PowerVR Effect File PFX

Reference

Filename	:	PowerVR Effect File PFX.Reference.1.7f.External.doc
Version	:	1.7f External Issue (Package: POWERVR SDK 2.06.26.0716)
Issue Date	:	25 Feb 2010
Author	:	POWERVR

Contents

List of Figures

Error! No table of figures entries found.

1. Introduction

The POWERVR Effect File (PFX) is both a file format and an API. Both are described here with examples of PFX files and their use.

The value of effect files has already been demonstrated by developers in the PC games market, by game engines using Microsoft Effect Files (FX files) as supported by the DirectX SDK. The DirectX SDK comes supplied with helper libraries which, among many other things, can load an FX file and set up the state engine of the renderer (Direct3D) as described in the file.

Other parties with a greater interest in OpenGL have also seen the value of an effect file, and commenced development of COLLADA FX. COLLADA FX is part of the larger COLLADA effort, taken over by the Khronos Group. As of January 2007 (COLLADA 1.4.1), supported profiles are Cg, OpenGL ES and GLSL.

Both of these are very good, very complete designs that offer a lot of power and flexibility.

1. POWERVR Effect Files (PFX)

PFX is designed for OpenGL 2 and OpenGL ES 2 to be a small, simple, easy to use format and runtime (the runtime, in this case, could be a DLL, a statically-linked LIB, or even just a few CPP and H files). In essence it is the simplest format that allows the use of shared effect files and thereby a shader editor.

PFX is by no means a replacement for COLLADA FX; in COLLADA terminology PFX is a delivery format. Indeed shader editors which use that format could well have a plug-in to export to PFX. Such would be an identical approach to that used for scene and geometry data, where the developer exports data into a delivery format (such as POD) from a 3D content creation application such as Autodesk 3ds Max or Maya.

1.1. Example PFX file

A PFX file is a plain text file. Single-line comments are demarked with a double-forward slash, as used by C++: “//”.

```
[HEADER]
	VERSION		01.00.00.00
	DESCRIPTION	texture example
	COPYRIGHT	Imagination Technologies
[/HEADER]

[TEXTURES]
	FILE	marble	maskmain.pvr	LINEAR-LINEAR-LINEAR
[/TEXTURES]

[VERTEXSHADER]
	NAME		MyVertexShader

	// LOAD GLSL AS CODE
	[GLSL_CODE]
		attribute vec4 myVertex;
		attribute vec2 myUVMain;

		uniform mat4	myWVPMatrix;

		varying vec2	texCoordinateMain;

		void main(void)
		{
			gl_Position		= myWVPMatrix * myVertex;
			texCoordinateMain	= myUVMain.st;
		}
	[/GLSL_CODE]
[/VERTEXSHADER]

[FRAGMENTSHADER]
	NAME		MyFragmentShader

	// LOAD GLSL AS CODE
	[GLSL_CODE]
		uniform sampler2D	sampler2d;
		varying highp vec2	texCoordinateMain;

		void main (void)
		{
			gl_FragColor = texture2D(sampler2d, texCoordinateMain);
		}
	[/GLSL_CODE]
[/FRAGMENTSHADER]

[EFFECT]
	NAME		MyEffect1

	// GLOBALS UNIFORMS
	UNIFORM		myWVPMatrix		WORLDVIEWPROJECTION
	UNIFORM		sampler2d		TEXTURE0

	ATTRIBUTE	myVertex		POSITION
	ATTRIBUTE	myUVMain		UV0

	VERTEXSHADER	MyVertexShader
	FRAGMENTSHADER	MyFragmentShader
	TEXTURE		0			marble
[/EFFECT]
```

1.2. The HEADER block

Three items are required here:

1. The version of PFX used, to allow future changes to the format.
2. A description of what the PFX file does.
3. A copyright string.

1.3. The TEXTURES block

A list of textures, each with:

1. A texture name, to be referenced from the EFFECT block(s).

2. A texture filename, to be used by the application code (as only it knows from where to load textures).
3. Optional initial filter state (in the order “min,” “mag” then “MIP”) and then optional wrapping state (in the order S, T, R).

1.4. The VERTEXSHADER and FRAGMENTSHADER blocks

There must be at least one of each of these. Each contains a “NAME”, so that it can be referenced from the “EFFECT” block(s), and some GLSL code using one of the following methods:

1. A “FILE” statement followed by a filename. That file must contain exactly the text to be passed to the GLSL compiler.
2. A “GLSL_CODE” block, the contained lines of which are passed verbatim to the GLSL compiler.

1.5. The EFFECT block

1.5.1. Keyword NAME

This may be specified once only; it is the name that identifies the effect, and which is passed into the PVRTPFX API to load an effect from the parsed PFX file.

1.5.2. Section [ANNOTATION]

Lines of text between the “[ANNOTATION]” and “[/ANNOTATION]” lines are placed directly into a string and made available to the application code. Essentially this is ‘user data’ to be used for whatever purpose the application desires.

1.5.3. Keyword VERTEXSHADER

Used once to specify which vertex shader to use from the PFX file.

