ASL Specification

Table Of Contents

Introduction

Anim8or introduces a scripting language ASL with version 0.95. It is included as part of Anim8or. A scripting language allows you to automate tasks that you do often, to extend Anim8or's basic functionality with plug-ins, and to animate elements in your scenes more flexibly and quickly.

ASL is an interpreted language loosely based on the C programming language but with a somewhat restricted set of functionality. It also borrows a few simple syntactic constructs from C++. Aspects of ASL that are the same as or similar to those in C use the same syntax. Those that are different generally use unique syntax to reduce the confusion that users familiar with C might have. Finally ASL includes a number of concepts specific to supporting Anim8or.

ASL scripts can be included within an Anim8or project or kept in a source file on disk. Normally disk files will have an extension of .a8s or .txt but any other extension may be used. The .a8s extension is specifically for scripts that will be preloaded when Anim8or starts up. The .txt extension is for scripts that are run from disk when using Anim8or.

This spec is for Anim8or v1.0. Additions to the scripting language after v1.0 are listed throughout the spec in **BLUE**.

Kinds of Scripts

ASL has several different types of scripts for various purposes. For example there are scripts that export Objects, those that build and manipulate models, and ones to compute the value of a controller in a Scene. They all use a common language but each has its own requirements and restrictions.

Variables and Names

Built-in variable and function names begin with a letter or an underscore character '_' followed by any number of letters, digits and underscores. Upper case letters are distinct from lower case so that 'Anim8or' and 'anim8or are unique names. User declared variables must be preceded by a dollar sign '$' as in '$Anim8or'.

Data Types

The types supported by ASL are:

float

A **float** is a 32 bit IEEE floating point value. A float constant is a number with either a decimal point or an exponent or both:

```
1.234    .7    10.    1e10    3.141e-4
```

int

An **int** is a 32-bit signed integer. Constants may be decimal or hexadecimal:

```
1234    1    0x1234    0xface
```

void

The **void** type is used when defining functions that don't return a value.

```
void $print2(int $arga, int $argb)
{
    $file.print("$arga = %8.4g; $argb = %8.4g\n", $arga, $argb);
}
```

string

A **string** consists of zero or more ASCII characters. Constants are enclosed with double quotes. The backslash character '\' is used as a quote when double quotes are required in a string and to define non-printable characters. They are also used to define new-line characters and tabs as in C. Backslashes within constants must be doubled as well:

```
"hello"
"This is a string. "
"a double quote \" in a string"
"Line 1.\nLine 2.\n"
```

The string type has several useful functions.

```
typedef string {
    int length(void);
    string SubString(int first, int last);
    string Insert(string fStr, int pos);
    int GetChar(int pos);
    string SetChar(int val, int pos);
    string GetDir(void);
    string GetRoot(void);
    string GetExt(void);
};
```

The function `length()` returns the number of characters in a string.

`SubString()` returns a string composed of the first-*th* to the last-*th* characters in a string. Character positions are numbered from 0 to `length() - 1`. Out of bounds values are clamped to the dimensions of the string; if first is greater than last an empty string is returned.

`Insert()` inserts another string into a copy of a string before the character at position pos and returns the result. If pos is `<= 0` the new string is inserted at the first and if it is `>= length()` then it is added at the end.

`GetChar()` returns the value of the character at position pos. If pos is negative or `>= length()` it returns -1. `SetChar()` makes a copy of a string and sets the value of the character at position pos in the copy to val. If pos is out of range it does not change the string and returns the copy unaltered.

`GetDir()`, `GetRoot()` and `GetExt()` return the string corresponding to the director, root and extension of a string, respectively, assuming that it is a valid name of a file.

point2

The type **point2** is a vector of two floating point values. They are similar to the C struct:

```
typedef struct {
    float x, y;
} point2;
```

Individual floating point values are referenced using member notation:

```
$myvar.x    $myvar.y
```

Values are defined as a parenthesized list of two integer or floating point expressions:

```
(1.0, 7)    (0, 0)    (3.142, 1.732)    ($ii, 7*$p)
```

point3

Similarly the type **point3** is a vector of three floating point values. They are similar to the C struct:

```
typedef struct {
    float x, y, z;
} point3;
```

Individual floating point values are referenced using member notation:

```
$myvar.x    $myvar.y    $myvar.z
```

Values are defined as a parenthesized list of three integer or floating point expressions:

```
(1.0, 7, 9.999)    (0, 0, 0)    (3.142, 1.732, -1.414)
($loc.x, $loc.y*1.5, 0.0)
```

quaternion

The type **quaternion** is a vector of four floating point values. They are used to represents a rotation or orientation in Anim8or.

```
typedef struct {
    float x, y, z, w;
} quaternion;
```

Individual floating point values are referenced using member notation:

```
$q0.x    $q0.y    $q0.z    $q0.w
```

Quaternion values may be constructed from 4 scalar values:

```
(1.0, 7, 9.999, -1e8)    (sin($angle), cos($angle), 0, 0)
```

float4x4

The type **float4x4** is a 4 by 4 element array. They are used for things like three dimensional transformations.

file

A **file** is a computer file for reading or writing data. They can only be used in specific kinds of scripts such as import and export plug-in scripts, command scripts and for debug output messages.

Various Anim8or Types

A number of types are used to refer to specific parts of an Anim8or project. Their internal structure is generally hidden. You can access and set various values in these types using built-in functions which are described later.

project - An Anim8or project.
object - An Object within an Anim8or project.
figure - A Figure in an Anim8or project.
sequence - A Sequence in an Anim8or project.
scene - A Scene in an anim8or project.
material - A material.
shape - A 3D component in an Object. These include Spheres, Cubes, Meshes, Subdivision Objects, etc.
meshdata - The specific geometry of a shape as represented within Anim8or.
tridata - A simplified version of meshdata that is useful when exporting data.

arrays

Arrays in ASL are much more general than those in C. They are defined with an initial size but it can be increased or reduced by a running script. Elements of an array are referenced in the usual way:

```
$myarray[10]    $colors[$i + 12]
```

Additionally all arrays contain a struct like predefined member 'size':

```
$myarray.size
```

Referencing size returns the number of elements that the array currently holds. Assigning a value to size sets the number of elements to that value. When an array's size is increased in this manner new elements are assigned a value of 0, an empty string, or a null pointer depending on the type of the array. If the size is reduced the higher indexed values are discarded.

There are also two C++ like member functions for arrays `pop()` and `push()`:

```
<type> $myarray.pop(void);
int $myarray.push(<type> $value);
```

`pop()` removes the last elements from the array and returns its value, thus reducing the array's size by one. `push()` increases the size of an array by one and sets the value of the last element to its parameter. It returns the index of the element pushed.

Comments

You can use both C-style comment syntax. Text within comments is considered white space.

```
int $abc, /* $def;
float        $ghi,
          */ $jkl;     /* Variables $abc and $jkl are both declared as int. */
int $varName;          // $varName is the name of an int variable.
```

Predefined Constants

There are several predefined constants in ASL. General constants are listed below. Other values that are only meaningful as parameters to certain functions are listed later with those functions.

```
int true = 1
int false = 0
float PI = 3.1415926;
int VERSION;    // Current Anim8or version: v1.00 returns 100
```

Variable Declarations

Variables are declared similarly to how they are in C. However there are some differences:

- Variables cannot be initialized in the declaration.
- Declarations don't have to be grouped at the first. They can appear after executable statements, like in C++, but
- There are only two levels of scope, global and local to functions. A variable can be used anywhere within it's scope after it is declared.

```
int $i, $count;
float $sizes[10]; /* array initially with 10 elements */
shape $mymodel;
float pie = 3.1415926; /* Not allowed */
```

Expressions

ASL supports many of the operators found in C.

Function Calls

ASL has a number of built in member and non-member functions. These are described elsewhere in this document. ASL also supports user defined functions but with some restrictions. See the section **Functions** for details..

