrcData, const char *compilerArgs, ICgFXEffect **ppEffect, const char **ppCompilationErrors) – Creates an effect from a string containing the effect source and stores it in *ppEffect. Returns any compilation errors in ppCompilationErrors. Returns a HRESULT success code.
- **CgFXCreateEffectFromFileA**(LPCSTR pSrcFile, const char *compilerArgs, ICgFXEffect **ppEffect, const char **ppCompilationErrors) – Creates an effect from a file containing the effect source and stores it in *ppEffect. Returns any compilation errors in ppCompilationErrors. Returns a HRESULT success code.
- **CgFXCreateEffectCompiler**(LPCSTR pSrcData, const char *compilerArgs, ICgFXEffectCompiler** ppEffectCompiler, const char **ppCompilationErrors) – Creates an effect compiler from a string containing the effect source. Returns any compilation errors in ppCompilationErrors. Returns a HRESULT success code.
- **CgFXCreateEffectCompilerFromFileA**(LPCSTR pSrcFile, const char *compilerArgs, ICgFXEffectCompiler** ppEffectCompiler, const char **ppCompilationErrors) – Creates an effect compiler from a file containing the effect source. Returns any compilation errors in ppCompilationErrors. Returns a HRESULT success code.

Device and Resource Management Functions

- **`CgFXSetDevice`**`(const char* pDeviceType, LPVOID pDevice)` – Set the current device to be used for all subsequent effect invocations. The device types supported are: `"Direct3D8"`, `"Direct3D9"` and `"OpenGL"`. The parameter `pDevice` should contain a pointer to the particular (hardware) device.

- **`CgFXFreeDevice`**`(const char* pDeviceType, LPVOID pDevice)` – Frees the specified device for the particular device type. All resources associated with the device are released.

- **`CgFXGetErrors`**`(const char** ppErrors)` – Gets the current error string.

Interfaces

There are three key interfaces in the CgFX API:

- **`ICgFXBaseEffect`** – Base interface for **`ICgFXEffect`** and **`ICgFXEffectCompiler`**. Pure base-class with no direct implementation.
- **`ICgFXEffect`** – Interface derived from **`ICgFXBaseEffect`**. Represents a device-specific effect. An object of type **`ICgFXEffect`** can be created from an effect file through invocation of **`CgFXCreateEffectFromFileA()`** or from a string via **`CgFXCreateEffect()`**.
- **`ICgFXEffectCompiler`** – Interface derived from **`ICgFXBaseEffect`**. Represents a device-independent effect. An object of this type can be created from an effect file through invocation of **`CgFXCreateEffectCompilerFromFileA()`** or from a string via **`CgFXCreateEffectCompiler()`**. This provides a device-independent proxy for an effect when one is needed. An object of this type can be used to create an **`ICgFXEffect`** with a particular device by invoking the member function **`CompileEffect`**.

ICgFXBaseEffect Methods

One of the key ideas in the CgFX API is the notion of a handle. The **`CGFXHANDLE`** type is an opaque data type that is returned from many CgFX routines. It can be tested for **`NULL`**, which indicates that the routine that returned it failed for some reason, but otherwise has no user-visible semantics. The handle then is passed into other routines and acts as a reference to some item.

For example, **`ICgFXBaseEffect`** has two methods that return the handle representing one of the techniques in an effect:

```
CGFXHANDLE GetTechnique(UINT index);
CGFXHANDLE GetTechniqueByName(LPCSTR name);
```

The first method returns a handle to the *n*th technique in the effect, and the second returns a handle to the named technique. If *index* is greater than the number of techniques, or if there is no technique named *name*, the two respective methods return a **`NULL`** handle.

