COLOUR DRIVEN SHATTERING (TAKEN FROM LECTURE – DYNAMICS #1)

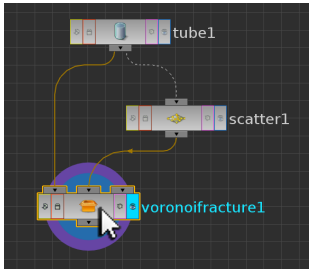
In a **new Houdini scene**, create a **Tube Object**, and inside at its Geometry Level specify in the **parameters** for the **Tube SOP**:

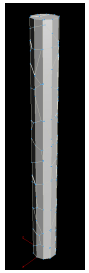
Primitive Type	**Polygon**		
☑	**End Caps**		
Center	**0**	**ch("height")/2**	**0**
Height	**20**		

Append a **Scatter SOP** to the Tube SOP, and specify in its **parameters**:

Force Total Count	**40**

As a new node, create a **Voronoi Fracture SOP**, and wire the **output** of the **Tube SOP** as the **first input**, and the **output** of the **Scatter SOP** as the **second input**. This will create fracture chunks across the tube.



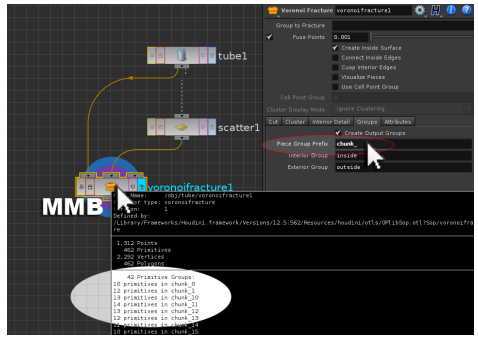


In the **parameters** of the **Voronoi Fracture SOP** specify:

Groups >

Piece Group Prefix	**chunk_**

This will assign a group called chunk_ to each individual shattered component of the Voronoi fractured tube. This can be verified by **MMB** on the **Voronoi Fracture SOP**.



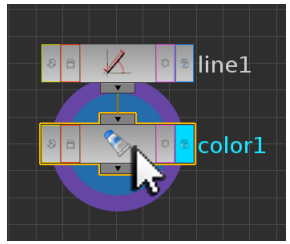
In a new part of the **Network Editor**, create a **Line SOP**. In its **parameters** specify:

Points	**50**

Keyframe (**Alt + LMB**) the **Length parameter** so that on **Frame 1**, the Distance parameter has a **value of 0**; and on **Frame 50** the Length parameter has a **value of 20**. If necessary **Scope** this **parameter** to ensure the animation curve is type **Bezier**. This will create an animated growth to the Line SOP.

To the output of the **Line SOP** append a **Color SOP**. In its **parameters** specify:

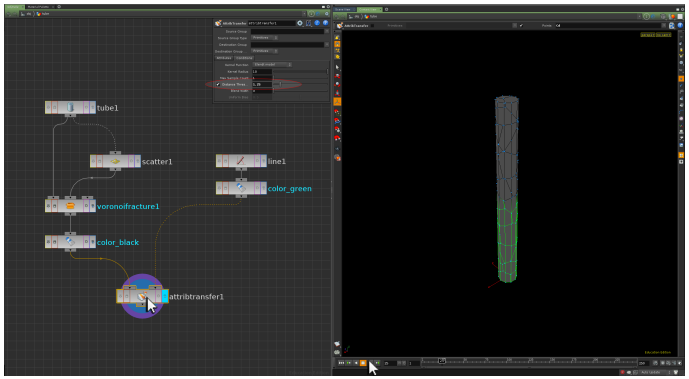
Color	**0**	**1**	**0**



This will colour the animated line green, which in turn can be used to drive dynamic animation of the fracture chunks.

Append a **second Color SOP** to the **output** of the **Voronoi Fracture SOP**, this time setting its colour to **black**. Setting the **Viewer** to **Hidden Line Ghost** will allow the fractured tube geometry to still be seen.

Append to this **second Color SOP** an **Attribute Transfer SOP**, and wire the output of the **green Color SOP** as its **second input**.



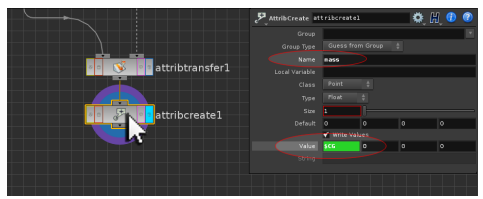
In the **parameters** of the **Attribute Transfer SOP** specify:

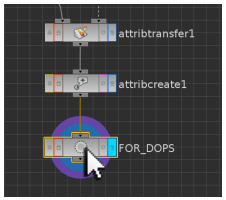
Attributes				
☐	**Primitives**			
☑	**Points**	**Cd**		
Conditions				
☑	**Distance Threshold**	**1.25**		

This will transfer the animated green line colour onto the fractured tube geometry.