Unary Operators

```
-    negation: int, float, point2, point3, quaternion
!    not:      int, float
```

Binary Operators

```
+    addition: int, float, point2, point3, quaternion
-    subtraction: int, float, point2, point3, quaternion
+    concatenation: string
*    multiplication: int, float, point2/3*float, float*point2/3,
       quaternion*float float*quaternion, quaternion*quaternion
/    division: int, float
%    mod: int
<<   left shift: int
>>   signed right shift: int
<  ==  <=    comparisons: int, float, string;
>  !=  >=    returns int with value of 1 or 0
&    bit-wise AND: int
^    bit-wise XOR: int
|    bit-wise OR: int
&&   logical AND: int, float; returns int 1 if both operands are non-zero
||   logical OR: int, float; returns int 1 if either operand is non-zero
```

```
++  --      prefix and postfix increment and decrement: int, float.
            returns an expression, not an l-value.
```

Statements

Expression

An **expression** is the simplest statement. Normally an expression statement will call a function that has some kind of a side effect such as setting an Objects location. Expression statements that don't have side effects such as `$i + 1;` are allowed but aren't very useful.

Assignment

An **assignment** statement sets the value of a variable to value of an expression.

```
$index = 0;    $total = $part1 + $part2*1.5;
```

Numeric and string types simply copy the value of the expression to the variable:

```
int, float, string, point2, point3, quaternion, float4x4
```

Complex types refer to objects within Anim8or. They are actually handles to the objects, not the actual objects. Assigning a value to a variable of these types doesn't change the Anim8or project but sets the object referenced by that variable. Member functions in these types are used to alter them.

```
material, meshdata, tridata, shape, object, figure, sequence,
scene, project
```

Variable of type file cannot be assigned.

Compound Statement

A **compound** statement is a list of zero or more statements enclosed in curly braces "{}". They may be used anywhere a statement may be used:

```
{
    $shape.loc.x = sqrt($val);
    $ii = $ii + 1;
}
```

If Statement

An **if** statement evaluates a control expression. If the value is non-zero then the <then-statement> is executed next otherwisre the <else-statement> is executed if it is present. The syntax is the same as C's:

```
if (<expr>)
    <then-statement>
else
    <else-statement>
```

As in C the "else" statement is optional:

```
if (<expr>)
    <then-statement>
```

While Statement

A **while** statement evaluates a control expression. If the value of <expr> is non-zero then it executes its subordinate <statement> and reevaluates the control expression. This continues until the expression evaluates to zero. The syntax is the same as in C:

```
while (<expr>)
    <statement>
```

Do While Statement

A **do while** statement begins by executing the body of the loop. It then evaluates a control expression. If the value of <expr> is non-zero then it executes its subordinate <statement> again. This continues until the expression evaluates to zero. The syntax is the same as in C:

```
do
    <statement>
while (<expr>)
```

For Statement

There are two forms of the **for** statement. The first is similar to that of C. It evaluates <expr1> as an initialization. Then it evaluates <expr2>. If the value is non-zero the main statement is executed, followed by <expr3>. <expr2> is then reevaluated and the process repeats until it is zero.

```
for ( <expr1>, <expr2>, <expr3> )
    <statement>
```

ASL For Statement

An alternate form of a **for** statement is supported to be compability with previous versions of Anim8or. It evaluates several control expressions and then executes its subordinate statement a number of times based on those values. These for statements do not have the same semantics as those in C. The <step> expression is optional:

```
for <var> = <init> to <limit> do
    <statement>

for <var> = <init> to <limit> step <step> do
    <statement>
```

<var> can be either an int or float. The value of <init> is assigned to <var> and the values of <limit> and <step> expressions are cast to the type of <var> and saved. If <step> is not present a value of 1 or 1.0 is used. Then the value of <var> is compared to <limit>. If <step> is greater than or equal to zero and <var> is less than or equal to <limit>, or if <step> is less than zero and <var> is greater than or equal to <limit> the <statement> is executed after which <step> is added to <var>. This procedure repeats until the comparison fails.

Break Statement

A **break** statement exits from the innermost enclosing `for` or `while` loop.

```
break;
```

Continue Statement

A **continue** statement finished executing the body of the innermost enclosing `for` or `while` loop and starts the next iteration.

```
continue;
```

Return Statement

A **return** statement exits from a user defined function. The type of <expr> must be assignment compatible with the type of the function. If the function returns **void** then there should be not return value. The syntax is the same as in C:

```
return <expr>
```

Special Type Details

The special ASL types are described here.

Project Type

```
struct project {
    object curObject;     // The current Object
    figure curFigure;     // The current Figure
    sequence curSequence; // The current Sequence
    scene curScene;       // The current Scene
    string GetName(void);      // Project name
    string GetFileName(void);  // Project file name
    string GetDirectory(int kind);
    int MarkMaterials(int val);
    int MarkObjects(int val);
    int MarkFigures(int val);
    int MarkSequences(int val);
    int MarkScenes(int val);
    material NewMaterial(string name);
    int GetNumMaterials(void);
    material GetMaterial(int index);
    texture NewTexture(string name);
    texture NewCubeMapTexture(string name);
    int GetNumTextures(void);
```

```
    texture GetTexture(int index);
    int GetTextureIndex(texture texVar);
    texture LookupTexture(string name);
    string GetDirectory(int index);
    int GetNumObjects(void);
    object GetObject(int index);
    int GetNumFigures(void);
    figure GetFfigure(int index);
    int GetNumSequences(void);
    sequence GetSequence(int index);
    int GetNumScenes(void);
    scene GetScene(int index);
};
```

The Mark() functions set the mark member of all Materials, Objects, Figures, Sequences and Scenes, respectively, in a project to a value of 1 if val != 0, otherwise to 0. Mark() is useful for keeping track of whether a particular data item has been seen before when a script is exporting or altering a project. This is done by first clearing all marks with Mark(). The components of a project can then be visited in any order. If a component's mark member is zero it is set to 1 and the component is processed. If mark is already 1 that component can be skipped.

NewMaterial() adds a new global material to the project and returns it. If one already exists with the same name it doesn't add anything and returns NULL. GetNumMaterials() returns the number of global materials in a project. GetMaterial() returns a specific material. Materials are numbered from 0 to GetNumMaterials() - 1.

NewTexture() adds a new texture object to a project with the given name and returns the texture. NewCubeMapTexture() adds a new cube map texture. If a texture with that name already exists no new texture is created and NULL is returned. GetTexture() returns the texture numbered index, and GetTextureIndex returns the texture index corresponding to its parameter. If there is no such texture they return NULL and -1, respectively. LookupTexture() returns the texture with the given name or NULL if no such texture exists.

The function GetDirectory() returns the name of one of the target directories that Anim8or maintains. These are settable in the **File→Configure** dialog. The names shown below are used to access the different directories. An invalid value returns an empty string.

```
enum {
    DIRECTORY_PROJECT,
    DIRECTORY_TEXTURE,
    DIRECTORY_IMPORT,
    DIRECTORY_IMAGE,
    DIRECTORY_SCRIPT
};
```

GUI Type

```
struct GUI {
    int Xenabled;     // A value of 1 means the axis is
    int Yenabled;     // enabled and 0 disabled
    int Zenabled;
};
```

The GUI type is used only for a predefined variable GUI. It gives scripts access to certain settings in Anim8or's interface. Unless stated otherwise you can read and assign its members from within a script. Invalid or out of range values are silently clamped to a valid value or ignored.

Note: The GUI variable is not accessible from invariant scripts such as controller, parameteric plug-in or export plug-in scripts. Such scripts must return consistent results.

