The ElectricImage Animation System

Shader API Documentation

Release 1.0

October 27, 1998

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.

Many people have requested direct support for Renderman™ shaders in ElectricImage. There are two very good reasons why EI shaders are better than the Renderman variety. Both are due to the fact that EI shaders are written in the C language. First is debugging. Because you write the shader in C and compile it with a symbol file, you can use a runtime debugger to watch what the shader is doing. Also, Renderman doesn't allow you to write your own specialty functions such as fractal noise. In Renderman, you are given Perlin noise. While Perlin noise is very useful, it is also very recognizeable. The ElectricImage shader system allows you to write and use different types of noise such as cellular noise.

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). First is

```
long 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.

Second is

```
long 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.

The third major shader function is

```
long 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

```
long 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.

Memory Allocation

Memory allocation is very important to shader operation. Memory allocation must be done only during the Initialize phase. You should not attempt to allocate memory elsewhere because the host, usually Camera, soaks up all available memory. Memory allocation is similar to a standard C malloc function.

The Shader API makes use of what are known and procedure pointers. In the C Language, it is possible to define a pointer to some routine. In the API, this is the mechanism that the shader uses to talk to the host application e.g. ElectricImage, Camera, or the Shader Test program. The details and inner workings of this mechanism are not important to shader authors.

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.

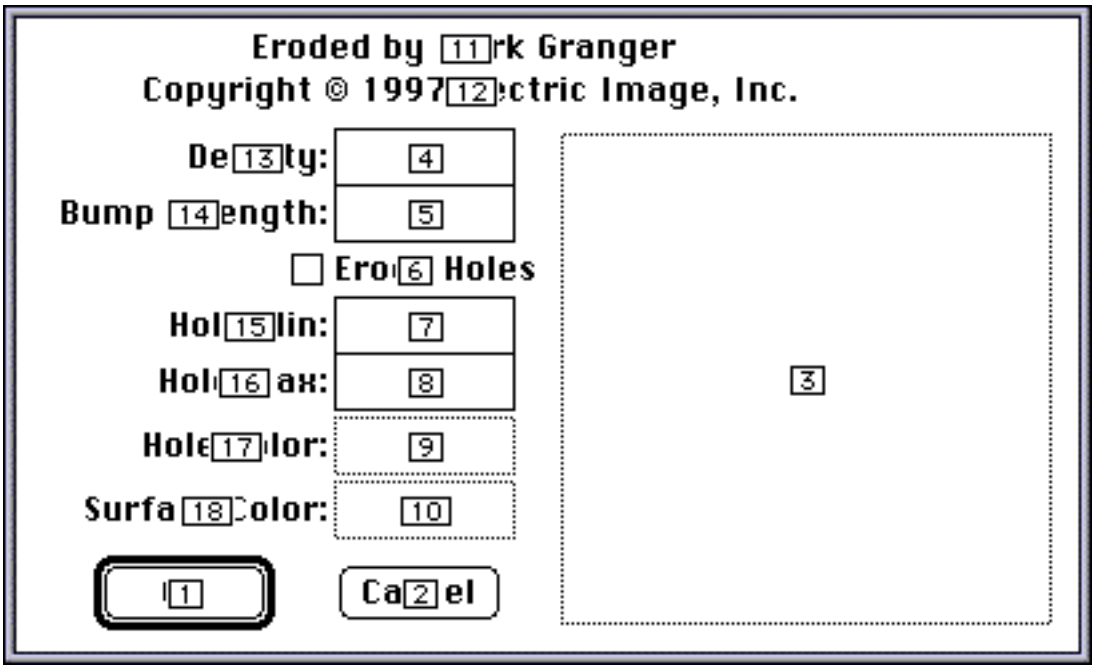
```
if (!InitNoise((NoiseMallocFunction)theInitializePtr->HostMalloc, NULL))
   return shaderMemoryError;
```

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.

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. There is quite a bit of built-in functionality for data entry and previewing.