Append to the **Attribute Transfer SOP**, an **Attribute Create SOP**. In its **parameters** specify:

Name	**mass**			
Value	**$CG**	**0**	**0**	**0**

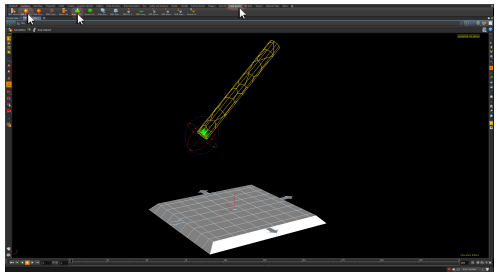


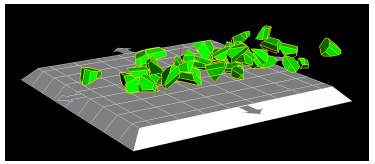


This will store the animated colour as an attribute that can be called in DOPs to drive the breaking of the fractured tube. As a final step, append a **Null SOP** to the network chain, and **rename** it to **FOR_DOPS**.

See file green_tube_Stage1.hipnc

At **Object Level**, **maximise** the **Viewer**, and activate the **Rigid Bodies Shelf**. From the **Rigid Bodies Shelf LMB** the **Ground Plane button**. This will initialise a Dynamics Network for the simulation.

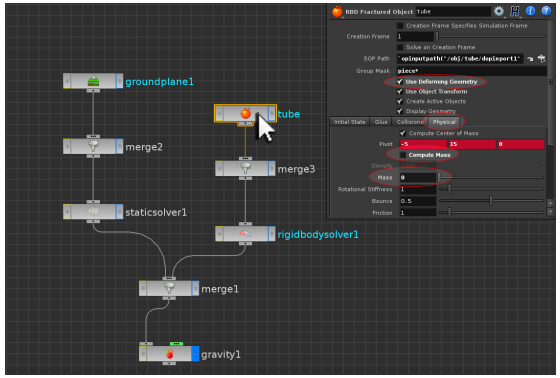




Move the **Tube Object** so that it sits angled above the **Ground Plane**, and from the **Rigid Bodies Shelf LMB** the **RBD Fractured Object button**. When prompted by Houdini for a type of fractured object, select the **RBD Fractured Object button**. This will import the Tube Object into the Dynamics simulation in a format allowing for greater customisation than the default Packed Object option. When **PLAY** is pressed, the tube lands on the Ground Plane and breaks apart on impact.

USING COLOUR TO DRIVE THE SIMULATION

Currently the tube falls uniformly to the ground plane and breaks apart under the influence of gravity. The animated colour of the tube can however be used to drive the breaking of the chunks and their response to forces. At Object Level, go inside the **AutoDopNetwork** node, and locate the **RBD Fractured Object DOP** reading in the tube geometry.



In the **parameters** for the RBD Fractured Object DOP specify:

☐ **Fracture By Name**

Group Mask **chunk***

☑ **Use Deforming Geometry**

Physical >

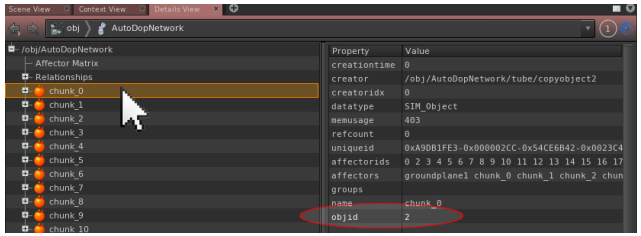
☐ **Compute Mass**

Mass **0**

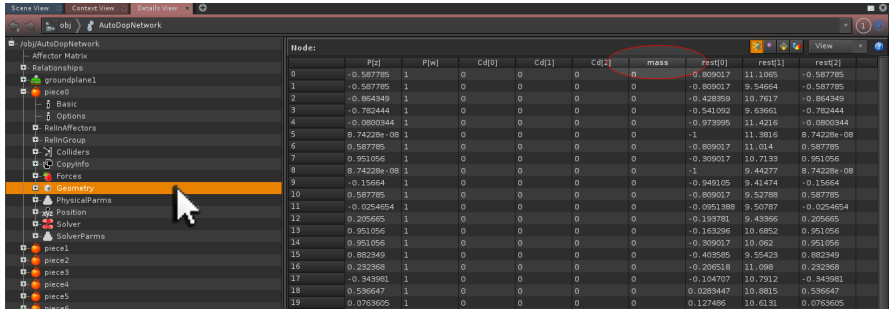
Activating the Use Deforming Geometry option will allow the animated colour information to become part of the DOPS calculation. Deactivating the Mass of the tube will stop the tube responding to DOP forces. Deactivating Fracture By Name will force the RBD Fractured Object DOP to read in each fracture chunk using the groups created by the Voronoi Fracture SOP. When **PLAY** is pressed, the tube now remains at its original location.