Material Type

```
struct material {
    string name;          // The current Object
    int Marked;           // 1 if Marked, 0 if not Marked
    int TwoSided;         // 1 if material is two sided, else 0
    int Toon;             // "toon" material - helps toon rendering

    // Attributes:

    int GetNumAttributes(void);
    attribute GetAttribute(int index);  // Return attribute number "index"
```

```
    attribute LookupAttribute(string name); // Return attribute "name" if
                                            // it exists, else 0.
    attribute NewAttribute(sring name); // Create a new material attribute,
                                        // Return existing one if one exists
                                        // with the same name

    // Surface properties:

    int LockAmbientDiffuse;
    float Ka;             // Ambient factor
    float Kd;             // Diffuse factor
    float Ks;             // Specular factor
    float Ke;             // Emissive factor
    float alpha;          // Transparency 0->1, 1 = opaque
    float roughness;      // Specular roughness factor
    float brilliance;     // Brilliance factor
    point3 ambient;       // ambient color
    point3 diffuse;       // diffuse color
    point3 specular;      // specular color
    point3 emissive;      // emissive color

    // Textures:

    texture GetTexture(int kind);
    int SetTexture(int kind, texture tex);
    int GetBlendMode(int kind);
    int SetBlendMode(int kind, int mode);
    int GetAlphaMode(int kind);
    int SetAlphaMode(int kind, int mode);
    int GetPercent(int kind);
    int SetPercent(int kind, int percent);

    // Back Surface Properties:
    //   Note: for single sided materials these act on the front surface.

    int LockAmbientDiffuse_back;
    int Toon_back;         // "toon" material - helps toon rendering
    float Ka_back;         // Ambient factor
    float Kd_back;         // Diffuse factor
    float Ks_back;         // Specular factor
    float Ke_back;         // Emissive factor
    float alpha_back;      // Transparency 0->1, 1 = opaque
    float roughness_back;  // Specular roughness factor
    float brilliance_back; // Brilliance factor
    point3 ambient_back;   // ambient color
    point3 diffuse_back;   // diffuse color
    point3 specular_back;  // specular color
    point3 emissive_back;  // emissive color

    // Backside textures:

    texture GetTexture_back(int kind);
    int SetTexture_back(int kind, texture tex);
    int GetBlendMode_back(int kind);
    int SetBlendMode_back(int kind, int mode);
    int GetAlphaMode_back(int kind);
    int SetAlphaMode_back(int kind, int mode);
    int GetPercent_back(int kind);
    int SetPercent_back(int kind, int percent);
};
```

The GetTexture() function returns the texture for the specifid kind if the material has one defined. Otherwise it returns NULL. Similarly the SetTexture() function sets the texture for the given kind. If tex is NULL then any currently defined texture of that kind is removed.

The following predefined names are accepted for texture kinds:

```
enum {
    TEXTURE_AMBIENT,
    TEXTURE_DIFFUSE,
    TEXTURE_SPECULAR,
    TEXTURE_EMISSIVE,
```

```
    TEXTURE_TRANSPARENCY,
    TEXTURE_BUMPMAP,
    TEXTURE_ENVIRONMENT,
    TEXTURE_NORMAL_MAP,
};
```

These constants are integers and can be used in expressions like any integer constant. The actual values may change in future releases so be sure to use these names in your scripts instead of relying on specific integer values when calling GetTextureName().

The functions SetBlendMode() and GetBlendMode() set and return a particular texture's blend mode. Similarly the functions SetAlphaMode() and GetAlphaMode() set and return a particular texture's alpha mode. The following enums correspond to the named modes.

```
enum BlendMode {
    BLEND_MODE_DECAL = 0,
    BLEND_MODE_DARKEN = 1,
    BLEND_MODE_LIGHTEN = 2,
};

enum AlphaMode {
    ALPHA_MODE_NONE = 0,
    ALPHA_MODE_LAYER = 1,
    ALPHA_MODE_FINAL = 2,
};
```

Texture Type

```
struct texture {
    string name;       // texture's name
    int Marked;        // Marked flag, 1 or 0
    const int CubeMap; // 1 if a Cube Map, else 0
    int InvertImage;   // invert image flag, 1 or 0
    string GetFileName(void);
    string GetCubeMapFileName(int face);
    int SetFileName(string name);
    int SetCubeMapFileName(string name, int face);
};
```

Assigning to name changes the name of a texture and all references to it in a material. If another texture already has that name then the assignment is not made and the texture retains its existing name.

GetFileName() and SetFileName() return and set the name of the file used for the texture image. If the texture is a cube map then the functions GetCubeMapFileName() and SetCubeMapFileName() must be used instead. The additional parameter face specifies which faces value to use. Out of range values return null strings. The following predefined names can be used for the faces:

```
enum CubeMapFace {
    FACE_POS_X = 0,
    FACE_NEG_X = 1,
    FACE_POS_Y = 2,
    FACE_NEG_Y = 3,
    FACE_POS_Z = 4,
    FACE_NEG_Z = 5,
    FACE_RIGHT = 0,
    FACE_LEFT = 1,
    FACE_UP = 2,
    FACE_DOWN = 3,
    FACE_FRONT = 4,
    FACE_BEHIND = 5,
};
```

Attribute Type

```
struct attribute {
    string GetName(void);         // Return attributes name
    int GetType(void);            // Returns attributes type (from table below)
    int GetBoolValue(void);       // Value of a Boolean attribute
    int GetIntValue(void);        // Value of an integer attribute
    float GetFloatValue(void);  // Value of a floating point attribute
    point3 GetPoint3Value(void); // Value of a point3 attribute
    quaternion GetQuaternionValue(void); // Value of a quaternion attribute
```

```
    string GetStringValue(void); // Value of a string attribute
};
```

Calling a GetXXXValue() function for the a type other than the attributes actual type returns 0. You should call GetType() prior to querrying an attributes value to verify that it has the expected type. The following predefined names should be used for the types:

```
enum AttributeTypes {
    ATTR_UNKNOWN = 0,
    ATTR_INT = 1,
    ATTR_FLOAT = 2,
    ATTR_POINT2 = 3,
    ATTR_POINT3 = 4,
    ATTR_QUATERNION = 5,
    ATTR_STRING = 6,
};
```

Spline Type

```
struct spline {
    float GetLength(void);       // Return the length of the spline
    point3 Eval(float t);        // Value of spline at position t from the start
    point3 GetTangent(float t);  // Tangent (direction) of spline at t
    quaternion Orientation(float t); // Orientation of spline at t
};
```

You can use Eval() and Orientation() to sample the position and orientation of a spline. Use the function toFloat4x4() to convert the orientation into a transformation matrix.

Object Type

```
struct object {
    string name;       // object's name
    int Selected;      // Selected flag, 1 or 0
    int Marked;        // Marked flag, 1 or 0
    material NewMaterial(string name); // Define a new material
    int GetNumMaterials(void);
    material GetMaterial(int index);
    material LookupMaterial(string name);
    shape LookupShape(string name);
    int GetShapes(shape fshapes[]); // return array of top level shapes
    int GetNumAttributes(void);
    attribute GetAttribute(int index);
    attribute LookupAttribute(string name);
};
```

GetNumMaterials() returns the number of materials in an object and GetMaterial() returns the indexth material in the object. Materials are numbered from 0 to GetNumMaterials() - 1. LookUpMaterial() returns a material with the given name of there is one in the object, otherwise it returns NULL.

GetShapes() fills its parameter with all of the top level shapes in an Object. A single Group shape is returned for each top level group. Individual shapes in a Group can be accessed from this group with its GetShapes() member function. The return value is the number of shapes returned. LookupShape() returns the shape with the given name if one exists. Otherwise it returns NULL.

Shape Type

```
struct shape {
    point3 loc;        // location relative to parent
    quaternion orientation; // relative orientation
    point3 bboxLo;     // read-only lower extent in local coordinates
    point3 bboxHi;     // read-only upper extent in local coordinates
    string name;       // shape's name
    int Selected;      // Selected flag, 1 or 0
    int Marked;        // Marked flag, 1 or 0
    int GetKind(void); // return kind of shape
    shape ConvertToMesh(void);  // convert 3d shape to mesh
    shape ConvertToSubdivided(void);
        // convert 3d shape to subdivision shape
    float4x4 GetGlobalTransform(void); // Transform matrix
    float4x4 GetGlobalNormalTransform(void);
        // Transform matrix for normals
```

```
    meshdata GetMeshData(void); // return mesh information
    tridata GetTriangleData(void);
        // return triangular representation of shape
};
```

GetKind() returns the type of shape it represents. The values returned are:

```
enum {
    SHAPE_KIND_UNKNOWN,
    SHAPE_KIND_SPHERE,
    SHAPE_KIND_RECT_SOLID,
    SHAPE_KIND_MESH,
    SHAPE_KIND_CYLINDER,
    SHAPE_KIND_PATH,
    SHAPE_KIND_TEXT,
    SHAPE_KIND_MODIFIER,
    SHAPE_KIND_SUBDIVISION,
    SHAPE_KIND_IMAGE,
    SHAPE_KIND_PARAM_PLUGIN,
    SHAPE_KIND_GROUP,
    SHAPE_KIND_NAMED,
};
```

SHAPE_KIND_NAMED is used to refer to Objects from a Figure or a Scene.

