Introduction

This document is meant to describe the differences between making shaders for the pre-Universe versions of Electric Image and Electric Image Universe.

Electric Image Universe shaders use Macintosh-style resource files (on non-Mac platforms, the resources are placed into a MacBinary file). You must use a Macintosh to create these resource files. See the section “Resources” below for more information. Also, see the document “DLOG Converter Notes.doc” for a discussion of how to create a shader’s user interface.

Platform

You must define one of the platform symbols in order to properly compile. The valid platform symbols are __MACINTOSH__, __WIN32__ and __SUN__. One of these symbols must be defined “outside” of the source (e.g. in a prefix file, in a precompiled header, as a command line argument, etc.).

File Name Extension

Similar to plugins, shaders should now include an extension. The extension used by Electric Image Universe shaders is “.shd.” You should name your shaders with this extension even on the Macintosh. In fact, your shader should be named the same on every platform you ship on (if your shader uses different names on different platforms, it may not work correctly with Renderama).

Resources

The resources used by a shader can be divided into two groups: platform-specific resources, and platform-independent resources.

The platform-independent resources are those actually required for the shader’s code to work. These resources include the WINT resource you create with the DLOG Converter program and any custom resources (if any) you create and which your shader loads and uses at runtime.

Platform-specific resources are those that are required by a given operating system. On the Macintosh, these resources include ‘vers,’ ‘BNDL,’ and ‘icl8.’

Previously in Electric Image shaders, both kinds of resources were present in the same file: the shader file itself. However, with Electric Image Universe, these resources should be separated into two files. Platform-specific resources continue to be stored in the shader file, while platform-independent resources should be stored in a separate file. This new file should use the same name as the shader, but with a different extension. That extension is “.rsc.”

You create the .rsc file on the Macintosh, using whatever method you like. One way is to put a new build target into your shader's CodeWarrior project and add all platform-indendent resource files to that target. You use that target to build the shader's .rsc file.

Note that on Windows, the .rsc file is really a MacBinary version of the Macintosh .rsc file. One way to create the Windows version of the .rsc file is as follows:

- build the .rsc file on a Macintosh
- stuff the .rsc file on a Macintosh using DropStuff to generate a .rsc.sit file
- transfer the .rsc.sit file to a PC (make sure to use a binary transfer so no conversions occur to the file)
- use Stuffit Expander to expand the .rsc.sit into a MacBinary file

Make sure when expanding the .rsc.sit file that Stuffit Expander is set to always expand resource forks into separate MacBinary files.

When delivering your shader, the shader and the .rsc file should both be placed in the EI Shaders directory.

Previews

Preview images must be stored in resources of type 'PICT'. The 'EIPV' type is no longer supported. Also, 16 bit-per-pixel preview images are not supported; previews must be 32 bit or 8 bit PICTs.

User Interface

A shader's resources must be converted using DLOG Converter, which will automatically convert the shader's DLOG and DITL resources to the appropriate framework resources.

See the accompanying document describing DLOG Converter.

Endian Issues

The data passed to the shader from the host will always be in big-endian format. This means that on the PC, you must byte-swap the data before using it. You can use the FIXFIELD macro for this purpose. FIXFIELD is defined in System.h. FIXFIELD will byte-swap its argument on little endian machines and do nothing on big-endian machines.

Example:

```
typedef struct {
   unsigned long     shaderOwnerType;
   unsigned long     shaderVersionNum;
   unsigned long     one;
   unsigned short    two;
   long              three;
} ShaderInterface;

static void
FixShaderInterface(ShaderInterface *theShaderInterface)
{
   FIXFIELD(theShaderInterface->shaderOwnerType);
   FIXFIELD(theShaderInterface->shaderVersionNum);
   FIXFIELD(theShaderInterface->one);
   FIXFIELD(theShaderInterface->two);
   FIXFIELD(theShaderInterface->three);
}
```

NOTE: Do not use the macro FixField (mixed case); use FIXFIELD (all caps).

NOTE: Do not use FIXFIELD on values of type ARGB. Color values will already be in the correct byte order when your shader is called.

Previously, the sample shaders contained code to get the shader's interface data that looked like this:

```
static void GetShaderInterface(void* theInterface,
               long theSize,
               ShaderInterfacePtr theShaderInterface)
{
   /* Copy the shader interface data */

   if (theSize == sizeof(ShaderInterfaceRec))
      *theShaderInterface =
            *(ShaderInterfacePtr) theInterface;
   else {
      theShaderInterface->shaderOwnerType = 0;
      theShaderInterface->shaderVersionNum = 0;
   }

   /* Validate the shader interface data */

   if (theShaderInterface->shaderOwnerType != shaderType)
      DefaultShaderInterface(theShaderInterface);
   else if (theShaderInterface->shaderVersionNum ==
               shaderRevVersion)
      FixShaderInterface(theShaderInterface);
   else if (theShaderInterface->shaderVersionNum !=
               shaderVersion)
      DefaultShaderInterface(theShaderInterface);
}  /* GetShaderInterface */
```

However, this code is no longer recommended because it will not work correctly. The reason is that since the data passed to the shader is always in a

big endian format, the test if (theShaderInterface->shaderOwnerType != shaderType) (which appears just after the comment “Validate the shader interface data”) will never be true, and the default data will be used instead.

Since the data is big-endian, it must be fixed before using it. The recommended way to write this function is

```
static void GetShaderInterface(void* theInterface,
               long theSize,
               ShaderInterfacePtr theShaderInterface)
{
   /* Copy the shader interface data */

   if (theSize == sizeof(ShaderInterfaceRec))
      *theShaderInterface =
            *(ShaderInterfacePtr) theInterface;
   else {
      theShaderInterface->shaderOwnerType = 0;
      theShaderInterface->shaderVersionNum = 0;
   }

   /* Validate the shader interface data */

   FixShaderInterface(theShaderInterface);
   if (theShaderInterface->shaderOwnerType != shaderType ||
            theShaderInterface->shaderVersionNum !=
            shaderVersion)
      DefaultShaderInterface(theShaderInterface);
}  /* GetShaderInterface */
```

Here we fix the shader interface data first by calling FixShaderInterface, which will apply FIXFIELD to each member of the shader interface structure. After that point, it is valid to test the values of the interface structure to see if it should be reverted to the default values.