Houdini Modelling 101

by Dr Patrick Janssen

Version 3
August 2010

Table of Contents

Background

This booklet was written for students at the Department of Architecture, National University of Singapore. This document focuses on the concepts rather than the user interface, and it assumes that you have a basic knowledge of Houdini and the user interface. This basic knowledge can best be learnt by watching the various videos online. (Note that Houdini changed its interface significantly when they released version 9, but version 10 and 11 remain very similar to version 9. So any videos for version 9 are still relevant.) A good place to start is:

- Houdini 9 Interface Lessons
- Surface Operations

There is also a very useful Quick Reference Guide that will be a good reminder.

This booklet will continue to evolve – this is version 3.

What is dataflow modelling

Dataflow modelling is a procedural approach to creating parametric models. It allows designers to to efficiently explore alternative forms without having to manually build each different version of the design model for each scenario. Such systems are used by architects and engineers to automate the design processes and accelerate design iterations.

The dataflow approach may be familiar if you have used programs like Bentley's Microstation Generative Components and McNeel's Rhino Grasshopper. In these cases, plugins have been created for standard CAD packages in order to enable users to work with the dataflow modelling approach. In the case of Houdini, the dataflow approach has always been at the core of the whole software, and everything that you do is controlled by dataflow networks

What is Houdini

Houdini is a high-end 3D CAD and animation software. Historically, Houdini's main strength has been its particle animation system. Houdini chief distinction from other packages is its use of the procedural dataflow approach not just for modelling, but for for all tasks including animation, rendering, and compositing. Houdini's approach to dataflow modelling is more powerful than either Grasshopper or GenerativeComponents. This means that Houdini is a little harder to learn at first, bit it scales better as more complex tasks are undertaken.

How Houdini works

Modelling in Houdini consists of creating dataflow procedures. So, for instance, if you want to create an extruded surface, you don't draw it. Instead, you create a dataflow procedure to draw it. Such procedures take the form of networks of nodes.

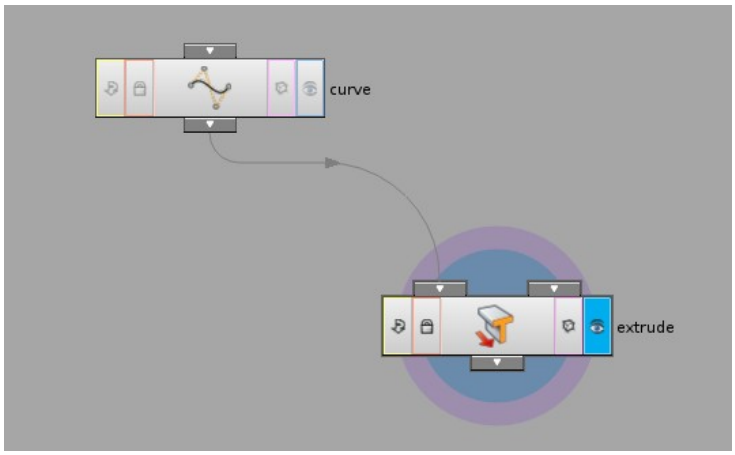

Figure 1: A network of two operator nodes. The first node creates a curve, and the second node extrudes the curve. The data for the curve flows down the wire, from the curve node to the extrude node.

Networks consist of nodes organised into a hierarchy and linked up using wires. There are two types of nodes: operator nodes and container nodes. Operator nodes do stuff, while container nodes are limited to containing operator nodes. The hierarchy of nodes is created using container nodes. Inside each container node, there may be one or more operator nodes and also other container nodes, all wired up together to achieve some result.

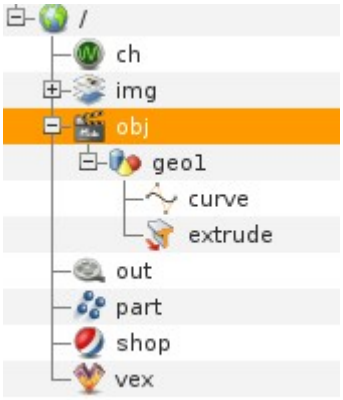

Figure 2: The hierarchy of nodes, shown in the Tree View. The '/' node is the root. Below the root are a number of top level default container nodes that always exist, including the 'obj' node, which contains all the geometry. The 'geo1' node is a container node that contains the two nodes shown in Figure 1: the curve and the extrusion.

Cooking is the process of driving information through the networks, from the outputs of operators to the inputs of operators to which they are wired, to create and animate the scene. By default, cooking is interactive, so every time you make a change, all affected nodes will be re-cooked.

Node Paths

The hierarchy of nodes is just like a folder system on the computer. You can navigate the hierarchy

using paths, like this: "`/a/b/c`". In this case the path starts at the root, indicated by the first "`/`". There is a container called 'a', which contains another container called 'b', which contains a node (could be either an operator or container) called 'c'. You can also use relative paths, using `..` to mean go up one level, and `.`, meaning the location where I am now. For example, the container of c is b, so if you are at c, you can point to it's container like this: `./...`

A nodes container is called it's parent. The nodes inside a container are called its children. All nodes have exactly one parent. Container nodes can have zero or more children, but operator nodes cannot have any children.

There are many different types of containers for gathering together operators that do many different things. The top-level containers (immediately after the first `/`) have fixed names that cannot be changed. The naming in Houdini is very distracting at first, but you get used to it. These are:

- `/obj`
  - OBJ network
  - contains geometry containers (called Surface Operator Networks or SOP Networks) which in turn contain the actual geometry in the scene, created using surface operators
- `/part`
  - Particle Operator Network, or POP network
  - contains operators for creating particle systems. Called a
- `/img`
  - Compositing Operators Network, or COP network
  - contains operators for manipulating 2d pixel data.
- `/out`
  - Rendering Operators Network, or ROP network.
  - contains operators for rendering
- `/shop` :
  - Shading Operator Network, or SHOP Network
  - contains operators for shading and materials
- `/ch` : contains operators for manipulating graphs
- `/vex` : contains some advanced stuff

The top level nodes may restrict the types of operator nodes that can be placed inside them. For example, nodes related to geometry like the `curve` and `extrude` nodes must be placed below the `obj` top level node. However, there are many exceptions. In particular, the `obj` container seems too be very flexible and con contain almost anything. For example nodes related to particles do not have to be under the `part` top level node - they can also be under `obj`.