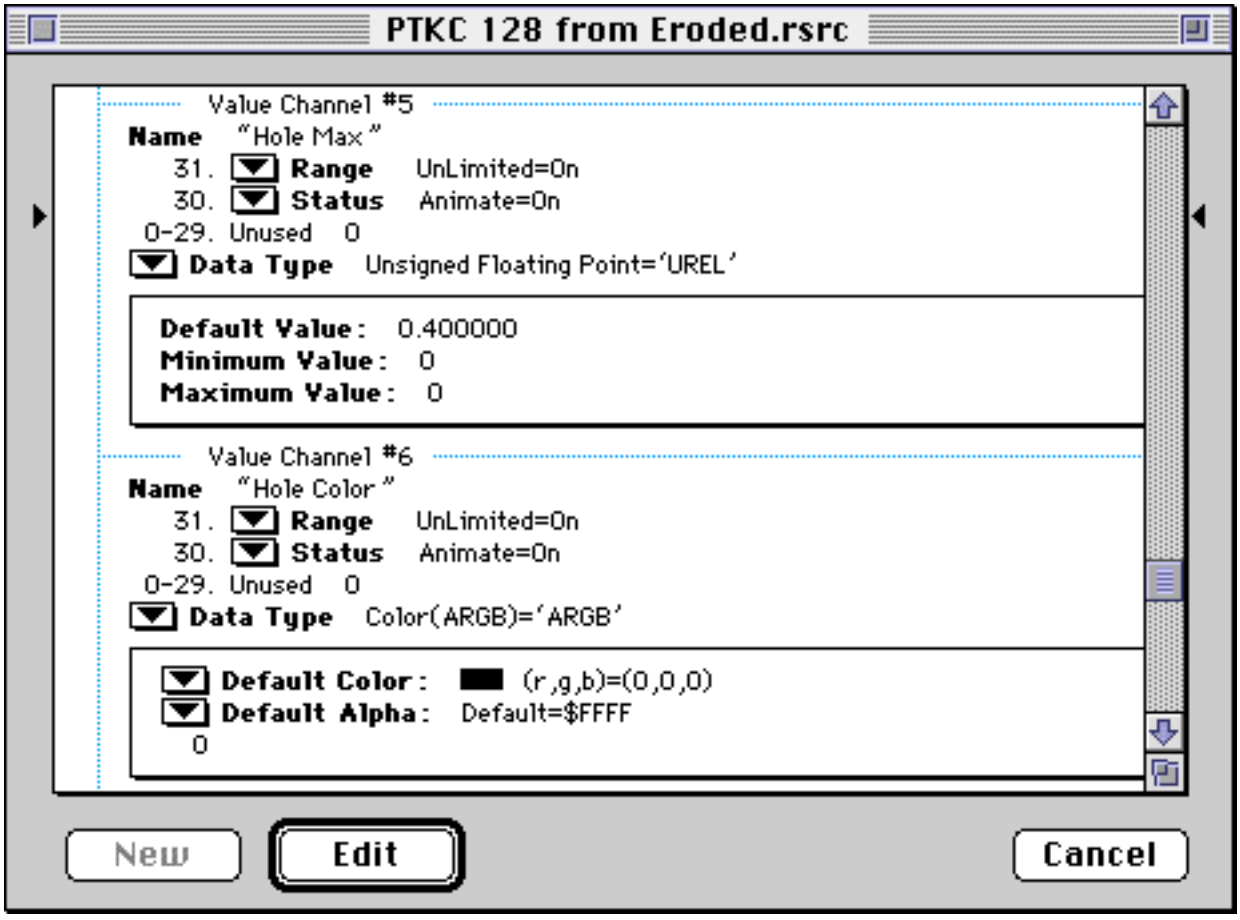
Have a look at the DLOG resource for the Eroded shader. There are three required items for shader interfaces. The OK button and the Cancel button should be items 1 and 2 respectively. Item 3 should be a user item. This is used for the shader preview. The programmer does not have to write any special code to generate the preview.

All data entry DITL elements must have item numbers 4 and higher. All non-enterable items such as static text or picture items must follow all enterable items. In the Eroded shader, items 4 through 10 are enterable. The rest are not.

Color button items are user items.

There is currently no facility for supporting custom controls such as sliders.

Once you have defined the actual dialog for the shader, you must also define the PTKC 128 resource. This is a template that defines the shader's control variables.



This is a snapshot of a portion of the Eroded shader. Each control variable has a Value Channel. Bit 31 tells the shader system whether or not this variable has limited range of settings. If it does, the maximum and minimum values are specified below the Default Value. Bit 30 tells the system whether or not to create an animation channel for this variable. Bits 0 through 29 are reserved for future use.

