File Formats

Version 6

Inside cover image created by Duncan Brinsmead.

ALIAS ■ 210 KING STREET EAST ■ TORONTO, CANADA M5A 1J7

Table of Contents

1 Maya ASCII file format

Developer File formats

About file formats

This documentation is directed at technical users with prior programming experience who want to do one of the following:

- edit Maya files
- write translators to/from Maya files

Note	.pda and .pdb file formats are documented in the dynExport command documentation.

Maya ASCII file format

Overview of Maya ASCII file format

Maya scene files define the geometry, lighting, animation, rendering, and other properties of a scene.

Maya scenes can be saved as binary or ASCII files. A Maya ASCII file can be easily edited.

If you have written a script using Maya's MEL programming language, you are already familiar with the Maya ASCII file format. A Maya ASCII file uses a tiny subset of the MEL language—in fact, of the hundreds of commands available in MEL, only eleven are used:

- file
- requires
- createNode
- setAttr
- addAttr
- parent
- connectAttr
- disconnectAttr
- select
- currentUnit
- fileInfo

These MEL commands are the only MEL commands that can be safely used in a Maya ASCII file.

If you are not familiar with MEL, you may want to look at the MEL documentation before continuing. See **"MEL"** in the *MEL and Expressions* book for details.

A statement in MEL consists of a *keyword*, followed by a series of *options* and *arguments*, and ends in a semicolon. A statement can span any number of lines in the file.

- A *keyword* is always the first word in a statement.
- An *option* provides more specific information to the statement.
- An *argument* further defines each option.

For the examples in this chapter, the following typefaces designate the keywords, options, and arguments:

Keywords and options are in bold-face type, such as **bump** or **-s**. The option is always preceded by a dash.

Arguments are in non-bold italic, such as *u* or *file.txb*. The names of the arguments are arbitrary labels. In your files, use the actual value or string.

Write file translators

To the Maya ASCII format

If you are writing a program to translate other kinds of files to Maya ASCII file format, the work is fairly straightforward. The bulk of what you need to do is find Maya node and attribute equivalences to the foreign data types. For this you will probably want to construct your own sample Maya files and refer to the document called *Node and Attribute Reference*.

Notes:

- Ensure that the first 6 characters in your file are "//Maya".
- Ensure that nothing is referenced until it is created.

From the Maya ASCII format

If you want to write a program to translate Maya ASCII files to other file formats, you have a difficult job ahead of you. To do the job properly, you would not only need to be able to read in the files, but also to read in the referenced files. Since MEL references can contain any arbitrary MEL code, you would either have to not support them, or write a full MEL interpreter.

An easier way to solve this problem is provided in the *Maya Developer's Tool Kit*. There you will find an example of a file translator plug-in under the MPxFileTranslator class (. Using the documentation there, and the example (called *lepTranslator*) as a basis, you can write a plug-in that will allow Maya to save files in the format you prefer.

It is also possible to write data out through MEL. Though this method is probably not appropriate for comprehensive, large-scale translators, it can be a quick and easy way to export relatively small and simple sets of ASCII data. Provided all of the data required is accessible through MEL, you can use the *fopen*, *fprint*, and *fclose* commands to write the data to a file.

Embedded comments

An embedded comment is text in a MEL file that is ignored when the file is read in.

If a line in the file contains two consecutive slashes ("//"), everything from there to the end of the line is considered to be a comment. This is commonly known as a "C++ style comment".

If a line in the file contains a slash followed by an asterisk ("/*"), everything from there on is considered a to be comment, until the next occurrence of an asterisk followed by a slash ("*/"). This is commonly known as a "C style comment".

Note	Although supported in MEL, C style comments are not generally used.

Organization of Maya data

To understand how Maya ASCII files work, you need to have some understanding of the way data is organized in Maya.

All the information in a scene is stored internally in *nodes*. A node is simply a block of data with a name. Different kinds of nodes have different kinds of data; the individual pieces of data are referred to as *attributes* of the node. All nodes of the same type have the same set of attributes. A node may also have additional "dynamic" attributes unique to itself.

Nodes in Maya are connected by two mechanisms. The first is *parenting*, which is used to group related geometry together. When a parent node is moved, all of its child nodes move with it. Only those nodes which represent geometry (such as curves and surfaces) and nodes that group these together (such as transforms) can be connected by parenting.

For example, a NURBS sphere in Maya may be represented by two nodes. One of these nodes, named "nurbsSphere1", is a type of node called "transform". Its attributes have information about its size and position, such as translation, scaling, and rotation. The other node, named "nurbsSphereShape1", contains information about the shape of the sphere, such as the exact positions of all of its control points. The transform node is the *parent* of the other node, so that you can move the sphere by setting the attributes of the transform.

The other mechanism that connects nodes together is called *attribute connections*. Any attribute of any node can be connected to any other, as long as the data types are compatible. When attributes are connected, changing the value of the source attribute will change the value of the destination attribute.

There are hundreds of different kinds of nodes in Maya, and more types can be added using plug-ins. Using these simple building blocks, Maya can represent elaborate models and animations.

For a complete description of all the different kinds of nodes that exist in Maya, and their attributes, see the *DG Node Reference* listings (on-line only).

