lw

COLLABORATORS			
	TITLE : lw		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 29, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 lw **1**

1.1 S0 . . . 1
1.2 s1 . . . 4
1.3 s1.1 . . . 5
1.4 s1.2 . . . 5
1.5 s2 . . . 6
1.6 s2.1 . . . 6
1.7 s2.2 . . . 6
1.8 s2.2.1 . . . 7
1.9 s2.2.2 . . . 8
1.10 s3 . . . 9
1.11 s3.1 . . . 9
1.12 s3.2 . . . 9
1.13 s3.3 . . . 9
1.14 s4 . . . 10
1.15 s4.1 . . . 10
1.16 s4.2 . . . 12
1.17 s4.2.1 . . . 13
1.18 s4.2.2 . . . 14
1.19 s5 . . . 14
1.20 s5.1 . . . 14
1.21 s5.1.1 . . . 15
1.22 s5.1.2 . . . 15
1.23 s5.1.3 . . . 16
1.24 s5.1.4 . . . 16
1.25 s5.1.5 . . . 17
1.26 s5.1.6 . . . 18
1.27 s5.1.7 . . . 19
1.28 s5.1.8 . . . 20
1.29 s5.1.9 . . . 20

1.30 s5.1.10 . . . 21
1.31 s5.2 . . . 22
1.32 s5.2.1 . . . 22
1.33 s5.2.2 . . . 23
1.34 s5.2.3 . . . 26
1.35 s6 . . . 27
1.36 s6.1 . . . 27
1.37 s6.2 . . . 28
1.38 s6.2.1 . . . 29
1.39 s6.2.2 . . . 30
1.40 s6.3 . . . 31
1.41 s6.4 . . . 31
1.42 s6.5 . . . 32
1.43 s6.6 . . . 32
1.44 s6.7 . . . 33
1.45 s7 . . . 34
1.46 s7.1 . . . 34
1.47 s7.2 . . . 35
1.48 s7.3 . . . 36
1.49 s7.4 . . . 36
1.50 s7.5 . . . 36
1.51 s7.5.1 . . . 37
1.52 s7.5.2 . . . 38
1.53 s7.5.3 . . . 38
1.54 s8 . . . 39
1.55 s8.1 . . . 39
1.56 s8.2 . . . 39
1.57 s8.2.1 . . . 40
1.58 s8.2.2 . . . 41
1.59 s8.2.3 . . . 42
1.60 s8.3 . . . 42
1.61 s8.3.1 . . . 43
1.62 s8.3.2 . . . 43
1.63 s8.3.3 . . . 45
1.64 s8.3.4 . . . 45
1.65 s8.3.5 . . . 46
1.66 s8.4 . . . 47
1.67 s8.4.1 . . . 47
1.68 s8.4.2 . . . 48

1.69 s8.5 48
1.70 s8.5.1 48
1.71 s8.5.2 49
1.72 s8.6 49
1.73 s8.6.1 50
1.74 s8.6.2 51
1.75 s8.7 51
1.76 s8.8 52
1.77 s8.9 53
1.78 s8.10 54
1.79 s9 54
1.80 s9.1 54
1.81 s9.2 56
1.82 s9.3 56
1.83 s9.4 57
1.84 s9.5 58
1.85 s9.6 58
1.86 s9.7 60
1.87 s9.8 62
1.88 s9.9 62
1.89 s10 63

Chapter 1

lw

1.1 S0

```
LightWave 3D Animation and Modeling Plug-ins -- Stuart Ferguson
6/28/95

1  Introduction
    1.1  Pre-Release Compatibility
    1.2  Final 4.0 Compatibility

2  Common Server Classes
    2.1  Utility Servers
    2.2  Object Import
       (1)  Common Server Classes
  2.2.1  Sending Mesh Data
     (2)  Common Server Classes
  2.2.2  Result Codes
     (3)  Common Server Classes

3  Common Globals
    3.1  Utility Globals
    3.2  File Requester
       (4)  Common Globals
    3.3  User Messages
       (5)  Common Globals

4  Modeling Datatypes
    4.1  Dynamic Types
       (6)  Modeling Base Types
       (7)  Modeling Types
       (8)  Modeling Types
       (9)  Modeling Types
       (10)  Modeling Types
       (11)  Modeling Types
       (12)  Modeling Types
       (13)  Modeling Types
       (14)  Modeling Types
    4.2  Element Operation Filters
  4.2.1  Layer Filters
     (15)  Modeling Base Types
  4.2.2  Element Filters
```

(16) Modeling Base Types

5 Modeling Server Classes
5.1 Mesh Editing
(17) Modeling Base Types
5.1.1 Local Data
(18) Modeling Servers
5.1.2 Getting Element Information
(19) Modeling Types
5.1.3 PointInfo
(20) Modeling Base Types
5.1.4 PolygonInfo
(21) Modeling Base Types
(22) Modeling Types
5.1.5 Main Data Struct
(23) Modeling Types
(24) Modeling Types
5.1.6 Error Codes
(25) Modeling Base Types
5.1.7 Query Functions
(26) Mesh Edit Count functions
(27) Modeling Types
(28) Mesh Edit Query functions
5.1.8 Element Traversal
(29) Modeling Base Types
(30) Mesh Edit Enumeration functions
5.1.9 Creating New Elements
(31) Modeling Base Types
(32) Mesh Edit Create functions
5.1.10 Modifying Existing Elements
(33) Mesh Edit Modify functions
5.2 Command Sequencing
(34) Modeling Types
5.2.1 Command Activation
(35) Modeling Servers
5.2.2 Commands
5.2.3 External Activation on Windows
(36) Trigger code

6 Modeling Globals
6.1 Dynamic Conversion
(37) Modeling Globals
(38) Modeling Types
(39) Modeling Types
6.2 Dynamic Requester
(40) Modeling Globals
6.2.1 Requester Usage
(41) Modeling Types
6.2.2 Control Descriptors
(42) Modeling Types
(43) Modeling Types
(44) Modeling Types
(45) Modeling Types
6.3 Dynamic Monitor
(46) Modeling Globals
6.4 Custom Commands
(47) Modeling Globals

6.5 Modeler Internal State
(48) Modeling Globals
6.6 Surfaces List
(49) Modeling Globals
6.7 Outline Font List
(50) Modeling Globals

7 Animation Datatypes
7.1 Coordinate and Range Scales
(51) Animation Types
7.2 Items and Properties
(52) Animation Types
(53) Animation Types
(54) Animation Types
7.3 Time
(55) Animation Types
7.4 Errors
(56) Animation Types
7.5 Instances and Handlers
(57) Animation Types
7.5.1 Instance Persistence
(58) Animation Types
(59) Animation Types
7.5.2 Handler Functions
(60) Animation Types
7.5.3 Interface Server

8 Animation Server Classes
8.1 Utilities
8.2 Image Post Processing
8.2.1 Input Buffers
(61) Animation Servers
8.2.2 Filter Access
(62) Animation Servers
8.2.3 Handler
(63) Animation Servers
8.3 Procedural Texture
8.3.1 Shader Access
(64) Animation Servers
8.3.2 Geometric Parameters
(65) Read-only shader parameters
8.3.3 Modifiable Parameters
(66) Modifiable shader parameters
8.3.4 Shading Functions
(67) Shader functions
8.3.5 Instance
(68) Animation Servers
(69) Animation Servers
8.4 Procedural Displacement Map
8.4.1 Displacement Access
(70) Animation Servers
8.4.2 Handler
(71) Animation Servers
8.5 Procedural Item Animation
8.5.1 Item Motion Access
(72) Animation Servers
8.5.2 Handler

(73) Animation Servers
8.6 Procedural Object Replacement
8.6.1 Object Replacement Access
(74) Animation Servers
(75) Animation Servers
8.6.2 Handler
(76) Animation Servers
8.7 Frame Buffers
(77) Animation Servers
8.8 Animation Output
(78) Animation Servers
8.9 Scene Conversion
(79) Animation Servers
8.10 General Function

9 Animation Globals
9.1 Item Information
(80) Animation Types
(81) Animation Globals
9.2 Object Information
(82) Animation Types
(83) Animation Globals
9.3 Bone Information
(84) Animation Types
(85) Animation Globals
9.4 Light Information
(86) Animation Types
(87) Animation Types
(88) Animation Globals
9.5 Camera Information
(89) Animation Globals
9.6 Scene Information
(90) Animation Types
(91) Animation Globals
(92) Animation Globals
9.7 Image List Information
(93) Animation Types
(94) Animation Globals
9.8 Compositing Information
(95) Animation Globals
9.9 Global Rendering Memory Pool
(96) Animation Types
(97) Animation Globals

10 Files
(98) Common LightWave Header
(99) LightWave Modeler Plug-in Header
(100) LightWave Rendering and Animation Plug-in Header

1.2 s1

This document describes the plug-in interfaces defined for the LightWave 3D animation and modeling programs. The two programs which make up the LightWave 3D suite each have different plug-in interfaces to allow access to their internal state and functions, as well as some

common interfaces which are shared between the two. These common interfaces are the subject of the first portion of this document. After the common interfaces, the plug-in interfaces specific to modeling and animation are described.

The reader should be familiar with the basic concepts of the LightWave plug-in design, such as server classes, local data, and global data. These are described in the document entitled "LightWave Plug-in Architecture," and should be considered a prerequsite to this document. In addition, some of the common server classes are defined in other documents which will be referenced.

The LightWave animation program will be called 'Layout' and the modeling program will be called 'Modeler' throughout this document.

1.1 Pre-Release Compatibility
1.2 Final 4.0 Compatibility

1.3 s1.1

LightWave 4.0 was released commercially in "pre-release" form in April 95. The pre-release Modeler fully supported the plug-in interface as it was defined at that time. The pre-release Layout only supported a small fraction of the interface defined for it:

The ImageFilterHandler class was enabled, but with no instance saving or loading and RGBA buffers only.

The ShaderHandler class was enabled, but with no instance saving or loading and no rayTrace or illuminate functions.

The SceneInfo global was fully available.

The ImageList global was available but without load or spot functions.

Since then development has continued both on the programs and on the plug-in interface. While most of the changes are backward compatible with the pre-release, a few are not. If you are using the pre-release, you will have to define the LW_PRERELEASE symbol to remove anything from this header that is incompatible with the pre-release versions of Layout and Modeler.

1.4 s1.2

This SDK goes with the LightWave 4.0 release. Modeler 4.0 implements all the features of the interface described in this document. While the interface for Layout 4.0 is described by this document, it varies slightly from the design given in beta versions of the specification.

The item paramters W_RIGHT, W_UP and W_FORWARD have been

removed. These can be calculated by inverting the matrix given by the values for RIGHT, UP and FORWARD.

The global Envelope Handler has been removed.

The ObjReplacementAccess field curType is always set to OBJREP_NONE and the fields curFrame, curTime and curFilename are not set.

The Item Motion callback 'getParam' only returns values for the directly keyframable values POSITION, ROTATION and SCALING. Other parameter values may be read using the global item info callbacks.

1.5 s2

These are the common server classes defined for both programs in the LightWave 3D suite. There are interfaces for these server classes present in Layout and Modeler and any plug-in of one of these classes may be shared by both programs.

2.1 Utility Servers
2.2 Object Import

1.6 s2.1

The "FileRequester" and "ImageLoader" utility server classes are used by both LightWave and Modeler. The interfaces for these server classes are defined in the "File Requester Plug-ins" and "LightWave Images" documents.

1.7 s2.2

When Layout or Modeler encounters a foreign object file which it cannot parse, it will call an "ObjectLoader" class server to import it. All the loaders defined for the host will be activated in sequence, and the first one to recognize the file will load it. The order in which loaders are called is not defined, although it may be alphabetical by server name.

At activate, an ObjectImport structure is passed to a plug-in object loader as its local data, and the loader should attempt to parse the input file given by the filename field. If it cannot open or recognize the file the loader should set the 'result' field to the appropriate code and return.

If it recognizes the file type it should send the mesh and surface data to the host by calling the callbacks. The 'data' field is an opaque pointer to some internal state for the host and should be the first argument to every callback. The 'monitor' field will contain a

pointer to a monitor which can be used to track the progress of loading. The monitor should not be used unless the object format is recognized.

(1) Common Server Classes

```
typedef struct st_ObjectImport {
  int               result;
  const char       *filename;
  Monitor          *monitor;
  char             *failedBuf;
  int               failedLen;

  void             *data;
  void            (*begin) (void *, void *);
  void            (*done) (void *);
  void            (*numPoints) (void *, int total);
  void            (*points) (void *, int numPts,
          const float *xyz);
  int             (*surfIndex) (void *, const char *name,
             int *firstTime);
  void            (*polygon) (void *, int numPts, int surf,
           int flags,
           const unsigned short *);
  void            (*surfData) (void *, const char *name,
            int size, void *data);
} ObjectImport;
. . .
```

2.2.1 Sending Mesh Data
2.2.2 Result Codes

1.8 s2.2.1

Sending mesh data to the host involves calling the functions provided in the ObjectImport structure in a semi-sequential order. The basics are to start the data transfer, send the points, define surface names, send the polygons, assign surface parameters to names, and complete the transfer.

