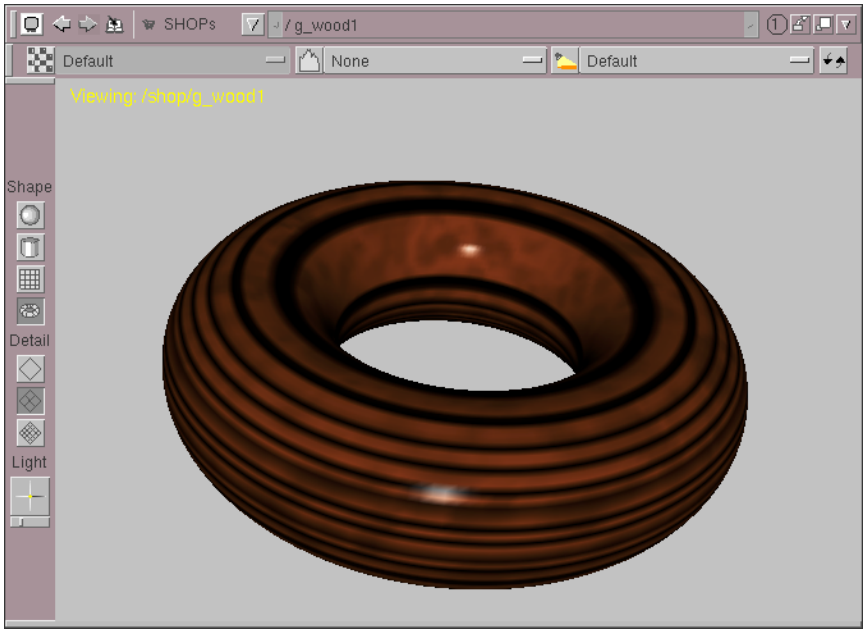
I SHOPs
Shader OPerations

I INTRODUCTION



Shader OPs (SHOPs) are based on a shading language similar to RenderMan™. The SHOP*s you place in the SHOP Editor provide a front-end to shader scripts written in the VEX scripting language*

CREATING YOUR OWN SHADERS

Please see the *VEX Scripting* p. 142 chapter for a complete tutorial on how to create your own VEX and SHOP shaders.

2 SELECT SHOP

2.1 DESCRIPTION

The Select SHOP allows you to select which SHOP to use for rendering based on valid render types and expressions.

When using multiple renderers, the Select SHOP will choose the first input which has a RenderMask which matches the target renderer. For example, if the first input is a RenderMan™ shader and the second is a VEX shader, then the first input will be used when generating RIB and the second will be used for *vmantra*.

Also, it is possible to animate which SHOP will be applied. This is useful when rendering with multiple passes. An expression can be written to switch inputs based on the render pass being executed.

2.2 PARAMETERS

INPUT 1 - 4 */activate1 - activate4*

Chooses the input (1-4) if the parameter is not 0. The first non-zero input which has a valid render mask will be selected.

2.3 SEE ALSO

- Switch SHOP

2.4 LOCAL VARIABLES

None.

3 SUB-NET SHOP

3.1 DESCRIPTION

The Sub-net SHOP is essentially a way of creating a macro to represent a collection of SHOPs as a single SHOP in the Layout Area. The Sub-net SHOP can contain an entire SHOP Network within it, stream-lining and simplifying your SHOP network both visually and conceptually.

Selecting *Edit Sub-Network...* from the SHOP's pop-up menu presents you with a new Layout area with four sub-network inputs. These four inputs are connected directly to the four inputs on the Sub-net SHOP in your original network. Proceed by attaching SHOPs as required to these four sub-network inputs. The display SHOP will be wired back to the output connector of the Sub-net SHOP in your original SHOP network. To get back to the original SHOP network, select *Quit...* from the *File* menu.

Please refer to the *Sub-Networks* p. 190 in the *Interface* section for a complete discussion and an example of how to use sub-networks.

Tip: Select several Operators that you want to make into a sub-network, and select *Collapse Selected* from the *SHOP's pop-up* menu to automatically create a sub-network out of them. You will see the selected Operators replaced by a single Subnetwork SHOP, and it will be properly rewired to contain the previously selected SHOPs.

3.2 PARAMETERS

INPUT #1 - #4 LABEL

These labels are displayed when you click with on one of the Sub-net SHOP's inputs. This is useful for remembering what the inputs are used for in your Sub-network.

4 SWITCH SHOP

4.1 DESCRIPTION

The Switch SHOP can be used to switch SHOPs based on a single expression. The input SHOPs are numbered starting at 0. This can be useful for switching SHOPs based on render passes. For example, setting the expression to something like:

```
$RENDERPASS
```

would allow you to switch SHOPs based on the *$RENDERPASS* variable (which would need to be set before rendering).

For finer control over switching between renderers, see the Select SHOP.

4.2 PARAMETERS

CHOOSE INPUT */input*

Which input SHOP to use.

4.3 LOCAL VARIABLES

None.

5 OTHER SHOP TYPES

5.1 DESCRIPTION

Because they do not interconnect like other types of OPs, but rather are programmed somewhat as self-contained shaders in the VEX language, the full gamut of SHOPs are not listed here. To understand the different SHOPs, you need to know only several basic types, whic are provided by the Generic SHOPs.

The Generic SHOP types allow you to generically test and create SHOP materials you've made using VEX Programming. You should see the next chapter *VEX Scripting* p. 142 for a tutorial.

5.2 SHOPS TO STUDY

The following generic SHOPs allow you to test your own SHOPs programmed in VEX. You might also want to study the VEX code for the Gingham and Riverbed SHOPs.

Generic Surface Displacement SHOP
: The Surface Displacement SHOP is a generic surface shader. It provides generic input fields for different renderers.

Generic Light SHOP
: This provides input fields for every renderer supporting light shaders.

Gereric Shadow SHOP
: This provides input fields for every renderer supporting shadow shaders.

Generic Fog SHOP
: This provides input fields for every renderer supporting fog.

2 VEX Scripting

1 WHAT IS VEX?

VEX stands for Vector EXpressions, and is a powerful yet relatively simple programming language available to all Houdini users. VEX does not require a third party compiler like C++, nor does it require the Houdini Developer's Kit (HDK). It is completely cross-platform, so a VEX operator will work without recompilation or rewriting on Windows NT/2000/XP, IRIX, Sun Solaris and Linux.

VEX is used in many different areas in Houdini: to alter or create pixels in the Compositor, to alter geometry points in the SOP editor, to alter particles in the POP editor, to alter or create channels in the CHOP editor and to affect the surface appearance at render time in Mantra 5. In fact, the latter usage in Mantra is probably the most "traditional" use of VEX, in that VEX started life as a shading language for use in the renderer and then quickly developed into a more general-purpose language.

The different uses for VEX (SOPs, POPs, COPs, CHOPs and Mantra5) are called "contexts" and each context is slightly different. For example, the SOP context deals with points while the COP context deals with pixels. However, VEX uses the same syntax and language structure regardless of which context is being used. In fact, in some cases, VEX operators from different contexts can be changed from one context to another simply by renaming the context in the operator.

VEX operators can be very roughly broken down into two types: VEX Shaders, and VEX Operators. This is because VEX Shaders do their work from within Mantra5, and VEX Operators are executed from within Houdini.

One of the other powerful features of VEX is that a parameter dialog is built automatically for you, so you can concentrate on functionality first, and then pretty up the operator with correct labelling, help etc. These parameter panels look and act exactly like 'normal' Houdini operators.

1.1 WHAT IS VEX NOT?

While VEX is a very powerful alternative to C++ it is not designed to replace the HDK in all situations. There are limitations in VEX and the user is cautioned not to expect VEX to solve all their problems.

For example, the VEX Surface Operators currently can only modify Point attributes. While this is a huge amount of power, it is limited in that it does not allow you to affect Primitives.

Likewise, the VEX Particle Operators cannot be used to create particles, only modify their attributes.

2 BASIC VEX CONCEPTS

Since all VEX operators are based on the same language and structure, there are some important basic concepts that are common to all operators.

Note: VEX is a relatively advanced topic, and it is beyond the scope of this document to explain many of the concepts required to use VEX. For example, a basic knowledge of the UNIX C-shell is required, as is a good understanding of the Houdini interface, creating operators, modifying parameters etc. If you are new to Houdini, please go through the basic tutorials first before embarking on your journey into VEX.

2.1 REQUIRED FILES

For a VEX operator to work in Houdini, there need to be two distinct files: The actual compiled VEX operator, *NAME.vex* and an accompanying dialog script, *NAME.ds* . There will also, generally speaking, be a third file *NAME.vfl* that contains the source code. However, this file is not, strictly speaking, required, so for example you might download a VEX operator from the Internet that does not include the source code (the *.vfl* file), but it will still function correctly in Houdini.

Any operators you write yourself will have all three files, since you do the actual writing of the VEX operator in the *.vfl* file. The other two files are generated automatically from the *.vfl* file as we shall see.

2.2 DIRECTORY STRUCTURE FOR VEX

The VEX operators use a directory structure to separate the different types of files. This directory structure acts very similarly to the rest of Houdini's internal workings, in that if you have files in your home directory, this files will take precedence over the standard Houdini files. In fact, this is the method we will look at in these exercises.

There is one main VEX directory, which is located in the *houdini5.0* directory. So, for example, in the standard Houdini installation, it is located in *$HH/vex* and in your home directory, it would be *$HOME/houdini5.0/vex* . Remember that $HFS is the environment variable pointing to where your Houdini is installed. This can be different on every installation of Houdini, so to make your life more simple, you can use $HFS in the C-shell to find the actual directory. Simply type:

```
cd $HFS
pwd
```

and you will see the actual directory that Houdini is installed in.

Within the */vex* directory, there are (up to) four Index files that 'point to' the dialog scripts for the non-shader VEX operators. These all start with VEX plus the type of operator. Currently, these are *VEXcop VEXpop VEXchop* and *VEXsop*. Also within the */vex* directory are subdirectories that contain the actual compiled VEX code. Each directory is named according to what type of VEX operator/shader it contains.

If you look in *$HH/vex* you'll notice that there are many more directories than *VEXsop* type files. This is because any VEX Shaders written to be used in Mantra5 also have another directory structure in the *houdini5.0* directory, to contain the dialog scripts for the SHOPs (Shader Operators) editor. We'll look at SHOPs later on, as their implementation is slightly different than all the other operator types.

Like all files in the $HH directory, if a file of the same name exists in the same place in your $HOME/houdini5.0 directory, the file in your directory will 'override' the file in $HH. This allows you have your own, personal, VEX operators in addition to the standard ones that are installed with Houdini. The specific files that Houdini looks at for this overriding feature are the *VEXcop* etc. files.

The last important directory (for now) is the *$HH/vex/Dialogs* directory. This directory actually has (up to) four subdirectories, corresponding to the four *VEXcop VEXchop VEXpop and VEXsop* files. These directories are where the actual dialog scripts go.

