I Scripting

This section covers Houdini's Scripting Language.

I INTRODUCTION

Much of what you do in Houdini's graphic interface can also be done using a text-based scripting language. These keyboard commands can also be used in sequence as a scripting language to create macros, or even to edit saved *.hip* files. You can alternate between text and graphic commands. Text commands can be edited in scripts and executed by Houdini. You can minimize typing by using abbreviations, aliases, variables and script macros.

There are several places where the scripting language can be accessed. The Textport in Houdini allows you to type commands directly and see the output immediately. The Operator Macros allows pre-defined scripts to be executed with a graphical interface to set their parameters. The stand-alone application *hscript* provides a non-graphical version of Houdini.

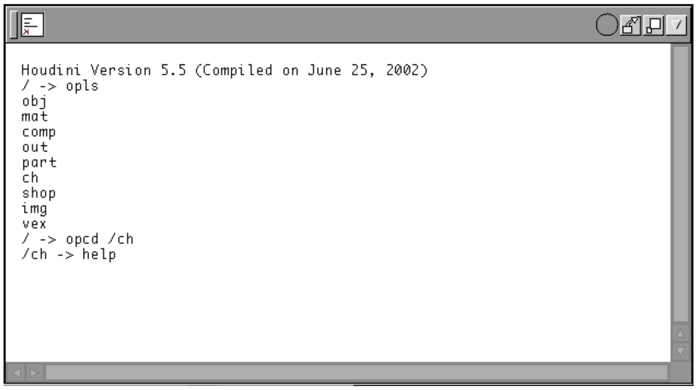
Houdini uses "type-ahead" which allows you to type commands before Houdini has finished executing the current text or graphic command.

The text command facilities enable you to work more efficiently because some complex operations can be achieved with a few keystrokes rather than many button clicks and cursor movements. For example, to add 1000 objects and lay them out in a grid can be completed with less than 10 lines of script.

For more information on the structure and uses of Houdini's scripting language, see the *User Guide*'s chapter on *Scripting*.

I.I OPENING A TEXTPORT

You can open a Textport by selecting the *Textport* from any pane *Types* menu.



```
Houdini Version 5.5 (Compiled on June 25, 2002)
/ -> opls
obj
mat
comp
out
part
ch
shop
img
vex
/ -> opcd /ch
/ch -> help
```

1.2 NAVIGATING WITHIN THE TEXTPORT

KEYBOARD SHORTCUTS

🖱	Select text. Selected text becomes red. You can only select a single line of text at a time.
🖱	Paste text.
🖱	Scroll Textport. Mouse pointer changes into a hand cursor which allows you to pan the text area in much the same way as you can pan the Layout area.
PgUp / PgDn	Scroll text up / down, one page at a time.
Home / End	Returns you to the beginning / end of the Textport entries (maximum of 2000 lines).

COPYING AND PASTING

copying

Select text within the Textport by dragging across the text with the left mouse (🖱). The text becomes highlighted in red and is copied to the clipboard as soon as you release the mouse button. You can copy only one line of text at a time.

pasting

Text from the clipboard (i.e. the last text that was higlighted) can be pasted by clicking within the Textport with the middle mouse (🖱).

2 THE SCRIPTING LANGUAGE

2.1 ORDER OF EXPANSION

Expansion of a Houdini command follows the C shell expansion standards very closely. There are some subtle differences.

LIMITATIONS

- The maximum line length for a Houdini command is 8 Kb (expanded)
- The maximum number of arguments on a command line is 1024
- The maximum number of number of nested *if* statements in a single source file is 128
- The maximum number of source files is limited by the system limit on open files
- There is no limit for nested loops

LEXICAL STRUCTURE

Houdini splits input lines into words at space characters, except as noted below. The characters ; < > ( ) = form separate words and Houdini will insert spaces around these characters except as noted below. By preceding a special character by a backslash (\), its special meaning can be suppressed.

evaluation of quotes

Strings enclosed in a matched pair of quotes forms a partial word. Within double quotes (“), expansion will occur. Within single quotes (‘) expansion will not be done. Within back-quotes (`) the enclosed string will be evaluated as a Houdini expression and the result will be considered to be a partial word. Unlike csh, inside a matched pair of quotes, the quote character may be protected by preceding the slash with a back-slash.

Back-quotes are evalutated with a higher priority than double quotes. This means that if a double-quoted argument encloses a back-quoted string, the back-quoted string may contain double quotes without terminating the initial double quote delimiter. For example, the string:

```
"foo‘ch("/obj/geo1/tx")‘"
```

will be parsed as a single argument.

Note: As a general rule, do not include spaces between your “back” quotation marks and what lies between them. Houdini may not evaluate them if there are extra spaces.

comments

The character # introduces a comment which continues to the end of the line. This character can be protected by a back-slash (\) or by enclosing the character in quotes.

COMMAND STRUCTURE

The output of a Houdini command can be redirected to a UNIX file by using the meta-character >. The output can be appended to a UNIX file by using >>. To redirect the error output and the standard output of a command to a UNIX file, the character sequence >& can be used.

Multiple commands can be specified on the same line by separating them with semicolons (;).

EXPANSION

Expansion is done in the following order: History substitution, Alias expansion, Variable & Expression expansion.

History substitution is not as sophisticated as the csh history mechanism. The supported substitutions are:

`!!`	Repeat last command
`!str`	Repeat last command matching str
`!num`	Repeat command “num” from the history list
`!-5`	Repeat the command run five commands previous

With the !! substitution, characters following the !! are appended to the command.

The resulting command is displayed in the Textport before the command is run.

Alias expansion is also not as sophisticated as csh. For example, one current limitation is that there is no recursive alias expansion. For example, the following sequence of commands will not produce the expected result:

```
  houdini -> alias opcd opcf
  houdini -> alias cd opcd
```

The cd alias will result in an unknown command “opcd” since the alias expansion terminates after the first expansion. As well, alias expansion does not include the history meta-character substitution that csh supports.

Variable and expression evaluation are done at the same time and have equal precedence. Variables are delimited by a dollar sign ($) followed by the variable name. A variable name must begin with letter or an underscore (_) followed by any number of letters, numbers or underscores. As well, the variable name may be delimited by curly braces ({}) in which case the contents of the curly braces is expanded before the variable name is resolved. This allows for pseudo array operations on variables. For example:

```
houdini -> set foo1 = bob
houdini -> set foo2 = sue
houdini -> for i = 1 to 2
> echo ${foo${i}}
> end
bob
sue
```

Expression evaluation is done on the string contained within matching back-quotes (‘). Inside the back-quotes, expression expansion is performed as opposed to command line expansion. The expression is evaluated, and the resulting string is used to replace the expression on the command line. If the expression evaluates to a type other than a string, the result is cast to a string and this is the value used.

COMMAND EXPRESSIONS

These differ from general Houdini expressions, though Houdini expressions can be used inside of command expressions. The command expressions are used in the “while” and “if” commands. The order of operations in a command expression is as follows:

`( )`	Parentheses
`== != < > <= >=`	Equal, Not Equal, Less Than, Greater Than, Less Than or Equal, Greater Than or Equal
`&& ||`	Logical And and Logical Or

Expressions can be enclosed in parentheses for clarity, but this is not necessary.

2.2 VARIABLES

There are two types of variables in Houdini, local variables and system or global variables. Local variables are local to Houdini (or the script being executed). When the script terminates, these variables will automatically be unset. Global variables will remain in the scope of all scripts and also to any UNIX programs started by Houdini. The “set” command will create local variables, while the *setenv* command will create global variables. For example:

```
houdini -> setenv agent = 99
houdini -> set local_agent = 45
houdini -> echo Agent $agent, this is agent $local_agent\n
Agent 99, this is agent 45
houdini -> unix echo ’Agent $agent, this is agent $local_agent’
local_agent - Undefined variable
```

Note, the single quotes prevent Houdini from expanding the contents of the command (see order of expansion).

All variables created by loops are considered local variables (i.e. the “for” loop will use local variables).

2.3 PATTERN MATCHING

Many of the *op* and *ch* commands allow patterns to specify multiple objects or channels. These patterns allow wildcards which will match all or parts of strings.

`*`	Match any sequence of characters
`?`	Match any single character
`[set]`	Match any characters enclosed in the square brackets. In Houdini, the [a-g] format is not currently supported, the characters must be listed.
`@ [group name]`	Expands all the items in the group. Since each group belongs to a network, you can specify a path before the @group identifier.

EXAMPLES

```
opcf /obj ; opls -d @geo
```

This example lists all the objects in the group named “geo”

```
opcf /obj/geo1; chadd @xform_sops tx ty tz
```

This example adds channels (*tx*, *ty*, and *tz*) to all the SOPs contained in the *xform_sops* group.

```
opcf /mat ; opls /obj/@lights
```

This example shows how to reference groups outside of the current folder.

See also: *Expression Language > Pattern Matching* p. 38.

2.4 COMMAND LOOPS

There are three different looping constructs in the Houdini scripting language:

for loop

```
for variable = start to end [step increment]
  ...
end
```

foreach loop

```
foreach variable (element_list)
  ...
end
```

while loop

```
while (expression)
  ...
end
```

The *for* loop will loop from the *start*, up to and including the *end*. *foreach* will cycle through every element in the *element_list* assigning the variable value to be a different element each iteration through the loop.

All variables in the *for* and *foreach* loops are local variables. To export the variable to other scripts (or to UNIX commands), simply set a global variable using *setenv* inside the loop. See *for* p. 112, *foreach* p. 112, and *while* p. 120.

EXAMPLE

You can use a loop to perform repetitive tasks for you. For example, if you wanted to wanted to merge 255 SOPs, it would be faster to write a short script than to do all that wiring manually. For example, if you named your SOPs consistently, like:

```
model-0, model-1, model-2... model-255
```

then you could execute the following script in a Textport:

```
for i = 0 to 255
  opwire model-$i -$i merge1
end
```

If you haven't been consistent with naming, you could also do it with a *foreach* .

2.5 CONDITIONAL STATEMENTS

The "if" command provides the ability for a script to check a condition and then execute one set of commands if the condition is true or an alternate set of commands if the condition is false. It should have an *endif* to signify the end. See *if* p. 114.

```
if ( expr ) [then]
     ...
  else if (expr2) [then]
     ...
  else
     ...
endif
```

2.6 ALIASES AND MULTIPLE COMMANDS

Some frequently used commands can be represented with a single word, an alias. For example:

```
houdini-> alias greet echo hello world
houdini-> greet
hello world
houdini-> alias mine "opset -d off * ; opset -d on geo1"
houdini-> mine
```

This will execute the string attached to the alias "mine" and turn off the display of all the objects then turn on object *geo1*.

The next two commands list, then undefine, an alias:

```
houdini-> alias
greet hello world
mine   opset -d off * ; opset -d on geo1
houdini-> alias -u greet
```

Houdini accepts several commands on the same command line separated by a semicolon. This does not apply to semicolons embedded in quotes. Aliases can contain commands embedded in quotes.

Note: Alias expansion is not performed if the local variable *noalias* is set.

2.7 USING ARGUMENTS IN SCRIPTS

The *source* command, when entered at the c-shell prompt, can have arguments after the *.cmd* file name. These arguments are set to Houdini variables so that they can be used by the script. For example:

```
houdini-> source repeat.cmd 1 10 2 blockhead
```

where *repeat.cmd* contains the Houdini script,

```
echo Hello, my name is $arg4
for i = $arg1 to $arg2 step $arg3
  echo I said, my name is $arg4
end
```

Note that there are four variables in the script: *arg1, arg2, arg3* and *arg4*. These are set to the *source* arguments *1, 10, 2* and *blockhead* respectively. This mechanism works well with the *-g* options of the *rcwrite* and *opdump* commands, which cause object names to be written out generically, as *$arg1, $arg2* and so on. In this way, names of objects can be changed when reading them as scripts.

$ARG0 – NAME OF THE SCRIPT

You can get the name of the script being run from *$arg0*. For example:

```
  source myscript.cmd 1 4.5 7 balloon
```

will come into the script with

```
$argc = 5
$arg0 = myscript.cmd
$arg1 = 1
$arg2 = 4.5
$arg3 = 7
$arg4 = balloon
```

This allows usages such as:

```
if $argc != 5 then
  echo USAGE: source $arg0 numclowns clownsize numtoys toytype
  exit
endif
```

$ARGC – NUMBER OF ARGUMENTS PASSED TO SCRIPT

The number of arguments passed to the script can be retrieved with the variable *$argc*, for example, from the *lookat.cmd* script:

```
# USAGE: lookat.cmd eyeobject focusobject
if $argc!= 2 then
 echo USAGE: source lookat.cmd eyeobject focusobject
 exit
endif
```

SHIFT COMMAND

In addition to using arguments, scripts can do very simple parsing of the command line using the *shift* command. Shift will shift the argument index one argument to the right. For example, the script for lattices sets the number of lattices (*NL*) to default to *3*, however, if the first argument passed to it is *-n* then the *NL* will be set to that argument; and the arguments shifted:

```
# lattice.cmd - builds a lattice deformation box around an object
set NL = 3
if "$arg1" == "-n" then
 shift
 set NL = $arg1
 shift
endif
...
```

Note that parsing occurs by shifting; this implies that the arguments *must be passed in a specific order.*

2.8 EXECUTING SCRIPTS

You can execute a list of commands located in a UNIX text file by running the *source* command. The following is fetched from the standard Houdini directory containing scripts, *$HH/scripts*

```
houdini-> source sixcreate.cmd
```

Normally, Houdini executes all the commands in any command file before redrawing the screen.

2.9 EXAMPLE SCRIPT

TEXTPORT EXAMPLE – WIRING OPS

You can use a loop to perform repetitive tasks for you. For example, if you wanted to wanted to merge 255 SOPs, it would be faster to write a short script than to do all that wiring manually. For example, if you named your SOPs consistently, like:

`model-0, model-1, model-2... model-255`

then you could execute the following script in a Textport:

```
for i = 0 to 255
  opwire model-$i -$i merge1
end
```

If you haven't been consistent with naming, you could also do it with a *foreach* .

GUESSING GAME

The following is a simple script which illustrates the use of loops, conditional execution, and variables. For more examples, see the *Scripting* section of the *User Guide.*

```
#  Houdini command script for the guessing game (guess.cmd)
#  First, let’s get a random seed
set foo = `system(date)`
set seed = `substr($foo, 14, 2)``substr($foo, 17, 2)`

# Then, pick a random number
set num = `int(rand($seed)*100)+1`
set guess = -1
echo Guess a random number between 1 and 100.
while ( “$guess” != “$num” )
  echo -n Enter guess (q to quit): “
  read guess

  if ( “$guess” == q || “$guess” == “”) then
      break;
  endif

  # Ensure they entered a number - i.e. convert to a number
  set iguess = `atof($guess)`
  if ( $iguess < $num ) then
      echo Too low
  else if ( $iguess > $num ) then
      echo Too high
  else
      echo Spot on!
  endif
end

# Come here if they selected “q” to quit:
echo The number to guess was $num
```

3 SCRIPTING WITH HSCRIPT AND THE C-SHELL

In many cases an animator or Technical director will not want to use the full graphical version of Houdini, but simply deal with the text version - *hscript* and use the C shell. The main advantage to this is the animator can write automated scripts to render sequences of frames without the need for an attendant animator.

Houdini's *hscript* allows you to accomplish virtually everything that you could with the full GUI interface, but with text-based commands. *hscript* also makes the transition from the GUI to the text version relatively easy with the *opscript* command.

This section assumes that you're relatively familiar with *hscript* and Houdini's textport commands and provides you with an explanation of how to incorporate *hscript* and C shell commands. The following discussion centers around writing render scripts, though much of this information can be used to write other kinds of scripts (such as adding in composite operations or file operations).

3.1 THE BASICS OF INCORPORATING C SHELL AND HSCRIPT

If you’ve used *hscript* within the confines of Houdini’s textport, then you’ll realize that *hscript* expects keyboard input. The trick to incorporating *hscript* and C shell is to redirect commands coming from the C shell so they appear to be keyboard input. Once this is done, commands can be sent to *hscript* as if you were typing the commands from the textport.

Following, is the basic form a script to do this takes:

```
#! /bin/csh -f

#add any standard C shell commands here.

hscript<<ENDCAT

#any text that follows here is redirected to hscript
#so enter hscript commands here

#stop processing hscript commands
ENDCAT

#add supplementary standard C shell commands
#end of shell script
```

3.2 SYMBOLS << AND >& EXPLAINED

The lines `hscript<<ENDCAT` and `ENDCAT` are significant. What does the `<< ENDCAT` mean? Simply put, anything between these lines is interpreted as *hscript* commands and any commands placed outside these lines is interpreted as standard C shell commands. Actually you don’t have to use `ENDCAT` – it could be any word that is not a reserved word in C shell or *hscript*. However it’s better to pick a standard and stick to what works.

Do not incorporate extra spaces or comments on the line that contains the* `ENDCAT` *terminator. If you do, you'll get an unknown variable error.

Alternatively, you might see (and use) the line `hscript<<ENDCAT >& renlog` or something similar. We've seen what `<< ENDCAT` means – now for the `>& renlog`.

The `>&` means any error messages that normally would appear on screen are instead written to a file. In this case the file is called `renlog`. You should note that `renlog` can be any file name.

3.3 EXAMPLES – RENDERING SCRIPTS

EXAMPLE 1 – BASIC RENDER

Looking at a very simple example, the following script works with two parameters– the *.hip* file to render and the file name to send the rendered image to. The script is named *simple_ren* and would be invoked as follows:

`basic_render` *`test.hip /usr/tmp/test_image.pic`*

source

```
#! /bin/csh -f

#check to see if user suppled the correct number of arguments
#- exit if not

if ($#argv < 2) then
  echo “USEAGE: ren_script <hip_file> <out_file>”
  exit
endif

# set up user supplied arguments
set HIP_FILE   = $1
set OUT_FILE   = $2

# start up hscript and allow commands to be sent from C shell

hscript << ENDCAT >& renlog
mread $HIP_FILE
opcd \/out

# set the ouput file name for the render output driver

opparm mantra1 picture ( $OUT_FILE )

#start the render

render mantra1

#signal an end to commands sent to hscript

ENDCAT
```

explanation

```
#! /bin/csh -f
```

This line is required to be at the beginning of each C shell script to denote that it contains commands that can be executed as a C shell script.

```
if (#$argv < 2) then
  echo “USEAGE: ren_script <hip_file> <out_file>”
  exit
endif
```

These lines check to see if you entered the correct number of arguments. The variable *#$argv* contains the number of parameters passed to the C shell script from the command line. For example, if you typed *basic_render test.hip*, the variable *#$argv* would be set to 1, and an error message would be generated because you omitted a destination image file as part of the command string.

```
set HIP_FILE   = $1
set OUT_FILE   = $2
```

These lines set the variables *$HIP_FILE* and *$OUT_FILE* to the parameters sent to the C shell from the command line. For instance, if you typed *basic_render test.hip /usr/tmp/test.pic*, *$1* and *$2* and, consequently, *$HIP_FILE* and *$OUT_FILE*, would be set to *test.hip* and */usr/tmp/test.pic* respectively.

```
hscript << ENDCAT >& renlog
```

Commands following this line will be interpreted as *hscript* commands up until it encounters the word `ENDCAT`. Also any error messages encountered in *hscript* will be sent to a file called *renlog*.

```
mread $HIP_FILE
```

Reads the *.hip* file specified by this variable.

```
opparm mantra1 picture ( $OUT_FILE )
```

Changes the default output paramter so the rendered image is sent to the file specified by *$OUT_FILE* (the default setting is ip).

```
render mantra1
```

Render the image using the *mantra1* output driver. Note that this case assumes that the output driver *mantra1* exists in the saved *.hip* file.

```
ENDCAT
```

This signals the C shell to stop redirecting commands to *hscript*. Don't insert extra spaces or comments or you will get the following error:

```
ENDCAT - << terminator not found
```

EXAMPLE 2 – RENDERING SEQUENCES OF FRAMES

```
#! /bin/csh -f

#check to see if user suppled the correct number of arguments
#- exit if not
```

```
if ($#argv < 4) then
  echo "USEAGE: ren_script <hip_file> <out_file> <start><end>"
  exit
endif

# set up user-supplied arguments

  set HIP_FILE   = $1
  set OUT_FILE   = $2
  set START      = $3
  set END        = $4

# start up hscript and allow commands to be sent from C shell

  hscript<<ENDCAT>&renlog
  mread $HIP_FILE

opcd \/out

# set the ouput file name for the render output driver

opparm mantra1 picture ("$OUT_FILE")

# turn on the frame range option and give it appropriate range.

opparm mantra1 trange ( on ) f ( $START $END 1 )

#start the render

render mantra1

# this signals and end to our commands to hscript

ENDCAT
```

EXAMPLE 3 – FOR – NEXT LOOP

```
#! /bin/csh -f

#check to see if user suppled the correct number of arguments
#- exit if not

if ($#argv < 4) then
  echo "USEAGE: ren_script <hip_file> <out_file> <start><end>"
  exit
endif

# set up user supplied arguments

set HIP_FILE   = $1
set OUT_FILE   = $2
set START      = $3
set END        = $4
```

```
# start up hscript and allow commands to be sent from C shell

hscript<<ENDCAT>&renlog
mread $HIP_FILE

opcd \/out

# set the ouput file name for the render output driver

opparm mantra1 picture ("$OUT_FILE")

opcook
for FRAME = $START to $END
  echo "Rendering frame
  fcur \$FRAME

  render mantra1
end

# this signals and end to our commands to hscript

ENDCAT
```

3.4 VARIABLE CAVEATS

Looking at the above listing, you might have noticed that some variables are prepended by a "\". This is to avoid the *unknown variable* message that occurs in C shell. We'll delve into this a little more deeply here. In short, prepending a backslash onto a variable name prevents C shell from expanding the variable before it gets to *hscript*.

From Houdini's textport you may do something like the following:

```
set hello_var = "Hello good citizen..."
echo $hello_var
```

or

```
for frame = 1 to 100 step 1
  echo "Rendering frame $frame...
end
```

This would work well. However, you will run into problems when you incorporate this same set of commands using C shell and *hscript*. For example:

```
hscript << ENDCAT
for frame = 1 to 100
  echo "Rendering frame $frame..."
end
ENDCAT
```

Running this script would give you the error: `frame - Undefined variable` In order to correct this you prepend a "\" to any variables you are manipulating (all those variable names that you would put a '$' in front of). The above example would be corrected as follows:

```
hscript << ENDCAT
for frame = 1 to 100
  echo "Rendering frame \$frame..."
end
ENDCAT
```

When the C shell encounters a *$* it tries to expand that variable into a value. However, you don't want C shell to do this, you want *hscript* to expand it instead. The backslash prevents the C shell from expanding the variable before it gets to *hscript*.

3.5 C SHELL SCRIPTING NOTES

There are several conventions to adhere to when writing shell scripts.

- make sure the first line in your script starts with `#! /bin/csh -f`
- make sure that the script is made executable with the *chmod()* function. For example `chmod 555 simple_ren` makes the script *simple_ren* readable and executable by everyone.
- watch what you put on the line containing the terminator *ENDCAT*.
- Be wary of of prepending variables declared in *hscript*, when running *hscript* from the C shell. These must be prepended by a "\" in order to work properly.

3.6 OBTAINING PARAMETERS FOR OUTPUT DRIVERS AND OTHER OPS

OPSCRIPT COMMAND

Up to this point we've used the *opparm* command to set up the parameters for our renders. Just how and where do we get these parameters? The easiest method of obtaining them is through the use of the *opscript* command.

The *opscript* command allows you to obtain the paramters (in text format) of any Operator available in Houdini. This means that you can create your scene and output drivers with Houdini's graphical interface and then use the *opscript* command to write these parameters to a file for use in shell scripts.