Other API methods take handles as parameters. For example,

```
HRESULT GetTechniqueDesc(CGFXHANDLE pTechnique,
   CgFXTECHNIQUE_DESC* pDesc);
```

Returns information about a technique in a **CgFXTECHNIQUE_DESC** structure:

```
struct CgFXTECHNIQUE_DESC {
    LPCSTR Name;
    UINT Annotations;
    UINT Passes;
};
```

The **GetTechniqueDesc()** method must be passed a **CGFXHANDLE** to a technique that was returned by **GetTechnique()** or **GetTechniqueByName()**.

In a similar manner, given the handle to a technique, there are methods to return the handle to one of the passes in the technique:

```
CGFXHANDLE GetPass(CGFXHANDLE technique, UINT index);
CGFXHANDLE GetPassByName(CGFXHANDLE technique, LPCSTR name);
```

These handles can be passed to the **GetPassDesc()** method, which returns information about the pass in a structure.

```
HRESULT GetPassDesc(CGFXHANDLE pPass, CgFXPASS_DESC* pDesc);
```

```
struct CgFXPASS_DESC {
    LPCSTR Name;
    UINT Annotations;
    DWORD VSVersion;
    DWORD PSVersion;
    UINT VertexVaryingUsed;
    CgFXVARYING VertexVarying[MAX_CGFX_DECL_LENGTH];
    UINT FragmentVaryingUsed;
    CgFXVARYING FragmentVarying[MAX_CGFX_DECL_LENGTH];
    UINT PSSamplersUsed;
    CgFXSAMPLER_INFO PSSamplers[16];
};
```

There is also a method that returns information about the complete effect. No handle is necessary, since the **ICgBaseEffect** pointer identifies the effect.

```
HRESULT GetDesc(CgFXEFFECT_DESC* pDesc);
```

Some of the key functions provided by **ICgBaseEffect** implementations center around parameter management. The values of parameters to the vertex and fragment programs in the effect can be queried and set via these methods. First, there are four methods to get the handles to parameters. The first parameter to each of them is a **CGFXHANDLE**, which can either be **NULL**, indicating that a regular parameter is desired, or it can be the handle to another parameter that is a structure, if the handle to a member of the structure is wanted.

```
CGFXHANDLE GetParameter(CGFXHANDLE parent, UINT index);
CGFXHANDLE GetParameterByName(CGFXHANDLE parent, LPCSTR name);
CGFXHANDLE GetParameterBySemantic(CGFXHANDLE parent, LPCSTR name);
CGFXHANDLE GetParameterElement(CGFXHANDLE parent, UINT element);
```

In the CgFX API, annotations are managed by many of the parameter-related methods (e.g. getting and setting their values, querying their type, etc.). The first parameter to the methods to get handles to annotations is a **CGFXHANDLE** to a parameter, a technique, or

a pass, giving the object that has the annotation. The second parameter identifies the particular annotation attached to the particular entity.

```
CGFXHANDLE GetAnnotation(CGFXHANDLE object, UINT num);
CGFXHANDLE GetAnnotationByName(CGFXHANDLE object, LPCSTR name);
```

These handles to parameters and to annotations can be used to get information about the parameter or annotation:

```
HRESULT GetParameterDesc(CGFXHANDLE pParameter,
   CgFXPARAMETER_DESC* pDesc);
```

This method returns information about it in a **CgFXPARAMETER_DESC** structure.

```
struct CgFXPARAMETER_DESC {
    LPCSTR Name;
    LPCSTR Semantic;
    CgFXPARAMETERCLASS Class;
    CgFXPARAMETERTYPE Type;
    UINT Rows, Columns, Elements;
    UINT Annotations;
    UINT StructMembers;
    UINT Bytes;
    DWORD Flags;
};
```

