Electric Image Animation System

EIAS Version 9.0
Shaders API

Table of Contents

Copyright 1998-2011, EIAS3D

1.0 Introduction

What exactly is a shader? Well, in its simplest form, it's a way of creating a texture map for a piece of geometry. If you've explored the shader library included with ElectricImage, you've seen everything from Fractal Noise to Parquet Flooring.

Some of you out there may be asking why a shader is so cool. Couldn't we just use a photograph of a brick wall rather than the Brick shader? Well, yes you could, but in order to get the same level of detail in a traditional texture map, you would have to scan it at very high resolutions. This results in a large file that has to be read from disk. So speed is one benefit. Detail is another. Since the algorithms for shaders typically have unlimited detail, they rapidly outdistance 2D texture maps.

Another major benefit of shaders is that they can be defined in three dimensions. When you apply a flat texture map to an object, the texture "bleeds" all the way through the object. Three-dimensional shaders can create different data in the third dimension. Fractal Noise is an example of this. The shading on the back side of a cube is different from that of the front side. Note: not all shaders are three dimensional.

Shaders can do some very complex magic. Since the shaders have access to virtually every material and surface variable, you can create shaders that can't be duplicated by traditional texture mapping. A shader that mimics a brushed aluminum surface with circular polishing can be created including circular highlights that track light sources.

So what's the down side? Writing shaders can make your brain hurt because you really have to visualize the end results in your mind and then break them down for coding. But probably the most difficult aspect of writing shaders is anti-aliasing. You will probably spend most of your time dealing with anti-aliasing issues. For shaders, anti-aliasing refers not just to resolving jaggies as we are used to but to how the shader is sampled. More on this later.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

2.0 Basic Shader Structure

So, on to the specifics of the ElectricImage Shader API. There are four functions that are required by the host (ElectricImage or Camera). The first is

SInt32 EIShaderInformation(EIShaderInformationRec *theInformationPtr)

This function has three purposes. First, the variable values are set up. Second, the shader specifies which material and surface attributes it needs access to. Third, the shader specifies which material and surface variables it is generating. The following code is from the Eroded shader.

```
/* Get the shader interface parameters */

GetShaderInterface(theInformationPtr->shaderInterface,
                        theInformationPtr->shaderInterfaceSize,
                        &theShaderInterface);
```

This function call sets up the shader variables. They are either initialized, read from the project file animation channels, or byte-swapped for use with other processor types such as Intel processors.
It's important to know that the shader variables defined in the source code match those in the resource template or you won't be able to access the variables from within ElectricImage. More on this later.

```
/* Setup the shader attribute access flags */

if (BTST(theShaderInterface.shaderErodeHoles, 31))
       BSET(theInformationPtr->shaderFeatureFlags, shaderClipFlag);

/* Setup the shader feature flags */

BSET(theInformationPtr->shaderFeatureFlags, shaderAntialiasFlag);
BSET(theInformationPtr->shaderFeatureFlags, shaderColorFlag);
SETFLAG(theInformationPtr->shaderFeatureFlags, shaderBumpFlag,
                        (theShaderInterface.shaderBumpStrength != 0.0));
```

The above code tells the host what shading attributes the shader generates. Here, we are generating a clip map only if the user has turned on the Erode Holes checkbox. We are also performing anti-aliasing, setting the surface color and generating a bump map.

If we wanted to build a shader that makes use of some surface attribute such as specular highlighting, you would set up access flags in the Information phase as well. For example, the Granite shader generates different surface shading when rendering areas of specular highlights.

Copyright 1998-2011, EIAS3D

The Second is

```
SInt32 EIShaderInitialize(EIShaderInitializeRec *theInitializePtr)
```

This is where you perform memory allocation. You also initialize noise libraries and transfer shader interface variables into the shader's data structures. Have a look at the initialize code from the Eroded shader.

```
ShaderDataPtr theShaderData;

/* Initialize the noise library */

if (!InitNoise((NoiseMallocFunction)theInitializePtr->HostMalloc, NULL))
        return shaderMemoryError;

/* Allocate the shader's private data record */

theShaderData = (ShaderDataPtr)theInitializePtr->HostMalloc(sizeof(ShaderDataRec));
if (theShaderData) {
ShaderInterfaceRec theShaderInterface;

/* Get the shader interface parameters */

GetShaderInterface(theInitializePtr->shaderInterface,
theInitializePtr->shaderInterfaceSize, &theShaderInterface);

        /* Setup the shader's private data */

theShaderData->shaderDensity = theShaderInterface.shaderDensity;
theShaderData->shaderBumpStrength = theShaderInterface.shaderBumpStrength;
theShaderData->shaderErodeHoles = BTST(theShaderInterface.shaderErodeHoles, 31);
theShaderData->shaderHoleMin = theShaderInterface.shaderHoleMin;
theShaderData->shaderHoleMax = theShaderInterface.shaderHoleMax;
theShaderData->shaderHoleColor.a = theShaderInterface.shaderHoleColor.argb.a / 255.0;
theShaderData->shaderHoleColor.r = theShaderInterface.shaderHoleColor.argb.r / 255.0;
theShaderData->shaderHoleColor.g = theShaderInterface.shaderHoleColor.argb.g / 255.0;
theShaderData->shaderHoleColor.b = theShaderInterface.shaderHoleColor.argb.b / 255.0;
theShaderData->shaderSurfaceColor.a =
                                        theShaderInterface.shaderSurfaceColor.argb.a / 255.0;
theShaderData->shaderSurfaceColor.r =
                                        theShaderInterface.shaderSurfaceColor.argb.r / 255.0;
theShaderData->shaderSurfaceColor.g =
                                        theShaderInterface.shaderSurfaceColor.argb.g / 255.0;
theShaderData->shaderSurfaceColor.b =
                                        theShaderInterface.shaderSurfaceColor.argb.b / 255.0;

        /* Return the pointer to the shader's private data to the host */

        theInitializePtr->shaderData = theShaderData;
        return shaderNoError;
} else
        return shaderMemoryError;
```

First off, this shader makes use of the Perlin Noise library so we need to allocate some memory for it and initialize the noise space. Second, we need to allocate memory for the shader's main data structure. If both of these memory allocation calls are successful, we then transfer the shader control variables from the host and copy them into the shader's main data structure. We'll discuss how the main data structure is defined later.

Copyright 1998-2011, EIAS3D

The third major shader function is

```
SInt32 EIShaderFinish(EIShaderFinishRec *theFinishPtr)
```

This is where the shader deallocates any memory that is reserved in the Initialize phase. All shaders need to deallocate the memory used by the main data structure. In the case of the Eroded shader, we also need to shut down the Noise library.

```
/* Get rid of the shader's private data */

theFinishPtr->HostFree(theFinishPtr->shaderData);

/* Finish the noise library */

FinishNoise((NoiseFreeFunction)theFinishPtr->HostFree);
return shaderNoError;
```

Finally, the most important function call is

```
SInt32 EIShaderShade(EIShaderShadeRec *theShadePtr)
```

This is where all the magic happens. The Eroded shader does the following:

```
ShaderDataPtr theShaderData = (ShaderDataPtr)theShadePtr->shaderData;
float factor = CalcEroded(theShadePtr);

/* Calculate the shader transparency */

if (theShaderData->shaderErodeHoles &&
BTST(theShadePtr->shaderFeatureFlags, shaderClipFlag))
        theShadePtr->shaderClip = factor;

/* Calculate the shader color */

if (BTST(theShadePtr->shaderFeatureFlags, shaderColorFlag))
        MixARGBColors(&theShaderData->shaderHoleColor,
                        &theShaderData->shaderSurfaceColor,
                        &theShadePtr->shaderColor, factor);

return shaderNoError;
```

We'll discuss the details of this magical function later.

Copyright 1998-2011, EIAS3D

3.0 Memory Allocation

Shader Memory allocation is similar to the standard C malloc function.. The call to allocate memory for the shader's main data structure is as follows:

```
theShaderData =
                (ShaderDataPtr)theInitializePtr->HostMalloc(sizeof(ShaderDataRec));
```

This looks a bit odd but the way it works is the memory allocation procedure pointer is part of the initialization data structure. Just pass the number of bytes you need and the pointer to that memory is returned.

