EIAS Version 9.0
Plug-Ins API

Table of Contents

Electric Image Animation System

Electric Image Animation System

1.0 Introduction - Overview of the Plug-In Toolkit

EIAS plug-ins are dynamic libraries that expand the host's (Animator and/or Camera) functionality. Since a plug-in may be developed by 3rd party companies and individuals, features which are needed for specific purposes can be added whenever they are needed. There are two kinds of EIAS plug-ins:

- The Model plug-in can create vertex and facet geometry either algorithmically or by referencing groups which have been linked to them in Animator. By convention a model plug-in file should use the .plm file name extension.

- The Light Flare plug-in allows special visual effects to be added to an image after it has been rendered. By convention a flare plug-in file should use the .plf file name extension.

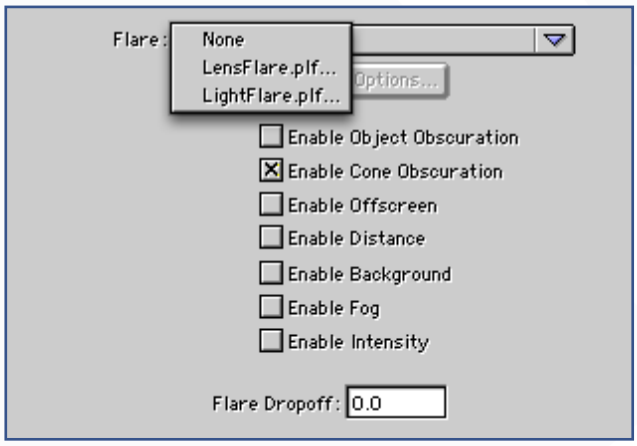
1.1 Plug-in installation and Interaction with the Host

No specific installation is required for a plug-in. The plug-in's libraries file (together with its .rsc file) should be placed into the "EI Sockets" folder that is located inside the home/root EIAS directory (where EIAS main applications "Animator" and "Camera" are installed).

When launched, the Animator automatically scans the "EI Sockets" folder for all available model plug-ins and adds them to the project's plug-ins list. As it's described below, a plug-in should have an exported "Main" function to be recognized by Animator and Camera.

When the user selects an item from the Plug-in menu, Animator calls the plug-in's library to display its user interface.

The Light Flare plug-ins behavior is similar but they appear in the Light Info window (Flare Tab) as shown below.



All plug-ins (Model and Flare) must define its "Main" exported function to be recognized/accepted by the host.

1.1 Plug-in Examples

Let's consider the functionality of some plug-ins shipped with EIAS.

Particle — A model plug-in which generates geometry algorithmically. It creates a series of particles. For each particle it computes its position at the current frame by taking into consideration the time the particle was created, the particle's initial direction and velocity, the force of gravity and the life span of the particle. It then generates a vertex and a facet for the particle and sends them to the host where they are displayed (Animator) or rendered (Camera).

MrNitro — An example of a plug-in which generates geometry by referencing the groups which are linked to it in Animator. A linked group is referred to as a child group. MrNitro computes the radius of a simulated explosion at the current frame in the animation. It then determines which facets of its child groups are inside the radius. Facets outside the blast radius are simply copied and sent back to the host without modification. Facets partially or completely inside the blast radius are broken up into smaller fragments. Each fragment's position and rotation is calculated based upon the force and speed of the shock wave, rotation speed, gravity, air resistance and turbulence. Each fragment is then sent to the host be displayed or rendered. It is possible that hundreds of fragments or more may be generated by a single facet in a child group. If instead of rendering the fragments, the host wrote each fragment to a data file, even a short animation could generate hundreds of millions of fragment facets that would occupy many gigabytes of disk storage. This demonstrates how model plug-ins can save time and disk space by generating geometry as it is being displayed or rendered.

LensFlare — This plug-in contains a database of ring and spike elements. Each ring element has a record that describes its size, location and an array of color values. LensFlare uses the ring element record to generate a circular image of the ring that is then added to the flare plug-in's frame buffer. By creating a series of these rings in different colors along the diagonal line between a light source and the center of the image, a lens flare effect is created. The spike elements are just thin lines drawn radially from the light source's position. Each spike element also has a record in LensFlare's resource file containing the spikes length, direction, position and color table. In addition to sending the LensFlare plug-in the information it needs to draw the effect, Camera can also determine the visibility of the flare's light source and will turn off the flare plug-in if it is obscured by another object in the frame.

1.2 Software Licensing

Developers can release their plug-ins (commercial or freeware) in any way they want. No license from EIAS3D is required for the use of plug-in libraries and headers from the SDK.

1.3 Plug-In Hosts

The term "host" refers to an application that calls the plug-in. There are two hosts:

EIAS Animator (Animator). This is the main user interface application. Animator allows users to design animations by adding objects to a project list and animating a wide range of values for each object. Animator adds plug-ins to its project list and allows users to interact with the plug-in's user interface and to preview the results before rendering. Animator also keeps track of any animation channels created by the plug-ins.

Camera. This application renders control files it receives from Animator. It has almost no user interface of its own. It creates animations in EI's own multi-frame Image file format. Camera calls each plug-in at every rendered frame in the animation in order to generate model data or light flare effects.

IMPORTANT:: a plug-in can behave differently depending on the host where the plug-in is running. Many host

calls (described below) are available in Animator only. For example, animation channel management is available only in Animator. On other hand, some calls can be meaningful in Camera only. Don't be confused, it's normal/ expected, animation and render are different functions.

1.4 Basic Data Types

Generally developers need to use standard C/C++ types. The only exceptions are the 'long' and 'unsigned long' types.

IMPORTANT: Avoid using long (unsigned long) types, they cause problems because these types are 8 bytes long on Macintosh 64 bit but 4 bytes long on Windows and Macintosh 32 bit.

The following types, defined in file PlatformTypes.h, have the same size and range of values on both 32/64 bits, for the Macintosh and Windows platforms:

SInt32 - 4 bytes signed integer on all platforms
UInt32 - 4 bytes unsigned integer on all platforms

The following types, defined in file PlatformTypes.h, have mutable size depending on 32 or 64 bits is in use:

slong - 4 bytes signed integer in 32 bits, but 8 bytes signed integer on 64 bits
ulong - 4 bytes unsigned integer in 32 bits, but 8 bytes unsigned integer on 64 bits

There are very few cases when you need to use these types.

1.5 Using C or C++ for Development

For maximum compatibility the whole plug-ins' SDK is designed to use only the C language. However, there are no obstacles to using C++ as well. Few code examples are given in C++.

2.0 The Communications Process

2.1 The Plug-In's 'Main' Function

An EIAS plug-in can be thought of as a simple program. The host application can be thought of as a mini operating system for the plug-in. To be recognized by the host and to receive the host's commands, a plug-in should export a function named *Main* (or *main*). No other exports are necessary. The plug-in is "launched" every time the host sends it a command and the Main function receives control. The function's declaration is as follows:

```
SInt32 Main( UInt32 theCommand, SInt32 theCommandRecSize, void * theCommandRec );
```

The passed arguments are:

theCommand - this parameter allows the plug-in to determine which operation the host is expecting it to perform. This parameter is a four character token. There is a unique token for each command the plug-in may accept from the host. Upon entry to the plug-in's main routine, the command token is examined and a sub function is then called based upon its value. The following sample code shows how a simple plug-in would handle an incoming command token:

```
SInt32 Main( UInt32 theCommand, SInt32 theCommandRecSize, void * theCommandRec );
{
	SInt32 theError = noErr;
	switch (theCommand) {
		case pluginInitialize:
			theError = DoInitialize(theCommandRecSize, theCommandRec);
			break;

		case pluginInterface)
			theError = DoInterface(theCommandRecSize, theCommandRec);
			break;

		case pluginModelSetup:
			theError = DoModelSetup(theCommandRecSize, theCommandRec);
			break;

		case pluginModelGenerate:
			theError = DoModelGenerate(theCommandRecSize, theCommandRec);
			break;

		/* others host's commands here */

		default:
			theError = pluginCommandErr;
	}
	return theError;
} /* Main */
```

The plug-in command constants such as *pluginInitialize* are defined in the file *PluginClient.h*. When a command is passed to the plug-in which the plug-in does not support, the plug-in should return the result code *pluginCommandErr* to tell the host that the command could not be accepted by the plug-in. The host software should not report an error to the user in this case unless the requested command was essential to the operation of the host. The required command set may change in the future but for now only the Initialize and *Generate* commands are considered essential for correct plug-in operation. A plug-in author should try to support as many commands as possible for future compatibility and smooth plug-in operation.

theCommandRecSize - This parameter is 32-bit integer which contains the size of the record that theCommandRec points to. It is used to help validate the command's parameter record. The following sample shows how a plug-in would validate the parameters for a model setup command:

```
SInt32 DoModelSetup( SInt32 theCommandRecSize, void * theCommandRec )
{
	SInt32 theError = noErr;
	PluginModelSetupPtr theParameters = (PluginModelSetupPtr) theCommandRec;
	if (theCommandRecSize < sizeof(PluginModelSetupRec)) {
		theError = pluginParameterErr;
	else {
		/* Setup the model plug-in */
	}
	return theError;
} /* DoModelSetup */
```

By comparing the parameterSize against the size of PluginModelSetupRec, the plug-in can determine if enough data was sent for the pluginModelSetup command. The incoming record size is only checked to see if it is smaller than the size of the expected type. This means the plug-in would allow records of a larger size than expected. In future versions of the host software, record definitions may be expanded to include other parameters and information. Since all parameters that are expected to return information are initialized to their defaults by the host, earlier version plug-ins should remain compatible with later versions of the host software.

theCommandRec - is a generic pointer to a plug-in command record. Each command has its own record. A plug-in must examine the command token before it is able to examine the contents of the command's record. For example, here are two typical plug-in commands that the host will make to a plug-in along with their respective records: (An entire listing of all the command and record types available, including these, can be found in *PluginClient.h*)

```
#define pluginCheck 'CHEK'
typedef struct {
	PluginInfoRec theInfo;
	UInt32 theCommand;
} PluginCheckRec, *PluginCheckPtr;

#define pluginModelSetup 'SETM'
typedef struct {
	PluginInfoRec theInfo;
	SInt32 theFrameIndex;
	double theFrameTime;
	double theFrameTimeDelta;
	Matrix4 theCameraMatrix;
	Matrix4 theCameraRotMatrix;
	Matrix4 theCameraBlurMatrix;
	Matrix4 theGroupMatrix;
	Matrix4 theGroupRotMatrix;
	Matrix4 theGroupBlurMatrix;
} PluginModelSetupRec, *PluginModelSetupPtr;
```

2.2 The PluginInfoRec structure

Every plug-in command contains a plug-in information record. This record always follows first in any command data structure. The host uses this record to send the size and pointer for the ControlData and PluginData records to the plug-in.

```
typedef struct {
        Ptr theControlData;           /* Pointer to the plug-in's ControlData record */
        SInt32 theControlDataSize;    /* Size of the plug-in's ControlData record */
        Ptr thePluginData;                  /* Pointer to the plug-in's PluginData record */
        SInt32 thePluginDataSize;           /* Size of the plug-in's PluginData record */
        EIPluginRoutine theHostProc;  /* The host application's command routine */
        Ptr theHostReference;         /* The host application's reference */
} PluginInfoRec, *PluginInfoPtr;
```

Every time a plug-in is called by the host, the plug-in should examine PluginInfoRec to locate and validate its private data stored during the previous call and pointed to by *theControlData* and *thePluginData*. A plug-in is responsible for the allocation of both data blocks that should be performed via a *pluginAllocate* call to the host. After the records are allocated, the plug-in should initialize their contents. When the next command is sent to the plug-in, the data records will have the same size and contents as they did when the plug-in completed the previous command. Plug-ins should not assume the blocks addresses are unchanged from one host's call to another.

2.2.1 The ControlData Record

This record is used to store information that must be saved with the project file for use the next time the project is opened. It is also used to pass plug-in information from one host to another. Most commonly, the information stored in the control data record was entered by the user in the plug-in's interface dialog box.

In the case of the Mesh plug-in, for example, the mesh type and density is entered by the user in the plug-in's interface dialog and then stored in the control data record. This control data record is saved in the project to which the plug-in has been added. When Camera is launched to render an image or animation, the control data record is written to Camera's control file. The same control record will be read by Camera and passed to the plug-in during the rendering process so that the correct mesh is generated at each frame of the animation. To keep the size of the project and Camera control files to a minimum, the control data record should be made as small as possible.

See also chapter "Memory Allocation and Initialization" below for more details.

2.2.2 The PluginData Record

Originally this memory block was intended to store information that the plug-in only needs during a series of commands within a single host application. (e.g. *thePluginData* content is not stored from one project to another). But now this usage is obsolete and supported for backward compatibility only. Instead, the Plug-ins should use other memory allocation methods (see details in section 3).

2.2.3 Using theHostReference

The host passes *theHostReference* in *PluginInfoRec* structure. This pointer "identifies" the plug-in in the actual project. For example, two or more instances of the same plug-in can be added to the project and theHostReference is used to differentiate between them. The same plug-in's code and same host commands are called for all of them, but every plug-in receives its own (unique) *theHostReference* at every call. Also, when a plug-in calls the host, *theHostReference* should be written in the plug-in command record (see below).

Electric Image Animation System

2.2.4. Calls from the Plug-in to the Host

When the plug-in processes a command, it may need to request services from the host. These requests consist of commands to the host, just as the host sends commands to the plug-in. These services cover a multitude of activities which include allocating memory, creating or deleting groups, adding vertex or facet data, and others. All host command calls are of the form:

```
SInt32 Host( UInt32 theCommand, SInt32 theCommandRecSize, void* theCommandRec );
```

Here *theCommand* and *theCommandRecSize* are the host command token and the size of the host command record respectively. *theCommandRec* is the address of the host command record that should be one of those defined in *PluginClient.h* and the corresponding *theCommand* token. Example:

```
#define hostAllocate 'ALOC'
typedef struct {
        Ptr theReference;               /* Sends: the private data used by host callbacks */
        Ptr theControlData;             /* Returns: the new control parameter data location */
        SInt32 theControlDataSize;      /* Sends: the requested new control parameter data size */
        Ptr thePluginData;              /* Returns:the new plug-in private data location */
        SInt32 thePluginDataSize;       /* Sends: the requested new plug-in private data size */
} HostAllocateRec, *HostAllocatePtr;
```

The plug-in is responsible for initializing the command record with correct data. Note that the field *theReference* should be the *theHostReference* value, passed by the host to the plug-in in *PluginInfoRec.* In this example, the call to the host would look like this:

```
theError = theInfo.theHostProc(hostAllocate, sizeof(HostAllocateRec), &theHostAllocate);
```

The host's command function pointer is passed to the plug-in in the plug-in's theInfo record. If the host returns an error code to the plug-in, the plug-in should abort its current command and return the error code back to the host. If pluginNoErr is returned (0), the plug-in can use the records referenced by *theControlData* and *thePluginData*.