THE AUTODOPNETWORK DETAILS VIEW

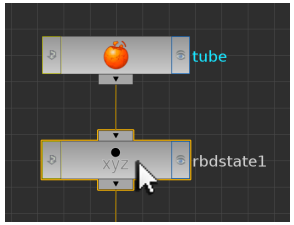
Over the **Viewer Pane**, activate a **Geometry Spreasheet**, and set its **Pane Number** to **1** so that it returns information about the AutoDopNetwork.



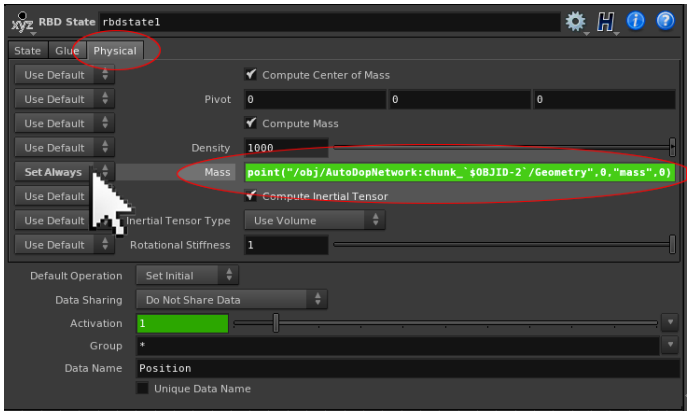
Each chunk of the tube is represented by a **$OBJID** number. The $OBJID number for **chunk_0** is **2**, **chunk_1** is **3**, **chunk_2** is **4** etc. The $OBJID number for each chunk piece of the tube can be used to animate each chunk and their responses to forces individually.



Each piece also has internal listings for all the attributes of the dynamic object. The **Geometry** listing for each piece in the **Details View** reveals the animated mass attribute created before the tube was imported into DOPs. When **PLAY** is pressed, this mass attribute for each piece updates, as each chunk turns green (due to the **Use Deforming Geometry** option activated on the **RBD Fractured Object DOP**). This animated mass information can be retrieved on a per chunk basis so that as each chunk turns green, it responds to dynamic forces.



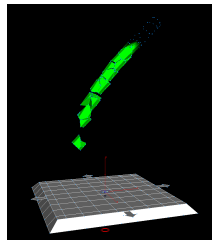
Append to the **RBD Fractured Object DOP**, a **RBD State DOP**. This operator can be used to animate and control RBD behaviours of the tube geometry over time (rather than simply their Initial State behaviours created by the RBD Fractured Object DOP).



Under the **Physical** section of the **parameters** for the RBD State DOP, activate the **Mass** parameter as **Set Always** and specify:

Mass **point("/obj/AutoDopNetwork:chunk_`$OBJID-2`/Geometry",0,"mass",0)**

This will activate the animated mass on each chunk individually, so as they turn green, they respond to dynamic forces. When **PLAY** is pressed, each chunk falls to the ground plane, as it turns green.



The animated mass value can also be utilised to drive other parameters on the RBD State DOP. Under the state section of the parameters, specify:

Set Always
Angular Velocity **if(ch("mass")>0,rand($OBJID)*500,0)** **0** **0**

When **PLAY** is pressed, each chunk now has rotational velocity assigned to it creating a more tumbling momentum when each chunk breaks off. When the chunks hit the ground however, they continue to roll around because of this constant application of Angular Velocity being generated by the RBD State DOP. Deactivating the RBD State DOP after the tube has turned green can rectify this.

In the **parameters** for the **RBD State DOP**, **RMB** on the **Activation parameter** and choose **Delete Channels** from the resulting menu. In its place specify:

Activation **if($SF>=50,0,1)**

This will deactivate the RBD State DOP after 50 frames. When **PLAY** is pressed, the chunks tumble rotate as before; however no come to a standstill after they land on the ground plane.

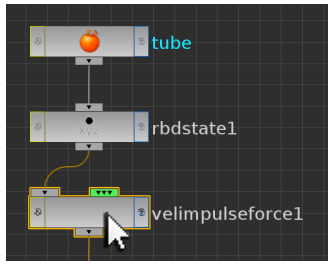
NOTE: $SF (Simulation Frame) is being used rather than $F due to DOPs being a contained environment separate from the main scene.

NOTE: Deactivation of the RBD State DOP after 50 frames does not affect the animated mass of the tube chunks.

ADDING VELOCITY

As a final step velocity can be added to the chunks to move them in a specific direction. If this is done in the RBD State DOP, again constant velocity would be applied to the chunks and turn off after 50 frames. This visually would result in all the chunks appearing to come under the influence of gravity at the same time. A more subtle application of velocity to the chunks can be achieved by using a **Velocity Impulse Force DOP**.

To the **RBD State DOP append** a **Velocity Impulse Force DOP**.