Deallocation of this memory should be done only in the Finish phase. Again, the procedure point to the Free function comes along with the call from the host to your Finish routine.

```
theFinishPtr->HostFree(theFinishPtr->shaderData);
```

Allocation of memory for Noise libraries is somewhat different.

```
if (!InitNoise((NoiseMallocFunction)theInitializePtr->HostMalloc, NULL))
        return shaderMemoryError;
```

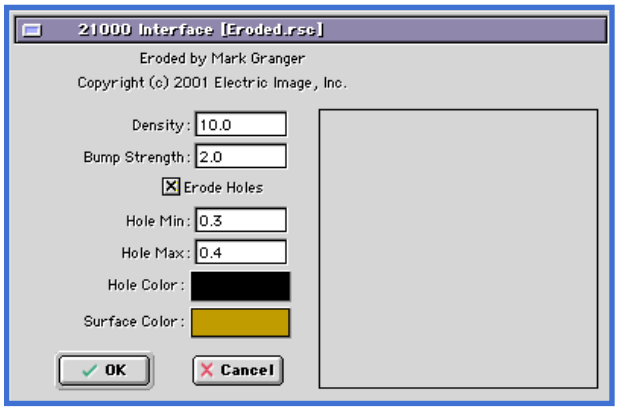
While the procedure pointer to the malloc function is still used, it must be typecast to NoiseMallocFunction.

You should check any pointers you allocate to see if they are NULL. If so, then the memory allocation was not successful and you should return a shaderMemoryError.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

4.0 Shader Interfaces



Another nice feature of the ElectricImage Shader API is that the author does not have to spend enormous amounts of time designing a user-interface. Here are the steps to create or edit a shader's UI interactively via the EI Interface Builder utility:

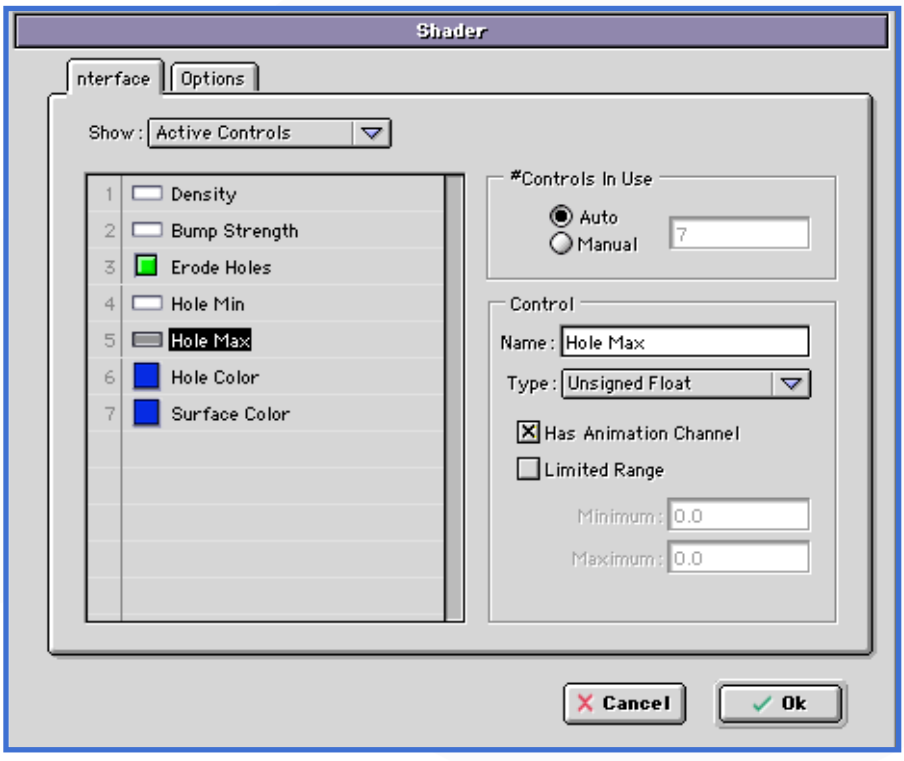
- Run the EI Interface Builder and open an existing .rsc file or create a new one.

- Open or create the shader's UI dialog with WINT ID = 21000.

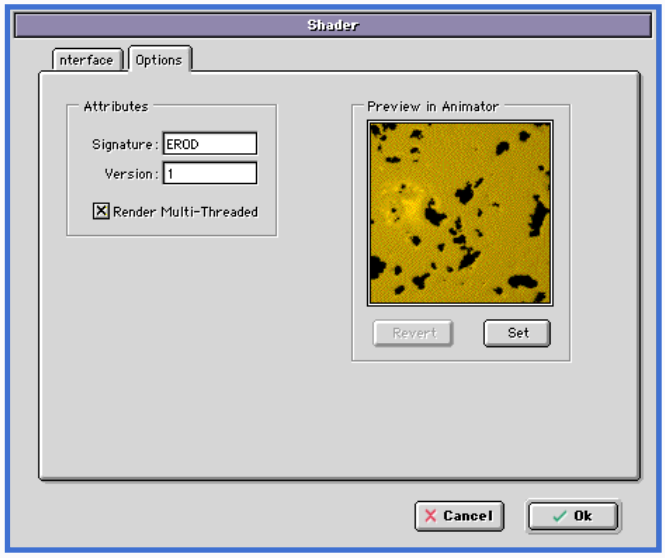
Have a look at the WINT resource for the Eroded shader in the EI Interface Builder. There are three required items for shader interfaces: OK button, Cancel button and a user item (invisible Group) that is used for the shader preview. The programmer does not have to write any special code to generate the preview.

- Edit the existing items, create or delete new edit boxes, check boxes, popups or color buttons. Note that values you enter in the controls will be used by Animator as default values.

- When you have finished editing, select "Shader.." from the Utility menu. Before opening the Shader dialog, the EI Interface Builder can ask you to create an OK button, a Cancel button, or a Preview Group if they are missing and/or perform "Recompute IDs" if new items were added or old ones deleted.



Copyright 1998-2011, EIAS3D



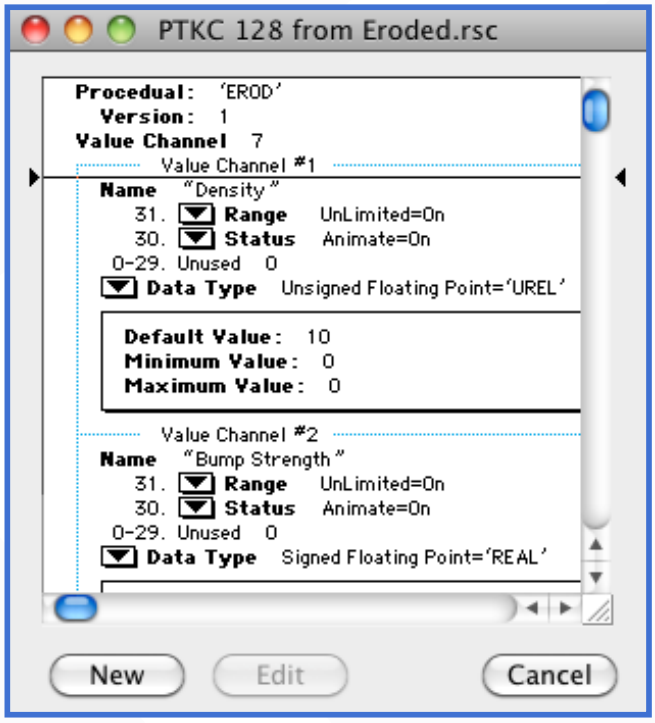
The list contains all controls that can be used by Animator. With the choice “Auto” (“#Controls in Use” panel), all controls are mapped into your ShaderData record and automatically displayed in Animator. If you plan on creating an advanced shader with a custom UI then you can limit the number of mapped controls to handle others manually via your own code.

- Edit the items’ attributes used by Animator: Name, Type, Animation Channel and Limited Range flag. Set the desired items order by dragging them in the list.

- specify the shader’s additional options in the “Options” Tab.

- Press Ok and save your changes.

4.1 What to Store in Resource Files

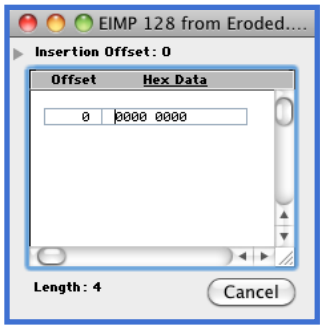


Note: this section describes the low-level format of resources. In addition, it covers how you can create and edit all resources (except WINT) via Resourcerer or another native resource editor on the Macintosh. However direct resource editing is longer and requires more effort than using the EI Interface Builder utility. It’s described here for backward compatibility only. Normally you create and edit all resources via EI Interface Builder.

1) Resource type ‘WINT’, ID = 21000. This is the shader’s UI dialog shown in Animator. This resource should be created via the EI Interface Builder application on the Macintosh.