In fact, the *.ds* files do not need to go into their own separate directory. They can be wherever you like, as long as the Index file (*VEXcop* etc.) points to the correct location. For this exercise, we'll follow the standard Houdini installation, which uses the *$HH/vex/Dialogs* directory. As you become more familiar with the workings of VEX, you may wish to re-organize the location of your files.

To summarize an example for your home directory:

$HOME/houdini5.0/vex/
- Chop/
- Cop/
- Dialogs/
  - Sop/
  - Pop/
  - Chop/
  - Cop/
- Displacement/
- Fog/
- Light/
- Mat/
- Pop/
- Shadow/
- Sop/
- Surface/
- **VEXchop**
- VEXcop
- VEXpop
- VEXsop

2.3 COMPILING AND FILE PLACEMENT

All VEX operators need to be compiled with a programme called *vcc* . Generally, the sequence for creating a VEX operator is:

1. Create VEX code (from scratch or modifying other code)

2. Compile VEX code/create dialog script (*.ds* file) if dialog info has changed

3. Move *.ds* file to appropriate location if *.ds* file was created

4. Reload the *.ds* file and/or the VEX function, depending on what you changed/ added in step 2.

5. Test VEX operator.

6. Repeat.

The *vcc* command is a command-line operation for compiling, as is most of the interaction when creating VEX operators. VEX requires typing! But don't let that scare you.

When you compile a VEX operator and create a *.ds* dialog script, you'll need to move the dialog script to the appropriate location. Also, the very first time you create a new VEX operator, you need to edit the *VEXsop* (or whichever is appropriate) file to tell Houdini that your new operator exists.

3 CREATING A SIMPLE VEX SOP OPERATOR

Let's take a look at a step-by-step example of creating a VEX operator from scratch. This will be a very simple example, that will move points on a grid randomly based on their point numbers. Also, except where noted, this exercise should work the same on Windows NT, IRIX and Linux. On Windows NT we will use the Houdini C-shell, so the GNU tools need to be installed and configured properly, and a command-line editor needs to be installed. Notepad is the standard Windows NT editor. Contact your systems administrator if your C-shell does not appear to be working.

Note that the following initial steps of creating directories and copying some files only needs to happen once. Once it is set up, you can add VEX operators simply by creating them and adding a reference to the appropriate file.

Note: When instructed to type a line, be sure to type the Enter key at the end of each line, or the command will not be executed.

3.1 EXAMPLE

1. Open a C-shell. This shell must have access to all Houdini standalone tools. To test this, type:

```
vcc -h
```

A long list of text should appear, which is in fact the help for the *vcc* program that will be used later.

If this does not appear, contact your systems administrator, or check the *Getting Started* section for correct procedures for installing and using Houdini.

2. Type:

```
cd $HOME
```

3. Ensure there is a houdini5.0 directory. Remember that **ls** is the Unix command to list the contents of a directory. The $HOME/houdini5.0 directory will be created automatically when you run Houdini, as long as you've run Houdini previously, this directory should exist.

4. Type: *cd houdini5.0* which will take you into that directory. Since this is the first time (presumably) that you are working with VEX, there will most likely not be a vex directory. However, type *ls* to check. If you see a *vex* directory, someone else may have been working in your home account, as the *vex* directory is not created automatically by Houdini.

5. Type: *mkdir vex* assuming the *vex* directory does not exist.

6. Type: *cd vex* to enter the newly created *vex* directory.

7. Type: *mkdir Sop* to create the Sop VEX subdirectory. Note that the capital *S* is important! Houdini is case sensitive and if you do not use a capital 'S' and lower case 'op' Houdini will not see your operators, and you may get errors.

8. Type: *cd Sop* to enter into the Sop VEX subdirectory.

9. Type: *pwd* to confirm that you are in the correct place.

10. Type: *touch RandMove.vfl* This creates an empty file called *RandMove.vfl* which will be the name of our new VEX SOP. The only reason to create the empty file is because occasionally on Windows NT a file must exist before you can edit it (depends on the editor). You can skip this step on Irix, Sun Solaris or Linux.

11. You now need to edit this file. Since there are a variety of text editors and operating systems, it is not possible to give examples of all the different commands that could be used to edit this file. However, some common examples will be given:

In IRIX: type: *jot RandMove.vfl &* which brings up the *Jot* text editor
In Linux: type: *nedit RandMove.vfl &* which brings up the *Nedit* text editor.
In Windows NT: *notepad RandMove.vfl &* which starts the *Notepad* editor.

Tip: On Windows NT, an excellent (and inexpensive) text editor is *Textpad*. You can find it at *www.textpad.com* .

12. You now have an empty file, into which you can type your VEX code. For now, type in the following VEX code exactly as given. We will compile it and test it first, to make sure that Houdini is able find the operator. Later, we'll go back over the code and look at the conventions used.

This VEX operator will simply move points randomly along their normals. It will have the visual effect of 'randomizing' the surface. Enter the following code:

enter this code

```
sop
RandMove (float  height = 1;
                     int  myseed = 1;)
{
  vector  Nf = normalize(N);
  float  r = random(ptnum+random(myseed)*Npt);

  P += Nf * r * height;
}
```

Note: after the last **}** there must be a carriage return (Enter key).

13. After entering the code, save the file. You do *not* need to quit your text editor! In fact, it's good practice not to, as it will allow you to make changes very quickly and not have to constantly be re-loading the file.

14. In the C-shell (which should still be using *$HOME/houdini5.0/vex/Sop* as the current directory) type: *vcc -u RandMove.vfl* and check for errors. If all goes well you will see:

```
vcc -u RandMove.vfl
REMARK (3005) Outputting VEX code to ./RandMove.vex
REMARK (3006) Outputting dialog script to ./RandMove.ds
```

15. If you get any warnings or errors, double check that your code is exactly the same as above. Common mistakes are: forgetting the **;** (semicolon) at the end of the lines, forgetting the **{ }** (curly braces) and not capitalizing *N Npt* and *P.*

16. Assuming there were no errors, if you type **ls** in the Unix shell you should now see two new files, *RandMove.ds* and *RandMove.vex* in the current directory. Any file with a *.ds* suffix is a Dialog Script. Houdini uses these to build Parameter Dialogs. These files are generated only when you specify the *-u* option (known as a Flag) when compiling with *vcc*.

17. Now that there are *.vex* and *.ds* files, we're almost ready to start Houdini and test the new operator. All we have to do now is move the *.ds* file to its correct location, and alter a file to tell Houdini this new operator exists.

18. Type:

```
cd  $HOME/houdini5.0/vex/
```

to change directories into the *vex* directory. From here, type:

```
mkdir  -p  Dialogs/Sop
```

This will create the Dialogs directory and a Sop directory inside it. If the Sop directory already exists, you may get a warning, but nothing bad will happen.

19. Once the directory is created, the *.ds* file can be moved into its final resting place. Type:

```
mv $HOME/houdini5.0/vex/Sop/RandMove.ds $HOME/houdini5.0/vex/Dia-
  logs/Sop
```

20. Now, to create the Index file. Type:

```
touch VEXsop
```

This file called *VEXsop* contains the information that Houdini needs to know about any VEX Sop operators you have made. Since you almost always want your own operators to co-exist with the default installed VEX operators, we also need to tell Houdini to use yours along with the defaults.

21. Edit the file *VEXsop* file in your preferred text editor (jot, Notepad, or nedit) as you did with the *RandMove.vfl* file above.

22. Add the following two lines:

```
include $HH/vex/VEXsop
v_RandMove vex/Dialogs/Sop/RandMove.ds -label "VEX Random Mover"
```

and save the file. Then quit the text editor.

23. Launch Houdini. Generally, you should just be able to type *houdini* in the current C-shell to do this.

24. Edit the SOPs for the *ground* object, and append your *VEX Random Mover* to the Grid SOP. Turn on the Display flag, play with the *Height* and *Seed* parameters, and enjoy. You will find the *VEX Random Mover* operator in the *Filters* subfolder when appending SOPs.

25. The last step may not work as expected. The most obvious problem will be if the operator just doesn't appear in the list of SOPs. If this happens, there are three main places to check:

- Make sure that the file *VEXsop* exists in the correct location, which is: *$HOME/houdini5.0/vex/*

- Edit the *VEXsop* file and make sure the lines from step #22 are typed in correctly.
- Make sure that the file *RandMove.ds* is in the correct place, which is: *$HOME/houdini5.0/vex/Dialogs/Sop* or wherever you indicated in the *VEXsop* file.

26. If all of these are correct, and it is still not working, take a short break, have a nice relaxing swim in the river, and then scream very loudly. This usually corrects the problem. Seriously, the problem could be as small as a letter not capitalized, or something spelled wrong. Thoroughly check every period, comma, and capitalisation.

3.2 ANALYSING THE EXAMPLE CODE

The previous procedure was really only necessary once, in that for each VEX operator you need to edit the VEXsop file only once to tell Houdini about the *.ds* file, you need to create the various directories only once, etc. Once this is all set up, you only need to deal with your actual *.vfl* file, and perhaps moving a new *.ds* file to its correct place. These steps can also be automated, as we shall see.

Now that the basic procedure for creating the operator has been handled, we'll now take a look at the code to see what actually is happening, and make some changes to make the operator more flexible.

If you've closed the text editor, return to *$HOME/houdini5.0/vex* in the C-shell, and edit the *RandMove.vfl* file. If you're starting the text editor from the command line, make sure you use & at the end of the command (this doesn't work for VI however). This will allow you continue to the use the C-shell without having to quit the text editor.

DEFINING THE CONTEXT

Take note of the basic structure of a VEX operator. It starts with:

```
sop
```

This is the first line of the operator, and it defines what *context* the operator is, which is basically what type of operator you're creating. There are quite a few different contexts, which will be explored in later exercises. An operator of one context cannot be used in a different context! So, you cannot use an operator of type **sop** to modify Pops, for example. However, in some cases it is very easy to modify an operator of one context to work in a different context.

NAMING THE OPERATOR

```
RandMove (float  height = 1;
                              int  myseed = 1;)
```

The *RandMove* is the name of the operator. When you compile the operator with *vcc* the resulting *.vex* file will have this name. When writing operators, the normal convention is that the same name is used for the name of all the related files. So, in this case, the source file is *RandMove.vfl* the compiled result is *RandMove.vex* and Dialog script is *RandMove.ds* .

list of used parameters

After the name of the operator, any *parameters* are listed between the opening parenthesis and the closing parenthesis. The parenthesises are very important, and they need to be there even if there are no parameters. For example if there were no parameters, you would use:

```
RandMove()
```

to indicate your operator name and that there are no parameters.

The parameters being used, *height* and *myseed* need to be defined, which means they need to be told what type of numbers are associated with them, and they need to be given a default value. All parameters in VEX must have default values, which means the operator will do 'something' as soon as it is used.

