Chapter 0

Houdini

0.1 Houdini – an Introduction

Houdini of SideFX® Software is one of the leading computer graphics software. It allows not only for 3D modeling and special effects in visual entertainment industry, but also an accessible tool for geometry processing, visualization, and a platform for experimenting, developing, and commercializing numerical algorithms. Houdini Apprentice (non-commercial) License is free that comes with almost full functionality, with a limitation on the rendering resolution and that there is a watermark on a corner of the rendering output. The non-commercial Houdini files cannot be converted into a commercial file either, to prevent any commercial use. For educational and academic purposes Houdini Apprentice License is enough.

From a practical point of view, one of the great advantages of Houdini as a numerical tool is that Houdini takes care of the geometry processing framework including data structure and basic mesh processing algorithms. Such an environment would usually be tedious to implement by oneself. Another obvious great tool that comes with Houdini is its industry-standard renderer allowing us to generate high-quality, photorealistic screenshots or videos.

The main programming languages in Houdini are VEX and Python. VEX is a high-level, C-like shading language with highly optimized efficiency. You can do most mathematical manipulations and access the half-edge data structure using VEX. However, an important numerical tool that is missing VEX is a numerical linear algebra library. In the case one needs to build large linear system, one can pour the data into a Python block, where one can import library such as “numpy”, “scipy”, and other scientific computing packages.

Instead of a pure coding developing environment, a major working environment in Houdini is “wiring up a network”. A network visually consists of nodes, also called “operators”, and wires connecting the nodes. The code blocks can be written or imported in the nodes of the network,
while the wires represent the dependency or data flow. The program is executed by rendering a downstream node, which would trigger a “lazy evaluation”, i.e. an evaluation of all dependent nodes whose outcome has not been cached.

0.1.1 User Interface

By default, the interface of Houdini consists of a few sub-windows (docks). On the top one has the shelves dock which displays some operations in icons. At the bottom there is a timeline called the play bar. Beneath the shelves dock there are three panes subdividing the majority of the screen; each pane has several tags such as Scene View, Network View (labeled as a path “/obj”), and Parameters (displaying the parameters for an “operator”). We will mainly use

- Scene View – The 3D preview of the geometries in the scene.
- Network View – The panel for editing the network.
- Parameters – Parameters of a selected node.
- Geometry Spreadsheet – Table of variable values on the geometry.
- Render View – Result of the ray-traced render of a given renderer and camera.
- Material Palette – A list of materials with predefined shading properties.

0.1.2 Navigation

To navigate in the Scene View, use spacebar+mouse drag with holding either left click button, middle click or right click button. Holding spacebar makes a temporary view mode even when the mouse function is in a selection or transformation mode. The spacebar+mouse drag also works in the Network View. In case one messes up the view, spacebar+H sends you to a default view, and spacebar+G sends you to a view which targets on a selected object (in Scene View) or node (in Network View).

Each operator (node) in your network has an address such as “/obj/geo1/box1”. There are a few existing directories one can find in the dropdown menu on the top of the Network View pane. We will mainly work in

- /obj – Scene. It can contain geometries (SOP subnetwork), lights, cameras, and dynamics (DOP subnetwork).
- /out – Output. It can contain renderers (ROP nodes).
- /mat – Materials. It can contain materials (after moving them there from the material palette).
0.1. **Houdini – An Introduction**

0.1.3 An Example

Let us start off with an example. In this example we want to visualize the function

\[ f : \mathbb{C} \rightarrow \mathbb{R} \]

\[ f(z) := \text{Re}(z^n) \]

for \( n \in \mathbb{N} \). This is a simple example that involves coding in VEX and creating an integer parameter \( n \). Since there is no complex numbers in the current VEX language, it is a good chance to practice defining our own functions.

First of all, let us do a little bit of math. Write \( z = x + iy = re^{i\theta} \). Then \( f(z) = r^n \cos(n\theta) \) where \( r = \sqrt{x^2 + y^2} \) and \( \theta = \text{atan2}(y, x) \).

What we are going to do next is to create a plane, called “grid”, representing a subset of \( \mathbb{C} \). Then we assign the height of the grid to be the value \( f(z) \) at \( z \in \mathbb{C} \). This resulting grid would be the surface of the function graph of \( f \).

Create a Grid Geometry

Move the cursor on the Network View, in \( \text{obj} \), press [tab] and then type “geometry”. [tab] allows one to add a new node, and by typing “geometry” we search for the Geometry node. Place the Geometry node anywhere, creating a node called geo1.

Observe that the node geo1 has two node flags, green (select) and blue (display), on its sides. One can turn on and off the flags by clicking them.