2) Resource type ‘PTKC’, ID = 128. This holds information about the shader’s parameters as described below. This resource should be created via the Resorcerer application on the Macintosh. To edit ‘PTKC’ you need an additional resource ‘TMPL’ used by Resorcerer. You can grab it from the EI Shader SDK resource files.

Copyright 1998-2011, EIAS3D



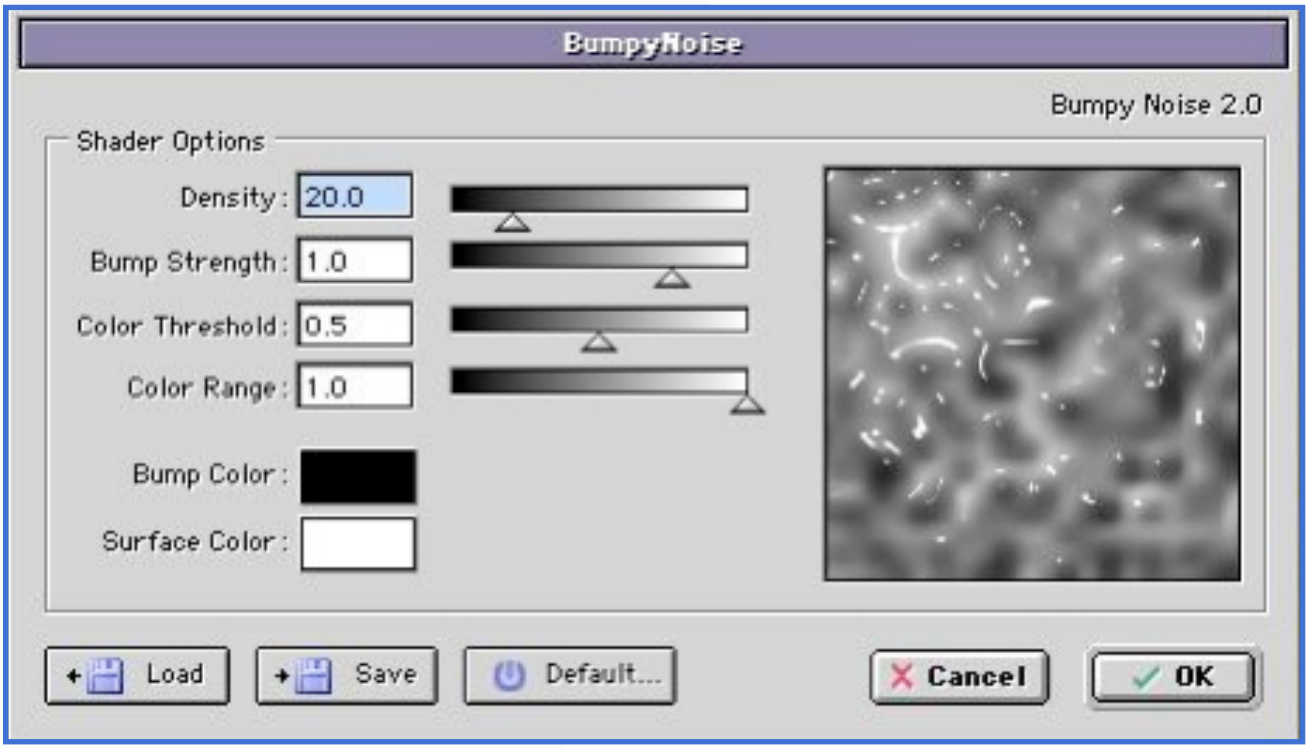
3) Resource type 'PICT', ID = 21000. This is the picture used by Animator as the shader's thumbnail.

4) Resource 'EIMP', ID =128 is used by Camera to detect how the shader should be used at render time: as multi-threaded (supports being called in parallel) or not. The resource is 4 bytes long, a zero value means multithreading is enabled, any other means that multi-threading is not supported by the shader.

The shader's resource file can optionally contain platform-dependent resources.

Copyright 1998-2011, EIAS3D

5.0 Advanced Shader Interfaces



Since EI 5.5, shaders can have a custom UI similar to EI plug-ins. The custom UI can be combined with the older PTKC-style interface in an arbitrary way. For example a shader can handle only few controls and leave most of the others for the host.

If a shader wants a custom UI, it must define the function

```
SInt32 EIShaderInterfaceSetup( EIShaderInterfaceSetupRec * theInterfaceSetup );
```

This function should have ‘export’ attribute, same as EIShaderShade, EIShaderInformation and others routines called by host. The EIShaderInterfaceSetupRec is defined as:

```
typedef struct {
	void * hostDialogReference;                  /* The host’s shader dialog reference */
	SInt32 hostResourceFile;                     /* The host’s shader resource file */
	EIHostInterfaceSetupProc HostInterfaceSetup; /* Interface setup callback */
} EIShaderInterfaceSetupRec
```


Copyright 1998-2011, EIAS3D

When EIShaderInterfaceSetup is called, a shader should copy the EIShaderInterfaceSetupRec content into its internal variabes.

```
/* Shader's variables */

EI_Dialog * shaderDialog;
EI_Resource shaderResource;
EIHostInterfaceSetupProc shaderSetupProc;


/* Save theInterfaceSetup  values */

shaderDialog = (EI_Dialog *) theInterfaceSetup->hostDialogReference;
shaderResource.data = theInterfaceSetup->hostResourceFile;
shaderSetupProc = theInterfaceSetup->HostInterfaceSetup;
```

As a result, the shader received the handles of the UI dialog and the shader's resource file. From here the shader can use all the EI UI API functions and install UI callbacks for controls and the main dialog, load alerts, and other dialogs from resources and so on. The shader should not destroy handles to dialog and resources it receives. The host does this automatically.

Using HostInterfaceSetup callback, shaders can simulate input for PTKC controls. For example:

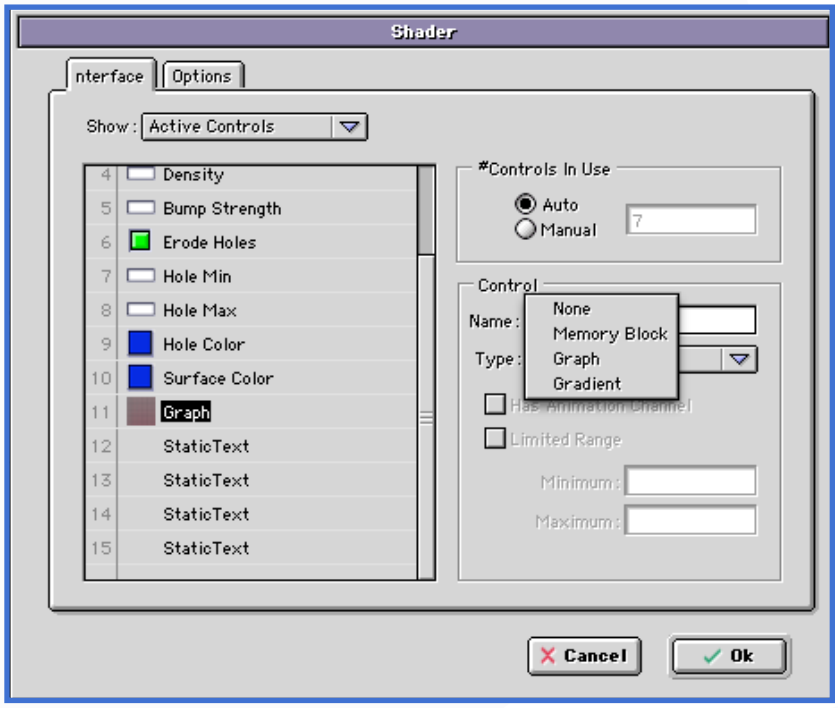
```
EI_Control * control = EI_FindControlByID(shaderDialog, MY_EDIT_ID);
EI_SetEditTextString(control, "0.0");
shaderSetupProc(shaderDialog, hostSetShaderItem, MY_EDIT_ID, 0);
```

This has the same effect as if a user typed "0.0" in the edit box with MY_EDIT_ID. The shader's preview will be updated by the host because MY_EDIT_ID is in the list of PTKC items.

Copyright 1998-2011, EIAS3D

6.0 Custom Shader's Data

Since EI 7.0, shaders can have Custom items, including Memory Block, Graph, and Gradient. The corresponding UI item should be created as a Group EI control.