Geometry

For now, we will focus on the `obj` container, which will usually contain all you scene geometry, and may also contain a lot of other stuff. Inside this container, you will create one or more geometry containers. And inside the geometry container, you will create one or more Surface Operators, or SOPs. (This is an odd name, especially since SOPs don't only create surfaces – they also create points and curves, etc. But I guess there was a reason – and now we are stuck with it.)

For example, the path to a curve might look something like this:

- `/obj/geo1/curve`

In this case, `/obj` is the main container node, `geo1` is the name of the geometry container node, and `curve` is the name of the sphere SOP. (See figure 2.)

In the documentation, geometry containers are also referred to as objects. The geometry container object lets you translate (move, rotate, and/or scale) and shade the entire object, as opposed to individual surfaces inside the object. So you can see it as being similar to grouping in other software. There are also other ways of grouping, to be discussed later.

For operator nodes, they can either be *generators* or *processors*. Generators generate new geometry, like the `sphere` node. Processors process data from further up the network. For example, the `subdivide` node does not generate any geometry, and instead processes the geometric data passed into it. Some nodes do both – they process some data from further up and generate some new geometry. To understand how nodes process data, you first need to understand the data itself.

Attribute data

The geometric data that nodes process is represented as a set of attributes. Attributes include things like x,y,z positions, normal vectors, colour values, etc. The data is organised as a kind of huge geometry spreadsheet of numbers (see figure 3.) For example, if you look at the points in the geometric attribute data spreadsheet, the point numbers are listed down the side, the point attributes are listed across the top, and the point attribute values are contained in the table. Each node will create, add to, and/or filter this data. So you should think of attribute data as flowing through a network of nodes, being passed from one node to the next.

Node: extrude

View

Points
Vertices
Primitives
Detail

	P[x]	P[y]	P[z]	P[w]
0	1.05595	0	0.365836	1
1	-0.748954	0	0.992726	1
2	-1.36831	0	-0.10009	1
3	-0.840612	0	-1.28547	1
4	1.05595	1	0.365836	1
5	-0.748954	1	0.992726	1
6	-1.36831	1	-0.10009	1
7	-0.840612	1	-1.28547	1
8	-0.840612	0	-1.28547	1
9	-1.36831	0	-0.10009	1
10	-0.748954	0	0.992726	1
11	1.05595	0	0.365836	1
12	1.05595	1	0.365836	1
13	-0.748954	1	0.992726	1
14	-1.36831	1	-0.10009	1
15	-0.840612	1	-1.28547	1

Figure 3; The Geometry Spreadsheet for the extrude node, showing default point data. This spreadsheet is visible by selecting 'Details View'. Point data can be shown by selecting 'Points' from the drop down menu.

Houdini organises its attribute data in a very standardised way. Whether you are working on primitives, NURBS surfaces, meshes, etc, the data is always stored in the same way. The difference lies in how the data is used to generate geometry.

There are four levels to the organisation: points, vertices, primitives, and global (or detail).

- *Point* attributes include things like position and normals.
- *Vertex* attributes include things like colour.
- *Primitive* attributes include things like normals, area, and length.
- *Global* (or *Detail*) attributes include things that relate to all the geometry within that node, for example the total surface area of a set of primitives. For these attributes, there can only be one instance of the attribute value in the geometry. This is also sometimes referred to as a 'detail' attribute, which seems contradictory, since it sounds like the opposite of 'global'. I prefer 'global', so I will stick to that in this text.

Points are points in space, and primitives are entities like polygons or surfaces created from set of points. However, rather than referencing the points directly, the primitives are defined using vertices. The reason for this is that primitives can share points. For example, imagine two adjacent triangles that share two points. The way this is represented is that in total there will be four points, six vertices and two primitives. For the shared points two different vertices will reference the same point.

Global attributes are a bit different for the others since in this case, the table can only ever have one row. For the points, vertices, and primitives, there will usually be multiple rows. The reason that there is only one row for globals is that they are attributes that are actually global to that whole set of geometry. For example, you may have a box with six faces, and each face can have an area. This area information can be represented as an attribute for each face primitive – so, in the primitives table, there would be six rows, and each row would include a column for the area. If you then wanted to include the total area of the box surface, you could then add up the six areas, and create a global attribute for all the faces.

Some attributes (such as position) are automatically generated by Houdini as you model the points, but you can also manually set attributes and add your own custom attributes.

When creating SOP networks, it is very important to keep reviewing at the attribute data, and tracing how it changes as it flows through the network. To see the attributes on the geometry in a node, right-click on the node's tile in the network editor and choose Spreadsheet to open the geometry spreadsheet for the node.

Primitive types

Geometry is made up of primitives of various different types. These can be categorised into the number of parametric dimensions – a point is zero dimensional, a curve is one dimensional, a surface is two dimensional, and a volume is three dimensional.

- One dimensional primitives (curves):
    - Polygon Primitive: Shapes constructed from a series of straight edges. These edges are defined by a series of vertices.
    - NURBS Primitive: Smooth curves constructed from control vertices. NURBS curves have a start and end vertices, with any number of control vertices in-between. The curve will always pass through the first and last vertices, but not necessarily through the other in-between control vertices.
    - Bezier Primitive: A special case of a NURBS curve. With Bezier curves, the curve must have two control vertices - the first point is the start vertex, the second and third

vertices are control vertices, and the fourth vertex is the end vertex.

- Two dimensional primitives (surfaces):
    - Primitive Primitive: Simple geometric entities such as circles, spheres, and tubes controlled by a small number of parameters. These entities are very lightweight and consume very little memory.
    - Polygon Primitive: Similar to a one-dimensional polygon, except in this case, it is closed to form a surface. If the vertices do not lie on the same plane, then the polygon surface will be non-planar.
    - Polygon Mesh Primitive: A collection of polygons with guaranteed ordering.
    - Mesh Primitive: A collection of polygons with guaranteed ordering. It is much more efficient that the equivalent polygons, and unlike most regular polygons you can convert it directly to NURBS. Unlike the polygon mesh primitive, the mesh primitive represents a complex surface as a single primitive.
    - NURBS Primitive: A NURBS surface that employs a series of blending functions called “bases” to generate a smooth surface from a sequence of control vertices (CV’s) which define a NURBS hull.
    - Bezier Primitive: A special case of a NURBS surface. With Bezier surfaces, the surface must have 4 control vertices and 12 edge vertices, forming a grid of 4 x 4 vertices.
- Three dimensional primitives (volumes):
    - Volume Primitive: Volumes are boxes with a position, size, and orientation, subdivided into a 3D grid of points, with a value stored at each point.

For more information see:

- Polygons and meshes
- NURBS Curves and Surfaces
- Volumes

Components

Looking at the attribute data in the spreadsheet is one thing. But if you are modelling, you often need to work interactively by selecting geometry.