PARAMETER TYPES

The different types of variables will be explored as we work through the exercises. The two *types* we've used here, *float* and *int* are indicating that the *height* variable can be any number or fraction of a number, such as 1, 1.5, 154.3223 etc. The *int* indicates that the *myseed* parameter can only be an integer number, in other words it cannot have decimals, only whole numbers such as -1, 5, 100. As we'll see later, there are also some other parameter types.

SEPARATING PARAMETERS

The syntax for defining the parameters is important too. Parameters of different types need to be separated by a semi-colon, however all the definitions don't have to be on the same line. So, these two lines:

```
RandMove (float  height = 1;
                            int  myseed = 1;)
```

are the same as:

```
RandMove (float  height = 1;  int  myseed = 1;)
```

Separating the lines like the first example makes reading the code easier. Also, there will be situations where you may have many parameters, which would create a very long line if left on the same line. Generally, lines should only be split at a semi-colon or comma.

BEGINNING AND ENDING OF THE BODY { }

Once the name of the operator and its parameters are defined, there are two curly braces { } used to indicate that everything else that follows is the actual program that does the work. So, in our RandMove example, there are three lines that do the actual 'work' of this operator. These three lines are contained within the curly braces. A VEX operator *must* start with an open curly brace defining the start of the code, and it *must* end with a close curly brace. If you omit either curly brace, you will get strange error messages (often "unexpected end of source").

DEFINING THE VARIABLES

```
vector  Nf = normalize(N);
```

This line defines a *local variable*, based on a *global variable*. The local variable, **Nf**, only gets used within the VEX operator and does not show up in the VEX operator's parameter panel in Houdini. The global variable, *N*, already exists 'automatically' within the VEX operator, and cannot be renamed. Global variables will be looked at in more detail in later exercises. The local variable can be named anything as long as the name doesn't clash with a global variable or another already existing variable.

Like with parameters, a local variable needs to be 'defined' with the type and also needs to be given a default value. In this case, we are defining the *Nf* local variable as a type *vector* and assigning it the value of *normalize(N)*.

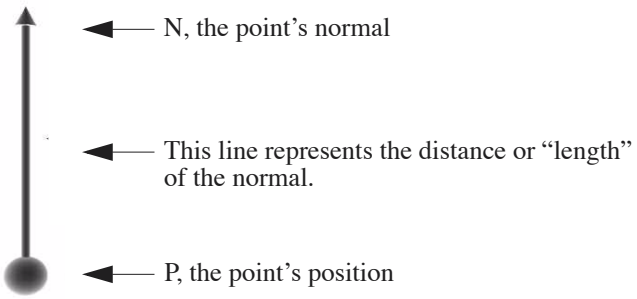
vector variables

A variable of type *vector* is actually three numbers, which can represent different things. For example, a colour can be represented as a *vector* since colours are defined by Red, Green and Blue. Positions can also be represented by type *vector* since a position is defined by X, Y and Z. This is the case here, since *N* is a global variable that holds the value of the point's Normal.

point normals

The Normal of a point is generally perpendicular to the surface that the point is part of. So, in the case of our Ground object grid, the Normals point straight up in the Y direction, since the grid lies flat in Z and X.

The *N* variable actually holds the 'end' point of the Normal. Since a normal is merely a line drawn from the point to wherever the normal ends, *N* is relative to the position of the point! There is also another global variable, *P*, which holds the point's position. Therefore, the normal for a point is a line drawn between *P* and *N*.



the normalize() function

The *function normalize(N)* takes whatever the *N* normal value is, and scales it so that the length will be within the range of 0 - 1. This makes the new local variable *Nf* more predictable, since we know that *Nf* will always be a length of within the range of 0 - 1 no matter what direction it points.

A function is simply a command that takes *arguments* (generally numbers, but not always) and 'returns' a value. In the case of the *normalize()* function above, the argument is **N** and it 'returns' new numbers based on calculations it makes on the argument, in this case a vector of length 1.

For example, if you have a vector of 42, 6, 100 and you 'normalize' that vector, you get 0.386641, 0.0552345, 0.920575 which are three numbers that have the same relationship to each other as 42, 6, 100 , but within a range of 0.0 - 1.0 .

It is important to note that the semi-colon **;** at the end of the line is required, or an error will occur. The semi-colon indicates the end of a command, and usually finishes a statement.

randomization

```
float  r = random(ptnum+random(myseed)*Npt);
```

This statement is defining another local variable, **r**, as a float, and uses two more global variables *ptnum* and *Npt* as well as a parameter defined at the beginning, *myseed*. The *ptnum* global variable is the number of the current point being 'worked on', which will be explained below. *Npt* is the total number of points being processed by the VEX operator.

The *random()* function returns a pseudo-random number between 0 and 1, based on another number (the argument) inside the ( ) known as the 'seed'. Pseudo-random means that the number appears random to humans, but in fact will always produce the same sequence if you use the same number as a seed. Houdini always uses pseudo-random numbers whenever a *random()* function is called, otherwise it would be impossible to replicate an effect!

By using the *ptnum* global variable, for every point on the geometry, a different pseudo-random number will be assigned to the **r** local variable.

This is because VEX operators work on a point by point (or pixel by pixel in the compositor, or particle by particle in POPs) basis, which means that if you have 100 points coming into the RandMove VEX Sop, the three lines of code between the **{ }** curly braces get executed 100 times, once for each point. This means that you can actually look at a single point, go through a VEX operator and predict the result, although it's hard to predict the *random()* function.

As an example, when point # 1 gets evaluated, the global variable *ptnum* will equal 1, the parameter *myseed* will equal whatever the user has set it to (let's say it's set to its default of 1) and *Npt* will equal the total number of points being worked on, which in a default grid is 100.

This line also illustrates how functions can be 'nested' inside other functions. These functions are evaluated from the inside out. In our example, this means:

random(myseed) gets evaluated first and returns a number, let's say 0.6523.
Then, this gets used in the next level of the function evaluation, like so:

```
float  r = random(ptnum+0.6523*Npt);
float  r = random(1+0.6523*100);
```

VEX evaluates math operations in the same order as you learned in highschool math. In other words, any division or multiplication is handled first, then addition and subtraction, so:

```
float  r = random(1 + 65.23);
float  r = random(66.23);
r  = 0.8201 (for example)
```

ACTUALLY CHANGING THE POINT POSITION

Now that you have numbers for each of your local variables *Nf* and *r* the final line in the operator does that actual change in position:

```
P += Nf * r * height;
```

Since all of the variables used here either exist (are global, like *P*) or have been calculated (local variables, like **Nf** and **r**) or are user-defined parameters (like *height*) this line has all the information it needs to proceed. As an aside, this is why all parameters must have default values, so that the operator will still function even if the user doesn't input a parameter value. In this case, *height* will equal 1 if the user does nothing.

the p global variable

The global variable *P* is special, because it is one of the global variables that you can read a value from (the point's original position) and also write a value back to (the point's new position). Some global variables, like *ptnum* and *Npt* cannot have a value written back to them, in other words you can get the value but you cannot change that value.

By changing the *P* global variable, you are changing where that point is in space! This is the fundamental core of any VEX operator: Altering a global variable, which then alters whatever it is you're working on, in this case geometry points.

SUMMARY

There are a couple of concepts to explore here. First of all, notice that *Nf* is a type vector, whereas *r* and *height* are type float. How can it be possible to multiply these? Whenever a vector is multiplied by a float, the float is actually 'copied' and made into a triplet like a vector. For example, if the value of *Nf* is 0, 0.8, 0.2 and you multiply that by *r* which from our above example is 66.23, the result will be (0*66.23), (.8*66.23), (.2*66.23) or 0, 52.984, 13.246 which is a vector. This is known as 'promoting' the float to a vector for multiplication purposes. The *height* gets dealt with the same way.

The second concept is actually shorthand. The operator *P* += is the same as: *P* = *P* + so once the *Nf* * *r* * *height* is evaluated, it is added to the value of *P* and then *P* is set to that value.

For an example (this is only an example, the numbers used may not relate at all to our real operator) if the result of *Nf* * *r* * *height* is: 0, 52.984, 13.246 then setting *P* is calculated like so:

```
P += {0, 52.984, 13.246}
```

which is the same as:

```
P = P +{0, 52.984, 13.246}
```

which takes the original position of the point *P*, adds the newly calculated number and then assigns that number back to the *P* position of the point, thereby moving it.

That's it! Now you know the basics of VEX.

4 CREATING A VEX SURFACE SHADER

The previous example modified geometry within Houdini. Now, let's look at altering the surface appearance at render time in *mantra5*. This type of VEX operator is generally known as a 'Shader' and only gives you a result when you render the surface it's applied to in *mantra5*.

Initially, we'll create a shader that mimics the standard 'Lambert' shading found in all 3D animation packages. This lighting model does not have specular highlights, only a surface colour. Later, we'll modify this shader to create texture maps and add specular highlights.

Also important in this exercise is creating and locating the Dialog scripts used in the SHOPs (Shader Operator) editor in Houdini. The location of the SHOPs *.ds* files is different than the other VEX operators. As before, we'll create the shader first, test it and then explain the code.

4.1 EXAMPLE

1. Quit Houdini. Remember that any time you create a new VEX operator you will at least need to reload (using dsreload) before you can use it.

2. Open a C-shell as you did in the previous exercise.

3. Move to the VEX directory in your home directory. Type:

```
cd  $HOME/houdini5.0/vex
```

4. Like the SOP VEX Operator we created previously, this Surface Shader will have its own directory. Type *ls* and check to see if a *Surface* directory exists. If not, create it with *mkdir Surface* and as always note that the first letter is capitalized. If you attempt to create this directory and it already exists, you will get a warning but no harm will be done.

5. Enter into the *Surface* sub-directory by typing: *cd Surface*

6. Create and edit a file called *Lambert.vfl* in this directory, in the same way you did in the previous exercise. The current directory should be *$HOME/houdini5.0/vex/Surface* . You can check this by typing *pwd* . Note that $HOME will be replaced by the actual path to your home directory.

7. Type the following code into a text editor:

enter this code

```
surface
Lambert(
      vector  amb=1;
      vector  diff={0.545, 0.525, 0.306};
      )
{
  vector      nml;
  nml = frontface(normalize(N), I);
  Cf  = amb  * ambient() + diffuse(nml);
  Cf *= diff;
}
```

Note: Don't forget the Enter after the last } . This may not be absolutely be necessary, however occasionally on Windows NT there are problems if you don't include a trailing carriage return. If you get an error "unexpected end of source" when you compile, a missing carriage return may be the problem.

8. Save the file. If you started the text editor using the *&* symbol at the end, you don't need to quit the editor. The *&* leaves the C-shell prompt available for use.

9. Compile the shader in the same way as before:

```
vcc  -u  Lambert.vfl
```

and check for errors. If there were none, you should see:

```
vcc  -u  Lambert.vfl
REMARK (3005) Outputting VEX code to ./Lambert.vex
REMARK (3006) Outputting dialog script to ./Lambert.ds
```

10. Once again, there are now three *Lambert* files: *Lambert.ds, Lambert.vex* and *Lambert.vfl* . Once again, the *.ds* file needs to moved, and a file needs to be edited to 'tell' Houdini about the new shader. The concepts here are very similar to the RandMove.ds file created in the previous exercise, but the location of the files is different.