In the shader's data structure all custom items should be defined as void * or any pointer. This field should not be converted between little/big endian. A shader should not attempt to release or replace this pointer.

```
typedef struct {
        UInt32 shaderOwnerType;
        UInt32 shaderVersionNum;
        void * shaderMemoryBlock;
} ShaderInterfaceRec, *ShaderInterfacePtr;
```


Copyright 1998-2011, EIAS3D

Electric Image Animation System

6.1 Custom Shaders - Using the Memory Block

For custom items with a Memory Block type, the item's value is a pointer to a EI_MemoryBlock structure

```
typedef struct {
        UInt32  blockSize;                      /* Data size in bytes */
        void *  blockData;                      /* Data pointer */
        UInt32  blockSign;                      /* Data signature, should be 'MEMO' */
} EI_MemoryBlock, EI_MemoryBlockPtr;
```

The shader should examine and read this structure's content. The data pointed to by blockData should not be modified. When a shader needs to update data, it should call the EIHostInterfaceSetupProc callback received from EIShaderInterfaceSetup. A new data block should be passed as the argument. Example:

```
        /* Fill EI_MemoryBlock */

        EI_MemoryBlock theBlock;
        theBlock.blockSize = (UInt32) myDataSize;
        theBlock.blockData = myDataPtr;
        theBlock.blockSign = 'MEMO';

        /* tell host to update */

        shaderSetupProc(shaderDialog, hostSetShaderItem, MY_MEMO_ID, &theBlock);
```

The host copies the data pointed to by blockData, thus the original data can be released after the update call.

6.2 Custom Shaders – Using the Graph and Gradient Controls

For these custom controls the pointer is EIGraphData or EIGradientData opaque pointers respectively.

At render time there is no UI thus shaders can get UI data by using EI_GraphDataRequest (EI_GradientDataRequest). See header files EI_GraphData.h (EI_GradientData.h).

During preview, user input is automatically stored. If a shader needs more advanced actions, for example reset graph, change interpolation style, etc., then the shader can obtain the control's handle and use the UI functions defined in EI_GraphControl.h (EI_GradientControl.h)

Copyright 1998-2011, EIAS3D

7.0 Shader Data Structures

There are two data structures that you need to define to store the shader variables. In the case of the Eroded shader, they look like this.

```
typedef struct {
        UInt32 shaderOwnerType;
        UInt32 shaderVersionNum;
        float shaderDensity;
        float shaderBumpStrength;
        SInt32 shaderErodeHoles;
        float shaderHoleMin;
        float shaderHoleMax;
        ARGB shaderHoleColor;
        ARGB shaderSurfaceColor;
} ShaderInterfaceRec, *ShaderInterfacePtr;

typedef struct {
        float shaderDensity;
        float shaderBumpStrength;
        SInt32 shaderErodeHoles;
        float shaderHoleMin;
        float shaderHoleMax;
        ARGBReal shaderHoleColor;
        ARGBReal shaderSurfaceColor;
} ShaderDataRec, *ShaderDataPtr;
```

The first is used to transfer data to and from the user-interface. The first two fields are required. ShaderOwnerType is a four-character code that identifies and distinguishes one shader from another. ShaderVersionNum is the version of the shader. This allows you to write new versions of a shader and be able to add data fields not present in previous versions without destroying any pre-existing animation data.

Usually, variables in the ShaderInterfaceRec will correspond directly to those in the ShaderDataRec. However, in the case of color variables, Camera uses floating-point representations whereas EI animation channels are integer-based. Therefore, they must be converted when the data is transferred from the interface to the guts of the shader.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

8.0 About the Shader Shade Record

Most shaders will make use of the EIShaderShadeRec structure. The shader is given access to this during the EIShaderShade call.

```
typedef struct {
        void *shaderData;                               /* Pointer to the shader's private data */
        EIHostLightProc HostLight;                      /* The lightsource access callback */
        EIShaderPixelRec *shaderPixel;                  /* Pixel perameters */
        EIShaderAttributesRec *shaderAttributes;        /* Shading attributes */
        EIShaderMapRec *shaderMap;                      /* Mapping parameters */
        UInt32 shaderAccessFlags;               /* Flags specifying which host parameters the shader may write */
        UInt32 shaderFeatureFlags;                      /* Flags specifying which features the shader supports */
        float shaderDX, shaderDY, shaderDZ;             /* The displacement vector */
        float shaderBumpA, shaderBumpB, shaderBumpC;    /* The bump perturbation vector */
        float shaderClip;                               /* The shader clip factor */
        RGB_Real shaderTransparency;                    /* The shader transparency color */
        RGB_Real shaderReflection;                      /* The shader reflection color */
        ARGB_Real shaderColor;                          /* The shader color and opacity */

        /* Version 2.0 */

        EIHostCastRayProc HostCastRay;                  /* The ray trace callback */
        RGB_Real shaderHighlight;                       /* The shader highlight color */
        RGB_Real shaderRefraction;                      /* The shader refraction color */

        /* Version EI 7.0 */

        EILayerShadeProc  HostLayerShade;               /* The update layer(s) callback */

        /* Version EI 8.0 */

        EIModelShadeProc HostModelShade;                /* The model data access callback */

} EIShaderShadeRec, *EIShaderShadePtr;
```

Through this data structure, the shader can access almost anything. The first parameter, `*shaderData`, is a pointer to the shader's programmer-defined variables. Next, `HostLight`, is a procedure pointer for accessing light sources. Third, `*shaderPixel` , is a very large data structure that describes specifiy rendered pixel attributes. Following this, `*shaderAttributes`, is the structure for accessing and modifying material attributes. Next, `*shaderMap`, are the animator-defined map parameters.

Copyright 1998-2011, EIAS3D

shaderAccessFlags is used to specify which parameters the shader needs to be able to read and write. Access to a particular attribute is true or false and they are encoded into a single 32-bit variable. The bit numbers are defined below.

```
/*  shaderAccessFlags  --  Shader attribute value access flags  */

#define  accessReferenceFlag                        0
#define  accessAmbientFlag                          1
#define  accessSpecularFlag                         2
#define  accessTransparencyFlag                     3
#define  accessReflectionFlag                       4
#define  accessLuminanceFlag                        5
#define  accessDiffuseFlag                          6
#define  accessTransmissionFlag                     7
#define  accessEdgeTransparencyFlag                 8
#define  accessEdgeTransparencyValueFlag            9
#define  accessEdgeDiffuseValueFlag                 10
#define  accessEdgeSpecularValueFlag                11
#define  accessEdgeReflectionValueFlag              12
#define  accessEdgeAmbientValueFlag                 13
#define  accessEdgeLuminanceValueFlag               14
#define  accessEdgeTransmissionValueFlag            15
#define  accessHighlightFlag                        16
#define  accessRefractionFlag                       17
#define  accessTerminatorFlag                       18
#define  accessGlowFlag                             19
#define  accessBloomFlag                            20
#define  accessAlphaFlag                            21
#define  accessOpacityFlag                          22
#define  accessGlossFlag                            23
#define  accessShadeFlag                            24
#define  accessEdgeTransparencyDropoffFlag          25
#define  accessEdgeDiffuseDropoffFlag               26
#define  accessEdgeSpecularDropoffFlag              27
#define  accessEdgeReflectionDropoffFlag            28
#define  accessEdgeAmbientDropoffFlag               29
#define  accessEdgeLuminanceDropoffFlag             30
#define  accessEdgeTransmissionDropoffFlag          31
```

Similarly, the shader needs to specify what its purpose in life is. This is done through the shaderFeatureFlags bitfield parameter. There are three types of flags encoded into this parameter. The Mode flags specify if the shader calculates specific color channels and/or if it is an “illumination” or “lightsource” shader. The Checker-Board shader sets the shaderColorFlag. That means that the shader generates the surface color and overrides any other surface colors. The Granite shader doesn’t set this flag. Instead, it uses surface color, specular, and reflection values.

Copyright 1998-2011, EIAS3D

A lightsource shader is a special purpose shader that allows you to do things light a circular brushed aluminum surface where the specular highlight discs interact with lightsources in the scene.

```
/*  shaderFeatureFlags  --  mode  flags  --  if  none  are  set,  the  shader  generates  a  surface
material  */

#define  shaderIlluminationFlag          0       /*  lightsource  shading  is  performed  by the shader  */
#define  shaderColorFlag                 1       /*  shader  is  designed  to  calculate  a  specific  channel  color  */
```

The Write flags pertain to the more physical attributes of a surface that that the shader can generate.