GetMeshData() returns a structure with the description of the faces and materials defined by the shape. The meshdata structure holds the basic data used by Anim8or for a Mesh. Other shapes such as parameteric shapes create meshdata structures for Anim8or when it needs to display them. Each point is uniquely stored in this format. Points used in multiple faces may use different normals or texture coordinates.

GetTriangleData() returns a tridata structure. This is a simplified representation of what is stored in a meshdata struct. It consists of triangles built from uniforms arrays of geometry, texture coordinates, normals, etc. Points from a shape that are used in multiple faces with different normals or texture coordinates are duplicated. This representation is suitable for direct rendering as vertex arrays in OpenGL or D3D.

GetGlobalTransform() returns a matrix that transforms the points in a tridata variable to their location in the world view.

The following types are subclasses of the shape type. They have all of the members that shape has plus the additional ones shown. If you are unsure about it, you should check that a shape is the appropriate kind with GetKind() before accessing these specialized members as it will cause an error.

Cube Type

```
struct cube : shape {
    float xsize;       // x dimension
    float ysize;       // y dimension
    float zsize;       // z dimension
    int xdivisions;
    int ydivisions;
    int zdivisions;
};
```

Variable members like xsize can be both read and assigned. If assigned an out of range value it is silently clamped to the supported range.

Sphere Type

```
struct sphere : shape {
    float diameter;
    int lat;           // number of faces in latitude
    int lon;           // and longitude
};
```

Cylinder Type

```
struct cylinder : shape {
    float length;
    float startdiameter;
    float enddiameter;
    int lat;           // number of faces in latitude
    int lon;           // and longitude
```

```
    int CapStart;      // 1 = cap start end, 0 = don't
    int CapEnd;        // 1 = cap final end, 0 = don't
};
```

Mesh and Meshdata Types

The mesh and meshdata types are similar but not entirely equivalent. **mesh** is a subclass of shape which contains a mesh or subdivision shape. It is entirely editable and can have points, textue coordinates, materials etc. added, deleted or modified.

The **meshdata** type, on the other hand, is a read-only copy of an arbitrary 3D shape. It is used in scripts for exporting shapes and internally by Anim8or for drawing. Parameteric shapes such as spheres create meshdata objects when Anim8or needs to draw them, for example.

There are some functions common to the two types that return internal values, and some functions found only in the mesh type that can modify it. Finally there are some functions found in both types that select and mark individual points, edges and faces that are useful for recording which parts have been processed but don't really change anything in them.

Mesh Members

The following functions in the mesh type are used to add, delete and change specific values:

```
struct mesh : shape {
    float smoothangle;
    int Open(void);      // Open a mesh for editing
    int Close(void);     // Finish editing a mesh
    int GetIsNew(void);  // True when a plug-in is first built

// Adding points, texture coordinates, faces, etc.:

    int AddPoint(point3 loc);
    int AddTexCoord(point2 uv);
    int OpenFace(int matNo, int flags);
    int CloseFace(void);
    int VertexN(int index);
    int TexCoordN(int index);
    int AddMaterial(material mat);

// Mesh altering functions:

    int SetPoint (int index, point3 val);
    int SetTexCoord(int index, point2 val);
    int SetMaterial(int index, material mat);

// Point, edge and face value sets:

    int SetFacePointIndex(int faceIndex, int vtxIndex, int index);
    int SetFaceTexCoordIndex(int faceIndex, int vtxIndex, int index);
    int SetFaceMaterialIndex(int faceIndex, int matIndex);
    int SetEdgeSharpness(int edgeIndex, int val);

    int SetFaceHasTexCoords(int index, int val);

// Delete functions:

    int DeletePoint(int pointIndex);
    int DeleteEdge(int edgeIndex);
    int DeleteFace(int faceIndex);
    int DeleteSelectedPoints(void);
    int DeleteSelectedEdges(void);
    int DeleteSelectedFaces(void);
    int DeleteMarkedPoints(void);
    int DeleteMarkedEdges(void);
    int DeleteMarkedFaces(void);

// Data clean-up:

    int RemoveUnusedTexCoords(void);
    int RemoveUnusedNormals(void);
    int RemoveUnusedBinormals(void);
    int RemoveUnusedMaterials(void);
```

```
    int RemoveUnusedData(void);
};
```

Meshes and subdivision shapes can be edited to add new points, faces, etc. A call to Open() enables editing. Once editing is complete Close() should be called to flush any unsynchronized data to the actual shape. These two functions return 1 if they succeed and 0 if they fail. Attempting to open a mesh that is already open or a shape that isn't a mesh or subdivision object will fail. Similarly calling Close() on an unopened mesh will fail.

AddPoint() and AddTexCoord() add new vertices and texture coordinates to a mesh. The return value is an integer index that is used to refer to the new point or texture coordinate when adding new faces. AddMaterial() adds a new material to the Mesh and returns its index.

OpenFace() begins the definition of a new face. It returns the index of the newly created face. If an error occurs it returns 0 and no new face is added. The parameter matNo sets the face's material. It is a zero based index into the mesh's materials. The bit vector flags specifies the additional properties the face will have. A value of FACE_HAS_TEXCOORDS specifies that the face will have texture coordinates. All other bits are ignored.

GetIsNew() returns true the when a plug in mesh is first created, otherwise false. This allows a script to do one time initializations of materials and to preserve any material set by the user on subsequent calls.

VertexN() adds a new point to a face. index is the value returned by a call to AddPoint(). The return value is index if the call succeeds. If it fails the return value is -1. A call will fail if the value of index is not a valid point index for this mesh. VertexN() must be called after all other properties for a point have been set such as those set by a call to TexCoordN().

TexCoordN() sets the texture coordinate for the current point. index is the value returned by a call to AddTexCoord(). The return value is index if the call succeeds. If it fails the return value is -1. A call will fail if the value of index is not a valid texture coordinate index for this mesh.

CloseFace() must be called after a face is defined to complete the definition.

SetPoint(), SetTexCoord() and SetMaterial() change the values of the point, texture coordinate or material indicated by the index parameter. This has the effect of moving a point, changing a texture coordinate, and changing a material for all faces in that mesh that currently use that value. The functions like SetFacePointIndex() however change the index number for a particular point in face number faceIndex. Other faces that used the same original point are not altered.

SetEdgeSharpness() sets the subdivision sharpness for an edge in the mesh. The edge will stay creased through this number of subdivisions. The sharpness can have a value of 0 to 7 where 0 is smooth and 7 is maximally creased.

SetFaceHasTexcoords() sets or clears the use of texture coordinates by this face. It returns the previous setting.

DeletePoint(), DeleteEdge() and DeleteFace() delete the point, edge and face corresponding to the value of the parameter. They return 1 if it was deleted, else 0 if it doesn't exist. DeleteSelectedPoints/Edges/Faces() and DeleteMarkedPoints/Edges/Faces() delete and selected and marked component of that kind in a mesh. The return the number of things deleted. Note: after something is deleted the remaining components are renumbered. Thus if you have saves the index of a point in a variable it may no longer refer th the same point, or any valid point.

There are several functions that remove unused data from a mesh. They can reduce the amount of storage required for a mesh after its geometry has been edited. They don't delete any points or edges. What they do is remove unused auxiliary data that is not references by any geometry in the mesh such as texture coordinates or normals. RemoveUnusedTexCoords() removes unreferenced texture coordinates, for example. Note that RemoveUnusedMaterials() doesn't delete any materials from an object but it does remove unreferenced materials from a mesh's material table.

Note: Most mesh member functions cannot be applied to non-mesh shapes because the geometry of parameteric shapes in cannot be directly edited. However SetMaterial() can be applied to material index 0, the default material.