The SHOPs pane in Houdini provides an integrated interface to shaders of different renderers. For example, both Renderman™ shaders and VEX Shaders can be used within the SHOPs editor. The actual shader itself, however, will be located in different places depending on the type of shader it is. So, in our case, the VEX shader is located in the *$HOME/houdini5.0/vex/Surface* directory. A Renderman™ shader would not be located there, but elsewhere in the directory structure. The *.ds* files, however, are all located together, in a directory structure based at *$HOME/houdini5.0/shop* and having a very similar structure to *$HOME/houdini5.0/vex* in that there subdirectories that hold the *.ds* files for the different shader types, and corresponding files that need to be edited to 'tell' Houdini about the shader. To summarize:

$HOME/houdini5.0/shop	**/displace**
	/fog
	/light
	/shadow
	/surface
	SHOPdisplace
	SHOPfog
	SHOPlight
	SHOPshadow
	SHOPsurface

Since we are writing a Surface shader, it should be fairly clear where the *.ds* file should be: *$HOME/houdini5.0/shop/surface* .

11. Move the *Lambert.ds* file to the appropriate location by typing:

```
mv  Lambert.ds $HOME/houdini5.0/shop/surface
```

12. You now need to edit the *SHOPsurface* file to 'tell' Houdini about the shader. Change directories into the *$HOME/houdini5.0/shop* directory. Create a new

SHOPsurface file if necessary *(touch SHOPsurface)* and edit it with your favourite text editor.

13. If the file is empty (it's a new file) add the following line:

```
include  $HH/shop/SHOPsurface
```

As with the *VEXsop* file we created above, this line makes sure that the standard Houdini surface shaders are 'seen' by Houdini. Without this line, only your own shaders would be recognized by Houdini.

14. Add the following line:

```
v_lambert  shop/surface/Lambert.d -label "VEX Lambert"
```

Again, like the VEXsop file, this indicated where the .ds file is, what label it should have in the SHOPs editor (VEX Lambert) and what the default name of the operator will be (v_lambert).

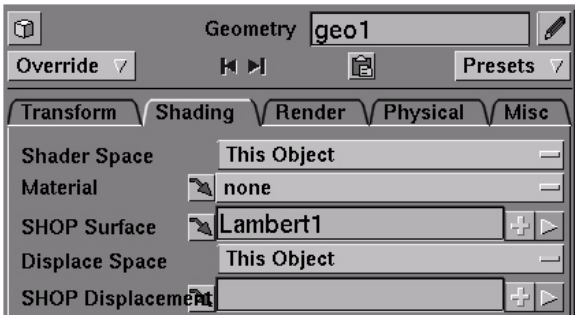
15. Save the file and exit the text editor.

16. Start Houdini. Change to the default Shops Desk, which should have the an Object Network Pane, a SOP network pane, a Shop network pane and an Object Viewer.

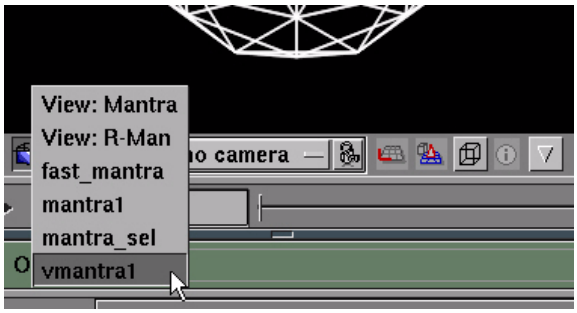
17. Delete all objects except *light1*, *cam1* and *ambient1*. Create a new geometry object (it will be called *geo1*) and inside the SOPs for *geo1*, delete the current SOP and insert a Sphere SOP.

18. In the SHOP editor, add your Lambert SHOP. This should be in the *Generators* sub-folder.

19. In the *geo1* object's Shading page, select the Lambert shader.



20. Select the *vmantra* renderer from the Viewport Render button.



21. Behold the wonder! Well, okay, it's a smooth-shaded sphere, but the surface appearance was defined by you in the Lambert shader. If you adjust the 'diff' parameter in the Lambert SHOP, and re-render, you can adjust the colour. You will notice that it is not very intuitive to deal with three numbers for RGB. Fortunately, there is a way to deal with this problem by 'hinting' to the **.ds** script what the numbers actually mean.

4.2 ADDING UI HINTS TO A VEX OPERATOR

1. Save this file, just in case. In Houdini 4.1 and later, there is no need to quit Houdini in order to reload dialog scripts.

2. In the C-shell, change directories back to *$HOME/houdini/vex/Surface* and edit the *Lambert.vfl* file with a text editor.

3. At the very beginning of the file, before the **surface** statement, add these lines:

```
#pragma  hint  amb  color
#pragma  hint  diff  color

#pragma  label  amb  "Ambient Colour"
#pragma  label  diff   "Diffuse Colour"
```

4. Save the file and compile with:

```
vcc  -u  Lambert.vfl
```

5. Move the **.ds** file to the correct directory:

```
mv  Lambert.ds  $HOME/houdini/shop/surface
```

6. To reload the .ds files to see the changes, open a Houdini Textport and type:

```
dsreload
```

You may want to make a function-key alias. You can do this with the *Dialogs > Aliases/Variables* menu, or by typing: *alias F9 dsreload* in the Textport. This will assign F9 to execute 'dsreload' whenever you type F9. Use another key if necessary.

SUMMARY

All VEX operators can use the #pragma syntax. In this case, we told the compiler that the **diff** and **amb** parameters were actually colours, and we also gave each

parameter more descriptive labels. See the reference manual for a complete list of #pragma commands.

4.3 ANALYSING THE CODE

As you can probably see from the first *RandMove.vfl* VEX operator, the syntax and structure of this shader are very similar to the VEX operator. Of course, since these two operators do very different things (one moves points, one colours a surface) the guts of the operators differ significantly, but they share common features. Ignoring the *#pragma* lines for a moment, here's a breakdown of the shader:

```
surface
Lambert(
  vector  amb=1;
  vector  diff={0.545, 0.525, 0.306};
                        )
{
    vector nml;
    nml = frontface(normalize(N), I);
    Cf  = amb  * ambient() + diffuse(nml);
    Cf *= diff;
}
```

The first line of the actual shader defines the context of the operator. In this case, it is a *surface* operator, designed to alter the surface appearance.

The second line gives the name of the shader, in this case *Lambert*. Like with the RandMove VEX Sop, this will be the name of the compiled *.vex* file and the *.ds* dialog script.

You'll recall that after the name of the operator there needs to be an open parenthesis and close parenthesis ( ) that contain any user-modifiable parameters. In this shader, this is the same.

```
  vector  amb=1;
  vector  diff={0.545, 0.525, 0.306};
```

These two lines declare a parameter *amb* to be a type vector with a default value of 1 (actually 1,1,1) and parameter *diff* also to be a type vector with a default value of 0.545, 0.525, 0.306 .

In the case of the *diff* parameter, the { } curly braces are used to indicate that the three numbers belong together in a group.

After the parameters are declared, the { opening curly brace indicates that the main part of the shader is beginning.

```
vector nml;
nml = frontface(normalize(N), I);
```

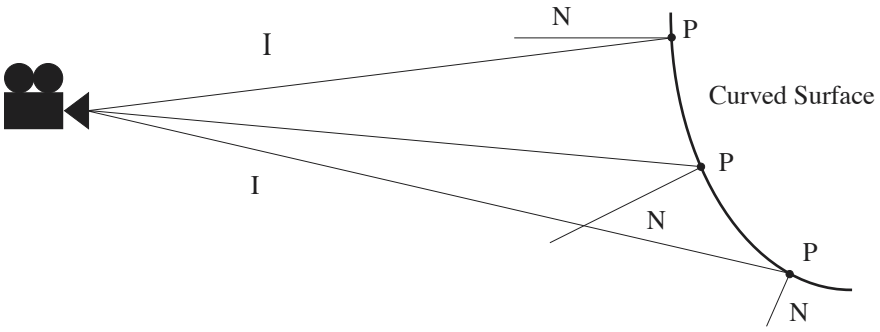
These two lines declare a local variable *nml* of type *vector* and then assign a value to it. In fact, the declaration of type can be combined with the assignment, like so:

```
vector  nml = frontface(normalize(N), I);
```

Either method is correct, the former simply being somewhat easier to read.

In the VEX SOP operator, the few lines of the actual operator got executed once for each point in the geometry, in our example above that was 100 times. In our surface shader, the four lines of code that 'do the work' get executed once for each time the geometry gets shaded. When *mantra5* shades an object (i.e. when a ray hits an object), the surface shader gets evaluated to compute what the color of the surface is at that point. All of the global variables are initialized to the correct surface parameters for the point being shaded (i.e. the *P* variable is set to the position that the ray intersected the geometry). Search the Reference manual for more information on *Shading Quality* and other topics.

The actual function *frontface* is a commonly-used command in a shader. Remember that the global variable *N* indicates the normal, in this case of the current point being coloured (shaded). The normal is very important in a shader, because it tells the renderer which way the surface is facing. For example:



In the above diagram, the camera is looking at a curved surface. There are three points *P* (out of potentially hundreds of thousands or millions) being used to represent what is happening.

Each point has a normal *N* which indicates which way that particular point is facing. On a curved surface, the various points will be facing different directions depending on which way the surface is facing.

From the camera to the point, there is a line **I** that is used to determine 'where' the camera is, relative to the point being shaded.

```
nml = frontface(normalize(N), I);
```

The *frontface()* function takes the normalized *N* of the point (remember that the *normalize()* function returns a vector whose length has been set to one) and compares it to the **I** vector (i.e. where the camera is). If it finds that the *N* is facing away from the camera, it 'reverses' the normal to make sure it is visible by the camera. Without this, there might be situations where your geometry has holes in it, or the 'wrong' side gets rendered. This line assures that the camera can always see the normals for all the points it is shading.

```
Cf  = amb  * ambient() + diffuse(nml);
```

This line actually alters the colour of the current point. The global variable *Cf* is used in the same way that *P* was used in the RandMove example. In this case, instead of moving the point, we're colouring it with some functions.

The real work in this line is being done by the *diffuse(nml)* function. For each point, it takes the normal's direction, looks at all the lights in the scene, and shades the

point appropriately. It is the *diffuse()* function that actually creates the appearance of three dimensions for this object!

The *ambient()* function, multiplied by the user-adjustable parameter *amb*, will colour the overall geometry based on the Ambient light value. If the parameter *amb* is set to 0,0,0 (black) then no ambient light is used. The ambient and diffuse values are added together for the final result, and assigned to the colour of the point.

```
Cf *= diff;
```

The final line takes the user-adjustable parameter *diff* (which is a colour) and the *Cf* calculated from the line above, multiplies them and then sets *Cf* to that final colour. Remember that *Cf *= diff;* is the same thing as: *Cf = Cf * diff;*

This is the final result for that tiny piece of the surface! Only several hundred thousand (or millions) more to go, and the object is fully rendered.