The Eroded shader generates Clip values and Bump values in addition to color values.

```
/*  shaderFeatureFlags  --  write  flags  */
/*  The  shader  sets  these  flags  to  indicate  which  values  are  written  by  the  shader  */

#define  shaderClipFlag                12      /*  shader  writes  shaderClip  */
#define  shaderTransparencyFlag        13      /*  shader  writes  shaderTransparency  */
#define  shaderDisplacementFlag        14      /*  shader  writes  shaderDisplacement  */
#define  shaderReflectionFlag          15      /*  shader  writes  shaderReflection  */
#define  shaderBumpFlag                16      /*  shader  writes  shaderBump(ABC)  */
```

When a shader generates Clip, Transparency, Displacement, Reflection, or Bump information, the generated values are returned to the host in the associated fields of the EIShaderShadeRec.

The Read flags allow the shader to access mostly anti-aliasing information.

```
/*  shaderFeatureFlags  --  read  flags  */
/*  The  shader  sets  these  flags  to  indicate  which  values  are  read  by  the  shader  */

#define  shaderPositionVectorsFlag           28      /*  shader  reads  shadeD(XYZ)D(UV)  */
#define  shaderAntialiasPositionFlag         29      /*  shader  reads  shade(XYZ)(12)  */
#define  shaderAntialiasNormalFlag           30      /*  shader  reads  shadeNormal(ABC)(12),
                                                         shaderNormal(XYZ)(12)  */
#define  shaderAntialiasFlag                 31      /*  shader  reads  map_(XYZ)(12), map_D(XYZ) */
```

Shaders don't automatically get access to everything. They can only access parameters that are specifically requested. This keeps the volume of data being transferred to a minumum so that Camera runs as fast as possible.

Copyright 1998-2011, EIAS3D

NOTE: A shader should examine the anti-alias flags listed above (bits 28..31) for any shaded point. Do not assume a flag should be ON because it was set by the shader via EIShaderInformation. It is true only for “primary” (screen) points. In the case of ray-trace reflections, refractions and others, the anti-alias flags are always OFF.

9.0 Detecting a Shader’s Host

Some shaders do create advanced effects at render time and cannot provide an effective preview. To detect when a shader is being called by Camera, the shader must define an exported EIShaderHostInfo function. This function is called by the host and ID of the host is returned. Example:

```
SInt32 EIShaderHostInfo( EIShaderHostInfoRec * theShaderHostInfo )
{
      /* Save host ID in shader’s variable */

      theHostID = theShaderHostInfo->hostID;
      return shaderNoError;
}

/* Check host */

if (theHostID == QUADWORD(‘EIAR’)) {
  /* Camera is running, do things for render */
}
else {
 /* The shader runs in Animator */
}
```

It’s important to note that when the shader is being called by Animator it does not only mean “preview in Animator”. In Animator, shaders can be called for different purposes, for example, Shader Variance palette drawing. The shader must handle all calls by Animator correctly.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

10.0 Determining Render Context

Since EI 6.0, a shader can determine what it was called for in the render pipeline by examining bits 4..7 of shaderFeatureFlags.

```
#define shaderModeFlag0              4
#define shaderModeFlag1              5
#define shaderModeFlag2              6
#define shaderModeFlag3              7

The values are:

#define shaderShadeNormal            0      /* the screen point is shaded */
#define shaderShadeZBuffer           1      /* the Z-Buffer point is shaded */
#define shaderShadeGlow              2      /* the glow point is shaded */
#define shaderShadeReflectionRT      3      /* the RT reflection point is shaded */
#define shaderShadeRefractionRT      4      /* the RT refraction point is shaded */
#define shaderShadeShadowRT          5      /* the RT shadow point is shaded */
#define shaderShadeHostCastRay       6      /* the point hit by HostCastRay is shaded */
#define shaderShadeBufferGI          7      /* the GI buffer point is shaded */
#define shaderShadeRayGI             8      /* the GI reverse illumination point is shaded */
#define shaderShadeSky               9      /* the sky (background/foreground) point is shaded */
#define shaderShadeReflectionSky     10     /*  the  sky  reflection  (background/foreground)  point  is
shaded */
#define shaderShadeReflectionSkyRT   11     /* the RT reflection sky point is shaded */
#define shaderShadeRefractionSkyRT   12     /* the RT refraction sky point is shaded */
#define shaderShadeUnknown           13     /* the shaded point is unknown */
```

Based on this information, shaders can optimize their calculations and/or create special effects for ray-trace reflections, refractions, shadows and so on. Example:

```
SInt32 shadeMode = (theShaderShade->shaderFeatureFlags >> 4) & 0xF;

switch (shadeMode) {
        case shaderShadeNormal:
              /* Do a full processing */
        break;

        case shaderShadeReflectionRT:
        case shaderShadeRefractionRT:
              /* Do a simplified processing */
        break;
}
```


Copyright 1998-2011, EIAS3D

11.0 About Shader Access Attributes

Shaders have the ability to use all of an object's material settings. This is especially useful for cases where the shader is responsble for generating things like specular highlights. An example of this would be a window pane shaped specular reflection as seen in cartoons or Pixar's Tin Toy. The shader needs to set the size the window pattern based upon the Specular Size parameter. It may also need to soften the edges based on other parameters.

Here is the Shader Attributes structure:

```
/* Shading attributes record */

typedef struct {
        RGBReal reference;                  /* surface color */
        RGBReal ambient;                    /* ambient reflection color */
        RGBReal specular;                   /* specular reflection color */
        RGBReal transparency;               /* transparency components */
        RGBReal reflection;                 /* reflection color factor for material */
        RGBReal edgeTransparency;           /* edge reflection color (for backward compatibility) */
        RGBReal luminance;                  /* self illumination components */
        RGBReal diffuse;                    /* diffuse reflection color */
        RGBReal transmission;               /* The diffuse transmission color */
        RGBReal glow;                       /* The glow color */
        float alpha;                        /* surface alpha mask (1 = solid, 0 = invisible) */
        float opacity;                      /* opacity transparency (1 = solid,0 = transparent) */
        float edgeOpacity;                  /* edge opacity (for backward compatibility) */
        float gloss;                        /* gloss factor (1 = no gloss, 0 = 100% gloss) */
        float shade;                        /* shade factor (1 = no shading, 0 = 100% shading) */
        float bloom;                        /* bloom factor (1 = full bloom, 0 = no bloom) */
        float spread;                       /* specular highlight spread */
        float refraction;                   /* transparency refraction index */
        float terminator;                   /* edge terminator factor */
        float highlight;                    /* highlight dropoff factor */
        float edgeTransparencyValue;        /* the edge transparency value */
        float edgeDiffuseValue;             /* The edge diffuse value */
        float edgeSpecularValue;            /* The edge specular value */
        float edgeReflectionValue;          /* The edge reflection value */
        float edgeAmbientValue;             /* The edge ambient value */
        float edgeLuminanceValue;           /* The edge luminance value */
        float edgeTransmissionValue;        /* The edge transmission value */
        float edgeTransparencyDropoff;      /* the degree to which the transparency changes
                                               from the center to the edge (0.0=none) */
        float edgeDiffuseDropoff;           /* the degree to which the diffuse changes
                                               from the center to the edge (0.0=none) */
        float edgeSpecularDropoff;          /* the degree to which the specular changes from
                                               the center to the edge (0.0=none) */
        float edgeReflectionDropoff;        /* the degree to which the reflection changes
                                               from the center to the edge (0.0=none) */
        float edgeAmbientDropoff;           /* the degree to which the ambient changes from
                                               the center to the edge (0.0=none) */
        float edgeLuminanceDropoff;         /* the degree to which the luminance changes
                                               from the center to the edge (0.0=none) */
        float edgeTransmissionDropoff;      /* the degree to which the transmission changes
                                               from the center to the edge (0.0=none) */
        UInt32 shadeFlags;                  /* see "GATR Group Shading Flags" in FACTStuff.h
                                               */
        UInt32 shadeFlags2;                 /* see "GATR Group Shading Flags2"in FACTStuff.h */
} EIShaderAttributesRec, *EIShaderAttributesPtr;
```

Copyright 1998-2011, EIAS3D

All of these values correspond directly to surface settings found in the ElectricImage Material Editor. It is important to know that these values vary over the surface being rendered. In the example of specular highlighting, the highlight doesn't exist everywhere. It also changes with the object's relationship to the surrounding lights. These value vary during the course of rendering and animation. Earlier shaders and/or textures will modify these values.

Copyright 1998-2011, EIAS3D

12.0 About Shader Pixel Variables

The EIShaderPixelRec contains a great deal of information that is specific to the pixels being rendered. Most of this information you will never use. The values that are used most often are

```
float shadeNormalA0, shadeNormalB0, shadeNormalC0;    // N at P' before any bumps
                                                          have been applied
```

This triple represents the surface normal at the current rendering pixel before any bump information is applied. This is important because you may want to write a shader that creates its own bumps but you do not want your bump calculations affected by other shaders or textures that generate their own bumps. An example of this would be a moon crater shader. Here, we want to generate a bump map that represents a crater. The shader needs to disturb the surface in a direction that is perpendicular to it. So, we need the true surface normal. But, bump mapping is a process that modifies surface normals. These values solve this problem.