2.3 Communication Example

What follows is a procedure called ModelGenerate that summarizes and demonstrates the steps described in this section. This procedure really does nothing useful since the pointer to the new group isn't used for anything, but it does illustrate the process. It would be called by the plug-in's main routine whenever theCommand token it received was pluginModelGenerate. In the example below, the record of type HostModelAddGroupRec is initialized, the hostModelAddGroup command is sent to the host, and the pointer to this new group is retrieved from the record if pluginNoErr is returned.

```
SInt32 ModelGenerate (SInt32 theCommandRecSize, void* theCommandRec)
{
        Ptr theNewGroup;
        PluginModelGeneratePtr thePluginModelGeneratePtr = (HostModelGeneratePtr)theCommandRec;
        HostModelAddGroupRec theHostModelAddGroup;
        SInt32 theError = pluginNoErr; /* Initialize theError variable */

        /* Check to see that plug-in command record is of the expected size */
        if (theCommandRecSize < sizeof(PluginModelGenerateRec))
                theError = pluginParameterErr; /* If it's not the right size, return an error */
        else {
```

```
                /* Initialize the host command record's fields */
                theHostModelAddGroup.theReference = thePluginModelGeneratePtr->theInfo.
theHostReference;
                theHostModelAddGroup.theGroupTemplate = nil;
                theHostModelAddGroup.theGroupVertexFlags = 0;
                theHostModelAddGroup.theGroup = nil;

                /* Make the call to the host */
                theError = thePluginModelGeneratePtr->theInfo.theHostProc(hostModelAddGroup,
                                        sizeof(HostModelAddGroupRec), &theHostModelAddGroup);

                /* If no errors occurred, extract the information that the Host 'returns' */
                if (theError == pluginNoErr )
                        theNewGroup = theHostModelAddGroup.theGroup;
        }
        return theError;
} /* ModelGenerate */
```

3.0 Memory Allocation and Initialization

3.1 Working with PluginControlData

As it was described above, the *PluginControlData* is a memory block stored by host with project file. This block should be allocated only via *hostAllocate* call to host.

It's important to understand that host provides *PluginControlData* storage as "memory dump", no any Endians or others conversions performed. Developer is responsible for these data parsing and validating.

3.1.1 Simple Use of PluginControlData

The simplest way is just to fill *PluginControlData* with plug-ins's data structure.

```
typedef struct {
        UInt32          thePluginID;            /* Internal plug-in's ID */
        UInt32          thePluginVersion;       /* Internal plug-in's version */
        char            theFlag;                /* User-defined data */
        double          theValue;               /* User-defined data */

        /* Others data fields */

} MyPluginControlDataRec, * MyPluginControlDataPtr;
```

However, this way has a defect. A directly stored structure is often incompatible between different platforms and even compilers. For example if *theValue* field stored on Mac platform, it will NOT have a correct value in Windows platform because its offset (from structure begin) is different. The structure's size also can be different. To avoid this problem use explicit alignment.

```
#pragma pack(push, 4)
typedef struct {
        UInt32          thePluginID;            /* Internal plug-in's ID */
        UInt32          thePluginVersion;       /* Internal plug-in's version */
        char            theFlag;                /* User-defined data */
        double          theValue;               /* User-defined data */

        /* Other data fields */

} MyPluginControlDataRec, * MyPluginControlDataPtr;
#pragma pack(pop)
```

Note: do NOT align data on 2 or 1 byte boundary, it can degrade performance dramatically.

3.1.2 Endian Issues

Plug-ins should not assume the data is already stored in the Endian of actual machine. The necessary conversion should be provided by the plug-ins. Example:

```
#define myPluginID      'PLUG'
#define convertOk        0
#define convertErr      -1

SInt32 ValidateMyControlData( MyPluginControlDataRec * theData )
{
        /* Check if data are already in actual Endian */

        if (theData->thePluginID == myPluginID)
                return convertOk;

        /* Check if data are in reverse Endian */

        if (ReverseLong(theData->thePluginID) == myPluginID) {

                /* swap Endian */

                FixField(theData->thePluginID);
                FixField(theData->thePluginVersion);
                FixField(theData->theFlag);
                FixField(theData->theValue);

                /* Convert all others data fields  */

                return convertOk;
        }

        /* Otherwise data are invalid  */

        else
                return convertErr;
}       /* ValidateMyControlData */
```

3.1.3 Handling Complex PluginControlData Structures

The approach described above works fine for simple structures, but is not very suitable for more complex data,

such as variable size arrays, strings and data pointers. In this case, developers can use *PluginControlData* block in the role of a *memory stream.* The typical approach is:

- Allocate the plug-in's data structures dynamically, by using hostMemAllocate command;

- When the host's command arrives, read *PluginControlData* content into the allocated plug-in's data. This reading is similar to file I/O: data is read field by field with necessary Endian conversion;

- When the plug-in's data is modified and should be stored, a plug-in writes the data into the*PluginControlData* block. Same as reading, writing is performed in *stream style.*

Note that only few of the host calls require *PluginControlData* reading or writing. In most cases it's enough to validate the *theHostReference* pointer.

This approach requires a bit more effort but allows the use of data of any complexity. See the *ColorMesh* plug-in's example from this SDK for details.

3.2 Dynamic Memory Allocation

Normally plug-ins should allocate/release dynamic memory via *hostMemAlloc/hostMemFree* calls to to the host.

The correct memory allocation strategy is to allocate data in chunks, portion by portion. For example, if you need to allocate a huge image, allocate it line by line, one memory block per line.

You can use different sizes of blocks. For example, 1K, 4K, 64K, 1MB, 4MB - all these sizes are acceptable/valid but they should not be exactly aligned on 1K or 1MB bounds. However an allocation of 512MB or larger blocks isn't recommended and can cause potential problems. It is even worse to allocate "a large block as needed". Keep in mind that some operating systems will not allow you to allocate a single block larger than 1GB, no matter how much physical RAM is installed. Consider how to subdivide such large data sets into pieces and allocate them part-by-part.

In other hand, do not make the mistake of creating an algorithm that allocates memory a few bytes at a time. This will degrade performance. Operating systems require a significant amount of time to allocate memory. For example: allocating one 1MB block of memory may be hundreds of thousands of times faster than allocating 250,000 4-byte blocks.

3.2.1 Using standard C memory allocation (new/delete or malloc/free)

The user is responsible for assigning an amount of RAM to use for rendering (in Camera's settings). The *hostMemAlloc* fails if this limit is exceeded. Using standard C memory allocation: new/delete or malloc/free is not disabled but keep in mind: by this method you can allocate much more memory than was assigned for rendering. Although the OS can give you more memory, eventually the consequences can be negative.

3.2.2 Memory allocation in 64-bits

It is important to understand that the above common memory allocation rules are the same for 32-bit and 64-bit modes. The only difference is the 2GB barrier. With 64-bits, the address space is about 150 TB (terabytes). However, don't rush to assign a huge amount of RAM to Camera. If, for example, you allocate 8 GB with only 4 GB of physical RAM, the result will be a dramatic slowdown because the OS swaps data between RAM/disk and no number of processors will make it faster.

Electric Image Animation System

4.0 Frame-to-Frame Data Storage

Plug-ins that create complex simulations may need to keep state information from past frames in order to generate data during later frames. An example of this is a particle simulator plug-in that can leave trails of particles behind as its location is animated.

In order for the plug-in to keep state data from frame-to-frame, it must create a temporary data file. The plug-in should create this file if it does not already exist. At each frame, the plug-in should write all necessary data to the temporary file in order to use it during the following frames. Camera will delete the temporary file automatically before it quits if the file's creator is "EIAR", its type is "EIDT", the file's name ends in ".temp", and it is located in the temporary folder whose path can be obtained via the EI_GetTemporaryDirectory API function..

With network rendering (Renderama), different frames (or even different parts of one frame) can be rendered on different machines and the render order is unpredictable. Therefore in this case there is no way to accumulate data in a temporary file. A solution to this problem is to give the user the ability to pre-calculate the data for all the frames to be rendered and store them in file. Then user can copy the file to each render slave.

For plug-ins that perform long and complex calculations, the temporary files can also be used to optimize render speed. Keep in mind that the same plug-in can be called several times during the rendering of a single frame, but in different render phases (such as for each buffer shadow, each glow pass and so on). In this case, the plug-in does not store data frame-by-frame but overrides the temporary file at each new frame. Typically, a plug-in stores render time in file and validates it.

The developer must take care of unique temporary file names in order to prevent them from overwriting one another. Typically the file name is constructed by using the following information:

- Running process unique ID (obtained by native OS calls);

- Plug-in's instance unique ID (typically created in Animator);

5.0 User Interface

The host will send a *pluginInterface* command to the plug-in when the user wishes to edit the plug-in's parameters. This command is sent by Animator but not by Camera. Animator sends this command when the plug-in is first added to the project or when the user clicks the Options button in the plug-in Group Info window or Light Info window.

5.1 Modal Plug-In's UI

In most cases, the plug-in responds by displaying a modal dialog box and accepting user input. The standard and recommended approach for plug-ins is to use the cross-platform EI UI API to load the dialog from the plug-in's resource (.rsc) file, display it, and to handle interaction with user. See the *EI UI API.pdf* for details.

Using a native UI instead is not disabled but it can require enormous efforts from the developer to provide the necessary UI functionality on both the Mac and Windows platforms.

5.2 Modeless Plug-In's UI

It's possible to use a modeless UI. In this case, the plug-in should call the EI_ShowDialogModeless API function to display a modeless dialog and return control to the host immediately. The host does not close the modeless dialog automatically, therefore the plug-in is responsible for hiding and destroying it when the user closes the plug-in's dialog or when the plug-in is going to be removed from the project (e.g when the *pluginFinish* host's command is received).

Typically a modeless UI is best used with plug-ins that use the *pluginProjectInfo* command (described below) to modify Animator's animation channels.

Electric Image Animation System

6.0 The Model Plug-in Interface

6.1 Data Structures for Model Objects

The purpose of a model plug-in is to create new model data and/or to modify model data in its child groups. The model data is comprised of one or more groups. Each group contains a list of vertices, a list of facets and miscelaneous information: shading, texture mapping, and reflection mapping records. A facet is a structure containing between one and four vertices and a color. A facet may represent a point, line, triangle or convex quadrangle. The facet vertices index the group's vertex list. `All` vertex indices are one based.

```
typedef struct {
        ARGB    theColor;
        short   theNumVertices;
        SInt32 theVertices[4];
} Facet;
```

A vertex always contains a three dimensional position and may also contain a color, surface normal, texture position, and motion blur position. Each group has a flag for each optional value to determine if it is to be computed and stored at each vertex.

```
typedef struct {
        ARGB theColor;
        DCoordinate theNormal;
        DCoordinate thePosition;
        DCoordinate theBlurPosition;
        DCoordinate theTexturePosition;
} Vertex;
```

Colors are represented by a union containing a 32 bit integer and four-byte channel values for alpha, red, green and blue.

```
typedef union {
        UInt32 colorValue;
        struct {
                Byte a, r, g, b; /* alpha, red, green, and blue */
        } argb;
} ARGB;
```

Positions and normals are represented as 64-bit double precision X, Y and Z coordinates.

```
typedef struct {
        double x, y, z; /* x, y and z coordinates */
} DCoordinate;
```

During the Generate command, a simple plug-in, such as Particle, can create model data by sending a series of hostModelAddVertex commands and then sending a series of hostModelAddFacet commands. A more complex plug-in, such as Mr. Nitro™, will loop for each of its child groups creating a new group with the hostModelAddGroup command, copying and modifying the vertex and facet data with the hostModelGetVertex and hostModelGetFacet commands, and then deleting the original child group with the hostModelDeleteGroup command. A plug-in can find out how many vertices and facets a child group contains by sending a hostModelGetGroup command. This command also gets the group's reference pointer, vertex flags, transformation matrix, rotation matrix and motion blur matrix.

Morph is an example of a model plug-in that generates new model data by referencing the model data in its child groups. Morph blends the vertex values of two child groups together by performing the following algorithm:

```
send hostModelGetGroup for group 1
send hostModelGetGroup for group 2
for i = 1 to (number of vertices) {
        send hostModelGetVertex for vertex i in group 1
        send hostModelGetVertex for vertex i in group 2
        calculate a new vertex by interpolating the group 1 and group 2 vertex values
        Send hostModelAddVertex to add the new vertex to the plug-in group
}
for i = 1 to (number of facets) {
        send hostModelGetFacet for facet i in group 1
        send hostModelAddFacet to copy the facet to the plug-in group
}
send hostModelDeleteGroup for group 1
send hostModelDeleteGroup for group 2
```

This algorithm works by first calculating and copying the vertex list and then copying the facet list.

The following algorithm would allow for a differing number of vertices in each group as long as the facet list was the same:

```
send hostModelGetGroup for group 1
send hostModelGetGroup for group 2
for i = 1 to (number of facets) {
        send hostModelGetFacet for facet i in group 1
        send hostModelGetFacet for facet i in group 2
        create a new facet record based upon the group 1 facet
        for j = 1 to (number of vertices of facet i) {
                get vertex j of facet i in group 1
                get vertex j of facet i in group 2
                calculate a new vertex by interpolating the group 1 and group 2 vertex
                        values
                Send hostModelAddVertex to add the new vertex to the plug-in group
                Store the new vertex index in the new facet record
        }
        send hostModelAddFacet to add the new facet record to the plug-in group
}
send hostModelDeleteGroup for group 1
send hostModelDeleteGroup for group 2
```

This algorithm creates a larger number of vertices in the final model but is more robust when applied to models imported into Animator.

6.2 Static and Dynamic Model Types

A model plug-in may generate either static or dynamic models. A static model is one that never changes. The Standard Shapes plug-in, for example, creates several types of basic three dimensional primitives whose shape never change unless the user picks a new shape from the plug-in's user interface. A dynamic model may change over time or may change in relation to changes in its own plug-in group or one of its child groups. The Particle plug-in, for example, creates a stream of particles that change at every frame.

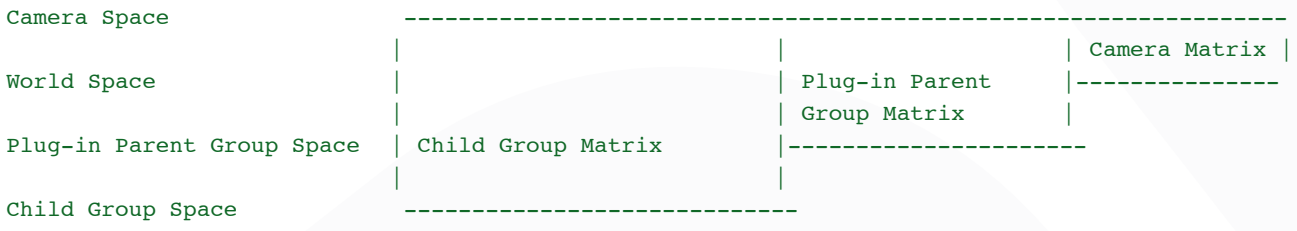
When a model plug-in receives a *pluginInterface* command and determines that it must regenerate its model data due to changes the user has made in its user interface dialog, it should send a *hostModelReset* command to the host. When the host receives this command, it will send a pluginModelSetup command and a pluginModelGenerate command to the plug-in.

A model plug-in should set the appropriate sensitivity flags in the thePluginPermission field of the pluginInformation command record. The sensitivity flags cause the host to send a pluginModelSetup command to the plug-in whenever the time, plug-in parent group or a plug-in child group changes. When a dynamic model plug-in receives a pluginModelSetup command, it will either send a hostModelReset command every time or only when it detects a change which will force it to re-generate its model data. A static model plug-in should disable its sensitivity flags and should not send a hostModelReset when it receives *pluginModelSetup* command.

Both static and dynamic plug-ins should always re-generate their model data when they receive a *pluginModelGenerate* command.