Components are the sub-elements of geometry that can be interactively selected and manipulated in the user interface. The types of components are *points*, *vertices*, *primitives*, and *edges*.

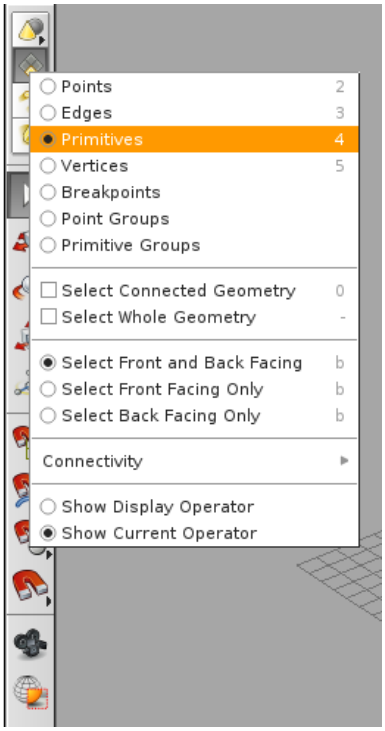

Figure 4: In order to work with different types of components, you can change the selection settings. In this case, the selection setting is set to 'primitives'.

You may be wondering how this relates to attribute data. For attributes, you have points, vertices, and primitives (lets ignore the global attributes for now). And for components, you have one extra one – edges.

The attribute data is the lower level representation – it is at this level that everything is stored, including edges. However, the information about how vertices are joined up to create edges is stored implicitly in the numbering system for the vertices.

For example, lets say you create a polygon mesh cube consisting of just six primitive faces. Looking at the vertices in the Details View, you will notice that they are numbered like this:

Node: box1 View

	Point Num
0:0	0
0:1	1
0:2	3
0:3	2
1:0	4
1:1	5
1:2	7
1:3	6
2:0	6
2:1	7
2:2	2
2:3	3
3:0	5

Figure 5: The Geometry View for a polygon mesh cube. The column on the left shows the number system for vertices, and the column on the right indicates the point number. The first face for this cube as four corners, numbered 0:0, 0:1, 0:2, and 0:3. The '0' before the colon indicates the face

number, and the number after the colon indicates the vertex number.

Thus vertices are like a middle-men between the points and the primitives. The points and primitives do not store any relationships. It is vertices that store all the relationships. This is done as follows:

- In the vertex number, the number before the colon indicates the primitive number
- In the vertex number, the number after the colon indicates the edge order. For meshes, consecutive numbers are connected by edges. So 0:0 is connected to 0:1, 0:1 is connected to 0:2, and 0:2 is connected to 0:3. In addition, the last one is connected to the first one: so 0:3 is connected to 0:0. For other geometry types such as NURBS it is slightly different.
- The value for the Point Num attribute indicates the point number that this vertex is associated with.

Another aspect of components is that they can be grouped. More on this later.

Wiring of nodes

Now you should have an understanding about attribute data, and how that relates to the components that can be selected in the user interface. Now, lets go back and look at the nodes in a little more detail.

Nodes have inputs and outputs. The output from one node can be connected to the input of another node, which will result in a line, or 'wire' connecting the two nodes. Connections between nodes have different meanings in different network types.

- In Object (OBJ) networks, the connections control parenting (connect child objects to the outputs of parent objects). (See figure 6)
- In geometry (SOP), particle (POP), and compositing (COP) networks, connections control the flow of geometry data through the network. (See figure 1.)
- In render output (ROPs) networks, connections control dependencies between render passes. More on this later.

For the moment, will will focus on OBJ and SOP networks. With OBJ networks, the parenting relationship means that a child object's transformations (moves, rotations, and scales) will be relative to those of a parent object.

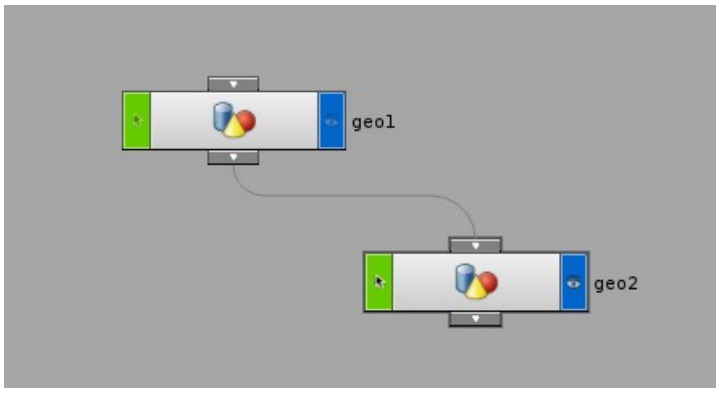

Figure 6: Two geometry containers (or objects) wired together at the 'obj' level. The 'geo1' node is the parent of the 'geo2' node. This means that any transformations you apply to 'geo1' will also affect 'geo2', but transformations applied to 'geo2' will not affect 'geo1'.

With SOP networks, the flow of geometry means that the attribute data from one node will flow

down to the next node. Often you will see that the attribute data gets larger as it flows through the network – i.e. attributes are added as you go.

Grouping

When working with geometry using in networks, attribute data from different geometric entities often merges together. In order to un-merge this attribute data, you can collect components into groups. This allows you you can move, edit, etc. a group of points or faces together. Groups are essential to modelling just about anything in Houdini so it is important to understand how they work.

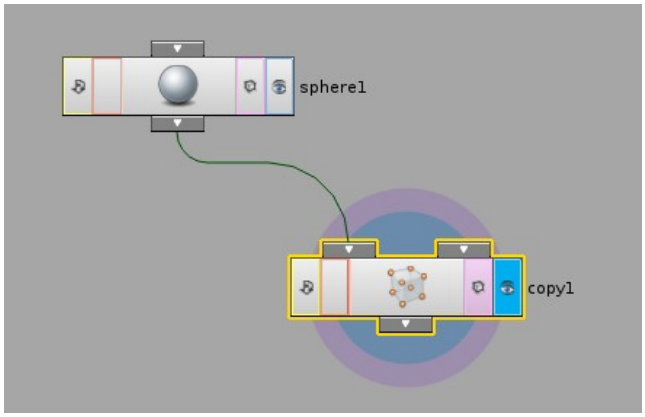

Figure 8: Geometry network a sphere is copied a number of times. What happens if you now want to further manipulate just one of the resulting spheres?

