Electric Image Animation System

EIAS VERSION 9.0
LAYER SHADERS API

Table of Contents

http://eias3d.com

1.0 Layers in EIAS

The term “Layers” refers to additional files created by Camera at render time for each rendered frame. In most cases these files are images containing different parts/components of the main rendering, such as Specular, Luminance, Diffuse etc. However the layer’s content is not limited to imagery. Other kinds of data, usable for post-processing, can be stored as well: Z-distance, surface normals, UV coordinates and so on. By default, the rendered layer files are written into a Photoshop .psd file,created for each rendered frame.

The EI Layer architecture is open. By using the API, developers can create custom layers, write them into .psd standard output or into files of arbitrary formats. The API also allows developers to implement different EI post-processing effects (section 6).

Why do I need post-processing effects in EIAS if AE and other post-processing software already exist to do them?

It is a different kind of post-processing. By using this API, you can perform post-processing before the image is anti-aliased, e.g. at the sub-pixel level. Such effects can be hard to implement or unachievable at all in a post-processing application that operates with the final (screen), anti-aliased pixels.

2.0 Layer Shaders Overview

The Multi-Layers API is an extension to EI shaders API. So, a layers shader should be written generally in same way as a usual material shader. The difference is that a layers shader does not export *EIShaderShade* function. Instead it exports one or more of following functions:

```
EIShaderInitLayers
EIShaderFillLayers
EIShaderSlabLayers
EIShaderProcLayers
```

A layers shader should export *EIShaderInitLayers* function always. Otherwise user cannot apply it to layers shaders list in Render Dialog. All others functions are optional, but if a shader provides some of them, the corresponded flag should be set in *shaderFeaturesFlag* field at host call *EIShaderInformation*

shaderFillLayersFlag	// The shader is called after each sub-pixel shading to save layer’s information
shaderSlabLayersFlag	// The shader is called before and after every render pass
shaderProcLayersFlag	// The shader is called for post-processing

3.0 EIShaderInitLayers

3.1 Basic structures/callbacks description

```
SInt32 EIShaderInitLayers( EIShaderInitLayersRec * theHostInitLayersPtr );
```

The EIShaderInitLayersRec is decalred as:

```
typedef struct {
        void * shaderData;
        UInt32 theRenderPassID;
        EILayerCreateProc HostLayerCreateProc;
        EILayerEnumProc HostLayerEnumProc;
} EIShaderInitLayersRec, *EIShaderInitLayersPtr;
```

shaderData - the pointer to the shader's private data previously allocated at EIShaderInitialize

theRenderPassID – reserved, actually not used

HostLayerCreateProc – the layer's create callback

HostLayerEnumProc - the layers' enumerating callback

To create a layer the shader fills the *EI_LayerInfoRec* structure and call *HostLayerCreateProc.*

```
typedef SInt32 (*EILayerCreateProc)( EI_LayerInfoPtr );

typedef struct {
        UInt32 layerID;              // unique layer's ID
        UInt32 layerFlag;            // layer's flag (see "Layers Flags" above)
        UInt32 layerDataType;        // layer's data type (see "Layers Data Types" above)
        UInt32 layerDataSize;        // sub-pixel's data size in bytes (used for custom or
                                     user-defined layers)
        SInt32 layerContentID;       // layer's content ID (see "Layers Contents" above)
        uchar layerDefault[64];      // default sub-pixel's value block
        UInt32 layerParentID;        // ID of layers's parent (0 = none)
        UInt32 layerModePS;          // layer's Photoshop mode (add, multiply etc.)
        PStr63 layerName;            // layer's name
        UInt32 layerCreator;         // signature of layer's creator
        void * layerUserData;        // pointer to user-defined data
        SInt32 layerTransRange[2];   // range of phong transparency passes ("0" means all
        further passes)
        UInt32 layerReserved[8];     // reserved, must be zero
} EI_LayerInfoRec, *EI_LayerInfoPtr;
```

If HostLayerCreateProc returns 0 (no error), the layerID is filled by unique ID of created layer. Shaders should save this ID into their private data for further manipulations with the layer.