Begin Callback 'begin' is called to mark the start of new mesh data. The second argument is for special information and should normally be null. (It might be possible to call this more than once, although 'done' would have to be called before calling 'begin' a second time.)

Points Callback 'numPoints' is called with the total number of points. Then 'points' is called with 1 or more point coordinates until the total number of points is reached. Points are numbered from zero in the order added, and that implicit index is used to create polygons. All points must be added before any polygons may be created.

Surfaces The callback 'surfIndex' is called with a surface name to get a surface ID number for that surface. This ID number

is used to create polygons. The function may optionally return a boolean flag to indicate if this is the first time this surface name has been given an ID.

Polygons For each polygon, the 'polygon' function is called with a list of point indices for the polygon, the number of points, mode flags and a surface index. The mode flags word is a collection of bits. If the CURVE bit is set, this is a curve rather than a face. If the DETAIL bit is set, then this polygon is a detail of the last top-level polygon. If the STARTCC or ENDCC bits are set, then this curve has start and/or end points which are continuity control points.

(2) Common Server Classes

```
. . .
#define OBJPOLF_FACE      0
#define OBJPOLF_CURVE    (1<<0)
#define OBJPOLF_DETAIL   (1<<1)
#define OBJPOLF_STARTCC  (1<<2)
#define OBJPOLF_ENDCC    (1<<3)
. . .
```

Surface Data
A block of raw surface parameters may be assigned to a name at any time by calling the 'surfData' call with the surface name and byte block.

Done Callback 'done' is called when data transfer is complete.

If a failure occurs partway through loading a file, the loader can set the result field and return without having to do any other cleanup.

1.9 s2.2.2

The loader must set the 'result' field to one of these following values before it returns. OK indicates successful parsing of the object file. BADFILE indicates that the loader could not open the file. NOREC indicates that the loader could not recognize the format, and ABORTED indicates the that the user manually aborted the load. Any other failure is indicated by the generic FAILED value. In this case, the loader may also place a human-readable error message into the buffer pointed to by 'failedBuf,' provided that 'failedLen' is non-zero.

(3) Common Server Classes

```
. . .
#define OBJSTAT_OK       0
#define OBJSTAT_NOREC    1
#define OBJSTAT_BADFILE  2
#define OBJSTAT_ABORTED  3
#define OBJSTAT_FAILED   99
```

1.10 s3

This section contains descriptions of the data pointers which can be accessed by passing specific global ID strings to the global functions of both Modeler and Layout. Common servers can access these globals regardless of which program they are running under.

3.1 Utility Globals
3.2 File Requester
3.3 User Messages

1.11 s3.1

The global ID "Host Display Info" returns a HostDisplayInfo structure initialized for the host application's main window. This structure is described in the "LightWave Plug-in Architecture" document and defined in the 'hdisp.h' header file.

The global ID "File Type Pattern" returns a file type function used to get filename filters for different file types. This is used by the file requester class primarily and is described in the "File Requester Plug-ins" document.

1.12 s3.2

The global ID "File Request" returns a 'FileReqFunc' pointer. Servers can use this function to request filenames from users with the same file requester used by the host. The 'hail' string is the title of the request and the 'name' & 'path' buffers should be filled in with the starting base name and path for the request. These buffers will be modified and the 'fullName' buffer filled with the final complete name for the user-selected file. 'bufLen' is the length of all the passed buffers. The function returns 0 if the user elected to cancel, 1 if they hit Ok, and negative values for any errors.

(4) Common Globals

```
typedef int             FileReqFunc (const char *hail, char *name,
            char *path, char *fullName,
            int buflen);
. . .
```

1.13 s3.3

The global ID "Info Messages" returns a pointer to a MessageFunc structure which provides simple functions for displaying messages to the user. The functions will display different types of messages, each with one or two lines of text. The second string argument can be null for one-line messages.

```
    (5)  Common Globals
  . . .
  typedef struct st_MessageFuncs {
    void            (*info)    (const char *, const char *);
    void            (*error)   (const char *, const char *);
    void            (*warning) (const char *, const char *);
  } MessageFuncs;
```

1.14 s4

The servers and globals for Modeler share a set of type and value definitions. These are basic to an understanding of the Modeler servers and globals.

```
      4.1  Dynamic Types
      4.2  Element Operation Filters
```

1.15 s4.1

Dynamic Values are values of variable type. Unlike normal C types which have a fixed interpretation, dynamic values have a type which can vary according to what is needed. The possible types for a dymamic value are given by the following definitions.

```
    (6)  Modeling Base Types

  typedef int             DynaType;
  #define DY_NULL         0
  #define DY_STRING       1
  #define DY_INTEGER      2
  #define DY_FLOAT        3
  #define DY_DISTANCE     4
  #define DY_VINT         5
  #define DY_VFLOAT       6
  #define DY_VDIST        7
  #define DY_BOOLEAN      8
  #define DY_CHOICE       9
  #define DY_SURFACE      10
  #define DY_FONT         11
  #define DY_TEXT         12
  #define DY_LAYERS       13
  #define DY_CUSTOM       14
  #define DY__LAST        DY_CUSTOM
  . . .
```

A dynamic value datatype is a structure whose first field is a DynaType code for the type of the value, followed by varient fields which hold the value in a form appropriate to the given type. The different varient forms of value encoding are listed here.

DY_STRING and DY_SURFACE type values contain a pointer to a string

buffer and a buffer size. If the buffer size is zero, the buffer is read-only.

(7) Modeling Types

```
typedef struct st_DyValString {
  DynaType         type;
  char            *buf;
  int              bufLen;
} DyValString;
. . .
```

Integer values are used for types DY_INTEGER, DY_BOOLEAN (zero or non-zero), DY_CHOICE (0 - n-1), DY_FONT (font number 0 to n-1) and DY_LAYERS (bit mask for layer set). The default value is only used in requesters as the reset value.

(8) Modeling Types

```
. . .
typedef struct st_DyValInt {
  DynaType         type;
  int              value;
  int              defVal;
} DyValInt;
. . .
```

Floating point values are used for types DY_FLOAT and DY_DISTANCE (distance measure in meters). The default value is again used when resetting a requester.

(9) Modeling Types

```
. . .
typedef struct st_DyValFloat {
  DynaType         type;
  double           value;
  double           defVal;
} DyValFloat;
. . .
```

The DY_VINT type is an integer vector with three components. The single default value resets all three components of the vector when used in a requester.

(10) Modeling Types

```
. . .
typedef struct st_DyValIVector {
  DynaType         type;
  int              val[3];
  int              defVal;
} DyValIVector;
. . .
```

Floating point three-component vectors are used for the types DY_VFLOAT and DY_VDIST, with the latter being distances encoded in meters.

(11) Modeling Types

```
  . . .
  typedef struct st_DyValFVector {
    DynaType         type;
    double           val[3];
    double           defVal;
  } DyValFVector;
  . . .
```

The custom dynamic type, DY_CUSTOM, is used to encode values which do not fit one of the standard types. The meaning of the fields following the type for a custom value are defined by agreement between the sender and receiver and are usually a set of 4-byte numbers and pointers, although they can be anything. Usually anywhere a custom value is required, an alternate string form can also be accepted.

```
    (12)  Modeling Types
  . . .
  typedef struct st_DyValCustom {
    DynaType         type;
    int              val[4];
  } DyValCustom;
  . . .
```

A DynaValue type is the union of all possible value type varients plus the type code itself which is the only field set for DY_NONE and DY_TEXT types.

```
    (13)  Modeling Types
  . . .
  typedef union un_DynaValue {
    DynaType         type;
    DyValString      str;
    DyValInt         intv;
    DyValFloat       flt;
    DyValIVector     ivec;
    DyValFVector     fvec;
    DyValCustom      cust;
  } DynaValue;
  . . .
```

Error codes returned from the dynamic data type functions.

```
    (14)  Modeling Types
  . . .
  #define DYERR_NONE               0
  #define DYERR_MEMORY            (-1)
  #define DYERR_BADTYPE           (-2)
  #define DYERR_BADSEQ            (-3)
  #define DYERR_BADCTRLID         (-4)
  #define DYERR_TOOMANYCTRL       (-5)
  #define DYERR_INTERNAL          (-6)
  . . .
```

1.16 s4.2

At any given moment Modeler holds some set of layers, each containing a potentially large collection of point and polygon elements. The user selects which subset of elements are to be affected by an operation by picking layers as active and inactive, and selecting elements in those layers with the element selection tools. Plug-in operations can select what elements to operate on as a function of the user's selections.

4.2.1 Layer Filters
4.2.2 Element Filters

1.17 s4.2.1

EltOpLayer codes are used to select which layers will be affected by an operation.

PRIMARY The primary layer is the single active layer that is affected by mesh edits.

FG The foreground layers are those which are active and displayed.

BG The background layers are those which are inactive but still displayed.

SELECT Select layers are all displayed layers, foreground and background.

ALL All layers are all layers in the modeler system whether they contain data or not.

EMPTY Empty layers are those with no data elements in them.

NONEMPTY Non-empty layers are any layers which contain some data.

Individual Layers
In addition to the defined values, codes from 101 to 110 can be used to select the individual layers 1 through 10.

(15) Modeling Base Types

```
. . .
typedef int              EltOpLayer;
#define OPLYR_PRIMARY    0
#define OPLYR_FG         1
#define OPLYR_BG         2
#define OPLYR_SELECT     3
#define OPLYR_ALL        4
#define OPLYR_EMPTY      5
#define OPLYR_NONEMPTY   6
. . .
```

1.18 s4.2.2

```
EltOpSelect is a selection mode to pick elements from the selected
layers for operations.

 GLOBAL      All elements, whether selected or unselected, will be
       affected by the operation.

 USER        Only those elements selected by the user will be
       affected.  This includes the implicit selection of all
       elements when nothing is explicitly selected, and
       selections by volume.

 DIRECT      Elements selected directly with the point or polygon
       selection tools will be affected.  This is the case for
       both points and polygons regardless of the current select
       mode.

    (16)  Modeling Base Types
  . . .
  typedef int              EltOpSelect;
  #define OPSEL_GLOBAL     0
  #define OPSEL_USER       1
  #define OPSEL_DIRECT     2
  . . .
```

1.19 s5

```
There are two types of servers defined for Modeler which can perform
modeling operations.  Mesh Edit servers can perform a single mesh data
editing operation by affecting the data elements at a fairly low
level.  Command Sequence servers can execute a sequence of editing
operations, including most of those accessable to the user as well as
low-level mesh edits.

      5.1  Mesh Editing
      5.2  Command Sequencing
```

1.20 s5.1

```
The "MeshDataEdit" class provides the capability of editing existing
layer data though low-level point and polygon operations.  The
available MeshDataEdit servers in a Modeler session are presented to
the user in the "Custom" popup in the "Tools" menu.  Editing is done
through functions which operate on elements represented by opaque
pointers.  The editing state itself is also maintained as an opaque
pointer and is the first argument to most of the calls.

    (17)  Modeling Base Types
  . . .
  typedef struct st_Vertex        *PntID;
  typedef struct st_Polygon       *PolID;
```

```
  typedef struct st_EditState     *EditStateRef;
  . . .
```

A mesh edit operation is a single undoable modification to layer data. The server starts the operation and is given an EditStateRef pointer to refer to the ongoing state of the edit operation. The server may then add new elements and modify or delete existing elements. As the server requests changes, those are logged by the host but will not be applied until the operation is complete. At any time the server may abort the operation and the pending changes will be discarded, or it can accept the changes and they will be applied as the last step before the server exits.

5.1.1 Local Data
5.1.2 Getting Element Information
5.1.3 PointInfo
5.1.4 PolygonInfo
5.1.5 Main Data Struct
5.1.6 Error Codes
5.1.7 Query Functions
5.1.8 Element Traversal
5.1.9 Creating New Elements
5.1.10 Modifying Existing Elements

1.21 s5.1.1

Upon activation, a mesh edit server gets a 'MeshEditBegin' function pointer as its local data. To initiate the mesh editing operation, the server calls this function and gets back a 'MeshEditOp' which contains the data for the edit as well as pointers to all the editing functions. This can be called only once for each activation.

```
    (18)  Modeling Servers

  typedef MeshEditOp *    MeshEditBegin (int pntBuf, int polBuf,
            EltOpSelect);
  . . .
```

The first two arguments to the function are the client data sizes (in bytes) for points and polygons, respectively. If non-zero, the host will allocate a block of memory for each and every point and polygon for the exclusive use of this edit operation. These client data buffers can be used to associate any information with specific points and polygons for the course of the edit operation, and will be freed when the operation completes. The third argument is the selection option and determines what elements are initially selected.