This Lambert shader can be easily modified to be a Constant shader, in other words, a shader that colours every point on the surface the same colour:

In your Lambert shader, change it to be:

```
#pragma  hint  diff  color
#pragma  label diff "Diffuse Colour"
surface
Constant(
  vector  diff={0.545, 0.525, 0.306};
                       )
{
        Cf = diff;
}
```

and make sure you save it as *Constant.vfl* , and not Lambert.vfl .

Compile it with: *vcc -u Constant.vfl* and move the *.ds* file to the correct directory (same as Lambert.ds). Likewise, edit the *SHOPsurface* file to refer to the new shader.

Try out your shader on a sphere or any other object in Houdini. It will always be a flat colour, whatever you select in the parameters. This is in fact the simplest form of shader. It does no calculations to determine where lights are (like the *diffuse()* function) but of course, it doesn't result in the appearance of a 3D object.

5 A SIMPLE DISPLACEMENT SHADER

5.1 INTRODUCTION

A displacement shader works very much like the Random Mover VEX SOP we created earlier, except that instead of moving *P* in geometry, it moves *P* when you render. Since the area being shaded at render time is very small, a very detailed displacement can be accomplished.

In fact, the *RandMove.vfl* file we created can very quickly and easily be made into a Displacement shader. It won't be pretty, and it won't do much, but it will work.

5.2 EXAMPLE

First, you'll need to make a new directory in *$HOME/houdini5.0/vex/* called *Displacement*. Copy the *RandMove.vfl* file that you wrote earlier into *$HOME/vex/Displacement* and then edit the new file:

```
displacement
RandMove (float  height = .1;
                  int  seed = 1, multrand = 100;)
{

  vector Nf = normalize(N);
  float  r = random(P*random(seed)*multrand);

  P += Nf * r * height;
  N = computenormal(P);
}
```

Once you have succesfully compiled the shader using: *vcc -u* , you'll need to move the *.ds* file to *$HOME/houdini5.0/shop/displace* and reference your *.ds* file in the *SHOPdisplace* index file, like with the Surface shader we did earlier. You will need to create the *displace* directory in *$HOME/houdini5.0/shop* that you will move the *.ds* file to if it does not already exist. In the *SHOPdisplace* file, which should be in *$HOME/houdini5.0/shop* , add these lines:

```
include $HH/shop/SHOPdisplace
v_randmove shop/displace/RandMove.ds -label "VEX Random Displacer"
```

If you look at the above shader, there are two obvious differences. First, instead of *sop* as the context, we're using *displacement* as the context.

Second, after P has been modified, we need to recompute the normal N so that the surface shader will render correctly. This is because the displacement part of the render is calculated before the surface shading part. Since the P is being changed before the surface is actually coloured, the normal N needs to be recalculated. Other than that, it's pretty much the same as the SOP.

One thing to be very careful of when doing displacement shading is how much you move the P. If you move it too much, you will use a huge amount of memory. There is an art as much as a science to doing displacement shading. A good rule of thumb

is: Do large amounts of relatively crude displacement with geometry (mountains, for example) and do small amounts of relatively detailed displacement in the renderer, for example rocks on the side of the mountain. In the *RandMove* displacement shader, a *height* of greater than around 0.3 will cause excessive memory usage.

There is a critical parameter in an object called *Displacement Bound* that must be used when doing displacement shading or you will get cracks and holes and general nastiness in the render. The general rule of thumb is, the Displacement Bound should be set to slightly more than the height of your displacement. So, for example in our RandMove.vfl displacement shader, if your displacement height is .1, your the Displacement Bound setting in the object being shaded with RandMove.vfl should be .1 or slightly higher. One of the nice things about 4.0's new Shops editor is that you can channel reference very easily between shaders and any other parameter. If you Yank the *height* parameter of the RandMove Shop and Put the Yanked Reference into the object's Displacement Bound parameter, you have now linked the two.

6 CONTINUING ON YOUR OWN

The three examples cover the fundamentals needed to create a VEX Operator that will work successfully within Houdini, and also a VEX Shader that will work successfully within Mantra5. The actual mathematics used, especially to create shaders, is beyond the scope of this manual.

6.1 RESOURCES

There are several good books available which cover some of the mathematics and the concepts used in greater depth. These include:

- The Renderman Companion : A Programmer's Guide to Realistic Computer Graphics – Steve Upstill.
- Advanced RenderMan : Creating CGI for Motion Pictures – Anthony A. Apodaca, Larry Gritz.
- Texturing and Modeling, A procedural approach – David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley.
- A Simplified approach to Image Processing – Randy Crane.
- The Art and Science of Digital Compositing – Ron Brinkmann.

This is only a small list! There are many more books on image processing available. Writing shaders is not the easiest thing in the world, but with some reading and perseverance, you'll find yourself writing useful shaders in no time. Writing VEX Operators in Houdini tends to be easier, as less esoteric math is involved.

One of the best resources for learning VEX is the library of examples that are installed with the default Houdini 5.0 installation. These examples are constantly being developed, so it is impossible to list them all at the time of this manual's writing. Look in $HH/vex/etc. for examples of all the different contexts. Copy them to your $HOME/houdini5.0/vex/etc. directory and then modify at will! When writing VEX operators/shaders, it is generally more efficient to copy an existing operator/shader and modify it. This is common practice and is perfectly acceptable. It is polite (and often legally required) to include the 'history' of the operator/shader, including the original source of the operator/shader and its author, in a comment at the start of the *.vfl* file.

6.2 REFERENCES

Don't forget the online VEX reference documentation. Complete lists of the global variables plus all variations in the contexts are covered. This can be found in:

$HH/vex/html/index.html

CONTEXTS

A list of the different contexts:

sop	Used in the SOP Editor to modify geometry in Houdini.
cop	Used in the Composite Editor to create or modify images in Houdini.
pop	Used in the Particle Editor to modify particles in Houdini.
chop	Used in the Channel Editor to modify or create channels in Houdini.
surface	Used in Mantra5 to modify the appearance of a surface at render time.
displace	Used in Mantra5 to modify points on a surface at render time.
fog	Used in Mantra5 to create fog-like effects at render time.
light	Used in Mantra5 to create lights or lighting effects at render time.
shadow	Used in Mantra5 to create shadows and shadow effects at render time.

7 MORE OPERATOR EXAMPLES – COPS

7.1 INTRODUCTION

Now that we have covered the details of placing the *.ds .vex* and *.vfl* files, let's look at some simpler examples of different operators. From here on, the assumption will be made that you understand the directory structure for different operators, and the relevant files that need to be edited.

7.2 EXERCISE

Let's create a VEX COP, that will very simply 'fold' an image in half, like so:


We'll call the file *flipper.vfl* and in this case, create it in *$HOME/houdini5.0/vex/Cop*
.

You will start with the context:

cop

and give it a name: *flipper( )*

Open the actual execution lines with the curly brace. ***Tip:*** Enter both the open and close curly braces now, and then you won't forget the close brace later! Of course, you shouldn't forget to insert the actual execution lines before the close curly brace.

```
{
}
```

Now, the actual lines that do the flipping:

```
vector4  c1;
if(Y >= .5)
   c1 = cinput(X,1-Y);
    else
    c1 = cinput(X,Y);

R = c1.r;
G = c1.g;
B = c1.b;
A = c1.a;
```

So far, your result should look like:

```
cop
flipper( )
{
vector4 c1;
if(Y >= 0.5)
   c1 = cinput(X,1-Y);
    else
    c1 = cinput(X,Y);
R = c1.r;
G = c1.g;
B = c1.b;
A = c1.a;
}
```

Go ahead and compile the VEX COP at this point with *vcc -u* . Don't forget to move the *.ds* file to *$HOME/houdini5.0/vex/Dialogs/Cop* and also don't forget to edit the file *$HOME/houdini5.0/vex/VEXcop* to point to the new *.ds* file.

Start up Houdini and try it out. Feed in some different images. It should be pretty obvious what it's doing.

This COP introduces a couple new concepts. Obviously, it deals with pixels instead of points or surfaces. The global variables that deal with pixels in the COP context are R G B and optionally A. These need to be set to something at the end of the operator, otherwise no colour changes will happen.

Two other global variables are used in this operator. X and Y correspond to the current pixel's position relative to the whole image. These are always values of 0-1 so for example the pixel in the middle of the image would be represented by 0.5 in either X or Y.

There are also global variables available so that you can deal with pixels explicitly based on the resolution of the image. See the online VEX reference for details on all the COP context global variables.

In this example, the *cinput( )* function simply access the pixel located at whatever XY coordinates you give it. So, the little function that 'mirrors' the pixels says "If Y is greater than or equal to 0.5, make the variable c1 equal the 'mirror' pixel, otherwise just make it equal the current pixel". So, as Y moves across the image, it increases from 0 to 1, but when it hits 0.5, the pixels being used start decreasing from 0.5 back to 0. To illustrate:

```
0 .1 .2 .3 .4 .5 (1-.6) (1-.7) (1-.8) (1-.9) (1-1) which is equal to
0 .1 .2 .3 .4 .5 .4 .3 .2 .1 0
```

This means that *cinput()* takes the first half up to 0.5 normally, and then the last half descending from 0.5 back to 0.

8 VEX HINTS AND TIPS

VEX is a reasonably straight-forward language, but there are situations that can trip up someone writing VEX code. Here are some common problems, and their solutions. Also look at *Type Casting* in the online VEX reference for additional information.

Some of these hints and tips assume a good working knowledge of VEX and also may be fairly job-specific. Look them over, but once you've had a chance to learn VEX, return and you may find something you didn't understand the first time through has become clear.

8.1 ASSIGNING DIFFERENT TEXTURE COORDINATES FOR SURFACE AND DISPLACEMENT SHADERS

There may be situations where you want a Surface shader to use different texture coordinates than a displacement shader on the same geometry. This is easy to accomplish because VEX will allow any shader to take in any attributes. For example, you could modify the standard VEX v_dmap shader found in $HH/vex/Displace to use a new attribute called **uv_disp** instead of the standard **uv**:

```
displacement
va_dmap(string map="";
  float  amp=0.1;
  int  forpoly=1;
  vector  uv_disp=0)
```

Basically, all you need to do is rename the "uv" attribute to "uv_disp" using the Attribute SOP. You also have to change the *isbound()* function call.

Then in the SOP chain:

```
<source_stuff>
        |
<texture_displacement>   # Apply texture coords for displacement
        |
<attribute SOP>    # Rename the uv attribute to be uv_disp
        |
<texture_surface>  # Texture coordinates for surface shader
```

This way, you have two sets of texture coordinates. One for surface and the other for displacement. Each one has it's own name, and the shader uses the one that's named for it.