The created layer will be saved in output PSD file if a shader sets *LFLAG_LAYER* bit of *layerFlag* field.

3.2 Creating standard layers

A creating a layer can be the only task that a layer shader performs. The render engine does all the of the rest of the work automatically if:

layerDataType either one of the standard:

```
LDATA_ARGB8
LDATA_RGB8
```

layerContentID is one of standard:

```
LCONTENT_CUSTOM        // custom (unknown) content
LCONTENT_DIFFUSE       // material's diffuse
LCONTENT_LIGHTS        // sum of all lights values
LCONTENT_LUMINANCE     // luminance
LCONTENT_SPECULAR      // specular
LCONTENT_AMBIENT       // ambient
LCONTENT_REFLECTION    // material's reflections summary
LCONTENT_REFL_BM       // bitmap reflections
LCONTENT_REFRACTION    // RT refractions
LCONTENT_TRANSPARENCY  // transparency
LCONTENT_FOG           // fog
LCONTENT_SHADE_COLOR   // final shaded (non-clamped) color
LCONTENT_COVERAGE      // sub-pixel's coverage (ARGB_Real)
LCONTENT_GROUP_ID      // group ID assigned at render time
LCONTENT_UV            // UV coordinates as ARGB_Real (a=1 indicates "natural" UV's)
LCONTENT_PIXEL_MASK    // a packed array of bits representing the curent facet's
                       visibility at each sub-pixel
LCONTENT_MOTION_BLUR   // motion blur x and y vectors
```

3.3 Creating custom layers

The Multi-Layers API supports the creation of other layers besides standard ones listed above. The layers can be saved in the PSD format or not, filled by host and/or by shaders etc. The host guarantees the layer's data storage for each rendered sub-pixel and their availability for further manipulation.

The layer can have no pixel data (*layerContentID* = *LCONTENT_CUSTOM* and *layerDataSize* = 0). Nevertheless the layer will be created. It tells host the layer service should be active regardless if there is PSD output or not.

3.4 Applying layers to specific Phong transparency passes

The *layerTransRange* field tells the host to make a layer active for some Phong Transparency passes only.

The Camera renders directly visible surfaces first (it's the first Phong transparency pass). If other objects are found behind Phong- transparent objects, they are rendered again (second/next passes) and rthe esult is combined with the previous one. This process is repeated until all is done.

If *layerTransRange[0]* = 0, the layer is active for any pass. Otherwise it defines the first pass the shader should be applied.
If *layerTransRange[1]* = 0, the layer is active for all passes starting from *layerTransRange[0]*.
Otherwise it defines the last pass the shader should be applied.

If a layer is inactive at some passes, the layer's content remains unchanged for each sub-pixel.

4.0 EIShaderFillLayers

```
SInt32 EIShaderFillLayers( EIShaderShadeRec *theShadePtr );
```

The EIShaderFillLayers call allows shaders to read/write layers' data of the shaded sub-pixels. Unlike *EIShaderShade*, the *EIShaderFillLayers* is called after the sub-pixel is fully shaded.

The *EIShaderShadeRec* structure is expanded with a new field HostLayerShade, its type *EILayerShadeProc* is declared as:

```
typedef SInt32 (*EILayerShadeProc)(SInt32 theCommand, uInt32 theID, ARGB_Real * color);
```

The format is:

```
theCommand = LSHADE_CMD_GET_LAYER     // get the actual layer’s content
       theID: ID of layer to interest (previously returned by EIShaderInitLayers)
       color: on output filled with layer’s color (actually shaded pixel)

theCommand = LSHADE_CMD_GET_PROPERTY  // get the property color
       theID: one of standard layerContentID constants
       color: on output filled with color of the property (actually shaded pixel)

theCommand = LSHADE_CMD_SET_LAYER     // set the layer’s content
       theID: ID of layer to set (previously returned by EIShaderInitLayers)
       color: input color to set

theCommand = LSHADE_CMD_SET_PROPERTY // set the property color
       theID: one of standard layerContentID constants
       color: input color to set

theCommand = LSHADE_CMD_GET_PREV      // get the previous layer’s content
       theID: ID of layer to interest (previously returned by EIShaderInitLayers)
       color: on output filled with layer’s color before actual shaded pixel is applied
```

A shader can fill the created layer directly via the *LSHADE_CMD_SET_LAYER* command and/or byusing the render properties mechanism.