Other useful value are things like shadeTime. The Wave shader is an animated effect. You use the shadeTime value to generate the correct wave look. You can also get information about the camera so you could write your own Camera Map shader.

You may notice some redundancy in some of the shader data structures. For example, there are transparency values in both the EIShaderShadeRec and the EIShaderAttributesRec. There are three flavors of surface color, reflection color and transparency color. The first is in the EIShaderAttributesRec. This is used by Camera as a color filter during pixel shading. The second flavor is in the EIShaderPixelRec. This value is maintained by Camera and is built up over calls to all of the shaders and textures. The final flavor is in the EIShaderShadeRec. This is set by the shader at each pixel. Camera is responsible for mixing it into the value in the EIShaderPixelRec.

The pixel transparency value is replaced by the shader transparency value by Camera (it mixes them together with the shader's opacity setting). The transparency controls how much light passes through the surface from whatever is behind the surface. This works in conjunction with the clip value and is used to make colored filters and additive-style transparency. If the transparency is (0, 0, 0) the surface is not transparent. If the transparency is (1, 1, 1), the surface is fully transparent. Any shaded color on the surface would be added to the background in this case. You would want to use a fully transparent surface for a shader which simulated flames, for example. A transparency of (1, 0, 0) would create a surface with a red transparent filter. Only red light from the background would pass through the surface.
The pixel reflection value is replaced by the shader reflection value by Camera (it mixes them together with the shader's opacity setting). The reflection is added to the surface color after it has been filtered by the reflection value in the EIShaderAttributesRec. If your shader wishes to, it can add the shader reflection value in the EIShaderPixelRec. This will cause the shaders reflection to get added to the pixel reflection from any previous shaders instead of replacing it (which would be more natural).

The opacity channel of the EIShaderAttributes works the same way as the clip. The concept of a clip map differs only in that it was designed with texture maps in mind. We want to have texture maps produce a very sharp edged anti-aliased clip region regardless of how big or small the texture was in the rendered image. The opacity value is used as a soft edged transparency. When the texture maps get large in the image, the edges of the transparency become blurred.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

13.0 About Shader Map Variables

Below are the shader mapping variables. We are all familiar with the ElectricImage mapping modes Flat, Cubic, Cylindrical, and Spherical. In the world of shaders, the shader is defined in a cube one unit on a side of technically infinite resolution. Transformations take place between the user-specified mapping mode and parameters to the shader space during rendering. These transformations are stored in the map matrices.

```
/*  Mapping  variables  */

typedef  struct  {
       RMatrix4  mapMatrix;                /*  The  4  by  4  transformation  matrix  for  the
                                                  original  group  coordinates  */
       RMatrix4  mapWrapMatrix;            /*  The  4  by  4  transformation  from  shader  space
                                                  back  into  camera  space  */
       RMatrix3  mapBumpMatrix;            /*  The  3  by  3  pixel  bump  alignment  matrix  */
       float  mapBumpScale;                /*  The  bump  scaling  factor (applied  by  the
                                                   host)  */
       float  mapOpacity;                  /*  The  opacity  factor  (applied  by  the  host)  */
       float  map_X0,  map_Y0,  map_Z0;    /*  Mapping  coordinates  at  the  current screen
                                                   pixel*/
       float  map_X1,  map_Y1,  map_Z1;    /*  Mapping  coordinates  at  screen  pixel  x  +  1  */
       float  map_X2,  map_Y2,  map_Z2;    /*  Mapping  coordinates  at  screen  pixel  y  -  1  */
       float  map_DX,  map_DY,  map_DZ;    /*  Mapping  pixel  coordinate  deltas  */
       bool  mapPerspectiveFlag;           /*  True  if  the  mapMatrix  contains  a
                                                  non-linear  transformation  */

}  EIShaderMapRec,  *EIShaderMapPtr;
```

When you are performing anti-aliasing, you typically want to perform some type of averaging between adjacent pixels in high-contrast areas. The map_0, map_1, and map_2 triples allow you to calculate shading values for surrounding pixels in order to smooth out transitions. The map_D triple gives you information about the distance between two adjacent pixels in shader space.

mapBumpScale and mapOpacity values are applied by Camera and should be ignored by the shader. The user can adjust the bump scale in separately from the shader's bump scale (if it has one). The mapOpacity is used by Camera to mix the the surface color, transparency and reflection with previous values for the pixel.

mapPerspectiveFlag is set if the shader is appied with a non-linear projection matrix. An example of this would be a camera projection.

Copyright 1998-2011, EIAS3D

14.0 Anti-aliasing

Anti-aliasing is perhaps the most complex and time-consuming task in writing shaders. There are many books and SIGGraph papers on the subject. There is no one magic anti-aliasing algorithm that you can use for every application. Basically, aliasing comes from performing discrete sampling of a continuous function. It's somewhat of a level of detail issue. You may have a shading algorithm with lots of fine details but two adjacent pixels don't take this fact into account. Or, you may have a shading algorithm that has a very hard edge that would produce jaggies if not handled properly.

Let's look at the example of the CheckerBoard shader. CheckerBoard is both a 2D and a 3D shader. We'll look at the 2D case but the 3D case is virtually identical. This shader creates a grid of squares that alternate between two colors. The transistion areas are essentially straight lines. When these are viewed at some angle other than absolutely horizontal or vertical, you get jaggies. To resolve this, we'd like to blend the two colors right at the transition point. But how do we know that we are at this point? This is where being able to look at adjacent pixels comes in handy.

The call to generate a give pixel color looks like this:

```
factor = Calc2DCheckerFactor( theMap->map_X0 * density, theMap->map_Y0 * density,
                              theMap->map_DX * density, theMap->map_DY * density);
```

This function returns what number between 0.0 and 1.0. This is then passed on to a function that mixes the two colors. Normally, the CheckerBoard function will return either 0.0 or 1.0. It will return something in between at the transition points. The following code generates the blend factor. Note the shader can be viewed from such large distances that the eye would not see the grid, the function quickly returns an even blend factor. This is mainly a speed issue.

```
static float Calc2DCheckerFactor(float x, float y, float dx, float dy)
{
        SInt32 xi, yi;
        float maxDensity = MAX(dx, dy);

        if (maxDensity > 0.75)
                return 0.5;

dx *= 0.5;              dy *= 0.5;

        xi = FLOOR(x); yi = FLOOR(y);

        x -= xi;                y -= yi;

        if (x > 0.5)
                x = 1.0 - x;
        if (y > 0.5)
                y = 1.0 - y;
```


Copyright 1998-2011, EIAS3D

```
        if (x < dx)
                x = 0.5 + 0.5 * x / dx;
        else
                x = 1.0;

        if (y < dy) {
                y = 0.5 + 0.5 * y / dy;
                if (x > y)
                        x = y;
        }

        if (xi & 1)
                x = 1.0 - x;
        if (yi & 1)
                x = 1.0 - x;

        if (maxDensity > 0.5)
                x += (0.5 - x) * (maxDensity - 0.5) * 4.0;

        return x;
}  /* Calc2DCheckerFactor */
```


Copyright 1998-2011, EIAS3D

15.0 Illumination Shaders

Illumination shaders allow you to completely redefine how the surface reacts to light. Normally, a Phong shaded surface have fairly specific looks. There really isn't any way to simulate surfaces such as brushed metal or the diffraction qualities of a CD-ROM. This is where Illumination shaders come into play. Let's say you have a piece of stainless steel that has a single circular brush mark on it. Light that hits the surface from the 3 o'clock position produces a pie wedge shaped highlight in the 9 o'clock position. As you move the light source around, the highlight changes to maintain the opposite orientation.

In order to do this, you need to know where the light sources are. This comes from the HostLight call.

```
theShadePtr->HostLight(&theLight, &calcDiffuse, &calcSpecular, &dot,
               &normA, &normB, &normC, &curRed, &curGreen, &curBlue);
```

When you make this call,.if theLight is not NULL, there is another light that the shader must deal with. Camera will return the position of the light source in (x, y, z), the color of the light in (r, g, b), the dot product of the light with the visible side of the surface (negative values indicate the light is behind the surface). The diffuse and specular values are flags to indicate whether diffuse and specular illumination are enabled for that lightsource.