In its **parameters** specify:

Velocity Change	**0**	**rand($OBJID)/2**	**rand($OBJID*3)/3**
Activation	**if($SF>=50,0,1)**		

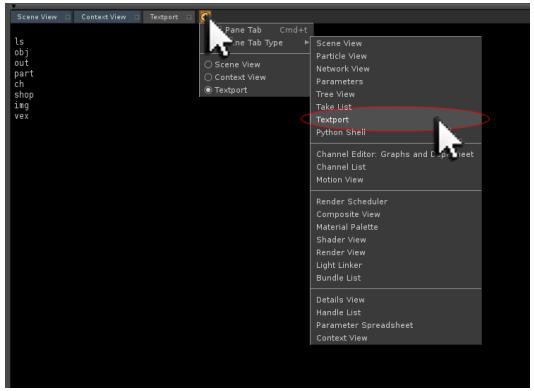
When **PLAY** is pressed, each chunk now is propelled slightly in the Y & Z Axis; however falls more naturally under the influence of gravity and comes to a rest when all chunk forces have been exhausted.

See file green_tube_end.hipnc

HSCRIPTING (TAKEN FROM LECTURE - HSCRIPTING #1)

H-Scripting is the **native scripting language** in Houdini. It can automate scene setups rather than the generation of networks by hand. It is a relatively simple language to learn due to its **similarities** to **Shell Scripting**. It is also very **Artist friendly**. H-Scripting is now complemented in Houdini by Python, which can perform similar automated setup tasks. While Python has an added benefit of being able to create interactive tools, it is however more verbose than H-Scripting and therefore trickier to learn. H-Scripting offers a good way to step into the world of scripting and can in turn open up the Python language due to the principles of scripting and programming being universal and language independent. It is also possible to create combination scripts which utilise both HScript and Python.

In a **new Houdini scene**, activate a **Textport** over the **Viewer** and type the **Shell Script** command **ls**. A list of all the Houdini Levels will be returned.



A **Textport** is set to **Houdini's Root Level** (/). In order to navigate through the Houdini Levels, the Shell Script command **cd** can be used. For example entering the command **cd obj** will navigate to the Object Level of Houdini. Typing **cd ..** will return the Texport location back up to its parent directory.

Default Shell Script commands can be entered into the Textport. They are however simply aliases of their equivalent HScript command.

opls HScript equivalent of **ls** (the Shell command to list all objects)

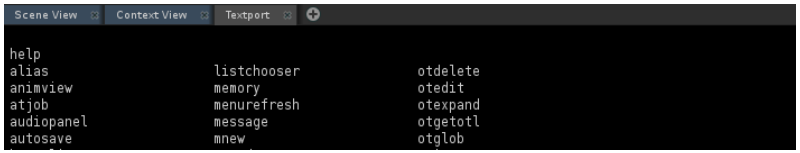
opcf HScript equivalent of **cd** (the Shell command to change directory)

oprm HScript equivalent of **rm** (the Shell command to remove an object)

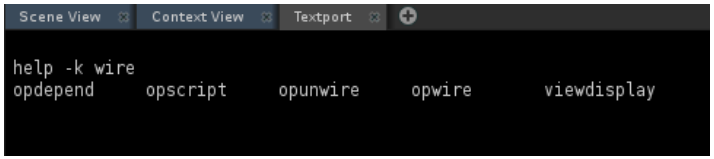
From this point on, the **HScript commands** will be used instead of their Shell Script command aliases to help promote understanding of the H-Script Language.

H-SCRIPT HELP

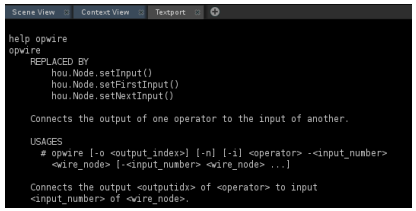
The command **clear** can be used to empty the Textport of earlier commands and returns. Typing the command **help** into the Textport will return a complete list of H-Script commands.



The H-Script help command also has a keyword search function that parallels the H-Expression language help command (exhelp –k). For example typing **help –k wire** will return all of the HScript commands that have the word wire as part of their internal helpcard.



The Help Card for an individual command can be seen by typing the command **help <command name>**. For example, the Help Card for the opwire command can be seen by typing **help opwire**.



CREATING OPERATORS

Type the command **clear** to reset the Textport, and type the following commands:

opcf /obj

opadd geo banana

THE OPADD COMMAND

The **opadd** command also has the **option** to **bypass** the **default operator creation scripts**. In the default creation of a Geometry Object, a File SOP is automatically placed inside it. If the command **opadd –n geo apple** were run, an empty geometry object called apple would be created with no File SOP inside it.

The **–n option** can also have **negative effects** depending upon context. Creating a geometry object with the –n option will currently have an adverse affect on the **Render Tab**, where **default parameters** will be **missing**. Similarly creating a camera using the command **opadd –n cam mycam** will create an empty Camera Object called mycam with some of its parameter tabs missing, and no preview geometry. The command **opadd –n hlight mylight** will create an empty Light Object called mylight with no preview geometry. **Approach the opadd –n option with caution and test it thoroughly before implementing it in a production script.**

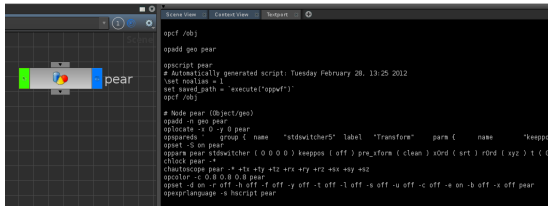
HSCRIPT & WILDCARDS

Wildcards are text characters that denote a special function. They can be very powerful and should be practiced before applying them to anything significant. The most well known wildcard is *, this simply means all. With the mouse over the Textport, type **oprm *.** This will remove all operators from the current Textport level of Houdini.