Double-click geo1 to “dive into” the node. Now we are in the path “/obj/geo1”. One can also press \([\text{U}]) and \([\text{I}]\) to move a level up or to dive into the nodes. In “/obj/geo1” one sees a default node file1. Delete file1 by selecting it and press \([\text{del}]\).

In “/obj/geo1” press [tab] and type “grid”. Observe that the available nodes seen after pressing [tab] are different from that at “/obj”. The nodes available in “/obj/geo1” are geometry nodes, a.k.a. SOP nodes, whereas the nodes available in “/obj” are object nodes. Now, place an SOP \( \Rightarrow \) Grid node. On the Parameters panel we find the parameters for the grid geometry. Change the Size to 2 times 2. Change Rows and Columns to, say, 50 times 50.

Note that a SOP node has several flags: bypass, lock, template/selectable template, and display/render. Some of them require [ctrl]+click to turn on/off.
CHAPTER 0. HOUDINI

Modify the Grid’s Attributes

Go to the Geometry Spreadsheet pane. There, we see a table of variable information of “Node grid1” (shown on top left). To the right of the label “Node grid1” we find four buttons: , , , indicating point, vertex, primitive and detail attributes. Click on , and one sees point attributes P[x], P[y] and P[z] in the spreadsheet. Together P is an attribute of type “vector”. They are the positions of each point of the grid object. Note that Houdini uses the convention that y-axis points “up”. (This can be changed in the Preferences of the software if you insist z-axis should point up.)

Now, we modify the P[y] attributes to deform the grid into a surface. In the Network View place a Point Wrangle node downstream of grid1. That is, wire them up like in Figure 1. In the Parameter panel of pointwrangle1 we find a place to write the following VEX code:

```
// FUNCTIONS
float myFunction(float x, y; int n){
    float theta = atan2(y, x);
    float r = sqrt(x*x+y*y);
    return pow(r, n)*cos(n*theta);
}

// MAIN
int n = 2;
@P.y = myFunction(@P.z, @P.x, n);
```

The Geometry Spreadsheet of “Node pointwrangle1” should show the effect of the VEX code. After switching on the display/render flag of pointwrangle1 node, one should see a saddle surface in the Scene View like Figure 2.

This VEX code is a script that runs on every points in parallel. To access (read/write) a point attribute p, we write “@p”, which is a shorthand of “v@p”. Here “v@” stands for “attribute of vector type”. In the case of position attribute P, the “v” in “v@p” is often omitted. To access the
components of the vector type \( \mathbf{P} \), we write \( v_0 P.x, v_0 P.y, v_0 P.z \) or \( v_0 P[0], v_0 P[1], v_0 P[2] \).

Other frequently used VEX types are given in Table 1.

**Add Parameter**

Here we will add a parameter for controlling \( n \) in \( f(z) = \text{Re}(z^n) \). At the top bar of the Parameters pane of *pointwrangle1*, click the dropdown menu 🌐 and select “Edit Parameter Interface...”. The parameter interface editor consists of three columns: “Create Parameters”, “Existing Parameters” and “Parameter Description”. Drag and drop an “42 Integer” type parameter from Create parameters to Existing Parameters as shown in Figure 3. In the Parameter Description of the newly created integer parameter, change the “Name” and “Label” to desired ones. Let us say Name is \( n \). Click Accept. Now, we have created a new integer type parameter with path “/obj/geo1/pointwrangle1/n”, visible as a slider on the Parameters pane. The value of the parameter can be read from a VEX code by the function \( \text{ch( string path )} \), or \( \text{chi, chf, chv, chs...} \) for reading integer, float, vector, string parameters specifically.

```vex
// FUNCTIONS
float myFunction(float x, y; int n){
    float theta = atan2(y, x);
    float r = sqrt(x*x + y*y);
    return pow(r, n) * cos(n*theta);
}

// MAIN
int n = chi("n"); // "n" is the relative path to "/obj/geo1/pointwrangle1/n"
@P.y = myFunction(@P.z, @P.x, n);
```