1.22 s5.1.2

Servers can get information about existing elements by reading them out into special information structures. The PointInfo and PolygonInfo structures are used to hold information about points and

polygons, respectively. Every element has an ID, a userData pointer, a layer number and flags.

PntID or PolID
This uniquely identifies each element and is the reference used for manipulating them.

userData This is a pointer to a memory block which has been allocated for the client specifically for this element according to the requested size in the call to MeshEditBegin. This is an area where the client can store computed values for points and polygon while it operates on them.

layer This is just the number of the layer where the element is located (0-9 currently).

flags All elements have flags bits for selection and deletion. The PPDF_SELECT bit is set if the element matched the selection criterion from the start of the edit, and the PPDF_DELETE bit is set if the element has been deleted in this session.

```
   (19)  Modeling Types
  . . .
  #define PPDF_SELECT     (1<<0)
  #define PPDF_DELETE     (1<<1)
  . . .
```

Except for the memory pointed to by the userData pointer, the contents of info structures or the data they reference are read-only and cannot be modified. Any attempts to do so will either be futile or catastrophic.

1.23 s5.1.3

In addition to the common parts, a PointInfo struct also includes the point position as a triple of floating point numbers for the X, Y and Z coordinates.

```
   (20)  Modeling Base Types
  . . .
  typedef struct st_PointInfo {
    PntID            pnt;
    void            *userData;
    int              layer;
    int              flags;
    double           position[3];
  } PointInfo;
  . . .
```

1.24 s5.1.4

In addition to the common parts of the info structure, a PolygonInfo struct encodes the polygon shape as the number of points and an array of their IDs. The surface assigned to the polygon is given by a name string.

```
    (21)  Modeling Base Types
  . . .
  typedef struct st_PolygonInfo {
    PolID            pol;
    void            *userData;
    int              layer;
    int              flags;
    int              numPnts;
    const PntID     *points;
    const char      *surface;
  } PolygonInfo;
  . . .
```

Polygons also have some additional flag bits. CCEND and CCSTART are set if the polygon has continuity points at either end. CURVE is set if this is a curve (it is a face if this is clear). DETAIL is set if the polygon is a detail.

```
    (22)  Modeling Types
  . . .
  #define PPDF_CCEND      (1<<2)
  #define PPDF_CCSTART    (1<<3)
  #define PPDF_CURVE      (1<<4)
  #define PPDF_DETAIL     (1<<5)
  . . .
```

1.25 s5.1.5

When the 'MeshEditBegin' function starts an edit operation, it returns a MeshEditOp structure which the client uses to execute the edit. This structure contains a few data fields and a large set of function fields.

```
    (23)  Modeling Types
  . . .
  typedef struct st_MeshEditOp {
    EditStateRef      state;
    int               layerNum;
    void             (*done) (EditStateRef, EDError, int selm);

    <Mesh Edit Count functions>
    <Mesh Edit Enumeration functions>
    <Mesh Edit Query functions>
    <Mesh Edit Create functions>
    <Mesh Edit Modify functions>
  } MeshEditOp;
  . . .

 state       The internal state of the edit is maintained in the
```

private 'state' field which is the first argument to every function.

layerNum Points and polygons may only be modified if they belong to the primary active layer which is given by this layer number. The primary layer is the lowest numbered foreground layer. All new data will be added to this layer and changes attempted on data in other layers will fail.

done The 'done' function completes the edit. If the error code is EDERR_NONE, the edit operation will complete and the cumulative edits will be applied to the data. If an actual error code is passed, the edit will abort and any changes made will be discarded. The 'selm' argument is bit flags which provide info on how to alter the selection based on editing changes. A value of zero leaves all directly selected elements selected after the edit. The CLEARCURRENT hint bit set will clear the current selected elements, and the SELECTNEW hint bit set will cause any newly created elements to become selected. Hints will not override selection settings made by the user, and only when elements are explicitly selected will new selections be made. The force bits will always force direct selection of the points and/or polygons created by this operation regardless of current user selections.

(24) Modeling Types

```
. . .
#define EDSELM_CLEARCURRENT     (1<<0)
#define EDSELM_SELECTNEW        (1<<1)
#ifndef LW_PRERELEASE
 #define EDSELM_FORCEVRTS       (1<<2)
 #define EDSELM_FORCEPOLS       (1<<3)
#endif
. . .
```

Other functions

The remaining functions allow for examining the state of the mesh data and modifying it. All changes for a given edit operation must be made through these functions. No data structures may be modified directly.

As changes are made they are buffered through the undo mechanism, so they are not reflected in the data until the operation is complete. For example, if a MeshDataEdit client reads the coordinates of a point and changes them (correctly using the 'pntMove' function) and reads the coordinates again, they will be the same as the first time. The coordinates will not change until the edits are sucessfully applied using the 'done' function.

1.26 s5.1.6

Errors are integer codes returned from functions and passed to the 'done' function. The exceptions are functions which create new

elements in which case an error is signaled by a null return value. The BADLAYER error will be returned for an attempt to operate on data not in the primary edit layer. BADSURF will be returned for an illegal surface name. BADARGS is the catch-all for other invalid arguments.

(25) Modeling Base Types

```
. . .
typedef int             EDError;
#define EDERR_NONE      0
#define EDERR_NOMEMORY  1
#define EDERR_BADLAYER  2
#define EDERR_BADSURF   3
#define EDERR_USERABORT 4
#define EDERR_BADARGS   5
. . .
```

1.27 s5.1.7

Clients can get a count of the number of points or polygons in specific layers. The 'mode' argument to the count functions specify all the elements, only the selected elements or only the elements deleted in this edit session.

(26) Mesh Edit Count functions

```
int             (*pointCount) (EditStateRef, EltOpLayer, int mode);
int             (*polyCount)  (EditStateRef, EltOpLayer, int mode);
```

(27) Modeling Types

```
. . .
#define EDCOUNT_ALL             0
#define EDCOUNT_SELECT          1
#define EDCOUNT_DELETE          2
. . .
```

Given a point or polygon ID, the client can get info for that element. The returned info pointer is only valid until the next call to an info function (including enumeration). The normal vector for a polygon may also be found given its ID. The 'polyNormal' function returns zero if the polygon has fewer than 3 vertices, or the normal is degenerate for some reason. If it returns 1, then the normal has been written to the caller's vector.

(28) Mesh Edit Query functions

```
PointInfo *     (*pointInfo)  (EditStateRef, PntID);
PolygonInfo *   (*polyInfo)   (EditStateRef, PolID);
int             (*polyNormal) (EditStateRef, PolID, double[3]);
```

There is only one of each of the PointInfo and PolygonInfo structs for every usage. The same pointer is returned from each query call and passed to the enumeration functions, so the client must copy any information needed before calling the query function again.

1.28 s5.1.8

The client can traverse all the elements in a layer or combination of layers by passing a callback to be called for each element. These enumeration functions (given by the prototypes below) take as arguments a client data pointer which can be arbitrary, and the info structure for the current element. If the client returns an error code (or any non-zero value for that matter) from this function, the scan will be aborted and that code will be returned.

```
    (29)  Modeling Base Types
  . . .
  typedef EDError         PointScanFunc (void *, const PointInfo *);
  typedef EDError         PolyScanFunc (void *, const PolygonInfo *);
  . . .
```

The following functions initiate a scan of points or polygons in layer data. The client provides an enumeration callback and client data pointer as well as specifying which layers to include in the scan. The function will be called for each point and polygon in order. If the selection mode used to begin this edit was DIRECT, the order of the selected elements is the same as the order that the user selected them. In other select modes, the order is the creation order for points and undefined for polygons. The return value is EDERR_NONE (0) if the scan completed, and the non-zero error code returned by the enumeration callback if the scan was aborted.

```
    (30)  Mesh Edit Enumeration functions

  EDError         (*pointScan) (EditStateRef, PointScanFunc *,
              void *, EltOpLayer);
  EDError         (*polyScan)  (EditStateRef, PolyScanFunc *,
              void *, EltOpLayer);
```

1.29 s5.1.9

A new data element is added by calling the appropriate function, which creates the new element but does not add it to the layer until the edit is completed. Polygons are created from lists of PntIDs which can be the IDs of pre-existing points or of points created in this session, as long as the existing ones are in the primary layer.

addPoint New points are created by passing a vector of X, Y and Z coordinates to this function.

addPoly Polygons are created from a surface name (or null for default), number of points and point list. The first, second and last points are used to compute the polygon normal.

addCurve Curves are created the same way as polygons except that they have an additional flag value which may have PPDF_CCSTART and/or END set. Closed curves must have both of these bits set and have the first and last two

```
      points overlapping.

addQuad and addTri
      These two functions create quadrangles and triangles
      using the default surface and obeying the user's new data
      options with respect to two-sided and triangles only.
      These are used by operations which create new objects
      from scratch, like the sphere or box tools in Modeler.

addPatch    This will add a set of polygons to create a polygonal
      patch from bounding curves obeying the user's new data
      options.  It takes the number of divisions in the C and R
      directions, the length/knot flags in the C and R
      directions, and three or four boundary curve
      descriptions.  Each boundary curve is a curve-type
      polygon and the indices of the start and end knots of the
      curve to be used for patching.

   (31)  Modeling Base Types
 . . .
 typedef struct st_PBoundCv {
   PolID            curve;
   int              start, end;
 } PBoundCv;

   (32)  Mesh Edit Create functions

 PntID           (*addPoint) (EditStateRef, double *xyz);
 PolID           (*addPoly)  (EditStateRef, const char *surf,
            int numPnt, const PntID *);
 PolID           (*addCurve) (EditStateRef, const char *surf,
            int numPnt, const PntID *, int flags);
 EDError         (*addQuad)  (EditStateRef, PntID, PntID,
            PntID, PntID);
 EDError         (*addTri)   (EditStateRef, PntID, PntID, PntID);
 EDError         (*addPatch) (EditStateRef, int nr, int nc, int lr,
            int lc, PBoundCv *r0, PBoundCv *r1,
            PBoundCv *c0, PBoundCv *c1);
```

1.30 s5.1.10

```
These functions are used to alter existing data.  If called with
elements created in this edit session they will return BADLAYER.

 remPoint, remPoly
      Remove existing data elements.  These will remove points
      and polygons from the current data set.  The PPDF_DELETE
      flag bit will be set for these elements after this
      function is called.

 pntMove     Move a point.  The point will be moved to the new
      coordinates.

 polSurf     Change polygon surface.  The polygon will be altered to
      use the new named surface.
```

```
 polPnts     Change point list.  The polygon will be changed to have a
       new set of points given by the list of IDs.  The PntIDs
       may be for existing points or points created this
       session, but should not refer to points that will be
       deleted.

 polFlags    Change polygon attributes.  The first mask is the set of
       attributes to change and the second is their new values.
       Only PPDF_CCEND and PPDF_CCSTART may currently be
       modified.

    (33)  Mesh Edit Modify functions

  EDError         (*remPoint) (EditStateRef, PntID);
  EDError         (*remPoly)  (EditStateRef, PolID);
  EDError         (*pntMove) (EditStateRef, PntID, const double *);
  EDError         (*polSurf) (EditStateRef, PolID, const char *);
  EDError         (*polPnts) (EditStateRef, PolID, int, const PntID *);
  EDError         (*polFlag) (EditStateRef, PolID, int mask, int value);
```

1.31 s5.2

The "CommandSequence" class servers can execute a sequence of Modeler commands and/or mesh edits. CommandSequence servers are presented to the user in the "Custom" popup in the "Objects" menu, and the user has the ability to configure the server to take different string arguments. The argument string selected by the user is pass to the server at activation.