The variables data type is also specified in the Value Channel Definition. Valid types are Signed and Unsigned floating point numbers, Signed and Unsigned integers, Booleans, Pop-ups, RGBs, and ARGBs.

The order of the Value Channel entries must correspond to the order of the enterable items in the dialog.

It is important to note that as of this writing, the default values of the user-interface come not from the C source code as you might expect. Instead, they come from the PKTC record entries themselves.

Also, you'll see that the shader interface has a preview box. This is a required element just as the Ok and Cancel buttons. It's important to know that while the preview is handled for you by the host, currently, this preview only shows you X and Y shader space. A shader that does things specifically on the Z axis won't show you very much in the preview.

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 {
	OSType shaderOwnerType;
	ulong shaderVersionNum;
	float shaderDensity;
	float shaderBumpStrength;
	long shaderErodeHoles;
	float shaderHoleMin;
	float shaderHoleMax;
	ARGB shaderHoleColor;
	ARGB shaderSurfaceColor;
} ShaderInterfaceRec, *ShaderInterfacePtr;

typedef struct {
	float shaderDensity;
	float shaderBumpStrength;
	long 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.

Also, the ShaderDataRec may contain variables that you don't want to provide an interface for. These may be computed values that you want to store between calls to the shader.

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 */
       ulong shaderAccessFlags;                        /* Flags specifying which host parameters the
                                                           shader may write */
       ulong 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 */
       RGBReal shaderTransparency;                     /* The shader transparency color */
       RGBReal shaderReflection;                       /* The shader reflection color */
       ARGBReal shaderColor;                           /* The shader color and opacity */
} 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.

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 CheckerBoard 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.

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.

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) */
      ulong shadeFlags;                       /* see "GATR Group Shading Flags" in FACTStuff.h */
      ulong shadeFlags2;                      /* see "GATR Group Shading Flags2" in FACTStuff.h */
} EIShaderAttributesRec, *EIShaderAttributesPtr;
```

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.

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.

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.

About 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)
{
       long 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;

       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 */
```

About 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.

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

   /* 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 */
```

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);
```

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.

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`.

Shader Preview Thumbnails

When you add a shader or a texture to a group inside of ElectricImage, the file dialog box has a thumbnail preview on the left side. Shaders can have a preview as well. The procedure for creating this thumbnail is fairly simple.

1. Render any scene you wish with the shader applied to any geometry to an image file. Note, that the standard EI shaders have a simple 2D thumbnail but there's no reason why you couldn't use a teapot or some other image as a thumbnail. Generally, the rendered image should be square i.e. 320 by 320 pixels.
2. In EI, add any model and add the previously created image as a texture. When you do this, the file dialog will say "Click here for preview." When you do this, ElectricImage generates a thumbnail image in the EIPV resource in the texture's file.
3. In Resorcerer or ResEdit, copy this resource into the shader's resource file.

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:

```
long EIShaderSerial(EIShaderSerialRec *theSerialPtr)
{
	OSType theChallengeType = QUADWORD('SERL');
	char testString[8];

	theSerialPtr->HostChallenge(theChallengeType, &testString[0]);

	return shaderNoError;
}  /* EIShaderSerial */
```

What you can't do in the current API

While the current EI Shader API is robust and flexible, there are some things that you cannot do.

- It is not possible to do volumetrics.
- You cannot access the geometry of the model being shaded.
- You cannot build fancy user-interfaces. Custom control types such as sliders are not supported.
- The API does not allow you to change the rendering pipeline.