By scrolling the parameter controlling \( n \), we see the graph of \( f(z) = \text{Re}(z^n) \) for different values of \( n \). See Figure 4.
## Table 1: Frequently used VEX types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Prefix for attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>i@ attrib_name</td>
<td>Integer numbers.</td>
</tr>
<tr>
<td>float</td>
<td>f@ attrib_name</td>
<td>Single precision floating point numbers.</td>
</tr>
<tr>
<td>vector2</td>
<td>u@ attrib_name</td>
<td>Two floating point values. For ( X \in \text{vector2} ), access the two component by ( X[0], X[1], ) or ( X.x, X.y, X.u, X.v ). One can use ( \text{vector2} ) to store complex numbers.</td>
</tr>
<tr>
<td>vector</td>
<td>v@ attrib_name</td>
<td>Three floating point values. For ( X \in \text{vector} ), access the three component by ( X[0], X[1], X[2], ) or ( X.x, X.y, X.z ). <strong>Houdini</strong> reads attributes ( v@p, v@Cd, v@uv ) respectively for point position ( \in \mathbb{R}^3 ), RGB color ( \in [0,1]^3 ), and 2D or 3D texture space ( \in [0,1]^3 ).</td>
</tr>
<tr>
<td>vector4</td>
<td>p@ attrib_name</td>
<td>Four floating point values, quaternions ( \mathbb{H} ). For ( X \in \text{vector4} ), access the four component by ( X[0], X[1], X[2], X[3] ), or ( X.x, X.y, X.z, X.w ). The last component ( X.w = X[3] ) corresponds to the real part of a quaternion. One can also use ( \text{vector4} ) to represent a homogeneous coordinate for ( \mathbb{R}\mathbb{P}^3 ), or an RGBA color data ( p@Cd ).</td>
</tr>
<tr>
<td>string</td>
<td>s@ attrib_name</td>
<td>Strings.</td>
</tr>
<tr>
<td>int[]</td>
<td>i[]@ attrib_name</td>
<td>Arrays of a type.</td>
</tr>
<tr>
<td>float[]</td>
<td>f[]@ attrib_name</td>
<td></td>
</tr>
<tr>
<td>vector2[]</td>
<td>u[]@ attrib_name</td>
<td></td>
</tr>
<tr>
<td>vector[]</td>
<td>v[]@ attrib_name</td>
<td></td>
</tr>
<tr>
<td>vector4[]</td>
<td>p[]@ attrib_name</td>
<td></td>
</tr>
<tr>
<td>string[]</td>
<td>s[]@ attrib_name</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3: Adding an integer type parameter.
CHAPTER 0. HOUDINI

Create New Attributes

In the above example codes, we have only been modifying the values of the existing attribute \( v@P \). We did create local variables such as \( \text{int } n=\ldots \), but they are accessible only in this block of code. If we want to pass down some data downstream to other nodes, we create new attributes. In the following code, we create two new attributes called \( \text{phase} \) and \( \text{rho} \), defined as \( \text{phase} = n\theta \) where \( z = e^{i\theta} \), and \( \text{rho} = |z| \). In case you wonder, the meaning of \( \text{phase} \) is clear: it is the complex phase of \( z^n \); but the definition for \( \text{rho} \) has no mathematical reason here (it is not the magnitude of \( z^n \)).

```c
// FUNCTIONS
float myFunction(float x, y; int n){
  float theta = atan2(y,x);
  float r = sqrt(x*x+y*y);
  return pow(r,n)*cos(n*theta);
}

// MAIN
int n = chi("n");
@P.y = myFunction(0P.z,0P.x,n);
f@phase = n*atan2(0P.x,0P.z); // Remember, y points up.
      // This is the polar angle of ZX-plane
f@rho = sqrt( @P.z*0P.z + @P.x*0P.x );
```

Now you should observe two new columns appear in the Geometry Spreadsheet. \( \text{phase} \) and
Assign Color

Assigning color is a way to provide informative data visualization. Since pointwrangle1 has been a self-contained code with a single purpose “calculate $f(z)$ and make the surface”, it makes sense to assign color in a different node downstream. Create a new Point Wrangle wired downstream below pointwrangle1. In it (pointwrangle2), we create a new point attribute v@Cd of vector type.

```cpp
#include "math.h" // for using PI
v@Cd = hsvtorgb( f@phase/(2*PI), f@rho, 1.0 );
```

Here hsvtorgb is a function that sends the HSV color space to the RGB color space. Hence, the phase data maps to the hue for the color. And we use rho to tune down the saturation near $z = 0$ where phase is discontinuous.

When a geometry that is displayed, and if Houdini finds an attribute specifically v@Cd (in this case on points), Houdini will paint the point color according to v@Cd (Figure 5).
Figure 6: A ray-traced rendering of the function graph $f(z) = \text{Re}(z^4)$.

**Rendering**

To generate a visually plausible rendering, in addition to a beautiful geometry we need the following basic ingredients.

- Polishing geometry.
- Assigning material to geometry.
- Lighting.
- Camera.