Modeling commands are identified by unique case-insensitive names and by unique integer codes. Codes may be looked up given the command string.

```
    (34)  Modeling Types
  . . .
  typedef int             CommandCode;
  . . .
```

Commands are executed by passing the command code and a list of arguments in the form of DynaValues. The values can have any type which is can be converted to the required type of each positional argument. A command sequence server can execute any sequence of commands and may combine them with mesh edit operations as well.

```
      5.2.1  Command Activation
      5.2.2  Commands
      5.2.3  External Activation on Windows
```

1.32 s5.2.1

A CommandSequence server gets a ModCommand structure passed to its activation function. The activation function performs the sequence of commands and mesh edits and returns when complete.

data Internal host data passed as the first argument to the 'lookup' and 'execute' functions.

argument String argument to this command as set in the custom command list.

lookup Function which converts a command name to a command code for use with the 'execute' function. This is a separate step so that the string lookup does not have to be done on every command invocation. Case is not significant. Since the codes are fixed for a session, they can be looked up the first time the server is used and cached after that.

execute Function which performs the modeling function. Takes a command code as found by 'lookup' and an array of DynaValue arguments. Which elements to be affected by the command can be selected using the EltOpSelect mode. If non-null, the result pointer will be written with the return value of the command. Commands with no result will write DY_NULL on this value. The return value is zero for success and an error code for failure. Some possible codes are: 1 = out of memory, 2 = I/O error, 4 = user abort, 2901 = wrong number of arguments, 2902 = wrong argument type, 2903 = bad argument value.

editBegin This function can be used as described in the secion on mesh editing to start a mesh edit operation from a command sequence server. Any edit operation must be complete before more commands are executed.

```
   (35)  Modeling Servers
 . . .
 typedef struct st_ModCommand {
   void             *data;
   const char       *argument;
   CommandCode     (*lookup)  (void *, const char *cmdName);
   int             (*execute) (void *, CommandCode cmd,
             int argc, const DynaValue *argv,
             EltOpSelect, DynaValue *result);
   MeshEditBegin    *editBegin;
 } ModCommand;
```

1.33 s5.2.2

Here follows a complete list of the commands supported by the command mode interface and their arguments. Optional arguments are listed in square brackets. A more complete description of each command may be found in the Modeler ARexx documentation.

```
NEW, UNDO

DELETE, CUT, COPY, PASTE


LOAD, SAVE  filename<string>

SETLAYER, SETBLAYER
      mask<layers>

SURFACE     name<string>

FIXEDFLEX   axis<X|Y|Z>, start<dist>, end<dist>, [ease<i;o>]

AUTOFLEX    axis<X|Y|Z>, direction<+|->, [ease<i;o>]

DEFORMREGION
      radius<vector>, [center<vector>, axis<X|Y|Z>]

MOVE, SHEAR, MAGNET
      offset<vector>

ROTATE, TWIST, VORTEX
      angle<float>, axis<X|Y|Z>, [center<vector>]

SCALE, TAPER, POLE
      factor<vector>, [center<vector>]

BEND        angle<float>, direction<float>, [center<vector>]

JITTER      radius<vector>, [type<GAUSSIAN|UNIFORM|NORMAL|RADIAL>,
      center<vector>]

SMOOTH      [iterations<int>, strength<float>]

QUANTIZE    size<vector>

MERGEPOINTS [mindist<dist>]

MAKEBOX     lowcorner<vector>, highcorner<vector>,
      [nsegments<vector>]

MAKEBALL    radius<vector>, nsides<int>, nsegments<int>,
      [center<vector>]

MAKETESBALL radius<vector>, level<int>, [center<vector>]

MAKEDISC, MAKECONE
      radius<vector>, top<dist>, bottom<dist>, axis<X|Y|Z>,
      nsides<int>, [nsegments<int>, center<vector>]

MAKETEXT    text<string>, index<number>, [cornertype<SHARP|BUFFERED>,
      spacing<number>, scale<number>, axis<X|Y|Z>, pos<vector>]

LATHE       axis<X|Y|Z>, nsides<int>, [center<vector>,
      endangle<float>, startangle<float>, offset<dist>]
```

```
EXTRUDE     axis<X|Y|Z>, extent<dist>, [nsegments<int>]

MIRROR      axis<X|Y|Z>, plane<dist>

PATHCLONE, PATHEXTRUDE
      filename<string>, [step<float>, start<float>, end<float>]

RAILCLONE, RAILEXTRUDE
      segments<int>, [divs<KNOTS|LENGTHS>, flags<o;s>,
      strength<float>]

AXISDRILL   operation<CORE|TUNNEL|SLICE|STENCIL>, axis<X|Y|Z>,
      [surface<string>]

SOLIDDRILL  operation<CORE|TUNNEL|SLICE|STENCIL>, [surface<string>]

BOOLEAN     operation<UNION|SUBTRACT|INTERSECT|ADD>

BEVEL       inset<dist>, shift<dist>

SHAPEBEVEL  pattern<custom>

            The patten for a shapebevel is either a string containing
      pairs of inset / shift values, or a custom dynavalue with
      the val[0] field set to the number of pairs, and the
      val[1] field cast to a pointer to an array of doubles
      holding the pairs.

SMOOTHSHIFT offset<dist>, [maxangle<float>]

FLIP, TRIPLE, FREEZECURVES


ALIGNPOLS, REMOVEPOLS, UNIFYPOLS


CHANGESURFACE
      surface<string>

SUBDIVIDE   mode<FLAT|SMOOTH|METAFORM>, [maxangle<float>]

FRACSUBDIVIDE
      mode<FLAT|SMOOTH|METAFORM>, fractal<float>,
      [maxangle<float>])

SEL_POINT   action<SET|CLEAR>

            action, VOLUME, lo<vector>, hi<vector>

            action, CONNECT

            action, NPEQ, npol<int>

            action, NPLT, npol<int>

            action, NPGT, npol<int>
```

```
SEL_POLYGON action<SET|CLEAR>

            action, VOLEXCL, lo<vector>, hi<vector>

            action, VOLINCL, lo<vector>, hi<vector>

            action, CONNECT

            action, NVEQ, nvert<int>

            action, NVLT, nvert<int>

            action, NVGT, nvert<int>

            action, SURFACE, surface<string>

            action, FACE

            action, CURVE

            action, NONPLANAR, [limit<float>]

SEL_INVERT

SEL_HIDE    state<SELECTED|UNSELECTED>

SEL_UNHIDE

CMDSEQ      name<string>, [arg<string>]

PLUGIN      module<string>, [class<string>, name<string>,
      username<string>]
```

1.34 s5.2.3

When Modeler is running under Windows, CommandSequence class servers in the program can be triggered by other Windows programs. The Modeler main window looks for messages with a code created by the function RegisterWindowMessage() with the string "LWM CmdSeq Trigger". This message code is unique throughout the Windows session and the arguments of this message describe the server to activate. The first argument (wp) should be null, and the second argument (lp) should be two global atoms containing the CommandSequence server name and argument string, combined with the MAKELONG() macro.

The following Windows function triggers a server in Modeler given the handle to Modeler's main window. Atoms are created to pass the server name and argument (if any) and the message is posted to Modeler's window. If the PostMessage fails, this function frees the atoms, otherwise Modeler will free them when it processes the message. The message code could be looked up only one time if multiple messages are to be sent, and SendMessage could be used for synchronous triggering.

(36) Trigger code

```
  static void
TriggerModeler (
  HWND                      wnd,
  const char               *server,
  const char               *argument)
{
  UINT                      msg;
  ATOM                      name, arg;

  msg = RegisterWindowMessage ("LWM CmdSeq Trigger");

  name = GlobalAddAtom (server);
  if (argument && argument[0])
    arg = GlobalAddAtom (argument);
  else
    arg = 0;

  if (!PostMessage (wnd, msg, NULL, MAKELONG (name, arg))) {
    GlobalDeleteAtom (name);
    if (arg)
      GlobalDeleteAtom (arg);
  }
}
```

1.35 s6

This section contains descriptions of the global data pointers which can be accessed from Modeler's global function.

6.1 Dynamic Conversion
6.2 Dynamic Requester
6.3 Dynamic Monitor
6.4 Custom Commands
6.5 Modeler Internal State
6.6 Surfaces List
6.7 Outline Font List

1.36 s6.1

The global ID "LWM: Dynamic Conversion" returns a DynaConvertFunc which can be used to translate a dynamic type element to another type. An error may be returned if the conversion cannot be performed, and hints may be provided when converting strings to integer bitfield or choice values.

```
    (37)  Modeling Globals

  typedef int             DynaConvertFunc (const DynaValue *,
            DynaValue *,
            const DynaStringHint *);
  . . .
```

String hints are choice hints and/or bitfield hints. The choice hint is list of strings and values used when converting between DY_STRING and DY_CHOICE types. The pairs indicate a mapping between choice values and strings. The list is terminated with a null item string. The bitfield hint is a list of character codes and bit values used when converting between DY_STRING and DY_INTEGER types. If the character (upper or lower case) is present in the string, the bit value will be ORed into the result, and visa-versa. The list is terminated with a zero bitval.

```
    (38)  Modeling Types
  . . .
  typedef struct st_DyChoiceHint {
    const char      *item;
    int              value;
  } DyChoiceHint;

  typedef struct st_DyBitfieldHint {
    char             code;
    int              bitval;
  } DyBitfieldHint;
  . . .
```

Either field in the string hint structure may be null.

```
    (39)  Modeling Types
  . . .
  typedef struct st_DynaStringHint {
    DyChoiceHint    *chc;
    DyBitfieldHint  *bits;
  } DynaStringHint;
  . . .
```

1.37 s6.2

The global ID "LWM: Dynamic Request" returns a set of functions for creating and displaying a simple requester. The requesters that can be created with this interface are like simple forms. There is a title and a series of lines each with a label and a control for a single value. The controls are described by DynaValues, with the DynaType determining the type of control and the value determining its setting. The user can change the value of the controls while the requester is displayed.

```
    (40)  Modeling Globals
  . . .
  typedef struct st_DynaReqFuncs {
    DynaRequestID   (*create)   (const char *);
    int             (*addCtrl)  (DynaRequestID, const char *,
              DyReqControlDesc *);
    DynaType        (*ctrlType) (DynaRequestID, int);
    int             (*valueSet) (DynaRequestID, int, DynaValue *);
    int             (*valueGet) (DynaRequestID, int, DynaValue *);
    int             (*post)     (DynaRequestID);
    void            (*destroy)  (DynaRequestID);
```

```
  } DynaReqFuncs;
  . . .
```

```
      6.2.1  Requester Usage
      6.2.2  Control Descriptors
```

1.38 s6.2.1

The basic idea is to create a requester, set up its controls, set their values, post the requester, read out the modified values and destroy the requester. The set/post/get cycle may be done any number of times once the requester is created.

create Create is used to allocate an instance of a dynamic requester with a given title string. Multiple requesters can be allocated at the same time, although only one may be displayed at once. The create function returns an ID which is used throughout the rest of the interface.

```
    (41)  Modeling Types
  . . .
  typedef struct st_DynaRequest   *DynaRequestID;
  . . .
```

addCtrl Controls are added to the requester starting from the top. The function takes a label and a description (below) and returns an index number for the new control which is used to set and get its value. The description contains the control type and other information necessary for its display.

ctrlType This function just returns the type of a control given its index.

valueSet Calling this function with a control index and a DynaValue sets the given control to that value. The type of the value does not have to be the same as the control as long as the one can be converted to the other. Note that any "default" fields in the DynaValue will be used to reset the control if the user selects "Reset."

valueGet This function is the opposite of the above, reading out the value of the control into the provided DynaValue.

post Once the requester is created and the proper values are set, this function displays the requester and allows the user to change the values. It returns zero if the user presses "Cancel" to dismiss the requester and one if the user presses "Ok."

destroy When all interaction is done, a call to destroy frees all resources and completes the process.

1.39 s6.2.2

Controls in a dynamic requester are determined primarily by a DynaType for the type of data being edited, however for some types additional settings may be required for correct display of the value.

Controls of type DY_STRING have a width, in characters, of the input field. This is an average width on systems with variable pitch fonts.

```
    (42)  Modeling Types
  . . .
  typedef struct st_DyReqStringDesc {
    DynaType         type;
    int              width;
  } DyReqStringDesc;
  . . .
```

Controls of type DY_CHOICE present a set of labeled buttons for the user to select between. The descriptor contains a pointer to an array of strings (terminated with a null pointer) for the labels of the choice items. If the vertical flag is true, the choices will be set in a vertical layout, otherwise they will be horizontal.

```
    (43)  Modeling Types
  . . .
  typedef struct st_DyReqChoiceDesc {
    DynaType         type;
    const char     **items;
    int              vertical;
  } DyReqChoiceDesc;
  . . .
```

The DY_TEXT control type is a constant control for displaying lines of text. The text lines are contained in an array of strings (terminated with a null pointer).

```
    (44)  Modeling Types
  . . .
  typedef struct st_DyReqTextDesc {
    DynaType         type;
    const char     **text;
  } DyReqTextDesc;
  . . .
```

The control descriptor is the union of all these varient records plus the DynaType alone. If there is no special descriptive data for a type, then only the type code is needed to create a control of that type.

```
    (45)  Modeling Types
  . . .
  typedef union un_DyReqControlDesc {
    DynaType         type;
    DyReqStringDesc  string;
    DyReqChoiceDesc  choice;
    DyReqTextDesc    text;
```

```
  } DyReqControlDesc;
```

1.40 s6.3

The global ID "LWM: Dynamic Monitor" returns a structure holding functions which can be used to create a monitor for providing feedback on the progress of an operation and allow user to abort it. Monitors are described in the "LightWave Plug-in Architecture" document and are declared in the ‘moni.h’ header file.

create Clients create a monitor instance with header and optional footer text. Once created, the monitor object can be used as described in the above referenced document. The create func may return null if the Modeler bar graph is already in use, since there can be only one.

destroy When done, the monitor must be destroyed by calling this function. This must be called whether the operation was completed or not.

```
   (46)  Modeling Globals
 . . .
 typedef struct st_DynaMonitorFuncs {
   Monitor *       (*create) (const char *, const char *);
   void            (*destroy) (Monitor *);
 } DynaMonitorFuncs;
 . . .
```

1.41 s6.4

The global ID "LWM: Custom Commands" returns a set of functions for manipulating the custom commands and function key mappings. These may be changed by a server, but there should be some provision for setting them back to the user’s defaults when complete.

listAdd Adds an new custom command to the list. The name will be the string that the user sees in the custom popup, the server is the internal server name of the CommandSequence plug-in to activate, and the arg string is the agrument that will be passed to that plug-in. This returns false if the new item could not be added.

listRem Removes a entry from the custom list given its name.

funGet Gets the server and argument strings for the current association with function key N. The bufLen is the length of the string buffers. This returns false if the key is not assigned.

funSet Sets the server to activate and the argument to pass when the given function key is pressed.

```
    (47)  Modeling Globals
  . . .
  typedef struct st_CustomCommandFuncs {
    int             (*listAdd) (const char *name,
              const char *server,
              const char *arg);
    void            (*listRem) (const char *name);
    int             (*funGet) (int n, char *server,
             char *arg, int bufLen);
    void            (*funSet) (int n, const char *server,
             const char *arg);
  } CustomCommandFuncs;
  . . .
```

For ARexx scripts on the Amiga Modeler, the server name is "$REXX" and the argument is the script name.