ECHOING HSCRIPT COMMANDS

When a new operator is created, a H-Script is called to set up its initial state. This information can be retrieved using the **opscript** command. With the mouse over the Textport type:

opcf /obj
opadd geo pear
opscript pear



The opscript command will return the entire set of HScript commands used to create this operator. This is a useful way to decipher a node's creation for the purposes of generating other HScripts.

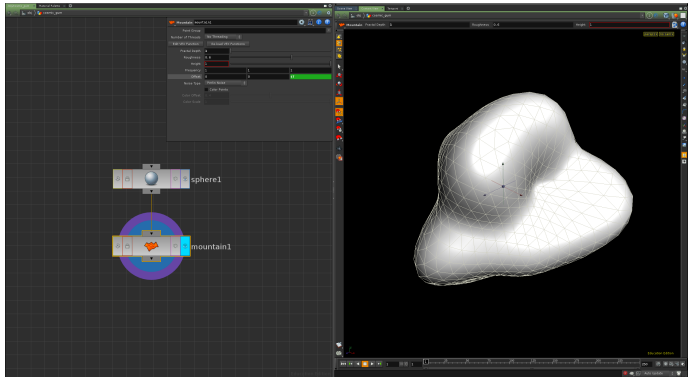
CREATING A SIMPLE SCENE BY HAND AND THEN BY HSCRIPT

A good way to start understanding H-Scripting (or indeed Python Scripting) is to build a network setup by hand, and then create a script to automatically achieve the same results. In a **new Houdini scene**, create a **Geometry OBJ** called **cosmic_gum**. At **Geometry Level** create a **Sphere SOP** with its **Parameters** set to:

Primitive Type	**Polygon**
Frequency	**10**

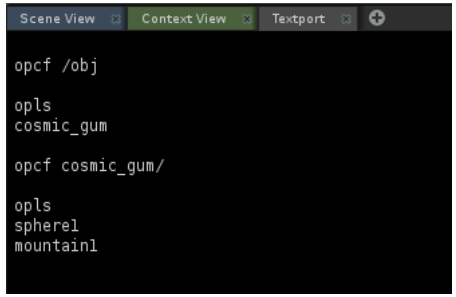
RMB on the output of the **Sphere SOP** and create a **Mountain SOP**. In the **Parameters** for the Mountain SOP specify:

Fractal Depth	**1**		
Offset	**0**	**0**	**$T**

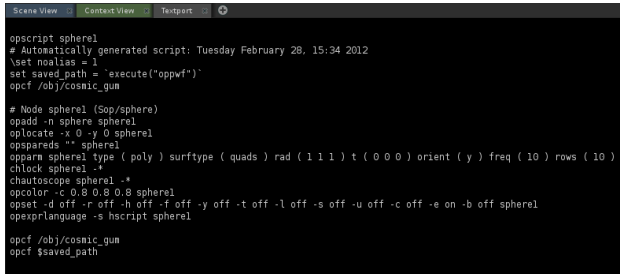


This will create an animated blob.

Reactivate the **Textport** over the Viewer, and use the **clear** command to remove any previous Textport entries. Using the **opcf** and **opls** commands, change into the cosmic_gum directory and confirm that the nodes can be seen.



Using the **opscript** command, determine what HScript commands are used to construct the Sphere SOP.



From this return the following smaller script can be derived:

opcf /obj/
opadd geo cosmic_gum
opcf cosmic_gum
opadd sphere my_sphere
opparm my_sphere type (poly) freq (10)

NOTE: Any Parameters not specified upon creation will use their default values instead.

CREATING AN EXTERNAL H-SCRIPT

While these commands can be entered into the Textport directly, it can be simpler to access them from a text file external to the Houdini environment; or to integrate a script into a Shelf Tool. This example will look at generating an external script to recreate the cosmic_gum.

Launch a **text editor** and type the following:

#THIS SCRIPT GENERATES COSMIC GUM

#switch to OBJ Level
opcf /obj

#remove any previous versions of the cosmic_gum
oprm -f cosmic_gum

#add the cosmic_gum geometry object
opadd geo cosmic_gum

#go into the cosmic_gum object
opcf cosmic_gum

#remove the default file sop
oprm file1

#add a sphere sop
opadd sphere my_sphere

#set its parameters
opparm my_sphere type (poly) freq (10)

Any line of the script that starts with a # is a comment. Comments are useful for noting the key points of a script. They are also useful when reading through a script after its creation. A good script will be well commented throughout in order to heighten its legibility.