**Polishing Geometry** This is optional. We now have a infinitesimally thin surface without thickness. For an artistic project one often need to give a realistic thickness to the surface. Here is a quick trick for doing so. The PolyExtrude node can displace the surface towards its normal with a desired distance. Use a Merge node, which can take multiple inputs, to merge the result after the PolyExtrude and the geometry before the PolyExtrude to form a solid shell. It may be a good idea to put a Fuse node below the Merge node. See Figure 7.

At this stage, you can right-click the last node, choose “Save → Geometry...”, and send the saved geometry to a 3D printer.
Figure 7: Polishing geometry. The result of `pointwrangle2` is a surface without thickness. The `merge1` node merges a copy of the surface off-set with a distance (`polyextrude1`) and a copy of the original surface, forming a shell with some thickness. Since the two sides of the shell should have different face orientation, we also pass through a `Reverse` node to flip the orientation of one of the sides. If we check “Add Vertex Normal” in `polyextrude1`, we also need a `Normal` to add vertex normal on the other side.
CHAPTER 0. HOUDINI

Assigning Material to Geometry  First of all we need to create a material object. You can use the Network View and navigate to “Material Palette”. This is a shaders land, in which the available nodes are called shader operators. Drop an Plastic node in the region on the right. Now you have created a new material with the path “/mat/plastic”.

Now return to /obj/geo1/. Put a SOP node Material and wire that to the downstream of the last existing node. In its parameter pane, find the field Material and fill in the path /mat/plastic. You can also click the button on the right to browse your materials.

Lighting  Go to “/obj” in the Network View. Drop a node Sky Light. It will actually create two nodes: a parallel light and an environment light. You can also use other light sources. We would often put a floor to cast shadow on (put another Geometry node, and place a large Grid).

Camera  Here is the easiest way to add a camera with a desired view. First, navigate (spacebar + mouse drag/scroll) the 3D view in the Scene View pane to a desired view. On the top right corner of the Scene View, you will find two yellow framed boxes: persp1, no cam. Click on “no cam” and select “New Camera”. By doing so you create a new camera “/obj/cam1”. To adjust the view, click the button “lock camera/light to view” on the right bar of the Scene View pane, and navigate the view.

Render  Go to the Render View pane. On the top bar “Render Control” there is a button “Render”, a path to the output driver “/out/mantra_ipr”, and a path to the camera. The renderer node mantra_ipr is automatically created in “/out”. If you are interested you can go there and tune its parameters. Finally, press the button “Render” to produce the render (Figure 6).

Note that after pressing the Render button, Houdini will keep re-rendering whenever we make any changes even when we are at Scene View. You can turn the rendering off by pressing the “Stop” button.

0.2 Mesh Data Structure in Houdini

In this section, we learn the representation of a geometry in Houdini and the basics of accessing attributes using the VEX language. While there are many types of geometries such as polygons, Bazier surfaces, volumes, etc., we focus only on the geometries that are discrete surfaces (polygonal meshes).
0.2. MESH DATA STRUCTURE IN Houdini

In Houdini, each geometry, the outcome of a SOP node in a Geometry network, is a collection of points, vertices, primitives, with several attributes defined on these fields. In the case of a polygonal mesh (“Polygon” type),

- the primitives correspond to the set of faces (which are polygons) of the mesh.
- the points correspond to the set of points of the mesh that always come with position ($\mathbf{P}$ attribute).
- the vertices represent the set of polygon corners, each of which identifies a vertex of a polygon to a point.

You can think of points as anchors that are placed in $\mathbb{R}^3$, and primitives are set of abstract faces that come into appearance through vertices which tell how the faces “hang” on the anchors.

It is worthwhile to mention that Houdini’s “points” are often called “vertices” in mathematics. They can be attached by multiple faces in a polygonal mesh. In contrast, each Houdini’s “vertex” is attached to a unique point and a unique face. Houdini’s vertex is a polygon corner.

**Example 0.1** (Construct a polygonal mesh from scratch). Create an Attribute Wrangle node in a Geometry network. Set the parameter “Run Over” to “Detail (only once)”. The following VEX code will construct a simple polygonal mesh:

```vex
// Create set of points
int pt0 = addpoint( geoself(), {0,0,0} );
int pt1 = addpoint( geoself(), {1,0,0} );
int pt2 = addpoint( geoself(), {0,1,0} );
int pt3 = addpoint( geoself(), {0,0,1} );
int pt4 = addpoint( geoself(), {1,0,1} );
```
// Create set of primitives
int prim0 = addprim( geoself(), "poly" );
int prim1 = addprim( geoself(), "poly" );
int prim2 = addprim( geoself(), "poly" );