6.3 Matrix Operations

Model data is created and stored in camera space coordinates. Depending upon the needs of a plug-in, these coordinates may need to be converted into child group, plug-in parent group or world space coordinates. This can be done by using the matrices that are provided in the pluginModelSetup, and hostModelGetGroup commands. The pluginModelSetup commands provides the plug-in parent group and camera matrices. The hostModelGetGroup command supplies the child group matrices. The following chart shows the different spaces and how the different matrices are used to get from one to another:



Since all vertex positions are in camera space coordinates, a plug-in must perform matrix operations if it wishes to compute vertex positions in world, plug-in parent group or child spaces. If a plug-in needs to work in child group space, for example, it must first get a coordinate in camera space coordinates using hostModelGetVertex. It should then apply the inverse of the child group matrix to the vertex position. A new vertex position is computed and then the child group matrix is applied to it to bring it back into camera space coordinates. The vertex can then be added to the new group using the hostModelAddVertex command. Since the inverse of the child group matrix is not supplied as part of the hostModelGetGroup command, the plug-in is responsible for inverting the child group matrix. A matrix inversion routine is provided in PluginMatrix.c that is included with the SDK. There is a set of three matrices provided with the camera, plug-in parent group and child group matrices. These are: the position, rotation and motion blur matrices. The position matrices work as described above. The rotation matrices contain only the rotation components of the position matrices and are used to compute vertex normals. The motion blur matrices contain the position matrices from the previous frame and are used to compute vertex motion blur positions.

7.0 The Lens FlarePlug-in Interface

The Lens Flare Plug-in Interface was designed to allow a developer to generate and superimpose 2D graphical images over an image as it is being rendered. This allows the simulation of the interaction of light sources with the camera's lens elements. Lens flare plug-ins also have access to the host channel commands in Animator. The plug-in can create animation channels for any of its user interface values. This gives the user the ability to apply Animator's advanced animation features to the plugin's editable values.

7.1 The Setup Process

The *pluginFlareSetup* command contains the information a lens flare plug-in needs to draw its lens flare effect during the pluginFlareGenerate command. This information includes the image buffer resolution, light source location in space and in screen coordinates, and the current time and frame number. The plug-in should store this information in its PluginData record for use during the pluginFlareGenerate command.

7.2 The Generate Process

The pluginFlareGenerate command contains information about the buffer into which the flare plug-in is to draw its effect. The host may choose to send the plug-in a pluginFlareGenerate command for each scanline in the buffer or it may send the plug-in a single pluginFlareGenerate command to generate the entire buffer. A flare plug-in should be developed to allow it to efficiently generate its effect on a scanline-by-scanline basis. This will enable the plug-in to work well regardless of how the host chooses to generate the flare buffer. Camera currently generates the lens flares on a scanline basis. The following structures are used to define the flare buffer:

```
typedef struct {
        SInt32 top, left, bottom, right;
        SInt32 width, height;
} L_Rect;

        typedef struct {        /* The flare buffer contains pixels of type FlareColor (0..16384) */
        Ptr     bufferBase;     /* The buffer's base address */
        SInt32  bufferRowBytes; /* The number of bytes in a single scanline of the buffer */
        L_Rect  bufferRect;     /* The enclosing rectangle of the buffer */
        short   bufferDepth;    /* The bits per pixel of the flare buffer (8 or 48) */
        SInt32  bufferIndex;    /* The index color of flare rings for 8 bit wire frame buffers */
} FlareBuffer;
```

The FlareBuffer's bufferDepth value can be either 8 or 48 bits. Other bit depths may be added in the future but for now these are the only ones a flare plug-in needs to be able to draw into. When the buffer is 48 bits deep, each pixel contains 16 bit red, green and blue channels. The following record defines the structure of a 48 bit flare buffer pixel:

```
typedef struct {
        unsigned short red, green, blue; /* 0..16384 */
} FlareColor, *FlareColorPtr;
```

The host program begins by initializing the buffer to zero (black) and then sends a *pluginFlareGenerate* command to each of its flare plug-ins. Each flare plug-in should calculate a color value and sum it with the contents of the flare buffer at every pixel. The values in the 16 bit channels range from 0 (black) to 16384 (white). The plug-in should limit each channel's value to 16384 to prevent overflow. After calling each flare plug-in, the host will sum the buffer's contents with the rendered image.

When the buffer is 8 bits per pixel, the flare plug-in should draw an outline of its effect into the buffer. The plug-in should draw the outline with the bufferIndex specified in the FlareBuffer record. In this mode, the host simply initializes the flare buffer to zero and the flare plug-in replaces the pixel values in the buffer with the bufferIndex value as it draws its outline. 8 bit flare buffers are only used in ElectricImage to provide a quick preview for the size and position of the flare effect. The LensFlare plug-in generates a circle with the radius and location of each of its flare elements. The Glow plug-in generates a square with the size and location of its glow circle.

8.0 The Project Channel Interface

Plug-ins may have their own animation channels that are displayed in Animator's Project window . A plug-in may create channels to animate the following data types:

```
#define channelLabel            'LABL' /* A container label (contains other channels) */
#define channelEnd              'LEND' /* The end of a container label */
#define channelReal             'REAL' /* Real (floating point) data (8 byte Double) */
#define channelCRGB             'CRGB' /* RGB 16 bit color data (6 byte RGBColor) */
#define channelARGB             'ARGB' /* ARGB 16 bit color data (8 byte ARGBColor) */
#define channelCHSV             'CHSV' /* HSV 16 bit color data (6 byte HSVColor) */
#define channelAHSV             'AHSV' /* AHSV 16 bit color data (8 byte AHSVColor) */
#define channelInteger          'SINT' /* Signed integer count data (4 byte SInt32) */
#define channelUnsigned         'UINT' /* Unsigned integer count data (4 byte UInt32) */
#define channelBoolean          'BOOL'  /* Boolean flag data (2 byte Boolean) */
#define channelUnsignedReal     'UREL' /* Unsigned Real (floating point) data (8 byte double) */
#define channelCoordinate       'CORD' /* 3 Axis Coordinate data (3 * 8 byte double) */
```

The *channelLabel* and *channelEnd* types are used to define container channels. The Plug-in creates a channel container for each of its sub elements. This helps to visually organize the channel data when a large number of animation channels are required. All of the animation channels created between the channelLabel and channelEnd types are nested inside a hierarchical block. The user can open this block to display the sub-channels by clicking on the channelLabel name in the Project window.

The plug-in creates its channels when the host sends it a *pluginInitializeChannel* command. The plug-in sends a *hostAddChannel* for each of its animation channels. The plug-in sends the host the name, type and default value of the animation channel. The host returns a channel reference to the plug-in. The plug-in saves this reference number and uses it to later retrieve data values from the animation channel. The plug-in can delete some or all of its animation channels with the *hostDeleteChannel* and add new ones if the user changes parameters during the pluginInterface command. If a plug-in determines that its animation channels have become invalid for some reason, it can send a hostResetChannel command to cause the host to delete all of the plug-in's animation channels and send the plug-in another *pluginInitializeChannel* command.

The plug-in can send either a hostAddChannel or a hostInsertChannel command to add animation channels. The command *hostAddChannel* is used to during a complete rebuild of the animation channel hierarchy after the host calls *pluginInitializeChannel* or the plug-in calls *hostResetChannel*.

The *hostInsertChannel* command can be used to insert a new animation channel into an already exisiting channel hierarchy without rebuilding a completely new list. The main addition to the hostInsertChannel command is the parent reference which must be known in order to add a child to the list.

When the plug-in receives a pluginLoadChannel command, it should load its ControlData values at the specified frame number from each of its animation channels with the *hostGetChannel* command. Animator sends a *pluginLoadChannel* command for every frame to be rendered and then writes the plug-in's ControlData record to Camera's control file. This makes the information contained in the plug-in's animation channels available to the plug-in when it is called from Camera.

After the host sends a *pluginInterface* command, it will send a *pluginUpdateChannel* command to the plug-in. The plug-in should respond by sending a *hostSetChannel* command for each of its animation channels (other than the *channelLabel* and *channelEnd* channels). The host will create a new key frame if the Project window is set to time or key frame mode and the new value does not match the value in the animation channel at the current frame. In frame mode, Animator will create a custom frame when a difference is detected in the channel value.

The host will send a *pluginFinishChannel* command to the plug-in if its animation channels have been removed. The plug-in should reset its channel reference numbers to indicate that the channels are no longer valid.

Electric Image Animation System

9.0 Software Copy Protection

A plug-in can get the host application's serial number string by sending a *hostChallenge* command. The serial number is stored as a seven-character Pascal string and begins with two ASCII characters followed by 5 numeric digits. It is up to plug-in authors to develop their own copy protection scheme based upon the host serial number. One method is to create an authorization string that is coded with the contents of the serial number string. In order to activate the plug-in, the user must enter this string in a modal dialog box. The plug-in then stores the authorization string in its resource file and validates it during every generate command.

Plug-ins should validate copy-protection in Animator only, not in Camera

API Reference

10.0 Calls from the Host to the Plug-In

The following calls are common for Model and Flare plug-ins.

10.1 PluginInitialize

This is the first command in the plug-in command sequence. A plug-in's activation is bounded by *pluginInitialize* and *pluginFinish* commands. While a plug-in is active, the host will maintain its control data and plug-in-data records.

The Host will also maintain the plug-in's control data record between activations. Animator keeps each instance of a plug-in active while the project it has been added to is open. Camera will activate each instance of a plug-in once for every frame (or sub-frame) it renders. If a Particle plug-in is rendered in an animation with shadows for example, it will be activated twice per frame: once for the shadow and once for the final shaded frame. Camera does not currently save the contents of a plug-in's control data record between activations. If a plug-in needs to be able to "remember" what occurred on an earlier frame, it must create its own temporary storage file on the local computer's hard disk and access that file at every frame.

```
#define pluginInitialize 'INIT'
typedef struct {
        PluginInfoRec theInfo;
} PluginInitializeRec, *PluginInitializePtr;
```

The control data and plug-in data records should be allocated during this call. The sizes of these records may need to be adjusted in later plug-commands but they should be at least allocated to size zero during the pluginInitialize command. The control data record will usually already be allocated with its contents from the plug-in's last activation or from the plug-in's default values. The plug-in must be able to handle the case when this record is empty (size zero). The plug-in should also have a way of examining the contents of the control data record to determine if it contains the correct information and belongs to the correct version of the plug-in.

After allocating the control data and plug-in data records, the plug-in should initialize their contents with default values. During this initialization, the plug-in may need to access its own resources.

10.2 PluginFinish

This is the last command in the plug-in command sequence. It tells the plug-in that it is about to become deactivated.

```
#define pluginFinish 'FINI'
typedef struct {
        PluginInfoRec theInfo;
} PluginFinishRec, *PluginFinishPtr;
```

The plug-in should release any resources and close any files it has opened during the earlier commands. It is not necessary for the plug-in to dispose of its control data and plug-in data records since they will be taken care of by the host. Animator will save the contents of the control data record and dispose of the plug-in data record. Camera disposes of both records. Most plug-ins will not need to do anything during the *pluginFinish* command.

10.3 PluginInterface

This command tells the plug-in to display its user interface dialog box and accept user input.

```
#define pluginInterface 'INTF'
typedef struct {
        PluginInfoRec theInfo;
} PluginInterfaceRec, *PluginInterfacePtr;
```

Most plug-ins have user editable values. These values are presented to the user for editing with a user interface dialog box. When the plug-in receives a *pluginInterface* command, it should display its user interface dialog box and allow the user to edit control values until the user hits the OK or Cancel button. If the user hit the OK button, the plug-in should store the edited control values into its control data record. A model plug-in should send a hostModelReset command to the host if the new control values will cause the generated model data to change. The host should respond by sending the *pluginModelSetup* and *pluginModelGenerate* commands to the plug-in.

Animator sends the *pluginInterface* command to the plug-in when it is first added to the project or when the user clicks on the Options button for a model plug-in's group or a flare plug-in's light source. Animator will send a *pluginUpdateChannel* command to a plug-in following the *pluginInterface* command. This allows the plug-in to update its animation channel values at the current frame.

10.4 PluginInitializeChannel

The Animator sends this command to the plug-in to tell it to add animation channels to its object in the Project window.

```
#define pluginInitializeChannel 'ICHN'
typedef struct {
        PluginInfoRec theInfo;
        SInt32 theStartFrame;
        SInt32 theEndFrame;
        double theStartTime;
        double theFrameTime;
} PluginInitializeChannelRec, *PluginInitializeChannelPtr;
```

theStartFrame sends the animation's starting frame index.

theEndFrame sends the animation's ending frame index.

theStartTime sends the time in seconds of the animation's starting frame. Most animations start at time zero and count upwards in 1/30 of a second increments.

theFrameTime sends the duration in seconds of each frame of the animation. A 30 frames-per- second animation would have a frame duration of 1/30 of a second or 0.033333.

Animator will send this command to a plug-in after it has been added to a project or when the project is opened. When a plug-in receives this command, it should throw away any channel information it currently has and send a hostAddChannel command for each variable it wishes to make available for user animation.

When a plug-in is first added to the project, ElectricImage will create new channels when it receives *hostAddChannel* commands. When a project is opened, it will use the *hostAddChannel* commands to validate the channels that were stored in the project file. This makes it possible to update a plug-in to a newer version without losing animation information in existing project files. The LensFlare plug-in creates channels for each flare element. These channels control the flare element's visibility, size, color and intensity.

10.5 PluginFinishChannel

This command tells a plug-in that its channels are no longer valid and should no longer be accessed.

```
#define pluginFinishChannel 'FCHN'
typedef struct {
        PluginInfoRec theInfo;
} PluginFinishChannelRec, *PluginFinishChannelPtr;
```

Most plug-ins do not need to do anything when they receive this command. This command was provided for later expansion of the channel command set. Animator currently initiates all channel activity so the plug-in should not be accessing its animation channels unless it is told to do so by the host.

10.6 PluginUpdateChannel

This command tells a plug-in to update the values in a range of frames of its animation channels.

```
#define pluginUpdateChannel 'UCHN'
typedef struct {
        PluginInfoRec theInfo;
        SInt32 theStartFrame;
        SInt32 theEndFrame;
        double theStartTime;
        double theFrameTime;
} PluginUpdateChannelRec, *PluginUpdateChannelPtr;
```

theStartFrame sends the animation's starting frame index.

theEndFrame sends the animation's ending frame index.

theStartTime sends the time in seconds of the animation's starting frame. Most animations start at time zero and count upwards in 1/30 of a second increments.

theFrameTime sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.

This command is currently used by Animator to update the contents of the plug-in's animation channels after the user has accessed the plug-in's interface dialog box. The plug-in should respond by sending a hostSetChannel command for each of its animation channels. The host compares each plug-in channel value to the one stored in the project at the current frame. If the values are different, either a key frame or custom frame is created at the current frame with the new value depending upon whether the project window is in key frame or frame mode.

When LensFlare receives this command it will update its channels with the variables stored in its control data record. The same control data record variables can be edited in the user interface dialog and loaded with the pluginLoadChannel command.

10.7 PluginLoadChannel

This command tells a plug-in to load the current frame's plug-in channel values into the plug-in's control data record.

```
#define pluginLoadChannel 'LCHN'
typedef struct {
        PluginInfoRec theInfo;
        SInt32 theStartFrame;
        SInt32 theEndFrame;
        double theStartTime;
        double theFrameTime;
        SInt32 theFrame;
} PluginLoadChannelRec, *PluginLoadChannelPtr;
```

theStartFrame sends the animation's starting frame index.

theEndFrame sends the animation's ending frame index.

theStartTime sends the time in seconds of the animation's starting frame. Most animations start at time zero and count upwards in 1/30 of a second increments.

theFrameTime sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.

theFrame sends the current frame index.