1.42 s6.5

The global ID "LWM: State Query" returns a set of functions for querying Modeler's global state. It can be queried at any time although it may only be altered at specific times.

numLayers This returns total number of data layers in Modeler.

layerMask This returns bits describing the set of layers included in each of the possible EltOpLayer selections. If the set includes layer 1 then bit 0 is set, if it includes layer 2 bit 1 is set, and so forth.

surface This returns the name of the default surface.

bbox This returns the number of points in the given layer, and if minmax is a non-null pointer, it is treated as an array of 6 doubles and is filled with the bounding box information for the layer (x0, x1, y0, y1, z0, z1).

```
    (48)  Modeling Globals
  . . .
  typedef struct st_StateQueryFuncs {
    int             (*numLayers) (void);
    unsigned int    (*layerMask) (EltOpLayer);
    const char *    (*surface) (void);
    unsigned int    (*bbox) (EltOpLayer, double *minmax);
  } StateQueryFuncs;
  . . .
```

1.43 s6.6

The global ID "LWM: Surface List" returns a set of functions which can be used to read and modify Modeler's surface list. Clients may add, rename and modify the contents of surfaces at any time, but there is

no capability to remove them. Note that adding surfaces or renaming them will alter the relative order of surfaces in the list.

next This returns the name of the surface after the given one. If the argument is null, it returns the first surface. If the argument is the last surface in the list, it returns null.

create This creates a surface of the given name with no data.

rename This changes the name of a surface, reordering it in the list.

getData This returns the size and contents of the surface data parameters for the named surface.

setData This resets the data parameter block to the given size and contents.

(49) Modeling Globals

```
. . .
typedef struct st_SurfaceListFuncs {
  const char *    (*next)    (const char *name);
  void            (*create)  (const char *name);
  void            (*rename)  (const char *name,
           const char *newName);
  void *          (*getData) (const char *name, int *size);
  void            (*setData) (const char *name, int size,
           void *data);
} SurfaceListFuncs;
. . .
```

1.44 s6.7

The global ID "LWM: Font List" returns a set of functions for reading and modifying Modeler's font list. The list may be modified at any time, but keep in mind that altering the list may affect stored font choices in your requesters, if any.

count This returns the total number of fonts in the list.

index This returns the list index for a named font, -1 if not found.

name This returns the name of a font given its list index, null if out of range.

load This loads the given file as a Type-1 font and returns the new list index. The fonts at this old index and above are all shifted up. It returns -1 for errors.

clear This removes the font at the given index from the list, shifting all the others down.

(50) Modeling Globals

```
  . . .
  typedef struct st_FontListFuncs {
    int             (*count) (void);
    int             (*index) (const char *name);
    const char *    (*name)  (int index);
    int             (*load)  (const char *filename);
    void            (*clear) (int index);
  } FontListFuncs;
```

1.45 s7

The servers and globals for Layout share a set of type and value definitions which are basic to understanding concepts dehind the Layout servers and globals.

7.1 Coordinate and Range Scales
7.2 Items and Properties
7.3 Time
7.4 Errors
7.5 Instances and Handlers

1.46 s7.1

There are several conventions used to interpret different types of values within LightWave and throughout this external interface.

positions Positions are always given as an array of three doubles which are the X, Y and Z coordinates (respectively) of a position with respect to some known coordinate system. This system is always listed along with the vector (usually object or world).

directions Direction vectors, such as normals, are also an X Y Z array of doubles, but are normalized to be unit vectors. The coordinate system for these is also always listed.

rotations Rotations are arrays of three doubles representing the Euler angles (heading, pitch and bank, respectively) of an item in some coordinate system. Angles are represented in radians.

colors Colors are given as an array of three doubles giving the intensities of the red, green and blue componenets of the color, respectively. The values are scaled so that 0.0 is the minimum intensity and 1.0 is the maximum, although out-of-range values are allowed.

percentages Values which are represented on the LightWave user interface as percentages are represented internally as doubles scaled from 0.0 to 1.0. A luminosity of 65.7%, for example, would be held internally as the floating point value 0.657.

Floating point values with a nominal range of 0.0 to 1.0 will sometimes be converted to single-byte values for storing in image buffers. Colors and alphas are converted this way for final image output and other values are used this way internally. The floating point value is clipped to be strictly within the 0.0 to 1.0 range and is then scaled and converted to a BufferValue type so that 0.0 is 0 and 1.0 is 255.

```
    (51)  Animation Types

  typedef unsigned char     BufferValue;
  . . .
```

1.47 s7.2

A LightWave item is anthing which can be keyframed in the layout interface. All objects, lights, bones and cameras in LightWave are items and have a unique LWItemID value.

```
    (52)  Animation Types
  . . .
  typedef void *           LWItemID;
  #define LWITEM_NULL      ((LWItemID) 0)
  . . .
```

Types of items are given by LWItemType codes.

```
    (53)  Animation Types
  . . .
  typedef int              LWItemType;
  #define LWI_OBJECT       0
  #define LWI_LIGHT        1
  #define LWI_CAMERA       2
  #define LWI_BONE         3
  . . .
```

All items have a set of vector parameters which servers can read (and sometimes write) using property codes.

```
 POSITION    item location in its parent's coordinates.

 ROTATION    item rotation in its parent's coordinates.

 SCALING     item X, Y and Z scaling factors relative to its parent.

 RIGHT, UP, FORWARD
       +X, +Y and +Z direction vectors, respectively, for the
       item in world coordinates.  These three concatenated
       together are the transformation matrix for the item.

 PIVOT       item pivot point in its own coordinates.

 W_POSITION  item location in world coordinates.
```

```
    (54)  Animation Types
  . . .
  typedef int              LWItemParam;
  #define LWIP_POSITION    1
  #define LWIP_RIGHT       2
  #define LWIP_UP          3
  #define LWIP_FORWARD     4
  #define LWIP_ROTATION    5
  #define LWIP_SCALING     6
  #define LWIP_PIVOT       7
  #define LWIP_W_POSITION  8
  . . .
```

1.48 s7.3

Time values in LightWave are given in two ways. A frame number is the index of a single image (typically the current image) in the output sequence of still images that make up the animation. A time value is the precise instant of an event in seconds. Assuming a scene rendered at 30 frames per second and without motion blur (or with a blur length of zero), frame N is a snapshot of the animation at a time in seconds of N/30. If there is motion blur, then some events in frame N will be from times slightly before N/30 seconds, and if the motion blur length is greater than 100%, then some events may even overlap with the times of events in previous frames.

```
    (55)  Animation Types
  . . .
  typedef int              LWFrame;
  typedef double           LWTime;
  . . .
```

1.49 s7.4

Server functions return errors to LightWave by returning a string pointer. A null string pointer indicates no error, and a non-null pointer points to an error string. The string will be displayed for the user and, except where otherwise indicated, the user will have the option to ignore the error and continue with the operation.

```
    (56)  Animation Types
  . . .
  typedef const char *     LWError;
  . . .
```

1.50 s7.5

Most LightWave plug-ins are "handlers" which manage "instances." An instance is a specific collection of user settings for a texture, image filter, etc., which persist across sessions by being stored in

scene and object files. A ‘LWInstance’ is any longword value which identifies a specific instance for a specific server, usually a pointer to allocated memory.

```
    (57)  Animation Types
  . . .
  typedef void *           LWInstance;
  . . .
```

7.5.1 Instance Persistence
7.5.2 Handler Functions
7.5.3 Interface Server

1.51 s7.5.1

Instances have to load and save their data to and from ASCII scene files and binary object files, and sometimes both, so the data read/write mechanism provides servers with functions to read and write data in both these formats. The ‘read’ function reads bytes from the source and returns the number of bytes read. The ‘write’ function writes bytes to the output and tracks any errors internally. The format of the file is given by ‘ioMode’ code, either OBJECT or SCENE.

```
    (58)  Animation Types
  . . .
  #define LWIO_OBJECT      0
  #define LWIO_SCENE       1
  . . .
```

If the mode is OBJECT, the format is binary and no scene-specific information should be stored. The read and write functions deal in raw bytes which can have any value from 0 to 255. They read or write the number of bytes requested using the passed buffer.

If the mode is SCENE, the format is ASCII and bytes stored must be in the extended ASCII range of 32 to 255. Values outside this range are ignored or undefined. The read and write functions in this case deal with lines. The write function writes a line at a time and looks for a null terminator in the input rather than the length. The read function can read partial lines if a length less then or equal to the total line length is requested. If the length is greater than the remaining line length, the length is returned and the buffer is null-terminated. The read function returns -1 for the actual end of input, since a read length of zero is valid for a blank line.

```
    (59)  Animation Types
  . . .
  typedef struct st_LWLoadState {
    int               ioMode;
    void             *readData;
    int              (*read) (void *readData, char *buf,
           int len);
  } LWLoadState;

  typedef struct st_LWSaveState {
```

```
    int               ioMode;
    void             *writeData;
    void            (*write) (void *writeData, char *buf,
            int len);
  } LWSaveState;
  . . .
```

Plug-in clients which write instance data must do their own versioning so they can read old forms of their own data, and their own bit twiddling to read and write binary data on machines with different byte order and floating point formats. Clients must also make sure they do not read past the end of their own data. This last restriction may be lifted in future versions.

1.52 s7.5.2

A server manages its instances by providing LightWave with functions to create, destroy, load and save them. The server activation function gets a handler structure which it initializes with the standard instance handler functions listed here, plus whatever else is required by the specific class of plug-in.

create create a default instance. Any failure should return a null pointer and optionally set the error value.

destroy dispose of an instance.

copy copy the contents of the "from" instance to the "to" instance.

load read an instance description from a file into an already created instance.

save write an instance description to a file.

The create function should create a default instance which can then be modified by the interface function. The load and copy functions will overwrite existing instances with new values read from a file or a source instance.

```
    (60)  Animation Types
  . . .
  typedef struct LWInstHandler {
    LWInstance      (*create) (LWError *);
    void            (*destroy) (LWInstance);
    LWError         (*copy) (LWInstance from, LWInstance to);
    LWError         (*load) (LWInstance, const LWLoadState *);
    LWError         (*save) (LWInstance, const LWSaveState *);
  } LWInstHandler;
  . . .
```

1.53 s7.5.3

There is often another server associated with a handler and that is the "Interface" server. The activation function for the interface server is called with a LWInstance as its local data. The server will then allow the user to edit the instance description and return. The interface server is just another function that operates on instances of a specific type, but it is separate from the other instance handler functions for two reasons. The first is that the user interface code is frequently the largest and least often used part of a handler, so it makes sense to allow it to be loaded separately only when needed. The second is to permit plug-in authors to easily make "render-only" versions of their plug-in servers for rendering accelerators or packaging bundles.

For example, if the plug-in type was "XXX", there would be two classes of server, "XXXHandler" and "XXXInterface". Then for a specific server of the XXX type, called "MyXXX", there would be a MyXXX defined for the XXXHandler class which would provide all the normal handler functions, and there would also be a MyXXX server of the XXXInterface class which would perform the user interface.

1.54 s8

The many server classes for Layout provide a wide range of capabilities and extentions to basic LightWave rendering and animation. Since there are so many servers there may be multiple ways to accomplish the same effect, some better than others.

8.1 Utilities
8.2 Image Post Processing
8.3 Procedural Texture
8.4 Procedural Displacement Map
8.5 Procedural Item Animation
8.6 Procedural Object Replacement
8.7 Frame Buffers
8.8 Animation Output
8.9 Scene Conversion
8.10 General Function

1.55 s8.1

The "ImageSaver" class, described in the "LightWave Images" document is used by Layout to save output images in different formats.