DUMPING OPSCRIPT PARAMETERS

To write an operator's parameters from the graphical interface to a file for inclusion in a script, do the following:

1. Create a new output driver with Houdini's graphical interface (or modify an existing one like *mantra1*);

2. Open the Textport (Alt Shift T);

3. Type `opcd /out` in the Textport. This moves you to the output directory;

4. Type `opscript <drivername> > <filename>`. If the driver name is called *mantra1* and the filename you want to save to is called */usr/tmp/driver.cmd* then type: `opscript mantra1 > /usr/tmp/friver.cmd`

SAMPLE OUTPUTS FROM OPSCRIPT

The following output was generated with the *opscript* command:

```
opscript mantra1 > /usr/tmp/driver.cmd :
```

```
1: # Node mantra1
2: opadd -n mantra mantra1
3: oplocate -x 0 -y 0 mantra1
4: opparm mantra1 execute ( 0 ) trange ( on ) f ( 1 150 1 ) \
5: renderer ( Mantra2 ) camera ( cam1 ) visible ( * ) tscript ( off ) \
6: script ( /usr/tmp/test.cmd ) binary ( off ) picture ( ‘$HIP/$F.pic’ ) \
7: dof ( on ) jitter ( 1 ) dither ( 0.004 ) gamma ( 1 ) sample ( 3 3 ) \
8: field ( frame ) blur ( deform ) tres ( on ) res ( 320 243) \
9: resmenu ( “640 486 1.0” ) aspect ( 1 ) command ( ‘mantra3’ )
10: opset -d off -r off -t off -l off -s off -u off -c off -C off -p off mantra1
11: opcf /out
```

Lines 4-9 specify parameters for the ouput driver *mantra1*. Normally they would appear as one line but have been formatted on multiple lines here for the sake of legiblity.

To use this information in a script, you would remove lines 3, 10, and 11, (these are not necessary unless you are importing it into the graphical version of Houdini) leaving you with:

```
1: # Node mantra1
2: opadd -n mantra mantra1
4: opparm mantra1 execute ( 0 ) trange ( on ) f ( 1 150 1 ) \
5: renderer ( Mantra2 ) camera ( cam1 ) visible ( * ) tscript ( off ) \
6: script ( /usr/tmp/test.cmd ) binary ( off ) picture ( ‘$HIP/$F.pic’ ) \
7: dof ( on ) jitter ( 1 ) dither ( 0.004 ) gamma ( 1 ) sample ( 3 3 ) \
8: field ( frame ) blur ( deform ) tres ( on ) res ( 320 243) \
9: resmenu ( “640 486 1.0” ) aspect ( 1 ) command ( ‘mantra3’ )
```

This fragment adds an output driver called *mantra1* which tells *hscript* to render using *mantra*. It then uses the *opparm* parameter to set the requisite parameters.

At first glance, the number of parameters used with the *opparm* command seems cumbersome. In many cases, you might want to use default settings and just tweak one or two parameters. For example, assume that the only thing you want to change from the defaults is the size of the output image. To accomplish this, your script fragment might look like the following:

```
opadd -n mantra mantra1
opparm tres ( on ) res (640 480 )
```

In this case you are overriding the default resolution with *tres (on)* and specifying the custom resolution with *res (640 480)*. To use this in a file you could do two things:

1. Copy these lines into your C shell script (the approach used thus far), or;

2. Reference this file in the C shell using the *source* command.

3.7 USE OF THE SOURCE COMMAND

Though it hardly seems a chore to simply copy these lines into your shell script in these simple cases, imagine that you have many script fragments that are changed repeatedly during a production by your Technical Director and these are stored in a central repository. Instead of having the Technical Director or the animator make these changes in the C shell script, all that is necessary is to change the *.cmd* file itself and the changes are automatically reflected. The use of the source command can make a file cleaner and easier to maintain, but it depends on the individual and the overall complexity of the script.

The two examples below outline render scripts that are functionally equivalent, except the first example uses the *source* command to reference a file created by *opscript*, while the second reproduces the code “inline”.

USING SOURCE

```
#! /bin/csh -f

set HIP_FILE = “/usr/temp/test.hip”

hscript << ENDCAT  # start up hscript and allow commands to be
mread $HIP_FILE    # sent from the C shell

#CREATE RENDER OUTPUT DRIVER
#source in the render.cmd file that we have previously created
 source /usr/tmp/render.cmd

#start the render
render mantra1

ENDCAT # this signals and end to our commands to hscript
```

WITHOUT USING SOURCE

```
#! /bin/csh -f

set HIP_FILE = “/usr/temp/test.hip”

hscript << ENDCAT # start up hscript and allow commands to be
mread $HIP_FILE # sent from the C shell

#CREATE RENDER OUTPUT DRIVER
# Node mantra1
opadd -n mantra mantra1
opparm mantra1 execute ( 0 ) trange ( on ) f ( 1 150 1 ) \
renderer ( Mantra2 ) camera ( cam1 ) visible ( * ) tscript ( off ) \
script ( /usr/tmp/test.cmd ) binary ( off ) picture ( ‘$HIP/$F.pic’ ) \
dof ( on ) jitter ( 1 ) dither ( 0.004 ) gamma ( 1 ) sample ( 3 3 ) \
field ( frame ) blur ( deform ) tres ( on ) res ( 320 243) \
resmenu ( “640 486 1.0” ) aspect ( 1 ) command ( ‘mantra3’ )

#start the render
render mantra1

ENDCAT # this signals an end to our commands to hscript
```

3.8 DEFAULT PARAMETERS FOR OUTPUT DRIVERS

The following are the default parameters set for each output driver when using the *opadd* command or when adding a new driver in the graphical interface.

It is not necessary to include all parameters available when executing an *opscript* command–just the ones you want changed from the defaults. Also note that some parameters remain consistent for each driver and should not be changed. These include:

- execute (0)
- renderer (Mantra2)
- resmenu (“640 486 1.0”)

PARAMETER MEANINGS

trange (on/off)	when rendering, this parameter determines whether or not to use the frame range specified by *f ( )*. When off, only the current frame will be rendered.
f (start end inc)	specifies the frame range that is rendered.
camera (cam)	specifies which camera is used for the render.
visible (scope)	specifies which objects are rendered. The wildcard * means all objects are made visible in the render.
tscript (on/off)	specifies whether or not to generate the *.ifd* (instantaneous frame description for *mantra*) or a *.rib* (RenderMan interface bytestream for Renderman) instead of a rendered picture. This option will override that set with the *picture ( )* parameter.
script (script_name)	specifies the name of the script generated when the *tscript ( )* option is on.
binary (on/off)	specifies whether the output file will be saved in ASCII or binary format.
picture (file_name)	the name of the output file when generating images.
dof (on/off)	specifies whether to use depth of field.
jitter (value)	specifies the jitter value used in conjuction with anti-aliasing techniques.
dither (value)	sets the dither level.
gamma (value)	sets the gamma value for the frame.
sample (xval yval)	sets the number of supersamples in the horizontal and vertcal directions on a per pixel basis.

field (frame/even/odd)	allows you to specify full frame, even field or odd field dominance.
blur (off/deformation/transformation)	specifies what type of motion blur to use for Mantra and Renderman.
tres (on/off)	specifies whether to override the default camera resolution.
res (x y)	specifies resolution of frame in pixels when *tres()* is on.
aspect (value)	specifies the aspect ratio.

3.9 OUTPUT DRIVER SAMPLES

MANTRA

```
opparm mantra1 execute ( 0 ) trange ( off ) f ( 1 150 1 )
renderer ( Mantra2 ) camera ( cam1 ) visible ( * ) tscript ( off )
script ( “” ) binary ( on ) picture ( ip ) dof ( off ) jitter ( 1 )
dither ( 0.004 ) gamma ( 1 ) sample ( 3 3 ) field ( frame )
blur ( none ) tres ( off ) res ( 320 243 ) resmenu ( “640 486 1.0” )
aspect ( 1 ) command ( ‘mantra3 -v 0.015’ )
```

RENDERMAN

```
opparm rman1 execute ( 0 ) trange ( off ) f ( 1 150 1 )
renderer ( Mantra2 ) camera ( cam1 ) visible ( * ) tscript ( off )
script ( “” ) binary ( on ) picture ( ip ) dof ( off ) jitter ( 1 )
dither ( 0.004 ) gamma ( 1 ) sample ( 3 3 ) field ( frame )
blur ( none ) tres ( off ) res ( 320 243 ) resmenu ( “640 486 1.0” )
aspect ( 1 ) command ( render ) device ( framebuffer )
```

COP

```
opparm cop1 execute ( 0 ) trange ( off ) f ( 1 150 1 )
icenetname ( “” ) copname ( “” ) copoutput ( ip ) tres ( off )
res ( 320 243 ) fraction ( 1 )
```

GEOMETRY

```
opparm geometry1 execute ( 0 ) trange ( off ) f ( 1 150 1 )
objectname ( geo1 ) sopname ( font1 ) sopoutput ( ‘$HIP/$F.bgeo’ )
```

SCENE

```
opparm scene1 execute ( 0 ) trange ( off ) f ( 1 150 1 )
renderer ( Mantra2 ) camera ( cam1 ) visible ( * ) tscript ( off )
script ( “” ) binary ( on ) picture ( ip ) dof ( off ) jitter ( 1 )
dither ( 0.004 ) gamma ( 1 ) sample ( 3 3 ) field ( frame )
blur ( none ) tres ( off ) res ( 320 243 ) resmenu ( “640 486 1.0” )
aspect ( 1 ) command ( mantra )
```

3.10 A FINAL EXAMPLE (RENDERING AND COMPOSITING)

This example is slightly contrived, but it is a good example of things you can do with scripting. Let's say you have two animators who are working on the same scene and at the end of the day, you want to render their contributions and make a side by side comparison to see which one is better.

We'll assume that we have two different .hip files – one from each animator. We then want to render a sequence of frames from each *.hip* file and then use the compositor in Houdini to place them side by side and then write the resulting "comparision frames" out to disk. You want to be able to do this automatically at night and then come in the next morning to see the results.

This involves two steps:

- Render the two sequences of frames, using a variation of the first render script example.
- Compositing the images together, and then writing them out to disk.

We'll build our COP network interactively in Houdini and save it out as a *.hip* file. We'll then change the parameters of the composite from the C shell based on the render parameters we give it. This makes life much easier while still giving you all the flexibility you need. You could build the entire compositing section with hscript. This is included as listing 5 for comparision purposes.

BUILDING THE COP NETWORK

The idea will be to take a neutral background colour (like black) which is double the size of the two input images, then use the Over COP to position and overlay the images correctly over the background.

1. Start Houdini and create a new COP network. Call it *compare*.

2. In the compositor place down a constant COP. Set alpha to zero, so it is completely transparent and make sure its name is "color1".

3. Place two File COPs making sure their names are *file1* and *file2*.

4. Append an Over COP to each of the file COPs and in the *Spatial Shift* tab select the *no scale* checkbox, making sure they are named *over1* and *over2*.

5. Attach the output from the Constant COP (*color1*) into the second inputs of each of the Over COPs.

6. Put down another Over COP and run the outputs from the first two Over COPs into the inputs of the third Over.

7. Go to the Output Editor, and place a Composite output driver, making sure it is called *cop1*.

8. Save this file and call it *compare.hip*. (Don't worry about other parameters, such as the resolution of the images, since we'll be setting those in the C shell.)

WRITING THE RENDER SCRIPT

Create a render script called *compare* and save it in the same directory as *compare.hip*. The full script is below and makes several assumptions. First, the *.hip* files that are the animator-supplied *.hip* files, must both have an output driver called mantra1. Second, the render script is called *compare* and is located in the same directory as the *compare.hip* file.

usage

```
compare <hipfile1> <hipfile2> <from> <to> <xres> <yres>
```

example

```
compare animator1.hip animator2.hip 1 50 640 480
```

This will render out frames 1 to 1 50 from each of the files called *animator1.hip* and *animator2.hip* at 640 × 480 resolution and the composite the images together using the *compare.hip* file created above.

CODE FOR COMPARE RENDER SCRIPT

```
#! /bin/csh -f

#check to see if the user supplied the correct number of arguments
#- exit if not
if ($#argv < 6) then
  echo "USEAGE: ren_script <hipfile1> <hipfile2> <from> <to> <xres> <yres>"
  exit
endif

set HIP_FILE1   = $1    # set up user supplied arguments
set HIP_FILE2   = $2
set START       = $3
set END         = $4
set XRES        = $5
set YRES        = $6
set OUT_FILE1   = sequence1_\$F.pic
set OUT_FILE2   = sequence2_\$F.pic
set COMP_OUTPUT = comp_\$F.pic

#Render out the first sequence of images
#
hscript <<ENDCAT>&renlog     # start up hscript
mread $HIP_FILE1             # read in the first .hip file

# set the ouput file name for the render output driver
opcd \/out
opparm trange ( on ) f ( $START $END 1 )
opparm tres (on ) res ( $XRES $YRES )
opparm mantra1 picture ( '$OUT_FILE1' )

render mantra1

#this signals an end to C shell to stop sendng commands to hscript
ENDCAT
#Render out the second sequence of images
#
hscript <<ENDCAT>&renlog
mread $HIP_FILE2                #read in the second .hip file

opcd \/out
opparm mantra1 trange ( on ) f ( $START $END 1 )
opparm mantra1 tres (on ) res ( $XRES $YRES )
opparm mantra1 picture ( '$OUT_FILE2' )

render mantra1

#stop sending commands to hscript (effectively quitting hscript)
ENDCAT

#Do the composite, putting both images side by side
#
hscript<<ENDCAT>&renlog     # start up hscript
mread "compare.hip"         # read in the compositing .hip file
```

```
opcd \/comp\/compare
opparm file1 source ( '$OUT_FILE1' )
opparm file2 source ( '$OUT_FILE2' )
opparm color1 size ( \`$XRES*2\` $YRES )
opparm over2 offoffpixel ( $XRES 0 )

opcd \/out
opparm cop1 icenetname ( compare ) copname ( over3 )
opparm cop1 trange ( on ) f ( $START $END 1 )
opparm cop1 copoutput ( '$COMP_OUTPUT' )
render cop1

ENDCAT
```

NOTES ON THE RENDER SCRIPT

The first half of the script is nothing new. It merely sets up our user-supplied parameters, and then renders a sequence of images from the two .hip files supplied as command line paramters. The third section which does the actual composite is a little different, but if you understand the rendering sections, you should have little trouble understanfing how the compositing section works.

The trickiest part in this whole script is the line:

```
opparm color1 size ( \`$XRES\*2\` $YRES )
```

Note the backslashes here. These are necessary because we are dealing with the C shell. In *hscript* we could just type in something like `` echo `1+1` `` and get 2. If we try this through the C shell we get an error. So the general rule is if you're trying to do arithmetic in *hscript* via the C shell, put a \ in front of the backquote `.

The compositing section works by setting up all of the compositing parameters on the fly. It first tells the File COPs where to find the files. It then tells the Color COP to resize itself so it is double the X reolution of the two rendered images (so we can place them side by side). Finally it tells the two Over COPs where to put the two rendered images in relation to the background (in this case image one is shifted up to the halfway mark and image two is shifted over to the right and up half way).

The final step is to tell the composite output driver the frame range and file names for output.

THINGS TO WATCH OUT FOR

This is a small list of things to watch out for when using hscript within the C shell. Readng this will probably save you a lot of aggravation and time if you find your scripts don't work as they should. Many errors can be attributed to the C shell expanding strings and variables when in fact, you don't want them to.

One other thing to keep in mind is that hscript and C shell are not syntactically the same–though it may appear to be at first glance. Entries that you make in Houdini's textport may need some editing when you use them as input from the C shell.

Read the compare script above to see some of following items put into practice.

- *Passing in strings to hscript*
  Passing variable strings to *hscript* should be encapsulated by single forward quotes. So if you set a variable string such as *set OUT = image_$F.pic* then make sure when you use it with *hscript* you type in '*$OUT*'.
  If you want to pass in a string with spaces in it (like "*hello there*"), make sure to put it in double quotes.

- *Variables passed in as numerics*
  Any variable passed into *hscript* as a numeric can be left as is (unless you are performing arithmetic with it).
- *Specifying Directories and using '/'*
  If you change directories in *hscript* (e.g. *opcd /out*) then make sure that any forward slashes are protected by a backslash (*opcd \/out*).
- *Arithmetic*
  If you want to do some arithmetic via *hscript* from within C shell, make sure that any arithmetic is surrounded by a \` to prevent C shell from doing any expansion with the backquote. For instance in *hscript* you might type: *echo `1+2`*. If you are doing this within C shell you would use: *echo \`1+2\`*.
- *Watch the ENDCAT*
  When you have the *hscript<<ENDCAT ... ENDCAT* sequence, take care not to put any trailing spaces or comments after the *ENDCAT*. C shell is very literal when looking for the terminating token. Look at the example scripts to see how they are laid out.
- *Make sure your C shell script starts with the line : #! /bin/csh -f*
  This tells UNIX that it is dealing with a C shell script. Also make sure that this entry is on a line by itself.
- *Make sure scripts are executable*
  Your C shell scripts have to be executable or you won't be able to run them.

To make this a little clearer, let’s look at a script segment that works when type manually into Houdini’s textport, and then compare that with a version that must be modified to work with C shell. The differences are set in bold type.

segment 1 (as it would appear manually typed into houdini's textport)

```
set OUT_FILE1 = source1_\$F.pic
set OUT_FILE2 = source2_\$F.pic
set COMP_OUTPUT = comp_\$F.pic
set XRES = 256
set YRES = 256

opcd /comp/compare
opparm file1 source ( $OUT_FILE1 )
opparm file2 source ( $OUT_FILE2 )
opparm color1 size ( `$XRES*2` $YRES )
opparm over2 offoffpixel ( $XRES 0 )
```

segment 2 (modified for use with c shell)

```
set OUT_FILE1 = source1_\$F.pic
set OUT_FILE2 = source2_\$F.pic
set COMP_OUTPUT = comp_\$F.pic
set XRES = 256
set YRES = 256

opcd \/comp\/compare                   #directory operation
opparm file1 source ( '$OUT_FILE1' )   #using a string variable
opparm file2 source ( '$OUT_FILE2' )       #directory operation
opparm color1 size ( \`$XRES*2\` $YRES )   #doing arithmetic
opparm over2 offoffpixel ( $XRES 0 )
```

3.11 BUILDING COMPLEX FILENAMES

There are many times when you want to build complex filenames, based on frame numbers or based on a filename referenced by the user. For example, passing in a .hip file to your script, then converting this filename to a .pic file (i.e. convert "input.hip" to "output.pic"). Below are C shell example fragments.

EXTRACTING THE BASE FILENAME, PATH AND EXTENSION

```
set HIP_FILE  = \/usr\/tmp\/myfile.hip
set HIP_PATH  = $HIP_FILE:h        #keep the path/drop filename
set FILE_NAME = $HIP_FILE:t        #keep the filename/drop path
set BASE      = $FILE_NAME:r       #drop the extension
set EXT       = $FILE_NAME:e       #just keep the extension
```

The variables would have the following values:

```
$HIP_FILE  = /usr/tmp/myfile.hip
$HIP_PATH  = /usr/tmp
$FILE_NAME = myfile.hip
$BASE      = myfile
$EXT       = hip
```

BUILDING A FILE NAME WITH A NEW EXTENSION

Continuing from above we will build a new filename with a new extension. We have all of the individual components and now we want to put them back together again.

```
set NEW_EXT = .pic
set NEW_FILE = $HIP_PATH\/$BASE${NEW_EXT}
```

Notice the braces. This allows you to put togther two variables as one string. Note that you could say *$BASE$NEW_EXT*, but it's good to get in the habit of using the braces, as it might save you trouble later on. Also notice that we put in a \/. If you look closely at the results of *$HIP_PATH*, it doesn't include the trailing forward slash, so we have to include it ourselves.

Now we end up with the following results:

```
$HIP_FILE  = /usr/tmp/myfile.hip
$HIP_PATH  = /usr/tmp
$FILE_NAME = myfile.hip
$BASE      = myfile
$EXT       = .hip
$NEW_EXT   = .pic
$NEW_FILE  = /usr/tmp/myfile.pic
```

We could have accomplished the same thing much more quickly. All we really wanted to do was change the extension. The following would be equivalent:

```
set $HIP_FILE = \/usr\/tmp\/myfile.hip
set FILE_NAME = $HIP_FILE:r           #keep all but extension
set NEW_EXT   = .pic
set $NEW_FILE = $FILE_NAME${NEW_EXT}
```

The reason for breaking up the full path into its individual components is so that you can change the individual components and put them back together again.

MEANING OF :H, :R, :E, :T

More information on the following, can be found by doing a *man csh* and searching for *History Substitution.*

```
:h Remove a trailing pathname component, leaving the head.
:r Remove a trailing suffix of the form `.xxx', leaving the
   basename.
:e Remove all but the suffix.
:t Remove all leading pathname components, leaving the tail.
```

4 SCRIPTING TRICKS

4.1 GROUP NAMES IN SCRIPTING COMMANDS

You specify a group within scripting commands by using the @ character before the group name. For exampe, if you wanted to turn on/off the display of all objects in a group, you would use the *opset* command (e.g. *opset -d on/off geo**). In the command, we can specify an OP name (e.g. geo1), a pattern for an OP name (e.g. geo*), or OP Groups (e.g. @myGroup).

To test this, use the Object Editor's *Edit* > *Edit OP Groups*... menu command to create an object group called "myGroup". It should contain geo1, geo2, and ambient1. You can list the objects included in *myGroup* with the *opgls* command:

```
/obj -> opset -d on @myGroup
```

Turns on the display flags for all the objects listed in *myGroup* .

Using the @ character to expand a group name can also be applied to any other Houdini scripting command that normally accepts only object names.

Tip: Try assigning this to a Function Key using the *Edit* > *Edit Aliases*... dialog.

4.2 EMBEDDING COMMANDS

Without the Group expransion using the @ character, we would have to turn on/off display of all the objects in a group by using the *opgls* command delimited with single quotes. For example, if you didn't have the @ character, you would need to use something like:

```
-> opcf /obj

/obj -> opgls -l myGroup
myGroup
  geo1
  geo2
  ambient1
```

```
/obj -> opset -d on `run("opgls -l myGroup")`
```

This works, because where it expects an object name (or pattern), you're telling it to evaluate what is within the single quotes as the name(s). The *opgls* command simply lists all the objects contained in the specified group. So, in effect, what you're doing is listing the names of all the objects within your group by having the *opgls* command list them out for you. You'll get a single error (this doesn't affect correct operation however), because the actual group name (i.e. "myGroup") is listed along with all the object names in that group, and *hscript* won't find it as a valid object.

4.3 SETTING ACCORDING TO THE DISPLAY FLAG

If you want to set all the objects within a group (say “bugs”) but only if their Display flag is set (Object Editor), you can use a *foreach* loop to check and set the status of objects in the group which have their Display flag on. For example:

```
opcf /obj

#Loop through all the objects in the group
foreach obj ( ‘run("opglob @bugs")‘ )
  opset -l on $obj/‘opflag($obj, "d")‘
```

4.4 TRAVERSING AN OBJECT HIERARCHY

The following script can be used to traverse an object hierarchy. This script simply prints out the hierarchy (with appropriate indenting), however, it can easily be modified to do other things.

```
# hscript command file to traverse a hierarchy of nodes
if ( $argc != 2 && $argc != 3 ) then
  echo "Invalid usage: $arg0 opname [prefix]
  exit
endif

if ( $argc == 2 ) then
  set indent = ""
  set level = 1
else
  set indent = ""
  for i = 0 to $arg2
    set indent = "$indent "
  end
  set level = ‘$arg2 + 2‘
endif
echo "$indent"$arg1

set nout = ‘opnoutputs($arg1)-1‘
if ( $nout != -1 ) then
  for i = 0 to $nout step 1
  source $arg0 ‘opoutput($arg1, $i)‘ $level
  end
endif
```

2 Scripting Commands

Following, is a list of scripting commands available in the Houdini scripting language. Commands can be broken up into different logical groups.

For C-shell scripting commands, consult a text on UNIX.

1 INTRINSIC COMMANDS

This set of commands provide an intrinsic level of control for scripting. These commands are most like their *csh* equivalents.