Tutorial

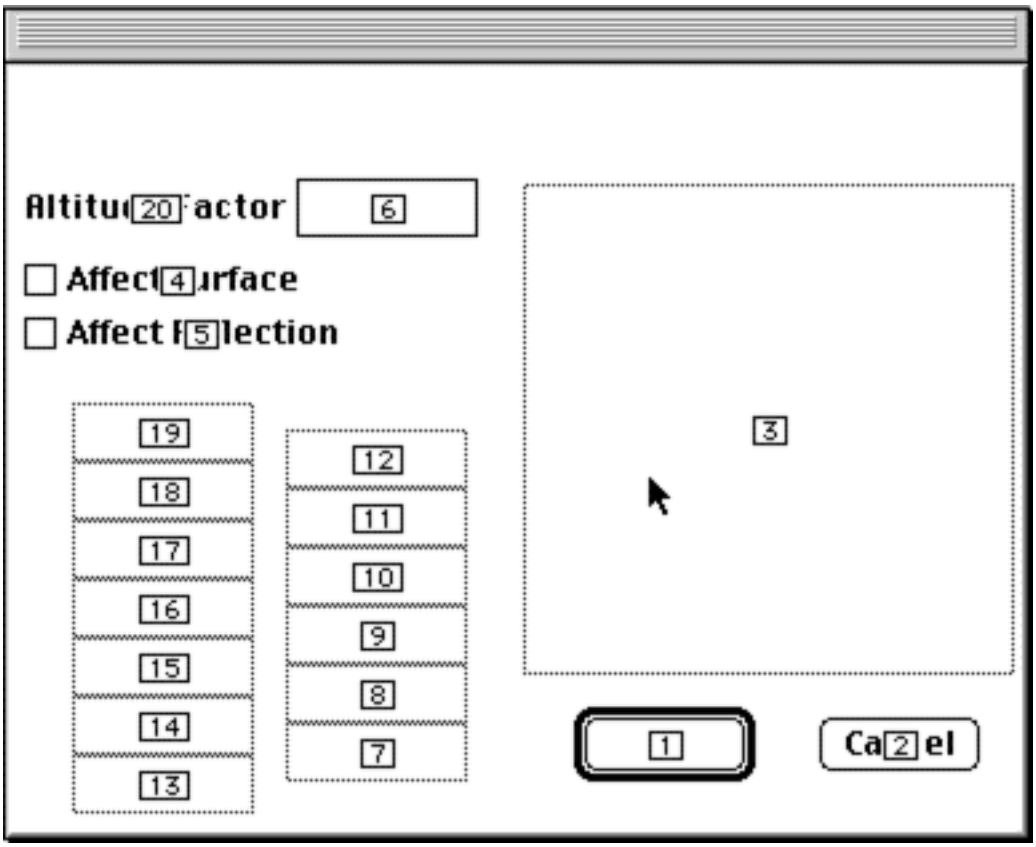
In this section, we will convert a typical Renderman shader into an ElectricImage shader. The following code creates the familiar “desert and sky” chrome ramp.

```
#define BROWN     color (0.1307,0.0609,0.0355)
#define BLUE0     color (0.4274,0.5880,0.9347)
#define BLUE1     color (0.1221,0.3794,0.9347)
#define BLUE2     color (0.1090,0.3386,0.8342)
#define BLUE3     color (0.0643,0.2571,0.6734)
#define BLUE4     color (0.0513,0.2053,0.5377)
*#define BLUE5     color (0.0326,0.1591,0.4322)
#define BLACK     color (0,0,0)

surface
metallic()
{
    point Nf = normalize(faceforward(N, I));
    point V = normalize(-I);
    point R;        /* reflection direction */
    point Rworld;   /* R in world space */
    color Ct;
    float altitude;

    R = 2 * Nf * (Nf . V) - V;
    Rworld = normalize(vtransform("world", R));
    altitude = 0.5 * zcomp(Rworld) + 0.5;
    Ct = spline(altitude,
        BROWN, BROWN, BROWN, BROWN, BROWN,
        BROWN, BLUE0, BLUE1, BLUE2, BLUE3,
        BLUE4, BLUE5, BLACK);
    Oi = Os;
    Ci = Os * Cs * Ct;
}
```

Basically, this shader is computing a fake reflection by using the shading normals and the camera angle as the incident ray. The details of why the shader creates what it does is beyond the scope of this discussion. What we want is to know how to convert this to EI Shader form.

This shader doesn't have any inputs in its Renderman form but we'll add a few just to illustrate how this is done. We'll define color buttons for each of the colors in the gradient. We'll also define checkboxes to illustrate how they are placed in a bitfield.

First let's create the user-interface. You need a Macintosh resource editor to view and make changes to shader resources. ResEdit or Resorcerer will work. Open the resource file and have a look at the dialog.