When the plug-in receives this command, it should send a hostGetChannel command to the host for each of its animation channels. This will update the values in the plug-in's control data record to match those in the animation channels at the current frame. Animator sends this command to the plug-in before sending a *pluginInterface* command. It also sends this command to the plug-in as each frame is sent to Camera in the control file. Because the contents of the plug-in's control data record are written to the control file at each frame, the plug-in is able to have access to the animated values before rendering.

10.8 PluginDescribe

This command tells the plug-in to describe one of its error codes.

```
#define pluginDescribe 'DESC'
typedef struct {
        PluginInfoRec   theInfo;
        SInt32          theResult;
        Str255          theDescription;
} PluginDescribeRec, *PluginDescribePtr;
```

theResult sends the plug-in error code which should be described.

theDescription returns the plug-in error code's description as a string.

Most result codes returned by plug-ins are defined in the 'PluginClient.h' file. If a plug-in returns its own custom result code, the host will send a pluginDescribe command to the plug-in. The plug-in should return a brief description of *theResult* in *theDescription*.

11.0 Calls from the Host to the Model Plug-In

The following calls are specific to the host's communication with model plug-ins.

11.1 pluginInformation

This command is used by the Animator to get information about the plug-in.

```
#define pluginInformation 'INFO'
typedef struct {
        PluginInfoRec theInfo;
        UInt32 thePluginType;
        UInt32 thePluginVersion;
        Str255 thePluginName;
        Str255 thePluginCopyright;
        Str255 thePluginAuthor;
        Str255 thePluginDescription;
        UInt32 thePluginPermission;
} PluginInformationRec, *PluginInformationPtr;
```

thePluginType sends the plug-in's unique type.

thePluginVersion sends the plug-in's version number.

thePluginName sends the plug-in's name.

thePluginCopyright sends the plug-in's copyright notice.

thePluginAuthor sends the plug-in's author name, address, phone or any other desired data.

thePluginDecription sends a string which describes the plug-in's capabilities.

thePluginPermission sends a flag which determines whether EIAS has permission to automatically copy this plug-in to slave machines for rendering.

If the host needs to get information about a plug-in in response to request by the user or for other purposes, it should send a pluginInformation command to the plug-in. The plug-in should simply fill in all of the fields in the PluginInformationRec and return to the host. The "|" character is used to delineate a new line in the information strings. The host is responsible for displaying the information strings correctly to the user.

The thePluginPermission flags are defined as follows:

```
#define pluginSensitivityFlag          0      /* The plug-in supports the sensitivity flags */
#define pluginTimeSensitivityFlag      1      /* Sensitive to changes in time */
#define pluginParentSensitivityFlag    2      /* Sensitive to changes to its own group */
#define pluginChildSensitivityFlag     3      /* Sensitive to changes to its child groups */
#define pluginProjectSelectionFlag     28     /* See below */
#define pluginProjectInfoFlag          29     /* See below */
#define pluginPermissionCopyFlag       31     /* unused (obsolete) */
```

pluginSensitivityFlag - if this flag for a model plug-in is set, the host will use the plug-in sensitivity flags to determine when to sent a pluginModelSetup command to the plug-in.

pluginTimeSensitivityFlag - if this flag is set, the plug-in will receive a pluginModelSetup command whenever the current time changes.

pluginParentSensitivityFlag - if this flag is set, the plug-in will receive a pluginModelSetup command when any change is made to the plug-in's own group.

pluginChildSensitivityFlag - if this flag is set, the plug-in will receive a pluginModelSetup command whenever any change is made to one of the plug-in's child groups. These changes can occur either by user interaction (such as dragging the time thumb or a group) or as part of an animated sequence (such as during a shaded preview).

pluginProjectInfoFlag - if this flags is set, the plug-in will receive a global information about most of projects events, such as data changing, objects adding and deleting and others. See section PluginProjectInfo Interface for details.

pluginProjectSelectionFlag - this flag tells host to inform plug-in about changing selection in Animator Project Window. The flag has effect only together with pluginProjectInfoFlag.

When the plug-in receives a pluginModelSetup command, it must send the host a hostModelReset command in order to receive a pluginModelGenerate command. Most model plug-ins should send a hostModelReset command every time they receive a pluginModelSetup command. More advanced plug-ins could try to determine what has changed since the last pluginModelSetup command before deciding if it is necessary to re-generate their model data.

Animator sends a pluginInformation command when the user requests information about a plug-in (by holding down the command key and clicking on the plug-in's icon in the object palette) and before rendering to a remote Camera.

11.2 PluginModelSetup

This command is sent to the model plug-in to allow it to prepare for a pluginModelGenerate command.

```
#define pluginModelSetup 'SETM'
typedef struct {
        PluginInfoRec  theInfo;
        SInt32         theFrameIndex;
        double         theFrameTime;
        double         theFrameTimeDelta;
        Matrix4 theCameraMatrix;
        Matrix4 theCameraRotMatrix;
        Matrix4 theCameraBlurMatrix;
        Matrix4 theGroupMatrix;
        Matrix4 theGroupRotMatrix;
        Matrix4 theGroupBlurMatrix;
} PluginModelSetupRec, *PluginModelSetupPtr;
```

theFrameIndex sends the frame number of the current frame in the animation. This number is usually in the range of frames in the animation. Most animations start at frame index zero and count upwards by one.

theFrameTime sends the time in seconds of the current frame in the animation. Most animations start at time zero and count upwards in 1/30 of a second increments.

theFrameTimeDelta sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.

theCameraMatrix sends the camera's transformation matrix.

theCameraRotMatrix sends the camera's rotation only matrix.

theCameraBlurMatrix sends the camera's motion blur transformation matrix (at the time of the previous frame).

theGroupMatrix sends the group's transformation matrix.

theGroupRotMatrix sends the group's rotation only matrix.

theGroupBlurMatrix sends the group's motion blur transformation matrix (at the time of the previous frame).

If a plug-in requires any information from the pluginModelSetup command record during the following pluginModelGenerate call, it must save it in its internal data. Simple dynamic model plug-ins should send a hostModelReset command when they receive a pluginModelSetup command. More advanced dynamic model plug-ins may attempt to determine if a change has been made which will force them to re-generate their data before sending a hostModelReset command. Static model plug-ins should not send a hostModelReset command.

11.3 PluginModelGenerate

This command is sent to the model plug-in following a pluginModelSetup command. It tells the plug-in to begin generating a new model using the settings in the model setup
record.

```
#define pluginModelGenerate 'GENM'
typedef struct {
        PluginInfoRec theInfo;
        UInt32  theChildGroups;
        UInt32 theVertexFlags;
} PluginModelGenerateRec, *PluginModelGeneratePtr;
```

theChildGroups sends the number of child groups linked to the plug-in's group. The plug-in may access its child groups to generate new model data.
Note: theChildGroups counts only linked groups, not lights or effectors, even if they are linked to the plug-in.

theVertexFlags sends the vertex interpolation flags for the plug-in's group. These flags tell the plug-in that it should generate color, texture and/or normal vertex information. See next chapter for each flag's detailed description.

When a model plug-in receives a generate command, it should generate model data based upon the settings in the model setup record. The plug-in can add groups, vertices and facets to the host to accomplish this. It can either produce the data algorithmically, or it may extract the information from its child groups if any were linked to it.

Model data generated by the plug-in during the previous generate command is automatically deleted by the host before it sends the new generate command to the plugin. The plug-in simply generates new model data from scratch every time it receives a generate command.

11.4 Group Vertex Flags

The vertex flags are used by the host to keep track of which additional values are stored with each vertex.

```
#define pluginVertexColorFlag  31 // Set when a group has colors stored at each vertex
#define pluginVertexNormalFlag 30 // Set when a group has normals stored at each vertex
#define pluginVertexTextureFlag 29 // Set when a group has texture positions stored at each vertex
#define pluginVertexBlurFlag     28 // Set when a group has motion blur positions stored at each vertex
```

Every vertex record contains x, y and z position coordinates. A vertex record may also contain an ARGB color value, a 3D surface normal, a 3D texture position and a 3D motion blur position. If a model plug-in creates its own custom groups, it will usually use the vertex flags supplied by the host in the PluginModelSetup command. However, a model plug-in can override any of the vertex flags by enabling or disabling the respective bit in the theGroupVertexFlags field of the HostModelAddGroup command record.

Vertex colors may be used to smoothly blend between colors across the model's surface. The Mr. Nitro plug-in sets vertex colors so that the model appears to change color as it explodes. Other applications for vertex colors would be pre-computed radiosity shading and finite element analysis.

Vertex normals are used to smoothly shade a polygon surface. If a plug-in is generating its own model data, it will need to compute a surface normal for each vertex if it wishes the model to have smooth appearance when rendered. If a group's vertex normal flag is not set, it will have a faceted appearance when rendered.

Vertex texture positions can be used to create animated models with textures that 'stick' to their surfaces. The Mesh plug-in creates vertex texture positions that allow a flat textured plane to warp into a sphere. A model that has texture vertices has two different shapes, one in the camera's space and one in its texture's space. At a particular frame, the mesh appears to have some intermediate shape between a flat plane and a sphere. In texture space, however, the mesh remains as a flat grid. This trick allows the texture to remain 'stuck' to the mesh regardless of how the vertices change in relation to the camera. Vertex texture positions can also be used to create custom texture applications. If every facet had its texture vertex positions set to the corners of a unit square, the texture applied to the group would appear to be mapped to each individual facet.

Vertex motion blur positions are used to store the vertex position at the previous frame of the animation. This information is used while rendering to produce motion blur streaks.

11.5 PluginCheck

This command is sent to the model plug-in to check if the command can be processed by the plug-in.

```
#define pluginCheck 'CHEK'
typedef struct {
        PluginInfoRec theInfo;
        UInt32 theCommand;
} PluginCheckRec, *PluginCheckPtr;
```

theCommand - a command token to validate. The plug-in should return error code if the command can't be processed.

12.0 Calls from the Model Plug-In to the Host

The following calls are specific to the model plug-in's communication with host.

12.1 Memory Allocation and Releasing Calls

HostAllocate

This command is used by the plug-in to increase or decrease the size of the control data and plug-in data records.

```
#define hostAllocate 'ALOC'
typedef struct {
        Ptr theReference;
        Ptr theControlData;
        SInt32 theControlDataSize;
        Ptr thePluginData;
        SInt32 thePluginDataSize;
} HostAllocateRec, *HostAllocatePtr;
```

theControlData sends and returns pointers to memory allocated for the ControlData record. This field should always be initialized with its corresponding field in theInfo record (found in whatever record was passed to the plug-in during the current call from the host).

theControlDataSize sends the amount of memory needed for the ControlData record. This value is typically sizeof(ControlDataRec).

thePluginData and **thePluginDataSize** sends the previously allocated pointer and size of memory block. Returns a newly allocated pointer or zero if error occurs. This memory allocation technique is obsolete and deprecated. New plug-ins should always pass NULL *thePluginData* and zero *thePluginDataSize.* Instead new plug-ins should use memory allocation calls described in next sections.

HostMemAlloc and HostMemFree

These calls correspond to standard malloc/free memory allocation in "C" language. The difference is that *HostMemAlloc* fails if Camera memory limit exceeded.

```
#define hostMemMalloc   'MMAL'          /* Allocate or reallocate a memory pointer */
typedef struct {
        Ptr theReference                /* <==  Sends the private data used by host callbacks */
        ulong theSize;                  /* <==  Sends the requested memory size */
        void *thePtr;                   /* <==> Sends and returns the pointer to the allocated memory */
} HostMemMallocRec, *HostMemMallocPtr;

#define hostMemFree     'MFRE'          /* Free a memory pointer */
typedef struct {
        Ptr theReference;               /* <==  Sends the private data used by host callbacks */
        void *thePtr;                   /* <==  Sends the pointer to the allocated memory */
} HostMemFreeRec, *HostMemFreePtr;
```

Although *theSize* value can be 8-bytes in 64-bit mode, it's not recommended to allocate such large blocks, see section 3.2.1 for details.

Electric Image Animation System

HostPermAlloc and HostPermFree

These calls are similar to *HostMemAlloc* and *HostMemFree* but available in Camera only and intended for special purposes: passing data from model plug-ins to shaders or to exchange data between plug-ins at render time. The plug-in does not release block allocated by *HostPermAlloc,* it remains valid until Camera automatically releases it at rendered frame finish. The allocated block is typically linked to group via *HostModelSetGroupData* and then can be obtained via *HostModelGetGroupData,* See these calls descriptions for more details.

```
#define hostPermMalloc  'PMAL'  /* Allocate or reallocate a permnanant memory pointer */
typedef struct {
        Ptr theReference        /* <==  Sends the private data used by host callbacks */
        ulong theSize;          /* <==  Sends the requested memory size */
        void *thePtr;           /* <==> Sends and returns the pointer to the allocated memory */
} HostPermMallocRec, *HostPermMallocPtr;

#define hostPermFree    'PFRE'  /* Free a permnanant memory pointer */
typedef struct {
        Ptr theReference;       /* <==  Sends the private data used by host callbacks */
        void *thePtr;           /* <==  Sends the pointer to the allocated memory */
} HostPermFreeRec, *HostPermFreePtr;
```

HostFree

This command will return the amount of free memory available to the plug-in. The plug-in can (and should) use this information when allocating its data. See section 3.2 for details

```
#define hostFree 'FREE'
typedef struct {
        Ptr theReference;
        ulong theFreeSize;
} HostFreeRec, *HostFreePtr;
```

theFreeSize returns the amount of free memory available to the plug-in.

12.2 Calls to Read Child Groups and Model Creation

This set of calls is used by plug-ins to extract their child groups information, such as their vertices, facets and attributes information, and to create output models geometry.

HostModelGetGroup

The get group command is used to get information about a child group of a model plugin. This command may only be sent to the host during a model generate command.

```
#define hostModelGetGroup 'GETG'
typedef struct {
        Ptr     theReference;
        SInt32  theGroupIndex;
        Ptr     theGroup;
        UInt32 theGroupVertexFlags;
        UInt32 theGroupVertices;
        UInt32 theGroupFacets;
        Matrix4 theGroupMatrix;
        Matrix4 theGroupRotMatrix;
        Matrix4 theGroupBlurMatrix;
} HostModelGetGroupRec, *HostModelGetGroupPtr;
```

theGroupIndex sends the index value of the desired child group. This value is in the range from 1 to the number of child groups linked to the model plug-in. The number of child groups is found in the *theChildGroups* field of the *PluginModelGenerateRec*. Not-groups children (such as lights, effectors) are not counted.

theGroup returns a reference for the requested child group. Once this group reference is retrieved, it can be used in other commands such as *hostModelGetVertex* and *hostModelGetFacet* to get more information about the child group. The group's reference is a private value defined by the host.

theGroupVertexFlags returns the vertex mapping flags for the requested child. These flags tell the plug-in if the child group contains color, texture and/or normal vertex information. For more information of mapping flags, see section 11.4

theGroupVertices returns the number of vertices of the child group.

theGroupFacets returns the number of facets of the child group.

theGroupMatrix returns the child group's transformation matrix.

theGroupRotMatrix returns the child group's rotation only matrix.

theGroupBlurMatrix returns the child group's motion blur transformation matrix (at the time of the previous frame).

To begin retrieving data from a child group, a group pointer must be retrieved using this command. The children are indexed from 1 to the number of child groups linked to the plug-in. The number of children is found in the theChildGroups field of the pluginModelGenerateRec. A model plug-in such as Mr. Nitro will loop for each child group that is attached to it. The pointer to each child is retrieved with the *hostModelGetGroup* command and then each vertex and facet is accessed with *hostModelGetVertex* and hostModelGetFacet respectively. Mr. Nitro uses the vertex and facet information to create explosion fragments that are sent back to the host with *hostModelAddVertex* and hostModelAddFacet commands.