Organization of Maya ASCII files

The ASCII files that Maya generates are organized into eight sections:

- Header
- (Non-Procedural) File references
- Requirements
- Units
- File references
- Nodes, attributes, and parenting
- Script nodes
- Disconnections
- Connections

The following describes each section. These sections must occur in the order specified for the file to load properly.

Header

The file header consists of a block of comments to help identify where and when the file was created. Like all comments, this block is ignored by the code that reads in a Maya file. There is one exception, however: the first six characters of the file must be "//Maya".

A typical Maya ASCII file header looks like:

```
//Maya ASCII 1.0 scene
//Name: solstice.ma
//Last modified: Sun, Dec 21, 97 10:18:26 AM
```

File references

The next section of the file specifies all the non-procedural references. That is, all the other Maya files that are being referred to by this one, if any. For each file that is referenced, there will be a single *file* command to read it in. All the objects in the referenced Maya file will be available in this file, but their names will be prefixed with a string, usually the file name. The syntax of the *file* command when used for referencing is:

`file -r -rpr` ***prefixString fileName***`;`

or

`file -r -ns` ***nameSpace fileName***

The **-r** option specifies that the file is to be referenced. The *-rpr* option specifies the string that will be prefixed to all the object names in the file. For complete information about the *file* command, see the *MEL Command Reference* online document.

Here is an example:

```
file -r -rpr "solar" "/u/sally/work/solar.ma";
```

If the file "solar" contained an object called "sun", that object would be accessible in this file using the name "solar_sun".

Defer loading file references

Use the *-dr* flag to defer loading file references.

Requirements

The next section specifies the requirements. This consists of a series of *requires* commands. This section of the file tells Maya what software is needed to load the file properly. Specifically, what version of Maya, and what plug-ins.

A statement in the Requirements section looks like this:

`requires` ***productName version***

This is an example of a typical requirements section:

```
requires maya "2.0";
requires specialPlugIn "1.2";
```

(For a full description of the *requires* command, see the *MEL Command Reference* online document.)

Units

This section of the file consists of a single *currentUnit* command, stating what units are used in this file. This setting will determine how all the numbers found in the file are interpreted.

Example:

```
currentUnit -l cm -a deg -t ntsc;
```

This example would set the current linear unit to centimeters (other options are millimeters, meters, kilometers, inches, feet, yards, and miles), the angular unit to degrees (other option is radians) and the time unit to NTSC. (For a full description of the *currentUnit* command and all of its options, see the *MEL Command Reference* online document.)

Important!Be aware that if you change default linear units from cm to feet/miles you may encounter unexpected results.

File Information

This next section consists of several lines providing information about the file. The first five of these are defined by Maya: the application name (Maya), productization (e.g. Complete), version, cut identifier (date and time), and operating system and version. If you have provided additional fileInfo commands specific to your file, they will also appear in this section.

Maya saves out the current values for the 5 predefined fileInfo statements each time you save your file. Any other fileInfo statements (or binary equivalents) found when loading a file, or issued during the Maya session, are preserved during a session and saved back out with the file.

Nodes, attributes, and parenting

This section of the file contains the bulk of the data.

This is where new nodes are created with the *createNode* command, and their attributes are set with the *setAttr* command.

Nodes from referenced files (and globally defined nodes) can also be selected here with the *select* command, and have their attributes set with the *setAttr* command.

New “dynamic” attributes may be added to a node with the *addAttr* command.

Finally, nodes can be parented to other nodes here with the *parent* command.

(For complete information on all of these commands, see the online *MEL Command Reference* document.)

The creation of a new node looks like this:

createNode ***nodeType*** **-n** ***nodeName***;

If this node is a node that can be parented (*i.e.*, it represents geometry or a group of geometry), and it has a parent node that has already been created, the parenting can also be specified in the command:

createNode ***nodeType*** **-n** ***nodeName*** **-p** ***parentNodeName***;

Some nodes (such as the default cameras) are common to every Maya scene. For these nodes, the *-s* option is specified. This tells Maya not to bother creating a new nodes if a node having the same name and parent) already exists. (This case occurs when a file is being referenced in.)

After the new node is created, it is automatically selected. The *createNode* command is then usually followed by a series of *setAttr* commands, to set the data in the node. Since the node is already selected, these commands only need to specify the attribute names and values.

The setAttr command looks like this:

setAttr ***attributeName value***;

or sometimes:

setAttr ***attributeName*** **-type** ***typeName value***;

Every attribute has a default value, so *setAttr* commands are only stored for those attributes whose value is not default. (Or when the value of an attribute from a referenced file is changed.) Here is an example definition, of a sphere:

```
createNode transform -n "sphere";
      setAttr ".s" -type "double3" 2.44 2.44 2.44;
      setAttr ".t" -type "double3" -6.96 0 6.9;
createNode nurbsSurface -n "sphereShape" -p "sphere";
      setAttr ".tw" yes;
      setAttr ".rtw" yes;
      setAttr ".ipo" no;
```

The pattern is similar when setting attributes of nodes that were created in a referenced file. Since these nodes have already been read in (and created), instead of using a *createNode* command, a *select* command is used. For example, say the file called "lunar" references another file called "solar", which contains an object called "sun". In the file called "lunar", the scale of this object is changed to 3. This is how that would look in the Maya ASCII file called "lunar".

```
select -ne solar_sun;
      setAttr ".s" -type "double3" 3.0 3.0 3.0;
```