1.1 ALIAS

SYNTAX

```
alias [name [value]]
alias -s
alias -u name [name2...]
```

EXPLANATION

Creates an alias for a command or sequence of commands. Aliases may contain semi-colon separated statements.

`-s`	Display a list of current aliases in “source-able” format so they can be sourced into other HIP files.
`-u`	Will undefine the aliases listed.

EXAMPLE

```
alias ls opls
```

This alias command changes the *opls* (operator list) command to the abbreviated character string *ls*.

TEMPORARILY DISABLING ALIASING

In the Houdini shell, if the local variable *noalias* is set, then alias expansion is not done on commands in the script. This variable can be used to force scripts to use the original commands instead of user aliases. Since local variables are local per script, once the script exits, alias expansion will continue as before. However, if the variable is set, all subsequent (nested) scripts will have alias expansion turned off. Example:

```
alias echo "This is a bad alias"
set noalias = 1
echo "foo bar"
set -u noalias
```

In this script, even though the alias is set, the echo statement will work (since the noalias variable is set).

1.2 BREAK

SYNTAX

```
break [n]
```

EXPLANATION

Breaks out of a loop without executing any of the remaining statements. The loop will be terminated without completing its iterations. The integer specified by n determines how many loops to break out of. By default, *n* == 1.

EXAMPLE

```
break 3
```

This command string would break you out of three levels.

1.3 CMDREAD

SYNTAX

```
cmdread [-q] filename
```

EXPLANATION

Cmdread runs the commands in the filename specified. If the *-q* option is specified, no warnings about missing filenames will be displayed. See also: *source*.

1.4 CONTINUE

SYNTAX

```
continue [n]
```

EXPLANATION

Continue a loop without executing the statements following the continue statement. The loop will continue iterating. The integer specified by *n* determines how many loops to affect.

EXAMPLE

```
continue 3
```

This command executes the script, omitting the next three loops.

1.5 ECHO

SYNTAX

```
echo [-n] list
```

EXPLANATION

The words in list are displayed. The *-n* option will prevent a trailing line feed from being displayed.

EXAMPLE

```
echo bafflegab
```

This command would produce the text *bafflegab* below the command line.

```
echo `npoints(“/objects/object_name/sop_name”)`
echo `npoints(“/objects/gg/s”)`
```

If object is *gg* and the SOP is s then following expression will display the number of points in the SOP *s*.

1.6 EXCAT

SYNTAX

```
excat [pattern]
```

EXPLANATION

This command displays the source for all expression functions in the current .hip file. If a pattern is specified, only those expression functions matching the pattern are displayed.

EXAMPLE

```
excat fps
```

This will display all expressions in the .hip file that contain *fps*.

1.7 EXEDIT

SYNTAX

```
exedit [pattern]
```

EXPLANATION

This command allows you to edit expression functions. If no pattern is specified, you can add new functions to the current list. If a pattern is specified, the functions which match the pattern will be edited.

Warning: If a function is renamed or removed from the edit session, this does *not* mean that the old function will be removed from the current function list. This must be done through the *exrm* command.

EXAMPLE

```
exedit $F
```

Allows you to edit the expression functions containing *$F*.

Tip: You can also edit expression functions using the dialog displayed from the *Edit > Edit Aliases/Variables...* menu command.

1.8 EXHELP

SYNTAX

```
exhelp [pattern]
```

EXPLANATION

Displays help text for all expression functions matching the pattern specified. If no pattern is specified help for all the expressions is shown.

EXAMPLE

```
exhelp sin
```

Displays the online help for the command chadd.

1.9 EXLS

SYNTAX

```
exls
```

EXPLANATION

List all the current expression functions.

1.10 EXREAD

SYNTAX

```
exread diskfile [diskfile2...]
```

EXPLANATION

This command reads external files of expression functions.

EXAMPLE

```
exread /n/usr/staff/betty/[filename]
```

This command reads the expression functions in the path and file(s) specified.

1.11 EXRM

SYNTAX

```
exrm [pattern]
```

EXPLANATION

All expression functions matching the pattern will be removed.

1.12 EXIT

SYNTAX

```
exit
```

EXPLANATION

Terminates a source file. This will terminate all “if” statements and “for” loops correctly. It is not possible to specify an exit status, except that the *setenv* command can be used to return a status in a global variable.

1.13 FOR

SYNTAX

`for` *VARIABLE* `= START to END [step INC]`

EXPLANATION

The “for” loop construct. The loop will set the value of *var* to start. On each iteration of the loop, the value of *var* will have *inc* added to its value. The loop will terminate after the end is passed. If the end value is achieved exactly, the loop will iterate for this value. By default, the step size is 1. The end of the “for” loop is flagged by the end command. For example:

```
houdini -> for i = 1 to 3
> echo -n $i,
> end
1, 2, 3,
```

The variable you specify loops from the beginning to the end according to the increment you set.

EXAMPLE

```
for i = 1 to 3
for i = 1 to 100 step 3
```

In the examples above, the variable *i* will loop, in the first instance, from one to three. In the second instance, the variable will loop from one to one hundred in increments of three.

1.14 FOREACH

SYNTAX

`foreach` *VAR* `(list)`

EXPLANATION

The *foreach* loop construct. The loop will set the value of *VAR* to a different word in the list for each iteration of the loop. The list is processed in the order specified. The end of a *foreach* loop is always signified by the *end* command. For example:

```
> foreach obj ( `execute(“opls”)` )
> echo -n $obj,
> end
cam1, geo1, geo2, light1, light2,
```

1.15 HELP

SYNTAX

```
help [command_pattern] [-k expression]
```

EXPLANATION

If no command is specified, a list of available commands is displayed.

If a command is specified, help for that command will be displayed.

The -k option allows you to search for keywords. All commands which contain the keyword will be displayed.

EXAMPLE

```
help echo
```

Displays the help available for the *echo* command.

```
help -k expression
chkey excat exedit exhelp exls exread exrm opcopy opfind
```

Each of these commands has the word ’expression’ somewhere in the help for the command.

1.16 HISTORY

SYNTAX

```
history [-c]
```

EXPLANATION

Displays the command history. If you employ the *-c* option, the command history is cleared.

1.17 IF

SYNTAX

```
if ( expr ) [then]
     ...
  else if (expr2) [then]
     ...
  else
     ...
endif
```

EXPLANATION

If *expr* is true, the commands up to the first *else* are executed. If *expr* is false and *expr2* is true, then the commands between the two *else* statements are executed. If *expr2* is false, the commands between the *else* and the *endif* are executed. It is not necessary to specify the two *else* statements.

It is not possible to specify commands following the *if* statement. Any arguments (except the trailing *then*) are considered to be parts of the condition.

A matching *endif* statement should always be used after an *if* statement if the *if* statement is more than one line.

1.18 JOB

SYNTAX

```
job [unix_path]
```

EXPLANATION

Sets the job variable to the path you specify.

EXAMPLE

```
job /n/usr/caesar
```

This command line changes the job directory.

1.19 MEMORY

SYNTAX

```
memory
```

EXPLANATION

Displays the current memory usage of the application that is running.

1.20 PROMPT

SYNTAX

```
prompt [new_prompt]
```

EXPLANATION

Change the current prompt. Before the prompt is displayed, the value of the prompt is expanded. Therefore, it is possible to set the prompt to something very meaningful.

EXAMPLE

```
  houdini -> prompt ‘$HIPNAME Frame $y -> ‘
  untitled1.hip Frame 1 ->
```

1.21 PRINT

SYNTAX

```
print label expression
```

EXPLANATION

Displays the value of the expression to stdout and returns the same expression value. This can be used to diagnose parameters in OPs or channels.

Note: “print” in shell-speak doesn’t actually print to the printer, but displays the result in the shell in which the command is executed.

EXAMPLE

```
print("wheel:", sin($T))
```

1.22 QUIT

SYNTAX

```
quit
```

EXPLANATION

Terminates the application. Some applications will not warn of quitting without saving (i.e. hscript), while others will (i.e. Houdini).

1.23 READ

SYNTAX

```
read [-g] variable_name [variablename2...]
```

EXPLANATION

Will read the following line into the variable names specified. The first argument will be put into the first variable. The last variable specified will contain the remaining arguments of the input line. If the *-g* option is specified, the variables will be set as global variables instead of local variables. The -g option makes the variables global (see *set* p. 117).

1.24 RKILL

SYNTAX

```
rkill [process]
```

EXPLANATION

Any background render which has a process ID matching the process pattern specified on the command line will be terminated. Since the process argument specified can be a pattern, it is possible to kill multiple renders at once.

EXAMPLE

```
rkill 9382
rkill *
```

The first example stops the specific background rendering process, while the second stops all background rendering in progress.

1.25 RPS

SYNTAX

```
rps
```

EXPLANATION

This command lists active background render processes. The command lists the process identification number, the host on which the command is running, and the name of the command being run.

1.26 SET

SYNTAX

```
set [-g] varname = value
set -p name = value
set -u name
set [-s]
```

EXPLANATION

The *set* command is used to assign local variables to the value given (use the *setenv* command to set global variables). With no arguments, it will list all current variables and their current values.

The *-g* (global) option on the *set* command makes it work like *setenv*, otherwise the variable will be Local to the script file where the command is executed.

If no name is specified and the -s option is given, it will output the list of variables in a form which is useful for sourcing into another .hip file. This makes it easier to move variables from one .hip file to another.

The -p option will set the variable in the caller (or parent) script. If the currently running script is at the topmost level, then this option has no effect. This option lets you return values from within sourced scripts. For example, to set a return value into the variable name passed into our script as the first parameter, you would do something like:

```
set -p $arg1 = $returnValue
```

The -u option will un-set the specified variable.

See also: *setenv* .

EXAMPLE

```
setenv -l HOUDINI_LOD = 2
```

Temporarily changes the Houdini Level of Detail to *2* within the current script.

THE DIFFERENCE BETWEEN SET AND SETENV

Using the *setenv* command in the scripting language is different than using the *set* command in two ways:

1. The *set* command is local to the script which is currently running. This means that when another script is called, or the current script exits, the variable is no longer visible. Also, this means that you can re-use variables within different scripts without over-writing their values.

2. The *setenv* command will create a global variable. The *setenv* command will also export all variables to any processes started from Houdini. For example:

```
hscript-> set foo = 0 ; unix echo '$foo'
foo undefined variable
```

```
hscript-> setenv foo = 0 ; unix echo '$foo'
0
```

1.27 SETENV

SYNTAX

```
setenv [-l] varname = value
setenv -u name
setenv [-s]
```

EXPLANATION

The *setenv* command sets the global variable you specify by name to the value specified. If you do not provide a name, a list of all variables is displayed. When not providing a name, and using the *-s* option, the command will produce output suitable for loading as a script.

The *-l* (local) option on *setenv* makes it work like *set* – forcing the variable to act locally, meaning their values are discarded once the current script file ends.

If no name is specified and the -s option is given, it will output the list of variables in a form which is useful for sourcing into another .hip file. This makes it easier to move variables from one .hip file to another.

The *-u* option will un-set the specified variable.

See also: *set* .

Note: It is important to note that this command actually sets a real UNIX environment variable, its influence is therefore both within the Houdini shell, and in your standard UNIX shell. You can find a complete list of Houdini-related environment variables in: *Environment Variables* p. 211.

1.28 SHIFT

SYNTAX

```
shift
```

EXPLANATION

When a script is sourced, the arguments are set to variables *$arg0*, *$arg1* ... The shift command will shift the arguments so that *$arg1* goes into *$arg0*, *$arg2* goes into *$arg1* etc. The *$argc* variable is decremented to reflect the changes.

1.29 SOURCE

SYNTAX

```
source filename [arg1...]
```

EXPLANATION

Sources a command script and executes the commands contained in the script until the exit command is reached or the end of file is reached. The arguments to the script are passed in local variables *$arg0... $argn*. The number of arguments is passed as *$argc*. This command is often used to load in files generated by *opscript* and *opwrite*.

EXAMPLE

```
source 123.cmd
```

This command runs the commands in the file *123.*`cmd`.

1.30 TIME

SYNTAX

```
time [command]
```

EXPLANATION

The *Time* command allows you to time other commands (i.e. a render command or a *source* command). The time displayed shows how much user/system and real time the command took.

EXAMPLE

```
0.0u 0.0s 0.0r
% time render mantra1
0.1u 0.2s 18.7r
```

This indicates that Houdini took .1 seconds of CPU, .2 seconds of system time and then had to wait 18.7 seconds of real time for *mantra* to finish rendering.

1.31 UNDOCTRL

SYNTAX

```
undoctrl [on|off]
undoctrl -s
```

EXPLANATION

This can turn on or off the undo mechanism in Houdini. With no options, the current state will be printed out. Please use this command with extreme caution. Turning off the undo mechanism can cause scripts to execute with greater speed, but the changes made by the script will not be undo-able. As well, be careful to restore the undo state at the conclusion of the script.

The second usage with the -s option queries the memory usage of the undo mechanism.

Note: Be careful to restore the undo state at the conclusion of the script! It would be a shame to lose hours of work because a script forgot to turn undo's back on.

1.32 VERSION

SYNTAX

```
version
```

EXPLANATION

Displays the current version of the program you are running.

1.33 WHILE

SYNTAX

```
while (expression)
  ...
end
```

EXPLANATION

The *while* loop construct. A *while* loop will iterate continuously while the *expression* evaluates true. When the *expression* is false, the loop will terminate. This means you will have to manually include a variable within the expression, and increment that variable somewhere within the body of the loop in order for a while loop to finish its execution.

Warning: It is very easy to create endless loops which will not terminate if you are not careful about incrementing the variable within the expression somewhere within the body of your loop. You may want to use the *foreach* and *for* loop constructs which implicitly increment your variable.

EXAMPLE

```
set i = 0
while ( $i < 10 )
  set i = `$i+1`
  echo $i
end

output: 1 2 3 4 5 6 7 8 9
```

2 UNIX RELATED COMMANDS

These commands provide a minimal interface to the UNIX shell.

2.1 UCD

SYNTAX

```
ucd [path]
```

EXPLANATION

Changes the current working directory to the one you specify in the path statement.

EXAMPLE

```
ucd /n/usr/staff/mulder
```

In this example the working directory would be altered to *mulder*.

2.2 UPWD

SYNTAX

```
upwd
```

EXPLANATION

This command displays the current UNIX working directory.

2.3 UNIX

SYNTAX

```
unix command [argument1...]
```

EXPLANATION

Runs the UNIX command you specify. The command will be run in its own csh.

EXAMPLE

```
unix csh -f -c
```

In the example above, the csh will be started.

3 PLUG-IN COMMANDS

These commands are provided through plug-in modules. These plug-ins do not have to be loaded, but provide extra functionality if they are.

For an example of how to create a Tcl/Tk script, see *Scripting* chapter of the *User Guide*.

3.1 TCL

SYNTAX

```
tcl [args]
```

EXPLANATION

This command allows you to run scripts written in the Tcl language from within Houdini, and is useful for customising Houdini's interface and dialog boxes.

Tcl starts a tcl shell with the arguments given. There is an additional command in tcl "hscript" which can be used to run any Houdini command. tcl is a public domain scripting language which has many powerful features (see tk).

You can run a sample Tcl script by typing *tk hbrowser.tk* in the Textport. This script simply brings up a file requester style browser which shows the Houdini objects instead of files. By double clicking on an object, you will see the contents of the object. This is a quick way of seeing what objects are available.

Tcl is a scripting language which is in common use around the world. Tk is an extension to Tcl which allows you to create Windows and interface elements through the scripting language. You should be able to find books that discuss Tcl/Tk (commonly pronounced "tickle") in any computer book store.

3.2 TK

SYNTAX

```
tk [args]
```

EXPLANATION

Tk is a version of Tcl which supports X11 and Motif extensions. This allows you to build custom user interfaces in scripts. Again, there are several books which describe the tk language as well as a wealth of information on the world wide web. There are some simple example scripts installed in *$HH/scripts/tk* .

4 CHANNEL AND OPERATOR COMMANDS

These commands deal with channels and operators in general. The commands have been written so there is a minimal number to become familiar with, yet powerful enough to do almost anything that can be done through the graphical interface.

4.1 BONECONVERT

SYNTAX

```
boneconvert [-r | -m] [-x] [-t]
```

EXPLANATION

This command is used to update old hip files to use the new bones introduced in Houdini 5. The conversions performed are:

- All bones which have lock channels in their translate parameters matching:

  lock(0), lock(0), lock(-ch(strcat("../", strcat(opinput(".", 0), "/length"))))
  are changed to: *lock(0), lock(0), lock(0)* .

  The new bone objects have an output transform that places all child objects at their end points. To force this conversion, use the -t option.

- The *Top Cap* and *Bottom* parameters in the CRegion SOP of bone objects have their multiplication factor removed and multiplied into the values of the object-level cregion parameters. This will only be performed if the object-level cregion parameters have no channels. To deal with special cases, please see the options described below.

- All bone objects have their xray flag turned on. Use the -x option to avoid doing this conversion.

- Adds the command "bonefixchops $OPSUBNAME" to the delete script.

OPTIONS

The -r option forces the conversion of the CRegion SOP parameters even if the object-level cregion parameters already have channels. This option is useful if you have channel references in the object-level parameters that mirror other capture regions. The CRegion SOP parameters are forced to be correct without interpretation of the parameter.

The -m option not only forces the conversion of the CRegion SOP parameters like: -r, but it will also attempt to add the multiplication factor if the object-level parameters have channels on them. This option will not have different behaviour if the object-level cregion parameters do not have channels. It will also fail to add the multiplication factor if the cregion SOP parameters do not have an expression of the form <number>*<expression> .

4.2 BONEFIXCHOPS

SYNTAX

```
bonefixchops [-r] bone_object
```

EXPLANATION

This command is used to clean up InverseKin CHOPs that may reference the given bone object before the bone is deleted. For example, if an InverseKin CHOP is using an Inverse Kinematics solver on a bone chain from bone1 to bone4, and you execute "bonefixchops bone4", this CHOP will be changed to apply its solver to the chain from bone1 to bone3. If you have an InverseKin CHOP that is using an Inverse Kinematics solver on bone1 only, and you execute "bonefixchops bone1", the CHOP will be deleted. This command is used in the default delete script of bone objects.

If the -r option is used, then it will recursively destroy all outputs of the found InverseKin CHOPs as well.

4.3 BONEMOVEEND

SYNTAX

```
bonemoveend bone_object [-f "world"|"parent"] [-x xpos] [-y ypos]
  [-z zpos]
```

EXPLANATION

This command adjusts the length and rest angles of the given bone object so that in the rest chain the bone would end at the specified position.

4.4 BOOKMARK

SYNTAX

```
bookmark [-a path] [-l] [-r path_pattern]
```

EXPLANATION

This command is used to add, list and remove path bookmarks.

OPTIONS

`-a path`	Add path to bookmarks.
`-r path_pattern`	Remove path from bookmarks, wildcards such as *, ? and [ ] are valid.
`-l`	List current bookmarks.

4.5 CHCP

SYNTAX

```
chcp source_channel_name destination_channel_name
```

EXPLANATION

Copies the contents of one channel to another. If the destination channel already exists, its contents are deleted first.

EXAMPLES

```
chcp /obj/logo/tx /obj/sky/tx_copy
```

Copies the previously created tx channel of the logo object to be a spare channel of *sky* named tx_copy.

```
chcp /obj/logo/tx /obj/logo/ty
```

Copies the previously created tx channel of the *logo* object to the ty channel, overwriting any existing keyframe information of ty.

```
chcp /obj/logo/tx /obj/sky
```

Copies the previously created tx channel of the *logo* object to the sky object.
The new channel is named /obj/sky/tx.

4.6 CHADD

SYNTAX

`chadd [-f fstart fend] [-t tstart tend]` *object's name1* `[name2...]`

`[-f fstart fend]`	Represents the frame range you want to add the channel to.
`[-t tstart tend]`	Represents the time range you want to add the channel to.
`object's name`	Represents name of the object you wish to add a channel to.

EXPLANATION

Adds channels to the specified objects. You can specify objects using pattern matching i.e. *geo**. By default, the channels have a segment stretching from the beginning to the end of the animation. Specifying a frame, or time, range causes the segment to adopt that range.

EXAMPLE

`chadd` *geo** `tx ty tz spare1`

Adds channels tx, ty, tz and spare1 to all objects matching *geo**.

4.7 CHGADD

SYNTAX

```
chgadd -f group_name [second_name...]
```

EXPLANATION

This command creates a new channel group (or groups). If the *group_name* specified already exists, *chgadd* will not add a new group. The *-f* option can force *chgadd* to add a group. If there is already a group by that name, the new group is given a unique name.

EXAMPLE

```
chgadd bison
```

Creates the channel group *bison*.

4.8 CHCOMMIT

SYNTAX

```
chcommit [-l] [channel_pattern...]
```

EXPLANATION

Simulates adding a keyframe (i.e. clicking the red *Key* button in the Playbar). The *-l* option will not modify, but only list pending keyframe changes to the Textport. If no *channel_pattern* is specified, all pending keyframes changes are assumed.

4.9 CHGLS

SYNTAX

```
chgls [-l][-g] [pattern...]
```

EXPLANATION

This command lists channel groups. The *-l* option lists the contents of the channel group as well. The *-g* option generates commands which can be used to re-create the channel group (or groups) specified. If a pattern is specified, then only the groups matching the pattern are listed.

EXAMPLE

```
chgls -l
```

Lists the contents of all available channel groups

4.10 CHGOP

SYNTAX

```
chgop group_name operation channel_pattern [second_pattern...]
```

EXPLANATION

This command performs operations on groups of channels. It allows for the addition or removal of channels from a group. *group_name* designates the name of the channel group to modify. The *operation* variable permits three actions on the group:

`set`	Sets the contents of the group
`add`	Adds channels to a group
`remove`	Removes channels from a group

The *channel_patern* variable designates the list of channels to work on.

EXAMPLE

```
chgop group1 set /o*/g*/r?
```

Sets the group's contents to the channels.

```
chgop group1 add /o*/g*/t?
```

Adds channels to group one.

```
chgop group1 remove /o*/g*/tx
```

Removes *tx* channels from group one.

4.11 CHGRM

SYNTAX

```
chgrm group_pattern
```

EXPLANATION

This command remove a channel group or groups.

EXAMPLE

```
chgrm bison
```

Removes the channel group *bison*.

4.12 CHHOLD

USAGE

```
chhold [-b | -e] [channel_patterns]
Or:  chhold [-s] [-l]
```

Allows you to put channels into a 'hold' (or pending) state at the current time. This can be used in conjunction with the chcommit command to force the creation of keys.

OPTIONS

`-b  (begin)`	Turn on the hold status for the given channels so that they remain in a pending state even if time changes. If no patterns are given, then all currently scoped channels will be affected.
`-e  (end)`	Releases the previously held channels. If no patterns are given, then it will release all held channels.
`-s  (status)`	Queries the current hold status.
`-l  (list)`	Lists currently held channels.

4.13 CHKEY

SYNTAX

```
chkey [-f frame] [-t time] [-v value] [-m slope] [-a accel] [-F
  function] channel_pattern
```

Edit or insert a key frame by specifying the following:

`-v`	The value at the key frame
`-m`	The slope at the key frame
`-a`	The acceleration at the key frame
`-F`	The expression function for the segment following the keyframe.

EXPLANATION

Edits or creates a key frame with the values you specify. If a key frame already exists the time/frame you specify, the values for that key frame are modified. The channel pattern specifies which channels are affected by the command.

EXAMPLE

```
chkey -f 1 -v3 -F 'cubic ()' geo*/t?
```

Will, at frame one, set the value to three and the expression function to *cubic ()* for all channels matching the pattern *geo*/t?*. The expression function should usually be bracketed by ' ' in order to prevent expansion of its contents. In this example, the () would be expanded, producing a syntax error.

4.14 CHLS

SYNTAX

`chls [-1]` *object_name*

EXPLANATION

This command lists the existing channels. If you specify the *-1* option, the key frames for the channel are listed.

EXAMPLE

`chls -1` *geo*/**

This command string would produce a listing of the keyframes and channels for all geometry objects (*/*). When you use this command, ensure that you specify the object name.