Mesh and Meshdata Members

The following functions in the mesh and meshdata types are used to access internal values:

```
struct mesh : shape {  and  struct meshdata {

// Metric queries:
```

```
    int GetNumPoints(void);
    int GetNumNormals(void);
    int GetNumTexCoords(void);
    int GetNumBinormals(void);
    int GetNumMaterials(void);
    int GetNumFaces(void);
    int GetNumEdges(void);

// Value queries:

    point3 GetPoint(int index);
    point3 GetNormal(int index);
    point2 GetTexCoord(int index);
    point3 GetBinormal(int index);
    material GetMaterial(int index);

// Face metric queries:

    int GetNumSides(int faceIndex);
    int GetFaceHasNormals(int faceIndex); add
    int GetFaceHasTexCoords(int faceIndex); add
    int GetFaceHasBinormals(int faceIndex); add

// Point, edge and face value queries:

    int GetFacePointIndex(int faceIndex, int vtxIndex);
    int GetFaceNormalIndex(int faceIndex, int vtxIndex);
    int GetFaceTexCoordIndex(int faceIndex, int vtxIndex);
    int GetFaceBinormalIndex(int faceIndex, int vtxIndex);
    int GetFaceMaterialIndex(int faceIndex);

    int GetEdgeIndex(int faceIndex, int index);
    int GetEdgeSharpness(int faceIndex);
    point3 GetEdgePoint0(int edgeIndex);
    point3 GetEdgePoint1(int edgeIndex);
    int GetEdgeIndex0(int edgeIndex);
    int GetEdgeIndex1(int edgeIndex);
};
```

Mesh metric queries return the number of items of that kind in the mesh. The corresponding data value queries get the value of a particular one. GetNumPoints() returns the number of points, for example, and GetPoint() returns a specific point. Points, normals, binormals, texture coordinates and materials are zero based arrays. Thus the valid index for a point is 0 to GetNumPoints()-1.

Note: Edges are stored in a one based array however so valid indices for the edges in a mesh are 1 to GetNumEdges(). There is no edge number 0.

Face data queries return the parameters for a particular point or edge of a face. The first parameter is the face index and the second parameter is the point or edge index. The second parameter in an index into the 0 based array of points or edges and should have a value of 0 to GetNumSides()-1. GetNumSides() returns the number of sides or edges.

GetNumEdges() returns the number of edges in a mesh. Edges are numbered from 1 to N. GetEdgeIndex() returns the index of a particular edge in a face. The value returned can be positive or negative. If it is negative the edge is reversed when the face is viewed with the vertices in a clockwise orientation. GetEdgePoint0() and GetEdgePoint1() return the point for the beginning and ending points for an edge. GetEdgeIndex0() and GetEdgeIndex1() return the point index for the beginning and ending points for an edge.

The GetFaceHas() functions return true (1) or false (0) depending on whether the face has that particular property or not. For example GetFaceHasNormals() returns true if normal data is defined for the face.

The face value query functions return negative values when any of their arguments are out of range with the exception of GetEdgeIndex which returns 0.

Mesh and Meshdata Selection and Marking Members

The following functions in the mesh and meshdata types are used to mark and select individual points, edges and faces:

```
// Deselect or demark all points, edges or faces.
//     Returns count of components cleared:

    int DeselectPoints(void);
    int DeselectEdges(void);
    int DeselectFaces(void);
    int DemarkPoints(void);
    int DemarkEdges(void);
    int DemarkFaces(void);

// Test individual points, edges or faces for mark or selection:

    int GetPointSelected(int index);
    int GetEdgeSelected(int index);
    int GetFaceSelected(int index);
    int GetPointMarked(int index);
    int GetEdgeMarked(int index);
    int GetFaceMarked(int index);

// Set or clear point, edge or face's selection or marking.
//     Returns previous value:

    int SetPointSelected(int index, int val);
    int SetEdgeSelected(int index, int val);
    int SetFaceSelected(int index, int val);
    int SetPointMarked(int index, int val);
    int SetEdgeMarked(int index, int val);
    int SetFaceMarked(int index, int val);
}
```

Each individual point, edge and face may be marked and selected. Selected parts are often visible in the working views. The selected property is the same thing that is set when you select individual points, edges and faces when editing a mesh. The marked property is a temporary setting that is never visible. It is used throughout Anim8or to keep track of what parts of a mesh have been examined or altered during internal operations. Scripts can use these same properties when they examine any modify meshes as well. The marked property is transitory after a script has run. The selected property is preserved for mesh shapes but not for meshdata objects since meshdata objects are copies of the geometry of a shape, not the actual geometry.

The GetXXXSelected() and GetXXXMarked() functions test whether a particular point, edge or face in a mesh is selected and marked, respectively. The corresponding SetXXXSelected() and SetXXXMarked() set the respective selected and marked properties and return the original values. Finally the DeselectXXX() and DemarkXXX() functions clear all of the selections and markings for their respective kind of components.

Together these functions are useful when processing a mesh for export or modification.

Tridata Type

```
struct tridata {

// Tridata metric queries:

    int GetNumPoints(void);
    int GetNumTriangles(void);
    int GetNumMaterials(void);

// Tridata value queries:

    point3 GetPoint(int index);
    point3 GetNormal(int index);
    point2 GetTexCoord(int index);
    int GetIndex(int index);
    int GetMatIndex(int index);
    material GetMaterial(int index);
};
```

The tridata type is basically a set of arrays of data representing the points in a mesh. GetNumPoints() returns the total number of unique points, normals and texture coordinates it has. GetNumTriangles() returns the number of triangles, and GetNumMaterials() the number of materials.

The data for a particular triangle is indicated by the result of calling GetIndex() with three consecutive values. The number of points is not necessarily three times the number of triangles since points can be used in multiple triangles. To find the data for triangle N call GetIndex with 3*N, 3*N+1 and 3*N+2 where 0 <= N < GetNumTriangles(). The return values are passed to GetPoint() to find the value of the three vertices. The following code shows reads the data for triangle N:

```
tridata $tdata;
int $N, $index1, $index2, $index3;
point3 $p1, $p2, $p3;

$index1 = $tdata.GetIndex($N*3);
$index2 = $tdata.GetIndex($N*3 + 1);
$index3 = $tdata.GetIndex($N*3 + 2);
$p1 = $tdata.GetPoint($N*3);
$p2 = $tdata.GetPoint($N*3 + 1);
$p3 = $tdata.GetPoint($N*3 + 2);
/* The values of $p1, $p2 and $p3 define triangle $N */
```

Float4x4 Type

```
struct float4x4 {
private:
    float mat[4][4];
        // Transform matrix - not accessible to scripts.
public:
    point3 Project(point3 p0); // Transform point by mat
};
```

Predefined Variables

There are several predefined variables in ASL.

```
project project;
```

The current project is kept in the variable project.

```
GUI GUI;
```

The graphical user interface settings are accessible through the variable GUI in non-invariant scripts.

```
float time;
```

The current Scene time in seconds is stored in time. Normally there are 24 frames per second but you can change this in a Scene's Environment settings. time has a value of 0.0 when not used in a Scene script. You cannot set the value of time by assigning to it. It is read only.

```
int frame;
```

The current Scene frame number is stored in frame. frame has a value of 0 when not used in a Scene script. You cannot set the value of frame by assigning to it. It is read only.

```
string version;
```

The version of Anim8or is available in version.

Functions

User Functions

As of v1.0 ASL supports user functions. The format is simialr to C functions:

```
<type> name ( <type> ident, <type> ident, ...)
{
    <statements>
}
```

There are several restrictions that apply to ASL functions as compared to C functions:

1. No forward declarations. Function headers must be followed by the definition in the form of a compound statement "{...}".

2. Recursion is not allowed. Anim8or doesn't check for self-recursion yet (a last minute bug). If you try it be prepared for chaos.

3. Functions must be declared before they are called.

4. The "main" function is called "$main". It is optional. When not present, any statements that aren't in a function are gathered into a defaule "main" function, no matter where they appear. Kind of whacky, I know, but this allows you to more easily add functions to existing scripts.

5. No array parameters. All parameters and function results are the predefined types: **int**, **string**, **float**, **point2**, **point3**, **quaternion**, **float4x4**, **shape**, **meshdata**, **spline**, **material**, **attribute**, **texture**, **object**, **figure**, **sequence**, **scene**, and **tridata**.

6. All parameters are passed by value. Remember: types that represent objects within Anim8or (shape, meshdata, material, etc.) are actually handles so any changes to the parameter within a function affect the same object as outside the function. Future plans: support for an **out** modifier to return values through parameters.

7. Function format parameter and local declarations are in a separate scope from global variables. Declarations within compound statements {} are still in the enclosing scope however. I hope to fix this soon as well but it could potentially break existing scripts. Let me know if you think this would be a big problem.