// Create set of vertices
addvertex( geoself(),prim0,pt0);
addvertex( geoself(),prim0,pt2);
addvertex( geoself(),prim0,pt1);
addvertex( geoself(),prim1,pt1);
addvertex( geoself(),prim1,pt2);
addvertex( geoself(),prim1,pt4);
addvertex( geoself(),prim2,pt0);
addvertex( geoself(),prim2,pt1);
addvertex( geoself(),prim2,pt4);
addvertex( geoself(),prim2,pt3);

Here

int addpoint ( int geo, vector position)

modifies the geometry geo (geo ∈ int is a handle usually obtained by the function geoself ()) by creating a point at a desired position, and returns a handle of this newly created point. Similarly,

int addprim ( int geo, string type)

modifies the geometry geo by adding a primitive of a desired type. A polygon is specified with type = "poly", and an open polygon (polygonal curve) is specified with type = "polyline". This newly created primitive (polygon) is not visible before adding vertices. The function

int addvertex ( int geo, int prim, int pt)
adds an extra vertex to the polygon prim and attach it to the point pt.

In the above example, prim0 and prim1 are triangles, and prim2 is a quadrilateral, determined by number of vertices added to it.

0.2.1 Attributes

A polygonal mesh has three major fields: points, primitives and vertices. Formally, they are labeled and form finite sets

\[ M_{pt} = \{pt_0, pt_1, pt_2, \ldots, pt_{n_{pt}-1} \} \]
\[ M_{prim} = \{prim_0, prim_1, \ldots, prim_{n_{prim}-1} \} \]
\[ M_{vtx} = \{vtx_0, vtx_1, vtx_2, \ldots, vtx_{n_{vtx}-1} \} \].

When reading a geometry (when VEX code is in the downstream of an SOP node), the numbers \( n_{pt}, n_{prim}, n_{vtx} \) can be read by

\[ n_{pt} = npoints(0), \quad n_{prim} = nprimitives(0), \quad n_{vtx} = nvertices(0). \]

Here the argument of \( npoints(\text{int input_num}) \) denotes “reading the geometry that comes from the first (0-th) input wired to this node”.

A **point attribute** is a function defined on \( M_{pt} \). For example, the point attribute \( @P \) is a function

\[ P : M_{pt} \rightarrow \text{vector} \cong \mathbb{R}^3. \]

Similarly, a **primitive attribute** is a function defined on \( M_{prim} \), and a **vertex attribute** is a function defined on \( M_{vtx} \). There is a forth type of attributes called **detail attribute**. A detail attribute is a constant for a geometry as a whole.

There are a few basic attributes that come with every geometry.

\[ p : M_{pt} \rightarrow \text{vector} \quad (\text{the position of a point}) \]
\[ ptnum : M_{pt} \rightarrow \text{int} \]
\[ pt_i \rightarrow i \quad (\text{the point index number}) \]
\[ primnum : M_{prim} \rightarrow \text{int} \]
\[ prim_i \rightarrow i \quad (\text{the primitive index number}) \]
\[ vtxnum : M_{vtx} \rightarrow \text{int} \]
\[ vtx_i \rightarrow i \quad (\text{the vertex index number}). \]
0.2.2 Read Attributes

Suppose we are writing VEX code in an Attribute Wrangle that is wired downstream of another node. We can read-access the attributes from the geometry from that upstream node with the VEX functions `point`, `prim`, `vertex` and `detail`.

\[
\begin{align*}
\text{Type} & \quad \text{point} ( \text{int} \ \text{input\_num}, \ \text{string} \ \text{attrib\_name}, \ \text{int} \ \text{pt}) \\
\text{Type} & \quad \text{prim} ( \text{int} \ \text{input\_num}, \ \text{string} \ \text{attrib\_name}, \ \text{int} \ \text{prim}) \\
\text{Type} & \quad \text{vertex} ( \text{int} \ \text{input\_num}, \ \text{string} \ \text{attrib\_name}, \ \text{int} \ \text{vtx}) \\
\text{Type} & \quad \text{detail} ( \text{int} \ \text{input\_num}, \ \text{string} \ \text{attrib\_name})
\end{align*}
\]

where the first slot `input_num` specifies the input (for example the 0-th input) of this Attribute Wrangle node that contains the geometry we want to read the attribute from. For example,

\[
\text{vector} \ P_5 = \text{point}(0, "P", 5);
\]

evaluates the attribute $P$ on $\text{pt}_5 \in M_{\text{pt}}$ of the geometry from the 0-th input.