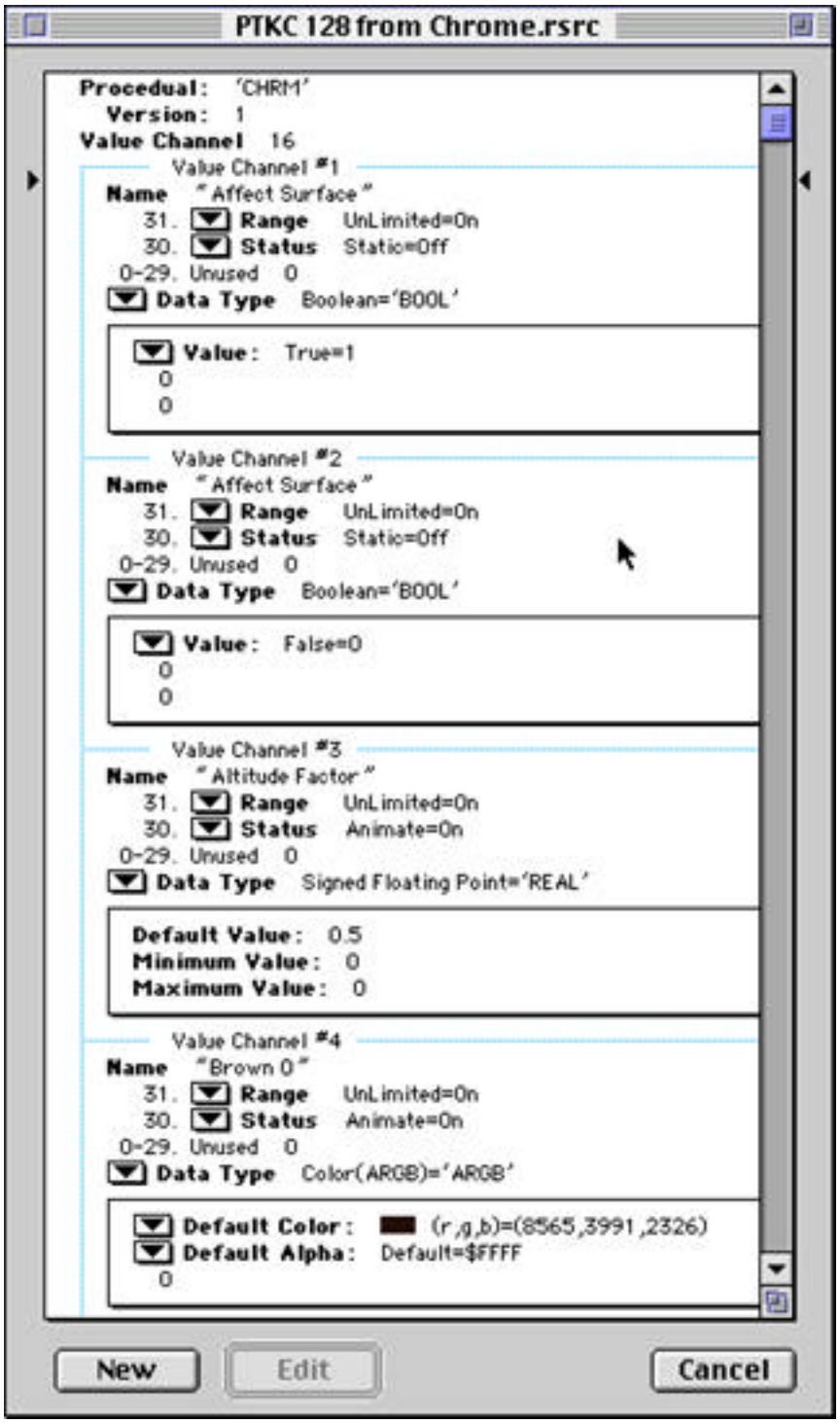
It's important to note the item numbers in this dialog. The OK button should always be 1, the Cancel Button should always be 2, and the Preview box, which is a User Item, should always be 3. These three items are required for all shaders.

It's a good idea to set up any additional controls with certain conventions. Group like-typed items together. Note that we've created two Checkboxes as items 4 and 5. Item 6 is an Editable Text item and will be used to change the rate at which the gradient is traversed. Note that the label text for this

edit item is not item 7. Any Static Text items must follow the active controls otherwise the system will ignore some of the active controls.

We've created thirteen user items to be used as color buttons for the chrome ramp. Arrangement of the on-screen locations of controls is totally up to the user. We could have set them up in one column but that would have left a lot of empty space in the dialog.