It is also possible to add new (dynamic) attributes to nodes. This is done with the *addAttr* command, used similarly to the *setAttr* command. For this example, say you have created a sphere, and added a float-valued attribute to it called "squish", which can range from -1 to 1, and set that attribute to 0.3. When you save the file, the code will look like this:

```
createNode transform -n "sphere";
      addAttr -ci true -sn "squish" -ln "squish"
            -min -1 -max 1 -at "double";
      setAttr -k on ".squish";
      setAttr ".squish" 0.3;
// etc...
```

Not all of the parenting in the file can be done using the **-***p* flag on the *createNode* command. For example, nodes may need to be parented to other nodes in referenced files. All the remaining parenting relationships are established with the *parent* command, which looks like this:

```
parent childNodeName parentNodeName;
```

The *parent* command is also used to do instancing. (That is, when you want to have two nodes that share children). This is done using the **-***add* flag, like this:

```
parent -add childNodeName parentNodeName;
```

For the special case of instancing an object at the top level (also called the *world*), the **-w** flag is used in conjunction with the **-***add* flag.

```
parent -w -add childNodeName;
```

(A parent node name is not specified, because the parent is the *world*, which contains all things.)

Script nodes

Script nodes hold MEL code as part of a scene file. They are also set up to (possibly) execute after loading from a file, just before closing a file, or just before the node is deleted. The "before" scripts are executed when a file is loaded. If a file is closed or the node is deleted, the "after" script is executed. If a scene contains several script nodes, there is no guaranteed order of execution and the scripts should not depend on a specific execution sequence. See the MEL command documentation for *scriptNode* and the node documentation for *script* node for details on how to create script nodes.

Because MEL gives access to virtually everything you can do in Maya, the possibilities opened up by script nodes are endless. Specifically, script nodes can be useful for things such as scene cleanup or custom UI.

Disconnections

In files that contain file references, this next section of the Maya ASCII file breaks attribute connections among nodes from referenced files. This is done with a series of disconnectAttr commands which have the following syntax:

```
disconnectAttr sourceAttributeName destinationAttributeName;
```

For example:

```
disconnectAttr "sphere.tx" "cone.ry";
```

For complete information on the disconnectAttr command, see the *MEL Command Reference* online document.

Connections

The next section of the Maya ASCII file establishes the attribute connections among all the nodes that have been created and referenced. This is done with a series of *connectAttr* commands. (For complete information on the *connectAttr* command, see the *MEL Command Reference* online document.)

The format of these commands is this:

`connectAttr` ***`sourceAttributeName destinationAttributeName`***`;`

For example:

```
connectAttr "sphere.tx" "cone.ry";
connectAttr "sphere.squish" "sphere.sz";
```

Limitations to editing Maya ASCII files

Maya ASCII files support only the statements described above.

Note	It is possible to edit a Maya ASCII file and add MEL commands to it. However, we advise you against doing this.

If you do decide to add custom MEL code directly to a Maya ASCII file, keep the following in mind:

- If you edit a Maya ASCII file and add arbitrary MEL commands to it, these commands will not be saved out after you read in the file.

  Instead, they will be replaced by the explicit descriptions of the nodes, attributes, and connections that resulted from executing the commands. If you wish to add commands that will stay the way they

are, you must put them in a file with a .mel extension, and reference that file (using Create Reference... in the File menu, or the *file -r* command.)

- There are other limitations when you are adding arbitrary MEL commands directly into a Maya ASCII file.

  There is no guarantee that all commands will execute successfully. For example, for performance reasons we suspend all *undo* operations during loading of Maya ASCII and binary files. Some commands rely on undo capability internally to perform their work; these commands will only work from within .mel files (since loading of such files does not turn off the undo capability).

2 Animation curves

Developer File formats

Animation curves

Overview of animation curves

Maya provides a plug-in for importing and exporting animation to a separate file. This facilitates copying animation from one scene to another. For example, you could animate a scene with a lightweight representation of a character and then transfer the animation to the final rendered version of the character in another file.

Import anim curves

Internally, the plug-in imports and exports to the API clipboard. As part of the import and export process, the plug-in executes the copyKey or pasteKey MEL commands to transfer the anim curves from the API clipboard to the Maya objects. Using the plug-in will not change the contents of the Maya animation clipboard.

Import lets you set the same options as Edit > Keys > Paste keys which are used when the animation is applied to the selected object(s).

To import anim curves

1. Make sure the animImportExport plug-in is loaded.
2. Select the Maya objects to be animated.
3. Select File > Import.
4. Select the .anim file to import.

Export anim curves

Exporting anim curves uses the copyKey MEL command to copy the keys to the API clipboard and then exports the clipboard contents to the anim file. The entire operation is contained in the file export.

Note	Since MEL commands are used, any limitations they have are also limitations of the plug-in. For example, if an anim curve is upstream from an attribute, but there are blend nodes in-between the attribute and the anim curve, the export command will not work.

The File > Export Selection options window lets you set the same options as Edit > Keys > Copy keys which are used to export the animation from the selected object(s).

When an anim file is exported, any information previously on the API clipboard is replaced with the new anim curves after the file export.