Each of the standard *layerContentID* constants is associated with some “render *property*” attribute that can be get/set with the *HostLayerShade* callback function. The “property” is a value that the host is aware of and can calculate for the shader on its request.

5.0 EIShaderSlabLayers

```
SInt32 EIShaderSlabLayers( EIShaderSlabLayersRec * theHostSlabPtr );
```

This function allows shaders to write custom output files (besides the standard PSD) and to prepare the post-processing phase. The EIShaderSlabLayers is called by the host each time a new *Render Slab* starts or ends and each time a new Phong transparency pass starts or ends. As you know, for large renders, the host can process the image step by step sequentially. The *Render Slab* is the actual portion of the whole rendered screen that is being processed currently.

The EIShaderSlabLayersRec is declared as:

```
typedef struct {
        void * shaderData;                 // The pointer to the shader's private data

        L_Rect shadeRect;                 // The rendering rectangle
        L_Rect shadeSlabRect;             // The rendering slab rectangle
        L_Rect shadeImageRect;            // The whole rendering image rectangle
        L_Rect shadeSlabImageRect;        // The slab rendering image rectangle

        float shadePixelWidth;            // The antialiasing pixel width
        float shadePixelScale;            // The zoom factor
        float shadePixelRatio;            // The y/x pixel ratio
        float shadeOffsetX, shadeOffsetY; // The x and y image offset
        UInt32 shadePerspectiveFlag;      // 1 for perspective rendering

        UInt32 shaderPassStartFlag;       // 1 = pass start, 0 = pass end
        UInt32 shaderPassNomer;           // The nomer of phong transparency pass,
                                          // last pass = bit 31 is set

        char * shaderFullPSDName;         // The full name of output PSD file (C string)
        char * shaderSetName;             // The name of layer's set the shader is applied to
                                          // (C string)
        char * shaderFileName;            // The full name of the shader file (C string)

        EILayersReadProc HostLayersReadProc;         // The layers read callback
        EILayersWriteProc HostLayersWriteProc;       // The layers write callback

} EIShaderSlabLayersRec, *EIShaderSlabLayersPtr;
```

The rendering rectangles (shadeRect and shadeSlabRect) are passed in sub-pixels. The image rectangles (shadeImageRect and shadeSlabImageRect) are passed in final screen pixels.

The strings *shaderFullPSDName*, *shaderSetName* are passed to construct the name of the custom output file regardless of whether or not a PSD file is being created. Shaders should create their custom files in same folder as the *shaderFullPSDName* points to. During network rendering, it is guaranteed that files will be transferred from slave(s) to master.

The string *shaderFileName* gives the shader the ability to find their “rsc” files and to report advanced error information. It is recommended to use this string because the host cannot know all the details about what has happened in the shader’s thread.

Shaders should not modify any of the passed strings.

The callbacks *HostLayersReadProc* and *HostLayersWriteProc* perform layer reading and writing. The typedef(s) are:

```
typedef SInt32 (*EILayersReadProc) ( L_Rect * theRect,
        UInt32 theLayersCount,
        UInt32 * theLayersID,
        uchar * theData );

typedef SInt32 (*EILayersWriteProc)( L_Rect * theRect,
        UInt32 theLayersCount,
        UInt32 * theLayersID,
        uchar * theData );
```

theRect – the rectangle area (in sub-pixels) to read/write data. The rectangle should be within the bounds of the actual *shadeSlabRect*

theLayersCount – the number of layers to read/write

theLayersID – the pointer to an array of layers IDs. The count of elements should correspond to *theLayersCount*

theData – the data to read/write. The data should be organized for writing (and expected for reading) as pixel's blocks, i.e. the first are data for the first pixel (left top in *theRect*), second are next pixel in row etc. Inside pixel's block, the layersdata follows in the order specified by *theLayersID* array. The data size of each layer should correspond to the data size specified when the layer was created. No pad/fill bytes should be added/expected.