8.2 DISPLACING POINTS IN GEOMETRY BASED ON A TEXTURE MAP

Displacing points on geometry based on a texture map is quite easy. There is one potential hang-up. People familiar with writing shaders may expect the **texture()** function to work, which will not. In fact, you need to use the **colormap()** function, like so:

```
sop
    dmap(string map=""; float scale=0.1; vector uv=0)
    {
        vector   amount;
        amount = colormap(map, uv);
        P += normalize(N)*scale * luminance(amount);
    }
```

This VEX SOP assumes that UV coordinates have been applied, typically with a Texture SOP. Look in the *$HH/vex/Sop* directory, there may be an example there.

8.3 DETECTING IF UV COORDS ALREADY EXIST

Many SOP operators, Surface shaders, Displacement shaders, POP operators etc use uv coordinates to access positions on surfaces, or in space or whathaveyou. If uv coordinates haven't been applied, the VEX operator/shader often fails completely unless a test is made. The following code fragment (shown in bold) uses the *isbound()* to 'detect' if an attribute exists. This attribute does not need to be UV, it could be any attribute.

This sets the local variable to be equal to UV if UV exists, and if it does not exist, it sets *coord* to be the point's position within the bounding box of the object being shaded.

```
sop
    dmap(string map=""; vector uv=0; ...)
    {
        vector   coord;
        if (isbound("uv"))
             coord = uv;
        else coord = relbbox(P);
        ...
    }
```

This will set up coord to be $BBX, $BBY, $BBZ if there is no UV attribute.

8.4 GETTING VALUES TO AND FROM VECTORS

Many things in VEX deal with vectors. RGB, uvw, xyz etc. are very commonly used. Often, you need to extract a single value, for example just the U or just the V values from the texture coordinates. There are two ways to accomplish this, and also to set a vector to a group of values:

Based on the assumption of a vector UV that holds the texture coordinates. The slow way to extract is:

```
u = uv.x;
v = uv.y;
w = uv.z;
```

The slow way to set:

```
uv.x = u;
uv.y = v;
uv.z = w;
```

The fast way (~ 4 - 5 times faster in most cases):

```
assign(u, v, w, uv);    // Assign the three floats from the vector
set(uv, u, v, w);     // Set the components of the vector.
```

Whenever possible, use the *set()* and *assign()* functions for maximum speed.

8.5 CREATING NEW ATTRIBUTES IN VEX

VEX can be used to create any attribute names you like. For example, to create the 'rest' attribute in VEX:

```
sop
restpos(export  vector  rest=0)
{
    rest = P;
}
```

The export keyword will cause an attribute to be created with the variable name rest. However, if the rest attribute already exists, in this case it will be set to *P* .

8.6 USING THE REST ATTRIBUTE IN A SHADER

When you write a shader based on P for example a noise shader, if the object deforms you may find that your texture 'swims' across the surface and doesn't appear to really be on the surface. This can easily be corrected by using the 'rest' attribute created in the Rest SOP. To do this, take a shader written as:

```
surface
testrest(float freq = 1;)
{
  vector  PP;
  PP = wt_space(P)*freq;
  Cf = noise(PP);
 }
```

and change it to:

```
surface
testrest(vector rest=0; float freq = 1;)
{
  vector  PP;
  PP = wt_space(rest)*freq;
  Cf = noise(PP);
 }
```

Naturally, the 'rest' attribute needs to exist or this will not work. Again, you can use the *isbound()* function to check for this and perhaps use *P* as in the first example if the *rest* attribute does not exist. For example:

```
surface
testrest(float freq = 1; vector rest =0;)
{
vector  PP;
 if(isbound("rest"))
  PP = wt_space(rest)*freq;
else
 PP = wt_space(P)*freq;
 Cf = noise(PP);
 }
```

This would use the *rest* attribute if it exists, and use P if not.

SHARING VEX ATTRIBUTES

Any parameter to a shader may be overridden by Any point, vertex, primitive or even a detail attribute. You just have to name your attribute to be the same name as the parameter in the VEX function.

Thus, to get a constant shader which uses point colors, you only need to use the Attribute SOP to rename the "Cd" attribute to be the "clr" attribute.

The technique of naming attributes can be used to modulate displacement amounts over surfaces (create a single float attribute and call it "amp" then apply the fractal dent displacement shader) – watch those displacement bounds though.

The technique can also be used to change any color in any shader. Try creating a point color attribute called "spec" and apply the brushed aluminum shader.

It can also be used in SOPs or POPs. Create a scalar (single float) attribute called "height" and run your geometry through a VEX Mountain SOP.

So, although not all VEX shaders support the "Cd" attribute, there's a fair amount of control.

8.7 DEBUGGING YOUR CODE

There may be times when your VEX code compiles without errors, but doesn't perform as expected. This is where the *printf()* function comes in handy. You can insert a *printf()* in your code and when the code is actually executed by Houdini, it will display the numbers being generated within the VEX operator/shader. Be warned that this can significantly slow down the execution of the operator/shader.

For example, if you wanted to see the numbers being generated by the *noise()* function in the above example, insert a *printf()* like so:

```
surface
testrest(float freq = 1; vector rest =0;)
{
vector  PP;
 if(isbound("rest"))
  PP = wt_space(rest)*freq;
else
 PP = wt_space(P)*freq;
printf("The noise is %g\n",noise(PP));
```

```
Cf = noise(PP);
}
```

See the online HTML reference for the complete syntax of the *printf()* function. Inside a shader like this, you'll also need to make sure that the Verbose option is enabled in Mantra5. If you're using Windows NT, you will also need to specify the *Output to File* option in the Render Command for Mantra5. Generally on NT, use the 'consolewait' option to get immediate feedback from your verbose output. See the *Output* section for Mantra5 options.

8.8 CREATING GROUPS IN SOPS AND POPS AND ADDING TO THEM

A common use of VEX is to group points or particles. This is done identically in particles and in SOPs:

```
sop
testgroups(string  groupname = "buddy";)
  newgroup(groupname);
    if (ptnum <= 5)
        addgroup(groupname, ptnum);
```

This very simple VEX SOP will create a new group called "buddy" and add all points numbered five or less to the group.

8.9 BUMP MAPPING IN VEX

There are three ways to implement a bump map in VEX:

1. Use the *bump()* functions:

```
surface
bumpy(float amount=.05)
{
    float       du, dv;
    float       tanu, tanv;
    vector      Nf;

    tanu = normalize(dPds);
    tanv = normalize(dPdt);
    bumpmap("mapname", du, dv, s, t);

    Nf = normalize(frontface(N, I));
    Nf += du*tanu + dv*tanv;
    Nf = normalize(Nf);

    Cf = diffuse(Nf);
    ...
}
```

2. Simulate moving *P* and recompute the normals after the fact.

```
surface
bumpy(float amount=0.1)
{
```

```
    vector      Nf;
    vector      PP;
    vector      mapclr;

    mapclr = texture("mapname", s, t);
    Nf = normalize(N) * amount;
    PP = P + luminance(mapclr) * Nf;
    Nf = computenormal(PP);
    Nf = normalize(frontface(Nf));

    Cf = diffuse(Nf);
    ...
}
```

3. Use a combination of the two.

```
surface
bumpy(float amount=.1)
{
    float       du, dv, lum;
    vector      mapclr;

    mapclr = texture("mapname", s, t);
    lum = luminance(mapclr);
    du = Du(lum) * amount;
    dv = Dv(lum) * amount;
    Nf = normalize(N) + du * normalize(dPds)
                      + dv * normalize(dPdt);
    Nf = normalize(frontface(N, I));
}
```

The second or third method will probably give you better anti-aliasing since the *texture()* function has built-in filtering of the texture map, while the *bump()* functions only do point sampling.

8.10 ALL VEX FUNCTIONS IN RADIANS

All VEX functions are calculated in Radians, so, for example, when using the *sin()* expression, you need to use Radians in the arguments. Use the *radians()* function to convert from degrees to radians, and the *degrees()* function to convert from radians to degrees.

8.11 USER-DEFINED FUNCTIONS

When creating user defined functions, one must be careful about which variables get modified. If the compiler gives an error about trying to modify a 'non-lvalue' then most likely you are attempting to modify a value that cannot be changed. An 'lvalue' is shorthand for Left Hand Value, which is a value that _can_ be modified.

For example, notice this code snippet from a VEX Cop. It does ***not*** work:

```
vector4
fwrapinput(float  fx, fy)
```

```
{
       fx = fx % 1; if (fx < 0) fx += 1;
       fy = fy % 1; if (fy < 0) fy += 1;
       return  finput(fx, fy);
 }
...
assign(r1,g1,b1,a1,fwrapinput(X-2,Y-2));
```

In this case, the *fwrapinput()* function is being declared as a user-function, and then gets used using X and Y as arguments. This does not work because all variables are passed by reference to a function, meaning that you can change their values (which you are doing).

The difficulty is that the value you're trying to modify is (X-2) which is an expression, not a value, thus you get an error.

The solution is to not modify the parameters passed in:

```
vector4
fwrapinput(float fx, fy)
{
    float  tx, ty;
    tx = fx % 1; if (tx < 0) tx += 1;
    ty = fy % 1; if (ty < 0) ty += 1;
    return finput(tx, ty);
}
```

8.12 TRANSFORMING PIXELS WITHOUT STREAKING IN COPS

When transforming pixels in a VEX COP, if you attempt to access a pixel that doesn't exist (i.e. one that is 'off the edge' of the image) VEX returns the value of the pixel at the edge of the image. This results in streaking along the edge of images that are being transformed. The following user-defined function will alleviate this problem by wrapping the input or clamping the input.

```
vector4
fwrapinput(float fx, fy)
{
  float  tx, ty;
  tx = fx % 1; if (tx < 0) tx += 1;
  ty = fy % 1; if (ty < 0) ty += 1;
  return  finput(tx, ty);
  }

vector4
clampinput(float fx, float fy)
{
  vector  clr;
  if (fx < 0 || fx > 1 || fy < 0 || fy > 1)
    clr = 0;
  else  clr = finput(fx, fy);
  return  clr;
}
```

8.13 TIPS FOR FIXING ERRORS

Reasonably often, when you compile a VEX shader/operator, a huge list of errors will be displayed. This can be intimidating, but don't let it stop you. In fact, one small mistake in the source code often results in many errors being displayed. The trick is to find the 'earliest' error and fix that. Then, when you recompile the code, you may find that all the errors have gone away. When an error message is displayed, it will generally indicate what line number the error is on. Find the first error, and fix it; then save and recompile. Often, the other errors will then go away. If they don't, find the next error, fix it, and repeat the process.

If you try to fix too many errors at once, you may find that what you're trying to fix is not an error at all.

8.14 SEARCH PATH FOR VEX OPERATORS/SHADERS

When loading VEX shaders and OPs, Houdini will search the Houdini path for the shader/operator (i.e. the *.vex* file). It searches $HFS/houdini5.0, $HOME/houdini5.0, etc. as is usual in Houdini. What this means, is that the Index files (VEXcop, SHOPsurface etc) only need to point to the dialog scripts. The actual *.vex* file can be anywhere in the search path and will be found.