1.5.4. Keyword FRAGMENTSHADER

Used once to specify which fragment shader to use from the PFX file.

1.5.5. Keyword UNIFORM

Can be used multiple times; each entry specifies a ‘uniform’ variable name in the GLSL program (i.e. no distinction is made between whether the uniform is used in the vertex shader or in the fragment shader), and secondly a semantic.

1.5.6. Keyword ATTRIBUTE

Can be used multiple times; each entry specifies an ‘attribute’ variable name in the vertex shader, and a semantic.

1.5.7. Keyword TEXTURE

Can be used multiple times; each entry specifies a texture number to set, and which texture (from the “[TEXTURES]” block) to set.

1.6. Semantics

Semantics tell the application what data it should set to each variable – for both attributes and uniforms. In order to share effects between multiple programs, including an effect editor, these semantics should use a ‘standard’ set of names. The semantics supported by PVRShaman are listed in the PVRShaman documentation.

If a semantic in a PFX file ends with a number, this number is stripped off and used as an index; for example if two semantics are LIGHTDIRECTION0 and LIGHTDIRECTION1, these are read as being the same semantic, LIGHTDIRECTION, but with index values of 0 and 1 respectively.

1.7. Issues/Simplifications

PFX is reduced and simplified compared to some other effect file systems.

1.7.1. Why no render states?

PFX files can contain information to control only the following:

- GLSL program
- Textures
- Initial texture-filter state
- Uniforms
- Attributes

This is because the above state must be set for every effect and can be set, overwriting previous values, when an effect is used. However, other states such as alpha blend, or Z-buffer control, would be optionally set for every effect. That would require the addition of a “Deactive()” call after the render of each object, and also the runtime would be bulked up by the necessity for optimised code to track which states have been set and minimise calls to OpenGL to switch to the new state.

As a goal of PFX is simplicity, this has been left to the application to handle at a higher level where more information is available and better optimisations can be applied (e.g. sorting by effect).

1.7.2. Why no passes?

With GLSL there is little need for multi-pass effects; also it is too costly an operation for embedded devices and it is thus being discouraged by having no direct support for it.

However, applications can use the ANNOTATION section of an EFFECT for any purpose the developer desires, including daisy-chaining effects into passes if so desired.

1.7.3. Why no higher-level render representation?

This level of complexity was deemed inappropriate for a simple, easy to use format and API.

2. PVRTPFX: PVRTools implementation of a PFX API

PVRTPFX is currently available for OpenGL (OGL) and OpenGL ES 2 (OGLES2).

An application loads and uses a PFX file as follows:

During initialisation (before or after initialising OpenGL):

1. Load and parse the PFX file using the class CPVRTPFXParser.

After initialising OpenGL:

2. Load a named effect from the loaded file using the class CPVRTPFXEffect.
3. Repeat; if there are multiple effects in the effect file, for however many are to be loaded, instantiate a CPVRTPFXEffect and load the desired effect from the CPVRTPFXParser.
4. Associate semantic strings with application-defined enum values:
5. Pass to PVRTPFX an array of the structure SPVRTPFXUniformSemantic; each item of which contains a string and a value (e.g. an enum). These values are application-defined, and each item of the array is a semantic that is known to the program.
6. PVRTPFX will return an array of the structure SPVRTPFXUniform; each element of which contains the semantic value, an index value (see section 1.6), and the OpenGL location. Also returned is the length of this array.
7. Load textures required for the effect and pass them to PFX

When rendering, assuming objects have been grouped by effect:

1. For each effect in use
2. Call CPVRTPFXEffect::Activate(). This sets and enables in OpenGL the GLSL program and the textures.
3. For each mesh using this effect
4. Loop over the SPVRTPFXUniform array
5. “switch” on the application-defined semantic value, setting the GL attributes and uniforms using the specified index value as appropriate (e.g. to select data from the Nth light, or the Nth texture coordinate array).
6. Render the mesh
7. If necessary loop over the SPVRTPFXUniform array again, disabling any GL vertex attrib arrays that were enabled.

2.1. IntroducingPFX Trainging Course

The example Training Course application “IntroducingPFX” is available in the POWERVR SDKs. It uses, as part of PVRTools, the PFX API implementation PVRTPFX. All of this is available in source code.

2.2. PVREngine and PODPlayer

An example of a renderer that uses PFX is presented in the OpenGL ES 2 SDK in the form of PODPlayer and the supporting PVREngine framework. This source code shows one way of implementing the steps required to use PFX files, as described above.

2.3. PVRShaman: PFX editor

PVRShaman is designed as a comprehensive editor of PFX files allowing the development of effects in a user-friendly visual environment with realtime feedback on changes made.

Some of the features of PVRShaman include:

1. PFX files can be edited and recompiled in real time.
2. Materials can be assigned to objects and their parameters adjusted using the GUI.
3. Builds and runs on Mac OS, Windows and Linux.
4. Scenes can be loaded and the animation replayed while editing the materials and effects.

Please examine the PVRShaman documentation which is included in this utility's package for further information.

Source code for PVRShaman is not currently available.