6.0 EIShaderProcLayers

```
SInt32 EIShaderProcLayers( EIShaderProcLayersRec *theHostProcPtr );
```

The EIShaderProcLayers is a function called to render every additional (post-processed) layer that can be added to the output PSD file.

6.1 Post-Processing phase

The host detects a post-processing activity if one or more layer shader specifies *LFLAG_PROCESS* for one or more created layers and these shaders export *EIShaderProcLayers* together with the *shaderProcLayersFlag* set.

The post-processing phase is started after the whole render screen is finished and the output image is written. Then the host creates a series of render processes for each layer marked for postprocessing. Don't be confused, it does not mean the host loads again and again all of the geometry, lights etc. The additional render passes are very simple: the host just calls the shader(s) to fill the rendered screen. When done, the host automatically performs anti-aliasing and saves the result as a PSD layer until all the post-processed layers are done.

The typical post-processing shader scheme is:

> Create the needed channels and fill them with the appropriate data (by using *EIShaderInitLayers* and *EIShaderFillLayers*);
>
> Save filled layers in a temp file by using the *EIShaderSlabLayers* service;
>
> When *EIShaderProcLayers* is called, load previously saved layers from the temp file, combine them and use the *HostLayersWriteSlabLineProc* callback to pass the result to the host for anti-aliasing and storage in PSD file.

6.2 EIShaderProcLayersRec

```
    typedef struct {

        void * shaderData;                     // The pointer to the shader's private data

        L_Rect shadeRect;                      // The rendering rectangle
        L_Rect shadeSlabRect;                  // The rendering slab rectangle
        L_Rect shadeImageRect;                 // The whole rendering image rectangle
        L_Rect shadeSlabImageRect;             // The slab rendering image rectangle

        float shadePixelWidth;                 // The antialiasing pixel width
        float shadePixelScale;                 // The zoom factor
        float shadePixelRatio;                 // The y/x pixel ratio
        float shadeOffsetX, shadeOffsetY;      // The x and y image offset

        UInt32 shadePerspectiveFlag;           // 1 for perspective rendering

        UInt32 shaderLayerID;                  // ID of the post-processing layer
        SInt32 shaderTransparency;             // 1 if there is phong transparency

        PStr255 shaderStatusString;            // The status string

        EILayersWriteSlabLineProc HostLayersWriteSlabLineProc; // The write callback

    } EIShaderProcLayersRec, *EIShaderProcLayersPtr
```

Even the rendering rectangles (*shadeRect* and *shadeSlabRect*) and the image's rectangles (*shadeImageRect* and *shadeSlabImageRect*) look same as in *EIShaderSlabLayersRec*, but don't forget – they belong to the other (post-processed) render process.

The *shaderLayerID* tells the shader ID of a single layer that is being post-processed now.

The *shaderStatusString* should be modified by the shader at the start of post-processing work. Thehost shows this status string in the Camera window.

The *HostLayersWriteSlabLineProc* passes transfers to host the shader's data. The typedef is:

```
    typedef SInt32 (*EILayersWriteSlabLineProc)(SInt32 theLine, RGBA * theColor, RGBA * theAlpha );
```

Where:

theLine – index of the line passed to the host. It should be in the bounds of the *shadeSlabRect* rectangle;

theColor and theAlpha – the array of color and alpha values respectively. The count of the array's elements should correspond to *shadeSlabRect.width.* The values of *theAlpha* will be ignored (and *theAlpha* can be null) without Phong transparency (*shaderTransparency* = 0)

Each time a shader calls *HostLayersWriteSlabLineProc*, the host updates the status information in the Camera window. If *HostLayersWriteSlabLineProc* returns an error, the shader should stop work and return control to the host. It means the user cancelled the operation.