For any particular SOP network, the attribute data for various pieces of geometry will simply merge into one large set of data. For example, lets say you create a polygon sphere and copy it a three times using the `Copy` node. If you look at the geometry spreadsheet for the original polygon sphere, you will see that there are a bunch of primitives that define the faces of the sphere. If you look at the geometry spreadsheet after copying, there will now be lots more of primitives – three times as many. But there will be nothing that indicates which sphere each primitive belongs to. The same is true for points and vertices – everything just becomes one large flow of data.

Lets say you now want to move the second sphere. You can append a `Transform` node, and set the translation parameters. But this will move all three sphere, not just the second one. In order to limit the effect to just the second one, you will need to first need to switch on 'Create Output Groups' in the `Copy` node, and then specify the group name in the `Transform` node.

Most nodes that operate on points or primitives give you the option to apply the node's effects only to one or more groups in the input stream, instead of every point/primitive. So, you can identify groups of points based on certain properties, and then only apply some nodes to those groups, and not other points. The `Group` node lets you sort points or faces in its input geometry into groups based on an various criteria. Other operators such as the `Copy` node may also create groups as a side effect.

To verify that the groups have actually been created, you can open right-click on any node and view 'Extended Information...', which will display the group names and number of components for both Primitive Groups and Point Groups.

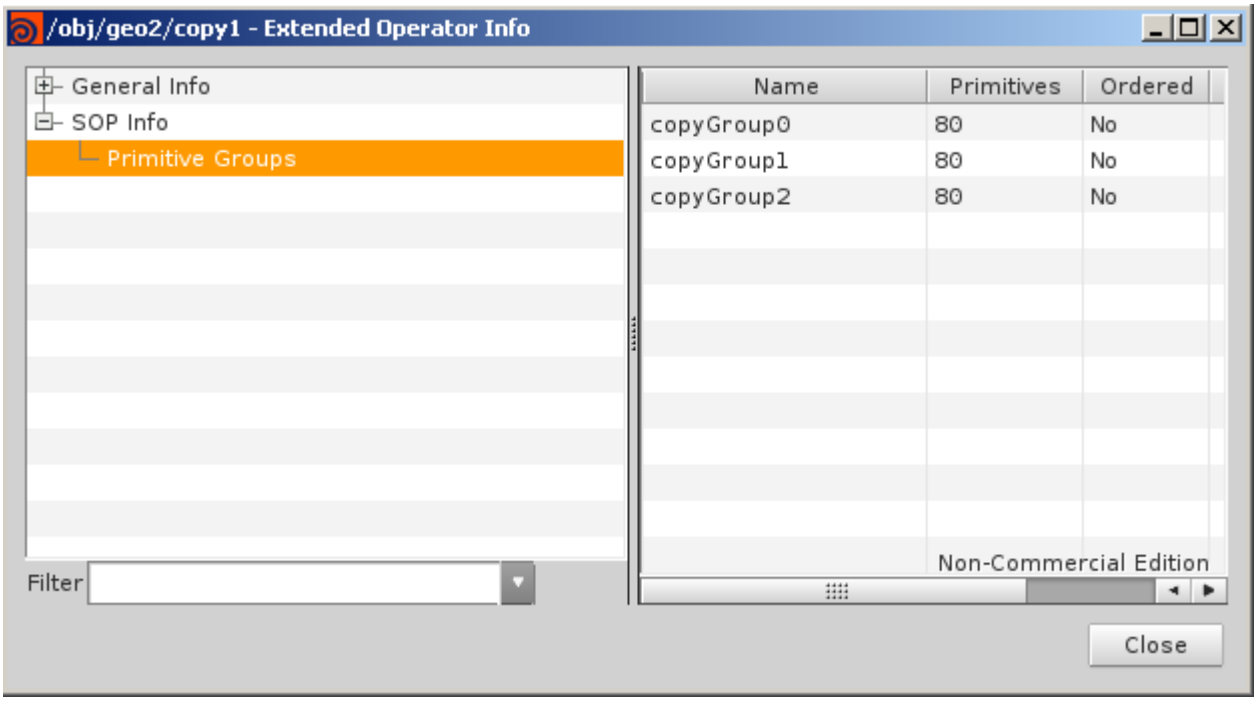

Figure 9: the Extended Information dialogue box for the Copy node shown in figure 8. This shows that three primitive groups have been created, with 80 primitives in each group. So each sphere consists of 80 faces, which have been placed in separate groups called 'copyGroup0', 'copyGroup1', and 'copyGroup2'.

Groups work at the geometry level, so that group Point and Primitive components. This means that the groups exist inside a geometry container such as `/obj/geo1`. However, often it is also useful to group together different geometry containers into larger groups. For example, maybe you have geometry in `/obj/geo1` and `/obj/geo2` grouped together into a single unit? One way of doing this is to create a special type of group called a bundle.

Parameters

Parameters are the options on an individual node, and they control how the node behaves. For example, each geometry container has a set of transform parameters (`Translate, Rotate, Scale,` etc) that controls the position, orientation and size of the geometry in the container.