Once the dialog is created, we need to create a template for the controls. This template is used by the system to a mapping between the shader's data structures and the user interface items. Here is a fragment of the PKTC resource for the Chrome shader:



The first field called Procedural is a four-character code used to identify the shader. Here we've chosen CHRM. The second field is a version number. Following these two fields are the Value

Channels for each shader variable. There must be a value channel for every control and they must be in the same order as they are in the dialog.

We've created two Boolean channels to correspond to the Affect Surface and Affect Reflectivity checkboxes. The third Value Channel we want to be a floating point number. You can limit the range of values that the user can type in by setting the Range flag to Limited and specifying Minimum and Maximum values. You can also create an animation channel in the EIAS Project window by setting the Status flag to Animate=On. Just click on the little down arrows to change the value.

Value Channels 4 through 16 are ARGB color channels. ARGB channels are 48 bits wide, sixteen bits per color component. However, the actual shader code uses floating point numbers for the color components. More on converting to and from these later.

Renderman also uses floating point numbers for its colors. The Renderman source looks like this:

```
#define BROWN     color (0.1307,0.0609,0.0355)
#define BLUE0     color (0.4274,0.5880,0.9347)
#define BLUE1     color (0.1221,0.3794,0.9347)
#define BLUE2     color (0.1090,0.3386,0.8342)
#define BLUE3     color (0.0643,0.2571,0.6734)
#define BLUE4     color (0.0513,0.2053,0.5377)
#define BLUE5     color (0.0326,0.1591,0.4322)
#define BLACK     color (0,0,0)
```

Simply take each floating point number and multiply it by 65535.

The Renderman source uses the above colors in a spline function. Since it uses some of the colors several times, we've chosen to define these explicitly.

Now we need to create the data structures that the shader will need. There are two main structures. One is used to control the shader internally and the other acts as a bridge between the value channels of the user interface and the shader control stucture.

The following is the shader control structure or more correctly, the shader data structure.

```
typedef struct {
	long shaderAffectFlags;
	float shaderAltitudeFactor;
	ARGBReal chromeColors[13];
} ShaderDataRec, *ShaderDataPtr;
```

The first field is a long integer and is used to store any On/Off type variables. For those of you who are new to the nuances of C programming, a Boolean variable only needs one Bit of memory to hold it. However, there is no true single bit data type. The data type `bool` actually holds 8 bits. Therefore, using `bools` would be a waste of memory. The most efficient way to store Boolean variables is to access the individual bits of a larger data type. Long Integers have 32 bits. They are numbered 0 through 31. The Shader API has macros for setting, clearing, and testing bits. So, in case of this shader, we're storing both the Affect Surface and Affect Reflection booleans in the shaderAffectFlags field.

The rest of the fields in the data structure are pretty self-explanitory. It's important to remember that you can put other fields in this structure that don't necessarily have user-interface counterparts. By doing this, you can easily access a variable in many parts of the shader code.

Here is what the Interface data structure looks like:

```
typedef struct {
	OSType shaderOwnerType;
	ulong shaderVersionNum;

	long shaderAffectFlags;
	float shaderAltitudeFactor;
	ARGB chromeColors[13];
} ShaderInterfaceRec, *ShaderInterfacePtr;
```

Each field corresponds directly with the contents of the PKTC resource including the four-character shader ID and the version number.

You'll notice a slight difference between the Interface structure and the Data structure. The data type of the color variables doesn't match. That's because the interface uses 32-bit integer colors whereas the shader itself uses floating point colors. You'll see later how to bridge these two data types.

There are four main functions that each shader must have. They are:

```
long EIShaderInformation(EIShaderInformationRec *theInformationPtr);
long EIShaderInitialize(EIShaderInitializeRec *theInitializePtr);
long EIShaderFinish(EIShaderFinishRec *theFinishPtr);
long EIShaderShade(EIShaderShadeRec *theShadePtr);
```

Let's look at the Information phase. For the Chrome shader, the routine looks like this:

```
long EIShaderInformation(EIShaderInformationRec *theInformationPtr)
{
	ShaderInterfaceRec theShaderInterface;

	/* Get the shader interface parameters */

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

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

	BSET(theInformationPtr->shaderFeatureFlags, shaderAntialiasFlag);

	if (BTST(theShaderInterface.shaderAffectFlags, 31))
		BSET(theInformationPtr->shaderFeatureFlags, shaderColorFlag);

	if (BTST(theShaderInterface.shaderAffectFlags, 30))
		BSET(theInformationPtr->shaderFeatureFlags, shaderReflectionFlag);


	return shaderNoError;
}  /* EIShaderInformation */
```

The Information phase serves two purposes. One is to get data from the user-interface. The other is to set any attribute flags or access flags. Attribute flags tell Camera what the shader is going to be generating such as bumps, clip maps, etc. Access flags tell Camera what surface attributes the shader might need to look at and/or modify.