Electric Image Animation System

HostModelSetGroup

This command is used to copy a group's position, shading and texture attribute records into the current group. It may only be sent to the host during a model generate. The command has effect in Camera only. In Animator, all geometry created by a plug-in always belongs to the plug-in group itself.

```
#define hostModelSetGroup 'CPYG'
typedef struct {
        Ptr theReference;
        Ptr theGroupTemplate;
        UInt32 theGroupVertexFlags;
} HostModelSetGroupRec, *HostModelSetGroupPtr;
```

theGroupTemplate sends a reference to the group which will be used as a template. All of the texture and shading information will be copied from the template group to the current group.

theGroupVertexFlags sends the mapping flags for the current group. For more information of mapping flags, see section 11.4

When a model generate command is sent to a plug-in, the host has already created a work group for the plug-in to add vertex and facet information to. The position, shading and texture attributes of the default work group are copied from the plug-in's group which the user setup in EIAS. If the plug-in wishes instead to assign attributes from one of the plug-in's child groups, it may send a hostSetGroup command to the host passing the pointer to the child group in theGroupTemplate. The plug-in will usually pass the value it received from the pluginModelGenerate command record in theGroupVertexFlags. If the plug-in cannot process one or more of the vertex options, it may choose to disable their flags in this field.

HostModelAddGroup

This command is used to add a new work group to the model plug-in. It can only be sent to the host during a model generate command. The command has effect in Camera only, in Animator, all geometry created by the plug-in is always belongs to the plug-in group itself.

```
#define hostModelAddGroup 'ADDG'
typedef struct {
        Ptr theReference;
        Ptr theGroupTemplate;
        UInt32 theGroupVertexFlags;
        Ptr theGroup;
} HostModelAddGroupRec, *HostModelAddGroupPtr;
```

theGroupTemplate sends a pointer to a group which will be used as the template. All of the texture and shading information will be copied from this group to the new group. A pointer value of null will cause the plug-in's own attributes to be copied to the new group.

theGroupVertexFlags sends the mapping flags for the new group. For more information on mapping flags, see section 11.4

theGroup returns a reference to the new group. The group's reference is a private value defined by the host.

All data added to the host by the model plug-in with the *hostModelAddVertex* and *hostModelAddFacet* commands will be placed into the current work group. Most plug-ins that generate their own model geometry and are not dependent upon the geometry in their child groups will not need to use this command. They can use the default work group that was created by the host before the PluginModelGenerate command was sent.

Plug-ins that depend upon the geometry in their child groups may wish to generate one new work group for each of their child groups. This will allow the shading and texture information in the child groups to be preserved in the new work groups. Mr. Nitro, for example, uses the *hostModelAddGroup* command before fragmenting each of its child groups. The fragments keep the texture and shading information that was applied to the child groups.

HostModelDeleteGroup

This command is used to tell the host to hide a child group linked to the plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelDeleteGroup 'DELG'
typedef struct {
        Ptr theReference;
        Ptr theGroup;
} HostModelDeleteGroupRec, *HostModelDeleteGroupPtr;
```

theGroup sends the reference of the plug-in's child group which should be hidden and not rendered in Camera or displayed in Animator drawing windows.

Child groups linked to a plug-in are usually rendered in Camera and displayed in the drawing windows of Animator. If a plug-in wishes to use the child groups to generate its own model data, it may wish to hide their original model data. In the case of Mr. Nitro, for example, the child groups are fragmented with a simulated explosion. If Mr. Nitro did not hide its child groups, both the original model data and the exploded fragments would be rendered and displayed. Instead, Mr. Nitro sends a *hostModelDeleteGroup* command for each child group so that they do not appear in the display or rendered animation. The child groups are not actually deleted. They will still be linked to the plug-in and will show up again during the next *pluginModelGenerate*

HostModelGetVertex

This command is used to get vertex information from a child group of the model plug-in.

```
#define hostModelGetVertex 'GETV'
typedef struct {
        Ptr theReference;
        Ptr theGroup;
        UInt32 theVertexIndex;
        ARGB theColor;
        DCoordinate theNormal;
        DCoordinate thePosition;
        DCoordinate theBlurPosition;
        DCoordinate theTexturePosition;
} HostModelGetVertexRec, *HostModelGetVertexPtr;
```

theGroup sends the reference of the plug-in's child group from which the vertex is to be retrieved. The group reference is found by using the *hostGetGroup* command.

theVertexIndex sends the index number of the desired vertex. The vertices are ordered from 1 to the number of vertices. The number of vertices in a plug-in child group is found by using the *hostGetGroup* command.

theColor returns the ARGB vertex color. The color value is only valid if the reference group's color vertex flag is enabled.

theNormal returns the 3D vertex normal in camera space. The normal value is only valid if the reference group's normal vertex flag is enabled.

thePosition returns the 3D vertex position in camera space. The position value is always valid.

theBlurPosition returns the 3D vertex motion blur position in camera space (at the time of the previous frame). The blur position value is only valid if the reference group's blur position vertex flag is enabled.

theTexturePosition returns the 3D vertex texture position. The texture position value is only valid if the reference group's texture vertex flag is enabled.

Model plug-ins that need to access the model data of their child groups can get information about a vertex by using the hostModelGetVertex command. The plug-in must put the index number of the desired vertex into theVertexIndex field of the *HostModelGetVertexRec*. The number of vertices in a child group is found in *theGroupVertices* field of the *HostModelGetGroupRec*. A plug-in can loop from 1 to theGroupVertices sending a *hostModelGetVertex* and a *hostModelAddVertex* command to copy all of the vertices from the child group to the current work group. In Mr. Nitro, the vertices are accessed one facet at a time by using the contents of theVertices in the *HostGetFacetRec*.

HostModelGetVertexArray

This command is an "array" version of *HostModelGetVertex,* available in Animator only.

```
#define hostModelGetVertexArray 'GARV'
typedef struct {
        Ptr             theReference;
        Ptr             theGroup;
        SInt32 theVertexIndex;
        SInt32 theVertexTotal;
        ARGB *theColor;
        DCoordinate *theNormal;
        DCoordinate *thePosition;
        DCoordinate *theBlurPosition;
        DCoordinate *theTexturePosition;
} HostModelGetVertexArrayRec, *HostModelGetVertexArrayPtr;
```

HostModelAddVertex

This command is used to add a new vertex to the plug-in's current work group. This command may only be sent to the host during a model generate command.

```
#define hostModelAddVertex 'ADDV'
typedef struct {
        Ptr theReference;
        DCoordinate thePosition;
        ARGB            theColor;
        DCoordinate theNormal;
        DCoordinate theBlurPosition;
        DCoordinate theTexturePosition;
        UInt32          theIndex;
} HostModelAddVertexRec, *HostModelAddVertexPtr;
```

thePosition sends the 3D vertex position in camera space. The position value is always used by the host.

theColor sends the ARGB vertex color. The color value is only used by the host if the current work group's color vertex flag is enabled.

theNormal sends the 3D vertex normal in camera space. The normal value is only used by the host if the current work group's normal vertex flag is enabled.

theBlurPosition sends the 3D vertex motion blur position in camera space (at the time of the previous frame). The blur position value is only used by the host if the current work group's blur position vertex flag is enabled.

theTexturePosition sends the 3D vertex texture position. The texture position value is only used by the host if the current work group's texture vertex flag is enabled.

theIndex returns the index of the newly created vertex.

The model-plug-in is responsible for setting all of the vertex information fields that are enabled in the current work group's vertex flags. The host uses the vertex information to create a new vertex and returns its index to the plug-in in theIndex. The new vertex index is passed back to the host in the hostAddFacet command. Because the vertex indexes used in the hostAddFacet command reference vertices that have already been created, the plug-in must add the vertices before it adds the facets which reference them.

HostModelAddVertexArray

This command is an "array" version of *HostModelAddVertex,* available in Animator only and may only be sent to the host during a model generate command.

```
#define hostModelAddVertexArray 'AARV'
typedef struct {
        Ptr             theReference;
        SInt32  theVertexTotal;
        DCoordinate     *thePosition;
        ARGB *theColor;
        DCoordinate *theNormal;
        DCoordinate *theBlurPosition;
        DCoordinate *theTexturePosition;
        SInt32 theIndex;
} HostModelAddVertexArrayRec, *HostModelAddVertexArrayPtr;
```

HostModelGetFacet

This command is used to get information about a facet in a child group of a model plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelGetFacet 'GETF'
typedef struct {
        Ptr theReference;
        Ptr theGroup;
        UInt32 theFacetIndex;
        ARGB theColor;
        short theNumVertices;
        Vertices theVertices;
} HostModelGetFacetRec, *HostModelGetFacetPtr;
```

theGroup sends the reference of the plug-in's child group from which the facet is to be retrieved. The group reference is found by using the hostGetGroup command.

theFacetIndex sends the index number of the desired facet. The facets are ordered from 1 to the number of facets. The number of facets in a plug-in child group is found by using the *hostGetGroup* command.

theColor returns the color of the facet.

theNumVertices returns the number of vertices contained in this facet. This number is in the range of 1 to 4 and represents a point, line, triangle or quadrangle respectively.

theVertices returns an array which contains the indexes to the vertices which define the facet. These indexes can be used to retrieve the vertices themselves. Only the array elements in the range of 0 to (theNumVertices – 1) contain valid information.

Model plug-ins that need to access the model data of their child groups can get information about a facet by using the hostModelGetFacet command. The plug-in must put the index number of the desired vertex into theFacetIndex field of the *HostModelGetFacetRec*. The number of facets in a child group is found in *theGroupFacets* field of the *HostModelGetGroupRec*. A plug-in can loop from 1 to *theGroupFacets* sending a *hostModelGetFacet* and a *hostModelAddFacet* command to copy all of the facets from the child group to the current work group. To get information about the vertices of a facet, a plug-in send a hostModelGetVertex command for each vertex index in the facet's theVertices array. Mr. Nitro does this for each facet before breaking up a facet into smaller fragments.

HostModelGetFacetArray

This command is an "array" version of *HostModelGetFacet,* available in Animator only.

```
#define hostModelGetFacetArray 'GARF
typedef struct {
        Ptr             theReference;
        Ptr             theGroup;
        SInt32 theFacetIndex;
        SInt32 theFacetTotal;
        ARGB *theColor;
        short *theNumVertices;
        Vertices *theVertices;
} HostModelGetFacetArrayRec, *HostModelGetFacetArrayPtr;
```

HostModelAddFacet

This command is used to add a new facet to the current work group. This command may only be sent to the host during a model generate command.

```
#define hostModelAddFacet 'ADDF'
typedef struct {
        Ptr theReference;
        ARGB theColor;
        unsigned short theNumVertices;
        Vertices theVertices;
} HostModelAddFacetRec, *HostModelAddFacetPtr;
```

theColor sends the color of the new facet.

theNumVertices sends the number of vertices contained in this facet. This number is in the range of 1 to 4 and represents a point, line, triangle or quadrangle respectively.

theVertices sends an array which contains the indexes to the vertices which define the facet. These indexes were returned by the host in the *hostModelAddVertex* command when the vertices were added to the current work group. The host only sets the array elements in the range of 0 to theNumVertices - 1.

In order to add new model data to a plug-in's current work group, *hostModelAddVertex* and *hostModelAddFacet* commands must be sent to the host. The hostModelAddVertex adds a new vertex to the current work group and returns an index value that the plug-in should put into the facet's theVertices array. After each of the facet's vertices (1 for a point, 2 for a line, 3 for a triangle and 4 for a quadrangle) have been created, the facet itself can then be added to the current work group with the *hostModelAddFacet* command.

Note: for facets with 1 or 2 vertices (points or lines) the unused elements (theVertices[2] and theVertices[3]) should be filled with zeroes.

HostModelAddFacetArray

This command is an "array" version of *HostModelAddFacet,* available in Animator only and may only be sent to the host during a model generate command.

```
#define hostModelAddFacetArray 'AARF'
typedef struct {
        Ptr     theReference;
        SInt32 theFacetTotal;
        ARGB    *theColor;
        short   *theNumVertices;
        Vertices *theVertices;
        SInt32 theIndex;
} HostModelAddFacetArrayRec, *HostModelAddFacetArrayPtr;
```

HostModelGINF

```
HostModelGINF
HostModelGATR
HostModelGBLR
HostModelTMAP
HostModelTATR
HostModelBMAP
HostModelBATR
HostModelRMAP
```

These commands allow a plug-in to access and modify the contents of various group attribute records.

```
#define hostModelGINF 'GINF' /* Set the values of the group's GINF record */
#define hostModelGATR 'GATR' /* Set the values of the group's GATR record */
#define hostModelGBLR 'GBLR' /* Set the values of the group's GBLR record */
#define hostModelTMAP 'TMAP' /* Set the values of the group's TMAP record */
#define hostModelTATR 'TATR' /* Set the values of the group's TATR record */
#define hostModelBMAP 'BMAP' /* Set the values of the group's BMAP record (second TMAP) */
#define hostModelBATR 'BATR' /* Set the values of the group's BATR record (second TATR) */
#define hostModelRMAP 'RMAP' /* Set the values of the group's RMAP record */
```

All of these commands have the same record structure. The record for hostModelGINF is as follows:

```
typedef struct {
        Ptr     theReference;
        Ptr     theGroup;
        GINFPtr theGINF;
} HostModelGINFRec, *HostModelGINFPtr;
```

theGroup sends the reference of the plug-in's child group from which the requested attribute record is to be retrieved. The group reference is found by using the hostGetGroup command.

GINFPtr Sends a pointer to the requested group attribute record.

A model plug-in can send these commands to get attribute records from any of its child groups, its own plug-in group or any of the groups it has created. The plug-in may change the contents of its own groups or any of its created groups but should not attempt to change the contents of one of its child groups.

This command should only be sent for the purpose of modifying the contents of plug-in group attribute records during the pluginModelGenerate command and only before the first vertex is added to requested group. The contents of the groups GINF, GATR, GBLR, TMAP, TATR, BMAP,
BATR and RMAP records are described in FACTStuff.h.

It is rare that a model plug-in will need to call one of these commands since it is usually left up to the user to apply textures and shading attributes to a group in Animator.

12.3 Calls to create, delete, and read and write animation channels

All these calls are available in Animator only. They can be conditionally divided into 2 categories:

- Calls to handle channels created by the plug-in itself. Animator collects them into “Socket” channels block. These channels can be created, deleted and modified by the plug-in in any way.

- Calls to handle ANY channels in a whole project, that can belong to the plug-in and not. These channels can be only modified by the plug-in but not created or deleted.

If a command data structure contains the field *theChannelOwner,* it tells that a channel is an arbitrary channel in project.

***NOTE**: keep in mind that thehost can disable modifications for some channels, see the HostGetChannelInfo (theFlag) for details.*

HostResetChannel

This command tells the host to delete all of the plug-in’s animation channels.

```
#define hostResetChannel ‘RCHN’
typedef struct {
        Ptr theReference;
} HostResetChannelRec, *HostResetChannelPtr;
```

If a plug-in determines that its channel data is no longer valid for some reason (the control data record contained invalid data, for example) it should send a *hostResetChannel* command to the host. After the current plug-in command returns to the host, the host should send a *pluginInitializeChannel* to the plug-in so that new channels can be created. A plug-in should not attempt to access its animation channels after it has sent a *hostResetChannel* command.

