I File Formats

This section details the various file formats available in Houdini.

1 GEOMETRY FORMATS SUPPORTED BY HOUDINI

1.1 GEOMETRY FORMATS

Extensions	Type	Read	Write	Notes
.geo	Internal	•	•	Houdini ASCII Geometry Format
.bgeo	Internal	•	•	Houdini Binary Geometry Format
.poly	Internal	•	•	PRISMS ASCII format (polygons only)
.bpoly	Internal	•	•	PRISMS Binary Format (polygons only)
.d	Internal	•	•	PRISMS Move/Draw format (polygons only)
.rib	Internal		•	RenderMan Geometry
.dxf	External	•	•	Uses *gdxf* – polygons only, no attributes
.obj	External	•	•	Uses *gwavefront* - polygons only. Only texture and point normal attribute (groups are kept on read)
.iv	External/ Internal	•	•	Uses *ginventor*. Does not support Bézier or Metaball primitives. Only some attributes.
.wrl	External		•	VRML used by world wide web. Uses *ginventor*.
.sdl	External	•		Uses *gsdl* command.
.eps	External	•		Uses *geps* command.
.med	External	•		Uses *gmed* command.
.lw	External	•		Uses *glightwave* command.
.ply	External	•	•	Uses *gply* command

.GEO HOUDINI ASCII GEOMETRY

This information is written out into the file as a series of mnemonics which can be read as ASCII text. Refer to the *.geo File Format Description* p. 281 in the *Formats* section for a comprehensive description of the structure of information in a *.geo* file.

.BGEO HOUDINI BINARY GEOMETRY

This file type is the same as the *.geo* format, except that the data is written in binary format. That is, the information is represented in a tightly-coded short-hand rather than as ASCII characters you could read as text. This means the file contains the same information as a *.geo* file, but is more compact and loads more quickly.

Refer to the *.geo File Format Description* p. 281 in the *Formats* section for a comprehensive description of the structure of information in a *.geo* file.

.POLY PRISMS ASCII POLYGON

This format only supports POLYgonal geometry types. It is included in the Geometry Editor to maintain compatibility with other, older, systems.

Tip: In a .poly file, the first point referred to by the poly's is point number 0.

.BPOLY PRISMS BINARY POLYGON

This format is a binary version of the *.poly* format.

.D (PRISMS ASCII MOVE / DRAW LINE FILE FORMAT)

The *d-file* is a seldom-used ASCII polygon file format which is simpler than the *.poly* format as it only has only two sections. The two *.poly* sections are the *POINTS* and *POLYS* sections. The *.d* format has only moves and draws, where a move signifies the beginning of a new polygon.

The *.d* format is suitable for user-written programs that create polygon information and where you do not want to use the Houdini polygon library. You have the ability to describe 2D or 3D lines and comments.

syntax

The standard syntax for a *d-file* is given below as a set of commands. *d-files* have one command per line, every command is a single character in the first column of a file, and commands are followed by white space, then by a list of arguments. The arguments are delimited by white space. White space is any combination of tabs and space characters.

Character	Command	Arguments (#, type)
`#`	comment	any text (<256 ASCII)
`m`	moveto	x y [z] (float)
`d`	drawto	x y [z] (float)

The *comment* command is ignored.

The *moveto* and *drawto* commands are the basic drawing commands that specify line endpoints in space.

.DXF

Stands for "Data eXchange Format". This is the format used by Autodesk® for their AutoCAD™ software. It supports only polygonal geometry types. Information on *.dxf* layers are split into groups when read into the SOP Editor. Texture and colour information is lost if a .dxf file is saved.

.OBJ

The "object" format used by Alias® Wavefront®. It supports polygons, texture, normals, and groups.

.IV (INVENTOR)

This is SGI's 3D geometry format. It supports polygons, spheres, cylinders, and NURBS-based geometry.

VRML (VIRTUAL REALITY MODELLING LANGUAGE)

This format is similar to the inventor format. There are extensions which can be added manually. There is a converter available which comes with the WebSpace® software provided by SGI®. The VRML format is used for 3D models on the World Wide Web.

.SDL (ALIAS)

Houdini is able to read Alias® SDL files – they cannot be created. Only spline surface geometry (and instances of such), and the name of each patch and shader is imported; colors and texture coordinates are not imported.

.EPS (ADOBE)

Houdini will read Adobe® extended postscript *.eps* files from Illustrator 5.5.
Fills, patterns, and colours are not imported. Produces planar geometry.

.MED (META EDITOR)

Houdini will read Meta Editor metaball files, although eccentric information is lost upon loading.

.LW (NEWTEK LIGHTWAVE)

Houdini will read NewTek' LightWave object file. Objects from version 3.5 and earlier are supported. Only the following LightWave features are handled:

- points
- polygons
- surface names (translated as primitive groups)
- surface color (translated as primitive colors)
- surface transparency (translated as primitive alpha)
- surface smoothing (currently translated as internal cusp operations)

Houdini will ignore all other features. It cannot load files saved with layer information. The file to be loaded *must* contain points. However, there need not be any polygons in the file.

Note: By default, Lightwave objects are rendered as faceted (no smoothing) whereas in Houdini you must explicitly cusp polygons in order to achieve this. As a result, if you load an object that has no smoothing values set (i.e. all faceted) you will end up with all points in the resulting object being "uniqued" – giving you a *much* larger dataset. In order to get around this, you must manually convert the file using the *glightwave* utility with the *-s* option.

.PLY FORMAT

The .ply format is a polygonal format. It can handle very large datasets, and supports vertex normals and colours. Houdini can both read and write .ply files using the *gply* standalone. It was designed at Stanford University and UNC Chapel Hill, and is mostly used for academic research and with Cyberware scanners.

1.2 GEO IO TABLE

If you have a format that you would like to be able to read, you can write a converter between your file format and the *.geo* format. Do this by adding your entry to the *GEOio* table in *$HFS/houdini/support* . The modified .geo file can appear anywhere in the Houdini search path. As well, future formats will be added through this file.

For technical details, contact Side Effects Software Support.

2 CHANNEL FILE FORMATS

2.1 CHOP FORMATS

Extensions	Type	Read	Write	Notes
.chan	External	•	•	Houdini ASCII Channel Format containing raw values in rows & columns.
.bchan	External	•	•	Houdini Binary Channel Format, equivalent to .chan
.clip	Internal	•	•	Houdini ASCII Native CHOP Format. It contains raw values like .chan plus more information.
.bclip	Internal	•	•	Houdini Binary Native CHOP Format, equivalent to .clip
.chn	External	•	•	Houdini ASCII Format for a group of channels expressed as keyframed segments (e.g. bezier(), ease(), etc.)
.bchn	External	•	•	Houdini Binary Format, equivalent to .chn
.aiff	Internal	•	•	Standard Audio Format
.aifc	Internal	•	•	Compressed Audio Format
.au	Internal	•	•	NeXT Audio Format
.sf	Internal	•	•	NeXTAudio Format
.snd	Internal	•	•	Audio Format
.wav	Internal	•	•	Windows Audio Format

DIFFERENCE BETWEEN INTERNAL/EXTERNAL FORMATS

External formats are implemented as standalone converter programs that are piped into Houdini. Internal formats are implemented as converters that are built-into the Houdini program itself. Both format types appear the same to Houdini users. Internal formats are slightly faster. New external formats can be interfaced to Houdini without requiring the Houdini Developers' Kit, only the public-domain source code as described below.

2.2 CHANNEL FORMAT DESCRIPTIONS

.CHAN HOUDINI ASCII CHANNEL

This information is written out into the file as ASCII text, one row per frame of data, and one column per channel. The file contains specific information regarding channel names, sample rates etc. This format can be imported or exported from the Houdini channel editor (and is even compatible with old *action* channel formats).

.BCHAN HOUDINI BINARY CHANNEL

This file type is the same as the *.chan* format, except that the data is written in binary format. That is, the information is represented in a tightly-coded short-hand rather than as ASCII characters you could read as text. It is more compact and loads more quickly.