If the parameter “Run Over” is turned to “Point” of this Attribute Wrangle node, the VEX code is executed for each $\text{pt}_j \in M_{\text{pt}}$ in parallel. In that case, one can directly access the point attribute value $P$ of this point by typing

$\&P$ or $v@P$.

For example, the following two values will be the same:

\[
v@P \quad \text{and} \quad \text{point}(0, "P", i@\text{pt\_num})
\]

provided that $\&P$ has not been modified from the 0-th input.

0.2.3 Create Attributes

To create a new attribute, say a point attribute named $x$ of type float, set “Run Over” to “Point” and simply type

\[
f@x = 0.0;
\]

In Houdini 14.0 or later versions, you can also add a point attribute by the VEX function call

\[
\text{addpointattrib} ( \text{geoself}(), "x", 0.0);
\]

which modifies the current geometry $\text{geoself}()$ by adding an attribute $x$ with default value $0.0$ telling that $x$ is of type float. Similarly, `addprimattrib`, `addvertexattrib` and `adddetailattrib` adds primitive/vertex/detail attributes. Adding attributes using these VEX functions does not require “Run Over” to be points, primitives, vertices or detail.
Note: For some reason you cannot create a point attribute and a vertex attribute with a common name. In principle you can create a point attribute and a primitive attribute with the same name. A primitive attribute named “p” is in fact predefined as the center of the polygon.

0.2.4 Set Attributes

To set-access a point attribute named \( x \) at pt\(_5\) to a different value, say 2.0, write

\[
\text{setpointattrib}( \text{geoself()}, \text{"x"}, 5, 2.0 );
\]

Here

\[
\text{setpointattrib}( \text{int geo_handle, string attrib_name, int pt,}
\text{Type value, string mode ="set" )}
\]

replaces the value of attrib_name on pt\(_pt\) by value. The optional slot mode can also be "add", "min", "max" that is handy for cumulating values.

Set-accessing primitive attribute is similar by using \text{setprimattrib}.

For vertex attribute it is slightly different. Suppose \( f : M_{vtx} \rightarrow \text{Type} \) is a vertex attribute. Then, to set the 5-th vertex attribute value, one has to write

\[
\text{setvertexattrib}( \text{geoself()}, \text{"f"}, 5, -1, \text{value})
\]

where \(-1\) is to “turn off” the extra int slot. As will be explained in the next subsection, an index can be specified also by a pair of integer \((\text{prim}, \text{ind})\). This is what this extra int slot is reserved for.

Finally, setting detail attributes is straightforward:

\[
\text{setdetailattrib}( \text{int geo_handle, string attrib_name,}
\text{Type value, string mode ="set" )}
\]

0.2.5 Vertex Topology

As explained earlier, a \((\text{Houdini’s})\) vertex, as a polygon corner, is uniquely associated to a primitive and a point. To read the primitive to which the vertex belongs, use the VEX function

\[
\text{int vertexexprim( int input_num, int vtx).}
\]

For example, in a \(\text{Vertex Wrangle}\) (an \text{Attribute Wrangle} with “Run Over” being “Vertices”) wired (to the 0-th input) downstream to some existing geometry, \text{vertexexprim( 0, i@vtxnum )} gives the primitive index number associated to this vertex.
Similarly, to read the point to which the vertex is attached,

\[
\text{int vertexpoint (int input_num, int vtx)}.
\]

While \text{vertexprim} gives the primitive the vertex belongs to, the VEX function

\[
\text{int vertexprimindex (int input_num, int vtx)}
\]

gives the index of polygon corner of the primitive it belongs to, which is an integer in \{0, 1, \ldots, m-1\} when the polygon is an \(m\)-gon. One can use \text{primvertexcount (int input_num, int prim)} to get this “\(m\)” for an \(m\)-gon.

Conversely, given a primitive \(\text{prim}\) and an index \(\text{ind} \in \{0, 1, \ldots, \text{primvertexcount (0,prim)}-1\}\),

\[
\text{int vertexindex (int input_num, int prim, int ind)}
\]

gives the vertex index number.

In summary, vertices are polygon corners. Suppose \(\text{prim}_j \in M_{\text{prim}}\) is an \(m_j\)-gon, \(i.e., m_j = \text{primvertexcount (0,j)}\). Then the set of all corners is naturally given by the (disjoint) union

\[
\bigcup_{\text{prim}_j \in M_{\text{prim}}} \{ (\text{prim}_j,k) \mid k = 0, 1, \ldots, m_j - 1 \}.
\]

The 1-to-1 correspondence

\[
M_{\text{vtx}} \cong \bigcup_{\text{prim}_j \in M_{\text{prim}}} \{ (\text{prim}_j,k) \mid k = 0, 1, \ldots, m_j - 1 \}
\]

is given by the map (\text{vertexprim}, \text{vertexprimindex}) from left to right, and the map \text{vertexindex} from right to left.