4.15 CHREAD

SYNTAX

```
1) chread [-f fstart fend] channel_pattern ...filename.{chan,bchan}
2) chread [-f fstart fend | -o fstart] [-r src_pat dest_pat]
  filename.{chn,bchn}
3) chread -i filename.{chn,bchn}
```

EXPLANATION

Usage 1: The specified file is assumed to be a raw channel data file (.chan or .bchan) and column data from the file will be read and matched with the channels listed in the order specified. Loading will only occur in the frame range specified or will start at the global animation start time if no range is given. The order of channels resulting from a pattern match is not well defined.

Note that only channels with raw segments in the frame range will have values assigned to them. You must convert segments to raw before reading values in with chread.

Usage 2: The specified file is assumed to be a keyframe data channel file and loaded into the current hip file. The file extension must be either be .chn (ASCII) or .bchn (binary). The -f option gives the frame range to load the data into. Keys will be scaled into the frame range if the file's range does not match. Instead of the -f option, the -o option simply gives the starting frame to load the data into (no scaling will be done). If the -f and -o options are omitted, then the file's frame range is used. If there are existing channels, then any animation outside the frame range will be preserved with keyframes set at the beginning and end of the frame range.

The -r option allows renaming of channel node paths before the channels are loaded from file. In the chwrite command, full paths of the nodes are saved out. This option allows the mapping of animation from one set of nodes into a different set. This renaming function will rename nodes from the data file in the same manner as how the Rename CHOP functions. Here are some examples:

old_path	src_pat	dest_pat	result
/obj/apple	/obj/a*le	/obj/b*	/obj/bpp
/obj/a_to_b	/obj/*_to_*	/obj/ *(1)_to_*(0)	/obj/b_to_a
/obj/Lleg	/obj/L*	/obj/R*	/obj/Rleg

For any nodes that do not match the src_pat (source pattern), then will be loaded into their original path. Note that if the destination node is not found, then loading will stop.

Usage 3: The -i option takes specified file (.chn or .bchn extension) and gives information regarding the file. This option is similar in output to the command line "chinfo" program.

See also: *chwrite chadd chkey chls*.

4.16 CHRENAME

SYNTAX

```
chrename channel new_name
```

EXPLANATION

The chrename command will rename a specified channel to a new name. When a channel is renamed, references to that channel are not automatically updated (see the *opchange* p. 141 command). If a channel which is bound to a parameter is renamed, that parameter will no longer be animated. If a user defined channel is renamed to a parameter channel, the parameter will become animated.

EXAMPLES

```
chrename /obj/geo1/tx translate_x
chrename /obj/geo1/spare1 tx
```

4.17 CHREVERSE

SYNTAX

`chreverse [-f fstart fend] [-t tstart tend]` *channel_pattern*

EXPLANATION

This command reverses selected channels for the frame range you specify. If you specify neither the frame nor the time range, the entire length of the channel is reversed.

EXAMPLE

```
chreverse geo1/r*
```

This command would reverse the rotation channels for the object *geo1*.

4.18 CHRM

SYNTAX

`chrm` *channel_pattern*

EXPLANATION

This action removes the channels you specify.

EXAMPLE

```
chrm geo*/t?
```

In the example above, the translation channels would be removed from the geometry object.

4.19 CHRMKEY

SYNTAX

`chrmkey [-f frame] [-t time]` *channel_pattern*

EXPLANATION

Removes the key frames for the designated channels. You must specify either the time or the frame to identify the key frame you want to remove.

EXAMPLE

```
chrmkey -f 30 geo2/rx
```

This command removes the keyframe (thirty) for the x axis rotation channel for the object *geo2*.

4.20 CHROUND

SYNTAX

```
chround [-v] -a
chround [-v] channel_pattern
```

EXPLANATION

The *chround* command moves keyframes to lie on exact frame values. If the -a (all) option is specified, then all channels in the entire session are scanned for keys that require shifting. If a channel pattern is given instead, then only the specified channels are scanned.

The -v (verbose) option causes the command to report all keys that are modified, showing the old an new frame positions.

This command is useful after changing the frame rate (FPS) of an animation which causes keyframes at the old frame rate to no longer lie on exact frames at the new frame rate.

4.21 CHSCOPE

SYNTAX

```
chscope [-w] [-c|-e @group_name] [channel_pattern ...]
```

EXPLANATION

The *chscope* command will set or modify the channel scope according to the patterns specified. Patterns may be prefixed by a '+' or ',' character to add channels to the current scope or by a '-' to remove channels from the scope. No prefix or an '=' character will set the scope to the given pattern. A channel group may be specified as '@group_name'.

If the -c (collapse) option is specified, then all channel group name patterns will be scoped into the Dopesheet as a single row. Similarly, the -e (expand) option will scope and expand all channel group name patterns into separate rows in the Dopesheet.

If the channel editor is open when this command is executed it will load the new channel scope. Multiple patterns may be specified, in which case an implicit '+' operation is assumed between each argument.

Note that the results will not be visible if the channel editor is not currently open. Specify the -w option to open a channel editor if it is closed.

If no scope patterns are specified this command will list all currently scoped channels from the current folder on down.

EXAMPLES

The following three examples are equvalent:

```
chscope tx+ry+sz
chscope tx,ry,sz
chscope tx ry sz
```

Othere usages:

```
chscope geo1/*-geo1/r?
chscope +light1/t?
chscope -light1/ty
```

4.22 CHSTRETCH

SYNTAX

```
chstretch [-f fstart fend] [-t tstart tend] [-F nframes]
[-T nseconds] [-v] [-p pivot_frame] [-s] channel_pattern
```

`-F nframes`	specifies the number of frames to add or subtract
`-T nseconds`	specifies the number of seconds to add or subtract
`-f start end`	specifies the frame range to stretch
`-t start end`	specifies the time range to stretch
`-v`	enables verbose mode for the command

EXPLANATION

This command stretches the channels specified. Either the -F or -T option must be specified to indicate how much time to add or subtract (if the value is negative), from the channel.

If the -f or -t options are specified, the channel will only be stretched in the specified range. If neither the -f or -t options are specified, then the whole channel will be stretched. If the -v option is specified, the command will be verbose about what channels are stretched.

EXAMPLE

```
chstretch -F 10 geo1/tx
```

Adds 10 frames to *geo1/tx*. The frames are added so that each segment between keyframes grows in proportion to the length of the channel.

```
chstretch -f 30 60 -T 4.1 -v geo1/*
```

Adds 4.1 seconds for all channels of geo1, however, it adds the time by moving keyframes between frame 30 and 60.

Warning: If the time/frame range specified does not fall directly on keyframes, then the following segments may also be adjusted slightly to accommodate the stretch.

For example, if there are keyframes only at frame 1 and frame 150, and the frame range specified is from frame 40 to 90, the keyframe at frame 150 will be adjusted to allow for a proper stretch.

Also, if time is being removed, and more time is specified than exists over the frame range, then some keyframes may not move as expected. i.e. it is not possible to reverse keyframes by removing more seconds than exist in the channel.

```
chstretch -f 30 40 -F -30 geo/tx
```

The command shown above will cause any keyframes between frame 30 and 40 to be squashed to frame 30 because it is impossible to remove 30 frames from the 10 frames specified.

4.23 CHWRITE

SYNTAX

```
1. chwrite [-f fstart fend] channel_pattern...filename.{chan,bchan}
2. chwrite channel_pattern ... filename.{chn,bchn}
```

EXPLANATION

Usage 1: This command will write out the specified channels as columns of raw values with one sample per frame across the given frame range. If no range is given the current global animation time range will be used. Channels will be output in columns in the order specified. The output will be saved as ASCII data unless the file has a suffix of ".bchan", in which case binary format will be used. The order of channels resulting from a pattern match is not well defined.

Note: Channels need not be raw in order to save them using chwrite. The channel values will be sampled at the current frame rate.

Usage 2: This command writes out the specified channels along with their full node paths into the specified file as keyframe data. The extension must be one of .chn (ASCII) or .bchn (binary). The chread command can be used to read these files.

See also: *chread, chadd, chkey, chls* .

4.24 HIP

SYNTAX

```
hip [unix_path]
```

EXPLANATION

If no path is specified, this command displays the current setting of the *$HIP* variable. If a path is specified, it sets the *$HIP* variable to the path.

EXAMPLE

```
hip /usr/people/helene/jobs
```

Sets the $HIP environment variable to: */usr/people/helene/jobs* .

4.25 MREAD

SYNTAX

```
mread [-m merge_pattern | -M] [-c] [-o] filename
```

EXPLANATION

Reads a hip file with the following options:

`-m`	The -m option *merges* the specified file into the current HIP file. A pattern is specified to indicate which sections of the file should be merged in.
`-M`	The -M option is a shortcut for *-m* * which will merge the entire contents of the specified file.
`-c`	If the -c option is specified, a merge will not be done, but instead, a list of *collisions* is reported. Collisions occur when an object in the merge file has the same name as an existing object in the HIP file.
`-o`	If the -o option is specified, the merge will attempt to *overwrite* the nodes whose names collide with those in the current session.

EXAMPLES

```
mread job3.hip # Replace current HIP file with job3
mread -m * job3.hip # Merge in everything from job3
mread -m *geo* job3.hip # Merge in all ops which match *geo*
```

See also: *mwrite, opread, source*.

4.26 MWRITE

SYNTAX

```
Usage1:  mwrite [-i] [filename]
Usage2:  mwrite -c filename
```

EXPLANATION

Write out a hip file containing the entire active session.

`-i`	If the -i option is specified, then the filename is automatically *Incremented*.
`-b`	If the -b option is specified, a numbered backup is created. That is, if the filename already exists, then it is renamed to a filename containing the next number in the sequence before saving. The -b and -i options are exclusive.
`-c`	If the -c option is specified, then the filename is mandatory and a partial hip file will be saved containing only the animation channels. To load such a channels-only hip file into a current session use the command: `mread -o -M <filename>`.

See also: *mread, opwrite, opscript*.

EXAMPLE

```
mwrite -i example.hip
```

The command above writes and increments the motion file *example.hip*.

4.27 KINCONVERT

SYNTAX

```
kinconvert
```

DESCRIPTION

Create InverseKinematic CHOPs for all bone objects that have a solver type other than ’none’. This command easily updates old (i.e. 3.0) .hip files to the new structure for performing IK.

4.28 NETEDITOR

SYNTAX

```
neteditor <options> [-d <desktop_name>] pane1 ...
```

EXPLANATION

Allows different options of the Network Editor to be set. A valid pane name must be specified. If no desk name is given, it will assume the current desk.

OPTIONS

`-d <string>`	Desktops to operate on. If blank, it will default to the current desktop.
`-x 0|1`	Display group dialog.
`-p 0|1`	Display parameters.
`-c 0|1`	Display the color palette.
`-e 0|1`	Display expose flag in list mode.
`-n 0|1`	Display operator names.
`-o 0|1`	Display the network overview
`-s 0|1`	Toggle connection style (direct or not)
`-z 0|1`	Minimize operator bar.
`-G <float>`	The split fraction for groups
`-P <float>`	The split fraction for parameters
`-S order`	Set the sort order. The order may be one of: *user* = User defined order *alpha* = Alphabetic *type* = Operator type *hier* = Hierarchical
`-v path x y scale`	Changes how the network specified by path should be displayed. The x and y refer to panning positions, and the scale to the zoom level.

4.29 NEXTKEY

SYNTAX

```
nextkey [-s | -c channel_list | -p]
```

If no arguments are specified, nextkey will take you to the next keyframe using the current method of key seeking.

`-p`	goes to the Previous key.
`-s`	seeks the next key by using the currently Scoped channels (i.e. the Channel List’s selection).
`-c`	seeks the next key by using keys from the specified channels.

4.30 OBJPARENT

SYNTAX

```
objparent [on|off|useflag]
```

EXPLANATION

With no arguments, the command returns the state of the "Keep Position when Parenting" option. With an argument, the command will set the specified option. Objects which are re-parented (using opwire) when the option is turned "on" will always maintain their world space position. If the option is "off", then it will never change the object's position. If the option is "useflag", then positioning of objects when re-parented will depend upon the *keeppos* (Keep position when parenting) parameter of the object.

4.31 OPADD

SYNTAX

```
opadd [-n] [-v] [type [name] [name2...]
```

`-n`	Suppresses initialization.
`-v`	The *-v* option causes the *opadd* command to display the name of the operator that was created.

EXPLANATION

This command lists all valid operators. If you specify the type, you add an operator of that type; if you designate a name, the operator is given that name. Before entering this command, you first have to go to the folder specific to that OP type.

You can specify any number of names after you have specified the type. For instance, `opadd geo arms legs torso hands feet` creates five geometry objects with those names.

When any OP (i.e. SOP, COP) is created a script can be run. This can be used to initialize certain parameters or create other OPs. The directories where the scripts are located are as follows:

OP Type	**Script Directory**
`Object`	`objects/`
`SOP`	`sops/`
`Shaders`	`materials/`
`Texture OPs`	`tops/`
`COP networks`	`comps/`
`COPs`	`cops/`

The script which is run is determined by the type of the OP being added.
For example, when adding a *geo* object, the script will be found in *objects/geo.cmd*

There is one argument passed to the script which is the name of the object.

These scripts are only run when you add an OP interactively, or add one via the *opadd* command. The *opadd* command has an option to prevent the script from being run *-n*.

EXAMPLE

```
opadd cam eyeball
```

The above command adds a camera object (cam) named *eyeball* to the working environment.

If you had a geometry object (geo1) and typed:

`` set test = `run(“opadd -v geo geo1”)` `` and then typed `echo $test` , `geo2` would be displayed.

4.32 OPCF

SYNTAX

`opcf` *op_path*

EXPLANATION

Changes the working operations directory to that specified in the path portion of the command. See also: *ucd*.

EXAMPLE

```
opcf /n/usr/doug
```

This command line changes the working directory’s path.

4.33 OPCHANGE

SYNTAX

```
opchange from_pattern to_pattern
```

EXPLANATION

This will search all operators for the from_pattern. If the pattern is found, it will be replaced with the to_pattern. All parameters of all operators will be searched for the pattern

The -s option causes *opchange* to operate silently.
Normally *opchange* outputs all the information about the changes being made.

EXAMPLE

```
opchange plastic constant
```

Changes the word plastic to the word constant wherever it is found.

See also: *opfind*

4.34 OPCHANGETYPE

SYNTAX

```
opchangetype [-n] [-p] [-c] -t operator_type operator_pattern
```

This will change the given operators to the specified type.

OPTIONS

`-n`	Keep name
`-p`	Keep parameters
`-c`	Keep network contents

EXAMPLE

```
opchangetype -n -p -t null /obj/logo
```

See also: *opadd, opget*, *opparm*, *opset*.

4.35 OPCOOK

SYNTAX

```
opcook [-f frame_range] [-i frame_inc] [-v] object
```

EXPLANATION

This command will cook, or process, any OP for a specified range. This may have uses for particle systems etc. When specifying the object, you either have to be in the correct directory (*opcd /obj* for object cooking) or specify implicitly the object to be cooked.

EXAMPLE

```
opcook -f 1 35 -i 1 -v /obj/geo1/particle1
```

Cooks a specific SOP (particle1) for the indicated frames.

4.36 OPCP

SYNTAX

```
opcp operator1 [operator2...] destination
```

EXPLANATION

This will copy the specified operators. If the destination directory is specified then the operator(s) are copied there. If not, the operator will be copied to a new operator with the destination name (a new name will be generated if there's already an object of that name).

EXAMPLE

```
opcp geo*
opcp /obj/geo1/* /obj/geo2
opcp geo1 fred
```

The first example copies all operators that start with *geo* to the current directory. The second example copies all the operators in *geo1* to *geo2*. The third example copies *geo1* to *fred*.

See also: *opname* p. 149.

4.37 OPDEPEND

SYNTAX

```
opdepend [-b] [-i] [-I] [-o] [-u idx] [-e] [-p] [-s] [-l level] node
   or opdepend {-n | -N} [-b] [-l level] node
```

EXPLANATION

Lists all the operators that are either dependent on this object or that this object depends on. The -i (inputs) option lists all OPs that are the inputs to this node (i.e that this object depends on). The -o (outputs) list all ops that depend on this node.

The -e option specifies “extra” inputs to this object. For example, a Texture TOP that references a COP network for its source image.

Note: To get correct results, you must cook the network first.

OPTIONS

`-b`	the output does not show full paths.
`-l`	Which level to descend to in the hierarchy.
`-i`	lists all OPs that are inputs to the node.
`-I`	lists all OPs that are indirect inputs to the node.
`-o`	lists all OPs that are outputs of the node.
`-O`	lists all OPs that are extra outputs of the node (i.e. list who depends on the node).
`-u`	with the -o option, specifies the index of the output to look at when finding outputs of the node.
`-e`	lists all extra (reference) inputs to the node (for example a CHOP that references a COP network for its channels).
`-p`	select the nodes specified.
`-s`	silent mode – no output to the textport.
`-n`	lists all name references starting from this node.
`-N`	lists all name dependents starting from this node.

EXAMPLES

```
opdepend -i -e  /mat/blue_plastic
opdepend -i -o -e /obj/geo1
opdepend -i -p -s /obj/logo
opdepend -n /obj/light1
opdepend -N /obj/logo
```

4.38 OPFIND

SYNTAX

```
opfind string
```

EXPLANATION

This command will search all nodes in the project to find the string specified. The string is case sensitive and also must match a "word".

EXAMPLE

```
opfind Welcome
```

In this case, opfind will look for the string "Welcome". The search is case sensitive, and looks for an exact match, therefore "welcome" or "elcome" would not display what you are looking for.

4.39 OPGADD

SYNTAX

```
opgadd group_name [second_name...]
```

EXPLANATION

Opgadd creates one or more operator groups. Use the *opgop* command to add and remove operators from each group.

See also: *opgls*, *opgop*, and *opgrm*. These commands allow you to perform operations on groups associated with a network. The groups can contain a list of operators from within the network.

Note: Names may contain only alphanumeric characters or underscores. The name must contain at least one alphaBETIC character or at least one underscore.

The following are legal op names: _ _1 1_ 12345a a12345 hello .
The following are illegal op names: 1 123 a:b fu-bar ?%!arrgh

4.40 OPGET

SYNTAX

```
opget [-q] [flag] ... operators ...
```

EXPLANATION

The *opget* command queries individual operator flags and outputs the result as an 'opset' command. The -q (quiet) option supresses messages from being displayed on an unknown flag or operator.

OPTIONS

`-d`	Display.

`-r`	Render.
`-t`	Template.
`-b`	Bypass.
`-l`	Lock (off, soft, hard (or equivalently ’on’)).
`-e`	Expose.
`-h`	Highlight.
`-f`	Footprint.
`-s`	Save data in motion file.
`-u`	Unload data after cook.
`-c`	Compress icon,
`-C`	Set to be the ’current’,
`-p`	Set the ’picked’ flag,
`-x`	Object pivot axes (Objects only)
`-y`	Xray (Objects only).
`-a`	Audio (CHOPs only).
`-o`	Export (CHOPs only).

EXAMPLE

```
opget -d geo*
opget -p -r light*
opget geo*/*
```

See also: *opadd*, *opchangetype*, *opparm*, *opset*.

4.41 OPGETINPUT

SYNTAX

```
opgetinput [-n num | -o outputop [-u outputidx]] inputop
```

EXPLANATION

With the -n option, this function returns the name of the node that is attached to input num of the inputop. It returns the empty string if no input is attached to the given input. With the -o option, this function returns the input number of the inputop that is connected to the outputop. If the outputop is not connected to the inputop, -1 is returned. If the outputop is connected to more than one input of the inputop, the highest input number is returned. When the -o option is specified, the -u option can also be used to specify which output index of the outputop must be connected to the inputop. The default for this option is -1, which indicates that any output of the outputop should be considered.

4.42 OPGLOB

SYNTAX

`opglob` *object_pattern*

EXPLANATION

Expands the pattern according to the parameters you specify and then displays the results of the expansion.

EXAMPLE

```
opglob geo...
```

4.43 OPGLS

SYNTAX

```
opgls [-l][pattern...][-g] [group_pattern]
```

EXPLANATION

Lists operator groups. The *-l* option will list the names of the operators in each group as well. The *-g* option generates commands which can be used to recreate the group(s) specified. This option can be used to implement scripts for specific groups. If a pattern is specified, then only groups which match the pattern will be listed.

4.44 OPGOP

SYNTAX

```
opgop group_name [set, add, remove] name_pattern
  [second_pattern...]
```

EXPLANATION

This command performs operations on operator groups (i.e. allows addition or removal of operators from the group). *group_name* is the name of the operator group to modify. It can be one of:

`set`	Set the contents of the group
`add`	Add operators to group

`remove`	Remove operators from group

The *name_pattern* specifies a list of operators to be added/removed from the group.

EXAMPLE

```
opgop group1 set geo* opgop group1 add light1 light2 light3 opgop
  group1 remove geo4
```

The above command sets the contents of group one to all geo objects, adds three light objects to that group, and then removes geo4 from the group.

4.45 OPGRM

SYNTAX

```
opgrm group_pattern [second_group...]
```

EXPLANATION

Removes the specified operator groups from the current network.

4.46 OPHELP

SYNTAX

```
ophelp [operator] [operator type]
```

EXPLANATION

Displays help for an operator or for an operator type. *Operator type* is of the form <class>/<type>

Available operator classes are:

`sop`	Surface Operators
`obj`	Objects
`ch`	CHOPs
`mat`	Shaders
`pop`	Particle Operators
`top`	Texture Operators
`cop`	Composite Operators
`out`	Drivers

EXAMPLE

`ophelp sop/add`	Displays the help for add SOPs.
`ophelp /obj/geo1`	Displays the help for the geo objects since */obj/geo1* is a geo object.

4.47 OPINFO

SYNTAX

```
opinfo operator [operator...]
```

EXPLANATION

Displays information about the operator. This typically displays the information contained when clicking on the Info pop-up of an operator tile.

4.48 OPLAYOUT

SYNTAX

```
oplayout [-p] [width] [height]
```

EXPLANATION

Arranges the operators according to your specifications in the options of the command string. If the -p option is used, this command will only place the selected operators where there is space available in the Layout area.

`-p`	lays out only those operators that have been selected
`width`	specifies the width in inches
`height`	specifies the height in inches

4.49 OPLOCATE

SYNTAX

```
oplocate [-d] [-x xval] [-y yval] object_pattern
```

EXPLANATION

This command locates the operators you specify in the worksheet view. The bottom left-hand corner of the worksheet has the coordinates 0,0 independent of any scrolling you might have done.

`-d`	The -x and -y are the delta values and are added to the current location
`-x`	Specifies the X position
`-y`	Specifies the Y position

If neither x nor y are specified, the current position is displayed.

EXAMPLE

```
oplocate geo2
```

This command provides you with the location of geo2 in the Layout Area, expressed in x,y coordinates.

4.50 OPLS

SYNTAX

```
opls [options] [object_pattern]
```

EXPLANATION

This command lists the operators you have specified in the *object_pattern*.

`-a`	Show all files (including hidden items).
`-l`	List items in *long** format.
`-R`	Recursively list contents of Operators. For example, list all SOPs contained in the objects or all operators in sub-networks.

* The *long* format lists eight flags along with the operator and the number of nodes it contains. The meanings of the flags are:

`d`	Display flag
`r`	Render flag
`t`	Template flag
`l/L`	Soft lock/Hard lock
`e/h`	Exposed/Hidden
`b`	Bypassed
`c`	Current
`s`	Selected

EXAMPLE

```
opls geo*
opls /*/*/*
```

These two examples lists all geo(metry) operators, and all operators, respectively.

4.51 OPNAME

SYNTAX

```
opname old_name new_name
```

EXPLANATION

Renames the specified operator.

Note: Names may contain only alphanumeric characters or underscores. The name must contain at least one alphaBETIC character or at least one underscore.

The following are legal op names: _ _1 1_ 12345a a12345 hello .
The following are illegal op names: 1 123 a:b fu-bar ?%!arrgh

EXAMPLE

```
opname harv.geo nell.geo
```

This command gives the *harv.geo* object the new name *nell.geo*.