.clip houdini ascii chop

This is the Houdini native CHOP format. This format contains all the information held by one CHOP. Currently this format contains named channels, each with an array of raw sample values. The clip also contains a start and end range, a sample rate, channel extend conditions and quaternion attributes.

.bclip houdini binary chop

This is the binary version of a clip. This format is recognized by the magic number 'bclp' in its first four characters.

.chn houdini ascii spline

This format is used to describe channels which contain segments, slopes, accelerations, interpolation types and other spline based attributes. It is compatible with the internal Houdini Channel Editor.

.bchn houdini binary spline

This is the binary version of a `.chn` format.

2.3 LOADING / SAVING CHANNEL FORMATS

HOW TO SAVE CHANNEL FILES FROM A CHOP

You can Save channel files from a CHOP into a file by selecting *Save Data Channels...* from a CHOP tile's pop-up menu. The format used is determined by the extension of the file name given.

HOW TO LOAD CHANNEL FILES INTO A CHOP

You can Load channel files into a CHOP by creating a File CHOP and specifying the Channel File name in the File CHOP's parameters.

HOW TO READ .CHAN / .BCHAN FILES INTO CHOPS

1. Enter the Channel Editor, and select the channel (say *geo1/ty*) you want to load the raw .chan/.bchan values into.
2. Load the .chan file using *File > Load Active Chan/Bchan*.

Enter the CHOP Editor, and place a Fetch CHOP. Select the object and channel by specifying them in the *Source* page of the Fetch CHOP's parameters.

2.4 SUPPORTED AUDIO FORMATS

These are the audio formats currently supported by the SGI version of Houdini. You can use the *claudio* converter to convert from a known audio type to future audio format as they are implemented by the SGI audio library.

The NT version of Houdini supports only the AIFF and WAV audio formats.

.AIFF	Older, uncompressed Audio Interchange Format standard for audio.
.AIFC	Extended AIFF-C standard format (default SGI audio format).
.AU	NeXT Format.
.SND	NeXT Format.
.SF	Berkeley/IRCAM/CARL Sound File Format.
.WAV	Windows RIFF WAVE Format.

2.5 ADDING OTHER CHANNEL FILE FORMATS

CHOPIO TABLE

If you have a channel file format that you would like to be able to read with Side Effects Software products, you can write a converter program between your file format and the Houdini *.clip* format.

The *CHOPio* file located in *$HFS/houdini* must be modified in order to use your new converter.

The modified converter program can appear anywhere in the search path.

chopio

The *CHOPio* file contains the commands used to convert to/from any of the external channel formats. In order for a new format to be converted transparently by Houdini, you must add the commands which will convert from your new file format to the Side Effects channel format; and a command which will convert from a Side Effects channel format to your new file format. The commands are the name of your program followed by suitable command-line arguments.

The new line of the file contains the new filename suffix, the command that reads your format, and the command that writes your format. *%s* represents the file name.

procedure

The procedure for adding a new format is:

1. Determine an unused extension (e.g. *.xclip*).

2. Use the source code to develop the program to convert to/from the two formats.

3. Put your new programs which read and write the format in your search path.

4. Come up with the two commands to read and write the format to Houdini on *stdin* and *stdout*, and add these commands to the *$HFS/houdini/CHOPio* file.

Note, you do not need to write both a converter for both reading and writing from the format if you do not need to both load and save the file format.

Note: Because files in *$HFS/houdini* are generally not to be modified, you should copy these files to a sub-directory of your home directory: *$HOME/houdini*, and modify them from there.

LOCATION OF SOURCE CODE

Example source code for reading and writing Houdini clip files is found in:

$HFS/houdini/public/CPD.tar.Z

2.6 CHOP INTERNALS

All CHOP channels are made of arrays 32-bit floating point numbers, including CHOPs containing audio samples.

A CHOP IS SAMPLED WHEN

- it is needed by a downstream CHOP
- it is accessed through the `chop()` function by another OP
- by displaying the channel in a graph
- by connecting the channel to an audio device (speaker)
- it is linked to an OP
- CHOPs are being output from Houdini to a file or the textport
- cooked by the opcook textport command

EXTEND CONDITIONS

Extend conditions: sampling CHOPs out of bounds: When using the `chop()` function to sample a channel, the index-value may be outside the interval of the CHOP. But a reasonable value is returned. The user is able to control the value of the channel outside its interval: See the Extend CHOP.

A CHOP holds it channels at a single sample rate. If other CHOPs need to get it at different sample rates, they will make their own temporary array at their desired sample rate.

FRAME DEPENDENCIES

CHOPs are frame-dependent only if its data channels change every frame Houdini advances to another frame. Most CHOPs will not be frame dependent, even if they have an animated curve in them because it will just be sampled (not cooked) each time it is polled for values at a certain frame.

Frame-dependent CHOPs are ones whose shape changes each time it is called such as:

- a CHOP that reads data from an external device each frame
- a CHOP that reads a SOP and converts it into a curve each frame
- a CHOP that has animated (non-constant) control channels.

CHOPs are independent of `$F`. Only when you use CHOPs in a display does `$F` come into play to use the CHOP and its sample rate to choose the index to sample.

3 SCENE FILE FORMATS

3.1 SCENE FORMATS

The following formats can only be generated – they cannot be read. They are used for output to rendering programs.

.IFD (MANTRA)

Stands for “Instantaneous Frame Description”. It contains all information necessary to render a 3D scene in *mantra*, including lights, shaders and geometry. For more information on how to create .ifd files see *mantra3 Output OP* p. 754 of the *Outputs* section.

.RIB (RENDERMAN BYTE STREAM®)

Pixar’s® RenderMan® geometry format. It contains all information necessary to render a 3D scene in RenderMan, including lights, shaders and geometry. *.rib* geometry can be created by Houdini, but not read.

4 IMAGE FILE FORMATS

4.1 IMAGE FORMATS

Extensions	Type	Read	Write	Notes
ip / iw / md	External		•	*iplay* / image Window
fip	Internal		•	Flipped *iplay* Window
vf/a60	Internal	•	•	VideoFramer/Abekas
.cin / .kdk	Internal	•	•	Kodak Cineon format
.fit	External	•	•	FIT tiled image format
.gif	External	•	•	GIF
.gif89	External	•	•	GIF89a (GIF w/alpha)
.jpg / .jpeg	Internal	•	•	JPEG (Very efficient for storage; lossy)
.pic *	Internal	•	•	Houdini picture format
.pic.gz **	Internal	•	•	gzip compressed .pic
.pic.Z	Internal	•	•	Houdini compressed
.qtl	External	•	•	Quantel YUV format
.rat	Internal	•	•	Random access texture map
.rla / rlb	Internal	•	•	Wavefront format
.rla16	Internal	•	•	Wavefront .rla 16 bit format
.pix	Internal	•	•	Alias .pix format
.sgi / .rgb .rgba	Internal	•	•	SGI format (a.k.a. .rgb by non-Houdini software)
.si / .pic	Internal	•	•	Soft Image format
.tif/ .tiff	Internal	•	•	TIFF format, with alpha
.tif3	Internal	•	•	TIFF RGB, no alpha
.tif16	External	•	•	TIFF 16 bit format
.tx	External	•	•	Renderman texture images (requires RenderMan t.kit)
.tga	Internal	•	•	Targa format
.vst	Internal	•	•	Targa Vista format
.vtg	Internal	•	•	Vertigo format (see below)
.yuv	Internal	•	•	Abekas YUV format

* Some vendors of other software packages use .pic to refer to SGI (.rgb / .sgi) images – this is not the same as the Houdini .pic format. Houdini should automatically load these correctly. However, should you persist in having trouble reading a .pic file, you may want to rename it to have a .sgi extension so as not to confuse it with Houdini .pic. Anywhere you see .pic in Houdini, it will refer to the Houdini .pic format and not the SGI format.

** Some formats can be compressed using gzip by appending .gz to the filename. This slightly increases the r/w times of images, but often yields smaller file sizes.