SAVING AND SOURCING A HSCRIPT

This HScript can be saved in any location in the computer. It will however need to be sourced into the Textport in order to run it. Scripts which are non-project specific can be saved in a created folder called scripts located in **$HOME/houdini14.0/scripts**. Houdini will need to be re-launched after this scripts directory is created in order to see it.

The file extension for a H-Script is normally **.cmd**; however any file extension will work. A recent production trend has been to assign to HScripts a .hsc extension due to the Windows OS natively using script files with the .cmd suffix.

A simple script called **banana.hsc**, saved in **$HOME/houdini14.0/scripts** can be sourced and run by typing in the **Textport**:

source banana.hsc

A simple script called **pear.hsc**, saved in **$HOME/my_project/scripts** can be sourced and run by typing in the Textport:

source $HOME/my_project/scripts/pear.hsc

Only scripts saved in the $HOME/houdini14.0/scripts directory can be sourced without specifying a full location path.

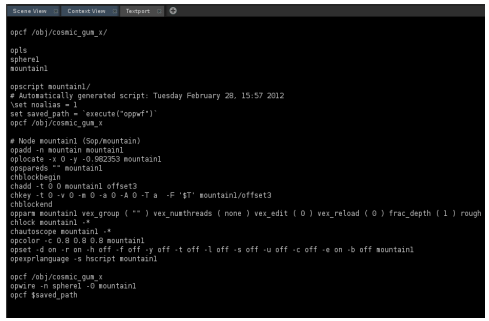
Save the script for creating the cosmic gum in a suitable location and source it in the Textport to ensure that it runs as expected.

THE HAND CREATED COSMIC_GUM OBJECT WILL ALSO NEED TO BE RENAMED SO NOT TO CONFLICT WITH THE SOURCED SCRIPT VERSION.

ADDING THE MOUNTAIN SOP

At present the simple script creates a polygon Sphere SOP called my_sphere. The original Mountain SOP can also be examined in the Textport using the **opscript** command to determine how it should be created.

Using the **opcf** and **opls** commands, navigate into the renamed **hand created cosmic_gum object**, and run the **opscript** command on the **mountain1** operator. As with the opscript return from the Sphere SOP, the majority of the output for creating the operator can be ignored; and the key creation commands can be deciphered.



The following commands can be derived from the opscript return for creating the Mountain SOP:

opadd -n mountain mountain1
chblockbegin
chadd -t 0 0 mountain1 offset3
chkey -t 0 -v 0 -m 0 -A 0 -F '$T' mountain1/offset3
chblockend
opparm mountain1 frac_depth (1) offset (0 0 offset3)
opset -d on -r on mountain1
opwire -n sphere1 -0 mountain1

SETTING KEYFRAMES & EXPRESSIONS

Setting the Offset Z Parameter of the Mountain SOP to $T is a straightforward matter when done by hand. When created using H-Script, it is slightly more convoluted process. The Textport return given by the opscript command on the Mountain SOP reveals a 3-stage process of adding a channel, setting its value, and then assigning it to the appropriate parameter. The first two of these stages are nested within the **chblockbegin** and **chblockend** commands.

chblockbegin
chadd -t 0 0 mountain1 offset3
chkey -t 0 -v 0 -m 0 -A 0 -F '$T' mountain1/offset3
chblockend

The **chadd** command is responsible for activating a channel for a particular operator. Before a keyframe or expression can be created, the channel must be declared as active beforehand. The **chkey** command is responsible for setting a value for the keyframe or expression and both can be simplified to the following pseudo-code:

chadd: at frame or time x (-f or –t value) on operator y activate channel z
chkey: at frame or time x (-f or –t value) set value on operator y / channel z

These two commands are then utilised by the opparm command to assign the value to the desired Parameter.

opparm mountain1 offset (0 0 offset3)

WIRING OPERATORS TOGETHER

The command responsible for wiring operators together is the opwire command. Using the cosmic_gum example as a basis, the syntax and meaning of it is derived as follows:

opwire -n sphere1 -0 mountain1
wire together the output of sphere1 into the first input (-0) of mountain1

COMPLETING THE SCRIPT

By examining the returns from both opscript command calls, a final cosmic_gum setup script can be created:

```
#THIS SCRIPT GENERATES COSMIC GUM
#switch to OBJ Level
opcf /obj
#remove any previous versions of the cosmic_gum
oprm -f cosmic_gum

#add the cosmic_gum geometry object
opadd geo cosmic_gum
#go into the cosmic_gum object
opcf cosmic_gum
#remove the default file sop
oprm file1
#add a sphere sop
opadd sphere my_sphere
#set its parameters
opparm my_sphere type (poly) freq (10)

#add a Mountain SOP
opadd mountain my_mountain

#activate its channels and values
chblockbegin
        chadd my_mountain offset3
        chkey -f 1 -F '$T' my_mountain/offset3
chblockend

#set its parameters
opparm my_mountain offset (0 0 offset3) frac_depth (1)

#wire the Sphere and Mountain SOPs together
opwire my_sphere -0 my_mountain

#activate the Display and Render Flags on the Mountain SOP
opset -d on -r on my_mountain

#layout the created nodes
oplayout -d 0
```

The script can be sourced as before to run it. The up and down arrows can be used in the Textport to scroll through previously entered commands to save retyping the source command.