Furthermore, these handles can be used to get and set the values of parameters via the following methods. These methods will fail and return an error code in **HRESULT** if the wrong number of array elements are passed or if the type of the parameter doesn't match the method used.

```
HRESULT SetFloat(CGFXHANDLE pName, FLOAT f);
HRESULT GetFloat(CGFXHANDLE pName, FLOAT* f);
HRESULT GetFloatArray(CGFXHANDLE pName, FLOAT* f, UINT count);
HRESULT SetFloatArray(CGFXHANDLE pName, const FLOAT* f, UINT count);
HRESULT GetInt(CGFXHANDLE pName, int *value);
HRESULT SetInt(CGFXHANDLE pName, int value);
HRESULT GetIntArray(CGFXHANDLE pName, int *value, UINT count);
HRESULT SetIntArray(CGFXHANDLE pName, const int *value, UINT count);
HRESULT SetDWORD(CGFXHANDLE pName, DWORD dw);
HRESULT GetDWORD(CGFXHANDLE pName, DWORD* dw);
HRESULT SetBool(CGFXHANDLE pName, bool bvalue);
HRESULT GetBool(CGFXHANDLE pName, bool* bvalue);
HRESULT GetBoolArray(CGFXHANDLE pName, bool *bvalue, UINT count);
HRESULT SetBoolArray(CGFXHANDLE pName, const bool *bvalue,
    UINT count);
```

There are also specialized methods for setting the values of vector and matrix parameters. Matrices can optionally be transposed before being bound to shaders.

```
HRESULT SetVector(CGFXHANDLE pName, const float *pVector, UINT
vecSize);
HRESULT GetVector(CGFXHANDLE pName, float *pVector, UINT vecSize);
HRESULT SetVectorArray(CGFXHANDLE pName, const float *pVector,
   UINT vecSize, UINT count);
HRESULT GetVectorArray(CGFXHANDLE pName, float *pVector,
    UINT vecSize, UINT count);
HRESULT SetMatrix(CGFXHANDLE pName, const float* pMatrix,
   UINT nRows, UINT nCols);
HRESULT GetMatrix(CGFXHANDLE pName, float* pMatrix, UINT nRows,
   UINT nCols);
HRESULT SetMatrixArray(CGFXHANDLE pName, const float* pMatrix,
   UINT nRows, UINT nCols, UINT count);
HRESULT GetMatrixArray(CGFXHANDLE pName, float* pMatrix,
    UINT nRows, UINT nCols, UINT count);
HRESULT SetMatrixTranspose(CGFXHANDLE pName, const float* pMatrix,
    UINT nRows, UINT nCols);
HRESULT GetMatrixTranspose(CGFXHANDLE pName, float* pMatrix,
    UINT nRows, UINT nCols);
HRESULT SetMatrixTransposeArray(CGFXHANDLE pName,
    const float *pMatrix, UINT nRows, UINT nCols, UINT count);
HRESULT GetMatrixTransposeArray(CGFXHANDLE pName, float *pMatrix,
    UINT nRows, UINT nCols, UINT count);
```

Finally, the following methods are used to set and get identifiers for parameters of the appropriate type. The methods will fail if the type does not match that of the parameter.

```
HRESULT SetTexture(CGFXHANDLE pName, DWORD textureHandle);
HRESULT GetTexture(CGFXHANDLE pName, DWORD* textureHandle);
HRESULT SetVertexShader(CGFXHANDLE pName, DWORD vsHandle);
HRESULT GetVertexShader(CGFXHANDLE pName, DWORD* vsHandle);
HRESULT SetPixelShader(CGFXHANDLE pName, DWORD psHandle);
HRESULT GetPixelShader(CGFXHANDLE pName, DWORD* psHandle);
```

ICgFXEffect Members

A **`ICgFXEffect`** interface object can be used to manage and use techniques for rendering. Not all of the techniques in an effect may be usable, so the **`ValidateTechnique()`** method can be used to determine if a particular technique can be run using the particular API being used and graphics hardware available on the system.

```
HRESULT ValidateTechnique(CGFXHANDLE technique);
```

Validation can fail for any of the following reasons:

- If an invalid state is used for any pass within the technique
- If an invalid value is set for any state in any pass within the technique
- If assembly vertex or pixel shaders do not compile for the particular device.
- If invalid shaders are assigned to vertex or pixel shaders.
- If the device does not support the functionality required to implement an effect. (For example, a DirectX8 device will not support vs_2_0 and ps_2_0 shaders.)