BOTTOM-UP VS TOP-DOWN IMAGES

When images are stored as files, no assumption is made about whether images are stored bottom-up or top-down. Images are simply a sequence of scan-lines. An image is typically displayed using *ip* – the *iplay* program – which by default, displays the image from bottom-up.

If you need to display images as top-down, use *fip* instead of *ip* . To set the default behaviour as top-down, set the FLIP enviroment variable in your *.login* file:

```
setenv FLIP
```

Thereafter, these commands will display images as top-down instead of bottom-up:

```
icp myImage.tif ip
iplay myImage.tif
```

CONVERTING BETWEEN IMAGE TYPES (ICP)

You can convert between any two image types (e.g. .jpeg .pic .tif) using *icp* .
For example:

```
icp myImage.pic myImage.jpeg
```

converts the file to .jpeg format.
For more information, see *StandAlone > icp – Copy / Crop Image* p. 364.

4.2 ABEKAS (YUV) FORMATS

Abekas images have fixed resolutions of 720×486 (NTSC) or 720×576 (PAL):

a60:*.rgb	Abekas RGB image on A60
a60:*.yuv	Abekas YUV image on A60
*.rgb	Abekas RGB image file
*.rgb.Z	Abekas RGB image file compressed
*.yuv	Abekas YUV image file
*.yuv.Z	Abekas YUV image file compressed

vf:hh:mm:ss:ff, $F,	Video framer output to video device.
	hh = hours
	mm = minutes
	ss = seconds
	ff = frames
	$F = frame offset
	inc = number of frames to record images to.

ip	Image placed in an IRIS window with the viewing controls of the *iplay* program.

You can record to Abekas and Accom by specifying it as a device (e.g. *a60:*) in the *Output Picture* filename instead of using an actual filename for the rendered output.

These images can be un/compressed using non-Houdini tools. See the UNIX *compress* and *uncompress* programs.

ABEKAS IMAGES

Abekas images exist in two places: in UNIX disk files and on an Abekas A60, A65 or A66. Abekas images are stored in two formats: RGB and YUV which Houdini can both read and write. All programs in the Image Toolkit as well as the Houdini renderers can handle all these combinations.

Currently, all Abekas images are a fixed resolution: 720 × 486 for NTSC and 720 × 576 for PAL. Abekas images containing the internal RGB format have a suffix of *.rgb* and the images containing YUV format have a suffix of *.yuv*. In both cases, the first part of the file name is the integer frame number on the Abekas where the image is, was, or will be located, such as: *350.yuv*.

When the Image Toolkit stores/fetches images to/from the Abekas, the image on the Abekas must be specified with machine name, *a60* or *abekas*. Examples are file names such as: *a60:101.rgb* or *abekas:123.yuv*.

Images on the Abekas are transferred in about thirty seconds in RGB format, and in six seconds in YUV format. In the former format, the images are internally converted to/from YUV format to Abekas format and conversion is much slower than on an IRIS.

.rgb and *.yuv* files can be saved at any resolution. However, the library can only read files which are one of several pre-defined file sizes. These file sizes determine the resolution of the image:

699840	720 × 486 (NTSC YUV file)
049760	720 × 486 (NTSC RGB file)
811008	720 × 576 (PAL YUV file)
1244160	720 × 576 (PAL RGB file)

This also means that compressed *.yuv* and *.rgb* files cannot be read since the file size is indeterminate. They have to be first uncompressed with the UNIX command, *uncompress*. However, compressed *.rgb* and *.yuv* files can still be created.

When copying files to the Abekas using the Image Toolkit, resolution is not checked and whatever you want to send is sent. How the Abekas will handle non-standard frame resolutions is unclear.

When reading from the Abekas device, the image is assumed to be at 720 × 486 resolution. However, if the environment variable *ABEKAS_PAL* is set, it is assumed that the images will be 720 × 576.

In addition, the environment variables:

ABEKAS_PAL_XRES
ABEKAS_PAL_YRES
ABEKAS_NTSC_XRES
ABEKAS_NTSC_YRES

can be set to override the above mentioned resolutions.

MODEL A60 - A65 SUPPORT

There is separate device driver support in the Image Tool Kit library for the model 65 Abekas to handle the different protocol required between it and the model 60. In order to decide which protocol to use, the following system is used:

1. Any Abeki with names *a60, a61, a62, a63, a64, abekas,* or *abaccus* is automatically assumed to have the model 60 protocol. Any with names *a65, a66, a67, a68,* or *a69* is automatically assumed to have the model 65 protocol. If these conventions are followed, no special action is required to be able to access the Abekas device.

2. If this fails, the hostname for the device is scanned for “a60” or “A60”. If this substring is detected, it is assumed the device follows the model 60 protocol. If this fails, the hostname is scanned for any occurrences of “a65” or “A65”. If this substring occurs in the hostname, then it is assumed that the device follows model 65 protocol.

3. In extreme cases, it is possible to completely bypass the above defaults with two environment variables, called *SESI_A60_HOSTS*, and *SESI_A65_HOSTS*. For instance, if your model 60 abekas is named "fred" and your model 65 is named “wilma”, then setting *SESI_A60_HOSTS* to *fred* and *SESI_A65_HOSTS* to *wilma* will assume that *fred* is a model 60 Abekas and *wilma* is a model 65 Abekas. The environment variables can contain multiple hostnames by delimiting them with semicolons. For instance, setting *SESI_A60_HOSTS* to *fred;barney;dino;pebbles* will ensure that fred, barney, dino and pebbles will be accessed as Model 60 Abeki.

Examples:

```
% icp a65:0 ip          (# model 65, by rule 1)

% icp a63:10 ip         (# model 60, by rule 1)

% icp myA60:40 ip       (# model 60, by rule 2)

% setenv SESI_A65_HOSTS "fred;barney"

% icp fred:30 ip        (# model 65, by rule 3)
```

4.3 ACCOM SUPPORT

The Accom Workstation Disk is supported via FBio. Filename extensions *.acc* and *.accom* assume that the frame is to be read from or written to the Accom device.

The network name of the Workstation Disk is assumed to be accom. This assumption may be altered by changing the scripts located in *$HFS/houdini/sbin* .

Use evaluative backquotes to compute the actual Accom frame number when writing and reading with the Composite Editor.

EXAMPLES

Write a *.pic* file to frame 342 on the Accom.

```
icp pretty1.pic 342.acc
```

Read an image from the Accom and display in an iris window.

```
icp 342.acc ip
```

Write a series of images from ice output cop (string specification).

```
‘$F + 339‘.accom
```

Where

.accom	The file extension.
339	The frame base number (minus one) on Accom.
$F	The current frame counter.
“	The evaluative backquotes.

To write to the Accom, you must make sure the resolution is 720 × 486 for NTSC, and 720 × 576 for PAL.

4.4 ALIAS (.PIX) FORMAT FILES

An Alias format file should have a .pix extension for creation. If no extension exists for an image, the first format in the index file (typically .pic) will be used.

4.5 CINEON AND TIFF (16-BIT FILE FORMATS)

Cineon format images (*.cin *.kdk) and 16-bit TIFF images (*.tif16) are recognised as supporting more than eight bits per channel. Also Wavefront 16 bit RLA format (*.rla16) is supported.

When the Composite Editor reads or writes Cineon format files directly, *no* gamma correction is performed.

Because *mantra* outputs *.cin* Cineon files, these images can be read into the Composite Editor as 16-bit images (actually 10 bit logarithmic) without color-depth information loss.

CINEON ENVIRONMENT VARIABLES

The input and output of Cineon format images perform a logarithmic to linear conversion as described in the document: “Greyscale Transformations of Cineon Digital Film Data for Display, Conversion and Film Recording”, Cinesite Digital Film Centre, Kodak Motion Picture & Television. This conversion process can be adjusted using the following environment variables:

cineon_flip

When set (to any value) flips all Cineon images in Y during input.

cineon_film_gamma

This value is used when scaling the between printing densities and “relative log exposure” values. Default value is 0.6.

cineon_white_point

The Cineon log scale value that is considered to be full white and will be mapped on input to the maximum channel value. Range 0 to 1023 (default 685).

cineon_black_point

The Cineon log scale value that is considered to be full black and will be mapped on input to zero. Range 0 to 1023 (default 85).