To export anim curves

1. Load the animImportExport plug-in.
2. Select the Maya objects to export animation from.
3. Select File >Export All or Export Selection.
4. Set the file type to animExport.
5. Enter or select the .anim file to export.

Anim File Format

Maya provides a file format that lets you export and edit anim curves.The file format is defined in a form that can be easily read and written by external applications without having to use the Maya API.

```
Anim File Format

//    A description of the anim file format.
//    August 16, 1998
//
//    The .anim file format (version 1.0):
//        // and # are both valid comment characters.
//
//        All of the lines in the file that do not contain curly braces
//        ('{' or '}') should end with a ';' After the ';' character, start
//        a new line.
//
//        The keywords and data are whitespace delimited.
//
//    Version 1.1 changes:
//    April 20, 1999
//        new weighted keyword for animData
//        new breakdown flag for keys
//

//  The version of the file format. This is a required line.
//
animVersion string

//  The Maya version. The string is the value of MGlobal::mayaVersion()
mayaVersion string

//    The following two lines are optional. If they are not included,
```

```
//    the clipboard is set to the range defined by the anim curves
//    contained in the clipboard.
//
//    These are used by anim curves that have time inputs.
//
startTime [float]            //  The starting frame for the clipboard.
endTime [float]              //    The ending frame for the clipboard.

//    The following two lines are optional. If they are not included,
//    the clipboard is set to the range defined by the anim curves
//    contained in the clipboard.
//
//    These are used by anim curves with unitless inputs.
//
startUnitless [float]        //    The starting value for for the clipboard.
endUnitless [float]          //    The ending value for the clipboard.

//    The following three keywords are used to set the units for the file.
//    Each anim curve may have its own units, but these are the default
//    units if the anim curve units are not given (see the animData section).
//
//    If the units are not given, then the ui units are used.
//
timeUnit [game|film|pal|ntsc|show|palf|ntscf|hour|min|sec|millisec]
linearUnit [mm|cm|m|km|in|ft|yd|mi]
angularUnit [rad|deg|min|sec]

//    All of the keywords described above can only be in the header section
//    of the file. As soon as anim curve information is encountered, the
//    header section is completed and the body of the file is begun.
//

//    The string is the name of the attribute the anim curve is connected to.
//    The next three ints are the row, child, and attr values used by the
//    clipboard. See the documentation for MAnimCurveClipboard for more
//    information.
//
//    If the anim curve is not connected to any attributes, the string
//    is not needed, but the following ints should be 0 0 0.
anim [string] [int] [int] [int]

//    The second form of the anim line has three strings which list what
//    the anim curve was connected to.
//
//    The strings are: the full attribute name, the leaf attribute name,
//    and the node name. The row, child, and attr ints are still required.
//
anim [string] [string] [string] [int] [int] [int]

//    The third and final form of the anim line is used for clipboard
```

```
//    place holder objects. These are used to skip node which do not
//    contain any anim curve data, but are positioned in a hierarchy
//    with nodes that have attached anim curves.
//
//    In this case, the string is the node name and the three ints are the
//    same as the other two formats.
//
anim [string] [int] [int] [int]

//    The animData must follow a line with a valid anim statement.
//
animData {
    //    The input type of the anim curve. Defaults to time.
    input [time|unitless]

    //    The output type of the anim curve. Defaults to linear.
    output [time|linear|angular|unitless]

    //    Whether or not the anim curve has weighted tangents.  Defaults to
false.
    //    This is available with animVersion >= 1.1
    weighted [1|0]

    //    The unit of the anim curve input, if it is a time input.
    //    The units default to the time units specified in the file header.
    inputUnit [game|film|pal|ntsc|show|palf|ntscf|hour|min|sec|millisec]

    //    The unit of the anim curve output. The output unit should match
    //    the output type of the curve. These default to the units specified
    //    in the header.
    outputUnit [game|film|pal|ntsc|show|palf|ntscf|hour|min|sec|millisec]
    outputUnit [mm|cm|m|km|in|ft|yd|mi]
    outputUnit [rad|deg|min|sec]

    //    The unit of the tangent angles, if there are any fixed tangents.
    //    The units default to the angular units specified in the file header.
    tangentAngleUnit [rad|deg|min|sec]

    //    The pre-infinity type. Defaults to constant.
    preInfinity [constant|linear|cycle|cycleRelative|oscillate]

    //    The post-infinity type. Defaults to constant.
    postInfinity [constant|linear|cycle|cycleRelative|oscillate]

    //    The start of the actual keyframe data. Each key is a row in the
    //    braced section.
    keys {
        [float] [float] [in tan] [out tan] [tan locked] [weight locked]
        // animVersion 1.1 adds breakdown information
```

```
        [float] [float] [in tan] [out tan] [tan locked] [weight locked]
[breakdown]
        .
        .
        .
//    The first two values are the input and output values in the
        //    units defined by the inputUnit and outputUnit keywords.
        //    The in and out tangents should be valid tangent types.
        //    These are followed by three int values for tangent locking,
        //    weight locking and the breakdown flag. If they are 0, the values
        //    are unlocked, or not a breakdown, otherwise they are locked.
        //
        //    If either, or both, or the tangents are fixed, then additional
        //    information is needed: a tangent angle and weight.
        //    These two values, per fixed tangent, are added at the end of
        //    the above line.
        //
        //    For example:
        //    1.0 2.0 fixed linear 1 1 0 62.345 0.04;
        //
        //    In the above case, 62.345 is the tangent angle for the first
        //    tangent and the tangent weight is 0.04.
        //
        //    An example with two fixed tangents:
        //    1.0 2.0 fixed fixed 1 1 0 62.345 0.04 45.3 0.023;
    }
}
The pattern of an anim line followed by animData should be used until all of
the anim curves are described.
The following example is an animated joint chain consisting of 4 joints. The
first three joints are animated and the fourth joint is not animated.
animVersion 1.1;
mayaVersion 2.0;
timeUnit ntsc;
linearUnit cm;
angularUnit deg;
startTime 1;
endTime 30;
anim rotate.rotateX rotateX joint1 0 1 0;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 linear linear 1 1 0;
    30 0 linear linear 1 1 0;
  }
}
```

```
anim rotate.rotateY rotateY joint1 0 1 1;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 linear linear 1 1 0;
    30 0 linear linear 1 1 0;
  }
}
anim rotate.rotateZ rotateZ joint1 0 1 2;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 spline spline 1 1 0;
    10 -16.774359 spline spline 1 1 0;
    15 -1.6493069 spline spline 1 1 0;
    22 -3.064691 spline spline 1 1 0;
    30 0 spline spline 1 1 0;
  }
}
anim rotate.rotateX rotateX joint2 1 1 0;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 linear linear 1 1 0;
    30 0 linear linear 1 1 0;
  }
}
anim rotate.rotateZ rotateZ joint2 1 1 1;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 spline spline 1 1 0;
    10 60.962438 spline spline 1 1 0;
    15 106.06094 spline spline 1 1 0;
```

```
    22 33.259896 spline spline 1 1 0;
    30 0 spline spline 1 1 0;
  }
}
anim rotate.rotateX rotateX joint3 2 1 0;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 spline spline 1 1 0;
    10 0 spline spline 1 1 0;
    15 0 spline spline 1 1 0;
    22 0 spline spline 1 1 0;
    30 0 spline spline 1 1 0;
  }
}
anim rotate.rotateY rotateY joint3 2 1 1;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 spline spline 1 1 0;
    10 0 spline spline 1 1 0;
    15 0 spline spline 1 1 0;
    22 0 spline spline 1 1 0;
    30 0 spline spline 1 1 0;
  }
}
anim rotate.rotateZ rotateZ joint3 2 1 2;
animData {
  input time;
  output angular;
  weighted 0;
  preInfinity constant;
  postInfinity constant;
  keys {
    1 0 spline spline 1 1 0;
    10 0 spline spline 1 1 0;
    15 0 spline spline 1 1 0;
    22 0 spline spline 1 1 0;
    30 0 spline spline 1 1 0;
  }
}
anim joint4 3 0 0;
```