It is possible to iterate through the valid techniques. If a **NULL** handle is passed in **hTechnique**, the first valid technique is returned. Otherwise, the next valid technique after **hTechnique** is returned.

```
HRESULT FindNextValidTechnique(CGFXHANDLE hTechnique,
    CGFXHANDLE *pTechnique);
```

The current technique can be set by passing a technique's handle to **SetTechnique()**.

```
HRESULT SetTechnique(CGFXHANDLE pTechnique);
CGFXHANDLE GetCurrentTechnique();
```

After the current technique has been set with **SetTechnique()**, the technique's passes can be run in order. The **Begin()** method prepares to run the technique; it returns the total number of passes in **pPasses**. Before each pass, **Pass()** should be called with the pass number before the geometry is drawn for that pass. After the last pass is finished, the **End()** method should be called. CgFX automatically handles loading the appropriate vertex and fragment shaders and/or setting up state in the fixed-function pipeline before each pass.

```
HRESULT Begin(UINT* pPasses, DWORD Flags);
HRESULT Pass(UINT passNum);
HRESULT End();
```

The **CloneEffect()** method creates a duplicate of the given effect. The new effect will use the same device as that of the original. Device-specific entities such as texture and shader handles will not be cloned and need to be set for the effect to provide a complete replica.

```
HRESULT CloneEffect(ICgFXEffect** ppNewEffect);
```

A few methods support device management for the Direct3D API. **OnLostDevice**() releases all device-specific resources associated with the effect. **OnResetDevice**() should be called if the application-provided device is reset, and **OnLostDevice()** should be called if it is lost.

```
HRESULT GetDevice(LPVOID* ppDevice);
HRESULT OnLostDevice();
HRESULT OnResetDevice();
```

ICgFXEffectCompiler Members

The **ICgFXEffectCompiler** interface only provides one method beyond those in the base **ICgFXBaseEffect** interface—**CompileEffect()**, which compiles the device-independent **ICgFXEffectCompiler** to a device-specific **ICgFXEffect**.

```
HRESULT CompileEffect(const char **compilerArgs,
    ICgFXEffect** ppEffect, const char** ppCompilationErrors);
```

Differences With Respect To Direct3D FX

CgFX provides similar functionality and uses a similar file format to the Direct3D FX routines and file format. The main differences between them include:

- CgFX supports additional profiles for shaders for OpenGL, including arbvp1, arbfp1, vp20, vp30, fp20, and fp30.

- CgFX does not support evaluating Cg functions and directly storing their results into texture maps

Cg Plug-ins Supporting the CgFX Format

At the time of publication, Cg plug-ins are available for major digital content creation (DCC) applications, such as Alias|Wavefront's Maya 4.5 and Discreet's 3ds max 5, which directly support the CgFX format.

The *Cg Plug-in for 3ds max* allows an artist to view and adjust the editable parameters of a CgFX shader right from within 3ds max. All changes made to the shader settings are displayed in real time in the native 3ds max viewports while running max under DirectX. This affords the artist more direct control of real-time 3D shaders.

The *Cg Plug-in for Maya* also allows an artist to view and adjust the editable parameters of a Cg shader "live," right within Maya's shading editor windows (such as the attribute editor and animation graph windows). Again, changes made to the shader settings are displayed in real time in Maya's OpenGL viewports.

Learning More About CgFX

CgFX-related software is available from the NVIDIA Cg Web site:

http://developer.nvidia.com/Cg

Refer to this site often to keep up with the latest applications, plug-ins, and other software that leverages the CgFX file format. Information on how to report any bugs you may find in the release is also available on this site. Also, see the *DirectX 9.0* effect reference documentation for additional specification details and examples.

Notice

Trademarks

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com