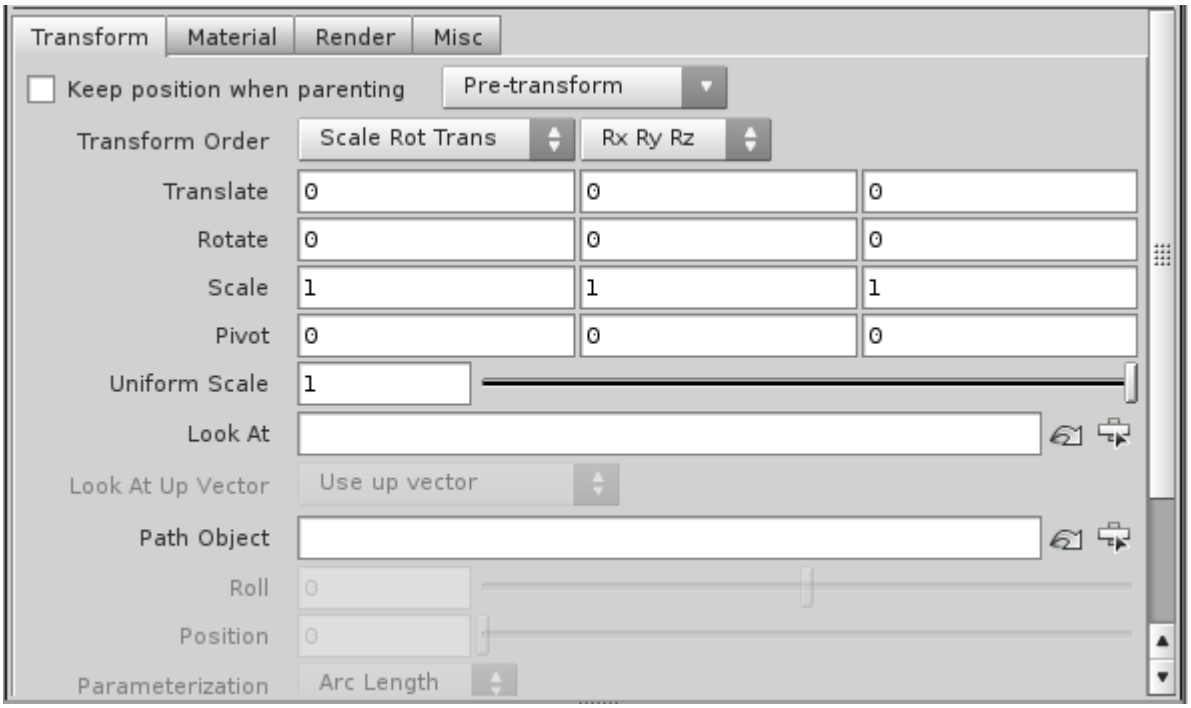

Figure 7: The transform parameters for a geometry container.

Each parameter has a name that is unique in that node. A geometry container node, for instance, has three translate parameters in the x, y and z directions, called `tx`, `ty`, `tz`.

Node parameters can also be accessed using paths. For example, to access the x translate parameter of a geometry container node (see figure 6) you can use the following path:

- `/obj/geo1/tx`

Paths such as these are called parameter paths – the last value is always a parameter name.

When you are setting a value for a parameter, you can just type in the value within the parameter dialogue box. But in many cases, what you actually want to do is to make a parameter dependent on some other entity. For example, maybe one parameter is related to another parameter in some way. In such a case, what you want to do is to write a parameter expression that links to some other node. Parameter expressions are very powerful, and in effect allow other relationships to be made. Parameter expressions will be discussed in more detail later.

Surface operators have parameters specific to each operation. For example, the sphere SOP has parameters like radius, centre, orientation, etc.

Keyframes

Parameters can also be changed over time, so that as you run an animation, the parameter value will change. This is often achieved using a technique called *keyframing*. Just in case you are not sure what a keyframe is, here is an extract from wikipedia:

“A key frame in animation and filmmaking is a drawing that defines the starting and ending points

of any smooth transition. They are called "frames" because their position in time is measured in frames on a strip of film. A sequence of keyframes defines which movement the spectator will see, whereas the position of the keyframes on the film, video or animation defines the timing of the movement...

In computer animation this workflow is basically the same. The animator creates the important frames of a sequence, then the software fills in the gap. For example, the animator can specify, in keyframes, the starting and ending position of an object, such as a circle. The software smoothly translates the object from the starting point to the ending point. This is called tweening. The animator can correct the result at any point, shifting keyframes back and forth to improve the timing and dynamics of a movement, or change an 'in between' into an additional keyframe to further refine the movement."

In Houdini, a parameter can be in one of two states: animated or not animated.

- If a parameter is not animated, then the parameter value will stay constant over time.
- If a parameter is animated, then the parameter value can change over time.

In order to animate a parameter, it must be keyframed. A keyframe contains an expression that is used to calculate the values of the parameter for any frames in-between the keyframes. Thus, the expression in a particular keyframe is active from that keyframe until the next keyframe. If there is no next keyframe, then it is active until the end of the animation.

Many parameters are already keyframed by default, so with these you don't need to do anything special – you can type an expression directly into the parameter input box. However, if the parameter is not keyframed by default, then you will need to keyframe it manually before you can type in an expression.

A special case that is nevertheless quite common, is an animated parameter with just one keyframe at frame 0. This basically means that the expression will be active for all frames. This is very useful, since it allows you to write expressions in parameters, even when you are not doing an animation. For example, you might want the height of one box to be twice the height of another box. This can be achieved by animating the `sizey` parameter of the second box and creating a keyframe at frame 0. The expression for the keyframe can then get the value of the `sizey` parameter for the first box and double it. You may never actually animate this – and if you did slide the time slider, nothing would change since the same expression is always active (unless of course some the first box is changing height due to some other logic). So, when someone says a parameter has an expression, they typically means that it has a single keyframe at time=0, and they're referring that that keyframe's expression.

Note also that animated parameters with keyframes are also referred to as animation channels. When you are working on complex animations, working with and tweaking hundreds of animated parameters becomes a daunting task. Houdini therefore provides a whole set of tools for viewing and manipulating animated parameters, and these are all found under the channels area. For example, you can view the value of the parameter graphed against time in the channel editor's graph view, and then directly edit the graph to change the animation.

Parameter expressions

In Houdini, once a parameter has been keyframed, an expression can be entered into the same input box where you would ordinarily type the value. This is a bit like a spreadsheet program, where you can either type a value, or you can type a formula to calculate the value.

Houdini provides two scripting languages to write parameter expressions – HScript and Python.

Hscript is a very simple scripting language that is easy to learn, whereas Python is a much more powerful scripting language. Python was actually only added to Houdini quite recently – before Python everyone used Houdin. For this document, we will focus on Hscript, for two reasons. First, most tutorials and videos on the Internet actually use Hscript. Second, there are some important functions that are not yet available in Python. (In fact, the first semester that we taught Houdini, we focused on Python instead of Houdini. But in the end we still had to use some Hscript here and there. So this time, we will try and do everything in Hscript.)

For learning Hscript in detail, there is an excellent set of videos that cover almost everything there is to know. But this may be a bit daunting at first, so here we will give a brief overview of the basics.

The simplest parameter expression may be something like this:

```
2/3
```

and the result will be 0.666...

In this case, the data type of these numbers is floating point numbers. The four possible data types are:

- Floats - all numbers including integers, and numbers with decimals (e.g. 2, 2.3, 2.345)
- Strings – any kind of text (e.g. 'abc', “abc”, “123”)
- Vectors – three floating point numbers for things like points (e.g. [1,2,3], [1.2, 2.3, 3.4])
- Matrix – 16 floating point numbers for 3D transformation matricies.

Of course, you need to produce the right type of result for the parameter where the expression is written. If you put this text expression into a parameter that requires a float, then you will get an error, since `abcdefghi` cannot be interpreted as a number. With floats, the usual operators can be used, such as +, -, /, * etc. With strings, you can use +, so for example:

```
'acd' + 'def' + 'ghi'
```

will produce the text string `abcdefghi`. With vectors you can use +, -, /, and *:

```
vector(0,1,1) + vector(2,0,0)
```

will produce a vector (2,1,1).