8.15 VEX FILE LOCATIONS AND OVERRIDES

The VEX files that override *$HH* files (and those referenced inside these files) can be anywhere, as long as the .ds file is within the HOUDINI_PATH . The default path is:

```
$HOME/houdini5.0
$HFS/houdini5.0
```

With a statement like:

```
setenv HOUDINI_PATH "/houdini5.0:$HOME/houdini5.0:$HFS/
houdini5.0"
```

the first place we would look would be:

```
/houdini5.0/vex/VEXcop
$HOME/houdini5.0/vex/VEXcop
$HFS/houdini5.0/vex/VEXcop
```

Then the actual .vex file could live in:

```
/houdini5.0/vex/Cop/*.vex
$HOME/houdini5.0/vex/Cop/*.vex
$HFS/houdini5.0/vex/Cop/*.vex
```

3 VEX Language Reference

1 BASIC LANGUAGE INFORMATION

Note: See: $HH/vex/html/index.html for a complete set of all VEX functions.

The VEX compiler (vcc) compiles VEX code into an "executable" form. VEX is loosely based on the C language but takes pieces from C++ as well as the RenderMan™ shading language.

Unlike C or C++, VEX has different *contexts* for compiling. These contexts define how the function is to be used. For example, one context is the COP context. Functions written in this context can only be used to do image compositing. The POP context is a different context where functions are used to define the motion or attributes of particle systems. While a COP function processes pixel color information, a POP function deals with particle velocities and positions. Therefore, the information and functions required for each context have to be slightly different.

Each VEX context has different global variables as well as a special set of runtime functions suited to the context.

It is also possible to define user functions which return one of the standard VEX types (or void). These functions must be declared before they are referenced. The functions are in-lined automatically by the compiler, meaning that recursion is not possible.

Like with the RenderMan™ shading language, parameters to user functions are always passed by reference. This means that modifications in a user function affect the variable the function was called with.

The RenderMan™ shading language has certain restrictions on user functions which do not exist in VEX. It is possible to have multiple return points from within user functions. It is also possible to reference global variables from within user functions without requiring "extern" declarations. Although it is possible to reference global variables, this practice is discouraged since this limits the function to be used solely within the context containing the global variables referenced. Since parameters are passed by reference, it is probably better coding practice to pass references global variables to be modified within a user function.

The compiler expects that each source file contains one (and only one) context function definition. Any number of user functions can be defined.

Parameters to context functions are dealt with in a special way with VEX. It is possible to override a parameter's value using a geometry attribute with the same name as the variable. Aside from this special case, parameters should be considered "const" within the scope of the shader. This means that it is illegal to modify a parameter value. The compiler will generate errors if this occurs.

The RenderMan™ style of parameter declaration is used by VEX. This is similar to the ANSI C/C++ style of parameter declarations with some minor differences for context functions.

1. Parameters to a context function must be declared with default values. This does not apply to user functions.

2. Parameters of the same type are declared in a comma separated list without needing to re-declare the type.

3. Different type declarations must be separated by a semi-colon. For example:

```
void
user_function1(float a, float b, vector c) {...}

cop
cop_function(float a=0, b=0; vector c=1) {...}

pop
pop_function(string a="string1", b="string2";
             float c = 1, d = 1.3;
             vector e={1,2,3}, f={-1,0,.1}) {...}
```

1.1 LANGUAGE STRUCTURE

The structure of VEX is similar to the structure of a C program. A function is declared and consists of statements which operate on variables. Expressions are defined using the standard C operators which have the precedence as follows:

Operator	Associativity	Function
( )	left to right	Function call or expression grouping, Structure member
! ~ + - ++ -- (type)	left to right	Logical negation, ones complement, unary plus, unary minus, increment, decrement, explicit type cast
* / %	left to right	Multiplication, Division, Modulus
+ -	left to right	Addition, Subtraction
< > <= >=	left to right	Less than, Greater than, Less or equal, Greater or equal
== !=	left to right	Equal, Not equal
&	left to right	Bitwise and
^	left to right	Bitwise exclusive or
|	left to right	Bitwise or
&&	left to right	Logical and
||	left to right	Logical or
? :	left to right	Conditional
= += -= *= /= %= &= ^= |=	right to left	Assignment (or short-hand assignment)
,	left to right	Comma

1.2 STATEMENTS

The basic control statements in VEX are:

`{ }` — Multiple statements may be grouped together to form one statement by enclosing the statements in-side of curly braces.

`if-else` — One of two statements will be executed.

```
if (boolean expression) statement [ else statement ]
```

where statement is either a single statement or a series of statements enclosed by curly braces ({ }). The *else* clause is optional.

`while` — A statement can be executed repeatedly with the while construct. The statement is repeated as long as the boolean expression is true.

```
while (boolean expression) statement
```

`for` — Like the *while* statement, the for statement can execute a statement repeatedly.

```
for (expr; boolean_expression; expr) statement
```

The first expression is executed once before the loop is entered. The boolean expression is evaluated before the execution of the statement. The statement is evaluated only if the boolean expression is true. The second expression is evaluated after the statement.

`break` — A *for* or *while* loop can be terminated by using the break statement. Unlike the C language, there is currently no continue statement.

`return` — A user function (not a context function) can terminate execution at any time by using the return statement. If the function is non-void (i.e. returns a value), then the value must be specified.

```
return expr
```

NOTE

Some contexts have additional statements (i.e. the surface context adds an illuminance statement to the language). Please see the context specific information for further details.

2 COMPILER PRE-PROCESSOR

2.1 DIRECTIVES

The compiler has a pre-processor which is responsible for macro expansion as well as stripping out comments. Comments can be in either the C form (/* */) or the C++ form (//). The pre-processor is also responsible for handling encryption of source code. The pre-processor supports many of the standard C Pre-Processor directives:

`#define name token-string`
: Replace subsequent instances of name with token-string.

`#define name(arg,...,arg) token-string`
: Replace subsequent instances of name with token-string. Each argument to *name()* is replaced in the token-string during expansion.

`#undef name`
: Un-define the name macro.

`#include "filename"`

`#include <filename`
: Include the contents of the filename specified at this point in the compilation. When the quoted notation is used, the "current" directory is searched before the standard locations (include path). The "current" directory is the location of the current file being processed.

`#ifdef name`
: The lines following will be compiled if and only if name is a defined macro.

`#ifndef name`
: The lines following will be compiled if and only if name is not a defined macro.

`#if constant-expr`
: The lines following will be compiled if and only if the constant-expr evaluates to non-zero. The expression may contain the operators:
• Logical And/Or/Not (&&, ||, !)
• Equality Operators (==, !=, <=, >=, <, >)
• Bitwise And/Or/Exclusive Or/Not (&, |, ^, ~)
• Arithmetic Operators (+, -, *, /, %)
• Parentheses.
Expressions are evaluated from left to right (unlike the ANSI C standard of right to left). As with the ANSI pre-procssor, all numbers must be integers.

Note: There is also a special function *defined(name)* which will return 1 if the name is a defined macro or 0 if it is not. For example, to test whether symbols foo and fum are defined:

```
#if defined(foo) && defined(fum)
```

`#else`	The lines following will be compiled if and only if the preceding test directive evaluated to zero.
`#endif`	Ends a conditional secion of code begun by a test directive (#if, #ifdef, #ifndef). Every test directive must have a matching #endif
`#pragma crypt`	The following lines should be encrypted.
`#pragma endcrypt`	End of an encryption block.

2.2 SYMBOLS

Symbols can be defined using the -D option of vcc.

The compiler (vcc) has several pre-defined macros which can be used for compiling.

__vex	This symbol is always defined so that you know the source is being compiled by vcc.
__vex_major	The major version number of the compiler being used to compile.
__vex_minor	The minor version number of the compiler being used to compile.
__LINE__	The current line of the source code being compiled.
__FILE__	The current file being compiled.
__DATE__	The current date (as a quoted string). Example: "Dec 31 1999"
__TIME__	The current time (as a quoted string). Example: "23:59:59"

3 COMPILER OPTIONS

The VEX compiler (vcc) is capable of compiling VEX code, generating dialog scripts for VEX functions and also giving quick help by listing the global variables and functions available in any given context.

`-?, -H, -h`	Show help message for the compiler
`-X context_name`	Rather than compiling code, this option will display the list of global variables, VEX constructs and all functions available for the context specified.
`-D name=def, -D name`	Define a name for the pre-processor. If no value is given with the name, the name is defined with a value of 1.
`-I path`	Add the path specified to the include path for the pre-processor. By default, the standard Houdini path is searched for include files (under vex/include).
`-o file`	By default, the compiler will generate the compiled .vex code in the current directory. This option allows you to specify an alternate location and name for the output. It is possible to specify "stdout" as a filename. In this case, output will be generated to the stdout file descriptor rather than a disk file.
`-c`	Generate a binary/crypted version of the function. This means that the generated .vex file will not be readable by a human. However, it can still be used by Houdini. See also #pragma crypt.
`-e file`	Send error messages to this file rather than stderr.
`-w wlist`	The -w will supress the printing of the specified warnings. The wlist should be a comma separated list of warning numbers to suppress.
`-q, -Q`	The -q will cause the compiler to omit printing of messages. The -Q option will supress both messages and warnings. Errors will still be printed out with either option.
`-i`	Make the generated .vex code more readable by indenting the output based on nesting.

`-u, -U`

Generate a corresponding dialog script for the VEX function. This dialog script will be usable by Houdini to let the user modify parameters interactively (rather than editing a string). If the -U option is specified, only the dialog script file will be generated, compilation of the code will be bypassed.

`-g nparms`

When generating dialog scripts (with the -u or -U option), it is possible to "auto-group" parameters in to groups of N parameters. If no groups are specified using the #pragma group directive, then this option will force groups to be created with a maximum of nparms per folder tab.

`-v`

If you only have compiled VEX code, this option can be used to extract the parameter information and build a dialog script for the compiled code. Warning: All pragma information is lost in the compiled code so it is much better to generate dialog scripts from the source code where possible.

4 ENCRYPTION

In some cases, VEX code may contain proprietary algorithms which the author doesn't wish to become public knowledge. The compiler has a special set of directives to turn on/off crypting (#pragma crypt, and #pragma endcrypt). For example:

```
float
wavenoise(float height, float distance)
{
#pragma crypt
    return sin(distance)*height;
#pragma endcrypt
}
```

When this code is compiled, the output of the compiler will be encrypted so that the code is reasonably secure. However, since VEX does not support dynamic linking (i.e. linking of pre-compiled code), there is a utility vcrypt which will encrypt the specific portions of the source files. The compiler can still read these encrypted files, however, the code contained will be secure.