SETTING THE CINEON VARIABLES

The CINEON environment variables should be set as follows for final composites requiring perfect conversion to/from logarithmic values:

```
setenv CINEON_WHITE_POINT 1023
setenv CINEON_BLACK_POINT 0
setenv CINEON_FILM_GAMMA  1.0
```

For previewing and test composites the default values that follow produce output that is better for viewing on screen:

```
setenv CINEON_WHITE_POINT 685
setenv CINEON_BLACK_POINT 85
setenv CINEON_FILM_GAMMA  0.6
```

Note: As of Houdini 1.1, the environment variables: *CINEON_OVER_EXPOSURE* and *CINEON_WHITE_VALUE* are obsolete.

Also, the lookup tables used in Houdini 2.0 can be enabled by setting the environment variable *CINEON_OLD_LOOKUP* if you require backwards compatibility.

SOME FIELD NOTES ON CONVERTING CINEON IMAGES

1. CGI imagery is calculated in gamma 1 linear RGB colour space. If you want to convert your 10 bit log data to gamma 1 linear data (so that everything is the same), the correct film gamma to use for this conversion is: 0.6 .

2. The 0 to 1 step in 10 bit log space is over *three hundred thousand* times smaller than the 0 to 1023 step. To convert the entire range from 0 to 1023 to linear gamma 1 space would require 19 bits! If you wishe to convert 10 bit log data to 16 linear data using the correct gamma of 0.6, the highest white point you can use without posterizing the data is about 825.

3. Using conversion parameters of a white point of 1023 and a film gamma of 1 *does* allow you to store the entire log range in 16 bits without loss. This is because the signal path that results has a gamma of about 1/0.6, or 1.6666, rather than 1. This essentially brightens the middle greys, allowing more of the 16 bit range to be devoted to the dark levels. If one tries to linearize the entire 0 to 1023 range using a film gamma of 0.6, the dark values will be heavily posterized, and the 10 bit log data will be ruined. Those users that choose to convert using a film gamma of 1.0, should be aware that the signal path that results has a gamma of 1.6666, and they may wish to render their images using that gamma as well.

4. It is also important to understand that the cinematographer has to point the camera into the sun to get densities in the negative that are anywhere near 1023. The vast majority of 10 bit log scans contain pixels with a maximum brightness in the high 700's to low 800's. This is important, because every 90 steps of log code values translates into a doubling (1 stop) of brightness. So, if you have material that never gets brighter than, say 843 (180 code values, or 2 stops, below 1023), when you linearize using a white point of 1023 (and a gamma of 0.6), your linear data will never get brighter than value 16383 (25% of 65535). The 16 bit levels from 16384 to 65535 (75% of your numerical precision) will never be used! (Using a gamma of 1.0, your 10 bit log value of 843 will convert to about 28600 in 16 bit space.) This is a gross waste of precision.

5. If you want to maximize your 16 bit precision, the optimal way to proceed is to ‘ride the content’. That is, you check all the film elements that will going into a shot and search for the brightest pixel. Once you find the brightest code value (say it’s 765) you add a little fudge factor to give yourself some wiggle room and pick a white point of 790 or so. You apply this one white point value consistently to convert all the layers in your shot. In practice, it is common for a single white point to be chosen for a sequence of shots that were all shot under the same conditions.

6. To find the brightest pixel, some people use a special purpose tool to scan every pixel in every frame of all the elements, printing out the highest value found. Alternately, you can set the white point to various values and read your images in and look at them. If any bright areas of the images are 1, you have to set the white point higher. You want the brightest pixels in your scans to be at about 90 percent brightness. You must set the white point using the environment variable CINEON_WHITE_POINT.

4.6 IP / IW – IPLAY / IMAGE WINDOW

The simplest and fastest way of displaying images is by specifying an Image Window *(iw)* as the output device. This simply displays the output image an an OpenGL window. Using *iw* doesn't provide the many controls that *ip* (iPlay Window) does, but is very fast.

If you want a Flipped image window, use *fiw* instead of *iw.*

Using *ip* as the output device invokes the *iplay* program to display images. This provides extra controls such as: Zoom, Alpha (transparency) display, and Contrast - Brightness. For information on *iplay*, see: *StandAlone > iplay - View Images* p. 382 .

You can use *iw* and *ip* anywhere you would normally specify an output device.
For example:

```
icp rain.pic iw
icp rain.pic ip
iplay rain.pic
icp rain.pic fiw
```

Note: Both *iw* and *ip* are "write-only" formats – you can not read from them.

4.7 JPEG COMPRESSED IMAGE FILES

The JPEG format *(.jpeg, .jpg, .JPEG, .JPG)* is a native Houdini format. JPEG is "lossy", meaning that information is lost. Compression is typically 10:1, meaning a 2Mb file will compress to about 200Kb with a relatively small loss in image quality. For images requiring fine areas of detail, you may want to consider using a non-lossy format.

You should only use *.jpeg* images in the final output stage in order to avoid generational losses due to accumulative lossy effects.

4.8 LZW AND GZIP COMPRESSED IMAGES

LZW COMPRESSED FILES

LZW compression provides much greater image compression than conventional run-length encoding. Adding a .Z suffix when creating the following image formats causes LZW compression to be performed on the image:

.pic.Z	(Houdini)
.rgb.Z	(Abekas RGB)

These compressed files can be read directly by all Image Toolkit programs. The compression is the standard UNIX compression, so existing *.pic* files can be compressed by running, for example:

```
csh-> compress name.pic
```

resulting in *name.pic.Z*. This can be displayed with:

```
csh-> iplay name.pic.Z
```

Compression typically takes three times longer than run-length encoding, but space savings are 0% to 80% better than run-length encoding. Decompression slows the reading of images by a factor of about two.

GZIP COMPRESSED FILES

As an alternative, you can also use a .gz extension for gzip compression, which is typically faster than using the .Z compression. For example:

```
csh-> gzip -9 name.pic
csh-> iplay name.pic.gz
```

4.9 MD – MDISPLAY WINDOW

Specifying “md” as the output destination copies an image to a *mdisplay* window (i.e. like *mantra* rendering to *iplay*). If an *mdisplay* window is already open, the same window is used to display the image (unless the server has been disconnected).

This allows things like the COP output driver to render sequences of images to a single iplay window. To use this, specify “md” instead of “ip” when specifying the output image name.

The Standalone application: *imdisplay* can be used to allow other renderers (or other applications) to take advatage of mdisplay’s capabilities.

Usage: `imdisplay image_name` or `imdisplay width height format`

4.10 RAT – RANDOM ACCESS TEXTURE FILES

Using RAT textures provides several advantages over using other image file formats for texturing. Random Access Texture maps (RAT) are a format which are tuned for texture mapping in renderings. It allows the renderer to access portions of the texture without having to load the whole image into memory at once.

The RAT file format also supports arbitrary channel depth, meaning that a single channel image can be used as a texture map.

BETTER MEMORY USAGE

Using .rat files is faster for texture mapping, and typically consumes less memory than other formats because the file format is “paged”. This means that portions of the map are flushed out if more data is required for rendering. Only the portions of the map required for rendering are loaded. By default, *mantra* allocates a maximum of 8 Mb of RAM for RAT files.

It is possible to specify the maximum amount of RAM (in Mb) used for texturing. The environment variable SESI_RAT_USAGE can be used to set to the maximum amount of RAM you want *mantra* to use. For example:

```
setenv SESI_RAT_USAGE 32
```

to allocate 32 Mb of RAM.