3 Maya Image File Format (IFF)

Developer File formats

Maya IFF

Overview of Maya IFF

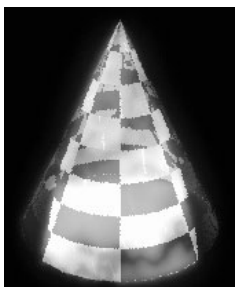
Maya outputs images in the Interchange File Format (IFF). IFF is a generic structured file access mechanism, and is not limited to images - for example, AIFF files are audio IFF. IFF images can have 8 or 16 bit per channel RGB, and optionally Alpha, and can optionally contain a 32-bit floating point depth map (sometimes called a Z-buffer).

Generate IFF files with Maya

Rendered images

Rendered images can be saved as IFF files. These IFF files have RGBA data with no depth map, by default. For example:

1. Load a scene, such as cone.ma
2. Switch to rendering mode, and select Render into new window from the Render menu.
3. When the render has finished, use the right mouse button to click on the image and select Save image from the Images menu.
4. Enter a file name in the file browser. The output image should look like cone.iff:



Shadow maps

Light sources in Maya can read and write shadow maps. These shadow maps are stored as a depth map in an IFF file. The depth map may be either a traditional Z buffer or a midmap. The stored value for a midmap

is half of the distance between the first and second surfaces. Midmaps reduce the number of incorrect self shadowing occurrences. For more information on midmaps, see Graphics Gems 3, p. 338.

Different light sources produce different shadow maps. A spot light subtending an angle below 90 degrees will produce one shadow map. If the subtended angle is greater than 90 degrees, it will produce five shadow maps - one central and five peripheral. A point light will produce one shadow map in a simple scene, and up to six shadow maps in more complicated scenes.

Try the following example:

1. Load a scene with a light in it, such as torusShadow.ma
2. Select the light and open the Attribute Editor.
3. Expand the Shadows section.
4. Check the Use depth map check box in the Depth Map Shadow Attributes section.
5. If you want a midmap, check the Use Mid Dist Depth Map check box in the same section.
6. Check the Write shadow map check box in the same section. Enter a file prefix in the Depth map name text field.
7. Render a scene.
8. You should now have one file*, each beginning with the prefix selected in step 6. The suffix will be the name of the light source and then the type of shadow map generated. For the torusShadow example, you should have one of two images, shadowmap_pointLightShape1_int.SM.iff or shadowmap_pointLightShape1_int.MIDMAP.SM.iff, depending upon your choice in step 5.