4.52 OPORDER

SYNTAX

```
oporder operator1 [operator2...] before_operator
```

EXPLANATION

The *oporder* command will reorder the specified operators such that they appear before the *before_operator*. These changes will be reflected in the *User Defined* sorting option for lists of operators in the user interface.

EXAMPLE

```
oporder geo1 cam2
```

This example places the geo1 operator before the cam2 operator in the Object Editor’s list view.

4.53 OPPARM

SYNTAX

```
opparm [-q] operator_pattern [-v] parameters
opparm [-q] -c operator_name parameter [parameter_choice]
opparm [-q] -d operator_pattern [-v] parameter_names
```

EXPLANATION

The first two instances of *opparm* will set parameters for the given operator. The third instance (-d) displays the contents of the listed parameter names for the given operator.

The parameters are operator dependent and thus are different for each object type. To get a list of parameters, please try ’opscript’.

If the -q (quiet) option is used, no warning or error messages are displayed.

When using *opparm* to set parameter values, the animation channels of the parameters are deleted by default. But if the -v option is specified and the value specified for the parameter is a number, then the channels aren't deleted. The opparm command will behave as if the values had been entered into a parameter dialog at the current time.

When using "opparm" to query parameters contents, the animation channels of the parameters will be displayed by default. Use the -v option to evaluate the channels at the current time and display parameter values.

clicking a virtual button (-c)

The *-c* option allows the scripting language to "click" these types of buttons for you.

If the parameter is a button which causes initialization or execution, setting the data does not invoke the function associated with the button. For these buttons, it is possible to use the second usage (-c) to invoke the function. Examples of these types of buttons are in the Creep SOP (initialize), the Skeleton SOP (clear capture regions), and many render output drivers (the Render button).

If the -c option is specified on the wrong type of parameter, a usage message is printed out. If an invalid parameter choice is specified, a list of valid choices is also printed.

EXAMPLES

```
opparm geo1 t ( 1 2 3 )
opparm -c /obj/geo1/creep1 Initialize initfill
opparm -d geo1 t
```

See also: *opadd*, *opchangetype*, *opget*, *opset*.

4.54 OPPWF

SYNTAX

```
oppwf
```

EXPLANATION

Displays the current working folder. Similar to the UNIX *pwd* command.

4.55 OPRAMP

SYNTAX

```
Form1: opramp operator_name pos r g b a
Form2: opramp -r operator_name position
```

The first form of this command's syntax adds a key in the color ramp of the specified OP (if it contains a color ramp) by specifying the name, position and the red, green, blue, and alpha values at that key.

The second form removes a key at the specified position (pos ranges from 0-1). This command is mostly useful for COP Ramps and Two tone shaders. The *-r* option removes a key frame from the ramp. The position of the key should be a value between zero and one. If you set a key within 0.01 units of an existing key, that key's color will be altered. You cannot remove the ramp's first and last keys. For a list of available keys, use the *opscript* command on the shader you want to modify.

The *opramp* command is a more general implementation of the *matramp* command, which is actually an alias for *opramp*. This allows control of COP Ramps from scripts. See also: *matramp*.

4.56 OPREAD

SYNTAX

`opread` *filename*

EXPLANATION

Reads the contents of the file into the current directory *oppw*. The file you specify should have been created using the *opwrite* command.

EXAMPLE

```
opread environment.hip
```

Reads the file *environment.hip* to the current directory.

4.57 OPRM

SYNTAX

`oprm` *object_pattern*

EXPLANATION

Removes the specified objects.

EXAMPLE

```
oprm geo*
```

This command removes all objects that adhere to the *geo** pattern.

4.58 OPSAVE

SYNTAX

```
opsave [-f start end] [-i inc] object filename
```

EXPLANATION

Opsave saves the contents of the designated object to the filename you specify. Options include the ability to specify a frame range as well as the increments to be saved.

EXAMPLE

```
opsave -f 1 10 -i 2 twist1 twist1\$F.rib
```

This command saves every other frame over the range one through ten for the object *twist1* to the file *twist\$F.rib.*

Note: opsave only saves the geometry belonging to a SOP.

4.59 OPSET

SYNTAX

```
opset [flags] on|off [node_name]
```

EXPLANATION

Sets the options for the specified node's flags. The flags are as follows:

`-d`	Display
`-r`	Render
`-t`	Template
`-b`	Bypass
`-l`	Lock (off, soft, hard / on)
`-e`	Expose
`-h`	Highlight
`-f`	Footprint
`-s`	Save data in motion file
`-u`	Unload data after cook (CHOPs only)
`-c`	Compress icon
`-C`	Current
`-p`	Picked
`-S`	Selectable in Viewport (objects only)
`-x`	Object pivot axes (objects only)
`-y`	X-Ray (objects only)
`-a`	Audio (CHOPs only)
`-o`	Export (CHOPs only)

EXAMPLE

```
opset -d on geo*
opset -l on -s on geo*/*
```

The first example toggles the display flags on for all geometry objects. The second example toggles both the save data and lock features on for all geometry objects.

4.60 OPUNWIRE

SYNTAX

```
opunwire node input_number [input_number...]
```

EXPLANATION

This command disconnects inputs from an operator node.

EXAMPLE

```
opunwire merge1 0 3 12
```

The above command disconnects inputs 0, 3, and 12 from the merge1 operator.
See also: *opwire* below.

4.61 OPUPDATE

SYNTAX

opupdate

EXPLANATION

This command causes all OPs which reference external disk files to re-cook if the referenced disk file has been changed. Any cached textures or geometry files which are out of date will be re-loaded.

See also: *texcache* p. 161, *geocache* p. 159.

4.62 OPWIRE

SYNTAX

```
opwire [-n] object -input_number wire_object [-input_number...]
```

EXPLANATION

This command ’wires’ or connects the output of one OP node to the input of another node. It is recommended that for multiple input OPs like the merge SOP that the inputs are filled up consecutively. Use *-input_number* to specify which input number to wire.

The *-n* option will ignore the *Keep Position when Parenting* flag on objects.

EXAMPLE

```
opwire twist1 -0 box1
```

Connects *twist1* to the first input of *box1*.

```
opwire box1 -0 merge1 ; opwire box2 -1 merge1
```

Connects *box1* to the first input of *merge1*, *box2* to the second input of *merge1*.

See also: *opunwire* above.

4.63 VARCHANGE

SYNTAX

```
varchange [-v] [-V]
```

EXPLANATION

When a variable value changes, the OPs which reference that variable are not automatically cooked. Running the *varchange* command causes all OPs which use a variable that has changed to be re-cooked. The -v and -V options represent different levels of verbosity. -v will cause all changed variables to be listed. -V will also list all the OPs which get changed due to the variable changes.

5 COMMANDS TO MANAGE TIME GROUPS

5.1 TMGADD

SYNTAX

```
tmgadd -t time | -f frame group_name [second_name...]
```

EXPLANATION

Creates one or more time groups.

EXAMPLE

```
tmgadd -f 35 group_one
```

Creates time group group_one at frame 35.

5.2 TMGLS

SYNTAX

```
tmgls [-l] [-k] [pattern...]
```

EXPLANATION

Lists time groups.

`-l`	Lists time groups. The -l option will list the full contents of a time group. If a pattern is specified, then only groups which match the pattern are listed.
`-k`	The -k option lists only some of the keys belonging to the time group. If a pattern is specified, then only groups which match the pattern are listed.

EXAMPLE

```
tmgls -l newt*
```

Lists the contents of groups matching the pattern *newt**.

5.3 TMGNAME

SYNTAX

```
tmgname old_name new_name
```

EXPLANATION

Renames the specified time group to the new name.

5.4 TMGOP

SYNTAX

```
tmgop -t time_from time_to | -f frame_from frame_to group_name
  operation channel_pattern [second_pattern...]
```

EXPLANATION

This command performs operations on time groups (i.e. allows addition or removal of keyframes from the group).

options

`group_name`	The name of the time group to modify	
`operation`	Can be one of:	
	set	Sets the contents of group
	add	Add keyframes to group
	remove	Remove keyframes from group
`channel_pattern`	specifies a list of channels to work on.	

EXAMPLE

```
tmgop -f 1 50 group1 set /o*/g*/r?
```

Sets the contents of time group *group1* to the rotate channels (i.e. r?) of all objects that start with the letter “g” (i.e. g*).

```
tmgop -f 1 50 group1 add /o*/g*/t?
```

Adds channels to group1.

```
tmgop -f 1 50 group1 remove /o*/g*/tx
```

Removes /*tx* channels from group1.

5.5 TMGRM

SYNTAX

```
tmgrm group_pattern [second_group...]
```

EXPLANATION

Removes the specified time groups.

5.6 TMSHIFT

SYNTAX

```
tmgshift [-a] [-g] [-r] -f numframes | -t time group_name
```

EXPLANATION

Shift the time group by a certain number of frames or by an increment of time.

`-a`	Will shift everything absolutely i.e. *tmshift -a -f 55 T1* will move the time group to frame 55
`-g`	Will only shift the time group and not its members
`-r`	The shift will only occur if it can within the keyframe boundaries.

6 OPERATOR-SPECIFIC COMMANDS

This set of commands deals with specific types of operators. Some operators have special features which don't fit into the paradigm of the general commands, so specific commands have been created to access these functions.

6.1 COMPFREE

SYNTAX

```
compfree
```

EXPLANATION

This command releases all cached images from all composite networks in the current Houdini session. Use after working in the composite editor, when current image caches are no longer required.

6.2 GEOCACHE

SYNTAX

```
geocache [-s] [-l] [-c] [-a 0|1] [-m max]
```

EXPLANATION

This command allows access to the internal geometry cache.
The geometry cache is used by VEX geometry functions.

query options

-s	See the current settings
-l	List the contents of the geometry cache

control options

-c	Clear the cache
-n	Clear the cache only if newer files exist on disk.
-a	Turn auto-flushing of geometry files on or off. Leaving the geometry in the cache (i.e. no auto-flushing) improves performance at the cost of extra memory.
-m	Specify the cache size (in Kb). This defaults to 8192 (8 Mb).

See also:*texcache* p. 161, *texcache* p. 161, *opupdate* p. 154.

6.3 MATRMAN

SYNTAX

```
matrman material name
```

EXPLANATION

Saves the material specified in RenderMan Shader Language. This provides a simple way of generating RenderMan shaders very quickly. The output of this command may be re-directed to a file.

EXAMPLE

```
matrman phong1 > /tmp/phong1.sl
```

This command saves the material *phong1* as a RenderMan shader in the specified directory.

6.4 MATRAMP

SYNTAX

```
Form1: matramp material_name position r g b a
Form2: matramp material_name -r position
```

EXPLANATION

The first form of this command's syntax sets a key in a shader's ramp by specifying the name, position and the red, green, blue, and alpha values at that key.

The second form removes keys from a shader's ramp. The *-r* option removes a key frame from the ramp. The position of the key should be a value between zero and one. If you set a key within 0.01 units of an existing key, that key's color will be altered. You cannot remove the ramp's first and last keys. For a list of available keys, use the *opscript* command on the shader you want to modify.

EXAMPLE

```
matramp twotone1 0.5 18 2 6 1
```

This command generates a key on the Two tone shader's ramp at the halfway mark. Its red, green, blue, and alpha values are listed sequentially, producing a headache-inducing color ramp.

6.5 RENDER

SYNTAX

```
render [-V] driver_name
```

EXPLANATION

This command causes an output driver to render.

The -V option causes a line of output as each frame begins rendering.
For example: `11:46:06 Rendering frame 1 (1 of 3)`

EXAMPLE

```
% render mantra1
```

This will cause the output driver named *mantra1* to render.

6.6 TEXCACHE

SYNTAX

```
texcache [-s] [-l] [-c] [-a on|off] [-r xres yres] [-m max]
```

EXPLANATION

This command allows access to the internal texture map cache used by Houdini. The texture map cache is used when displaying texture maps in the Viewports (or also for background images when texturing is turned on).

query options

`-s`	See the current settings.
`-l`	List the contents of the texture map cache.

control options

`-c`	Clear the cache.
`-a`	Turn auto-flushing of maps on or off. Leaving maps in the cache improves performance at the cost of extra memory.
`-r`	Specify the maximum resolution of an image for the cache. This does not include maps stored in COPs.
`-m`	Specify the cache size (maximum number of images stored in the cache).

7 TIME RELATED COMMANDS

7.1 FCUR

SYNTAX

```
fcur [frame]
```

EXPLANATION

Sets the current frame to the frame specified.

EXAMPLE

```
fcur 15
```

Sets the current frame to frame 15.

7.2 FPLAYBACK

SYNTAX

```
fplayback [-i on|off] [-r on|off] [-f factor] [-s step_size]
```

EXPLANATION

These commands are used to save the state of the playbar to motion files. Now, the information will be retrieved from motion files when they are loaded.

Options:

`-i`	Integer frame values on or off
`-r`	Real-time on or off
`-f`	Real-time factor
`-s`	Non-real time frame step size

7.3 FPS

SYNTAX

```
fps [frames_per_sec]
```

EXPLANATION

Sets the output rate to the value you specify. If the value isn't specified, the current frames per second rate is used. See also: *chround*.

EXAMPLE

```
fps 30
```

The command above sets the output rate at thirty frames per second.

7.4 FRANGE

SYNTAX

```
frange [start end]
```

EXPLANATION

Specify the frame range for playing the animation. If the range is not specified, then the current range is displayed.

EXAMPLE

```
frange 20 60
```

7.5 FSET

SYNTAX

```
fset [nframes]
```

EXPLANATION

Sets the frame range is set to the number of frames specified. If no frame count is specified, the current frame range is displayed.

EXAMPLE

```
fset 30-100
```

The command above specifies that frames thirty to one hundred should be displayed.

7.6 FTIMECODE

SYNTAX

```
ftimecode [timecode]
```

EXPLANATION

Sets the current frame to the timecode specified. If none is specified, the current frame is displayed in timecode format.

7.7 TCUR

SYNTAX

```
tcur [time]
```

EXPLANATION

Sets the current frame to the time specified. If you don't set the time, the current time is displayed.

7.8 TSET

SYNTAX

```
tset [time]
```

EXPLANATION

Sets the frame range to the time specified. If no time is specified, the current time is displayed.

8 INTERFACE RELATED COMMANDS

These commands are specific to Houdini's user interface. They have no effect when running from the scripting version of Houdini *(hscript)*, but will produce warnings indicating that information may not be saved correctly.

8.1 QUAD-VIEW REFERENCING CONVENTIONS (FIND: UPDATE!)

The view commands described in this section allow access to the parameters of each of the individual viewports. If, for example, you wanted to change the level of detail in the perspective viewport in the modeler you could type:

```
viewdisplay -l 3 Build.pane1.world.persp1
```

To get a list of valid Viewport names, use:

```
  viewls -v
```

Note: As a convenience, <*viewport name*> can be any regular expression.
Thus, *Build** references *Build.top, Build.front1, Build.persp1* etc.

Currently, <*viewname*> cannot be a pattern, nor can the memory name when referencing a view memory with the form <*viewname*>*:*<*memory name/number*>.

Note that some commands do not support pattern matching of Viewport names:

- *viewcopy* does not support pattern matching for the source viewport
- *viewtransform* does not support pattern matching at all

8.2 ANIMVIEW

SYNTAX

```
animview [-v port|edit|both] [-c graph|table|both]
                        [-S <main_split>] [-s <chooser_split>]
```

EXPLANATION

Sets the Animation Editor screen layout options. This command is primarily used to save and restore the Animation Editor layout in a .hip file. The split fractions set the position of the screen area split bars, and are floating point values between zero and one.

OPTIONS

`-v <layout>`	Set Main View layout.
`-c <layout>`	Set Channel Editor layout.
`-S <split>`	Set main screen split fraction.
`-s <split>`	Set channel chooser split fraction.

8.3 AUDIOPANEL

SYNTAX

```
audiopanel [options]
```

EXPLANATION

Change the parameters for the audiopanel (in the *Options* menu).
If no options are specified, the current settings are displayed.

OPTIONS

<table>
<tr><th></th><th></th></tr>
<tr><td><code>-s n <name></code></td><td>Set the network menu.</td></tr>
<tr><td><code>-s o <name></code></td><td>Set the CHOP menu.</td></tr>
<tr><td><code>-s r on|off</code></td><td>Set the scrub repeat toggle.</td></tr>
<tr><td><code>-s s <value></code></td><td>Set the scrub sustain value.</td></tr>
<tr><td><code>-s f <frequency></code></td><td>Set the scrub rate value.</td></tr>
<tr><td><code>-t p reverse|stop|play</code></td><td>Set the test play direction.</td></tr>
<tr><td><code>-t l on|off</code></td><td>Set the test loop toggle.</td></tr>
<tr><td><code>-t r on|off</code></td><td>Set the test rewind toggle.</td></tr>
<tr><td><code>-o m on|off</code></td><td>Set mono output toggle.</td></tr>
<tr><td><code>-o t on|off</code></td><td>Set volume tied toggle.</td></tr>
<tr><td><code>-o u on|off</code></td><td>Set meter toggle.</td></tr>
<tr><td><code>-o l <value></code></td><td>Set the left volume.</td></tr>
<tr><td><code>-o r <value></code></td><td>Set the right volume.</td></tr>
<tr><td><code>-o d 0 | 1 | 2</code></td><td>Turn off audio | Timeline mode | Test mode.</td></tr>
</table>

Note: There can only be one -s, -t or -o option specified per command.

8.4 CHLAYOUT

SYNTAX

```
chlayout options network_pattern
```

EXPLANATION

The *chlayout* command is used to customize all aspects of each specific CHOP Network. If no options are specified, the current settings are displayed.

OPTIONS

`-a <val>`	Set the Notes CHOP index.
`-b <val>`	Set the number of graphs (less one).
`-B <val>`	Graph type: 0=all, 1=per channel, 2=per CHOP, 3=per name.
`-c <string>`	Graph scope exclusions of the form: “chop chanel chop channel...” .
`-d on|off`	Toggles display of dots at each sample of the CHOP channel. Useful for debugging sampling errors by visually examing the individual values of the channel. (shortcut: type D in the CHOP Graph).
`-e on|off`	Display extend regions toggle.
`-g low|medium|high`	Grid density.
`-h on|off`	Display handles toggle.
`-H on|off`	Horizontal adapt toggle.
`-i on|off`	Swap interface toggle.
`-l on|off`	Display labels toggle.
`-L 0|1`	List mode (0=network, 1=list)
`-m <index><val><val>`	Graph height start/end range.
`-n on|off`	Graph disable toggle.
`-o <val> <val>`	Graph dimensions.
`-p on|off`	Scope disable toggle.
`-r h <val><val>`	Graph horizontal start/end range.
`-r b <val> <val>`	Bar height start/end range.
`-s on|off`	Scroll lock toggle.
`-S <val>`	Scroll lock position.
`-t on|off`	Time bar toggle.
`-T 0|1`	List order: 0=alphabetic, 1=user defined.

`-u <UnitType>`	UnitType: “frames”, “samples”, “seconds”.
`-v graph|bar`	Graph viewing mode.
`-V on|off`	Vertical adapt toggle.
`-w on|off`	Full width toggle.
`-W <string>`	Viewport settings script.

8.5 CLOSEPORT

SYNTAX

`closeport port_number`

Closes a communication port created by the *openport* command.

See Also: *openport*; *Stand Alone* > *hCommands* p. 428.

8.6 CPLANE

SYNTAX

`cplane [options] viewers`

Modifies or displays current construction plane parameters.

OPTIONS

-o x y z	Sets origin to (x, y, z)
-n x y z	Sets plane normal to (x, y, z)
-x x y z	Sets plane horizontal axis to (x, y, z)
-u x y z	Sets the up-vector to (x, y, z)
-l [n|x|y|z]	Locks the up-vector to either the plane normal, world X-axis, world Y-axis or world Z-axis.
-s x y	Sets the grid spacing to x units along the X-axis and y units along the Y-axis.
-r a b	Sets the grid ruler to a units along the X-axis and y units along the Y-axis.
-c a b	Sets the # of grid cells to a along X and b along Y.
-d [on|off]	Turns construction plane display on or off

For details on how to specify viewers, type "help viewerformat"

EXAMPLES

```
cplane -o 0 0 0 *
cplane -o 1 1 1 -n 0 0 1 -r 10 10 Build.pane1.world
```

8.7 DOUBLEBUFFER

SYNTAX

```
doublebuffer [on|off]
```

EXPLANATION

This command turns double buffering on or off. If no options are passed to the command, the current state is returned.

Double buffering improves the quality of moving images on a 24 bit monitor. If *Double Buffering* is not used, everything is drawn directly to the screen without any synchronization to the monitor's refresh rate. This causes areas of the screen which are being frequently redrawn tend to be displayed while only partially redrawn, this produces a 'flickering' effect.

When *Double Buffering* is on, the new image is drawn in an offscreen buffer in video RAM and then swapped into the display almost instantaneously, thereby eliminating the display of half-drawn images, and the flicker.

Why not have it on all the time? Because some machines do not have sufficient video RAM to perform double buffering in full (24 bit) colour. The colour resolution must be halved and then still-frame image quality is reduced due to dithering.

8.8 EDITOR

Warning: This command is obsolete since 4.0. Similar commands are: *desk*, *pane*.

8.9 NETEDITOR

SYNTAX

```
neteditor <options> [-d <desktop_name>] pane1 ...
```

EXPLANATION

Allows different options of the network editor to be set. A valid pane name must be specified. If no desk name is given then it will assume the current desk.

OPTIONS

`-d <string>`	Desktops to operate on. If it is blank it will default to the current desktop.

`-x 0|1`	Display group dialog.
`-p 0|1`	Display parameters.
`-c 0|1`	Display the color palette.
`-e 0|1`	Display expose flag in list mode.
`-n 0|1`	Display operator names.
`-o 0|1`	Display the network overview.
`-s 0|1`	Toggle connection style (direct or not).
`-z 0|1`	Minimize operator bar.
`-G <float>`	The split fraction for groups
`-P <float>`	The split fraction for parameters

`-S order`	Set the sort order. The order may be one of:
	user = User defined order
	alpha = Alphabetic
	type = Operator type
	hier = Hierarchical

`-v path x y scale`	Changes how the network specified by path should be displayed. The x and y refer to panning positions, and the scale to the zoom level.

8.10 OMBIND

SYNTAX

```
ombind [-t <type>] [-d <settings>] <instance> <op_parm>
  <manipulator_parm>
```

EXPLANATION

Binds an operator parameter to the movement of a manipulator.

OPTIONS

`-t <type>`	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.
`<instance>`	The label to associate with the bind operation. If the label exists, the new binding is appended to the group of bindings with the existing label.
`<op_parm>`	String specifying both the operator and the parameter

	to be bound, delimited by a colon.
`<manipulator_parm>`	String specifying the manipulator and parameter to bind, delimited by a colon.
`<settings>`	The default settings for this manipulator type when bound to this type of operator. The meaning of these-settings varies between types of manipulators. This option only has meaning when the instance of the manipulator does not yet exist (i.e.it only works for the first ombind command in the set of commands for one-manipulator). To turn the manipulator off by default, use a default setting of "i".

EXAMPLE

```
ombind -t sop "First U" carve:group uisoparm:input
ombind -t sop "First U" carve:domainu1 uisoparm:k
```

The first command binds the group parameter of the carve sop, to the movement of the input parameter of the uisoparm handle.

The "First U" instance is created assuming it is not already created. The domainu1 binding to the manipulator parameter k is appended to the existing "First U" instance.

See also: ombindinfo, omls, omunbind, omwhere, omwrite.

8.11 OMBINDINFO

SYNTAX

```
ombindinfo [-t <type>] <operator>
```

Lists the parameters that are bound to a manipulator for the specified operator.

OPTIONS

`-t <type>`	Network type. Available types are: *obj*, *sop*, *pop*, and *top*. The default value is *sop* if the argument is unspecified.
`<operator>`	The operator.