Here's a simple function that returns the absolute difference between two squared floats.

```
float $absDiffSquared(float $argA, float $argB)
{
    float $answer;

    $answer = $argA*$argA - $argB*$argB;
    if ($answer >= 0.0)
        return $answer;
    return -$answer;
}
```

System Functions

System functions are available for all script types. They return various parameters about the current state of the active project.

```
void refresh(void);                   // Refresh the screen
void view(string viewName);           // Change view
void render(void);                    // Render an image
void renderer(string rendererName);   // Set the renderer
```

After a script has run the screen is automatically updated to the current status. You can redraw the screen while a script is running to show its progress with the refresh() function.

You can change the view to any standard view with the view() function. The value of viewName must be on of the standard Anim8or views:

```
"front", "left", "right", "back", "top", "bottom", "ortho", "user1",
"user2", "user3", "user4", "perspective", "camera", "stereo", "all", "one"
```

"all" changes to four view mode, and "one" changes to a single window of the focus view.

render() renders the current focus view with the currently active renderer.

renderer() sets the renderer. The value of renderName must match one of the available renderers. Normally this will include the following:

```
"scanline", "OpenGL", "ART Ray Tracer"
```

The available list for a particular computer is shown in the **Render→Renderer** dialog.

Math Functions

The following integer functions are supported:

```
int abs(int val);                 // absolute value
int min(int a, int b);            // minimum
int max(int a, int b);            // maximum
int clamp(int val, int min, int max);
      // clamp val to  min <= val <= max
```

The following floating point functions are supported:

```
float abs(float val);             // absolute value
float min(float a, float b);      // minimum
float max(float a, float b);      // maximum
float clamp(float val, float min, float max);
      // clamp val to  min <= val <= max

float floor(float val);           // floor function
float ceil(float val);            // ceiling function
float fract(float val);           // val - floor(val)

float cos(float val);             // cosine
float sin(float val);             // sine
float log(float val);             // natural logarithm
float exp(float val);             // exp
float asin(float val);            // arc sine
float acos(float val);            // arc cosine
float sqrt(float val);            // square root
float tan(float val);             // tangent
float atan(float val);            // arc tangent
float log10(float val);           // logarithm base 10
float cosh(float val);            // hyperbolic cosine
float sinh(float val);            // hyperbolic sine
float tanh(float val);            // hyperbolic tangent

float pow(float val, float pow); // val raised to power pow
float atan(float a, float b);    // arc tangent of (a/b)

float lrp(float val, float a, float b);
      // linear interpolate between a and b:
      //     if (val < 0.0) return a;
      //     else if (val > 1.0) return b;
      //     else return a*(1.0 - val) + b*val;
```

The following vector functions are supported:

```
float length(point2 val);        // length of vector
float length(point3 val);        // length of vector
float length(quaternion val);    // length of quaternion
point2 normalize(point2 val);    // convert to unit length
point3 normalize(point3 val);    // convert to unit length
quaternion normalize(quaternion val); // to unit length
float dot(point3 a, point3 b);   // dot product of a and b
point3 cross(point3 a, point3 b); // cross product of a, b
```

The following special functions are supported:

```
point4 RPYtoQuaternion(float roll, float pitch, float yaw)
      // Compute the primary unit quaternion defined by
      // applying a roll , then a pitch, and finally a yaw
      // specified in degrees.
float4x4 toFloat4x4(quaternion val); // Convert a quaternion
      // into a transformation matrix
```

The following functions support pseudo random number sequences. The intent is to allow some variability but it a controlled manner for scripts that should be repeatable such as parameteric plug-ins and controllers. Thus the seed is always initialized to the same value at the start of a script. If you want to change the sequence for a parameteric plug in, set a new seed derived from a parameter. This way the script will produce consistent results.

```
int irand(void);           // 16b random value 0 to 65535
float frand(void);         // float random value -1.0 to 1.0
int randseed(int);         // set new seed and return current one
```

File Functions

The file type has several member functions. Files are currently supported in Export Plug-In scripts, Object Command scripts and Untyped scripts.

```
int file.open(string fname, string mode);
```

The open() function attempts to open a text file using the value of fname. Relative path names start from the current working directory. If mode has a value of "r" the file is open for reading a text file, "rb" for reading a binary file, or "w" it is opened for writing, and of

"a" it is opened for append a text file, which is writing at the end after anything already in the file.

Note: file input is only supported using Animor's built in **scanner**.

If the value of fname is "$console" then the console (DOS text window) is opened for writing.

If the function succeeds it returns a positive value which represents the file handle, otherwise it returns zero.

It is an error to attempt to open a file that is already open.

```
int file.close(void);
```

The close() function closes an open file.

If the function succeeds it returns true (1), otherwise it returns false (0).

It is an error to attempt to close a file that is not open.

```
int file.IsOpen(void);
```

Returns 1 if the file is currently open, otherwise 0.

```
void file.print(string format, ...);
```

The print() function formats a string based on its parameters and outputs it to the file. It is similar to C's printf function but with a restricted set of formatting options. The formats supported are d, e, f, g and s. A maximum of 9 parameters are allowed after the format.

print() does not return a value.

```
string file.GetName(void);
```

The GetName() function returns the full path name of an open file. If the file isn't open it returns an empty string.

```
string file.GetDir(void);
```

The GetDir() function returns the directory of an open file. If the file isn't open it returns an empty string.

```
string file.GetRoot(void);
```

The GetRoot() function returns the root name of an open file. If the file isn't open it returns an empty string.

```
string file.GetExt(void);
```

The GetExt() function returns the extension of an open file. If the file isn't open it returns an empty string.

Text Scanner

The scanner type can be used to read trxt files as a sequence of C-like tokens, automatically converting quoted strings and integer and floating point numbers into accessable forms.

```
struct scanner {
    int token;       // Current token kind.
    int intval;      // Integer value of current token if type is ICONST_TOKEN.
    float floatval;  // Floating point value if token type is RCONST_TOKEN.
    string strval;   // String value for type SCONST_TOKEN.
    string ident;    // String value for type IDENT_TOKEN.
    int initialize(file <input-file>); // Initialize scanner. <input-file> must be
                                       // an "input" file. A "text" attributes combines
                                       // CR/LF and "binary" keeps them as separate characte
                                       // Returns the first token type int file, and ERROR_T
                                       // on failure.
    int finalize(void);                // Close scanner and free associated memory.
                                       // The associated file is left open.
    int scan(void);                    // Scan the next token. Returns new token type.
    void EOFisToken(int <flag>);       // If <flag> is 0 then EOF is treated as white
                                       // space (default setting).
                                       // If <flag> is 1 then EOF returns EOF_TOKEN.
    void scanToEOL(void);              // Scan up to EOL or EOF. Scanned characters are
                                       // available in member strval.
    void DollarIsIdent(int <flag>)     // Treat dollar sign $ as character in identifiers.
};
```

Usage:

```
scanner $scan;
file $in;

$in.open($filename, "r");  // File can optionally be opened using a #file directive.
$scan.initialize($in);     // Initialize with open input file, binary or text.
                           // First token is pre scanned.
$scan.EOFisToken(1);       // Optional. 1 returns EOL as a token. 0 skips EOLs.
$scan.DollarIsIdent(1);    // Optional. 1 causes scanner to consider '$' a letter
                           //      when scanning identifiers, 0 to consider it a symbol.
do {
    if ($scan.token == TOKEN_ICONST) {
        $ivar = $scan.token;
    } else ...             // Process token stream
    $scan.scan();          // Scan the next token.
} until ($scan.token == TOKEN_EOF);
$scan.finalize();          // Free scanner memory. Does not close file.
```