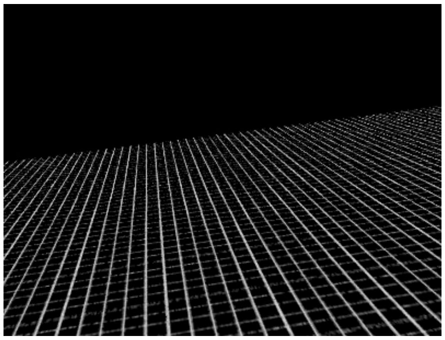
TEXTURE FILTERING

With RAT files, it is possible to specify an anti-aliasing filter which is used during texture mapping. It is also possible to specify a filter size which allows blurring of the texture map by the renderer.

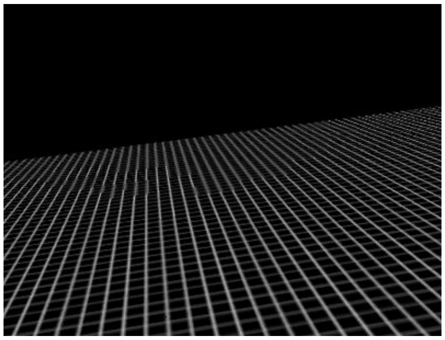
Note: Large blurs should not be done in the renderer.

RAT QUALITY BETTER THAN OTHER FORMATS

We recommend using .rat files for texture maps before any other format because the quality tends to be much better than using stochastic sampling. Compare:



Without RAT (Stochastic Dithering)



With RAT Textures

4.11 TARGA/VISTA IMAGES

The Targa and Vista file formats are treated identically. Support is included for:

- Image type 10 (RGB Run Length Encoded)
- Image type 2 (RGB Raw Data Stream)
- Data Bits 15, 16, 24 and 32

And all combinations of the above (i.e. Type 10, 16 bits per pixel).

4.12 TIFF IMAGES

The TIFF (Tagged Image File Format with suffix .tif) is the standard used by RenderMan and many Mac and PC applications.

When using a *.tif3* file extension, Houdini uses TIFF version 5.0 from mid-1990. Files saved in this format use LZW compression, RGB colour space, and use 32-bits per pixel (i.e. four channels per pixel – RGB and alpha, at eight bits per channel).

When using a plain *.tif* extension, Houdini uses a new TIFF library from early-1996 that is compatible with RenderMan 3.6.

Adobe-Deflate codecs are also supported.

4.13 VERTIGO IMAGE FORMAT

The preferred extension for Vertigo file format is *.vtg*. Since Vertigo chooses to name their images *.pic* by default, files with this extension are handled more intelligently now in Houdini.

When reading *.pic* files, the magic number is used to determine the file format, so Houdini *.pic* and Vertigo *.pic* files can co-exist.

When creating new files, Houdini files will be created by default. This can be overridden by setting the environment variable *VERTIGOPIC*. Put the following in your *.login* file:

```
setenv VERTIGOPIC
```

Compressed Vertigo files (.Z) cannot be read by the frame buffer library, nor can they be created. Use the UNIX *uncompress* program before using them in Houdini.

4.14 VIDEO FRAMER – VF

Houdini can be used with the SGI Video Framer to record animation and rendered images, and to scan external video images to and from sources such as Beta/SP YUV, RGB, digitally encoded/decoded NTSC, PAL, D1, D2, and S-Video.

All image tools and renderers can write to the Video Framer and read from the Video Framer directly. The device name is *vf*. This can be given as the output device of the renderers in place of the default *ip* (interactive window) device.

You can use *vf* in the image tools to copy from the Video framer to an IRIS window by typing:

```
  icp vf ip
```

You can write images to the Video Framer, as in the following example which loads SMPTE color bars:

```
  icp $HFS/houdini/pic/SMPTE.pic vf
```

The input images are always scaled to fill the full resolution of the Video Framer: 720×486 (NTSC) or 720×576 (PAL). This is the same as the Abekas device. Images are always fetched such that they all have square pixels (like the IRIS) and have full scan lines (640 NTSC and 576 PAL scan lines).

It takes about six seconds to read a YUV-encoded image from the Video Framer, and about three seconds to write an image to the Video Framer. Images are automatically flipped so they are top-to-bottom in the Video Framer, as they are in the Abekas A60.

By default, the Video Framer is set to read and write video in NTSC YUV format. To set it in the format you need, you require an environment variable, *VFMODE*. For example, put the following in your *.login*:

```
  setenv VFMODE vfr_r_y_525
```

All the legal modes are listed in the Video Framer manual.

If you read from the Video Framer in a video mode that is different from the mode in which you will write to it, you need two environment variables, *VFREADMODE* and *VFWRITEMODE*. For example, put the following in your *.login*:

```
setenv VFREADMODE vfr_rgb_525
setenv VFWRITEMODE vfr_r_y_525
```

Refer to the notes located in *$HFS/houdini/support/videoframer*.

VIDEO FRAMER AND VLAN CONTROL

At the same time that you load an image into the Video Framer you can trigger a video device on the VLAN (Video Local Area Network) to record the image.

It is possible for the VLAN device to be a tape device like Abekas A62, A66 and other video devices that have VLAN interfaces (call Video Media at: 408-227-9977, Fax: 408-227-6707).

You can specify which video frame by using the following syntax:

```
vf:hh:mm:ss:ff[,framenum][,nframes]
```

Where *hh* is the hours, *mm* is the minutes, *ss* is the seconds, and *ff* is the frames. This is the video time code where the first frame (or the only) frame is placed.

The following are examples of the syntax you may use when recording images:

```
vf:11:22:33:24
vf:11:22:33:24,61
vf:11:22:33:24,61,4
vf:11:22:33:24,,4
vf:11:22:33:24,\$F,2
```

The first records the image for 1 frame at time code 11 hours, 22 minutes, 33 seconds and 24 frames. The second example records the same image with an offset frame number of 61. Frame number 1 implies no offset (as Houdini frame 1 is the first frame), so this example will record at time code (assuming NTSC 30 frames per second) of 11:22:35:24, and is equivalent to:

```
vf:11:22:35:24
```

The next example records the same image at the same starting frame, but records four frames. This is useful when you are recording with a frame increment of two or more.

The fourth example is the same as the first but will record four frames.

The final example details the syntax that you use in shell scripts or in Houdini to use the current frame number *$F* as the offset on the tape. This is suitable for passing to the renderer. You will see that this is a choice in the Output Editor's Render Command dialog.

The following is an example which writes 150 consecutive frames of the animation in the Object Editor viewport to 150 frames of video tape:

```
picwrite -f 1 150 vf:01:00:00:00,\$F
```

This will record 150 frames starting at timecode 01:00:00:00.

If you have rendered on fields, then Houdini frame 1 is video frame 1, Houdini frame 3 is video frame 2, Houdini frame 5 is video frame 3, and so on. In order to record in this manner, put a "." between the time code values instead of ":". For example,

vf:11.22.33.24,\$F

If you want to record an image through a video framer on another IRIS, you can prefix it withthe IRIS name. For example,

```
irisB:vf:11:22:33:24
```

MULTI-ACCESS TO VIDEO FRAMER AND VLAN

Several processes on several machines can access the Video Framer to record images “simultaneously”. Because of a locking mechanism in the Video Framer device control, you can run several renders or composites on several machines, all referring to one video framer and even the same part of a video tape. Each attempted record will check the lock and will try for three minutes to gain access to record the image. Once the record is done, the device is freed for the next process that needs it. In this way, rendering, compositing, and recording can happen more asynchronously. Example:

```
machine irisA rendering 1 to 100 step 3
machine irisB rendering 2 to 100 step 3
machine irisC rendering 3 to 100 step 3
```

If they record to irisB’s Video Framer, they all need the same specification in the render dialog box:

```
irisB:vf:00:10:20:00,$F,1
```

The three minute timeout period is overridden with the VFRTIMEOUT variable, and the twenty minute period before an idle lock is broken with VFRFAILTIME, expressed in seconds.

4.15 WAVEFRONT IMAGES

Wavefront pictures are treated like other image file formats but have one additional feature. The gamma value placed in Wavefront .rla files defaults to 2.2, but can be overridden by setting the environment variable *WFGAMMA* to the desired value. Do this in a c-shell or in your .login script: *setenv WFGAMMA 2.2*.