OUTPUT FORMAT

```
instance manipulator { op_parm->manipulator_parm ...  }
```

EXAMPLE

```
ombindinfo carve
```

Lists the bound parameters of the carve sop operator.

See also: ombind, omls, omunbind, omwhere, omwrite.

8.12 OMLS

SYNTAX

`omls [-t <type>] [manipulator]`

Lists the available manipulators for the given operator type. If the manipulator parameter is specified, this command lists the bindable parameters of the specified manipulator.

OPTIONS

`-t <type>`	Network type. Available types are: *obj*, *sop*, *pop*, and *top*. The default value is sop if the argument is unspecified.
`manipulator`	If unspecified, omls lists the available manipulators. Otherwise a list of bindable parameters for the specified manipulator is displayed.

EXAMPLE

```
omls domain
```

Lists the bindable parameters of the domain manipulator.

See also: *ombind*, *ombindinfo*, *omunbind*, *omwhere*, *omwrite*.

8.13 OMPARM

SYNTAX

`omparm <manip_name> <manip_type> <op_node_name> <settings>`

For the given manipulator (handle) and operator node, set the manipulators's settings to the specified settings string. These settings will be used the next time the user enters the state for this operator node.

The values for the settings are specific to the type of manipulator and are undocumented. This command is used internally to save manipulator settings to the hip file.

See also: *ombind*, *ombindinfo*, *omunbind*, *omwhere*, *omwrite*.

8.14 OMUNBIND

SYNTAX

```
omunbind [-t <type>] <op_parm> [instance]
```

Removes the bindings between an operator and manipulator that match the specified search criteria.

OPTIONS

-t <type>	Network type. Available types are: obj, sop, pop, and top. The default value is sop if the argument is unspecified.
<op_parm>	Specifies the bindings to remove with respect to the bound operator. Legal formats include: operator_name operator_name:parm_name The operator name must be specified.
<instance>	Specifies the label of the manipulator that should be disconnected from the specified operator or parameter.

EXAMPLES

```
omunbind xform
```

Removes all bindings for the xform sop.

```
omunbind xform:tx
```

Removes all bindings to xform's tx parameter.

```
omunbind xform:tx Transformer
```

Removes all bindings of xform's tx parameter that are bound to the Transformer handle.

See also: ombind, ombindinfo, omls, omwhere, omwrite

8.15 OMWHERE

SYNTAX

```
omwhere [-t <type>] [manipulator]
```

Lists the operators bound to the specified manipulator.

OPTIONS

-t <type>	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.

[manipulator]	If unspecified, lists all operators bound to a handle.

EXAMPLE

omwhere pivot

Lists those operators bound to the pivot handle.

See also: ombind, ombindinfo, omls, omunbind, omwrite.

8.16 OMSBIND

SYNTAX

```
omsbind [-t <type>] <op_parm> <selector> <sel_description>
  <sel_prompt> <op_input_index> <op_intput_required>  <primmask>
  <allow_drag> <menu_name> <ast_sel_all>
```

Binds an operator parameter to a selector.

OPTIONS

-t <type>	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.
<op_parm>	String specifying both the operator and the parameter to be bound, delimited by a colon. The parameter specification is optional.
<selector>	The name of the selector type. A list of selectors can be obtained using the omsls command.
<sel_description>	A description of the purpose of the selector. Used with omsunbind.
<sel_prompt>	The string that is displayed in the status area when the selector is active.
<op_input_index>	Index of operator input where the result of this selection should be fed.
<op_input_required>	Specifies if this input is required.
<primmask>	A string representing the types of primitives that can be picked using this selector. This string can consist of one or more primitive types, or primitive types preceded by a "^" to remove that primitive type from the selectable primitive types. The available primitive types are: all, face, poly, nurbcurve, bezcurve, hull, mesh, nurb, bez, quadric, circle, sphere, tube, particle, and meta.

<allow_drag>	Determines if the user is allowed to select and begin manipulation with a single mouse click.
<menu_name>	Name of the operator's "Group Type" parameter (or "" if there is none). This lets the selector set this parameter to "Primitive" if the user selected primitives, "Points" if the user selected points, etc.
<ast_sel_all>	If set to a non-zero value, this indicates that the group parameter requires a "*" to select all geometry. A zero value indicates that the group parameter should be left blank if the whole geometry is selected.

EXAMPLE

```
omsbind -t obj blend objselect "Second Input" "Select the second
  blend input" 1 1 all 0 "" 0
```

Binds a simple object selector to the second input of the blend object.

See also: omsbindinfo, omsls, omsunbind, omswhere, omwrite.

8.17 OMSBINDINFO

SYNTAX

```
omsbindinfo [-t <type>] <operator>
```

Lists the selectors that are bound to the specified operator.

OPTIONS

-t <type>	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.
<operator>	The operator.

OUTPUT FORMAT

```
selector "Selector Label" "Selector Prompt"
  op_parameter op_input_index op_input_required primmask allow_drag
  menu_name ast_sel_all
```

EXAMPLE

```
omsbindinfo carve
```

Lists the bound selectors of the carve sop operator.

See also: omsbind, omsls, omsunbind, omswhere, omwrite.

8.18 OMSLS

SYNTAX

```
omsls [-t <type>]
```

Lists the available selectors for the given operator type.

OPTIONS

-t <type>	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.

EXAMPLE

```
omsls -t obj
```

Lists the selectors that can be bound to object parameters.

See also: omsbind, omsbindinfo, omsunbind, omswhere, omwrite.

8.19 OMSUNBIND

SYNTAX

```
omsunbind [-t <type>] <op> [instance]
```

Removes the bindings between an operator and selector that match the specified search criteria.

OPTIONS

-t <type>	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.
<op>	Specifies the operator from which to remove bindings.
<instance>	Specifies the label of the selector that should be disconnected from the specified operator.

EXAMPLE

```
omsunbind particle
```

Removes all selector bindings for the particle SOP.

```
omsunbind particle "Force Geometry"
```

Removes the selector with the label "Force Geometry" from the particle SOP.

See also: omsbind, omsbindinfo, omsls, omswhere, omwrite.

8.20 OMSWHERE

SYNTAX

```
omswhere [-t <type>] [selector]
```

Lists the operators bound to the specified selector.

OPTIONS

-t <type>	Network type. Available types are obj, sop, pop, and top. The default value is sop if the argument is unspecified.
[selector]	If unspecified, lists all operators bound to a selector.

EXAMPLE

```
omswhere everything
```

Lists those operators bound to the everything selector.

See also: omsbind, omsbindinfo, omsls, omsunbind, omwhere.

8.21 OMWRITE

SYNTAX

```
omwrite [bindings_file]
```

This command writes out all manipulator and selector bindings to the specified file. If no file name is specified, this argument defaults to $HOME/houdini5/OPbindings. The format of this file is such that it can replace the default bindings file found in $HH/OPbindings.

See also: ombind, ombindinfo, omls, omunbind, omwhere, omsbind, omsbindinfo, omsls, omsunbind, omswhere.

8.22 OPENPORT

SYNTAX

```
openport port_number
```

Opens a communication port to Houdini. This allows the *hcommand* program to run Textport commands from a remote process.

See Also: *closeport*; *Stand Alone > hCommands* p. 428.

8.23 PANE

SYNTAX

```
pane <options> [-d desk_name] pane_name
```

EXPLANATION

Allows different options of the pane to be set. A valid pane name must be specified. If no desk name is given then it will assume the current desk.

Also if no options are specified then all panes of that desktop will be listed.

OPTIONS

`-c 0|1`	Follow the parent's current node selection.
`-f 0|1`	Toggle fullscreen mode.
`-l 0-5`	Set the link value on the current pane. A zero turns linking off.
`-m type`	Valid types are: neteditor chaneditor geosheet listchans parmeditor textport uicustom viewer maniplist
`-h path`	Set the operator path.
`-T type`	Set the operator type. This is used to determine what type of network (SOP, CHOP, POP, etc.) the desktop referred to if the -h path doesn't exist.
`-o`	Open a new copy of the pane.
`-p`	Move the playbar to this pane.
`-s 0|1`	Split the pane in two: 0 – left/right, 1 – top/bottom.
`-t 0|1`	Tear off the window (1), or put it back (0).
`-z`	Close the pane.

EXAMPLES

`pane -f 1 pane2`	Turn fullscreen mode on for pane2.
`pane -h /obj pane2`	Set the path to objects for pane2.
`pane -l 0 -p pane2`	Turns linking off and moves the playbar to pane2.

8.24 PERFORMANCE

SYNTAX

```
performance [options]
```

EXPLANATION

Change the parameters for the Performance Monitor (noramlly to be found in the *Options > Performance Monitor.*

options

`-l off|window|stdout`	Set the Output Log mode.
`-c on|off`	Set Monitor OP Cook toggle.
`-o on|off`	Set Monitor Object Display toggle.
`-v on|off`	Set Monitor Viewport Display toggle.
`-f on|off`	Set Monitor Frame Length toggle.
`-i on|off`	Set Tile Statistics in OP Info toggle.
`-h on|off`	Set Tile Hilight when Cooking toggle.
`-s on|off`	Set Single Frame Capture toggle.
`-p on|off`	Set Pause toggle.
`-e on|off`	Set Enable Output toggle.

When no options are specified, the current settings are displayed.

8.25 PLAY

SYNTAX

```
play options
```

EXPLANATION

The *play* command controls the Playbar. If neither the -r or -s options are not specified, the playbar will start playing forwards.

options

`-1`	The -1 (number '1') causes playback to occur one time only, and will stop when it reaches the last frame.
`-l`	The -l (letter "l") causes play in loop mode, repeating over and over again until explicity stopped.
`-z`	Play in zigzag mode – alternately playing backwards and then forwards.
`-s`	Stops playback.
`-r`	Play in reverse.

Tip: To set the Playbar to a specific frame number, use the *fcur* ***command.***

8.26 VIEWBACKGROUND

SYNTAX

```
viewbackground [options] viewname
```

EXPLANATION

This command sets background/rotoscoping parameters for a Viewport. When no options are specified, the current settings of the specified Viewports and view-memories are displayed.

OPTIONS

`-b on|off`	Turn background image on or off.
`-t on|off`	Turn texture mapped backgrounds on or off.
`-a on|off`	Turn automatically placing background image on or off. To use fixed background image offset and scale, this option must be off and texture mapped backgrounds must be on.
`-o x y`	Offsets the image by x and y.
`-s x y`	Scales the image by x and y.
`-q quality`	Quality of the background image.
`-S file|cop`	The source of the background image.

For Files only:

`-F <filename>`	The filename of the disk file.
`-O on|off`	Toggles manual override of the file res.
`-r <xres> <yres>`	Manually sets the file res.

For COPs only:

`-c icename copname`	Specify the COP to use as a background image.
`-f frame`	The frame of the image to display
`-p <color> <alpha>`	Sets the Color and Alpha planes to use.

EXAMPLE

```
viewbackground -b off Build*
```

Entering this command in the Textport turns off the display of background images in all of the Build desk's Viewports.

8.27 VIEWCAMERA

SYNTAX

```
viewcamera [-c camera_name] viewname [viewname2...]
```

EXPLANATION

This command sets the camera to view through for the Viewport(s) specified. If the Viewport does not support viewing through a camera (i.e. Geometry Editor Viewports), or if the object specified is not a valid type (i.e. Geometry), an error is reported. It is not possible to specify a camera per memory.

Without any options, the cameras for the viewports specified are displayed.

See also: *viewls*, *viewdisplay*, *viewbackground*, *viewprojection*.

8.28 VIEWCOPY

SYNTAX

```
viewcopy fromview toview [toview...]
```

EXPLANATION

This command copies the settings of one viewport or view-memory to another. For example, you can set up memory buttons based on the current state of a viewport, copy the state stored in the memory buttons back to a viewport, or copy settings between different viewports.

If the destination is a view-memory that doesn't exist, it will be created.

You can specify more than one destination at the same time.

For details on how to specify viewports and memories, type: *help viewportformat*

EXAMPLES

```
viewcopy Build.pane1.world.persp1 Build.pane1.world.persp1:1
```

Copy the perspective view to memory location 1 of pane1 of the Build desk.

```
viewcopy Build.pane1.world.persp1:1 Build.pane1.world.persp1
  Build.pane1
```

Copy memory location 1 to the perspective view and also to a named memory location in the perspective view of the Build desktop's viewer in pane1.

8.29 VIEWDISPLAY

SYNTAX

```
viewdisplay [options] viewname
```

EXPLANATION

This command changes the Display options of a Viewport for either the view occupying a Viewport, or for one of the memory settings associated with it.

Tip: You need to supply a *viewname*. ***You can get this with the*** *viewls* ***command.***

Note: All viewing information (including memories) set using *viewdisplay* are lost when saving from non-graphical appliactions such as *hscript*.

OPTIONS

geo_type is one of:

`all`	Apply to All elements.
`sel`	Apply to Selected elements.
`unsel`	Apply to Un-selected elements.
`templ`	Apply to Templates
`target`	Apply to target output (i.e. the display SOP in the 'View Current' mode).

`-M` *geo_type mode*	Change the display mode, where *mode* is one of:

`wire`	Wireframe
`hidden_invis`	Hidden Line Invisible
`hidden_ghost`	Hidden Line Ghost
`flat`	Flat Shaded
`flat_wire`	Flat Wire Shaded
`shade`	Smooth Shaded
`shade_wire`	Smooth Wire Shaded
`vex`	VEX Shaded
`vex_wire`	VEX Wire Shaded

`-N geo_type m on|off`	Set display of points.
`-N geo_type n on|off`	Set display of point numbers.
`-N geo_type l on|off`	Set display of point normals.
`-N geo_type t on|off`	Set display of point texture coordinates.
`-N geo_type p on|off`	Set display of point positions.

`-E geo_type n on|off`	Set display of vertex numbers.
`-E geo_type t on|off`	Set display of vertex texture.
`-E geo_type g on|off`	Set display of vertex texture.

`-I n on|off`	Set display of primitive numbers.
`-I l on|off`	Set display of primitive normals.
`-I h on|off`	Set display of primitive hulls.
`-I t on|off`	Set display of primitive profiles.
`-I p on|off`	Set display of primitive profile numbers.
`-I b on|off`	Set display of primitive breakpoints.
`-B bw|wb`	Set the colour scheme: light or dark.
`-a on|off`	Set 'match selected with nonselected' on or off.
`-A templ|target on|off`	Turn faded look on or off for template or target output geometry.
`-b on|off`	Turn backface removal on or off.
`-C <value>`	Set constant sensitivity level.
`-D on|off`	Turn display geometry on or off

`-e value`	Set line width
`-f on|off`	Turn field guide on or off.
`-F on|off`	Turn filled selections on or off.
`-g on|off`	Turn guide geometry on or off.
`-h on|off`	Turn ’hulls only’ display on or off. If on, only the hulls will be drawn.
`-i on|off`	Turn footprint geometry on or off.
`-l <value>`	Adjust Level of Detail (for meta and quadrics). Default 1.0)
`-L on|off`	Turn multi-texturing on or off.
`-n <value>`	Option for scaling the display normal (default 0.2)
`-o on|off`	Turn object origin axes on or off.
`-O on|off`	Turn floating origin axes on or of (the one in the bottom-left corner of the Viewport).
`-q on|off`	Turn transparency on or off.
`-r on|off`	Turn projected textured and spotlights on or off.
`-R on|off`	Turn target output geometry on or off.
`-s on|off`	Turn safe area on or off.
`-S on| off`	Turn specular on or off in Shaded mode.
`-t on|off`	Turn texturing on or off in Shaded mode.
`-T on|off`	Turn template geometry on or off.
`-V <value>`	Set variable sensitivity level.
`-w on|off`	Turn wireframe move on or off.
`-x on|off`	Turn on grid in YZ plane (normal is X).
`-y on|off`	Turn on grid in XZ plane.
`-z on|off`	Turn on grid in XY plane.

EXAMPLES

```
viewdisplay -O off Build*
```

Turns off the Origin Axes display in all Viewports in the *Build* desktop.

```
viewdisplay -M all wire Build*
```

Sets all objects in the Build desk’s Viewports to wireframe mode.

```
viewdisplay -l 3 Build.pane1.world.persp1
```

Sets the display resolution in the *Build* desk's *persp1* Viewport for NURBS surfaces and metaballs to 3.

Tip: You could also set this in: i) Viewport Display Options (*Viewport* page), or ii) by setting the environment variable HOUDINI_LOD.

8.30 VIEWLAYOUT

SYNTAX

```
viewlayout -q -d [h | v] -s 1-4
```

EXPLANATION

The viewlayout command allows you to change how the view's viewports are arranged on screen. Currently, you can switch to a quad view with the -q option, switch to a double view with the -d option and switch to a single view with the -s option:

```
viewlayout -q modelmain
viewlayout -d h 1 3 modelmain
viewlayout -d v 2 4 modelmain
viewlayout -s 4 modelmain
```

the h and v after the -d option specify whether you want the two viewports arranged vertically or horizontally.

The numbers after the -d and -s options refer to the quadrants of the standard quad view:

1	Top Left
2	Top Right
3	Bottom Left
4	Bottom Right

8.31 VIEWLS

SYNTAX

```
viewls [-n] [-t type] [-l] [-v [-T viewport-type]] [pattern]
```

EXPLANATION

Lists all the available viewers using the following format:

viewername type

type will be either 'world', 'texture', or 'particle'
viewername will be of the form *Desk.pane.type*

OPTIONS

-n	Outputs a terse list with the names only.
-t	Restricts output to viewers of the specified type.
-l	Includes currently used view-memories, using the following format: *viewername:memname view-type*
-v	Lists the the Viewports that are associated with each viewer as well. It uses the format: *viewername.viewportname viewport-type*
-T	Restricts output to Viewports of the given type. The viewport-type can be one of: *perspective* *ortho_front* *ortho_right* *ortho_top* *uv*

If any patterns are specified, then output is also limited to viewers whose names match the given patterns.

For help on related commands type: *help view*

8.32 VIEWPROJECTION

SYNTAX

```
viewprojection [-o on|off] viewname
```

EXPLANATION

This command sets the projection type for a viewport.

options

`-o on|off`	Turn ortho viewing on or off

When no options are specified, the current settings are displayed.

EXAMPLE

```
viewprojection -o off objmain
```

This command disables orthographic projection in the Object Editor’s main Viewport.

8.33 VIEWTRANSFORM

SYNTAX

```
viewtransform [-p] [data] viewname
```

EXPLANATION

This command is not intended for use by humans. This sets the transform for a view (as well as some options). The command is really only used for saving to *.hip* files.

Options:

`-p`	Will display the *viewtransform* command.

A possible use for this (the -p option) is to save one transform and load it into another view. For example, you might want to save the current settings into a memory button, or move a memory button's settings into the current view.

All Viewport settings are saved to the *.hip* file using these commands.

8.34 VIEWTYPE

SYNTAX

```
viewtype [-t type] [name]
```

EXPLANATION

This command allows you to change the type of the given viewport(s) and or view memories. This type can be one of:

ortho_top / bottom / right / left / front / back
perspective
uv

For details on how to specify viewports and view-memories,
type: help viewportformat

EXAMPLE

```
viewtype -t ortho_top Build.pane1.world.persp1
```

will change the viewport called *persp1* in the first pane of the *Build* desktop to be an *ortho_top* Viewport.

8.35 VIEWUPDATE

SYNTAX

```
viewupdate [-c] [-u update_mode]
```

When changes are made in an interactive Houdini session the viewports will update in one of three possible ways depending on the state of the global update mode. This mode can be set to one of the following values:

`off | never`	Views update only on demand.
`on | changes`	Views update after changes.
`continuous | always`	Views update continuously.

The special update mode of “now” will force a single Viewport update if the current update mode is “never”.

With no options specified the viewupdate command will show the current status of the view update mode.

If the -c option is specified then this output will be in the form of a valid "viewupdate" command.

The -u option allows the current update mode to be modified.

This command correspond to the control offered by the global *Update* buttons at the top of each appropriate Editor.

EXAMPLE

```
viewupdate -u never
viewupdate -u now
viewupdate -c
```

Note: Currently the update mode applies only to 3D Viewports.

3 Uses of the Scripting Language

1 CREATING A SCRIPT FROM HOUDINI

1.1 OPSCRIPT / OPSAVE / OPWRITE COMPARED

There are three ways of writing out scripts from Houdini. It is important to know the differences and limitations of each method.

opscript	Saves text commands to build operators. This doesn't handle saving of locked data (i.e. a Model SOP). However, it does save the animation information etc.
opsave	Saves what ever the data is for the operator. This is only implemented for SOPs. This saves the data of the OP.
opwrite	Saves a CPIO archive snippet which is useful for copying operators (in their entirety) from one HIP file to another. This is the internal mechanism used for copy/pasting within Houdini. This method saves both the locked data and the animation information for the OP.

1.2 OPSCRIPT

SYNTAX

```
opscript [-r] [-g] [-b] [-v|-c] [-s] [-G] operator_pattern
```

EXPLANATION

This command will generate a series of commands which will re-create the objects specified. This command is very useful when the output needs to be modified, since the output is quite straightforward. All channel information as well as operator information is saved. It is also possible to create general scripts which take arguments for the objects to be created. In the general form, care is taken to check for the number of arguments passed to the script.

options

`-r`	This tells the command to work recursively through the entire operator hierarchy
`-G`	Operator groups in the objects will be saved with the objects.
`-g`	Specifies that the top-level arguments are in general form and that names must be used when sourcing the script file
`-b`	The brief option specifies that parameters at their default values are not displayed
`-v`	This tells the command to evaluate but not to display channel information
`-c`	Causes only the channels for the specified operator to be saved. The node creation, parms, flags and inputs will not be output when this option is specified. This is useful for saving animation data only.
`-s`	Output channel and keyframe times in samples (i.e. not in seconds).

Limitations: Some parts of operators have no script equivalent commands. For example, there is no way to load geometry into a SOP, so a modelled SOP will not be re-created correctly from the output of an *opscript* command.

The *opscript* command can be re-directed to a file, or displayed in the Textport.

To reconstruct the objects, the *source* command is used to read the script generated. See *source* p. 119 for details.

Opscript echos, or displays, the commands necessary to re-create the specified object. If you specify *-r*, the command covers the entire operator hierarchy. If you specify *-g*, the top-level arguments are general in form. The names of the objects must be specified using this option.

EXAMPLE

```
opscript -r /obj/geo*
opscript -r /comp/* > /tmp/mycompfiles.cmd
```

The first example above will provide you with the commands necessary to recreate all geometry objects in the operator hierarchy. The second example provides you with the commands necessary to recreate all operators in the hierarchy located in the directory specified.

See also: *opwrite*, *opsave*

1.3 MWRITE

EXPLANATION

This command saves the whole motion file into one CPIO archive. There are two shell scripts provided in the bin directory to take the CPIO archive apart and to reconstruct it (*hexpand* and *hcollapse*). Once the *.hip* file is expanded, each component can be edited by hand. Care must be taken to reconstruct the CPIO archive in the same order that it was originally. The *hexpand* and *hcollapse* commands take care of this automatically.

Limitations: Since this creates a CPIO archive, the contents are not easily editable. Also, it is not easy to generalize the file so that it performs like a macro.

The *mread* command can be used to load or merge a *.hip* file into an existing *.hip* file. If you specify the `-i` option, the filename is incremented, that is has a number appended to it to signify its position in a sequence of saved versions.

1.4 OPWRITE

SYNTAX

`opwrite object` *filename*

Saves the object's contents to the file you specify.

EXPLANATION

Like the *mwrite* command, this command saves into a CPIO archive. However, the *opwrite* command will save a partial *.hip* file. For example, it is possible to save only SOPs from a *.hip* file, or only objects, or only materials. This can be used to extract some information from one *.hip* file and move it into another *.hip* file.

Limitations: Unlike the *opscript* command, *opwrite* saves modelled data and *all* information required to reconstruct an operator (or operators). Unlike the *mwrite* command, this allows for partial save/load of motion data. However, like *mwrite* the output is in CPIO archive format, and is less flexible than the *opscript* command. Internally, this command is used for copy/paste operations.

The *opread* command is used to load the files created by the *opwrite* command.