HostAddChannel

This command tells the host to add a new plug-in animation channel.

```
#define hostAddChannel ‘ACHN’
typedef struct {
        Ptr theReference;
        UInt32 theChannelID;
        UInt32 theChannelType;
        Str255 theChannelName;
        Ptr theChannelDefault;
        Ptr theChannelMinimum;
        Ptr theChannelMaximum;
        Ptr theChannelBefore;
        Ptr theChannelReference;
} HostAddChannelRec, *HostAddChannelPtr;
```

theChannelID sends the animation channel’s unique ID. The ID is defined by the plug-in and will help to validate the channel in later commands.

theChannelType sends the animation channel’s four character type. The channel types are defined in

'EIPlugin.h'. The channel type is used by the host to display and animate the channel values and also to validate the channel in later commands.

theChannelName sends the animation channel's name. The name is used by Animator when displaying the channel in the project window.

theChannelDefault sends the pointer to the animation channel's default value. This value will be used by the host to initialize the contents of the animation channel. Initialize to 'nil' if the channel does not contain an animated value.

theChannelMinimum sends a pointer to the animation channel's minimum allowable value. The host should prevent the animated value from dropping below the minimum value. Initialize to 'nil' if the channel does not contain an animated value or if there is no minimum value for the channel (the host will automatically limit channel values to the largest possible ranges for their numeric type).

theChannelMaximum sends a pointer to the animation channel's maximum allowable value. The host should prevent the animated value from rising above the maximum value. Initialize to 'nil' if the channel does not contain an animated value or if there is no maximum value for the channel (the host will automatically limit channel values to the largest possible ranges for their numeric type).

theChannelBefore sends a reference to a previously added channel. The channel being added will appear immediately following the referenced channel. Sending null in this parameter will cause the new channel to be added to the end of the list.

theChannelReference returns a reference to the added channel. This reference should be stored for use in later commands.

This command is sent to the host to add each of the plug-in's animation channels when a plug-in receives a *pluginInitializeChannel* command. When Animator adds a plug-in to the project, it will add the plug-in's animation channels following the plug-in's object in the project list. When a project is opened, Animator will send the *pluginInitializeChannel* command to the plug-in and then validate the ID, and data type of each channel it receives against the channels that were stored in the project file. If the channels all match, it will keep the original channels. If the channels do not match, it will use the newly created channels and their default values. This will allow plug-in versions to be updated while still maintaining compatibility with older project files.

Animation channels may be added to either store data values or to form containers of other animation channels. This allows the plug-in to create a more structured display of its animation channels in Animator's Project window. Animation channels can contain a variety of data types:

```
/* Channel data types          Description Type Size */

#define channelLabel            'LABL' /* A channel container label none 0 */
#define channelEnd              'LEND' /* The end of a container label none 0 */
#define channelReal             'REAL' /* Real (floating point) data Double 8 */
#define channelCRGB             'CRGB' /* RGB 16 bit color data RGBColor 6 */
#define channelARGB             'ARGB' /* ARGB 16 bit color data ARGBColor 8 */
#define channelCHSV             'CHSV' /* HSV 16 bit color data HSVColor 6 */
#define channelAHSV             'AHSV' /* AHSV 16 bit color data AHSVColor 8 */
#define channelInteger          'SINT' /* Signed integer count data LongInt 4 */
#define channelUnsigned         'UINT' /* Unsigned integer count data LongInt 4 */
#define channelBoolean          'BOOL' /* Boolean flag data Boolean 2 */
#define channelUnsignedReal     'UREL' /* Unsigned Real (float) data Double 8 */
#define channelCoordinate       'CORD' /* Coordinate (float) data 3*Double 24 */
```

Each channel is added with a default value. The default is used to initialize the value of all of the frames in the channel when it is first added to the project. A channel may optionally have minimum and maximum value limits. The host will prevent the animated channel's values from exceeding these limits when they are provided by the plug-in.

Every channel should have a unique ID number. The plug-in is responsible for creating and maintaining the channel IDs. Both the plug-in and the host can use the channel ID to validate the contents of a particular animation channel.

HostInsertChannel

This command tells the host to insert a new plug-in animation channel into an already existing channel hierarchy.

```
#define hostInsertChannel 'PCHN'
typedef struct {
        Ptr theReference;
        UInt32 theChannelID;
        UInt32 theChannelType;
        Str255 theChannelName;
        Ptr theChannelDefault;
        Ptr theChannelMinimum;
        Ptr theChannelMaximum;
        Ptr theChannelBefore;
        Ptr theChannelParent;
        Ptr theChannelReference;
} HostInsertChannelRec, *HostInsertChannelPtr;
```

The structure is similar to HostAddChannelRec, but contains an additional *theChannelParent* field.

theChannelParent sends the reference to an existing parent channel. Sending null in this parameter will cause the new channel to be added at t he end of the complete hierarchy listing without any parents.

This command is sent to the host to insert a new animation channel into an already existing channel hierarchy. This command should be called during the pluginInterface command when the plug-in has determined that a new channel is needed. An existing animation channel hierarchy must exist.

HostDeleteChannel

This command tells the host to delete one of the plug-in's animation channels.

```
#define hostDeleteChannel 'DCHN'
typedef struct {
        Ptr theReference;
        Ptr theChannelReference;
        UInt32 theChannelID;
} HostDeleteChannelRec, *HostDeleteChannelPtr;
```

theChannelReference sends the animation channel's reference for validation This reference was returned to the plug-in in the *hostAddChannel* command.

theChannelID sends the animation channel's unique ID for validation.

If the plug-in needs to make changes in its animation channel list, it must first delete the unwanted channels. The plug-in does this by sending a *hostDeleteChannel* command for each animation channel to be deleted. The plug-in must pass the channel reference it received from the host in the HostAddChannelRec record. It must also send the channel's unique ID number for channel validation. W hen the hostDeleteChannel is called on an animation channel which is just a container (no animation values) then all of its offspring animation channels are deleted as well.

The LensFlare plug-in deletes all of its flare element channels when the user changes the lens flare type in the plug-in's user interface. It then adds the animation channels for the new flare type's elements to the project list. The LensFlare plug-in does not delete the animation channels that control the overall lens flare effects because they do not change by lens flare type.

HostGetChannel

This command tells the host to get the value for a plug-in animation channel at a particular frame number.

```
#define hostGetChannel 'GCHN'
typedef struct {
        Ptr theReference;
        Ptr theChannelReference;
        UInt32 theChannelID;
        UInt32 theChannelType;
        Sint32 theChannelFrame;
        Ptr theChannelData;
} HostGetChannelRec, *HostGetChannelPtr;
```

theChannelReference sends the animation channel's reference for validation. This reference was returned to the plug-in in the *hostAddChannel* command.

theChannelID returns the animation channel's unique ID for validation.

theChannelType returns the animation channel's data type for validation.

theChannelFrame sends the requested frame number. The host will return the animation channel's value at this frame.

theChannelData returns a pointer to the requested animation channel's value at the requested frame number.

This command is passed to the host to extract the value of a particular frame in an animation channel. Although a pointer to this value is returned, it is not a pointer to the actual frame's value. It is a pointer to a copy of this value. The plug-in must use a *hostSetChannel* command to actually change the frame value's contents.

The plug-in should send a hostGetChannel command for each of its animation channels in response to a pluginLoadChannel command. The plug-in should validate the channel's ID and data type and then copy the channel data into its control data record. The plug-in must not attempt to change the contents of theChannelData. To change the value of an animation channel, the plug-in should use the hostSetChannel command.

HostSetChannel

This command tells the host to change the value of a plug-in animation channel at a particular frame.

```
#define hostSetChannel 'SCHN'
typedef struct {
        Ptr theReference;
        Ptr theChannelReference;
        UInt32 theChannelID;
        UInt32 theChannelType;
        SInt32 theChannelFrame;
        Ptr theChannelData;
} HostSetChannelRec, *HostSetChannelPtr;
```

The structure is same as *HostGetChannelRec*.

When the host sends a pluginUpdateChannel command to the plug-in, the plug-in should respond by sending a hostSetChannel command for each of its animation channels and for each frame in the specified range. EIAS validates the channel's ID and data type and then compares the new channel value to the current channel value. If the two values are different a new key frame is created if the project is set to time or key frame mode. A custom frame is created if the project is in frame mode. Animator sends a *pluginUpdateChannel* for the current frame to the plug-in following a pluginInterface command.

HostGetChannelInfo

This command tells the host to return information about a animation channel

```
#define hostGetChannelInfo 'GCHI'
typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        UInt32 theChannelType;
        UInt32 theChannelFilter;
        UInt32 theChannelVelocity;
        UInt32 theChannelKeyTotal;
        UInt32 theChannelFlag;
        Ptr theChannelMinimum;
        Ptr theChannelMaximum;
} HostGetChannelInfoRec, *HostGetChannelInfoPtr;
```

theChannelOwner sends the pointer to the owner of the animation channel. A null represents the plug-in's reference.

theChannelReference sends a reference to specify the channel.

theChannelType returns the type of channel. See *hostAddChannel* for more information about the types.

theChannelFilter returns the type of channel value filter. This will tell you about the range of the values this channel can handle.

theChannelVelocity returns 0 if no velocity computation are performed on this channel.

theChannelKeyTotal returns the number of key frames available in the channel.

theChannelFlag returns a flags field for the animation channel. A zero value tells that channel cannot be modified.

theChannelMinimum returns a pointer to the animation channel's minimum allowable value. A null is returned if all possible values for this type are allowed.

theChannelMaximum returns the a pointer to the animation channel's maximum allowable value. A null is returned if all possible values for this type are allowed.

This command is send to the host to return important information about an animation channel. Remember that the host has the same privileges of modifying an animation channel than the plug-in.

The host as well as the plug-in can add, delete, or modify the channel's keys and state. Before the plug-in attempts to issue the more advanced channel commands, it should send this command to update its information about a key channel.

HostAddChannelKey

This command tells the host to add a key frame to an existing animation channel

```
#define hostAddChannelKey 'ACHK'
typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        UInt32 theChannelID;
        UInt32 theDataType;
        Ptr theData;
        double theKeyTime;
} HostAddChannelKeyRec, *HostAddChannelKeyPtr;
```

theChannelOwner sends the pointer to the owner of the animation channel. A null represents the plug-in's reference.

theChannelReference sends a reference to specify the channel.

theChannelID sends a plug-in maintained ID for the channel.

theDataType sends the type of data. Along with theData, this represents the default value for that key.

theData sends the pointer to the default data.

theKeyTime sends the time of the key. When creating keys, care should be taken so that the plug-in does not create keys at the same time. The host will return an error if this happens.

This command allows the plug-in to maintain the key frames of an existing channel. The plug-in should send the same default data type as the channel can accept. Use the hostGetChannelInfo command to inquire about the channel's capabilities. If the plug-in t ries to create a key frame with the same time as an existing one, the host will return an error.

HostDeleteChannelKey

This command tells the host to delete a key frame from an existing animation channel.

```
#define hostDeleteChannelKey 'DCHK'
typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        UInt32 theChannelID;
        SInt32 theKeyIndex;
} HostDelChannelKeyRec, *HostDelChannelKeyPtr;
```

theChannelOwner sends the pointer to the owner of the animation channel. A null represents the plug-in's reference.

theChannelReference sends a reference to specify the channel.

theChannelID sends a plug-in maintained ID for the channel.

theKeyIndex sends the index or types of keyframes to be deleted. A non-zero positive index indicates which key frame is to be deleted. The first key frame starts with 1 (one). A zero index indicates ALL keyframes are to be deleted. A value of -1 for index indicates only SELECTED key frames are to be deleted.

This command allows the plug-in to maintain the key frames of an existing channel. When specifying a key frame you can use the following constants to delete more than one key frame at a time.

HostGetChannelKey

This command tells the host to get a value from the specified key frame(s) of an existing animation channel

```
#define hostGetChannelKey 'GCHK'
typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        UInt32 theChannelID;
        UInt32 theDataType;
        Ptr theData;
        SInt32 theKeyPart;
        SInt32 theKeyIndex;
} HostGetChannelKeyRec, *HostGetChannelKeyPtr;
```

theChannelOwner sends the pointer to the owner of the animation channel. A null represents the plug-in's reference.

theChannelReference sends a reference to specify the channel.

theChannelID sends a plug-in maintained ID for the channel.

theDataType returns the type of data.

theData returns a pointer to the data. Make sure the plug-in checks the data type to extract the data with the correct structure.

theKeyPart sends the part of the key frame to be evaluated.

theKeyIndex sends the index of the key frame to be inquired. A non-zero positive index indicates the key frame. The first key frame starts with 1 (one).

This command is send to the host to get the value of the specified key frame of an existing animation channel. By using the following constants, the plug-in can extract different information:

```
#define PLUGIN_CHANNEL_MODE_PROCESS_VALUE      10
#define PLUGIN_CHANNEL_MODE_PROCESS_TIME       11
```

PLUGIN_CHANNEL_MODE_PROCESS_VALUE extracts the value of the key frame. Check the data type returned for the correct type.

PLUGIN_CHANNEL_MODE_PROCESS_TIME extracts the time of the key frame. The value is always a signed double (channelReal).

Before attempting to extract the key value, call the *hostGetChannelInfo* command to get the total number of key frames. If the plug-in attempts to extract information of a non - existing key frame, an error will be returned.

HostSetChannelKey

This command tells the host to modify a value of specified key frame(s) of an existing animation channel.

```
#define hostSetChannelKey 'SCHK'
typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        UInt32 theChannelID;
        UInt32 theDataType;
        Ptr theData;
        SInt32 theKeyPart;
        SInt32 theKeySubPart;
        SInt32 theKeyIndex;
} HostSetChannelKeyRec, *HostSetChannelKeyPtr;
```

This command allows the plug-in to modify the key frame values. The structure is similar to *HostGetChannelKeyRec.*

theKeyPart - it defines the mode in which the plug-in can modify a value of a key and should be one of following:

```
#define PLUGIN_CHANNEL_MODE_ADD_SYSTEM         1
#define PLUGIN_CHANNEL_MODE_ADD_KEY            2
#define PLUGIN_CHANNEL_MODE_ADD_FRAME          3
#define PLUGIN_CHANNEL_MODE_PROCESS_VALUE      10
#define PLUGIN_CHANNEL_MODE_PROCESS_TIME       11
```

The *PLUGIN_CHANNEL_MODE_PROCESS_VALUE* and the *PLUGIN_CHANNEL_MODE_PROCESS_TIME* are used in the same way as *HostGetChannelKey.*

PLUGIN_CHANNEL_MODE_ADD_SYSTEM adds a key frame value at the current time by using the system's mode. This means that keyframes will either be modified or created, or that custom frames will either be modified or created, which mode the system is currently in. *theKeyIndex* is ignored in this case.

PLUGIN_CHANNEL_MODE_ADD_KEY adds a key frame value by either modifying an existing key frame at the current time, or creating a new key frame with this value at the current time. theKeyIndex is ignored in this case.

PLUGIN_CHANNEL_MODE_ADD_FRAME adds a key frame value by either modifying an existing custom frame at the current time, or creating a new custom frame with this value at the current time. theKeyIndex is ignored in this case.

theKeySubPart - this additional field specifies which part of channel is modified. It can be one of following constants:

```
#define channelPartAll          'CPAL' /* This indicates all components of a value */
#define channelPartCoordinate_X 'CPPX' /* This indicates the X component of a coordinate */
#define channelPartCoordinate_Y 'CPPY' /* This indicates the Y component of a coordinate */
#define channelPartCoordinate_Z 'CPPZ' /* This indicates the Z component of a coordinate */
#define channelPartColor_A      'CPCA' /* This indicates the alpha component of a color */
#define channelPartColor_R      'CPCR' /* This indicates the red component of a color */
#define channelPartColor_G      'CPCG' /* This indicates the green component of a color */
#define channelPartColor_B      'CPCB' /* This indicates the blue component of a color */
```

theKeyIndex - it specifies which key should be modified. Any positive number indicates the indexed key frame to be modified. The first key frame always starts with 1. The following numbers have a special meaning:

```
#define PLUGIN_CHANNEL_SPECIFY_ALL 0
#define PLUGIN_CHANNEL_SPECIFY_SELECTED -1
```

PLUGIN_CHANNEL_SPECIFY_ALL allows the plug-in to modify all key frames in a channel.