5 ADDING OTHER IMAGE FILE FORMATS

5.1 FBIO TABLE

You can easily add additional file formats to be recognised by Side Effects programs. Two files located in: *$HFS/houdini* must be modified in order to do this:

FBFILES

Contains a list of valid image file format extensions that Houdini will recognise. In order for a new format to be recognised, it must be added to the bottom of the list in the file: *FBfiles*.

FBIO

Contains the commands used to convert to/from any of the image formats. In order for a new format to be converted transparently by Houdini, you must add the commands which will convert from the new file format to a Side Effects image format; and a command which will convert from a Side Effects image format to the new file format here. This is done in the file: *FBio*.

PROCEDURE

The procedure for adding a new format is:

1. Determine an unused extension (e.g. .xwd).
2. Put the programs to read and write the image format in your search path.
3. Add the new format's extension to: *$HFS/houdini/FBfiles* .
4. Come up with a set of commands to read and write the image format to Houdini on *stdin* and *stdout*, and add these commands to the *$HFS/houdini/FBio* table.

Note: Because files in *$HFS/houdini* are generally not to be modified, you should copy these files to a sub-directory of your home directory: *$HOME/houdini* , and modify them from there.

EXAMPLE

The file format *.xwd* is used by Adobe's FrameMaker for X-windows images. Adobe provides a program to convert this file format into SGI format with the command *fromxwd*, and the program *tiff2frame* to convert from a *.tif* format to a *.xwd* format. To add support for the *.xwd* format to Houdini, we do these steps:

1. We make a copy of *$HFS/houdini/FBfiles* and *$HFS/houdini/FBio* into our home directory: *$HOME/houdini/FBfiles* and *$HOME/houdini/FBio*.

2. If necessary, we add the location of the *fromxwd* and *tiff2frame* programs to our search path by updating our *.login* or *.local* files appropriately.

3. We add the following line to the end of *FBfiles* with an editor like *vi* or *jot*:

```
xwd
```

4. Then we add the following (single) line to the end of the *FBio* file. The first part of it declares the entry in the FBio table (.xwd); the second part contains the commands neccesary to read .xwd files enclosed in quotes; and the third part contains the commands necessary to write .xwd files – also enclosed in quotes:

```
.xwd
"fromxwd %s /tmp/x.sgi ; icp /tmp/x.sgi stdout ; rm /tmp/x.sgi"
"icp stdin /tmp/x.tif3 ; tiff2frame /tmp/x.tif3 %s ; rm /tmp/x.tif3"
```

After this, any Houdini program or standalone program (such as *icp*) can be used to create *.xwd* files simply by using the *.xwd* extension in the output file name. The first set of commands in quotes converts *from* the format; the second set converts *to* it.

In a similar manner, Side Effects programs are able to automatically get an image in any given format by simply adding the necessary commands to these files.

6 FORMAT OF HOUDINI IMAGES

6.1 GENERAL DESCRIPTION

Houdini image files are run-length encoded disk files with the suffix *.pic*. The files contain color and color map information as well as the size of the picture. The picture files can be saved and restored using the image copy command *icp*.

6.2 LOCATION OF SOURCE CODE

The source for reading and writing Houdini image files is in:

$HFS/houdini/public/sidefx.Pic.tar.Z

7 FORMAT OF HOUDINI MOVIE (.HMV) FILES

7.1 HOUDINI MOVIE FILE FORMAT (VERSION 1)

HEADER

All non-ASCII fields (i.e. integer or floating point) are interpreted relative to the indicated byte order (i.e. either MIPS or Intel).

- 4 bytes The ASCII bytes ".hmv".

- 4 bytes Integer 0x0000 for MIPS byte ordering 0xFFFF for Intel byte ordering (currently not implemented).

- 4 bytes Integer version number of HMV format. Current version is 0x0001 (version 1)

- 4 bytes Integer blocking factor. Only 4096 supported for version 1.

- 4 bytes Integer X resolution.
  4 bytes Integer Y resolution.

- 4 bytes Integer frame number of first frame.
  4 bytes Integer frame increment.
  4 bytes Integer frame number of last frame.

- 32 bytes Image format. 32 bytes, null terminated ASCII string:
  - "R8G8B8A8" RGBA 8 bit format. The order of image bytes on disk is: alpha, blue, green and red.
  - "R8G8B8" RGB 8 bit format. The order of image bytes on disk is: blue, green and red.
  - "B8G8R8" RGB 8 bit format. The order of image bytes on disk is: red, green and blue.
  - "Y8U8V8" YUV format. Every two image pixels are represented by: U, Y1, V and Y2 (i.e. two image pixels require four bytes of disk space). The image width must therefore be an even number of pixels.

- 4 bytes Field format. Integer 0x0000 for full frame images and 0x0001 for interlaced (i.e. 2 fields per image). 0x0001 is currently not implemented.

- 4 bytes 32 bit IEEE floating point pixel aspect ratio. A value of 0 is taken to mean 1.0. Pixel aspect ratio is determined as the horizontal to vertical ratio (H/V). (Currently not implemented).

- 4 bytes 32 bit IEEE floating point frames-per-second display rate. 0 is taken to mean 30 FPS. (Currently not implemented).

FRAMES

- Each image is stored on the next available “block size” boundary. For a blocking factor of 4096, the first image will therefore start on byte number 4097, assuming that bytes are numbered from 1.
- The file must be padded out to fill any remaining unused space at the end of the last block.

7.2 SEE ALSO

- *Outputs > Houdini Movie (HMV) Output OP* p. 738
- *StandAlone > hmvplay – Play Houdini Movies* p. 403

2 .geo File Format Description

1 FILE FORMAT

The .geo file format, which is an ASCII file, and the binary version of the .geo format (.bgeo) are the standard formats used to store Houdini geometry. The .geo format stores all the information contained in the Houdini geo-detail. This format is publically available to read and write Houdini geometry files.

1.1 HEADER SECTION

```
Magic Number:        PGEOMETRY
Point/Prim Counts:   NPoints # NPrims #
Group Counts:        NPointGroups # NPrimGroups #
Attribute Counts:    NPointAttrib # NVertexAttrib #
                     NPrimAttrib # NAttrib #
```

In each of these cases, the # represents the number of the element described. Groups are named and may be defined to contain either points or primitives. Each point or primitive can be a member of any number of groups, thus membership is not exclusive to one group.

Attributes in GPD have been generalized. Attributes can be assigned per point, per vertex, per primitive or on the detail. Therefore, the number of attributes is declared at the top of the file. Later, each of these attributes will be defined in full.

1.2 ATTRIBUTE DEFINITIONS

Internally, there are "dictionaries" to define the attributes associated with each element. These dictionaries define the name of the attribute, the type of the attribute and the size of the attribute. Also, the default value of the attribute is stored in the dictionary.

When the dictionary is saved, each attribute (in a specific order) is defined. The definition is basically as follows:

```
Name Size Type Default
```

For example, the attribute name for normals is "N", so the attribute definition would look like:

```
N 3 float 0 0 0
```

Specifying the attribute name “N”, that there are three elements in this attribute and the type is float. The default value would be (0, 0, 0)

Following the element definition is the attribute data associated with the element. There are braces delineating the attribute data. The attribute data appears in the order that the dictionary for the element was defined.

For example, a dictionary might look like:

```
PointAttrib
Cd 3 float 0 0 0     # Color attrib., 3 floats, default 0 0 0
Alpha 1 float 1      # Alpha attribute, 1 float, default 1
N 3 float 0 0 0      # Normal attribute
uv 2 float 0 0       # Texture coordinate
```

The data for the point might look like:

```
0 0 0 1              (1 0 0  1  0 0 1  .5 .5)
^^^^^^^              ^^^^^^^^^^^^^^^^^^^^^^^
Position             Attributes
```

The point would have:

```
Cd =                 (1, 0, 0)
Alpha =              1
N =                  (0 0 1)
uv =                 (.5, .5)
```

The types of attribute data supported are: integer, float, string and index. The “string” type is stored as a 32 character string since each attribute must have a fixed length. The integer and float types are pretty self-explanatory. The index attribute type is used for specifying things like material. It contains a list of strings which are indexed by integer values. Thus the storage for an index attribute is an integer. In the definition of the index attribute, the attribute values are defined as well.

```
mat 1 index 3 marble gold crystal_glass3
```

The default value for all index attributes is -1 indicating that the attribute has not been assigned.