Global variables

So far, this is not very exciting. So the next thing you can do is access a set of variables that contain values related to some aspect of Houdini. These variable names all start with a `'$'`. For example, some commonly used variables are as follows:

- `$PI` – The mathematical constant pi (3.1415926…).
- `$HIP` – The folder where your Houdini file is saved.
- `$F` – The current frame

Using these global variables, you can then write slightly more interesting expressions. For example, if you write the following expression in the ty parameter (which is the transform-y parameter) of an object:

```
$F/10
```

then, when you play the animation, you will see the geometry of that object rise upwards. So what is happening here is that the height of the geometry is varying according to the current frame – so at frame one, the y value is 1/10, and at frame 2 it is 2/10, at frame 3 it is 3/10, and so forth.

These variables are also referred to as *global* variables, since they are globally accessible form any parameter expressions. There are also some other variables, called *local* variables, that are specific to a particular node. These variables are only accessible from parameter expressions in that node. For example, the local variables for the Copy node in figure 8 has very useful and used a lot:

Functions

Now, going one step further, the next thing you can include in your expressions are functions. Functions make expressions very powerful and very versatile. Houdini provides a whole list of built-in expression functions that you can call from your parameter functions.

Although the list of functions looks daunting, in practice you only tend to use a very small number of them. But they are there if you need them. Here are some common functions:

- ch — Returns the value of a parameter.
- chs — Returns the value of a parameter and returns a string.
- point — Returns the value of a point attribute.
- vertex — Returns the value of a vertex attribute.
- prim — Returns the value of a primitive attribute.
- detail — Returns the value of a detail attribute.
- rand — Returns a pseudo-random number from 0 to 1.
- cross — Returns the cross-product of two vectors.
- if — Returns the value of the second or third argument depending on the truth of the first argument.

A very common and powerful way of using expressions is to calculate the value of a parameter based on either another parameter elsewhere in the network or an attribute value elsewhere in the network. Here is an example of a calculation using another parameter value:

```
ch('../geo2/tx') / 2
```

This uses the `ch` function gets the value of another parameter and divides by two. Here is an exmple of a calculation using a point attribute value:

```
point('../geo2/sphere1', 3, 'P', 0)
```

This uses the `point` function to 1) get the sphere node, 2) find the third point, 3) find the attribute called 'P' (which is the position, defined by a vector of 3 values – x, y, and z), and 4) find the first element in the vector (the x value).

One more function that needs to be mentioned is the `stamp` function. This is a very powerful function that is also very strange, and it takes a while to get used to. It is typically used together with a `Copy` node. The way the `Copy` node works is that you can make copies of the input geometry. However, all copies will be exactly the same. What happens if you actually want each copy to be different in some way? – for example – you may want to scale the geometry so that each copy gets smaller. In order to do this you can use a `stamp` function to scale the input geometry before it actually arrives in the `Copy` node. See copy stamping for more information. There is also an excellent video on stamping.

Multiple lines

So far, all the Hscript parameter expressions that we have looked have consisted of a single line of code. Most of the examples seem to take this approach, and you will often see long lines of code. This is because Hscript has the ability to pack a lot of stuff into just one line of code. However, sometimes you may prefer to split the code up over multiple lines. One reason is just to help with the readibility of the code. Having multiple lines also allows you to add comments to the code.

Here is an example of a small expression on one line:

```
point('../geo2/sphere1', 3, 'P', 0) + vector3(0,0,1)
```

Here is the same expression over multiple lines:

```
# get the point from the sphere and add a vector
{
pt_1 = point('../geo2/sphere1', 3, 'P', 0);
vec = vector3(0,0,1);
pt2 = pt1 + vec;
return pt2;
}
```

The first line is a comment, and has no affect. After that, there are a few things to note about this code:

- The code is enclosed in curly braces – if you have multiple lines, then you have to do this to define a block of code.
- Each line ends with a semicolon – again, you have to do this to tell Houdini where the end of the line is.
- You must use a return statement to return a value. The value that is returned will become the parameter value.

Quoting and backticks

One important aspect about HScript is difference between various ways of using quotes. This actually comes up quite a lot in the examples, and at first it is a little confusing. The simplest is single quotes, 'like this' - this is treated as plain text. In general, everything can be achieved with these single quotes, except that it requires more typing. The other two methods described below are just there as a shorthand to make expressions more compact. Here is is an example using single quotes:

```
'Current frame = '+$F
```

This will result in `$F` being evaluated as e.g. frame 2, resulting in the text string `Current frame = 2`. When using double quotes, “like this”, then you don't need the +. Any variables in the text will be replaced with the value of the variable – which is referred to as *expansion*. So, for example:

```
“Current frame = $F”
```

will result in `$F` being replaced by the frame number, again resulting in the text string `Current frame = 2`.

Lastly, when using backticks, `like this`, then the text will be treated as if it is Hscript, and evaluated. The result of evaluating the expression then replaces everything between the backticks. So, for example:

```
``'Current frame = '+$F`
```

will first result in the text `'Current frame = '+$F` being evaluated, which will then result in `Current frame = 2`. This is only really useful in special cases where the parameter is a text parameter.

Pattern matching

Lets say you want to search for all the files on your computer that have the extension ".hip". In order to do this, you can type "*.hip" in the search box. The string "*.hip" is called a pattern, and it will match and string that end with ".hip", such as "test.hip", "model.hip", etc.

In Houdini, pattern matching is used anywhere an expression function needs a name (such as a parameter or node name) to match multiple things at once. There are two types of patterns:

- String patterns match strings. For example, `[gG]eo*` matches everything beginning with "geo" or "Geo".
- Numeric patterns match sets of numbers. For example, `0-30:2` matches every other number between 0 and 30 (0, 2, 4, 6, … 30).

Multiple patterns can be combined, with each pattern being seperated by a space. (Do not use commas, as this is not allowed.) For example, `0-100:2 ^10-20` combines two patterns: the first pattern matches every other number between 0 and 100, and the second pattern excludes the numbers between 10 and 20. So in the end, the combined pattern matches (0,1,2,3,4,5,6,7,8,9,21,22,23,... 100).

String patterns are used a lot with group names. For example, you might have a set of groups whose names all start with the string "wall_" (e.g. "wall_1", "wall_2", etc). If you want to extrude just these walls and not any of the other geometry that may be present, then you can use the `extrude` node, and in the `Source Group` parameter, enter `wall_*`.