A more advanced HScripting example can be found in Lecture - HScripting2

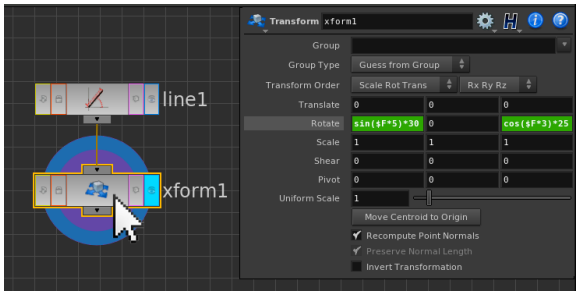
A SIMPLE PARTICLE WAND (TAKEN FROM LECTURE - PARTICLES #1)

Particles are a very versatile tool in Houdini. They can in essence be sculpted to create any shape and form required. Since Houdini 13, particles are now part of the Dynamics (DOPs) Level of Houdini, meaning that they will automatically interact with other dynamic solvers such as RBDs, Fluids, Volumetrics, Cloth etc. This lecture will however look at particle effects as a toolset in their own right.

In a **new Houdini scene**, create a **Geometry Object**, and inside at SOP Level, **replace** the **default File SOP** with a **Line SOP**. In the **parameters** for the **Line SOP** specify:

Length	**5**
Points	**20**

This will create a simple piece of geometry to emit particles from.

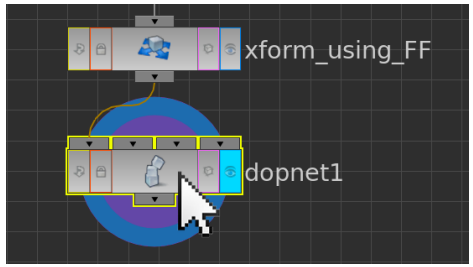


Append a **Transform SOP** to the Line SOP, and in its **parameters** specify:

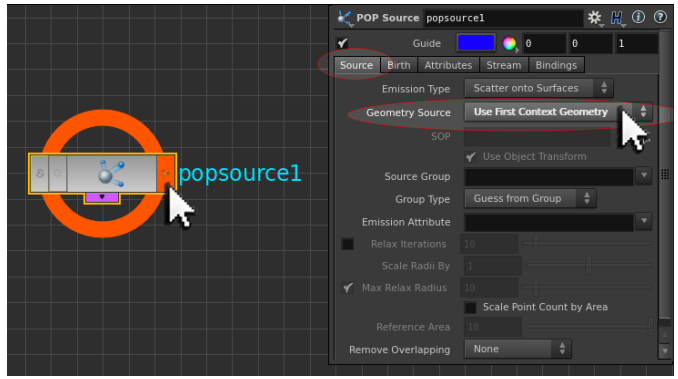
Rotate	**sin($F*5)*30**	**0**	**cos($F*3)*25**

This will animate the line creating a simple wand movement to it.

To the **output** of the **Transform SOP** append a **DOP Network** and **double LMB** on it to **go inside it**.



At **DOP Level**, create a **POP Source DOP**. This operator can read in geometry from SOPs and birth particles from them.



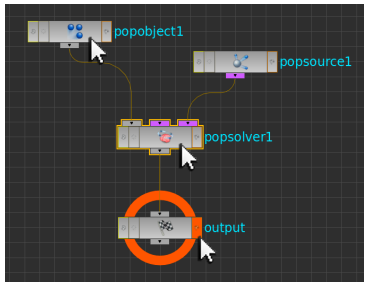
In the **parameters** for the **POP Source DOP** specify:

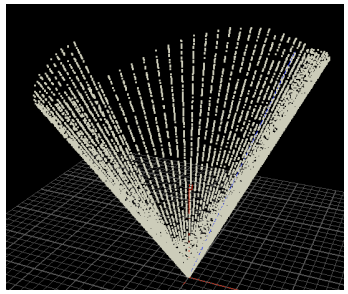
Source >

Geometry Source	**Use First Context Geometry**

Activating this option source the incoming geometry from the **first input** of the **DOP Network** node.

Alongside the POP Source DOP, **create** a **POP Object DOP** and a **POP Solver DOP**. Wire the **POP Object DOP** as the **first input** of the **POP Solver**, and the **POP Source DOP** as its **third input**. Wire the **output** of the **POP Solver** into the **Output DOP** and turn on its **orange Display Flag**.