1.3 POINT DEFINITIONS

If there are point attributes, the attribute dictionary is saved before the definition of the points.

```
Dictionary Name:     PointAttrib
Dictionary Data:     -- Attribute Definition --
```

Following the attribute dictionary, is the point data for the points. Each point is stored with four components (x, y, z & w). The positions are not true homogeneous coordinates. To get the homogeneous coordinate, simply multiply each x, y and z by w.

If (and only if) there is attribute data, the attribute data is defined following the point position. The attribute data is enclosed in parenthesis “()”.

1.4 PRIMITIVE / VERTEX DEFINITIONS

If (and only if) there are vertex attributes, the attribute dictionary is found here.

Following the vertex attribute dictionary is the primitive attribute dictionary (iff there are attributes for primitives).

Since every primitive may have local information which needs to be saved, the format of every primitive is different. In general, the format is:

```
PrimKey <local_information> [attributes]
```

Here, the local_information is primitive specific.

As part of the local information, a vertex or multiple vertices are specified. Each vertex is saved in the same format, which is:

```
point_number attribute_data
```

The point numbers start at 0 and go through NPoints - 1. If there is vertex attribute data, the data is delimited by parenthesis “()”. If there is primitive attribute data, it is delimited by brackets “[]”.

Each primitive has a unique identifier. The current primitives and their identifiers are:

```
Polygon:                 “Poly”
NURBS Curve:             “NURBCurve”
Rational Bezier Curve:  “BezierCurve”
Linear Patch:            “Mesh”
NURBS Surface            “NURBMesh”
Rational Bezier Patch:  “BezierMesh”
Ellipse/Circle:          “Circle”
Ellipsoid/Sphere:        “Sphere”
Tube/Cone:               “Tube”
Metaball                 “MetaBall”
Meta Super-Quadric       “MetaSQuad”
Particle System:         “Part”
Paste Hierarchy          “PasteSurf”
```

The primitive keys are case sensitive. For example:

```
VertexAttrib
uv 3 float 0 0 0
PrimitiveAttrib
Cd 3 float 0 0 0
Poly 3 < 0 (1 0.5 0) 1 (0 0 0) 2 (0 1 0) [1 1 0 .5]
```

Would specify a closed polygon (see below) which has three vertices referencing points 0, 1 & 2. Each vertex has 3D texture coordinates specified in (), the polygon has Color and Alpha specified in []. The color is yellow, with 50% alpha coverage.

When there are two or more consecutive primitives of the same type, this is specified as a run of primitives. In this case, the following should appear in the file:

```
Run # PrimKey
```

Where # is the number of primitives in the run. In this case, the following primitives are not saved with the PrimKey identifier since it is implicit in the run.

2 LOCAL PRIMITIVE INFORMATION

2.1 POLYGON LOCAL INFORMATION FORMAT

`#Vtx OpenClose Vertex_List`

`#Vtx`	Number of vertices in the polygon
`OpenClose`	A single character flag: “ < ” = Closed face “ : ” = Open face

2.2 NURBS / BEZIER CURVE LOCAL INFORMATION FORMAT

`#Vtx OpenClose Basis Vertex_List`

The basis definition for both NURBS and Bezier primitives starts with:

`Keyword Order`

Where:

`Keyword`	“Basis”
`Order`	The order of the basis (degree + 1)

THE NURBS BASIS

The NURBS basis requires an end condition flag and a list of knots sorted in increasing order. The complete definition of the NURBS basis is:

`Keyword Order EndCondition Knots`

Where:

`EndCondition`	“end” to touch the end CVs, “noend” otherwise.
`Knots`	Floating point numbers in increasing order.

The number of knots in the list is determined by the order of the basis, its end conditions, the number of CVs in the Vertex_List, and the OpenClose flag.

Let #K be the number of expected knots, and #Vtx the number of CVs. Then, if the EndCondition is false (i.e. “noend”).

`#K = #Vtx + Order - 2`

The two missing end knots (and the periodicity knots if closed) are generated internally. If theEndCondition is true (i.e. “end), then:

if the curve is open	#K = #Vtx - Order + 2
if the curve is closed	#K = #Vtx - Order + 3

THE BEZIER BASIS

The Bezier basis does not require a list of knots if the knots start at 0 and grow with unit increments (e.g. 0 1 2 3 ...) The complete definition of the Bezier basis is:

```
Keyword Order Knots
```

The number of knots in the list is determined by the order of the basis, the number of CVs in the Vertex_List, and the OpenClose flag.

Let #K be the number of expected knots, #Vtx the number of CVs. Then:

if the curve is open	#K = (#Vtx-1) / (Order-1) + 1
if the curve is closed	#K = (#Vtx) / (Order-1)

If the curve is closed, the periodicity knot is generated internally.

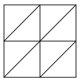
2.3 MESH LOCAL INFORMATION FORMAT

```
#Cols #Rows UWrap VWrap connectivity
```

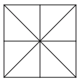
`UWrap / VWrap`	“open” or “wrap” columns or rows respectively
`connectivity`	“rows” – Rows only “cols” – Columns only “rowcol” – Rows & Columns “quad” – Quads “tri” – Triangulated quads “atri” – Alternate triangulated

The connectivity is ignored in many cases, but is critical for operations like sweeping or conversion to polygons.

Triangulated and Alternate meshes are structured like:



Triangulated (tri)
Mesh (default)



Alternate (atri)
Mesh

2.4 NURBS / BEZIER SURFACE LOCAL INFORMATION FORMAT

```
#Cols #Rows UWrap VWrap connectivity UBasis VBasis
                       Vertex_List Profiles
```

`#Cols, #Rows, UWrap, VWrap, connectivity, Vertex_List`	are the same as for Mesh.
`UBasis / VBasis`	are the same as for NURBS / Bezier Curve.

`Profiles`	is an optional list of profile curves (curves on surfaces). The structure of the profiles section is very similar to that of the main geometry, including a header section, points, primitives, point and primitive groups. The differences are that this section doesn't contain any attributes and has only four primitive types: polygon, NURBS curve, Bezier curve, and Trim Sequence. The profile header is "Profiles:". It is followed by "none" if there are no profiles. If there are profiles, the profile section has the following structure:
`Point/Prim Counts`	NPoints # NPrims #
`Group Counts`	NPointGroups # NPrimGroups #
`TrimLevel #`	# is a number representing the sea-level for nested trimmed loops, and can be either positive or negative. Usually it is 0.
`Point list`	u v w triplets
`Primitive list`	polygons, NURBS/Bezier curves, trim sequences
`Point groups`	Point group definitions
`Prim. groups`	Primitive group definitions

HEADER SECTION

`Point/Prim Counts:`	NPoints # NPrims # NLoops #
`Group Counts:`	NPointGroups # NPrimGroups #

In each of these cases, the # represents the number of the element described.

`Nested trim level:`	TrimLevel #

In this case, # represents the sea-level for nested trimmed loops, and can be either positive or negative. Usually it is 0.

A primitive is a 2D profile: a polygon, a Bezier curve, or a NURBS curve living within the domain of the spline surface. The points are 2D locations (i.e. UV pairs with a third, W (weight) component) in the surface domain.

The loops are trimming loops, also know as "trim regions", defined by the primitive profiles mentioned above. It is possible to have several profiles on a surface and yet no trim loops.

Groups are named and may be defined to contain either points or profiles. Each point or primitive can be a member of any number of groups, thus membership is not exclusive to one group.

POINT SECTION

Each point is stored with 3 components (x, y, w). The positions are not true homogeneous coordinates. To get the homogeneous coordinates, simply multiply each x, y by w.