0.2.6 Half-Edge Data Structure

Half-edge is a popular data structure in geometry processing, and it is available in \text{Houdini}. It uses only very few operations but allows one to easily visit neighboring points, faces, and half-edges in a polygonal surface.

As shown in Figure 9, each (interior) edge of a polygonal mesh are split into two directed \text{half-edges} (with opposite orientation), each of which belongs to a polygon. Boundary edges consist of only one half-edge. In \text{Houdini}'s convention, the half-edges within a polygon wind clockwise (when the normal points outward toward us).
0.2. MESH DATA STRUCTURE IN Houdini

Figure 9: Half-edges are directed and uniquely associated to a polygon.

Half-edges are Houdini vertices

Let us denote the set of all half-edges by $M_{he}$. Though there is no “half-edge attribute” in Houdini, one observes that $M_{he} \cong M_{vtx}$, thus “half-edge attributes” can be stored as vertex attributes. The correspondence $M_{he} \cong M_{vtx}$ is that a half-edge corresponds to its source vertex (a polygon corner). The VEX functions giving the bijection are

\[
\begin{align*}
\text{int } &he = \text{vertexhedge}(\text{int } \text{input\textunderscore num, int } \text{vtx}) \\
\text{int } &vtx = \text{hedge\_srcvertex}(\text{int } \text{input\textunderscore num, int } \text{he})
\end{align*}
\]

\[
M_{he} \xrightarrow{\text{vertexhedge}} M_{vtx}
\]

\[
M_{vtx} \xrightarrow{\text{hedge\_srcvertex}} M_{he}
\]

One can check that they are inverse of each other ($hedge\_srcvertex \circ vertexhedge = \text{id}$, $vertexhedge \circ hedge\_srcvertex = \text{id}$).

In most cases the integers $he$ (index for the half-edge) and $vtx$ (vertex number) are the same. That is, if you speculate the vertex attribute created as $i@\text{diff} = i@\text{vtxnum} - \text{vertexhedge}(0, i@\text{vtxnum})$; you would probably see its values be all zero. But one should distinguish the two in indexing, just in case.

Half-edges and points

Given a half-edge $he$ as a directed edge, you can get its source point and destination point by

\[
\begin{align*}
\text{int } &pt = \text{hedge\_srcpoint}(\text{int } \text{input\textunderscore num, int } \text{he}) \\
\text{int } &pt = \text{hedge\_dstpoint}(\text{int } \text{input\textunderscore num, int } \text{he})
\end{align*}
\]
You can see that \( \text{hedge}_\text{srcpoint} = \text{vertexpoint} \circ \text{hedge}_\text{srcvertex} \).

Note that there is no well-defined inverse function of \( \text{hedge}_\text{srcpoint} \), because given a point \( pt \) there are many half-edges that source from \( pt \). The function

\[
\text{int } \text{he} = \text{pointhedge} (\text{int input_num, int pt})
\]

returns one of the half-edges that has \( pt \) as the source point. That is, \( \text{hedge}_\text{srcpoint} \circ \text{pointhedge} = \text{id} \) but not the other way around. (That is, \( \text{pointhedge} \) is a right-inverse of \( \text{hedge}_\text{srcpoint} \).)

Half-edges and faces

Similarly, given a half-edge \( \text{he} \) you can get which primitive it belongs to by

\[
\text{int } \text{prim} = \text{hedge}_\text{prim} (\text{int input_num, int he}).
\]

One can check that \( \text{hedge}_\text{prim} = \text{vertexprim} \circ \text{hedge}_\text{srcvertex} \). The function

\[
\text{int } \text{he} = \text{primhedge} (\text{int input_num, int prim})
\]

returns one of the half-edges in the primitive \( \text{prim} \). That is, \( \text{primhedge} \) is a right-inverse of \( \text{hedge}_\text{prim} \).

Visiting neighboring half-edges

![Figure 10: \text{hedge}_\text{next} and \text{hedge}_\text{nextequiv} functions find neighboring half-edges quickly.](image)

The basic functions for traveling among half-edges are

\[
\text{int } \text{he}_\text{next} = \text{hedge}_\text{next} (\text{int input_num, int he})
\]

\[
\text{int } \text{he}_\text{prev} = \text{hedge}_\text{prev} (\text{int input_num, int he})
\]
The function \texttt{hedge\_next} returns the next half-edge in the same primitive (in the order suggested by the direction of the half-edge). The function \texttt{hedge\_prev} is the inverse function of \texttt{hedge\_next}. The function \texttt{hedge\_nextequiv} is often called “flip” in other implementation. It returns the other half-edge that shares the same edge. See Figure 10.