View and convert IFF images

Included with Maya is a stand-alone program called fcheck that can view IFF images and depth maps. The imgcvt program is able to convert IFF files to other formats (IRIX and Linux and Windows only). On IRIX, Linux and Windows, use the imgcvt program to convert IFF files to other formats.

Program with the IL and FL libraries

To use IFF images in custom software, we recommend using the IL and FL libraries. The FL library is for use with generic IFF files. 99% of the time, it won't be necessary to work with this directly. The IL library—based on the FL library—deals specifically with IFF images, and is much more useful for image reading and writing.

The IL library abstracts the contents and format of the file. You can specify the format in which to retrieve the data, and the library deals with the nitty gritty details. Using the library is fairly straightforward, and most of the necessary information can be gleaned by reading through the example programs listed below in the Maya Developer's Tool Kit.

Note	For stand-alone applications, the IL and FL libraries must be linked in. Maya includes these libraries, though, so plug-ins don't have to link in any special libraries.

Use example programs

To access these example programs and their descriptions, see the *Maya API Developer's Manual*, *Maya Example Plug-in Descriptions*, *Miscellaneous plug-ins*.

iffInfoCmd—a plug-in that retrieves information about an image. Requires iffreader.

iffPixelCmd—a plug-in that retrieves the value of any pixel in an image. Requires iffreader.

iffPpmCmd—a plug-in that converts an IFF to a PPM. Requires iffreader.

Additional resources

IL and FL man pages

The IL and FL man pages contain documentation for each of the functions that can be called from a plug-in to manipulate IFF files. See the **IFFmanPages** for details.

fcheck man page

The fcheck program provided with Maya allows you to view IFF files directly. See **"Overview of FCheck"** in the *Rendering Utilities* book for details.

imgcvt man page (IRIX, Linux and Windows only)

The imgcvt program provided with Maya allows you to convert IFF files to other formats. See **"imgcvt"** in the *Rendering Utilities* book for details.

IFF Format overview

The following is a quick description of the flib library. It implements a generic structured file access mechanism based on a generalized IFF format. This is what we currently use for images and textures.

Kernel

File type independence

The primary goal of the flib library is to present all file accesses in a homogeneous manner. Disk files, pipes, memory segments, and so on are all logically represented as files and are manipulated through the same set of functions. The flib kernel is composed of eight functions: FLopen, FLreopen, FLclose, FLread, FLwrite, FLseek, FLtell and FLflush. These low level IO routines can be used instead of the libc routines (open, fopen, read, fread, write, fwrite...).

Another advantage is the removal of certain restrictions caused by the files opening mode. For example, writing in a read open pipe file (such as "pipe:cat file") is made possible.

FLopen uses naming conventions to identify the type of logical file you wish to handle (no longer necessary to specify different methods for open, popen, fopen, socket, and so on). The currently recognized names are:

File name	Description
name	ordinary disk file
name.Z	compressed file
mmap:name	memory mapped file
pipe:cmd [args]	standard input (output) of cmd
fd:#	file descriptor number #, 0,1,2 for stdin, stdout, stderr
stdin, stdout, stderr	aliases for descriptors 0,1 and 2
host:name	file on remote host
user@host:name	file on remote host accessed via user account

File name	Description
mem:addr	memory segment at address addr

Certain limitations exist depending on the exact nature of the opened file object (e.g. can't FLseek on a pipe).

Files are buffered when this makes sense. Transfers can equally be accelerated by minimizing the amount of memory moves. FLbgnread, FLendread, FLbgnwrite and FLendwrite allow you to gain direct access to the read/write buffers in the library. These are particularly efficient for memory mapped files.

Format independence

File access libraries based on flib can provide an extra degree of independence with respect to the format of the data it contains by using the FLfilter function which lets you pass a file through an external filter before reading/writing it.

Control

Error handling is similar to that offered by the standard C IO functions and system calls. The state variable flerror is modified when a error occurs and several functions are provided to allow access to this value: FLerror, FLseterror, FLperror, FLstrerror, FLoserror, and FLsetoserror. The set of errors that are handled is a superset of the IRIX and Linux standard errors (errno, strerror, h_errno and hstrerror if supported by the system).

IO functions are implemented to avoid cascading errors. However, it is strongly suggested that you do not attempt to continue reading/writing when an error occurs.

Certain parameters of the library can be modified by calling FLconfig: creation of temporary files, mapping, automatic compression/decompression.

FLsetpath and FLaddpath allow you to define and augment the path used to resolve file names for read access.

FLbuildpath and FLfreepath are used to construct and destroy path that are activatable by the FLswitchpath call which optimizes frequent path changes. FLsetreorder can also be called to optimize path traversal.

Structured files

Flib implements a set of rules for file structure derived from IFF. The structure is based on the use of tags to identify blocks of data called chunks or structures of chunks called groups. Each tag is made up of four

characters and is immediately followed by the size of the chunk or group that it describes coded on 4 bytes. This structure is the same as in the IFF (Interchange File Format), with a few extensions. All data is written in big endian format, except for tags, which are handled as pseudo character strings. (Byte swapping is handled at compile time).

Block size data allows the parser to skip information it does not recognize.

There are two types of tags: tags that define the file structure (i.e groups) and tags that contain data.

Groups

Four tags are used to arrange blocks into groups: FORM, CAT, LIST, and PROP. The first four characters following the size are used to identify the type of the group.