Numeric patterns are used a lot together with nodes that require numeric patterns. For example, the `Group` node can be used to group together primitives given a particular numeric pattern. Entering a pattern such as `5-10` will mean that a new group will be created consisting of six primitives.

SOP Nodes

The key to modelling with Houdini is learning what each type of SOP node can do and how they can be used. Most SOP nodes are very versatile, and can be used in many different ways, so understanding what each SOP node can do is not as easy as it first sounds.

For example, consider the `Extrude` node. As the name suggests, this node can be used for extruding curves and surfaces. However, it can also be used for offsetting curves. If you are a beginner, and are trying to find out how to offset a curve, it make take you a while to discover that it is the `Extrude` node that you need. The best way to learn is to view the tutorial videos and open the example files.

In fact, on any node in a network, you can right click and open the help documentation for that node. This will give and overview of what that node can do. At the bottom of the help file, there will often be a number of example files that you can open. These files are very useful, and give you excellent examples to look at.

To get you started, below, I will highlight some of the more commonly used SOP nodes.

Transforming and copying

There are a bunch of general purpose nodes that are extremely useful and tend to pop up all over the place. These include:

- `Duplicate`: Duplicates any type of geometry one or more times.
- `Copy`: Similar to duplicate, but also has some more advanced features.
- `Sweep`: Can be used to copy any type of geometry along a curve.
- `Transform`: Transforms (translate, rotate, scale, shear) any type of geometry
- `Mirror`: Mirrors any type of geometry in a plane defined by an origin and direction.

The `Copy` node is an advanced version of the `Duplicate` node and is very powerful (but also slower). This is one of those nodes that again pops up everywhere. The node allows you to copy one piece of geometry onto the points in another piece of geometry, which is referred to as *instancing*. When you copy or instance geometry onto points, Houdini looks for specific attributes on the destination points to customize each copy/instance. For example, the normal vector, called the `N` vector, will affect the orientation of the copied geometry.

In addition, as has already been mentioned previously, a special technique called *stamping* allows you set up variables that vary per-copy, and then refer to those variables in the source network using the stamp function. See copy stamping for more information.

Working with attributes

You will often need to work directly with the attribute data the is flowing through a SOP network. There are a set of nodes designed specifically for working with the attributes for points, vertices and primitives. First, there are three nodes to extract points, vertices and primitive attributes:

- `Point`: Gives access to the points attributes.
- `Vertex`: Gives access to the vertex attributes.
- `Primitive`: Gives access to the primitive attributes.

These three nodes allow you to take some action on all the entities. For example, if you want to add a set of colour attributes to each point, then you can wire in a `Point` node, and then in that node specify that each point flowing through this nodes needs to have a colour added. These nodes therefore have a kind of built-in loop, where anything you specify is applied to each entity.

The `Vertex` node tends not to be used so much, but the `Point` and `Primitive` nodes are very useful and you will often be needed. The `Point` node allows you to add and set attributes such as position, normals, and colour. The `Primitive` node allows you to add and set attributes such as colour. In addition it also allows you to do many other things, such as transform individual primitives.

In addition to these three basic nodes, there are also some other nodes that allow you to work with attributes:

- `Attribute`: Renames or delete attributes.
- `AttribCreate`: Adds or edits user defined attributes of float, integer, vector, or string type.
- `AttribCopy`: Copies attributes from one piece of geometry to another piece of geometry with the same topology.
- `AttribTransfer`: Copies attributes from one piece of geometry to the closest points on a different piece of geometry with a possibly different typology.
- Paint: Add color or other attributes on geometry.

Finally, there are a few useful nodes that affect the attributes in particular ways:

- `Delete`: Deletes points or primitives (or groups).
- `Dissolve`: Deletes points and primitives, and repairs any holes left behind.
- `Connectivity`: Creates an attribute with a unique value for each set of connected primitives or points.
- `Measure`: Measures volume, area, and perimeter of polygons and puts the results in attributes.
- `Clean`: Helps clean up dirty models, by deleting unused points, removing degenerate primitives, etc

Unique-ing and consolidating points

There are some nodes to change the relationship between points and vertices. There are two things that you can do, referred to as *consolidating* (i.e. merging or fusing) and *unique-ing* (i.e. spliting), which are basically the opposite of each other. Remember that a vertex is a refrence to a point, and more than one vertex can reference the same point. First, lets consider consolidating. Lets say there are two points, `a` and `b`, referenced by two vertices, `p` and `q`. If the points a and b are the same (or similar), then they can be consolidated by replacing them with a single point, and `p` and `q` will now reference this new point. Now lets consider unique-ing, which is the opposite of consolidating. Lets say there is one point, `a` referenced by two vertices `p` and `q`. Unique-ing means that a duplicate is made of `a`, and the vertex `q` will now refrence this new duplicate point.

The key thing is to remember that when consolidating or unique-ing points, you are also consolidating or unique-ing all the attributes associated with those points. When consolidating, the

attributes from different points are averaged, and when unique-ing, the attributes from one point are duplicated. One important attribute to consider when consolidating or unique-ing are the normals that are associated with points. Some of the nodes have special aprameters for calculating normals.

- `Fuse`: Consolidate, unique, or snap points. When consolidating you can merge points based on 1) a minimum threshold distance between points, or 2) based on the group they are in. When unique-ing, points are duplicated so that there is a one-to-one mapping between points and vertices. When snapping, you can snap points based on 1) a minimum threshold distance between points, or 2) an orthogonal grid.
- `Facet`: Consolidate or unique points or normals. The nodes provides special parameters for dealing with surface normals.
- `VertexSplit`: Uniques points whose vertices differ by more than a specified tolerance for a particular attribute.

Working with groups

In addition to nodes for working with attributes, there are also nodes for working with groups.

- `Group`: Generates groups of points or primitives according to various criteria, including patterns, ranges, and expressions.
- `GroupCopy`: Copies groups from one piece of geometry to another piece of geometry with the same topology.
- `GroupTransfer`: Copies groups from one piece of geometry to the closest points on a different piece of geometry with a possibly different typology.
- `Partition`: Groups points and primitives based on a user-supplied rule (defined by a parameter expression).
- `Clip`: Groups or removes geometry on one side of a plane.

Note that many of the other nodes are also designed to work with groups. For example, if your are using the `Transform` node, you can chose either to transform all the geometry that it receives, or to transform only the geometry within some specified group. All the geometry not in that group will then be left unchanged.