PRIMITIVE SECTION

Since every profile may have local information which needs to be saved, the format of every primitive is different. In general, the format is:

```
ProfileKey <local_information>
```

Here, the local_information is profile specific.

As part of the local information, a vertex or multiple vertices are specified. Each vertex is saved in the same format, which is:

```
point_number
```

The point numbers start at 0 and go through NPoints - 1.

Each profile has a unique identifier. The current profiles and their identifiers are identical to their 3D counterparts:

```
Polygon:               "Poly"
NURBS Curve:           "NURBCurve"
Rational Bezier Curve:"BezierCurve"
```

The profile keys are case sensitive.
For example:

```
Poly 3 < 0 1 2
```

Specifies a closed polygon (see below) which has 3 vertices referencing 2D points 0, 1 & 2.

When there are two or more consecutive profiles of the same type, this is specified as a run of profiles. In this case, the following should appear in the file:

```
Run # ProfileKey
```

Where # is the number of profiles in the run. In this case, the following profiles are not saved with the ProfileKey identifier since it is implicit in the run.

The format of the three profile types - polygon, NURBS curve, and Bezier curve – is identical to that of the 3D primitives and won’t be listed again here.

TRIMMING SECTION

If NLoops is not zero, the surface will contain one or more trim regions. Each region can contain one or more profiles.

Typically, the profiles should intersect to form a closed loop. Sometimes, though, as in the case of a loop that intersects the domain boundaries, the loop is partially defined by the domain boundaries and need not be explicitly closed.

Single profile loops that are open and do not intersect the domain boundaries will be closed straight by Houdini.

The trimming section contains one or more lines like the one below, one line per trim region:

`TrimRegion [natural] #Profiles <profile_number ustart uend>...`

If “natural” is specified, open profiles are treated casually, i.e. their parametric direction is not checked and will not be reversed.

`profile_number`	is the index of each profile in the current trim region.
`ustart and uend`	are the parametric values defining the beginning and end of the profile. It is thus possible to use only a section of a profile for trimming.

To reverse the direction of the trim curve without reversing the vertices of the profile itself, specify a ustart greater than ustop. A profile can therefore be used in more than one trim region, and can have different orientations and lengths in each region.

When punching holes in a surface, an outer profile is needed to specify the area of the surface to be kept. Usually, the outer profile is a closed polygon that envelops the perimeter of the domain.

Example:

```
  TrimRegion 2   0  1 0   5 -3.5 8
```

The trim region has two profiles: 0 and 5. Profile 0 is reversed by evaluating between 1 and 0. Profile 5 is used between -3.5 and 8.

GROUPS SECTION

The point groups are saved first, followed by the profile groups. There is no identifier indicating the groups. The format for a group depends on whether it is ordered or unordered:

`GroupName Type NElements BitMask ElementList`

`GroupName`	is the name of the group.
`Type`	is "unordered" or "ordered".
`NElements`	Specifies the total number of bits in the BitMask. This is equivalent to NPrims in the profile header.
`BitMask`	A string of 0’s and 1’s, where 1 indicates inclusion in the group.
`ElementList`	If the groups is ordered, the element list contains the index of each selected point or profile in selection order. The first element of the list is the number of ordered elements in the list.

2.5 PASTE HIERARCHY LOCAL INFORMATION FORMAT

`#Features`	followed by as many lines as feature surfaces, in the order in which the surfaces are pasted. Each feature line has the format:

```
Feature prim_number height up_or_down <domain_xform>
```

`prim_number`	is the index of the spline surface in the list of primitives. height is the elevation of the pasted surface from its base.
`up_or_down`	is 1 is pasted upward, 0 if downward.

The domain transformation is either linear or bilinear.

LINEAR TRANSFORMATION FORMAT

```
Linear tx ty
UT_Matrix2 m00 m01 m10 m11
```

The translation in the domain is given by (tx,ty). The rotation and scaling components are captured in the 2×2 matrix.

BILINEAR TRANSFORMATION FORMAT

```
Bilinear origUL origUR origLR origLL
  warpUL warpUR warpLR warpLL
```

`L,U,L,R`	stand for Lower, Upper, Left and Right respectively. Each of the eight locations is a (u,v) pair in the surface domain.

Example of a paste hierarchy with three surfaces:

```
PasteSurf 3
Feature 0  0 1
Linear 0 0
UT_Matrix2 1 0 0 1
Feature 2  0.02 1
Bilinear
0 0.6
0.6 0.6
0.6 0
0 0
100.1 -22
100.4 -22
100.4 -28
100.1 -28
```

```
Feature 3  0.07 0
Bilinear
0 1
1 1
1 0
0 0
100.2 -21
100.45 -21
100.45 -26
100.2 -26
```

2.6 CIRCLE LOCAL INFORMATION FORMAT

`Vertex_Info Matrix33`

There is always only one vertex for a Circle. The 3×3 matrix contains scaling and rotation transformations about the center of the circle. Sheared circles are thus allowed.

2.7 SPHERE LOCAL INFORMATION FORMAT

`Vertex_Info Matrix33`

There is always only one vertex for a Sphere. The 3×3 matrix contains scaling and rotation transformations about the center of the sphere. Sheared spheres are thus allowed.

2.8 TUBE / CONE LOCAL INFORMATION FORMAT

`Vertex_Info Taper Closure Matrix33`

There is always only one vertex for a Tube/Cone. The vertex lies in the center of the tube (along the axis connecting the centers of the top and bottom circles/ellipses). The taper value affects the radius of the top circle. A regular tube has a taper value of 1. A cone’s taper is 0. The closure - “closed” or “open” – indicates whether the tube is end-capped. The 3×3 matrix contains scaling and rotation transformations about the center of the tube. Sheared tubes are thus allowed.

2.9 METABALL LOCAL INFORMATION FORMAT

`Vertex_Info Kernel_Function Weight Matrix33`

There is always only one vertex for a metaball. The kernel function is one of: “wyvill”, “quartic”, “blinn” or “links”. The 3×3 matrix contains scaling and rotation transformations about the center of the metaball. Sheared metaballs are thus allowed.

2.10 META SUPER-QUADRIC LOCAL INFORMATION FORMAT

```
Vertex_Info XY_Exponent Z_Exponent Kernel_Function
                        Weight Matrix33
```

There is always only one vertex for a meta super-quadric. The exponents are float values. The kernel function is one of: “wyvill”, “quartic”, “blinn” or “links”. The 3×3 matrix contains scaling and rotation transformations about the center of the metaball. Sheared metaballs are thus allowed.

2.11 PARTICLE SYSTEM LOCAL INFORMATION FORMAT

```
Part_Count Vertex_List
```

Where *Part_Count* is the number of particles in the system.

3 DETAIL & POINT/PRIM DEFINITIONS

3.1 DETAIL ATTRIBUTES

The Detail Attribute Dictionary is saved after the Primitives and before the group information.

3.2 POINT / PRIMITIVE GROUP DEFINITIONS

The Point groups are saved first, followed by the primitive groups. There is no identifier indicating the groups. The format for a group depends on whether it is ordered or unordered:

```
GroupName Type NElements BitMask ElementList
```

`GroupName`	is the name of the group.
`Type`	is “unordered” or “ordered”.
`NElements`	Specifies the total number of bits in the BitMask. This is equivalent to the number of elements in the detail.
`BitMask`	In the ASCII format, this is a string of 0’s and 1’s, where 1 indicates membership in the group.
`ElementList`	If the groups is ordered, the element list contains the index of each selected point or primitive in selection order. The first element of the list is the number of ordered elements in the list. In the case of primitive lists a second profile element may be described by appending a period and a secondary index number to each element. For example, 5 specifies the fifth primitive while 5.12 specifies the twelth profile curve of the fifth primitive. The list must be empty if the group is unordered.

3.3 OTHER INFORMATION

This is meant for saving information such as metaball expressions and surface hierarchies. Currently this section contains only the delimiting tokens, one per line:

```
beginExtra
endExtra
```

For now the Extra body is empty because all the metaballs are merged (“add”-ed implicitly) and there is no support for surface hierarchies.