EXAMPLE

```
opwrite light2 myfile.hip
```

This causes the contents of object *light2* to be saved to a file named *myfile.hip*.

1.5 FURTHER EXAMPLES

If geo1 is made with a Circle SOP, a smaller Circle SOP, followed by a Sweep SOP, followed by a Skin SOP (which is set for display and render), you have created a partial torus. Frequently, you want to use the Sweep SOP followed by the Skin SOP. So, in the Houdini Textport, you could type the following script:

```
houdini-> opcf /obj/geo1
houdini-> opscript * > skinner.cmd
# Automatic generated script
set saved_path = `execute(“oppwf”)`
opadd circle circle1
oplocate -x 2.1 -y 3.26667 circle1
opparm circle1 type ( nurbs ) orient ( xy ) rad ( 1 1 )
  t ( 0 0 0 ) order ( 4 ) divs ( 10 )
  arc ( openarc ) angle ( -90 90 ) impefect( on )
opset -d off -r off -t off -l off -s off -u off
  -c off -C off -p off circle1
opadd circle circle2
oplocate -x 0.644444 -y 3.24444 circle2
opparm circle2 type ( nurbs ) orient ( xy ) rad ( 0.2 0.2)
  t ( 0 0 0 ) order ( 4 ) divs ( 10 )
  arc ( closed ) angle ( 0 360 ) imperfect ( on )
opset -d off -r off -t off -l off -s off -u off
  -c off -C off -p off circle2
opadd sweep sweep1
oplocate -x 1.6 -y 2.05556 sweep1
opparm sweep1 xgrp ( “” ) pathgrp ( “” ) cycle ( all )
  angle ( off ) noflip ( off ) skipcoin ( on )
  usevtx ( off ) vertex ( 0 ) scale ( 1 )
  twist ( 0 ) roll ( 0 ) newg ( off )
  sweepgrp ( sweepGroup )
opset -d off -r off -t off -l off -s off -u off
  -c off -C off -p off sweep1
opwire circle2 -0 sweep1
opwire circle1 -1 sweep1
opadd skin skin1
oplocate -x 1.61111 -y 1.17778 skin1
opparm skin1 uprims ( “” ) vprims ( “” ) surftype ( quads )
  keepshape ( off ) closev ( nonewv ) force ( off )
  orderv ( 4 ) skinops ( all ) inc ( 2 )
  prim ( off ) polys ( off )
opset -d on -r on -t off -l off -s off -u off
  -c off -C on -p on skin1
opwire sweep1 -0 skin1
opcf $saved_path
```

This may look like a lot of noise. However, there are really only twenty commands being executed. Starting with this as a script setup, you would remove the commands relating to the Circle SOPs, and re-work the script slightly as follows:

```
#  Script to append a skin and a sweep to the current object.
#  The first argument = backbone; second = the # cross-section.
if ( “$argc” != 3 ) then
  echo Usage:  $arg0 backbone_sop cross_section_sop
exit
endif

# Here is a neat trick to find out the name of the SOP
# which was just added
set sweep_sop = `execute(“opadd -v sweep”)`
oplocate -x 1.6 -y 2.05556 $sweep_sop
opparm $sweep_sop xgrp ( “” ) pathgrp ( “” ) cycle ( all )
  angle ( off ) noflip ( off ) skipcoin ( on )
  usevtx ( off ) vertex ( 0 ) scale ( 1 )
  twist ( 0 ) roll ( 0 ) newg ( off )
  sweepgrp ( sweepGroup )
opset -d off -r off -t off -l off -s off -u off
  -c off -C off -p off $sweep_sop
# Here, we change our connections
opwire $arg2 -0 $sweep_sop
opwire $arg1 -1 $sweep_sop

set skin_sop = `execute(“opadd -v skin”)`
oplocate -x 1.61111 -y 1.17778 $skin_sop
opparm $skin_sop uprims ( “” ) vprims ( “” ) surftype ( quads )
  keepshape ( off ) closev ( nonewv ) force ( off )
  orderv ( 4 ) skinops ( all ) inc ( 2 )
  prim ( off ) polys ( off )
opset -d on -r on -t off -l off -s off -u off
  -c off -C on -p on $skin_sop
opwire sweep1 -0 $skin_sop
opcf $saved_path
```

2 ENVIRONMENTS

2.1 THE DEFAULT ENVIRONMENT FILE (123.CMD)

When Houdini is run without any arguments, a default file called *123.cmd* is loaded. This is the first file found in the Houdini search path, so it can be overridden creating a file *$HOME/houdini/scripts/123.cmd*. If you want the default start-up environment to contain nothing, leave an empty file there.

2.2 EDITING THE STARTUP SCRIPT

When you start Houdini, a default script is automatically read in called *123.cmd*. This script sets up the scene with a world object, two cameras, three lights and two geometry objects. A portion of this *123.cmd* looks like this:

```
# This is the startup script.
oadd -t world world
oadd -t camera cam1 cam2
oadd -t geometry geo1 geo2
oadd -t light light1 light2 light3
edit
odisplay -d on -s geo1,geo2,light*
odisplay -s geo2 -c 23
sgrid -s geo2:first -R 11 -C 11 -z 20 20 -P 0 -.8 0 -p xz -t quad -
  c rc
sopdisplay -s geo2:first -d -r
chadd -s light*/t?,r?,light?
...
```

This file can be modified to customize your default start-up environment. For example, you may want *Double Buffering* to default to *off*; so you could add the line:

```
doublebuffer off
```

Any line which starts with a # will not be executed: use them to add comments to your cripts.

2.3 SAVING AN ENVIRONMENT

1. Setup your environment (lights, cameras, default geometries, etc.).

2. Open a Textport with the *Dialogs* > *Textport* menu.

3. Save your environment by typing in:

```
opscript -r /obj/* > myEnvironment.cmd
```

This saves your environment into the file *myEnvironment.cmd*. Although it isn't necessary to know all the details at this point, it will help to know why this works later on. The *-r* option recursively writes all objects to the file. The */obj/** means that you will be writing out all objects (hence the *) the UNIX redirection character (>) sends the result to the file *myEnvironment.cmd*.

2.4 LOADING AN ENVIRONMENT

1. Open a Textport with the *Dialogs* > *Textport* menu.
2. Type:
   ```
   source myEnvironment.cmd
   ```

This merges the elements from your *myEnvironment.cmd* file with those which already exist. If you want to start from scratch, you must enter the *Objects* and delete all the objects first.

2.5 SAMPLE CUSTOM 123.CMD FOR UNIX SCRIPTERS

Here is a *123.cmd* file that provides some UNIX type aliases for common commands like *ls* and *cd*. If you put a *123.cmd* file in *$HOME/houdini/scripts*, it will be picked up before *$HH/scripts/123.cmd* .

If you're in a large shop where the system administrator has already set up a *123.cmd* file in */usr/local/houdini/scripts* (or somewhere else), you should probably source this in instead of *$HH/scripts/123.cmd* .

```
# Here, we source in the installed version of 123.cmd so
# that we'll pick up any changes in the distribution.
#
source $HH/scripts/123.cmd

#  Now, make some sane aliases
#
alias cd  opcf
alias ls  opls
alias pwd oppwf
alias add opadd
alias rm  oprm
alias date 'echo Frame $F: `system(date)`'
alias unsetenv setenv -u
alias unset set -u
```

You can always change the setup of Houdini so that it's got whatever objects/materials you want by default. Simply use the *opscript* command as mentioned above, to see what the commands are needed to generate an operator.

Once the *123.cmd* is read into a HIP file, the aliases are saved along with the file, so unless someone removes them, these aliases will be retained.

2.6 EDITING HIP SCRIPTS

Because Houdini *.hip* files are UNIX CPIO archives, you can use *cpio* to unpack, and edit them – the commands are the same commands listed here in the scripting commands. Once you are done, you can reassemble them with cpio again. To facilitate this, there are two scripts: *hexpand* and *hcollapse* .

HEXPAND

The script *hexpand* will expand a cpio file such as a *.hip* file into a directory structure that can then be edited using a text editor, and then collapsed back into a regular file using *hcollapse* .

Usage:

```
hexpand <hip_file>
```

HCOLLAPSE

Usage:

```
hcollapse [-r] <cpio_file>
```

`-r`	removes the contents file and the expanded directory

The script *hcollapse* will collapse a directory that was previously extracted from a cpio file using *hexpand* .

EXAMPLE

Select your *.hip* file in *spy*, and type:

```
!hexpand myHIPfile.hip
```

to expand the Houdini file into editable directories and files.

3 CREATION SCRIPTS

Every OP node in Houdini can have a “creation script” run when it is created. These scripts must appear somewhere in the Houdini search path (below), and should appear in a directory according to the type of the node being created.

$HOME/houdini/scripts

3.1 NAMES OF CREATION SCRIPT DIRECTORIES

The names for the creation script directories are:

scripts/obj	Objects
scripts/mat	Materials
scripts/comp	COPs (Composite) Networks
scripts/out	Output OPs
scripts/out	POP (Particle) Networks
scripts/ch	CHOP (Channel) Operators
scripts/shop	SHOP (Shader) Operators

3.2 EXAMPLES

ADDING A TOP TO A SHADER UPON CREATION

For example, if, every time you added a new Phong shader, you want to automatically create a Color TOP inside it, you can do it by creating a script:

```
% cd ~/houdini
% mkdir scripts
% cd scripts
% mkdir mat
% echo 'opcf $arg1 ; opadd texture' > mat/phong.cmd
```

The file *phong.cmd* should contain:

```
set save_dir = ’run("oppwf")’
opcf $arg1
opadd texture
opcf $save_dir
```

Important information about the script:

1. The script name must be the operator name with a *.cmd* file extension (to find out what the operator names are, you can do an *opadd* command with no arguments).

2. The script must exist in the correct directory

3. When the script is run, the name of the created object as the first argument.

4. The current working directory will be set to the owner of the node just created (i.e. when the script is run for a SOP, the current working directory will be the object containing the SOP). The current working directory is restored on completion of the creation script.

CREATION OF AN OBJECT

As an example, in terms of the Houdini scripting language, the creation of an object goes something like this:

```
# First, add a geometry object and get the name of the object
# into a variable
hscript-> set save_cwd = `execute("oppwf")`
hscript-> opcf /obj
hscript-> set added_name = `execute("opadd -v geo")`
hscript-> cmdread -q obj/geo $added_name
hscript-> opcf $save_cwd
```

This is just an example, in reality, none of the variables in the above example are actually used or set. The example is simply to illustrate the steps taken every time an object is added.

3.3 EXAMPLE SCRIPTS

You can find examples of creation scripts in: *$HH/scripts/obj/* .

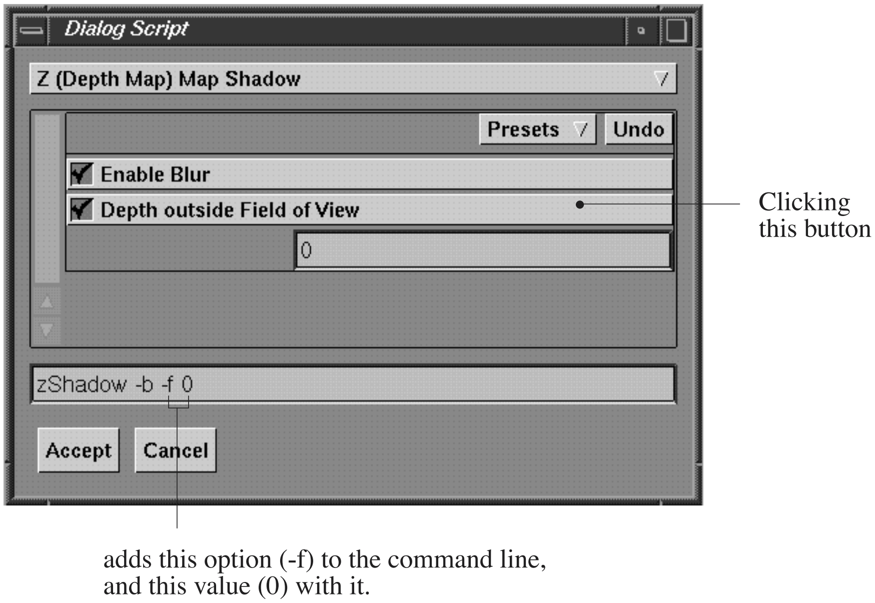
4 DIALOG SCRIPTS

4.1 INTRODUCTION

Dialog scripts provide a simple way of creating a graphical interface for building a command string (this is usually a set of parameters and options for a scripting command or shader). They do this by defining a series of buttons, menus and edit fields to generate the parameters for a command line option.

When available, it is possible to access a Dialog script by clicking on the ✜ button beside a parameter in Houdini. Some places they are used are: the *Shaders* ✜ button in the *Light* object, and the *Operator Scripts...* option in the *OP Layout* > Ctrl 🖱 pop-up menu.

EXAMPLE



☑ Enable Blur

☑ Depth outside Field of View

0

zShadow -b -f 0

Clicking this button

adds this option (-f) to the command line, and this value (0) with it.

In the example above, the dialog script in *$HH/config/scripts/MANlight.ds* is used to generate the command line options for a Z-depth Shadow (zShadow) with Blur enabled (-b) and Depth outside Field of View (-f 0) at a setting of zero.

4.2 SUPPLIED DIALOG SCRIPTS

Dialog scripts are supplied in *$HH/config/Scripts*. Each editor (i.e. SOP, COP, Object) has its own set of dialog scripts, so you can add your function to any of the editors. If you want to edit these, make a copy in your *$HOME/houdini/config/Scripts/* directory.

When Houdini reads a dialog script, it will automatically create a UI file and provide a graphical interface for editing a string. Houdini has “hooks” for a dialog script interface in several parts of the application. These are:

• OBJmacro.ds	Object Macros
• SOPmacro.ds	Macros for SOPs
• COPmacro.ds	Macros for COPs
• MATmacro.ds	Material Macros
• MANfog.ds	Shader strings for Mantra Fog shaders
• MANlight.ds	Shader strings for Mantra Light shaders
• MANlightfog.ds	Shader strings for Mantra Atmospheric Light shaders

• RMshader.ds	RenderMan Surface Shader Editor
• RMdisplace.ds	RenderMan Displacement Shader Editor
• RMlight.ds	RenderMan Light Shader Editor
• RMinterior.ds	RenderMan Interior Shader Editor
• RMatmosphere.ds	RenderMan Atmosphere Shader Editor

• Renderers.ds	An interface to common Renderer Commands. This shows up in the Output Drivers when there is a command for a specific renderer.

When adding a macro/shader to a dialog script, do not replace the scripts installed in *$HH/config/Scripts* . Because all scripts found in the Houdini search path are merged, it is preferable to install these scripts in any of the path directories (i.e. */usr/local/config/Scripts*).

4.3 SYNTAX OF A DIALOG SCRIPT

The syntax of a dialog script is quite simple. The script describes a nested set of parameters. Each parameter may be one of many types (i.e. a toggle button, a float or a string). Internally, each parameter is represented as either a string, float, or an integer vectored value. However, there are different ways of representing this value in the user interface. For example, a string might be a string that the user enters, or a file name, or a string chosen from a menu. The list of possible parameter types may be updated to include new types not listed here.

There are also special controls in a script for special features. For example, if the script is supposed to parse a RenderMan shader, there is a special keyword to use the RenderMan syntax. Other special features are the ability to group parameters onto individual pages.

GENERAL FORM

The general form of a dialog script is:

```
command {
  header
  parameter_list
  }
```

HEADER DEFINITION

The *header* definition is basically used to define what the command is at a global level. There are certain fields which can be specified at this point:

name	Specifies the actual command name or shader name.
help	Specifies help for the command or shader.
label	Specifies a more descriptive name which is used by the user interface.
default	Specifies a default command/string which is used at startup.
rman	Specifies that this command should be interpreted using RenderMan syntax.

Only the *rman* keyword doesn't take any options after it. All of the other keywords expect a string to follow. The *help* keyword expects a series of strings enclosed by a set of braces ({ and }). See below for an example.

Note***:*** When specifying strings, the script requires that the string be enclosed in double quotes (") if and only if there are spaces in the string. However, it doesn't hurt to put the double quotes for all strings.

Note: The name usually specifies the name of the command. Theoretically, it should not contain any blanks. However, this is not a limitation though presets may not work correctly.

PARAMETER TYPES

The form of a parameter definition is as follows:

```
parm {
  field  value
  }
```

Mandatory *fields* for each parameter are:

name	The internal name for the field. This is used when saving and loading presets for the dialog script.
label	The "visible name". This is what appears in the graphical interface. It should be more descriptive than the internal name.
type	This determines the type of the parameter. See the list below for available types.
required | *option*	Needed to specify whether this parameter is a required field or an optional field.

types

The *type* may be defined as one of the following:

• *toggle*	A toggle button.
• *string*	An editable string.
• *object*	An editable string. A menu of all objects can be used to stuff the string.
• *render*	An editable string. A menu of all output drivers can be used to stuff the string.
• *file*	A string which has a file browser button beside it.
• *float*	A single or vector of floating point values.
• *integer*	A single or vector of integer values.
• *direction*	Three floating point values. The size field must be set to 3 for this type of value.
• *color*	Three floating point values. Like direction, the size field must be set to 3. The user interface for this parameter type includes a set of color sliders.

OPTIONAL PARAMETER KEYWORDS

menu

menu Specifies a list of strings that the parameter may contain. The syntax for menus is:

```
menu {
  value  label
  value  label
  value  label
  }
```

where the *label* will show up in the UI, while the *value* is what is actually used for the field.

size Specifies the vector size of the parameter. By default, this is 1.

default Specifies the default value(s) for the parameter. The syntax is:

```
default { value1 [value2...] }
```

Where each *value* specified is for a different component of a vector. For example, if the *Size* of the parameter was 3, you would probably want to specify three default values. If no default is specified in the parameter declaration, the parameter will default to 0 (or an empty string).

callback

The keyword *callback* specifies an *hscript* command file to execute when the parameter's value changes. Please see also: *Ref > Shaders > VEX Compiler Pragmas > callback* .

4.4 EXAMPLES OF PARAMETER DEFINITIONS

EXAMPLE 1

```
parm {
  name    rolloff
  label   "Spot Light Rolloff"
  type    float
  option  -r
  size    1
  default { 1 }
  }
```

This example shows an optional parameter which is a single float value. It defaults to 1.

EXAMPLE 2

```
parm {
  name    scolor
  label   "Specular Color"
  type    color
  required
  size    3
  default { 1 1 1 }
  }
```

This example shows a required parameter which is a color. When the UI is built for this parameter, it will contain a set of RGB sliders for editing the colour value.

EXAMPLE 3

```
parm {
  name    nproc
  label   "Number of Processes"
  type    string
  option  -n
  menu    {
      1   "1 Process"
      2   "2 Processes"
      4   "4 Processes"
      }
  }
```

This example shows how you can use a menu to cheat and specify an integer as a string. The menu will simply use the value (i.e. 1, 2, or 4) but present the user with a list of choices.

4.5 SPECIFYING GROUPS (“PAGES”)

The dialog script language allows you to group parameters together in pages (with a page tab for each page). The syntax of the group specification is:

```
group {                 # First group of parameters/folder tab
  name   string         # Name in the tab
  label  string
  parameter_list
  }
group {                 # Second group of parameters/folder tab
  name   string         # Name in the tab
  label  string
  parameter_list
  }
...
```

It is possible to fit as many levels of pages as you want in the dialog script. However, there may be formatting problems when the UI is displayed if you make too many of them due to the horizontal size constrainsts of the screen. It is also possible to nest groups, though in general, this is bad UI design since it hides parameters from the user.

4.6 MIXING PARAMETERS

It is also possible to mix parameters so that some are always displayed, then the page tab is a separate entity. For example:

```
command {
  name        pagetab
  label       "Paged Tab Example"
  help        {
  "A simple dialog script to show that the always parameter"
  "is always visible even when the groups get switched."
  "The groups appear below the 'always' parameter."
  }

  parm {
      name     always
      label    "Always visible"
  }

  group {
      name     page1
      label    page1

      parm {
          name     opt1
          label    "Option 1"
      }
  }
  group {
      name    page2
      label   page2
```

```
        parm {
            name     opt2
            label    "Option 2"
        }
    }

    parm {
        name    always2
        label   "Shows up after the folder"
    }
}
```

4.7 EXAMPLES OF DIALOG SCRIPTS

Many examples can be found by looking in: *$HH/config/Scripts* .

4.8 TESTING DIALOG SCRIPTS

There is an application shipped with the HDK (Houdini Development Toolkit) called *dsparse*. This will parse a dialog script and allow you to experiment with the layout and adding parameters. It is necessary to purchase the HDK in order to get this utility.

4.9 COMMON PROBLEMS

When each dialog script first gets parsed, it creates a .ui file.
This file is created in:

/usr/tmp/Dialogs/...

This specifies the UI layout for the dialog script. When this file is created, the UI is locked for the script. Therefore, even if parameters change, the UI displayed will be the same. Therefore, when testing, it is mandatory that this created UI file is removed between each test.

5 RMANDS

SYNTAX

```
rmands [options] file1.via [file2.vma ...] [file1.slo...]
```

EXPLANATION

This shell command (not Houdini Textport) accepts for input .via and .vma files, and generates the Dialog Script for a SHOP *.slo* file.

Tip: Once you've created or edited a dialog script, you will typically want to click the *Reload* button (at the top of the Dialog Script Window) to ensure that they're properly updated. Please see: *Interface > Dialog Scripts* p. 219.

Warning: This program will remove existing .ds files!

OPTIONS

`-d`	Destination directory.
`-D`	Destination directory (puts script in this directory rather than a sub-directory as with the -d option).
`-i`	Run using *soinfo* (for RenderDotC).
`-e`	Run using *sletell* (for Entropy).
`-p`	Run using *sloinfo* (for prman).
`-old`	Generate old style dialog scripts (pre-shop).
`-g`	Specify maximum size per parameter page. The shader parameters will be split into multiple pages if there are too many parameters in the shader.
`-l file`	Create an operator type definition file for the operator.
`-L file`	Specifies an operator type library file that the operator definition should be added to. If not specified, the type definition file is also used as the library file.
`-b`	Create a backup of the operator type library file.
`-C icon`	With the -l option, the icon for the operator.
`-n name`	With the -l option, the name of the operator.
`-N label`	With the -l option, the label of the operator.

EXAMPLE

```
rmands -g 8 -d $HOME/houdini/shop *.slo
```

SHADER HELP

When *rmands* is generating the script for a .slo file, it will look for a file *$shader.hlp* (where *$shader* is the name of the shader) and include this file as help for the shader. *rmands* will look in the standard Houdini path (under the *ri_shaders* sub-directory) for the help files.

For example:

```
% vi ~/houdini/ri_shaders/myshader.sl
% vi ~/houdini/ri_shaders/myshader.hlp
```

SEE ALSO

- *Stand Alone Tools > dsparse – Dialog Script Parse* p. 431.

6 TCL / TK SCRIPTING

6.1 INTRODUCTION

Houdini comes with an embedded Tcl (Tool Command Language) / Tk (Toolkit) interpreter, which reads Tcl commands in much the same way csh commands are interpreted in UNIX. Tcl is pronounced “tickle”, and was developed by J.K. Ousterhout (University of California at Berkeley). Tcl and its extension, Tk, provide a means of creating your own interface tools like buttons, scrollbars, and menus. The language can be used in conjunction with *hscript*, the non-graphical version of Houdini, to generate simple, custom tools that integrate with the standard user interface.

To find out more about the language’s rules and syntax, a good place to start is at: *http://www.tcl.tk/* .

6.2 USING TCL AND HSCRIPT

To access at Tcl shell from within *hscript*, enter the *tcl* command. This starts a Tcl interpreter. If you plan to use the Tk extension, type the *tk* command. We will be using the Tcl/Tk shell in this example. You can start the Tcl/Tk interpreter from Houdini’s Textport, but it is easier to write, test and debug Tcl/Tk scripts through *hscript*.