The Chrome shader's Information phase first calls a separate routine to get the user-interface data. The routine GetShaderInterface looks 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 */
```

A pointer to the memory holding the user-interface variables is given to the shader when EI calls the Information routine. You should compare the size of this memory with what you expect the size of your interface data structure. This is a check against a mismatch between the PKTC channels and your internal data structure.

Next, we want to check to see if the shader type and version number match. Here, if they don't we call a routine DefaultShaderInterface. That routine would load in fixed values for the shader. This routine is only called if there is a miscommunication problem between EI and the Shader. The other thing we need to do is a test to see if we are running on a different platform such as Windows NT. There are some differences in the way Intel processors and Motorola processors store data. If the version number is stored in reverse, you know you need to reverse all the other interface data in order to use it.

Now, let's get back to the attribute and access fields. The Chrome shader specifies that it will be doing anti-aliasing. It also says that it might be generating surface colors and/or reflection colors depending on state of the checkboxes in the user interface.

The next phase is the Initialize phase. This is where you allocate memory for the shader's data structure and any noise libraries you might want to use. After you've successfully allocated memory for the shader data structure, you will then transfer data from the interface structure to the shader data structure.

```
long EIShaderInitialize(EIShaderInitializeRec *theInitializePtr)
{
   ShaderDataPtr theShaderData;
   short i;

   /* 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->shaderAltitudeFactor = theShaderInterface.shaderAltitudeFactor;
      theShaderData->shaderAffectFlags = theShaderInterface.shaderAffectFlags;

      for (i=0;i<13;i++) {
         theShaderData->chromeColors[i].a = theShaderInterface.chromeColors[i].argb.a / 255.0;
         theShaderData->chromeColors[i].r = theShaderInterface.chromeColors[i].argb.r / 255.0;
         theShaderData->chromeColors[i].g = theShaderInterface.chromeColors[i].argb.g / 255.0;
         theShaderData->chromeColors[i].b = theShaderInterface.chromeColors[i].argb.b / 255.0;
      }

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

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

We're calling a fairly mysterious-looking routine called HostMalloc. Basically, this little incantation allocates a block of memory and returns a pointer to it. Here we are asking it to create a block of memory big enough to hold the ShaderDataRec which is internal control variable structure. Once we've done this, we need to check to see if the allocation worked. If it did, then the pointer will have some address other than 0.

If the memory allocation was successful, we call the same GetShaderInterface routine that we did during the Information phase. This places all the user-interface data into the local variable

theShaderInterface. Now we need to transfer the interface variables into the ShaderDataRec variables. Usually, this is done with simple equates. However, in the case of colors, we need to perform the conversion from integer-based colors to floating point-based colors as we talked about earlier. This can be done by taking each integer-based color component and dividing it by 255.0.

After the data transfer and conversion is complete, we need to return the pointer of the ShaderDataRec memory back to the host so that it can use it in the Shading and Finish phases.

The Finish phase is pretty simple. Usually, all you will do here is deallocate memory that you allocated earlier and close down any noise libraries you might be using. The Chrome shader's Finish phase looks like this:

```
long EIShaderFinish(EIShaderFinishRec *theFinishPtr)
{
	/* Get rid of the shader's private data */

	theFinishPtr->HostFree(theFinishPtr->shaderData);

	return shaderNoError;
}  /* EIShaderFinish */
```

Finally, we come to the actual meat of the shader. The Shade phase is where the magic really happens. We're trying to make the EI equivalent of a Renderman shader so let's look at the Renderman source again.

```
surface
metallic()
{
    point Nf = normalize(faceforward(N, I));
    point V = normalize(-I);
    point R;        /* reflection direction */
    point Rworld;   /* R in world space */
    color Ct;
    float altitude;

    R = 2 * Nf * (Nf • V) - V;
    Rworld = normalize(vtransform("world", R));
    altitude = 0.5 * zcomp(Rworld) + 0.5;
    Ct = spline(altitude,
        BROWN, BROWN, BROWN, BROWN, BROWN,
        BROWN, BLUE0, BLUE1, BLUE2, BLUE3,
        BLUE4, BLUE5, BLACK);
    Oi = Os;
    Ci = Os * Cs * Ct;
}
```

The first thing we need is Nf. Nf is really a pixel surface normal i.e. a not just a polygon normal but one that is evaluated at each rendered pixel. It turns out that we've got this in the Pixel record stored in shadeNormalA, shadeNormalB, and shadeNormalC so we don't have to compute anything here.