In a “non-manifold” case, in particular when there are 3 or more faces sharing a common edge, repeated \texttt{hedge\_nextequiv} folds allow you to cycle through these primitives.

\begin{example}[Visit one-ring neighbor of a point] In many cases we would like to iterate through all faces that are incident to of a given point. (For example taking average/maximum/sum of some quantity on the faces around a point.)

\begin{verbatim}
int pt = i@ptnum; // The point around which we iterate.
// Suppose this is a Point Wrangle so @ptnum make sense.
int he0 = pointhedge( 0, pt ); // An initial half-edge.
int he = he0; // iterating half-edge

while( he!=he0 ){
    do{/* do things, for instance access the face
        int prim = hedge\_prim( 0, he );
    */
        he = hedge\_next( 0,
            hedge\_nextequiv( 0, he ));
    }
}
\end{verbatim}

\end{example}

Be sure that there is no boundary in the mesh.

\begin{example}[Area normal] Given a triangular mesh, the area normal on each face prim is \((AN)_{prim}\) where \(A_{prim}\) is the area of prim and \(N_{prim}\) is the normal of prim. Let \(Q \in \mathbb{R}^3\) be any point. Let \(P : M_{pt} \to \mathbb{R}^3\) be the point positions. Then

\[
(AN)_{prim} = \sum_{he<prim} -\frac{1}{2} (P_{src(he)} - Q) \times (P_{dst(he)} - Q).
\]

You can check that this expression is independent of the choice of \(Q\), and when \(Q\) is the position of one of the triangle vertex, the formula indeed gives the area normal. For numerical stability \(Q\) should be chosen not too far away from the triangle. The minus sign in front of \(\frac{1}{2}\) is due to the clockwise orientation the half edges wind a primitive. The advantage of this formula is that it works for all polygons rather than just triangles.

The following VEX code runs over primitives and creates this area normal.

\begin{verbatim}
\end{verbatim}
vector AN = (0,0,0);
vector Q = point( 0, "P", primpoint(0,i@primnum,0));
int he0 = primhedge( 0, i@primnum );
int he = he0;
do{
  vector P1 = point( 0, "P", hedge_srcpoint(0,he) )-Q;
  vector P2 = point( 0, "P", hedge_dstpoint(0,he) )-Q;
  AN += -cross(P1,P2); // hedge in a prim winds clockwise
  he = hedge_next ( 0, he );
}while(he!=he0);
v@AN = AN;
v@N = normalize(AN);

Example 0.4 (Point Normal). While normal vectors of triangle faces in a triangular mesh is well defined (since each triangle lies a plane), there are different ways to define normal vectors on points. One way of coming up with a point normal is by averaging the normals of the neighboring face normals weighted by the face area:

\[
(AN)_{pt} := \sum_{prim>pt} \frac{1}{3}(AN)_{prim}
\]

\[
N_j := \text{normalize} \left((AN)_{pt}\right).
\]

With \((AN)_{prim}\) on faces computed as in Example 0.3, we can assign the point normal \(N_{pt}\) on points by aggregating \((AN)_{prim}\) on its one-ring neighborhood: Run Over Points

v@AN = (0,0,0);

int he0 = pointhedge(0,@ptnum);
int he = he0;
do{
  vector primAN= prim(0,"AN",hedge_prim(0,he));
  v@AN += primAN/3.;
  he = hedge_next (0,hedge_nextequiv(0,he));
}while(he!=he0);

v@N = normalize(v@AN);

There is an alternative way instead of running over all points and visiting the one-ring neighborhood of the point. One can visit all faces and cumulatively distribute its \(A_jN_j\) to its vertex points. In this example you will create three Wrangle nodes in a row. First of all, create a point attribute \text{pointAN} initialized to zero: Run Over Points

v@pointAN = {0,0,0};
Then, in downstream, Run Over Primitives

```c
int he0 = primhedge(0,i@primnum);
int he = he0;

do{
    int pt = hedge_srcpoint(0,he);
    setpointattrib(geoself(),
                   "pointAN", pt, v@AN, "add" );

    he = hedge_next(0,he);
} while(he!=he0);
```

Finally, Run Over Points

```c
v@N = normalize(v@pointAN);
```