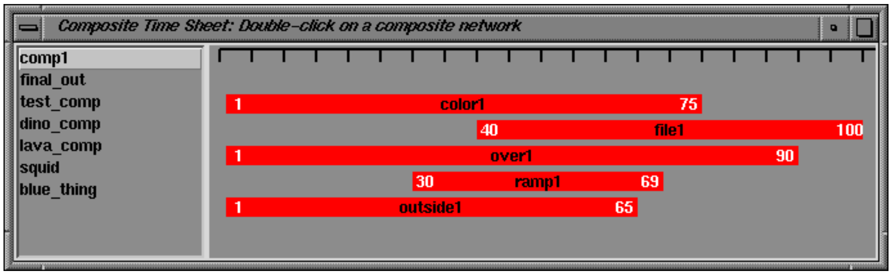
The Tcl/Tk interpreter takes over *hscript* until you type the *exit* command (see the section on Limitations below). Houdini adds a command to Tcl/Tk:

```
hscript command ?hscript_command_args?
```

hscript runs its first argument as an *hscript* command. It takes the remaining arguments (which are optional, as indicated by the question marks) and passes them on as arguments to the given *hscript* command.

For example, from the Tcl/Tk shell: *hscript opls /obj* lists all of the object operators in your current Houdini session.

6.3 EXAMPLE



The code that follows builds a graphical representation of the frame-ranges of COPs in a COP network.

CODE

```
## A little script that will show your COP frame ranges as bars

proc get_start_range { copname } {
    hscript set tmp=$copname
    set start [ hscript echo {`ch("$tmp/start")`} ]
    return $start
}

proc get_end_range { copname } {
    hscript set tmp=$copname
    set end [ hscript echo {`ch("$tmp/start")+ch("$tmp/length")-1`} ]
    return $end
}

proc draw_ruler {} {
    global left_margin pixels_per_frame

    set i 1
    set x1 $left_margin
    set x2 $x1
    .canvas create line $x2 5 $x2 15 -width 2
    while { $i <= 30 } {
  set x1 $x2
  set x2 [ expr $x1 + ( $pixels_per_frame * 5 ) ]

  .canvas create line $x1 5 $x2 5  -width 2
  .canvas create line $x2 5 $x2 15 -width 2

  incr i
    }

}

proc draw_tbar { cop_num cop_name start end } {
    global pixels_per_frame left_margin

    set bar_height 15
    set topy   [expr $cop_num * ($bar_height + 5)]
    set boty   [expr $cop_num * ($bar_height + 5) + $bar_height]
    set startx [expr $left_margin + ($start * $pixels_per_frame)]
    set endx   [expr $left_margin + ($end   * $pixels_per_frame)]

    .canvas create poly $startx $topy $endx $topy $endx $boty $startx $boty \
  -fill red
    .canvas create text [expr ($startx+$endx) *0.5] [expr ($topy+$boty) *0.5] \
  -text $cop_name
    .canvas create text [expr $startx+10] [expr ($topy+$boty)*0.5] \
  -text $start -fill white
    .canvas create text [expr $endx  -10] [expr ($topy+$boty)*0.5] \
  -text $end   -fill white
}

proc build_tbars { cop_network } {

    .canvas delete all
    draw_ruler

    hscript opcd /img/$cop_network

    set i 2
    foreach cop [ hscript opls ] {
  set start [ get_start_range $cop ]
  set end   [ get_end_range   $cop ]
  draw_tbar $i $cop [lindex $start 0] [lindex $end 0]
  incr i
    }
}

proc build_COPlist {} {
    hscript opcd /img
    foreach network [ hscript opls ] {
  if { $network == "." } { continue }
  .copList insert end $network
    }
}

## UI Widgets

listbox  .copList -bg grey70
canvas   .canvas -width 750 -height 500 -bg grey70
wm title . "Composite Time Sheet: Double-click on a composite network"

pack .copList .canvas  -side left -fill both
```

```
## mainline

set pixels_per_frame 5
set left_margin      10
build_COPlist
bind .copList <Double-1> {
    set copnet [.copList get [.copList curselection]]
    build_tbars $copnet
}
```

6.4 EXPLANATION

It is important to remember to use proper quoting in Tcl. The characters “ and $ have special meaning in Tcl, and they are expanded by the interpreter before they are processed by *hscript*. For example, the following will *not* work from Tcl/Tk:

```
 set cop file1
 hscript echo `ch(“$cop/framerange1”)`
```

Tcl will see the “ characters and expand *“$cop/framerange1”* to *file1/framerange1*. This is then passed to *hscript*, which executes *echo `ch(file1/framerange1)`*, which is incorrect because *hscript* expects *echo `ch(“file1/framerange1”)`* (a string needs the double quotes). What we really want to do is pass the double quotes through to *hscript* from Tcl/Tk.

We can accomplish this using the braces (i.e. {})in Tcl/Tk. Tcl does not perform any expansions within braces. So, using the braces, our command becomes:

```
hscript echo {`ch(“$cop/framerange1”)`}
```

But this is still incorrect, since Tcl/Tk will not expand *$cop* to *file1*. We want Tcl/Tk to not expand the quotes, yet we still want it to expand *$cop*. This is impossible, so we must work around it by setting a variable at the *hscript* level, then reference that variable in our *hscript* command. The correct command is

```
hscript set tmp=$cop
hscript echo {`ch(“$tmp/framerange1”)`}
```

The first line creates a variable called *tmp* at within *hscript* and sets its value to *file1* (since *$cop* is expanded by Tcl/Tk in this case since we have not used braces). The second *hscript* command then references this *$tmp hscript* variable.

You’ll notice that there are two separate name spaces involved here: the Tcl/Tk name space and the *hscript* name space. In the above example, the *$tmp* variable does not exist in the Tcl/Tk shell. That is, the Tcl/Tk command `puts $tmp` would cause an error since *$tmp* is only defined at the *hscript* level. Similarly, the variable *$cop* does not exist in the *hscript* shell, it is a Tcl/Tk variable. So, `hscript echo {$cop}` would also cause an error: since *$cop* is not an *hscript* variable.

LIMITATIONS

If you start a Tcl/Tk shell from Houdini, you will notice that the Tcl/Tk interpreter takes over the application until you have exited your script. That is, Houdini does not refresh nor recook while a Tcl/Tk interpreter is active. That does not mean you cannot modify Houdini objects interactively with a Tcl/Tk script. For example, we could enhance the above script to modify the frame ranges of the COPs by clicking and dragging the time bars. We would modify the COPs using the command *hscript opparm*.... When we exited the Tcl/Tk script, the COPs would show their new frame values.

A second limitation involves Tcl/Tk itself. In Tcl/Tk, all variables are treated as strings or lists of strings. This is fine for simple applications, but more complicated programming requires the use of more elaborate data types (integers, floating point numbers) and the ability to group these data types together (such as in C's concept of a structure or Python's concept of a mutable list). Tcl/Tk is great for writing small, simple, tools very quickly. More sophisticated programming tasks will naturally require a more sophisticated language.

4 Environment Variables

1 INTRODUCTION

There are many environment variables that affect the way Houdini operates. They set preferences for such things as your display, directory structures for loading and saving files, and paths for the licence server's location.

1.1 SETTING AN ENVIRONMENT VARIABLE

IN WINDOWS

Click with 🖱 on '*My Computer*', and select *Properties* from the menu that appears. In the dialog, switch to the *Environment* tab, and set your variable there.

FROM A SHELL

You can set these from any Window Shell. You can also set them from within the Houdini Textport with the *setenv* command. For example:

```
setenv UT_INTERRUPT_THRESH 50
```

sets the delay for the Interrupt Cook dialog so it waits 5 seconds before appearing.

ALIASES / VARIABLES DIALOG

You can also set and display *some* Houdini-specific variables in the *Dialogs > Edit Aliases/Variables*... dialog.

1.2 OBTAINING A LIST OF VARIABLES

HCONFIG

To get a listing of all Global Variables, in a Window Shell (*Options* menu), type:

```
hconfig -h
```

this will give you a listing of all current environment variables. The following pages contain a listing of some of those which are most commonly used.

2 ENVIRONMENT VARIABLES

2.1 DIRECTORY-RELATED VARIABLES

HFS	Root of the Houdini install tree. All files related to running Houdini reside here – binaries, DSOs, etc.
HOUDINI_PATH	The Houdini search path. The Houdini path is user definable, and is where Houdini will look to load support files. Extending it allows you to include a directory to be searched for site-wide configuration files.
HIPDIR	The directory in which the *.hip* file is located. It always returns the directory from where the Houdini file was opened. This variable changes automatically if you move the *.hip* file to another directory. This allows you to move the *.hip* file together with its subdirectories, so long as you use $HIPDIR throughout to point to your project files. If $HIPDIR is used within filename paths, then you can be anywhere on a network and access the project without having to change the $HIP variable. It is a good idea to organise you project so that it uses directories like: $HIPDIR/geo $HIPDIR/mat etc.
HIP	Stores the location of the hip file when it was first created – the job's home directory. This variable is static and won't change unless explicitly changed by the user, or if you've set the variable HOUDINI_HIP_FROM_PATH . To retrieve the current location of the hip file, use $HIPDIR. ***Note!*** Use the *hip* scripting command to set this. For example: `hip /usr/people/jobs` sets $HIP to the new path specified.
HOUDINI_HIP_FROM_PATH	Causes the $HIP variable behaviour to change such that the HIP variable is set to the .hip file directory name (so it acts like $HIPDIR).
HOUDINI_DSO_PATH	Search path for Houdini plug-in modules.
HOUDINI_MPLAY_LOCKPATH	Determines where mplay .lockfiles are put. This allows users with home directories on NFS to avoid the network penalties by setting the lock file to a local path (i.e. */tmp* or *c:/temp*).

HOUDINI_BACKUP_DIR	This is the directory used to store the backed-up hip files when the Numbered Backup preference is set. The default backup directory is: *./backup* and the backed up files will have a *_bak{num}* suffix.
HOUDINI_BUFFEREDSAVE	When enabled, .hip files are first saved to a memory buffer and then written to disk. This is useful when saving over the network from Windows 2000 machines, or other places where seeking to the network is expensive.
HOUDINI_TEMP_DIR	The directory to which Houdini writes */tmp* files to.
HOUDINI_UNDO_DIR	Allows you to override the location for files Houdini uses for undo operations. The default is */tmp*. The files will have names like *add### model###* and *del###* .

2.2 VIEWPORT RELATED VARIABLES

HOUDINI_DOUBLEBUFFER	Double buffering improves the quality of moving images on a 24 bit monitor. If *Double Buffering* is not used, everything is drawn directly to the screen without any synchronization to the monitor's refresh rate. This causes areas of the screen which are being frequently redrawn tend to be displayed while only partially redrawn, this produces a "flickering" effect. When *Double Buffering* is on, the new image is drawn in an offscreen buffer in video RAM and then swapped into the display almost instantaneously, thereby eliminating the display of half-drawn images, and the flicker. Why not have it on all the time? Because some machines do not have sufficient video RAM to perform double buffering in 24bit colour – so they halve the colour resolution, and then still-frame image quality is reduced due to dithering.
HOUDINI_LOD	Sets the Level of Detail in displays for NURBS surfaces and metaballs. The default is one. ***Tip:*** You can also set this in the Viewport Display Options > *Viewport* page, or by using the *viewdisplay* scripting command.
HOUDINI_MDISPLAY_WAIT_TIME	Specifies the time, in seconds, that *mantra* should spend looking for an *mdisplay* window before giving up, and displaying the *Abort* / *Retry* / *Fail* alert.

HOUDINI_VIEW_MANTRA	Specifies the command to use for default *View: Mantra* renderer in the Viewport.
HOUDINI_VIEW_RMAN	Specifies the command to use for *View: R-Man* in the default renderer in the Viewport.

2.3 RENDERMAN-RELATED VARIABLES

RMAN_EYESPLITS	Value for RenderMan eyesplits (setting any of these three variables sets the option in the RIB file).
RMAN_SHADERPATH	Search path for RenderMan shaders (sets option).
RMAN_TEXTUREPATH	Search path for RenderMan textures (sets option).
RMAN_INCLUDE_FIX	When generating RIB, the post-include file for objects occurs inside the transform block if there is motion blur. This allows for correct motion blurring of ReadArchive data. Setting this environment variable causes the post-include to be included *after* the transform block (as it was in H4.1 and prior).
SESI_SLO_PATH	Where to build .slo shaders for automatically generated RenderMan shaders (i.e. when rendering to RenderMan and converting materials on the fly).

2.4 ABEKAS RELATED VARIABLES

ABEKAS_NTSC_XRES	Variable to override the Abekas NTSC X resolution.
ABEKAS_NTSC_YRES	Variable to override the Abekas NTSC Y resolution.
ABEKAS_PAL	Variable to set the Abekas image support to PAL instead of NTSC.
ABEKAS_PAL_XRES	Variable to override the Abekas PAL X Resolution.
ABEKAS_PAL_YRES	Variable to override the Abekas PAL Y Resolution.
LOGIN_NAME_ENV	A65 login name.
LOGIN_PASS_ENV	A65 login password.
RE_OVERRIDE_XRES RE_OVERRIDE_YRES RE_OVERRIDE_WIDTH RE_OVERRIDE_HEIGHT	These four variables allow you to manually set the screen resolution that Houdini uses, and the physical monitor size (in millimeters).

SESI_A60_HOSTS	List of Abekas A60’s host names.
SESI_A65_HOSTS	List of Abekas A65’s host names.

2.5 CINEON RELATED VARIABLES

CINEON_FLIP	When set (to any value) flips all Cineon images in Y during input.
CINEON_FILM_GAMMA	Specifies the gamma for Cineon files (default value 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. (default 685) Range 0 to 1023.
CINEON_BLACK_POINT	The Cineon log scale value that is considered to be full black and will be mapped on input to zero. (default 85) Range 0 to 1023.

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: Since Houdini 1.1, the environment variables: *CINEON_OVER_EXPOSURE* and *CINEON_WHITE_VALUE* are obsolete. Also, for backwards compatibility the lookup tables used in Houdini 2.0 can be enabled by setting the environment variable *CINEON_OLD_LOOKUP*.

2.6 IMAGE RELATED VARIABLES

HOUDINI_TIFF_BOTTOMLEFT	Older versions of Houdini generated TIFF files with the first scanline of data representing the bottom of the image. This control can be turned on to replicate the behaviour of older verions of Houdini.

HOUDINI_TIFF_SAMPLEFORMAT	If set, then a sample format tag is written to TIFF files. By default, Houdini does not add the SAMPLEFORMAT tag into TIFF files (this maintains backward compatibility). If the sample format tag is added to a file, older versions of the TIFF library (i.e. 5.x) will not be able to read the image.
SESI_RAT_USAGE	Holds the number of megabytes of RAM to use for textures when using the .rat image file format for texture maps. i.e. *setenv SESI_RAT_USAGE 16* will use a maximum of 16 Mb of RAM.
SESI_TIFF_COMPRESSION	Compression type used for TIFF files. Valid values are: “LZW” or “none”.
VERTIGOPIC	Create Vertigo style .pic files instead of Houdini format .pic by default.
WFGAMMA	Gamma value for the header when writing Wavefront .rla or .rlb files.

2.7 MISCELLANEOUS VARIABLES

CAPTFRAME	The Capture Geometry state create keys at the capture frame on the capturing hierarchy bone chains. This variable is used by the object-level character states to determine which capture frame to use when needed.
HOUDINI_ENABLE_FPS_SCALE	Set this variable if you want your keyframes to scale as they did in Houdini 4 and prior when setting the FPS in the *Global Animation Parameters*.
HOUDINI_FORCECONSOLIDATELOW	When set, causes fast point consolidation to behave the same as the slow consolidate, fixing some point order bugs. Because it changes the previous point order, it is not the default (Houdini 4.2 and higher).
HOUDINI_HIPEXT	Houdini adds a ".hip" extension to filenames that are entered in the Save dialog (if they don't already have it). This behaviour can be disabled by setting this variable.

UT_INTERRUPT_THRESH	The time delay before the interrupt cook dialog will appear. The time is specified in tenths of seconds; the default value is 60 (six seconds).
HOUDINI_TEXTURE_DEFAULT_COLOR	Sets the default colour for missing texture maps: `setenv HOUDINI_TEXTURE_DEFAULT_COLOR "r g b alpha"` causes missing texture maps to return the colour specified when accessed in VEX.
HOUDINI_UISCALE	Allows you to adjust the size of Houdini's UI (Fonts, Icons, and Menus). At the default value of 100, things are rendered at 85 dpi (the previous default on non-Windows platforms). Higher values generate larger icons and fonts, smaller values smaller. To revert to the previous behaviour set it's value to -1.
HOUDINI_WORKSHEET_BOXPICK	Box selection of operators in the Layout Area is by default tied to 🖱 in order to be consistent with the box-selection operations in the 3D Viewports and the Channel Editor. If you prefer to use a different mouse button, set this variable to “middle” or “right”.
HOUDINI_NOSPLASH	Disables the Splash Screen on startup.
RAY_DCSIZE	Size of the dice cache for *mantra3*.
RAY_SCSIZE	Size of the split cache for *mantra3*.
RAY_NO_FOGBOX	*mantra* automatically adds a background object that encompasses the scene in order to make the atmosphere visible. The object is simply a large box located at *0.8*far* with a matte shader applied to it. You can disable this behaviour by setting this variable.
SESI_COPY_SUFFIX	By default the filenames are common between all processes, however by setting the environment variable, SESI_COPY_SUFFIX you can make the clipboard files unique.
SESI_FILE_VIEWER	Replaces *actview* as the file viewer within file dialogs.
HOUDINI_DSO_ERROR	If this variable is set, then DSO errors will be printed out. This is particularly useful for HDK developers.

2.8 WINDOWS NT RELATED VARIABLES

HOUDINI_ACCESS_METHOD	This value can be 0, 1, or 2. It specifies what method Houdini will use to check file permissions. 0 uses the default method, which does a real check of file permissions using the proper NT security model. But this method can be quite slow on systems which use a login server elswhere in their network. Method 1 uses the NT security model again, and is much faster than method 0, but doesn't work on Win2k SP2 (Microsoft broke an important function in that release). Method 2 just checks the file attributes, which is fast and works, but ignores all the NT security stuff, so files and directories which are read only because of NT security permissions will report that they can be written to.
CD_PATH *(spy only)*	Though *spy* recognizes the standard csh/tcsh CDPATH variable, it also recognizes a variable CD_PATH which is synonymous with the CDPATH variable. On some NT systems, having the environment CDPATH variable set causes incorrect behaviour in shells. The CD_PATH variable is a work-around for this situation.
HOUDINI_TEXT_SPACES_FIX	In some cases, users of the NT version of Houdini may have problems with spaces not appearing in the Textport and Info Text displays. If you are having this problem, you should set this environment variable.
HOUDINI_DISABLE_XMMX	Disabling XMMX disables support for intels XMMX instructions. These are only present on P3s+. The flag is present due to the possibility of errors being present in them. Crashing with an "invalid op code" when running VEX functions is a sign that this may be at fault.
HOUDINI_DISABLE_CPUID	Disabling CPUID is present for older 386 era Intel machines & clones which lack the CPUID instruction. We'd detect for it directly, but this test is ineffective, and it is highly unlikely anyone will use such a machine.
HOUDINI_WINDOW_CONSOLE	Setting this variable forces the creation of a floating console, regradless if its output is redirected. This is only necessary for broken shells that incorrectly startup Houdini window applications with redirected output. At present, only required for *cygwin*-compiled shells.

2.9 BACKWARDS COMPATIBILITY

HOUDINI4_COMPATIBILITY	When enabled, some of the quirks of Houdini 4 are turned on. This is designed to be used to allow old hip files to be loaded. Support for these inconsistences is not guaranteed in future versions of Houdidi. Currently, this will: i) Change the order of points in spheres, tubes, torii and circules under certain orientations; ii) Reverse the direction of the Clip SOP's Distance parameter; iii) Revert to the old Capture Region weighting method.
HOUDINI_H4_CREGION_WEIGHTING	Houdini weighs points for Capture Regions in the Capture SOP using the Elendt model by default. Set this if you prefer the older method (i.e. 1-distance_squared)

2.10 VARIABLES NOT SPECIFIC TO HOUDINI

EDITOR	The editor to use when typing Alt E in an edit field.
PAGER	Default pager when viewing a text file using *actview* (i.e. the Unix commands: *less* and *more*).

2.11 NOTES ON HOUDINI PATHS

houdini_path

Description: This is the main Houdini path. If other path variables aren't set, their values are implicitly derived from the HOUDINI_PATH (or its default value).

Default:

$HIP/
$HIP/houdini/$USER
$JOB/
$HOME/houdini/dso
/usr/local/houdini/dso
$HFS/houdini/dso

houdini_dso_path

Description: Where plug-in DSO's are searched for.

Default:

$HIP/dso
$HIP/houdini/$USER/dso
$JOB/dso
$HOME/houdini/dso
/usr/local/houdini/dso
$HFS/houdini/dso

houdini_ui_path

Desc: Typically not used. This is the path to files found in *$HH/config* .
This path is really used by developers only.

CLIP, TEXTURE AND GEO PATHS

The CLIP, VEX, TEXTURE and GEO paths interpret special characters.
The special characters searched for are:

`@`	Expands to the current HOUDINI_PATH For example, with the default HOUDINI_PATH, @/geo expands to: $HIP/geo $HIP/$USER/geo $JOB/geo $HOME/houdini/geo /usr/local/houdini/geo $HFS/houdini/geo
^	Expands to the appropriate sub-path (for VEX). For example, with the default HOUDINI_PATH when referencing a surface shader: *@/vex/*^ expands to:

	$HIP/vex/Surface $HIP/$USER/vex/Surface $JOB/vex/Surface $HOME/houdini/vex/Surface /usr/local/houdini/vex/Surface $HFS/houdini/vex/Surface If the specific path variable isn’t set, the default HOUDINI_PATH is used (as is evidenced by the defaults).
`&`	Expands to the existing search path. For example, to add a single directory to a path, you might use: *setenv HOUDINI_GEO_PATH '~/jobs;&'*

houdini_clip_path

Description: Place to search for channel clips

Default: .;@/clips

houdini_texture_path

Description: Place to search for images (in COPs, SOPs or as texture maps)

Default: .;@/pic;@/map;@/maps

houdini_geo_path

Description: Place to search for geometry files:

Default: .;@/geo

houdini_vex_path

Description: Place to search for compiled *.vex* code. This also applies to the default include path for *vcc* .

Default: .;@/vex/include;@/vex/^

The portions are:

`.`	Current directory

`@/vex/include`	Expands to $HIP/houdini/vex/include;
	~/houdini/vex/include;
	/usr/local/houdini/vex/include;
	$HFS/houdini/vex/include

`@/vex/^`	Expands to $HIP/houdini/vex/^ ~/houdini/vex/^ /usr/local/houdini/vex/^ $HFS/houdini/vex/^ Where you should substitute ^ for the context which the VEX function is being used (i.e. SOP, POP, Surface, Displacement).

2.12 UI PATH VARIABLES

The following path variables are used for the 4.0 (and greater) UI:

HOUDINI_CFG_PATH
HOUDINI_TOOLBAR_PATH
HOUDINI_CUSTOM_PATH
HOUDINI_DESK_PATH

Note: There is no variable HOUDINI_SHADER_PATH that corresponds to RMAN_SHADERPATH or RMAN_TEXTUREPATH. These variables are used to define paths for RIB generation (and are obsolete now).

Every file that is found under $HH can usually be replaced with a file which is found earlier in the search path. However, in some cases, all the files found in the path are ‘merged’. These files are:

$HH/FBio
$HH/GEOio
$HH/CHANio
$HH/CHOPio
$HH/RGBcolors
$HH/FBrender
$HH/FBres
$HH/help/exprhelp
$HH/help

The other way that files can be excluded from the default Houdini Path is if the file fits into a ’special’ category. For example, .vex files can have their search path overridden by the HOUDINI_VEX_PATH variable. This means that users can choose to ignore the default Houdini path. However, all other files (i.e. .ds dialog script files) are only searched for within the HOUDINI_PATH.