PLUGIN_CHANNEL_SPECIFY_SELECTED allows the plug-in to modify only selected key frames in a channel.

Modifying the key times can lead two or more key frames to share the same time, which is impossible. You cannot exist in two places at once. Some of the modes allow more than one key frame to be modified. Care should be taken to make sure that this will not happen. In case it does happen, an error will be returned and the operation will not be performed. To extract all key frame times, use *hostGetChannelKey* and *hostGetChannelInformation*.

HostSetChannelName

```
#define hostSetChannelName     'CCHN'
typedef struct {
        Ptr             theReference;
        Ptr             theChannelOwner;
        Ptr             theChannelReference;
        SInt32  theChannelID;
        PStr255 theChannelName;
} HostSetChannelNameRec, *HostSetChannelNamePtr;
```

theChannelOwner sends the pointer to the owner of the animation channel. A null represents the plug-in's reference.

theChannelReference sends a reference to specify the channel.

theChannelID sends a plug-in maintained ID for the channel.

theChannelName sends a new channel name to set.

NOTE: some channels cannot be renamed. In this case host return an error code.

Electric Image Animation System

13. Calls from the Host to the Flare Plug-in

PluginFlareSetup

This call is sent to the flare plug-in to allow it to prepare for a flare generate command.

```
#define pluginFlareSetup 'SETF'
typedef struct {
        PluginInfoRec theInfo;
        SInt32 theFrameIndex;
        double theFrameTime;
        double theFrameTimeDelta;
        L_Rect theFrameRect;
        double theFrameXScale;
        double theFrameYScale;
        double theFrameXCenter;
        double theFrameYCenter;
        double theBlurShutter;
        Matrix4 theCameraMatrix;
        Matrix4 theCameraRotMatrix;
        Matrix4 theCameraBlurMatrix;
        DCoordinate theLightWorldPosition;
        DCoordinate theLightBlurWorldPosition;
        DCoordinate theLightFramePosition;
        DCoordinate theLightBlurFramePosition;
        DCoordinate theLightDirection;
        DCoordinate theLightBlurDirection;
        ARGB theLightColor;
        double theLightIntensity;
        double theLightDropoff;
        double theLightWorldRadius;
        double theLightFrameRadius;
        double theLightOuterAngle;
        double theLightInnerAngle;
        double theLightSoftExponent;
} PluginFlareSetupRec, *PluginFlareSetupPtr;
```

theFrameIndex sends the frame number of the current frame in the animation. This number is usually in the range of frames in the animation. Most animations start at frame index zero and count upwards by one.

theFrameTime sends the time in seconds of the current frame in the animation. Most animations start at time zero and count upwards in 1/30 of a second increments.

theFrameTimeDelta sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.

theFrameRect sends the frame's rectangle. The rectangle's units are in pixels. The lens flare will be expected to generate its effects into subrectangles of the frame's rectangle.

theFrameXScale sends the frame's X scale value. This value is usually equal to the width of the frame's rectangle.

theFrameYScale sends the frame's Y scale value. This value is usually equal to the height of the frame's rectangle unless the frame was rendered with a non-square pixel aspect ratio.

theFrameXCenter sends the frame's X center location. This value is usually equal to the horizontal center of the frame's rectangle. If the flare plug-in generates a lens flare, the flare's elements would be calculated by the plug-in to pivot around the frame's center point.

theFrameYCenter sends the frame's Y center location. This value is usually equal to the vertical center of the frame's rectangle. If the flare plug-in generates a lens flare, the flare's elements would be calculated by the plug-in to pivot around the frame's center point.

theBlurShutter sends the motion blur shutter angle. This value is usually a number between 0 and 1. The default shutter angle is 0.5. The shutter angle is the percentage of time that the camera's shutter is open for each frame. A moving object would appear to travel half of its distance from the previous frame to the current frame in the rendered image with a 0.5 shutter angle.

theCameraMatrix sends the camera's transformation matrix.

theCameraRotMatrix sends the camera's rotation only matrix.

theCameraBlurMatrix sends the camera's motion blur transformation matrix (at the time of the previous frame).

theLightWorldPosition sends the light's 3D position in the world space coordinates. The camera's transformation matrix would have to be applied to this position to find the light's position in camera space coordinates.

theLightBlurWorldPosition sends the light's motion blur position in world space coordinates (at the time of the previous frame).

theLightFramePosition sends the light's position in frame coordinates. The host computes this point by applying perspective, scale and offsets to the light's position in camera space coordinates. The result is a 2D frame position. Only the X and Y coordinates are used by the flare plug-in when generating its effects in the frame buffer. The Z is computed by Camera as the distance from the camera to the light source in camera space coordinates.

theLightBlurFramePosition sends the light's motion blur position in frame coordinates (at the time of the previous frame).

theLightDirection sends the light's direction normal in camera space.

theLightBlurDirection sends the light's motion blur direction normal in camera space (at the time of the previous frame).

theLightColor sends the light's color. Only the R, G, and B components should be used by the flare plug-in to generate its effects.

theLightIntensity sends the light's intensity. This value is 1.0 by default. The light intensity can be used with the light's drop-off distance to simulate more realistic lighting effects.

theLightDropoff sends the light's drop-off distance. This is the distance over which the light should diminish towards black.

theLightWorldRadius sends the light's radius in world units.

theLightFrameRadius sends the light's radius in frame pixels.

theLightOuterAngle sends the spotlight outer cone angle. Both the inner and outer spotlight cone angles are 0.0 for non-spotlights.

theLightInnerAngle sends the spotlight inner cone angle. Both the inner and outer spotlight cone angles are 0.0 for non-spotlights.

theLightSoftExponent sends the spotlight transition exponent. This controls how fast the spotlight drops off between the inner and outer cones.

When the flare plug-in receives a setup command, it should store any information it needs into its internal data, allocate memory and setup its lookup tables. TheLensFlare plug-in, for example, will read in its flare element resources and generate look-up tables based upon the frame X and Y scales. When it receives the flare generate command, it will use the lookup tables to rapidly calculate element colors. Flare plug-ins must be able to draw into any sub-rectangle of the frame's rectangle from individual scanlines up to the entire rectangle. Camera is currently designed to draw individual scanlines from the top of the frame (low Y value) to the bottom of the frame (high Y value).

14. Miscellaneous Calls from Model Plug-Ins to the Host

14.1 HostCheckRec

This command tells the host to check the command token (specified in *theCommand* field) is a valid command recognized by host.

```
#define hostCheck 'CHEK'
typedef struct {
        Ptr     theReference;  /* <==  Sends the private data used by host callbacks */
        UInt32 theCommand;     /* <==  Sends the command to check for host acceptance */
} HostCheckRec, *HostCheckPtr;
```

14.2 HostVersion

This command retrieves information about host's version the plug-in is actually running in.

```
#define hostVersion 'VERS'
typedef struct {
        Ptr     theReference;   /* <==  Sends the private data used by host callbacks */
        SInt32 versionRelease;  /*  ==> Returns the release version of the host application */
        SInt32 versionRevision; /*  ==> Returns the revision version of the host application */
        SInt32 versionFix;      /*  ==> Returns the fix version of the host application */
        SInt32 versionInternal; /*  ==> Returns the internal version of the host application */
        PStr31 versionString;   /*  ==> Returns the short string version of the host application */
        UInt32 versionCountry; /*  ==> Returns the country code of the host application */
} HostVersionRec, *HostVersionPtr;
```

14.3 HostCreator

This command retrieves information about kind of host the plug-in is actually running in.

```
#define hostCreator 'CREA'
typedef struct {
        char *theReference;    /* <==  Sends the private data used by host callbacks */
        UInt32 theCreator;     /*  ==> Returns the creator of the host application */
} HostCreatorRec, *HostCreatorPtr;
```

The token 'EIAR' (returned in *theCreator* field) means the actual host is Camera, any other signature means Animator. Typically a plug-in calls *HostCreator* at *pluginInitialize* phase and stores received creator in its internal data.

14.4 HostChallenge

This command tells the host to respond to a plug-in's copy protection challenge.

```
#define hostChallenge 'CHAL'
typedef struct {
        Ptr     theReference;
        UInt32 theChallenge;
        ChallengeDataRec theChallengeData;
} HostChallengeRec, *HostChallengePtr;
```

theChallenge sends the challenge type. Send 'SERL' in this field to access the rightmost 7 characters of the host's serial string.

theChallengeData sends the challenge data record and receives the challenge response.
The challenge data record contains an 8 byte challenge or response and is defined as follows:

```
typedef union {
        unsigned char b[8];
        struct {
                unsigned char b0, b1, b2, b3, b4, b5, b6, b7;
        } b0;
        struct {
                short i0, i1, i2, i3;
        } i;
        unsigned char s[8];    /* Pascal 7 character string (length + seven characters) */
        struct {
                SInt32 l0, l1;
        } l;
} ChallengeDataRec, *ChallengeDataPtr;
```

The plug-in's copy protection can be implemented by requesting the host's serial string. The serial string contains the rightmost seven characters of the host's serial number string (which is usually only seven characters in length). The plug-in can use this information to create an authorization code for the copy of the plug-in. Animator and Camera have a unique serial number for each licensed copy of the animation system. The plug-in can request an authorization code from the user the first time it is run and then store the code into its own resource file. The next time the plug-in is opened, it can simply check to make sure that the serial string and authorization codes match.

Note: EIAS3D users can exploit any numbers of slave Cameras and do render without hardware dongle at all. Therefore a plug-in must check copy-protection only in Animator, not in Camera.

14.5 HostGetTimingInfo

This command tells the host to return the most recent animation timing information.

```
#define hostGetTimingInfo 'TINF '
typedef struct {
        Ptr theReference;
        double theTimingStart;
        double TheTimingCurrent;
        double theTimingFPS;
        SInt32 theTimingInterlace;
        SInt32 theTimingTotal;
        SInt32 theTimingSkip;
        SInt32 theRenderFrameStart;
        SInt32 theRenderFrameStop;
} HostGetTimingInfoRec, *HostGetTimingInfoPtr;
```

theTimingStart sends the system's start time of the animation. The default is 0.0, but the user may change it by dragging the start time icon in the time bar.

theTimingCurrent sends the system's current time.

theTimingFPS sends the system's frame-per-second factor. Video rates is usually 30 frames-per-second, or 60 fields-per-second (fields are treated as frames in EI). Film rates are 24 frames-per-second. The host may have a different FPS count, so the plug-in should call this command as frequently as possible.

theTimingInterlace sends the system's interlace option. 0 is no interlace, while 1 specifieds interlacing.

theTimingTotal sends the system's total number of frames. The stop time of an animation is computed as follows:

stopTime = theTimingStart + (theTimingTotal * theTimingFPS)

theTimingSkip sends the system's skip frames of an animation. The user may instruct the animation to skip frames during rendering. This is 0 if all frames are rendered, or positive if frames need to be skipped. For example, a value of 3 only renders every fourth frame.

theRenderFrameStart sends the system's frame start of a rendering. A value of 0 means that the animation starts at theTimingStart

theRenderFrameStop sends the system's frame stop of a rendering. A value of theTimingTotal means that the animation stops at the end of the animation.

NOTE: If theRenderFrameStart and theRenderFrameStop are the same value, look at theTimingCurrent for the current time to be rendered (single frame only).

This command is valuable to inform the plugin about the current state of the system's a n imation timing track. Usually, the plug-in might want to use this command before generating any model data to check how much needs to be done.

14.6 HostStatus

This command tells the host to display or update the status box and to handle background events or allow the user to abort the current plug-in command.

```
#define hostStatus 'STAT'
typedef struct {
        Ptr theReference;
        float theStatusPos;
        Str255 theStatusUnit;
        Str255 theStatusInfo;
        SInt32 theStatusMin;
        SInt32 theStatusMax;
        SInt32 theStatusValue;
        SInt32 theStatusTick;
} HostStatusRec, *HostStatusPtr;
```

theStatusPos sends the status box's completion percentage. This represents the percentage of the current plug-in operation which has been completed. The value can range between 0.0 and 1.0.

theStatusUnit sends the status box's unit name.

theStatusInfo sends the status box's information string. This will tell the user what operation the plug-in is currently performing. The plugin may need to perform several operations in order to complete a single command.

theStatusMin sends the status box's minimum value. This value is displayed at the left side of the status bar.

theStatusMax sends the status box's maximum value. This value is displayed at the right side of the status bar.

theStatusValue sends the status box's current value. This value should be between the status box's minimum and maximum values. It is used as a display value only and to calculate the status bar's position.

theStatusTick sends zero to change the status box's contents. Returns the next status update tick. The plug-in should send another hostStatus command after TickCount() is greater than theStatusTick.

When a plug-in must process data for more than 1/3rd of a second, it should call the *hostStatus* command. This command tells the host to display a status dialog box to tell the user how much of the current operation is complete. It also allows background events to occur and gives the user a chance to abort the current plug-in command. When the plug-in first calls the hostStatus command or when it wishes to change the status strings, it should set *theStatusTick* to zero. The host returns the next status event tick in *theStatusTick*. The plug-in should send another *hostStatus* command after this tick. If the plug-in does not send any *hostStatus* commands and processes information for a long time, the user may believe that the plug-in has crashed.

14.7 HostSynchronize

This command is available in Animator only. It tells host to synchronize ALL project's animation channels with specified frame or time.

```
#define hostSynchronize 'SYNC'
typedef struct {
        char *theReference;
        SInt32 theTimingType;
        SInt32 theTimingFrame;
        double theTimingTime;
} HostSynchronizeRec, *HostSynchronizePtr;
```

theTimingType sends the type of synchronization. The valid values are:

0 - current time
1 - by frame
2 - by time

theTimingFrame sends the frame to which the groups are synchronized. This value is used when theTimingType is set to 1 (by frame)

theTimingTime sends the time to which the groups are synchronized. This value is used when *theTimingType* is set to 2 (by time)

HostSynchronize is a powerful call that allows plug-ins to create interested and original effects. In fact a plug-in can extract "a whole history" of any project's element, not limited by its stage at actual frame. However this call should be used carefully, the rules are:

- If a plug-in modified actual project's time/frame by calling one or more *HostSynchronize* calls, the plug-in should finally call *HostSynchronize* again with *theTimingType* = 0 (synchronize to current time). Otherwise project remains in undefined stage.

- *HostSynchronize* is applied for ALL project's elements (not for plug-in's child groups only), thus it can take a valuable time for large projects.

- *HostSynchronize* does not apply non-linear transforms such as deformations, morphs and others. For example you can't extract an object's vertices/facets deformed at specified frame.

14.8 HostModelGetWeight

This command is available in Animator only. It retrieves information about weight maps applied to the specified group.

```
#define hostModelGetWeight 'GETW'
typedef struct {
        Ptr                     theReference;
        char                    *theGroup;
        SInt32                  theWeightMapIndex;
        PStr255                 theWeightMapName;
        unsigned char*          theWeightMapData;
} HostModelGetWeightRec, *HostModelGetWeightPtr;
```

theGroup sends the requested weight's group pointer

theWeightMapIndex sends the index (one-based) of weight map to retrieve. The passed 0 (zero) value returns the total number of weight maps (in same theWeightMapIndex field).

theWeightMapName returns the name of the weight map.

theWeightMapData returns the data of the weight map as normalized unsigned byte (0 is 0.0, and 255 is 1.0). The count of weight values is always equal to count of group's vertices.