If the compiler detects encrypted source in its input stream, then the final output will be encrypted. This guarantees the integrity of the encryption (meaning it's not possible to reverse engineer an encrypted function by compiling it and decoding the assembler output). To generate encrypted object code, use the -c option on vcc.

The usage of the vcrypt program is:

```
vcrypt [source [destination]]
```

If no source and destination files are specified, then input is read from stdin and output to stdout. If no destination file is specified, the crypted code is output to stdout.

The #pragma crypt does not require a closing #pragma endcrypt. The two directives can be thought of as turning encryption on and off.

It is also possible to generate encrypted compiled code by using the -c option on the vcc command line.

5 DATA TYPES

VEX supports a fixed set of data types and does not allow user data types to be defined. As well, arrays are not currently supported within VEX. The data types supported by VEX are:

Type Name	Definition	Example
int	Integer values	21, -3, 0x31
float	Floating point scalar values	21.3, -3.2, 1.0
vector	Three floating point values. These values can be used to represent positions, directions, normals or colors (RGB or HSV)	{0,0,0}, {0.3,0.5,-0.5}
vector4	Four floating point values. These values can be used to represent positions in homogeneous coordinates, colors (RGBA)	{0,0,0,1}, {0.3,0.5,-0.5,0.2}
matrix3	Nine floating point values representing a 3D rotation matrix or a 2D transformation matrix.	
matrix	Sixteen floating point values representing a 3D transformation matrix.	
string	A string of characters.	"hello world", "Mandril.pic"

The standard C operations are defined (with the standard precedence order). There are several special exceptions to the C standard.

- Multiplication is defined between two vectors or points. The multiplication performs an element by element multiplication (rather than a dot or cross product).

- The dot operator (.) is defined only for vector and vector4. The structure names have been arbitrarily chosen to be:
  - .x or .r to reference the first data element.
  - .y or .g to reference the second data element.
  - .z or .b to reference the third data element.
  - .w or .a to reference the fouth data element (for vector4 types only).

- Many operators are defined for non-scalar data types (i.e. a vector multiplied by a matrix will transform the vector by the matrix).

• Constants are declared in a similar fashion to C:

1, 392, -43	**integer** constants
1.0, 3.14, -1e3	**float** constants
{1,2,3}, {0,1,0}	**vector** constants
{1,2,3,4}, {0,1,0,1}	**vector4** constants
{ {1,0,0}, {0,1,0}, {0,0,1} }	**matrix3** constant
{ {1,0,0,0}, {0,1,0,0}, {0,0,1,0}, {0,0,0,1} }	**matrix** constant

Note: Please refer to the Functions Guide, located online at:

$HH/vex/html/functions.html

for a complete guide to operators and functions.

6 TYPE CASTING

There are two separate ways to cast a variable or return type in VEX. VEX is a polymorphic language, meaning that the same function can have different signatures to specify different calling syntaxes. For example, the noise() function can take different arguments to generate 1D, 2D, 3D, or 4D noise (noise(float), noise(float, float), noise(vector), noise(vector4)). However, unlike similar languages (i.e. C++), the return code is also considered in constructing the signature of the function. This is an open ended problem, so in many cases, it is possible to give the compiler a "hint" as to which version of the function should be used.

This is done by function casting which simply gives the compiler a hint as to which version of the function to use. For example:

```
float       n;
n = noise(noise(P));
```

As stated above, the noise function can take different sets of parameters. However, there are also versions of the noise() function which return different types. In particular, the noise() function can return either a float or a vector.

When generating code for the above fragment, the compiler has a choice for the nested noise function. It can choose from:

1. float noise(vector)
2. vector noise(vector)

Both of these forms of the noise() function are valid, so the compiler has to guess which version to use. When it makes a guess, it will print out a warning:

```
WARNING (2005) Implicit casting failed for noise - guessing
                  noise@VF - please try to use an explicit cast
```

To eliminate this warning, we can use an explicit function cast which takes the form:

```
n = noise( vector(noise(P)));
```

This form of cast generates no additional code, it just tells the compiler which version of the function to use.

The other form of casting involves additional code generation and therefore is less desirable than the previous function casting. The second form is the form which is used in C and C++.

```
n = noise( (vector) noise(P));
```

This form will cause the compiler to guess the return code and then take the returned value and cast it to the type specified. The compiler has some heuristics to attempt to minimize the cost of functions, so in the above case, the nested noise() function will return a float, which is then cast to a vector. This is most likely not the desired behaviour, and is also more expensive.

However, the second form of casting is occasionally necessary. Consider the following example:

```
int         a, b;
float       c;

c = a / b;
```

In this case, the compiler will generate the instruction for integer division. If the floating point result is desired, then, it is important to cast the integers to floats. This is done using the second form.

```
c = (float)a / (float)b;
```

This, however, will generate additional instructions to perform the cast of the variables.

The general rule of thumb is to try to perform function type casting as much as you want. There is no run-time cost to this casting and it ensures correct code generation. However, it is also a good rule of thumb to avoid variable casting since this can incur a run-time cost.

7 VEX COMPILER PRAGMAS

The VEX compiler (vcc) supports pragmas for automatically building UI dialog scripts. These pragmas are typically ignored unless the -u option is specified on the vcc command line. The pragma's allow specification of help, hints for how to represent the parameter, improving readability etc.

The pragmas supported are:

```
#pragma callback
#pragma crypt
#pragma help
#pragma info
#pragma name
#pragma label
#pragma hint
#pragma range
#pragma choice
#pragma group
#pragma rendermask
```

7.1 PRAGMAS

#pragma callback

Generates the *callback* keyword in the dialog script file when UI is generated by vcc. See: *Ref > Scripting > Dialog Scripts > Syntax of a Dialog Script > callback* .

#pragma crypt

If this pragma is found in the source, the generated VEX bytecode will be encrypted. This prevents users of the source from reverse engineering the object code.

#pragma help "text"

The help pragma will add the text argument to the help in the dialog script. This can be used to give hints to users of the VEX code as to what parameters mean, what the code is useful for, etc.

Example:

```
#pragma help "This is help for the VEX function."
#pragma help It gets added automatically to the help text
```

#pragma info "text"

Like the help pragma, this information is displayed in the help for the dialog script. However, the info text shows up in a separate section of the help at the very beginning of the help display. This is intended to be used to specify any copyrights, version information, etc.

Example:

```
#pragma info "Created by Bob Loblaws - (c)2002"
```

Caveats: In Houdini 4.0, only SHOPs display the info text.

#pragma name "text"

This defines the label which appears in the UI. This pragma is typically not required since the label is now usually defined in the operator table definition.

Example:

```
#pragma name "Shiny Marble"
```

#pragma label parameter_name "text"

This allows the definition of a more descriptive label for a parameter.

Example:

```
// The "amp" parameter represents noise amplitude
#prgma label amp   "Noise Amplitude"
displacement bumpy(float amp=0) {...}
```

#pragma hint parameter_name hint_type

This pragma gives more information about what a parameter is meant to represent. For example, in VEX, a vector may represent a point in space, a color or a direction. This hint allows precise definition of what a parameter is intended to be used for. The UI generated for the parameter will then reflect this hint.

Hint Type	Meaning
none	No hint is available.
toggle	The integer (or float) parameter represents a toggle button. The UI will generate a toggle button for this parameter and generate values 0 for off and 1 for on.
color	The parameter represents a color. The UI will generate color sliders for this parameter.
direction	The parameter represents a direction vector. The UI will generate a direction gadget for this parameter.
position	The parameter represents a position in space. There is currently no special UI for this hint.
angle	The parameter represents a direction vector. The UI will generate an angle gadget for this parameter.
file	The parameter represents a disk file. A file dialog will be available for this parameter.
image	The parameter represents an image on disk. A file dialog will be available for this parameter. Only recognized image files will be displayed.
geometry	The parameter represents an image on disk. A file prompter will be available for this parameter. Only recognized geometry files will be displayed in the file prompter.

hidden — There will be no UI generated for this parameter. This is quite useful when a parameter is intended to be overridden by a geometry attribute.

Example:

```
#pragma hint __nondiffuse   toggle  // Define as a toggle button
#pragma hint specularcolor  color   // This represents a color
#pragma hint rest           hidden  // Don't show rest parameter in UI
#pragma hint mapname        image   // This represents an image file
```

#pragma range parameter_name min_value max_value

This pragma defines an ideal range for a parameter. The slider generated for the float value will have the range specified by the minimum and maximum values specified. This works for both integers and floating point parameters.

Example:

```
#pragma range seed  0 10
#pragma range roughness 0.001 1
```

#pragma choice parameter_name "value" "label"

When a choice pragma is defined for a parameter, the parameter is then represented as a menu of all the choice pragmas defined for that parameter. This is an exclusive list and there is no easy way for a user to set the parameter to something other than a choice in the list.

This can also be used to define integer values. However, the integer values ignore the labels and instead number the choices in order from 0 to N (where N is the number of entries in the menu). Example:

```
#pragma choice operation "over"     "Composite A Over B"
 #pragma choice operation "under"    "Composite A Under B"
 #pragma choice operation "add"      "Add A and B"
 #pragma choice operation "sub"      "Subtract A from B"

 cop texture(string operation="over")
 {
     if (operation == "over") ...    // texture coordinates
     if (operation == "under") ...   // parametric coordinates
     if (operation == "add")  ...    // orthographic
     if (operation == "sub")  ...    // polar
 }
```

This would define a menu for the parameter "operation". The menu would consist of 4 entries. The values for the string parameter would be one of "over", "under", "add" or "sub". However, the user would be presented with more meaningful labels for the operation types.

```
 #pragma choice operation "0"        "Use texture coordinates"
 #pragma choice operation "1"        "Use parametric coordinates"
 #pragma choice operation "2"        "Orthographic Projection"
 #pragma choice operation "3"        "Polar Projection"

 sop texture(int operation=0)
 {
     if (operation == 0) ... // texture coordinates
     if (operation == 1) ... // parametric coordinates
     if (operation == 2) ... // orthographic
     if (operation == 3) ... // polar
 }
```

#pragma group group_name parameter_name1 parameter_name2 ...

This pragma allows you to group like parameters into a single folder in the dialog box. There can be multiple pragmas for each group.

Example:

```
// Group Ka, Kd, Ks, roughness into a folder called BRDF
#pragma group BRDF Ka Kd Ks
#pragma group BRDF roughness
```

#pragma rendermask

This pragma is only useful for SHOP dialog generation. Each SHOP has a mask defining which renderers can use the SHOP. It is possible to have a similar shader written in the RenderMan shading language and also in VEX (or another shading language). In this case, the rendermask can be specified to include more than just VMantra.

The rendermask parameter is closely bound the the code which generates scene descriptions for a renderer. Thus, the renderer names are quite specific. At the time that this document was written, the different renderers which support SHOPs are:

- RIB - RIB generation for RenderMan compliant renderers.
- vMantra - The version of mantra which uses VEX for shading.
- OGL - OpenGL(tm) rendering. This is a special renderer which automatically adds itself to most render masks. There is currently no way to prevent this.