FORM defines the beginning of a data block, in a way similar to a C struct.

```
FORM 38 TEXT
      CHAR 6 “Times”
      CHAR 12 “Hello World”
EOF
```

is similar to

```
struct Text t = {
      char *f = “Times”;
      char *c = “Hello World”;
};
```

The size of the group (38) is equal to the size of the data it contains (6 plus 12) plus the size of the headers (4 for TEXT, 8 for CHAR 6 and 8 for CHAR 12) giving, in this case, 6+12+4+8+8 = 38.

As in C structures you can nest groups as in the following example:

```
FORM 52 TEXT
      FORM 8 FONT
            CHAR 6 “Times”
            LONG 4 <12>
            LONG 4 <0>
      CHAR 12 “Hello World”
EOF
```

or in C terms:

```
struct Text t = {
      struct Font f = {
            char *n = “Times”;
            int s = 12;
            int d = 0;
      };
      char *string = “Hello World”;
```

```
};
```

This example may not show that blocks are not constrained to use a unique data type and may contain the equivalent of a complete C structure. The role of the FORM tag is to separate independent blocks of data that can be handled separately and to specify the meaning of each sub-unit. In the example above the CHAR chunk in the FONT FORM does not mean the same thing as the CHAR chunk in the TEXT FORM. The FORM tag is used to determine how you interpret an ordered set of data types.

CAT defines a concatenation of independent objects with no order relation between them. Two typical uses of CAT's are for libraries of objects (pictures in the upcoming example) or clipboards (second example).

```
CAT 3632 PICT
      FORM 1234 PICT ...
      FORM 2378 PICT ...
EOF

CAT 2130 CLIP
      FORM 1234 PICT ...
      FORM 876  DRAW ...
EOF
```

Searching through a structured file is generally greatly accelerated, even in a CAT that has no order amongst its members, through the knowledge of the size of every group or chunk specified in the header.

LIST defines an ordered set of objects (FORM data blocks) and, along with PROP, is used to group objects with similar properties, avoiding redundancy. For example a sequence of equal sized images might be represented in the following way:

One image would have a structure like:

```
FORM .... PICT
      IHDR 32  [image size info]
      BODY ... [image data]
EOF
```

then a sequence of like-sized images could be done as follows, sharing the common header information:

```
LIST ... ANIM
      PROP 44 PICT
            IHDR 32
      FORM ... PICT
            BODY ....
      FORM ... PICT
            BODY ....
      FORM ... PICT
            BODY ....
```

```
EOF
```

The information specified in a PROP construct applies until the end of the LIST. They can be redefined locally in a FORM the same way local C variables can (in the above example the common IHDR is valid in all PICTs that don't include an IHDR block of their own.

Data blocks

Data blocks are defined by:

[tag] [size] [data]

Example: an image could have the following structure:

```
FORM 12304 IMAG
      IHDR 200  ... picture header, size, maps ...
      LINE 800  ... data from line 1 ...
      LINE 800      ... data from line 2 ...
      ...
```

for a library:

```
CAT  64200 IMAG
      FORM 12304 IMAG
            IHDR 200
            ...
      FORM 12304 IMAG
            ...
```

and for a sequence,

```
LIST 64200 IMAG
      PROP 208 IMAG
            IHDR 200  ... Common header ...
      FORM 12394 IMAG
             ...
      FORM 12304 IMAG
            IHDR 200  ... Local redefinition ...
            ...
```

Alignment considerations

IFF blocks align to 2 byte boundaries. The size specified in the header does not take the padding into account. Current machines typically align their memory on 4 or 8 byte boundaries. Flib uses 8 extra TAGS to let you specify alignment information. Four are used to align to four byte boundaries (FOR4, CAT4, LIS4 and PRO4) and four are used to align to 8 byte boundaries (FOR8, CAT8, LIS8 and PRO8).

Data blocks inherit the alignment of the group that contains them (as well as any sub-groups; hence it's illegal to create a group aligned to 2 byte boundaries inside a group aligned to 4 byte boundaries. However the reverse is perfectly valid.)

Extensions

One of the major constraints of IFF is that you have to know the size of a group or a chunk before writing it to a file. If you want to change the information a block contains you have to be able to modify the header to reflect changes in the size of the structure. This poses no problem for seekable file (memory or disk files) but does pose problems for other types of files. Rather than create intermediate temporary files, flib implements a mechanism allowing you to say that you don't know the size of the block you are working on. Since negative block sizes are meaningless, two special values are set aside for this purpose: FL_szFile, indicating that the size will be written in later once the entire group has been written and FL_szFifo indicating that the size will not be written because the file is not seekable, A special zero sized block (GEND) is used to indicate the end of the structure.

Functions

Blocks can be read and written using calls to FLgetchunk and FLputchunk. For more direct control the user can call FLbgnget and FLbgnput to open a block. FLput and FLget supply services equivalent to FLread and FLwrite within a block. After appropriate number of FLput or FLget calls you close the block using FLendput or FLendget.

Groups are handled using FLbgnrgroup, FLbgnWgroup, FLendrgroup and FLendwgroup. Flib also implements a generic parser, FLparse, that can scan a file and check its consistency as well install callbacks for each step of the parse (start of group end of group).