Typically plug-ins do extract weight map information to distribute/weight their effects over group's surface. For example vertices with weight map values = 255 (maximal) are fully affected by the plug-in. The vertices with weight map values = 0 are not affected at all. A blended portion is applied for middle values and so on.

The modifying of *theWeightMapData* is not disabled but typically has no constructive goals. Note that modifications will not be displayed by host immediately. For groups that are plug-ins themselves the weight map information can be unstable because such plug-ins groups can change created geometry according to their settings.

14.9 HostModelGetGroupData/HostModelSetGroupData

This command is available in Camera only. It tells the host to bind a memory block to the specified group in order to use it (at render time) for other plug-ins and/or shaders.

```
#define hostModelGetGroupData  'GETD'
#define hostModelSetGroupData  'SETD'
typedef struct {
        Ptr     theReference;
        char    *theGroup;
        char    *thePluginData;
        UInt32 thePluginID;
} HostModelSetGroupDataRec, *HostModelSetGroupDataPtr;
```

theGroup sends the group pointer (obtained via *hostGetGroup/hostAddGroup*) to bind the data to.

thePluginData sends a pointer to the block to bind. The block should be allocated via the *hostPermAlloc* call.

thePluginID sends the group's plug-in ID, typically used by a data receiver to validate.

Note: in past this command was often used to pass geometry data (vertices and facets) to shaders. This method is no longer necessary. Shaders can extract geometry data directly themselves (see "V9 Shaders API.pdf" for details). Nevertheless this service is not obsolete because it allows the passing of non-trivial data between plug-ins and/or shaders.

Electric Image Animation System

15. Plug-in Calls to access project's objects

This set of commands is available in Animator only. It allows plug-ins to retrieve data of any object in Project Window. Objects without geometry data, such as lights, effectors etc. can be accessed as well.

Typically a plug-in first gets an unique ID of object to pass it in next calls. The ID can be obtained via *HostObjectInfo/ HostObjectAnimatedInfo* calls. It is the same ID returned by HostGetGroup call for plug-in's child groups, and the same ID that should be passed in theChannelOwner field to read/write animation channels.

15.1 HostObjectInfo/HostObjectAnimatedInfo

These commands retrieve information of any object (*hostObjectInfo*) or of animated object only (*hostObjectAnimatedInfo*)

```
#define hostObjectInfo                  'UOBJ'
#define hostObjectAnimatedInfo          'KOBJ'
typedef struct {
        Ptr     theReference;
        SInt32  theReferenceIndex;

        char*   theObject;
        SInt32  theObjectType;
        PStr255 theObjectName;
        SInt32  theObjectIndex;
        char*   theObjectParent;
        SInt32  theObjectFlag;
} HostObjectInfoRec, *HostObjectInfoPtr;
```

theReferenceIndex sends the index of the object in list (1 based). The 0 (zero value) returns total number of objects in the *theObjectIndex* field.

theObject returns unique ID of the object that should be stored to use in other calls.

theObjectType returns the object type, such as camera, light, etc.

theObjectName returns the object name as a Pascal string

theObjectIndex returns the object's index or, if *theReferenceIndex* is set to 0, total number of objects.

theObjectParent returns the ID of object's parent

theObjectFlag returns the object's flags. The flags are:

```
#define OBJECTINFO_FLAG_VISIBILITY                      0
#define OBJECTINFO_FLAG_ANIMATABLE                      1
#define OBJECTINFO_FLAG_SELECTED                        2
```

15.2 HostObjectTransform

This command retrieves a transformation matrix of the specified project's object.

```
#define hostObjectTransform 'TOBJ'
typedef struct {
        Ptr     theReference;
        char    *theTransformOwner;
        Matrix4 theTransform;
} HostObjectTransformRec, *HostObjectTransformPtr;
```

theTransformOwner sends the object ID of the requested object.

theTransform returns the global transformation of the object. The matrix contains all object's linear transformations from original (model) space to actual one, counting all transforms made by all object's parents and object's itself.

15.3 HostGetNameInfo/HostSetNameInfo

These calls get/set the project element's name respectively.

```
#define hostGetNameInfo 'GNAM'
#define hostSetNameInfo 'SNAM'
typedef struct {
        Ptr     theReference;
        char    *theNameOwner;
        PStr255 theName;
} HostObjectNameRec, *HostObjectNamePtr;
```

theNameOwner sends the object pointer of the requested object.

theName gets or sets the name of the object.

Electric Image Animation System

15.4 HostAddObject

This command adds a new project's element. See 'FactStuff.h' for detailed fields description

```
#define hostAddObject   'AOBJ'
typedef struct {
        Ptr theReference;
        char *theObject;                /*  ==> Returns the object pointer of the newly created
                                        object */
        char *theObjectParent;          /* <==  Sends the parent of the object about to be created
                                        (must be specified - default is the plugin group) */
        char *theObjectSibling;         /* <==  Sends the sibling of the object about to be
                                        created (must be specified - if not siblings use the parent) */

        SInt32  theObjectType;          /*  ==> Sends the object type (camera, light, etc.) to be
                                        created (no groups please) */
        SInt32  theObjectFlag;          /* <==  Sends the inheritance flags (see FACTStuff) */
        SInt32  theObjectHierarchy;     /* <==  Sends the hierarchy method (see Link window) */
        PStr255 theObjectName;          /* <==  Sends the name of the object. */
        DCoordinate theObjectWorldSize;/* <==  Sends the display size of the object in world space */

        SInt32  theObjectRotationOrder;/* <==  Sends the rotation order for the rotation
                                        transformation */

        DCoordinate theObjectLocalOffset;/* <==  Sends the local coordinate system origin (see
                                            linkage window) */
        DCoordinate theObjectLocalRotation;/* <==  Sends the local coordinate system orientation
                                            (see linkage window) */
        DCoordinate theObjectParentScale;/* <==  Sends the starting object scale */
        DCoordinate theObjectParentOffset;/* <==  Sends the starting object offset */
        DCoordinate theObjectParentRotation;/* <==  Sends the starting object rotation */

        SInt32  theConstraintType;      /* <==  Sends the constraint type (see FACTStuff) */
        SInt32  theConstraintFlag;      /* <==  Sends the constraint flags (see FACTStuff) */

        DCoordinate theConstraintLimitMinRotation;/* <==  Sends the constraint minimum rotation
                                                (see linkage window) */
        DCoordinate theConstraintLimitMaxRotation;/* <==  Sends the constraint maximum rotation
                                                (see linkage window) */
        DCoordinate theConstraintLimitCenterRotation;/* <==  Sends the constraint center
                                                rotation (see linkage window) */
        DCoordinate theConstraintLimitStiffness;/* <==  Sends the constraint rotation stiffness
                                                (see linkage window) */
        DCoordinate theConstraintLimitViscosity;/* <==  Sends the constraint rotation viscosity
                                                (see linkage window) */

        DCoordinate theConstraintLimitMinPosition;/* <==  Sends the constraint minimum position
                                                (see linkage window) */
        DCoordinate theConstraintLimitMaxPosition;/* <==  Sends the constraint maximum position
                                                (see linkage window) */

        DCoordinate theConstraintGravity;/* <==  Sends the constraint local gravity (see linkage
                                                window) */

} HostObjectCreateRec, *HostObjectCreatePtr;
```

The objects with geometry (groups or plug-ins) cannot be added. There is no way for plug-ins to delete the created objects. Therefore it's recommended to keep added objects organized, for example link all them to a single effector to make eventual removing easier for user.

15.5 HostModelGLNK

This command retrieves object's linkage information. See 'FactStuff.h' for detailed GLNK structure description.

```
#define hostModelGLNK 'GLNK'
typedef struct {
        Ptr     theReference;
        char *  theGroup;       /* <==  Sends the group pointer of the requested record's group  */
        GLNKPtr theGLNK;        /*  ==> Returns the pointer to the group link record */
} HostModelGLNKRec, *HostModelGLNKPtr;
```

15.6 HostModelGroupLimits

This command retrieves a specific information about object's transformation. See 'FactStuff.h' for details.

```
#define hostModelGroupLimits 'GLMT'
typedef struct {
        char *theReference;             /* <==  Sends the private data used by host callbacks */
        char *theGroup;                 /* <==  Sends the group pointer of the requested record's group */

        XCoordinate thePositionMinLimit; /* ==> returns the position minimum limit */
        XCoordinate thePositionMaxLimit; /* ==> returns the position maximum limit */

        XCoordinate theRotationMinLimit; /* Rotation minimum limit */
        XCoordinate theRotationMaxLimit; /* Rotation maximum limit */

        XCoordinate theRotationCenter; /* Rotation center */
        XCoordinate theStiffness;      /* Stiffness */
        XCoordinate theViscosity;      /* the Viscosity */

        UInt32 theFlag;                /* the Flags */
} HostModelGroupLimitsRec, * HostModelGroupLimitsPtr;
```

15.7 Obsolete calls from Plug-Ins to the Host

The following calls are obsolete. They have no effect and should not be longer used.

```
#define hostBackground          'BACK'
#define hostOpenResource        'ORES'
#define hostCloseResource       'CRES'
```

Electric Image Animation System

16. The PluginProjectInfo interface

This interface is available in Animator only. It allows plug-ins “listen” what’s going on in project and do involve into interaction process. It does not mean a plug-in is called for low-level user’s input, such as mouse clicks and keyboard typing. Instead the information about high-level actions are passed to a plug-in:

- Project is going to synchronize with new animation time;
- A new object is added to project, or existed one is deleted or renamed;
- Selection is changed in the project;
- One or more animation channels are changed;
- Project’s hierarchy structure is changed.

Note that a single input action can produce a chain of further secondary actions. For example if user has moved an object, it can affect other objects that are constrained to it, the modified objects can produce others transformations and so on.

A plug-in is notified after all actions processing are finished by host. The data of made actions are passed to plug-ins in a form of arrays.

16.1 PluginProjectInfoRec structure

If a plug-in needs to use the *PluginProjectInfo* interface, it must set *pluginProjectInfoFlag* (bit 29) when it receives *PluginInformation* host call. It tells host to call plug-in with *pluginProjectInfo* command and *PluginProjectInfoRec* structure.

```
#define pluginProjectInfo      'PINF'
typedef struct PluginProjectInfoRec {
        PluginInfoRec theInfo;
        SInt32  theVersion;

        SInt32  theFrameIndex
        double  theFrameTime;
        double  theFrameTimeDelta

        SInt32  theSessionID;
        SInt32  theProjectInfoMode;

        SInt32  theObjectNum;
        HostObjectInfoRec * theObject;

        SInt32  theActionNum;
        TSActionRec * theAction;

        SInt32  theChannelNum;
        TSChannelRec * theChannel;

        TSUserListRec * theUserList;
} PluginProjectInfoRec, *PluginProjectInfoPtr;
```

theVersion sends the version of this *PluginProjectInfoRec*

theFrameIndex sends the current frame index number

theFrameTime sends the time of the current frame

theFrameTimeDelta sends the time per frame (1/fps)

theSessionID sends the actual session ID. See more details in section “Modification Loop” below.

theProjectInfoMode sends the mode. It’s a reason why this *PluginProjectInfo* is called. The valid modes are:

```
#define projectInfoMode_Update        0
#define projectInfoMode_Track         1
#define projectInfoMode_Animation     2
#define projectInfoMode_Selection     3
```

projectInfoMode_Update - the call was made after project is updated (for example user has changed one or more UI values) or the project is synchronized with new animation time;

projectInfoMode_Track - user tracks object(s) in one of Animator’s windows;

projectInfoMode_Animation - the animation/preview is processing

projectInfoMode_Selection - the selection is changed in Animator’s Project Window. To receive this mode the plug-in should set plugin*ProjectSelectionFlag* when it receives *PluginInformation* host call.

theObjectNum sends the count of objects in following *theObject* array

theObject send a pointer to the array of objects involved into modification process. The count of elements is defined by *theObjectNum*

theActionNum sends the count of actions in following *theAction* array

theAction sends a pointer to the array of actions processed by host. The *TSActionRec* is defined as:

```
typedef struct {
        char *  theParent;          /* The parent node     */
        char *  theChild;           /* The child node      */
        SInt32  theAction;          /* The action          */
} TSActionRec, *TSActionPtr;
```

Where theAction can be one of following:

```
#define  tsa_Hierarchy          1          /* The Hierarchy dependency          */
#define  tsa_IKInverse          2          /* The IK Handler                    */
#define  tsa_IKInverseEnd       3          /* The IK Handler End                */
#define  tsa_Master             4          /* The Master Light, Material        */
#define  tsa_Deform             5          /* not used                          */
#define  tsa_Skin               6          /* The Skinizer                      */
#define  tsa_Constraint         10000      /* The Constrain                     */
#define  tsa_Expression         20000      /* not used                          */
#define  tsa_ObjectCreate       20001      /* The object is created             */
#define  tsa_ObjectDelete       20002      /* The object is removed             */
#define  tsa_ObjectRename       20003      /* The object is renamed             */
#define  tsa_ObjectDuplicate    20004      /* The object is duplicated          */
#define  tsa_PluginRequest      20005      /* The model plug-in finished        */
#define  tsa_ObjectVisibility   20006      /* The visibility is changed         */
#define  tsa_ObjectAnimation    20007      /* The object’s animation is changed */
#define  tsa_ObjectSelected     20008      /* The object becomes selected       */
#define  tsa_ObjectDeselected   20009      /* The object becomes deselected     */
```

theChannelNum sends the count of channels in following *theChannel* array

theChannel sends the pointer to the input channels array. The *TSChannelRec* is defined as:

```
typedef struct {
        char * theChannel;              /* The channel reference                */
        char * theOwner;                /* The channel's owner                  */
} TSChannelRec, *TSChannelPtr;
```

theUserList sends the list of data added by plug-ins. The *TSUserListRec* is defined as:

```
typedef struct TSUserListRec {
        IFFType thePluginID;            /* The ID of the plug-in adds this element */
        SInt32  thePluginDataSize;      /* The size of private plug-in's data      */
        struct  TSUserListRec * theNext; /* The next data element                  */
        char    thePluginData[4];       /* The private plug-in's data starts here  */
} TSUserListRec, *TSUserListPtr;
```

A plug-in can allocate a new *TSUserListRec* and append it to the end of list. The host guarantees these data are available for other plug-ins. Scanning *theUserList* a plug-in can know about others plug-ins and do communicate with them.

16.2 Modification Loop

Simple plug-ins do use *PluginProjectInfoRec* just to read interested information, such as which animation channels are modified, which objects are deleted and so on. Advanced plug-ins can modify animation channels in response. It tells host to start a new round of modification and a new *pluginProjectInfo* is passed to the plug-in. The loop continues until no more modifications done. Each round is identified by *theSessionID* field of *PluginProjectInfoRec*. The zero *theSessionID* value means first round, caused by user's activity. The plug-in should examine *theSessionID* to avoid repeated actions and circular dependencies.

16.3 HostProjectInfoRequest

The plug-in can send this command to host to initiate a new *PluginProjectInfo* call.

```
#define hostProjectInfoRequest 'PIRQ'
typedef struct {
        char *theReference;
} HostProjectInfoRequestRec, *HostProjectInfoRequestPtr;
```