1.56 s8.2

The "ImageFilterHandler" (and "ImageFilterInterface") class is used to apply image post processing (filtering) effects to the final rendered image. Each filter is applied after all the antialiasing and motion blur passes are complete, and the server modifying the red, green,

blue and alpha values of the final image.

8.2.1 Input Buffers
8.2.2 Filter Access
8.2.3 Handler

1.57 s8.2.1

In addition to looking at the RGBA of the image, the server can compute its effects based on a potentially large set of full-image buffers, given by the LWBUF codes below. Each of these is a full-screen array of 0-255 BufferValues indicating the presence or absence of that particular attribte for each pixel in the final image.

RED, GREEN, BLUE and ALPHA
These buffers are the outputs of the rendering pass and are the base which should be modified by the server. These are always provided to every image filter.

SPECIAL This value is assigned by the user on a surface by surface basis which is used only for this filter. This is designed to be used to activate the post processing effect for specific surfaces, and user-assigned percentages show up here as 0-255 values in the buffer.

LUMINOUS..RAW_BLUE
These eight buffers are the raw values of the surface parameters before shading.

SHADING This buffer is a picture of the diffuse shading applied to the raw shapes in the image.

SHADOW This indicates where shadows are falling in the final image. It may also be thought of as an illuminations map, showing what parts of the image are visible to the lights in the scene.

GEOMETRY The value in this buffer is computed from the dot-product of the surface normal with the eye vector. It reveals something about the underlying shape of the objects in the image. Where this buffer is 255 (or 1.0) the surface is facing directly toward the camera, and where this buffer is 0, the surface is edge-on to the camera.

DEPTH The depth buffer is a map of the distance of each pixel from the camera plane. This buffer is different from all the others because it is floating point, and because it is not anti-aliased or motion-blured.

(61) Animation Servers

```
#define LWBUF_SPECIAL    0

#define LWBUF_LUMINOUS   1
#define LWBUF_DIFFUSE    2
```

```
 #define LWBUF_SPECULAR   3
 #define LWBUF_MIRROR     4
 #define LWBUF_TRANS      5
 #define LWBUF_RAW_RED    6
 #define LWBUF_RAW_GREEN  7
 #define LWBUF_RAW_BLUE   8

 #define LWBUF_SHADING    9
 #define LWBUF_SHADOW     10
 #define LWBUF_GEOMETRY   11
 #define LWBUF_DEPTH      12

 #define LWBUF_RED        32
 #define LWBUF_GREEN      33
 #define LWBUF_BLUE       34
 #define LWBUF_ALPHA      35
 . . .
```

1.58 s8.2.2

At each frame that the filter is active, the server will get the image to process. It reads the contents of the image buffers and writes new RGB and Alpha data to the output buffer and exits when it has processed the entire frame. This processing is done using a 'FilterAccess' structure which contains data fields and functions.

width, height
: This is the total size of the input and output image buffers. Filters cannot change the image size and all buffers are the same size.

frame
: This is the frame number of this final image.

start, end
: These two times are the start and end times for the frame. The times are the same unless the frame has motion-blur, in which case the difference between them is the "exposure time" for the frame.

bufLine, fltLine
: The functions allow access to the input buffers and return pointers to a line of the buffer of the given type. For y=0, the top line of the buffer is returned; for y=1 the second to the top line, etc. 'bufLine' returns lines from byte-encoded buffers and 'fltLine' returns lines from float-encoded buffers (currently only LWBUF_DEPTH). Invalid type codes return null pointers.

setRGB, setAlpha
: The output buffers must be set using these functions which set the final value at a pixel location. The input RGBA buffers do not change as the output buffers are modified. A filter must set every pixel in the output image even if it does not alter the value, but it can set them in any order.

```
monitor     This monitor can be used by the server to update the host
      about its progress through the frame.  As with all
      monitors, the number of steps should be kept fairly low
      since checking for abort can have significant overhead on
      some systems.  Every line or every other line should be
      about right.

   (62)  Animation Servers
 . . .
 typedef struct st_FilterAccess {
   int               width, height;
   LWFrame           frame;
   LWTime            start, end;
   BufferValue *   (*bufLine) (int type, int y);
   float *         (*fltLine) (int type, int y);
   void            (*setRGB)   (int x, int y, BufferValue[3]);
   void            (*setAlpha) (int x, int y, BufferValue);
     #ifndef LW_PRERELEASE
   Monitor          *monitor;
     #endif
 } FilterAccess;
 . . .
```

1.59 s8.2.3

The activation function for an image filter gets passed a blank handler structure as its local data which the server must fill in. In addition to the normal instance functions, it must also provide a 'process' function and 'flags' function.

```
process     This is the function which filters a single frame given
      an instance and the access structure.

flags       This returns a set of bits representing the buffers this
      instance wants at processing time, where the bit numbers
      are the LWBUF values above.  Only buffers 0-12 need to be
      specified this way since the R, G, B and Alpha buffers
      are always provided.  Undefined bits should be clear by
      default.

   (63)  Animation Servers
 . . .
 typedef struct st_ImageFilterHandler {
   LWInstHandler     inst;
   void            (*process) (LWInstance, const FilterAccess *);
   unsigned int    (*flags) (LWInstance);
 } ImageFilterHandler;
 . . .
```

1.60 s8.3

The "ShaderHandler" (and "ShaderInterface") class is for modifying the attributes of a pixel as it is being rendered. These are sometimes called "procedural textures," but in the LightWave implementation they are quite a bit more powerful than that. Since it is called on a per-pixel basis, this interface is designed for speed.

As LightWave goes through the process of converting abstract 3D surfaces into imagery, it breaks surfaces down into tiny patches which each get a uniform color. Computing the color of these tiny spots is done by starting from a set of basic surface parameters which are approximately constant over the patch: base color, surface normal, luminosity, diffuse reflection, specular reflection, reflectivity, transparency, refractive index and roughness (or glossiness). From these values LightWave's illumination calculation computes the color and intensity of reflected light and transmitted light and determines the color of the spot as seen from the given viewpoint. Plug-in shaders can either alter the base parameters and let LightWave do the rendering calculation, or they can perform the illumination themselves and compute the preceived color directly.

8.3.1 Shader Access
8.3.2 Geometric Parameters
8.3.3 Modifiable Parameters
8.3.4 Shading Functions
8.3.5 Instance

1.61 s8.3.1

The spot evaluation function is called for every visible spot on a surface with a 'ShaderAccess' structure describing the spot to be shaded. The access structure for contains some values which are read-only and some which are meant to be modified. The read-only values describe the geometry of the pixel being shaded. The read-write values describe the current parameters of this pixel and should be modified in place to affect the final look of the spot. Since shaders may be layered, these properties may be altered many more times before final rendering. The access structure also contains special functions usable only while rendering.

```
    (64)  Animation Servers
  . . .
  typedef struct st_ShaderAccess {
    <Read-only shader parameters>
    <Modifiable shader parameters>
    <Shader functions>
  } ShaderAccess;
  . . .
```

1.62 s8.3.2

The spot parameters are read-only and describe the local geometry of

the spot being shaded.

sx, sy Spot location in the final image in pixel coordinates with (0,0) at the upper-left.

oPos, wPos Spot position in object coordinates and world coordinates.

gNorm Geometric normal in world coordinates. This is the raw polygonal normal at the spot, unperturbed by smoothing or bump mapping.

spotSize Approximate spot diameter. This is a very approximate value since spots on a surface viewed on edge are long and thin. This can be used to compute texture antialiasing.

raySource Origin of the incoming viewing ray in world coordinates. Often this will be the camera but it does not have to be.

rayLength The distance the viewing ray traveled in free space to reach this spot.

cosine This is the cosine of the angle between the viewing ray and the surface normal at this spot. It indicates how glancing the view is and gives a measure of how approximate the spot size is.

oXfrm, wXfrm
Object to world and world to object transformation matrices. This can be computed other ways, but are included here for speed and are intended to be used primarily for directional vectors.

objID The object being shaded. A single shader instance can be shared between multiple objects, so this may be different for each evaluation. For sample sphere rendering the ID will refer to an object not in the current scene.

polNum The polygon number of the object being shaded. While this will be the polygon number for normal mesh objects, it may represent other sub-object information in non-mesh objects.

(65) Read-only shader parameters

```
int              sx, sy;
double           oPos[3], wPos[3];
double           gNorm[3];
double           spotSize;
double           raySource[3];
double           rayLength;
double           cosine;
double           oXfrm[9],  wXfrm[9];
LWItemID         objID;
#ifndef LW_PRERELEASE
int              polNum;
```

```
  #endif
```

1.63 s8.3.3

These parameters are used by the renderer to compute the perceived color at the spot and may be modified by the shader. The shader must return the correct flags for any value it will modify or the change will not take effect (see below).

wNorm — Surface normal in world coordinates. Modifying this makes the surface look bumpy without altering the geometry (bump mapping). The shader needs to renormalize the vector after perturbation.

color — Base color of the spot.

luminous — Percentage luminosity.

diffuse — Percentage diffuse reflection.

specular — Percentage specular reflection.

mirror — Percentage reflectivity.

transparency — Percentage transparency.

eta — Index of refraction.

roughness — Surface roughness, often expressed as the inverse of glossiness.

(66) Modifiable shader parameters

```
 double          wNorm[3];
 double          color[3];
 double          luminous;
 double          diffuse;
 double          specular;
 double          mirror;
 double          transparency;
 double          eta;
 double          roughness;
```

To set the perceived color directly a shader can set all the parameters to zero except for luminous which is 1.0 and color which is the output color of the spot.

1.64 s8.3.4

Special functions are provided to shaders which are not available in any other context.

```
illuminate  This function returns the light ray (color and direction)
      hitting the given position from the given light at the
      current instant.  The return value is zero if the light
      does not illuminate the given world coordinate position
      at all.  The color includes effects from shadows (if
      any), falloff, spotlight cones and transparent objects
      between the light and the point.

rayTrace    This function may be called to trace a ray from the a
      given location in a given direction (in world
      coordinates).  The return value is the length of the ray
      (or -1.0 if infinite) and the color coming from that
      direction.  The direction used is the outgoing direction
      and must be normalized to be a unit vector.

   (67)  Shader functions

 int            (*illuminate) (LWItemID light,
             const double position[3],
             double direction[3],
             double color[3]);
 double         (*rayTrace) (const double position[3],
           const double direction[3],
           double color[3]);
```

1.65 s8.3.5

A shader instance may store its data in an object (in the case of a surface texture) or in a scene (in the case of a clip map) so the save/load functions should be prepared to deal with both cases.

```
init        Called at the start of rendering a sequence of frames.

cleanup     Called when current sequence is complete.

newTime     Called at the start of each new time within the current
      sequence.

evaluate    Called to compute the shading of each affected pixel
      within the current time.

flags       Returns a word containing status bits for the instance.
      Undefined flag bits should be clear by default.  The
      first nine LWSHF bits should be set only if the shader
      instance is going to modify that particular attribute.
      RAYTRACE must be set if the shader intends to use the
      'rayTrace' function.

   (68)  Animation Servers
 . . .
 #define LWSHF_NORMAL    (1<<0)
 #define LWSHF_COLOR     (1<<1)
 #define LWSHF_LUMINOUS  (1<<2)
 #define LWSHF_DIFFUSE   (1<<3)
```

```
#define LWSHF_SPECULAR  (1<<4)
#define LWSHF_MIRROR    (1<<5)
#define LWSHF_TRANSP    (1<<6)
#define LWSHF_ETA       (1<<7)
#define LWSHF_ROUGH     (1<<8)
#define LWSHF_RAYTRACE  (1<<10)
. . .
```



```
. . .
typedef struct st_ShaderHandler {
  LWInstHandler     inst;
  LWError          (*init) (LWInstance);
  void             (*cleanup) (LWInstance);
  LWError          (*newTime) (LWInstance, LWFrame, LWTime);
  void             (*evaluate) (LWInstance, ShaderAccess *);
  unsigned int     (*flags) (LWInstance);
} ShaderHandler;
. . .
```

1.66 s8.4

The "DisplacementHandler" (and "DisplacementInterface") class is called upon before rendering to modify the geometry of an object. This is done not only during rendering but also during interactive previewing in the Layout window. This means that a server should always be prepared to process a displacement instance at any time.