Toolbox

The flib library also provides tools for the handling of linked lists (FLxxxnode and FLxxxlist), buffers ((FLmalloc and al) and external filters (FLfilter and FLexec).

Introduction to the image library

The IO image library is part of the flib library. A set of routines allows to read and write images in a structured file.

File format

The format of an image file is very flexible. Constraints on the number and relative position of the different blocks are minimal and often purely of a logical nature (for example, "the header must come before the pixel blocks" rather than more static constraints such as "the header begins at offset 124").

An image file being first and foremost a file (as far as flib is concerned) the user is free to insert extra blocks. A minimal image is composed of a FOR4 group (aligned to a word) of CIMG type containing, in the following order:

- a BMHD header (bitmap header)
- a FOR4 group of type TBMP (tiled bitmap)

Pixel information is contained in the TBMP group, which can be quickly skipped if necessary.

In its minimal version, the TBMP group contains pixel related data blocks, in some order. For example, for a picture divided in four tiles we have:

```
FOR4 <size> CIMG
      BMHD 24 ... definition of size, maps, etc...
      FOR4 <size2> TBMP
            RGBA <ttile1>... tile 1 pixels ...
            RGBA <ttile2>... tile 2 pixels ...
            RGBA <ttile3>... tile 3 pixels ...
            RGBA <ttile4>... tile 4 pixels ...
```

The header is defined by structure ILheader. RGBA blocks have the following structure:

[x1, y1, x2, y2] : tile coordinates (2 bytes each)

[pixels] : encoded according to compression mode.

If the image has a z-buffer, it is described by ZBUF blocks with the same structure as the RGBA blocks, RLE encoded.

- End fields. They display at the end of any image data, and will be displayed by fcheck. Calls to FLIB after image loading can be used to check for and retrieve this data. ILIB ignores these fields. The following four fields provide information about the image:

HIST—string data giving the Maya command line from which this image was created.

VERS—string data giving the Maya cut information.

CLPZ—depth map specific field giving the clipping planes used. This information is stored as two float values.

ESXY—eye x-y ratios. This is a depth map specific field used to compute the xy eye coordinates from the normalized pixel coordinates stored as two float values.

Note: The ILIB library does not support these end fields.

Functions

Some functions allow the reading and writing of images in line to line mode without worrying about tile management. Images can also be automatically zoomed and/or corrected (by lookup) during read (correction on compressed data being significantly more efficient). Finally, an automatic conversion system makes it possible to read images stored under other formats.

For more details on routines from the image library, see the man pages as well as “includes” and examples provided with the library.

Extensions to the IFF format

PATH defines the search path for includes.

INCL is an include block.

EOVC is the end-of-variable-length-chunk marker.

GEND is the end-of-group marker.

Syntax:

```
      PATH  #     <directory names>
      INCL  #     <file names>
      EOVC  szUnknown
      GEND  0
```

PATH and INCL are data chunks and thus inherit alignment.

EOVC and GEND are use in fifo files since the chunk and group sizes can not be random accessed. Creating groups or chunks with unspecified size on a fifo will give something like:

```
FORM  sz_Fifo TYPE; start of form
      BLCK  sz_Fifodata; block 1
      EOVC  sz_Unknown; end of block 1
      ....            ;
      BLCK  sz_Fifodata; block 2
      EOVC  sz_Unknown; end of block 2
      GEND  0         ; no more block in this FORM
```

EOVC is a block/group end marker, while GEND is understood as a request to close the current group, going up one level.

Using unknown sizes when writing a seekable file will produce a very similar structure except that no EOVC is written. This is very useful when parsing a file under construction since the end of group can be detected w/o any random access to the group header. The size field of GEND is set to zero to allow other standard IFF parsers to skip it silently. The EOVC’s size field value will produce an error if read by a standard parser (w/o fifo extensions).

Warning	For compatibility with the previous version of the IFF parser GEND is still followed by sz_Unknown (and EOVC sz_Unknown if the file is a fifo).

Since there is no reliable way of skipping a block of unknown length it is STRONGLY RECOMMENDED that writer and reader of a fifo agree on the file's content.

Note that the implemented parser is smart enough to locate EOVCs in a file, but the skipping process is slow due to intensive read and compares. And again, this can NOT be considered 100% reliable.

4 Channel move files (.mov)

Developer File formats

Channel move files (.mov)

A move file is an ASCII file that stores the channel data (such as x translate, y translate, and z translate).

Format

The .mov file is a matrix of numbers where rows represent frames and columns represent channels. There is one row for each frame in the .mov file and one column for each channel.

Each row must start on a new line and each column must be separated by spaces. The channel data is represented as doubles. This allows for roughly 16 significant digits.

The following is a sample format from a .mov file. It has four rows and six columns. This means that the sample shows the channel data for four frames and six channels.

```
1.000  0.000 0.900 36.000 0.000
0.000
4.000  0.000 2.000 36.000 0.000
8.000
0.000  9.500 16.000 8.000 0.000
0.000
9.450  0.000 0.000 50.000 3.500
8.000
```

You can read and write .mov files to import and export motion data. Remember, however, that the .mov file is simply columns of numbers without any header information. In order for data to be written out and read back into the same objects and channels, the list of channels must be identical in the File Import/Export option boxes or with the movIn/ movOut commands.

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W