Illumination shaders require more work on the part of the programmer because you are responsible for determining whether or not a given light actually strikes the surface in question. The shader is also responsible for handling the minimum ambient light level, specular highlights, luminance, reflection color bias, etc.

The the following code essentially duplicates standard Phong shading.

```
SInt32 EIShaderShade(EIShaderShadeRec *theShadePtr)
{
        EIShaderAttributesPtr theAttributes = theShadePtr->shaderAttributes;
        EIShaderPixelPtr thePixel = theShadePtr->shaderPixel;
        float temp;
        float rNum, gNum, bNum; /* The calculated color values */
        float sRed = 0.0;               /* The total light falling upon the surface */
        float sGreen = 0.0;
        float sBlue = 0.0;
        float hRed = 0.0;               /* The total highlight falling upon the surface */
        float hGreen = 0.0;
        float hBlue = 0.0;
        bool transmissionFlag = (theAttributes->transmission.r > 0.0) ||
                                (theAttributes->transmission.g > 0.0) ||
                                (theAttributes->transmission.b > 0.0);
        char *theLight = NULL;


        /* Calculate the total illumnination color values */

        do {
```


Copyright 1998-2011, EIAS3D

```
float dot;
float normA, normB, normC;                    /* The current lightsource's normal */
float curRed, curGreen, curBlue;              /* The current lightsource's color */
SInt32 calcSpecular;
SInt32 calcDiffuse;



/* Calculate the lightsource illumination */

theShadePtr->HostLight(&theLight, &calcDiffuse, &calcSpecular, &dot,
                                    &normA, &normB, &normC, &curRed, &curGreen, &curBlue);

/* Calculate the diffuse and specular shading values */

if (calcDiffuse)
        if (dot > 0.0) {
                if (calcSpecular) {
                        float highVal = dot + dot;

                        /* Adjust the highlight value if the group
                                                    has a non-linear highlight factor */

                        highVal = (highVal * thePixel->shadeNormalA - normA) * thePixel->shadeXN +
                                (highVal * thePixel->shadeNormalB - normB) * thePixel->shadeYN +
                                (highVal * thePixel->shadeNormalC - normC) * thePixel->shadeZN;

                        if (highVal > 0.0) {
                                highVal = theAttributes->highlight * POW(highVal, theAttributes->spread);
                                if (highVal > 1.0)
                                    highVal = 1.0;

                                /* Components must be compared to prevent highlights from dark lights */
                                if (curRed > 0.0)
                                        hRed += curRed * highVal;
                                if (curGreen > 0.0)
                                        hGreen += curGreen * highVal;
                                if (curBlue > 0.0)
                                        hBlue += curBlue * highVal;
                                }
                        }

                        /* Adjust the dot if the group has a non-linear terminator factor */

                        if (theAttributes->terminator != 1.0)
                                dot = POW(dot, theAttributes->terminator);
                        sRed += curRed * dot;
                        sGreen += curGreen * dot;
                        sBlue += curBlue * dot;

                } else  if (transmissionFlag) {

                        /* Adjust the dot if the group has a non-linear terminator factor */

                        if (theAttributes->terminator != 1.0)
                                dot = POW(dot, theAttributes->terminator);
                        sRed += curRed * dot * theAttributes->transmission.r;
                        sGreen += curGreen * dot * theAttributes->transmission.g;
```


Copyright 1998-2011, EIAS3D

```
                        sBlue += curBlue * dot * theAttributes->transmission.b;
                }
} while (theLight);

/* The global ambient will insure that illumination does not fall below a minimum level */


sRed += thePixel->shadeAmbientR * theAttributes->ambient.r;
sGreen += thePixel->shadeAmbientG * theAttributes->ambient.g;
sBlue += thePixel->shadeAmbientB * theAttributes->ambient.b;

temp = thePixel->shadeMinAmbientR * theAttributes->ambient.r;
if (sRed < temp)
        sRed = temp;
temp = thePixel->shadeMinAmbientG * theAttributes->ambient.g;
if (sGreen < temp)
        sGreen = temp;
temp = thePixel->shadeMinAmbientB * theAttributes->ambient.b;
if (sBlue < temp)
        sBlue = temp;

/* Start with the surface color */

rNum = theAttributes->reference.r;
gNum = theAttributes->reference.g;
bNum = theAttributes->reference.b;

/* Calculate the diffuse color */

rNum *= theAttributes->diffuse.r;
gNum *= theAttributes->diffuse.g;
bNum *= theAttributes->diffuse.b;

/* Calculate the color values by using the light source illumination values.
                The glossy calculation is a simple image */
/* beautification algorithm which adds white as the light intensity increases past 75%. */

temp = theAttributes->gloss;
if (sRed > temp)
        rNum = rNum * temp + sRed - temp;
else
        rNum *= sRed;
if (sGreen > temp)
        gNum = gNum * temp + sGreen - temp;
else
        gNum *= sGreen;
if (sBlue > temp)
        bNum = bNum * temp + sBlue - temp;
else
        bNum *= sBlue;


/* Factor the luminance into the color values */

temp = theAttributes->shade;
if (temp > 0.0) {
        rNum += temp * (theAttributes->reference.r - rNum);
        gNum += temp * (theAttributes->reference.g - gNum);
```


Copyright 1998-2011, EIAS3D

```
        bNum += temp * (theAttributes->reference.b - bNum);
}

/* The luminance color will be added to the shaded color */

rNum += theAttributes->luminance.r;
gNum += theAttributes->luminance.g;
bNum += theAttributes->luminance.b;

/* Add the highlight to the color values if any */

if (BTST(theAttributes->shadeFlags2, groupShadeBiasSpecularFlag)) {
        float theSpecularValue = theAttributes->specular.r * 0.3 +
                                        theAttributes->specular.g * 0.59 +
                                        theAttributes->specular.b * 0.11;

        rNum += hRed * theSpecularValue * theAttributes->reference.r;
        gNum += hGreen * theSpecularValue * theAttributes->reference.g;
        bNum += hBlue * theSpecularValue * theAttributes->reference.b;
} else {
        rNum += hRed * theAttributes->specular.r;
        gNum += hGreen * theAttributes->specular.g;
        bNum += hBlue * theAttributes->specular.b;
}

/* Add the reflection to the color values if any */

if (BTST(theAttributes->shadeFlags2, groupShadeBiasReflectionFlag)) {
        rNum +=         theAttributes->reflection.r *
                                thePixel->shadeReflection.r *
                                theAttributes->reference.r;
        gNum +=         theAttributes->reflection.g *
                                thePixel->shadeReflection.g *
                                theAttributes->reference.g;
        bNum +=         theAttributes->reflection.b *
                                thePixel->shadeReflection.b *
                                theAttributes->reference.b;
} else {
        rNum += theAttributes->reflection.r * thePixel->shadeReflection.r;
        gNum += theAttributes->reflection.g * thePixel->shadeReflection.g;
        bNum += theAttributes->reflection.b * thePixel->shadeReflection.b;
}


/* Limit color values into the range of 0..1 */

if (rNum > 1.0)
        rNum = 1.0;
else if (rNum < 0.0)
        rNum = 0.0;
if (gNum > 1.0)
        gNum = 1.0;
else if (gNum < 0.0)
        gNum = 0.0;
if (bNum > 1.0)
        bNum = 1.0;
else if (bNum < 0.0)
        bNum = 0.0;
```


Copyright 1998-2011, EIAS3D

```
        /* Return the final color value */

        thePixel->shadeColor.a = theAttributes->alpha;
        thePixel->shadeColor.r = rNum;
        thePixel->shadeColor.g = gNum;
        thePixel->shadeColor.b = bNum;

        return shaderNoError;
}  /* EIShaderShade */
```


Copyright 1998-2011, EIAS3D

Electric Image Animation System

16.0 Shader Geometry Access

Since EI 8.0, shaders can read geometry data (vertices and facets) of shaded groups, in a way that is similar to model plug-ins. A shader should call the HostModelShade callback defined as:

```
SInt32 EIModelShade( SInt32 theCommand, UInt32 theCommandSize, void * theCommandData );
```

The commands and formats are described in the EIShaderModel.h file. The shader should pass shadeGroupReference and shadeFacetReference (obtained in the shaderPixel structure) to specify a group and/or facet to read geometry from.

17.0 Intercommunication between shaders and model plug-ins

Shaders can use custom data created by plug-ins by examining the following fields of the EIShaderInitializeRec structure.

```
void *shaderPluginData;             /* The plugin data block (usually nil) */
UInt32 shaderPluginID;              /* The ID of the plugin that created the data block */
UInt32 shaderPluginSize;            /* The size of the plugin data block */
```

Model plug-ins can create data by using hostModelGetGroupData/hostModelSetGroupData calls. See the Plug-in Manual for more details.