Creating geometry

There are a bunch of nodes for creating basic geometric entities. These nodes also define the type of primitives used to represent the geometric entity. For example, when creating a line, you can choose Polygon, Bezier, NURBS, or just plain Points. When creating a sphere, you can choose Primitive, Polygon, Polygon Mesh, Mesh, NURBS, and Bezier.

The basic geometry nodes include:

- `Line`: Creates a line from a position, direction, and distance.
- `Circle`: Creates open or closed arcs, circles and ellipses from a centre point and radius.
- `Curve`: Creates curves from a sequence of points.
- `Grid`: Creates a plane from a centre point, 2D size, and axis (orientation).
- `Sphere`: Creates a sphere from a centre point and a radius.

- `Tube`: Crates a tube from a centre point, radius, height and axis (orientation).
- `Metaball`: Creates metaballs and metasurfaces useful for modeling blob-like things that mold together or organic shapes.

These nodes are mostly fairly straightforward – they simply generate the geometry according to the parameters specified. Using parameter expressions, these parameters may come for som other part of the network.

Creating geometry from points

There are many ways of creating geometry. Some nodes create geometry directly from points

- `Scatter`: Scatters new points randomly across a surface.
- `Add`: Creates points or polygons given an x,y,z poistion.
- `Fit`: Create a spline or a surface by fitting a spline curve to points, or a spline surface to a mesh of points.
- `PolyLoft`: Creates new polygons using existing points.
- `TriBezier`: Creates a triangular Bezier surface.
- `PolySpline`: Fits a spline curve to a polygon or hull and outputs a polygonal approximation of that spline.
- `Triangulate2D`: Connects points to form well-shaped triangles.

The `add` node can be used in many different ways. In particular, it can be used to both create new points and to grab points within existing polygons.

Sweeping and extruding geometry

Surfaces are often created by using curves. The following nodes create surfaces in this way. The type of surface that is produced depends on the type of input. For example, if you sweep a polyline, you will get a polygon surace, whereas if you sweep a NURBS curve, you will get a NURBS surface.

- `Sweep`: Creates a surface by sweeping cross-sections along a backbone curve.
- `Rails`: Creates a surface by stretching cross-sections between two guide rails.
- `Revolve`: Creates a surface by revolving a curve around a centre axis.
- `Skin`: Creates a surface by building a surface between eny number of edge curves.
- `Bridge`: Like skin, but more advanced.
- `ExtrudeVolume`: Creates a volume by extruding surface geometry.

For poly surfaces:

- `PolyExtrude`: Creates polygon surfaces or mesh surfaces by extruding polygonal faces and edges.
- `Wireframe`: Creates polygonal tubes around polylines, creating renderable geometry.
- `PolyWire`: Creates polygonal tubes around polylines, creating renderable geometry with

smooth bends and intersections.

Joining, filleting and stitching geometry

General purpose nodes for joining, filleting and stitching include:

- `Stitch`:Stretches two curves or surfaces to cover a smooth area.
- `Join`:The Join op connects a sequence of faces or surfaces into a single primitive that inherits their attributes.
- `Fillet`: Creates smooth bridging geometry between two curves or surfaces.
- `Round`: Generates round fillets of a specified radius between two surfaces.

For poly surfaces:

- `PolyStich`: Stitches polygonal surfaces together, attempting to remove cracks.
- `PolyPatch`: Creates a smooth polygonal patch from a mesh primitive or a set of faces (polygons, NURBS or Bezier curves).
- `PolyBevel`: Bevels points and edges. In the simplest case of zero repetition, this node replaces points and edges with faces. With more repetition, it replaces those faces with more faces, leading to smoother bevels.
- `PolyKnit`: Creates new polygons to joining existing polygons.

Cutting and trimming geometry

General purpose nodes for cutting and trimming include:

- `Break`: Breaks the input geometry using the specified cutting shape.
- `Hole`: Makes holes in surfaces. Works with Polygonal and Bezier geometry types only. NURBS surfaces will be converted internally to Beziers.
- `Trim`: Trims away parts of a spline surface defined by a profile curve or untrims previous trims.
- `Carve`: Slices, cuts or extracts points or cross-sections from a primitive.
- `Shatter`: Shatters the input geometry by inducing multiple fracture lines in it.

Modifying geometry

For poly surfaces:

- `Divide`: Divides, smoothes, and triangulates polygons.
- `Subdivide`: Subdivides polygons into smoother, higher-resolution polygons.
- `TriDivide`: Refines triangular meshes using various metrics.
- `Facet`: Controls the smoothness of faceting of a surface. This node also lets you consolidate points or surface normals.
- `Cap`: Closes open areas with flat or rounded coverings,

- `PolyCap`: Fills in polygons between boundary edges.
- `End`: Closes, opens, or clamps end points.
- `PolyReduce`: Reduces the number of polygons while attempting to preserve its shape.

For spline surfaces:

- `Basis`: Provides operations for moving knots within the parametric space of a NURBS curve or surface.
- `Refine`: Increases the number of points/CVs in a curve or surface without changing its shape.
- `Resample`: Resamples one or more curves or surfaces into even length segments.

Intersecting and projecting geometry

For projecting and extracting curves:

- `Project`: Creates profile curves on surfaces.
- `Ray`: Projects one surface onto another.
- `Profile`: Extracts or manipulates profile curves.

For poly surfaces:

- `Cookie`: Combines two polygonal objects with boolean operators, or computes the contour line along the intersection between two polygonal objects.
- `PolySplit`: Divides an existing polygon into multiple new polygons.

For spline surfaces:

- `CurveSect`: Finds the intersections (or points of minimum distance) between two or more curves or faces.
- `SurfSect`: Trims or creates profile curves along the intersection lines between NURBS or bezier surfaces.

Deforming geometry

General deformations can be created using the `Twist` node:

- `Twist`: Applies deformations such as bend, linear taper, shear, squash/stretch, taper, and twist.

For offseting geometry (which will often also deform it):

- `Extrude`: Offsets any type of curve (as well as extruding curves into surfaces, and surfaces into volumes)
- `Peak`: Offests any type of surface (as well as more generally moving primitives, points, edges or breakpoints along their normals)

You can also deform geometry by morphing or blending from one shape into another shape with the same topology.

- `BlendShapes`: Morphs between shapes with the same topology.

- `WireBlend`: Morphs between curve shapes while maintaining curve length.

Control flow

- `Switch`: to be completed
- `ForEach`: to be completed

Some other useful nodes

- `Convert`: Convert geometry from one geometry type to another.

References

The main source of information were the Houdini help files for version 10, which can be found here:

- http://www.sidefx.com/docs/houdini10.0/