Token values:

```
enum {
   ERROR_TOKEN,         // the scanner encountered an error
   EOF_TOKEN,           // end of file
   EOL_TOKEN,           // end of line
   IDENT_TOKEN,         // C-identifiers
   ICONST_TOKEN,        // Unsigned integer constant: decimal, hex or octal format
   RCONST_TOKEN,        // Unsigned floating point constant
   SCONST_TOKEN,        // String constant.
   WCONST_TOKEN,        // L"xxx" - wide character string. Not currently supported by ASL.
   LPAREN_TOKEN,        // (
   RPAREN_TOKEN,        // )
   LBRACE_TOKEN,        // {
   RBRACE_TOKEN,        // }
   LSQUARE_TOKEN,       // [
   RSQUARE_TOKEN,       // ]
   SEMI_TOKEN,          // ;
   COMMA_TOKEN,         // ,
   COLON_TOKEN,         // :
   HASH_TOKEN,          // #
   DOT_TOKEN,           // .
   LT_TOKEN,            // <
   GT_TOKEN,            // >
   LE_TOKEN,            // <=
   GE_TOKEN,            // >=
   EQ_TOKEN,            // =
   NE_TOKEN,            // !=
   EQEQ_TOKEN,          // ==
   AND_TOKEN,           // &
   XOR_TOKEN,           // ^
   OR_TOKEN,            // |
   ANDAND_TOKEN,        // &&
   OROR_TOKEN,          // ||
   EXCLAIM_TOKEN,       // !
   PLUS_TOKEN,          // +
   MINUS_TOKEN,         // -
   STAR_TOKEN,          // *
   SLASH_TOKEN,         // /
   PERCENT_TOKEN,       // %
   TILDA_TOKEN,         // ~
   PLUSPLUS_TOKEN,      // ++
   MINUSMINUS_TOKEN,    // --
   LSHIFT_TOKEN,        // <<
   RSHIFT_TOKEN,        // >>
   BACKSLASH_TOKEN,     // \
   CONCAT_TOKEN,        // ||
   DOLLAR_TOKEN,        // $
}
```

Miscellaneous Functions

ASL scripts also include this function:

```
string PrintToString(string format, ...);
```

The PrintToString() function formats a string based on its parameters and returns it as a string variable. It is similar to C's sprintf function but with the same restricted set of formatting options as printf. The formats supported are d, e, f, g and s. A maximum of 9 parameters are allowed after the format.

Shape Creation Functions

ASL scripts can define new basic shapes with the following functions:

```
shape sphere(float diameter);
shape sphere(float diameter, int long);
shape sphere(float diameter, int long, int lat);
shape cube(float size);
shape cube(float size, int divisions);
shape cylinder(float diameter);
shape cylinder(float diameter, float length);
shape mesh(void);
```

The cube() functions create a Cube with the same dimensions and number of divisions on all sides. If you want to set individual sides to different values you can assign those members in the cube directly:

```
shape $myCube;
$myCube = cube(10.5, 6);
$myCube.xsize = 20;
$myCube.xdivisions = 10;
```

The cylinder() functions create a Cylinder with the same dimensions at both ends and with the default number of divisions. You can assign different values by assigning the members directly:

```
shape $ myCylinder;
$myCylinder = cylinder(10, 50);
$myCylinder.enddiameter = 5;
$myCylinder.lon = 25;
$myCylinder.CapStart = 0;
$myCylinder.CapEnd = 1;
```

The mesh() function creates an empty Mesh. You can then add points and faces as needed with its member functions. See the description of the mesh type above for details. For example the following code creates a new Mesh with a single square face:

```
shape $square;
int $vtx[4], $tex[4], $i;
$square = mesh();     // Create a new mesh
$square.Open();       // Open for editing
$tex[0] = $square.AddTexCoord((0.0, 0.0));
$tex[1] = $square.AddTexCoord((0.0, 1.0));
$tex[2] = $square.AddTexCoord((1.0, 1.0));
$tex[3] = $square.AddTexCoord((1.0, 0.0));
$vtx[0] = $square.AddPoint(( 0.0,  0.0, 0.0));
$vtx[1] = $square.AddPoint(( 0.0, 10.0, 0.0));
$vtx[2] = $square.AddPoint((10.0, 10.0, 0.0));
$vtx[3] = $square.AddPoint((10.0,  0.0, 0.0));
$square.OpenFace(0, 4);  // Start a new face with tex cords
for $i = 0 to 3 do {
    $square.TexCoordN($tex[$i]);
    $square.VertexN($vtx[$i]);
}
$square.CloseFace();     // Finish face
$square.Close();         // Close mesh or data won't be saved
```

Script Types

ASL scripts are used for a variety of different tasks. Each task has different requirements and restrictions. Plug-in scripts, for example, use a file name passed in from Anim8or and need to use that name when they are running. ASL uses **directives** to tell Anim8or how to pass information to and from scripts and often to tell Anim8or what kind of script it is.

Directives

Directives are source lines that begin with the hash symbol #. They must appear before any other statements but can appear after comments. The first directive typically defines what kind of script is being run. Any following directives link parameters and data to the outside world. Directive specifics are described next.

#command Directive

A **command directive** is used for a script that can be run from the menu in a particular editor. They are added to the Scripts menu for easy access when Anim8or first starts. To do this they must have the file extension ".a8s" and reside in the "scripts" directory. The format of a command directive is:

```
#command("<editor>");
```

Currently only the Object editor supports command scripts so the argument to the directive must be "object" as shown in the following example:

```
#command("object");
```

#plugin Directive

A **plugin directive** defines one of several kinds of plug-ins. The general syntax is:

```
#plugin("<editor>", additional parameters);
```

The number of parameters needed depends on what kind of plug-in the script is. The alternates are:

```
#plugin("object", "mesh", "<title>");
```

This defines a Parameteric Mesh plug-in script. The <title> is used to identify the Mesh type to the user. It should be short and it is better if it is a single word such as "spring".

```
#plugin("object", "export", "<title>", "<extension>");
```

This plug-in variant if for an Object export script. Such scripts are added to the **Object→Export** menu and work like built in exports. The title is used to identify the export kind to the user. The <extension> is the file extension for the output file. Some examples are:

```
#plugin("object", "mesh", "spring");
#plugin("object", "export", "Wavefront", ".obj");
```

#parameter Directive

A **parameter directive** defines a value that the use can pass to the script from Anim8or. They are only allowed for Parameteric Mesh plug-ins. The syntax is:

```
#parameter("<name>", <type>, <default>, <min>, <max>, ...);
```

Anim8or uses <name> in dialogs to set the value. <type> can be int or float. The default, minimum and maximum allowed values for this parameter are next.

These can be followed by one or more optional values that specify how the parameter scales when the user uses the Scale or Non-uniform Scale tool in the Object editor. The allowed values are:

```
scale   - parameter scales with the Scale tool.
scale_x - parameter scales in x axis with non-uniform scale
scale_y - parameter scales in y axis with non-uniform scale
scale_z - parameter scales in z axis with non-uniform scale
```

Some example parameter directives are:

```
#parameter("sides", int, 6, 3, 16);
#parameter("diameter", float, 10.0, 0.01, 99, scale);
#parameter("offset", float, 20.0, 0.0, 99, scale, scale_x);
```

The parameter "offset" is scaled by both the Scale tool and by the x-scaling of the Non-Uniform Scale tool.

#return Directive

A **return directive** tells Anim8or where to find the result of a script that returns a value. The format is:

```
#return(<variable-name>);
```

The user variable must be declared in the script. For Export scripts the type must be int and be assigned a value of 1 if the export is successful, otherwise zero. For Parameteric Mesh

plug-ins the variable should of type shape and returns the model created by the script.

Return directives are only allowed in Parameteric Mesh and Export plug-in scripts.

#file Directive

A **file directive** associates a variable with a file that Anim8or prompts the user for before the script runs. The syntax is:

```
#file(<file-var>, <options>);
```

The options are:

```
"text"     - Process file in 'text' mode. Combine CR/LF characters into '\n'
"binary"   - Process characters individually.
"input"    - Open the file for reading.
"extension:<.ext>:<description>"  - Use <.ext> file extension
             in file open dialog, and <description> as description.
"title:<heading>"  - Use <heading> as the title for user dilog.
```

The variable <file-var> must be declared in the script and be of type file.

For **Object Export Plug-In** scripts the option "text" is required and the other options are prohibited.

```
#file($uservar, "text");
```

For **Object Command** and **Untyped Command** scripts "input" is required.

```
#file($uservar, "text", "input", "extension:.obj:Wavefront .obj file",
      "title:Select a Wavefront .obj File");
```

#button Directive

A **button directive** defines an image that Anim8or uses on a toolbar button. The syntax is:

```
#button(<width>, <height>, <num-colors>, <data> ...);
```

<width> and <height> are the dimensions of the image. <num-colors> is the number of colors. This must be 2. <data> is a comma separated list of 32 bit decimal or hexadecimal integer constants that define the image. The data is ordered by row starting at the top of the image. Each new row starts a new value.