8.4.1 Displacement Access
8.4.2 Handler

1.67 s8.4.1

At its core a displacement handler takes point coordinates and moves them for each timestep. The access structure for a displacement map gets the position of the point to displace in two ways.

oPos This is the point location in object coordinates and is read-only. The server may use this in computations, but moving it has no effect.

source This is the location to be transformed in place by the displacement. If this is not a world-coordinate displacement, then the source coordinates are the in the object coordinate system but have been already displaced by any morphing or boning applied to the object, and may differ from the object coordinates. If the displacment is in world coordinates (see ‘flags’ below), then the source coordinates are morphed, boned and transformed by object motion (i.e. they are world coordinates).

```
. . .
typedef struct st_DisplacementAccess {
  double           oPos[3];
  double           source[3];
} DisplacementAccess;
. . .
```

1.68 s8.4.2

The handler functions for a displacement map are the same as a shader except for the lack of 'init' and 'cleanup' functions. The 'newTime' function also has a parameter for the ID of the object being affected by the displacement. The LWDMF_WORLD bit should be set in the 'flags' return value if the displacement will take place in world coordinates.

```
   (71)  Animation Servers
. . .
typedef struct st_DisplacementHandler {
  LWInstHandler     inst;
  LWError          (*newTime) (LWInstance, LWItemID,
            LWFrame, LWTime);
  void             (*evaluate) (LWInstance,
             DisplacementAccess *);
  unsigned int     (*flags) (LWInstance);
} DisplacementHandler;

#define LWDMF_WORLD     (1<<0)
. . .
```

1.69 s8.5

The "ItemMotionHandler" (and "ItemMotionInterface") class is used to apply animation behavior to any item in a scene which can be keyframed. After the keyframe position of the item is computed, the item motion server can alter the keyframed motion or replace it with a completely different one. Motions will be evaluated both during rendering and while interactively laying out a scene.

```
      8.5.1  Item Motion Access
      8.5.2  Handler
```

1.70 s8.5.1

At each time instant and for each affected item, the motion evaluation function will be called with an access structure holding the ID of the item and the time instant for which the motion should be computed. The server can query keyframe parameters for the item and sets it own values for the current time.

```
 item        This is set to the ID for the item to be affected by the
```

```
      procedural motion.

 frame, time This is set to the current instant for which the motion
      should be evaluated.

 getParam    Returns the keyframed motion set by the user for the item
      at any given time.  Only the POSITION, ROTATION and
      SCALING parameters may be queried.

 setParam    Used by the evaluation function to set the computed
      motion of the item at the current time.  Only the
      POSITION, ROTATION and SCALING parameters may be set.

    (72)  Animation Servers
  . . .
  typedef struct st_ItemMotionAccess {
    LWItemID          item;
    LWFrame           frame;
    LWTime            time;
    void            (*getParam) (LWItemParam, LWTime,
                double vector[3]);
    void            (*setParam) (LWItemParam,
                const double vector[3]);
  } ItemMotionAccess;
  . . .
```

Procedural motions are not currently allowed to interact. If a motion evaluation function attempts to read out the position of another object which is affected by a procedural motion, only the values of the keyframed motion will be returned.

1.71 s8.5.2

The handler for item motions adds only the ‘evaluate’ function to the standard set of handler functions. This computes the motion for an item at a given timestep, and may be called at any time.

```
    (73)  Animation Servers
  . . .
  typedef struct st_ItemMotionHandler {
    LWInstHandler     inst;
    void            (*evaluate) (LWInstance,
                const ItemMotionAccess *);
  } ItemMotionHandler;
  . . .
```

1.72 s8.6

The "ObjReplacementHandler" (and "ObjReplacementInterface") class allows another type of animation which can replace the entire object geometry at every single step. Replacement is done by object name, so the server evaluation function can provide a new object name to load

for each subframe timestep, or it can only load a new object
periodically, allowing the same geometry to persist for a length of
time.

Filenames are used instead of direct mesh replacement for generality.
An object replacement server could use a series of prebuilt objects,
like character heads for example, to do expressions or lip-syncing by
providing the name of the correct head at each step. Some animation
could be done very efficiently using a combination of object
replacement and object import servers. The replacement server could
write a brief description file for the parameters of a timestep
(positions and sizes of metaballs, for example) which the object
import server could then convert into a complete mesh while loading.
A simple form of this server could be used to replace objects with
nulls when they are not visible in the scene.

      8.6.1 Object Replacement Access
      8.6.2 Handler

1.73 s8.6.1

The access structure passed to the evaluation function contains
information about the currently loaded object and the next timestep.
The server compares the current settings and the next step and
provides a new filename if a different object should be loaded for the
next timestep to be evaluated. If the currently loaded geometry can
be used for the new frame and time, then the new filename can be set
to null.

 objectID   Item ID for the object whose geometry may be replaced by
       this server.

 curFrame, curTime, newFrame, newTime
       The frame and time values for the currently loaded
       geometry and the next step. New geometry should be
       loaded if the object needs to look different at the two
       different times. The times may not be sequential, since
       network rendering can cause the renderer to jump around
       between non-sequential times.

 curType, newType
       The type of the geometry currently loaded and needed for
       the next timestep. The server can provide different
       geometry for interactive previewing and actual rendering
       by examining this value. OBJREP_NONE is only used when
       there is no geometry loaded at all for the current time.

    (74) Animation Servers
  . . .
  #define OBJREP_NONE   0
  #define OBJREP_PREVIEW   1
  #define OBJREP_RENDER   2
  . . .

 curFilename This is set to the object geometry file currently loaded,

```
       and may be null if there is no geometry loaded.

 newFilename This is the filename of a new object file to be loaded as
       the geometry for this item at the new timestep, and is
       the only field set by the server.  It should only be set
       if the new geometry differs from that currently loaded,
       since loading new geometry incurs significant overhead.

    (75)  Animation Servers
  . . .
  typedef struct st_ObjReplacementAccess {
    LWItemID         objectID;
    LWFrame          curFrame, newFrame;
    LWTime           curTime,  newTime;
    int              curType,  newType;
    const char      *curFilename;
    const char      *newFilename;
  } ObjReplacementAccess;
  . . .
```

In Layout 4.0, curType is always set to OBJREP_NONE and curFrame, curTime and curFilename are not set. 4.0 treats every frame as if no model were loaded.

1.74 s8.6.2

In addition to the normal handler functions, the server provids an ‘evaluate’ function which is called for each affected object at each timestep to get new geometry. This function can be called at any time while rendering or setting up animations.

```
    (76)  Animation Servers
  . . .
  typedef struct st_ObjReplacementHandler {
    LWInstHandler     inst;
    void             (*evaluate) (LWInstance,
                ObjReplacementAccess *);
  } ObjReplacementHandler;
  . . .
```

1.75 s8.7

The "FrameBufferHandler" (and "FrameBufferInterface") class is used to display the output of rendering as each frame is completed. This is for the user to view, so the frame buffer should also be able to pause waiting for user input.

A frame buffer is an instance, but it may be very limited. The built-in frame buffers have no UI and no stored state.

```
 open        Open display at the given size.
```

```
 close       Close display and end display transations.

 begin       Start a new frame.

 write       Write a new line of RGB and alpha data to the
       framebuffer.  Lines always come from top to bottom and
       there are always enough to fill the width and height of
       the requested display.

 pause       Display the buffer to the user and wait for their signal
       to continue before returning.

The sequence of calls for rendering to the frame buffer can be
visualized as a regular expression:

open, (begin, (write)H, pause?)*, close

Any number of frames may be displayed in a session (even zero).  Write
will always be called for all the lines in the image and pause is
optional.

    (77)  Animation Servers
  . . .
  typedef struct st_FrameBufferHandler {
    LWInstHandler     inst;
    LWError          (*open) (LWInstance, int w, int h);
    void             (*close) (LWInstance);
    LWError          (*begin) (LWInstance);
    LWError          (*write) (LWInstance,
            const BufferValue *R,
            const BufferValue *G,
            const BufferValue *B,
            const BufferValue *alpha);
    void             (*pause) (LWInstance);
  } FrameBufferHandler;
  . . .
```

1.76 s8.8

```
The "AnimSaverHandler" (and "AnimSaverInterface") class is used to
write out animations.  The scheme is nearly identical to framebuffers,
except that there is no 'pause' function and in addition to the image
size, LightWave will also pass a filename for the animation file.

    (78)  Animation Servers
  . . .
  typedef struct st_AnimSaverHandler {
    LWInstHandler     inst;
    LWError          (*open) (LWInstance, int w, int h,
           const char *filename);
    void             (*close) (LWInstance);
    LWError          (*begin) (LWInstance);
    LWError          (*write) (LWInstance,
            const BufferValue *R,
            const BufferValue *G,
```

```
            const BufferValue *B,
            const BufferValue *alpha);
  } AnimSaverHandler;
  . . .
```

1.77 s8.9

The "SceneConverter" class is used in import foreign scene formats. When the user selects a file to load as a scene, LightWave first attempts to load it directly as an LWSC format file. It it cannot, it will pass the filename to each scene converter in sequence. The scene converter will attempt to read the file and rewrite it as an LWSC file. After successful translation the server will pass the name of the new scene back to LightWave. The file will be loaded and the server will be called back again to delete the translated scene file.

filename Filename of foreign scene file. This is set by the host before activating the server. This is the file to try to parse.

readFailure If the server can recognize the format but cannot parse the file for some reason, it should set this error return value.

tmpScene If the server sucessfully parses the foreign scene file, it should write a translation of that scene as a LWSC format file and return the name of this translation scene in this field.

deleteTmp After reading the temporary scene file set above, the host will call back this delete function to dispose of the file and any other temporary state. The ‘tmpScene’ and ‘deleteTmp’ fields should be set as a pair before the server returns.

```
   (79)  Animation Servers
  . . .
  typedef struct st_SceneConverter {
    const char       *filename;
    LWError           readFailure;
    const char       *tmpScene;
    void            (*deleteTmp) (const char *tmpScene);
  } SceneConverter;
```

When the server is called, only ‘filename’ will be set. It then must set the other three fields to one of the following configurations:

readFailure and tmpScene both null
This indicates that the server was unable to recognize the file format and no translation was done. LightWave will simply try the next translator.

readFailure set, tmpScene null
This indicates that the file format was recognized, but that a failure of some kind occured during translation.

LightWave will display this error and will stop
attempting to translate the file.

readFailure null, tmpScene set
This indicates successful translation. LightWave will
read 'tmpScene' as an LWSC file and then will call the
'deleteTmp' function to dispose of it. Note that if
tmpScene is set, deleteTmp must be set as well.

1.78 s8.10

The "LayoutGeneric" class is provided for general layout functionality which does not fit into any of the previous server or handler categories. Servers of this class can be activated by the user from the Layout interface to perform non-rendering functions, such as configuring external devices, performing calculations, etc.

Normal global information is available to this class of server, but the local pointer is unused.

1.79 s9

This section contains descriptions of the global data pointers which can be accessed from LightWave's global function. The ID string for each global is given in quotes.

9.1 Item Information
9.2 Object Information
9.3 Bone Information
9.4 Light Information
9.5 Camera Information
9.6 Scene Information
9.7 Image List Information
9.8 Compositing Information
9.9 Global Rendering Memory Pool

1.80 s9.1

The global ID "LW Item Info" returns functions for traversing the entire set of items in the scene and getting information about all of them. This information is common to all items. Any information specific to certain item types is given by separate global functions.

first Returns the ID of the first item of a given type. If
type is LWI_BONE, the second argument is the ID of the
boned object. If there are no items of this type this
returns LWITEM_NULL.

next Returns the next item of the same type as the argument.
If there are no more, this returns LWITEM_NULL.

```
firstChild  Returns the first child item of the parent item.  It
      returns LWITEM_NULL if none.

nextChild   Returns the next child item given a parent item and the
      previous child.  It returns LWITEM_NULL if that was the
      last one.

parent      Returns the item's parent, if any, and LWITEM_NULL if
      none.

target      Returns the item's target, if any, and LWITEM_NULL if
      none.

goal        Returns the item's goal, if any, and LWITEM_NULL if none.

type        Returns the type of an arbitrary item.

name        Returns the name of the item as it appears to the user.

param       Returns vector parameters from an item using a
      LWItemParam code to identify the parameter desired.  The
      value is written to the vector array for the given time.

limits      Returns upper and lower bounds on vector parameters.
      These may be limits set by the user on joint angles or
      ranges of movement.  LWVECF flag bits are returned to
      indicate which of the three vector indicies contain
      limits.  Any bits unset are unbounded.

   (80)  Animation Types
 . . .
 #define LWVECF_0        (1<<0)
 #define LWVECF_1        (1<<1)
 #define LWVECF_2        (1<<2)
 . . .

   (81)  Animation Globals

 typedef struct st_LWItemInfo {
   LWItemID        (*first)  (LWItemType, LWItemID);
   LWItemID        (*next)   (LWItemID);
   LWItemID        (*firstChild) (LWItemID parent);
   LWItemID        (*nextChild) (LWItemID parent, LWItemID
      prevChild);
   LWItemID        (*parent) (LWItemID);
   LWItemID        (*target) (LWItemID);
   LWItemID        (*goal)   (LWItemID);
   LWItemType      (*type)   (LWItemID);
   const char *    (*name)   (LWItemID);
   void            (*param)  (LWItemID, LWItemParam, LWTime,
            double vector[3]);
   unsigned int    (*limits) (LWItemID, LWItemParam,
            double min[3], double max[3]);
 } LWItemInfo;
 . . .
```