Copyright 1998-2011, EIAS3D

18.0 Multi-threaded Shaders

A multi-threaded shader should be compiled as a multi-threaded library and be able to run correctly in a multi-threaded environment. These shaders should not make assumptions about their calling sequence, should not write global variables at shading time, or provide a synchronization for such reading/writing. Here is a simple example:

```
/* Do NOT do this */

/* shader's global variables */

float nX, nY, nZ;

/* Copy actual surface normal */

nX = theShaderShade->shaderPixel->shadeNormalA;
nY = theShaderShade->shaderPixel->shadeNormalB;
nZ = theShaderShade->shaderPixel->shadeNormalC;

/* Call a function that uses nX, nY, nZ */

CalculateSomething();
```

This code will NOT work properly in a multi-threaded renderer. Any number of instances of the same shader can run simultaneously, therefore global variables (nX, nY, nZ in the example) can be re-written by another instance and the result of the CalculateSomething call will be junk.

EI 9.0 does not require shaders' multi-threading. Camera recognizes a shader is multi-threaded by examining the EIMP resource with ID 128 in a shader's resource file. If this resource found and its length is 4 bytes long (all zeroes), then this shader can be called in parallel. Otherwise the render threads will wait until the shader returns control to the host and only then call the shader again.

Copyright 1998-2011, EIAS3D

Some shaders can be applied 2 or more times to the same texture stack. A typical workflow is: The first instance stores some data (surface color for example) in order to use it by the next instance(s). There are no principal obstacles to do the same with multi-threading but a shader must not make assumptions that a second instance is called immediately after the first one. With multi-threaded rendering, the order of calls is not defined. A good idea is to examine the following field of EIShaderPixelRec

```
UInt32 shadeThreadIndex;      /* The shade thread index */
```

and use the array (one element per each thread) for stored values.

Copyright 1998-2011, EIAS3D

19.0 Miscellaneous

For some shaders, it is necessary to know the angle between the surface normal and the incident angle. The following code fragment shows how to do this from data provided by the Shader Pixel record.

```
ShaderDataPtr theShaderData = (ShaderDataPtr)theShadePtr->shaderData;

EIShaderPixelPtr thePixel = theShadePtr->shaderPixel;

float Xi = thePixel->shadeXN;
float Yi = thePixel->shadeYN;
float Zi = thePixel->shadeZN;

float Na = thePixel->shadeNormalA;
float Nb = thePixel->shadeNormalB;
float Nc = thePixel->shadeNormalC;

float Nf = (Xi * Na) + (Yi * Nb) + (Zi * Nc);
```

19.1 How EI Shaders relate to Renderman shaders

```
/* Comments in the EIShader.h file */

float shadeDistance;                                    /* |P| */
float shadeXN, shadeYN, shadeZN;                              /* P normalized */
float shadeNormalA, shadeNormalB, shadeNormalC;         /* N at P' - the surface normal */

/* Pseudo- Renderman translation */

float shadeDistance;                                    /* The distance of pixel to camera*/
float shadeXN, shadeYN, shadeZN;                              /* I or incident angle */
float shadeNormalA, shadeNormalB, shadeNormalC;   /* N or surface normal(after earlier perturbs) */
```

In Renderman, many variables are of the type Point. This is an X,Y,Z triple. In EI Shaders, point variables are stored as their individual components.

In Renderman, the variable P is used to define the surface position in space. Typically, this is transformed into texture space which is the UV parameter space. In EI Shaders, you can access the current map position by use of the map_X0, map_Y0, map_Z0 found in the `EIShaderMapRec`. These variables range from –0.5 to +0.5.In Renderman, the variable Ci is output color of the surface. The EI equivalent is `ARGBReal shaderColor` found in the `EIShaderShadeRec`. The variable Cs is the input surface color and is represented by `shadeColor` in the `EIShaderPixelRec`.

Oi and Os are opacity output and surface colors in Renderman. Here, you have a number of options. You could convert Renderman opacity to an alpha channel, a clip map value, or a transparency. The choice is yours. In the case of the Eroded shader, Clip mapping is used so that the eroded holes can have light pass through them.

The E variable is the position of the Camera. EI Shaders give you access to a full blown transformation matrix that allows you to find out where the camera is in World space. The matrix `*shadeCameraInvMatrix` from the `EIShaderPixelRec` contains the camera position in world space in the m30, m31, and m32 components.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

19.2 Distance from the Camera to the pixel being rendered

The camera is always at (0, 0, 0). The point being shaded is at `(shadeX, shadeY, shadeZ)`. The distance from the camera to the point being shaded is `SQRT(shadeX^2 + shadeY^2 + shadeZ^2)` which is `shadeDistance`. If you are only interested in the distance along the Z axis ignoring the X and Y distance, that is simply `shadeZ`.

19.3 Shader Preview Thumbnails

Shaders can have thumbnails shown inside Animator. To create a thumbnail simply add a 'PICT' resource with ID 21000 to shader's rsc file. The recommended image size is 128x128

19.4 Copy Protection

Shaders have the ability to query the ElectricImage hardware key to determine the user's serial number. Currently there are two slightly different serial number types. For Film versions of ElectricImage, the serial number starts with two uppercase letters followed by five digits. For Broadcast versions, the serial number is the same as for Film versions except that there is a capital B following the five digits.

The call to retrieve the serial number only returns the last seven characters of the serial number. For Film versions, this is the whole serial number. For Broadcast versions, the first letter is dropped.
The following code can be used to get this serial number:

```
Uint32 EIShaderSerial(EIShaderSerialRec *theSerialPtr)
{
        Sint32 theChallengeType = QUADWORD('SERL'); char testString[8];
        theSerialPtr->HostChallenge(theChallengeType, &testString[0]);
        return shaderNoError;
} /* EIShaderSerial */
```


Copyright 1998-2011, EIAS3D

20.0 Building an EI Shader

Shaders are built on the Macintosh using XCode 3.1 or later. They are built under Windows using MSVC 2008 or later.

You must define one of the platform symbols in order to properly compile. The valid platform symbols are:

```
__MACINTOSH__
__WIN32__
__WIN64__
```

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.).

20.1 File Names and Extensions

Similar to plugins, shaders should include an extension. The standard extensions used by Electric Image shaders are:

.shd	for surface shaders
.shl	for layers shaders

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).

On the Macintosh platform, a shader should be built via Xcode as a universal 32/64 bits library. On the Windows platform you must provide a second library to run in 64-bit mode with the, suffix _64 placed before the extension. Example:

Macintosh platform:
Eroded.shd - universal library holds both the 32-bit and 64 bit code

Windows platform:
Eroded.shd - shader’s library (dll) holds the 32-bit code
Eroded_64.shd - shader’s library (dll) holds the 64-bit code

On the Windows platform both of the shader’s files (Eroded.shd and Eroded_64.shd) should be placed into the “EI Shaders” folder.

Copyright 1998-2011, EIAS3D

Electric Image Animation System

20.2 Resource Files

All EI Shaders are required to have a separate resource file with the same name but with the .rsc extension. This file must be in the same folder as the shader's .shd/.shl file (the EI Shaders folder by default).

The resource file format is different on Macintosh and Windows.

Do NOT simply copy .rsc files from Macintosh to Windows or vice versa, such copies won't work.

The reason is that on Macintosh the resource file contains resources in the native Macintosh format (e.g. resources stored in the file's resource fork), but on Windows, the .rsc file is really a MacBinary II 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 on Windows 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 II files.

Note: on the Windows platform, the same resource file is used for both: 32 and 64 bits, for example a full install of the Eroded shader on Windows contains 3 files:

Eroded.shd
Eroded_64.shd
Eroded.rsc

Copyright 1998-2011, EIAS3D

20.3 Endian Issues

The data passed to the shader from the host will always be in big-endian format. This means that under Windows, 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 {
        UInt32 shaderOwnerType;
        UInt32 shaderVersionNum;
        UInt32 one;
        UInt32 two;
        SInt32 three;
} ShaderInterface;

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

N O T E : Do not use the macro FixField (mixed case); use FIXFIELD (all caps).

N O T E : Do not use FIXFIELD on values of type ARGB and custom controls. Their values will already be in the correct byte order when your shader is called.

Copyright 1998-2011, EIAS3D

In the example below, 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 they should be reverted to the default values.

```
static void GetShaderInterface(      void* theInterface, SInt32 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 */
```


Copyright 1998-2011, EIAS3D