Images with 2 colors are bit-masks. They use one bit per pixel. Zeros represent the background color and ones the foreground color.

Here is an example bitmap directive for a 17 pixel wide, 25 pixel high bitmap:

```
#button(17, 25, 2,
    0x00000fc6, 0x00007079, 0x00008009, 0x00010f89,
    0x00013871, 0x000107e1, 0x0000c003, 0x0000300d,
    0x00000ff9, 0x00007079, 0x00008009, 0x00010f89,
    0x00013871, 0x000107e1, 0x0000c003, 0x0000300d,
    0x00000ff9, 0x00007079, 0x00008009, 0x00010f89,
    0x00013871, 0x000107e1, 0x0000c002, 0x0000300c,
    0x00000ff0);
```

which defines this button: 

Specifics of Script Kinds

ASL has five different kinds of scripts. They are described next.

General Script

A **general script** can be run from the **Scripts→Run-Script-File** menu command. They normally do things like align the selected shapes, add new materials, or simple add some new shapes. If you give them a file extension of .a8s and they reside in the Script directory Anim8or will parse them when it loads and show any errors in the command window.

General Script Directives

General scripts cannot use any directives.

Command Script

A **command script** is a general script that is specifically built to run in a particular editor. They can do the same things that a general script can but if they use the .a8s file extension

and are in the Scripts directory they will be added to the Scripts menu for easy access.

Command Script Directives

Command scripts are identified by a #command directive at the start. They cannot use any other directives.

Here is an example script that adds a Sphere of diameter 20 to the current Object, sets the longitudinal and latitudinal divisions to 16 and the location to (100, 20, 0):

```
#command("object");
shape $mySphere;
$myShape = sphere(20);
$myShape.lat = 16;
$myShape.lon = 16;
$myShape.loc = (100, 20, 0);
```

Parameteric Mesh Plug-in Script

A **parameteric Mesh plug-in script** adds a new parameteric shape to Anim8or. Shapes of that type behave exactly like built-in shapes such as sphere and cube. They have a button in the left-hand toolbar and can be scaled, moved and rotated like other shapes.

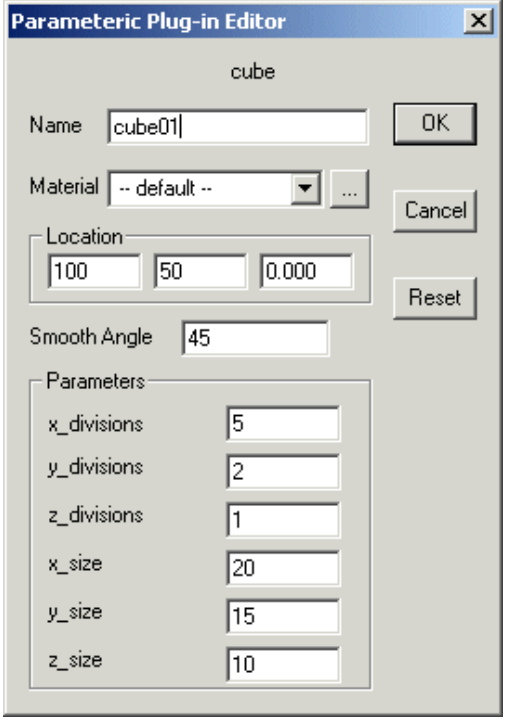
Parameteric Mesh Plug-In Script Directives

The first directive must be a mesh #plugin directive. The must be a #return directive for a variable of type shape. Normally there are one or more #parameter directives that define the parameters that the script uses to build the shape. Finally an optional #button directive defines the image for the toolbar button for this shape.

Here are the directives for a mesh plug-in for a cube:

```
#plugin("object", "mesh", "cube");
#parameter("x_divisions", int, 1, 1, 100);
#parameter("y_divisions", int, 1, 1, 100);
#parameter("z_divisions", int, 1, 1, 100);
#parameter("x_size", float, 10.0, 0.01, 1000, scale, scale_x);
#parameter("y_size", float, 10.0, 0.01, 1000, scale, scale_y);
#parameter("z_size", float, 10.0, 0.01, 1000, scale, scale_z);
#return($cube);
#button(24, 19, 2,
    0x0000ffff, 0x00010843, 0x00021085, 0x00042109,
    0x000ffff1, 0x00108431, 0x00210853, 0x00421095,
    0x00ffff19, 0x00842111, 0x00842131, 0x00842152,
    0x00842194, 0x00ffff18, 0x00842110, 0x00842120,
    0x00842140, 0x00842180, 0x00ffff00);
```

When a user double clicks on a parameteric mesh in the Object editor a dialog is shown where the values of these parameters can be edited. These directives would make a dialog similar to this:



The current value of a parameter is accessed form a script with the parameter function. The type of the value returned by parameter() depends on the type in the #parameter directive:

```
float parameter(string name);
int parameter(string name);
```

For example, the following code retrieves the values of x_size and x_divisions for the directives shown above:

```
float $xSize;
int $xDiv;
$xSize = parameter("x_size");
$xDiv = parameter("x_divisions");
```

Parameteric mesh plug-in scripts cannot define any new shapes. Instead an empty Mesh is defined by Anim8or prior to running the script and assigned to the variable specified in the #return directive. The script adds the vertices and faces necessary to build the shape's geometry.

Object Export Plug-in Script

An **Object export plug-in script** exports an Object to a file in a new format.

Object Export Plug-In Script Directives

The first directive must be an Object export #plugin directive. There must be a #file directive linking the export file to the script, and an integer #return directive for the script to report success or failure when it finishes.

Here is an example of the directives for an Object export plug-in script:

```
#plugin("object", "export", "Wavefront", ".obj");
#file($output, "text");
#return($result);
```

Object export plug-in scripts can open additional files if needed.

Controller Expression Script

A Controller expression script computes the value of a controller in a scene. Any controller can use a script instead of a spline curve to compute its value. The only requirement is that there must be an assignment to a pre-declared variable named after the controller. In the simplest form a controller expression is just an assignment statement. Here is a script for an element's size controller:

```
$scale = 1 + sin(time*PI*4)*0.5;
```

The size of the element oscillates between 0.5 and 1.5 every 2 seconds.

Controller scripts can be more complex that a simple assignment. They can declare variables, use control flow, etc. They don't have a history however and cannot store data between

frames.

Another useful property of Controller scripts is that they can refer to the value of other Controllers in Elements in the same Scene with the GetAttribute() functions:

```
int GetAttributeInt(string elName, string ctrlName);
float GetAttributeFloat(string elName, string ctrlName);
point3 GetAttributePoint3(string elName, string ctrlName);
quaternion GetAttributeQuaternion(string elName, string ctrlName);
```

The type of the Controller must match the type in the name of the GetAttribute() function or a runtime error will be generated. The parameters must be string constants, not variables or computed values.

Here is an example that sets an Element's position to 50 units above the position of an Element with the name "Blue Element":

```
point3 $bluePos;
$bluePos = GetAttributePoint3("Blue Element", "position");
$position = $bluePos + (0, 50, 0);
```

Calling a GetAttribute() function in a script introduces a dependency on the order in which controller values must be computed. This is similar to what happens when one Element is set to face another element. You have to be careful to not create a circular dependency or Anim8or won't be able to decide what values to use. Anim8or will warn you when this happens, however, so it shouldn't be a problem.

Debugging

ASL scripts can be tricky to debug. More help is coming in this area so please be patient. Currently you can trace line numbers of statements as they are executed, and use print() to write selected data to an output file, but that's about it.

Tracing Script Execution

The **Options→Debug→Trace-Script-Execution** command toggles tracing the line numbers of each statement executed in scripts. When enabled the line number for each statement is output to the script window before it is executed. You can also enable or disable tracing in a running script with the debug() function:

```
int debug(int level);
```

Setting the level to 1 or higher enables tracing during the currently running script. Setting it to 0 disables it. The initial value for a script is always copied from the global setting. If tracing is enabled it will be 1, if disabled it will be 0.

Tracing can produce a lot of output and slow Anim8or considerably for complex scripts.