1.81 s9.2

```
The global ID "LW Object Info" returns functions for object-specific
information.

 filename    Returns the filename for the object file.

 numPoints, numPolygons
       Returns the number of points and polygons in the object
       mesh.

 shadowOpts  Returns bits for shadow options, as below.

    (82)  Animation Types
  . . .
  #define LWOSHAD_SELF     (1<<0)
  #define LWOSHAD_CAST     (1<<1)
  #define LWOSHAD_RECEIVE  (1<<2)
  . . .

 dissolve    Returns the object dissolve percentage as a function of
       time.

    (83)  Animation Globals
  . . .
  typedef struct st_LWObjectInfo {
    const char *     (*filename) (LWItemID);
    int              (*numPoints) (LWItemID);
    int              (*numPolygons) (LWItemID);
    unsigned int     (*shadowOpts) (LWItemID);
    double           (*dissolve) (LWItemID, LWTime);
  } LWObjectInfo;
  . . .
```

1.82 s9.3

```
The global ID "LW Bone Info" returns functions for getting
bone-specific information.

 flags       Returns a set of flag bits for the given bone, as
       follows.

    (84)  Animation Types
  . . .
  #define LWBONEF_ACTIVE          (1<<0)
  #define LWBONEF_LIMITEDRANGE    (1<<1)
  . . .

 restParam   This gets vector parameters for the rest position of a
       given bone.  Parameters of the animated bone can be read
       from the normal item info functions.

 restLength  This gets the special rest length parameter of the given
       bone.
```

```
limits      For limited range bones, this gets the inner and outer
      limit radii for the bone.  Influence areas are in the
      shape of a cylinder with hemispherical ends centered at
      the tips of the bone.

   (85)  Animation Globals
 . . .
 typedef struct st_LWBoneInfo {
   unsigned int    (*flags) (LWItemID);
   void            (*restParam) (LWItemID, LWItemParam,
               double vector[3]);
   double          (*restLength) (LWItemID);
   void            (*limits) (LWItemID, double *inner,
            double *outer);
 } LWBoneInfo;
 . . .
```

1.83 s9.4

```
The global ID "LW Light Info" returns functions for getting
light-specific information.

 ambient     Returns the ambient light color (with intensity factored
       in) at the given time.  There is no light ID needed since
       this is global to the scene.

 type        Returns the type of the given light as one of the
       following values.

   (86)  Animation Types
 . . .
 #define LWLIGHT_DISTANT  0
 #define LWLIGHT_POINT    1
 #define LWLIGHT_SPOT     2
 . . .

 color       Returns the light color (with intensity factored in) at
       the given time.

 shadowType  Returns the shadow type for the given light as one of the
       following values.

   (87)  Animation Types
 . . .
 #define LWLSHAD_OFF      0
 #define LWLSHAD_RAYTRACE 1
 #define LWLSHAD_MAP      2
 . . .

 coneAngles  Returns the cone angles for spotlights.  Radius is half
       the total light code angle and edge is the angular width
       of the soft edge.

   (88)  Animation Globals
```

```
  . . .
  typedef struct st_LWLightInfo {
    void            (*ambient) (LWTime, double color[3]);
    int             (*type) (LWItemID);
    void            (*color) (LWItemID, LWTime, double color[3]);
    int             (*shadowType) (LWItemID);
    void            (*coneAngles) (LWItemID, double *radius,
                  double *edge);
  } LWLightInfo;
  . . .
```

1.84 s9.5

```
The global ID "LW Camera Info" returns functions for accessing
information specific to the camera.  A camera has an ID which must be
passed to these functions in anticipation of multiple cameras per
scene.

 zoomFactor  Returns the zoom factor for the camera at the given time.

 focalLength Returns the focal length of the camera lens at the given
       time.  Focal length is expressed in millimeters.

 focalDistance
       Returns the distance to the focal plane of the camera at
       the given time.

 fStop       Returns the F-Stop number at the given time.

 blurLength  Returns the blur length as a fraction of the frame time
       for the given time.

 fovAngles   Returns the camera field of view angles at the given
       time.  These are angles in radians centered around the
       camera direction.

    (89)  Animation Globals
  . . .
  typedef struct st_LWCameraInfo {
    double          (*zoomFactor) (LWItemID, LWTime);
    double          (*focalLength) (LWItemID, LWTime);
    double          (*focalDistance) (LWItemID, LWTime);
    double          (*fStop) (LWItemID, LWTime);
    double          (*blurLength) (LWItemID, LWTime);
    void            (*fovAngles) (LWItemID, LWTime,
                double *horizontal,
                double *vertical);
  } LWCameraInfo;
  . . .
```

1.85 s9.6

```
The global ID "LW Scene Info" returns a block of information about the
scene itself.  This is all strictly read-only.

 name        User's name for the scene.

 filename    Filename of the scene file.

 numPoints, numPolygons
       Total number of points and polygons for all the objects
       in the scene.

 renderType  This can be one of the following values.

    (90)  Animation Types
  . . .
  #define LWRTYPE_WIRE            0
  #define LWRTYPE_QUICK           1
  #define LWRTYPE_REALISTIC       2
  . . .

 renderOpts  This is a combination of bits for different rendering
       options.  EVENFIELDS is set only if field rendering is on
       and the first line of the output image is from the field
       that comes first in time.

    (91)  Animation Globals
  . . .
  #define LWROPT_SHADOWTRACE      (1<<0)
  #define LWROPT_REFLECTTRACE     (1<<1)
  #define LWROPT_REFRACTTRACE     (1<<2)
  #define LWROPT_FIELDS           (1<<3)
  #define LWROPT_EVENFIELDS       (1<<4)
  #define LWROPT_MOTIONBLUR       (1<<5)
  #define LWROPT_DEPTHOFFIELD     (1<<6)
  #define LWROPT_LIMITEDREGION    (1<<7)
  . . .

 frameStart, frameEnd, frameStep
       The range of frames defined for the scene.

 framesPerSecond
       Number of frames per real-time second.  This will be 30
       for video (even field rendered), and 24 for film.

 frameWidth, frameHeight
       Final output image size in pixels.

 pixelAspect Pixel aspect ratio as pixel-width / pixel-height.  Values
       greater than one mean short wide pixels and values less
       than one mean tall thin pixels.

 minSamplesPerPixel, maxSamplesPerPixel
       Limits on number of samples per pixel in the final image.
       Because of different rendering techniques and adaptive
       sampling it is impossible to compute a precise number of
       antialiasing samples at any pixel, but this gives a range
```

```
      for the current rendering options.

limitedRegion
      The location of the limited region area, given as x0, y0,
      x1, y1.

   (92)  Animation Globals
 . . .
 typedef struct st_LWSceneInfo {
   const char      *name;
   const char      *filename;
   int              numPoints;
   int              numPolygons;
   int              renderType;
   int              renderOpts;
   LWFrame          frameStart;
   LWFrame          frameEnd;
   LWFrame          frameStep;
   double           framesPerSecond;
   int              frameWidth;
   int              frameHeight;
   double           pixelAspect;
   int              minSamplesPerPixel;
   int              maxSamplesPerPixel;
   int              limitedRegion[4];      /* x0, y0, x1, y1 */
 } LWSceneInfo;
 . . .
```

1.86 s9.7

The global ID "LW Image List" returns functions for traversing LightWave's image list and accessing values in the image. Images are identified by an abstract data type.

```
   (93)  Animation Types
 . . .
 typedef void *           LWImageID;
 . . .

first       Returns the first image in the list, null if none.

next        Returns the next image after the given one, null if none.

load        Loads a file as an image, adds it to the list and returns
      it.

name        Returns the user's name for an image.

filename    Returns the filename for the loaded image.  This is the
      value that should be stored for later retrieval of the
      image using 'load.' If the ID refers to an image
      sequence, the frame number will be used to construct the
      appropriate image filename.

isColor     Returns true if the image has color data or false if only
```

greyscale.

needAA This needs to be called by shaders that want to use the "spot" functions to access values in the image in the course of their shading calculations. This function can only be called from a shader's 'init' function.

size Returns the width and height of the image in pixels.

luma Returns the greyscale value of the image from 0-255. If this is a color source image the value returned is the NTSC luminence.

RGB Returns the RGB color of the image from 0-255 at the given pixel.

lumaSpot, RGBSpot
Returns the floating point greyscale or color value of the image for a spot of the given diameter at the given center in the image. These functions can only be called during the spot evaluation function of a shader, and 'needAA' must have been called during the shader's initialization. If the spot size is small and 'blend' is true, the color value will be interpolated from between image pixels.

clear Removes the image from the scene, clearing all references.

(94) Animation Globals

```
. . .
typedef struct st_LWImageList {
  LWImageID       (*first) (void);
  LWImageID       (*next) (LWImageID);
  LWImageID       (*load) (const char *);
  const char *    (*name) (LWImageID);
    #ifndef LW_PRERELEASE
  const char *    (*filename) (LWImageID, LWFrame);
    #else
  const char *    (*filename) (LWImageID);
    #endif
  int             (*isColor) (LWImageID);
  void            (*needAA) (LWImageID);
  void            (*size) (LWImageID, int *w, int *h);
  BufferValue     (*luma) (LWImageID, int x, int y);
  void            (*RGB)  (LWImageID, int x, int y,
         BufferValue[3]);
  double          (*lumaSpot) (LWImageID, double x, double y,
             double spotSize, int blend);
  void            (*RGBSpot)  (LWImageID, double x, double y,
             double spotSize, int blend,
             double[3]);
    #ifndef LW_PRERELEASE
  void            (*clear) (LWImageID);
    #endif
} LWImageList;
```

```
  . . .
```

1.87 s9.8

```
The global ID "LW Compositing Info" returns a structure describing the
state of the built-in compositing function.  The three ImageID's are
the background image, the foreground image and the foreground alpha
image.

    (95)  Animation Globals
  . . .
  #ifndef LW_PRERELEASE
  typedef struct st_LWCompInfo {
    LWImageID        bg;
    LWImageID        fg;
    LWImageID        fgAlpha;
  } LWCompInfo;
  #endif
  . . .
```

1.88 s9.9

```
The global ID "Global Render Memory" returns functions for accessing
the Global Rendering Pool.  This is shared memory that can be used
while rendering.  This has two main uses: The first is for read-only
tables, like trig or random noise lookup tables which can be shared by
textures.  The second is for communication areas for textures that
wish to cooperate in terms of sharing computed values on a per-pixel
basis.  LightWave does nothing to manage this shared pool expect to
clear it out after rendering.

The memory chunks are pointers to blocks of memory of different sizes.
They are identified by arbitrary null-terminated character strings.

    (96)  Animation Types
  . . .
  typedef void *          MemChunk;

 first, next These functions allow traversal of the memory chunks in
       the list (pool).  Clients can use these functions if they
       need to search for more complex critera than just ID.

 ID, size    These return the ID string and size of a memory chunk
       given a pointer to the memory.

 find        This returns a pointer to a memory chunk which matches
       the given ID.  Multiple chunks may be created with the
       same ID, so this returns the first one.

 create      This creates a memory chunk with the given size and ID
       and returns a pointer to the memory.  For chunks to be
       unique it is best to try to find the ID before calling
```

```
      this function.

    (97)  Animation Globals
. . .
typedef struct st_GlobalPool {
  MemChunk        (*first) (void);
  MemChunk        (*next) (MemChunk);
  const char *    (*ID)   (MemChunk);
  int             (*size) (MemChunk);
  MemChunk        (*find)   (const char *ID);
  MemChunk        (*create) (const char *ID, int size);
} GlobalPool;
```

1.89 s10

Three header files describe the whole set of LightWave servers and globals. 'lwbase.h' is for the declarations common to both Layout and Modeler, 'lwmod.h' is for Modeler only and 'lwran.h' is for Layout only (Rendering and ANimation).

```
    (98)  Common LightWave Header

/*
 * LWSDK Header File
 * Copyright 1995  NewTek, Inc.
 */
#ifndef LW_BASE_H
#define LW_BASE_H

#include <moni.h>
#include <plug.h>

<Common Server Classes>
<Common Globals>

#endif

    (99)  LightWave Modeler Plug-in Header

/*
 * LWSDK Header File
 * Copyright 1995  NewTek, Inc.
 */
#ifndef LW_MOD_H
#define LW_MOD_H

#include <lwbase.h>

<Modeling Base Types>
<Modeling Types>
<Modeling Servers>
<Modeling Globals>

#endif
```

(100) LightWave Rendering and Animation Plug-in Header

```
/*
 * LWSDK Header File
 * Copyright 1995  NewTek, Inc.
 */
#ifndef LW_RAN_H
#define LW_RAN_H

#include <lwbase.h>

<Animation Types>
<Animation Servers>
<Animation Globals>

#endif
```