V is the negative of I which is the direction that the camera is pointing. It turns out that the Pixel record also has this stored in shadeXN, shadeYN, and shadeZN.

Note that Renderman stores vectors in a data type called point. EI shaders break out the components of a vector into separate variables.

Now, we want to compute R. The following code calculates R:

```
dotProd =    (thePixel->shadeNormalA * Vx) +
             (thePixel->shadeNormalB * Vy) +
             (thePixel->shadeNormalC * Vz);

Rx = 2.0 * thePixel->shadeNormalA * dotProd - Vx;
Ry = 2.0 * thePixel->shadeNormalB * dotProd - Vy;
Rz = 2.0 * thePixel->shadeNormalC * dotProd - Vz;
```

This code is virtually identical to the Renderman code. The difference is that Renderman has certain operators for dealing with vector algebra such as Dot and Cross products. EI Shaders don't have these so you need to write your own.

Once we've got R, we need to transform it into World space. To do this, we need to apply the inverse of the camera's rotation matrix. The following code creates this matrix:

```
CopyMem(thePixel->shadeCameraInvMatrix, &cameraInvRotMatrix, sizeof(Matrix4));
cameraInvRotMatrix.m.m30 = 0.0;
cameraInvRotMatrix.m.m31 = 0.0;
cameraInvRotMatrix.m.m32 = 0.0;
```

Basically, we're copying the inverse camera matrix from the Pixel record and zeroing out any translation.

Now the EI Shader equivalent of:

```
Rworld = normalize(vtransform("world", R));
```

is:

```
IwX = Rx; IwY = Ry; IwZ = Rz;
TransformPoint(&IwX, &IwY, &IwZ, &cameraInvRotMatrix);
```

To compute the altitude value, we use the following code:

```
altitude = theShaderData->shaderAltitudeFactor * IwY + 0.5;
```

There are some differences between Renderman and EI here. First, the altitude needs to be related to the Y component of Rworld instead of the Z component. Second, we've added out own altitude gradient transition rate variable just for fun.

Making use of the color gradient spline is a little different. The Renderman code is:

```
Ct = spline(altitude,
        BROWN, BROWN, BROWN, BROWN, BROWN,
        BROWN, BLUE0, BLUE1, BLUE2, BLUE3,
        BLUE4, BLUE5, BLACK);
```

The function Spline is built into Renderman but EI Shaders have a similar C function in the ColorSpline module. Invoking this is done with the following code:

```
ColorSpline(altitude, 13, &theShaderData->chromeColors[0], &chromeColor);
```

The difference is that we tell the spline routine how many knots there are and the address of the first knot. Note that the knots must be stored in memory in order.

Once we have the chrome color, we need to do something with it. This is another significant difference between Renderman and EI. Renderman has very few surface attributes that you can access directly. EI allows you to access just about everything. In our version of the Chrome shader, we have the option to set surface or reflection colors. This is done with the following code:

```
if (BTST(theShaderData->shaderAffectFlags, 31)) {
	theShadePtr->shaderColor = chromeColor;
}

if (BTST(theShaderData->shaderAffectFlags, 30)) {
	theShadePtr->shaderReflection.r = chromeColor.r;
	theShadePtr->shaderReflection.g = chromeColor.g;
	theShadePtr->shaderReflection.b = chromeColor.b;
}
```

Note that the difference between the surface color and the reflection color is that the reflection color does not have an alpha component. Therefore, we must set the color component values individually.

The Renderman shader sets opacity as well as surface color. We've chose not to do this.

So there we have it. When you apply the Chrome shader to the Teapot for example, it looks like this:

You can pretty much use the Chrome shader as a starting point for other shaders. Just change a few components and off you go.