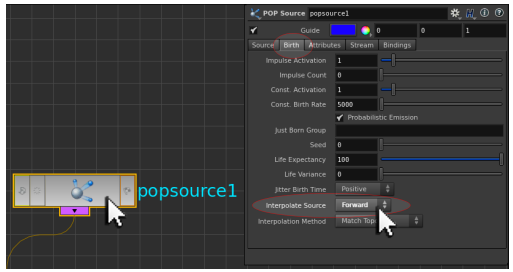


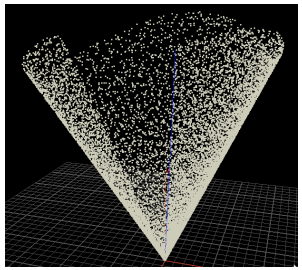
This will generate and solve particles from the geometry being read in by the POP Source DOP. When **PLAY** is pressed, a particle fan is created.

In the **parameters** for the **POP Source DOP** specify:

Birth >

Interpolate Source	**Back**





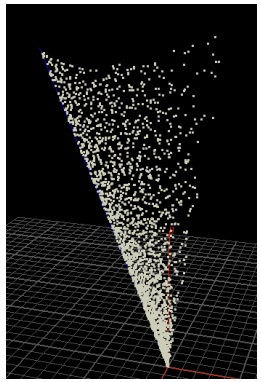
Reset the timeline to Frame 1 and press **PLAY** again. This time the particles are emitted more randomly from the edge of the line.

In the **parameters** for the **POP Source DOP** specify:

Birth >

Life Expectancy	**0.75**
Life Variance	**0.15**

When **PLAY** is pressed again, the particles die off after 0.75 seconds (with an additional life variance ranging between +0.15 and -0.15 seconds), creating a more ephemeral aesthetic.



IMPULSE ACTIVATION VERSES CONSTANT ACTIVATION

Examination of the Birth section of the POP Source DOP's parameters reveals two methods for birthing particles from incoming geometry. These are **Impulse Activation** and **Constant Activation**. By default Constant Activation is specified with a Constant Birth Rate of 5000 particles. **Constant Activation** works over **time**, meaning by default 5000 particles per second are born. **Impulse Activation** works each time the node cooks, meaning that an **Impulse Count** value will **birth** that many **particles each simulation frame**.

NOTE: The Constant and Impulse Activation parameters work as on/off switches. When these are set to a value of 0 no particle activation will occur. When set to a value of 1, particles will be born as per the Birth Rate value.

SCULPTING PARTICLES

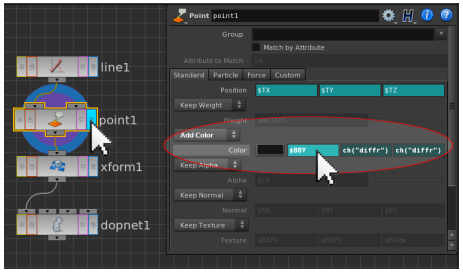
Creating visually rich particle systems is a combination of POP Operators and any incoming geometry behavior. Adding **custom attributes** to incoming geometry can dramatically improve particle aesthetics above and beyond what the POP operators can do by themselves. The art of particle work is therefore the understanding of how incoming geometry can be used to drive particle systems in bespoke ways.

To the **output** of the **Line SOP append** a **Point SOP**. This can be used to add a black and white ramp across the line, so it is white at its tip, and black at its base. In the **parameters** for the **Point SOP** specify:

Add Color

Color	**$BBY**	**ch("diffr")**	**ch("diffr")**

NOTE: Channel referencing can be activated to drive the green and blue channels from the $BBY expression of the red channel.

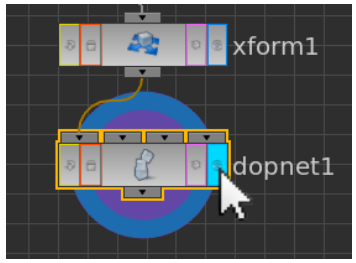


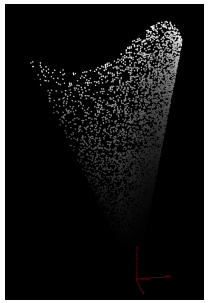
The visual end result of the **$BBY expression** is a **linear ramp** on the line. The colour of the ramp can be however be biased across the surface by wrapping the $BBY expression within a **Power Function**. This in essence bends the linear ramp to create a curve-based ramp instead. In the **parameters** of the **Point SOP**, modify the red channel expression of the Color parameter to:

Color	**pow($BBY,5)**	**ch("diffr")**	**ch("diffr")**

This will bias the white region of the ramp more to the tip of the wand. This black and white ramp can be used as a mask to procedurally determine where on the line geometry particles should be born.

NOTE: A custom parameter slider can be created on the Point SOP to procedurally control the numeric value specified in the Power Function, allowing it to be animated if necessary.





When the **Display and Render Flag** are reactivated on the **DOP Network**, and **PLAY** is pressed, the effect of the ramp can be seen on the particle colour. **See file H14_particle_wand_stage1.hipnc**

PASSING CUSTOM ATTRIBUTES INTO POPS

Custom attributes on geometry will by default be passed into a DOP Particle System. In this example, the curved ramp can be stored as a custom attribute and then passed into DOPS to control the emission of particles only towards the tip of the wand. This will also free up the colour of the particles allowing for another colour aesthetic to be assigned.

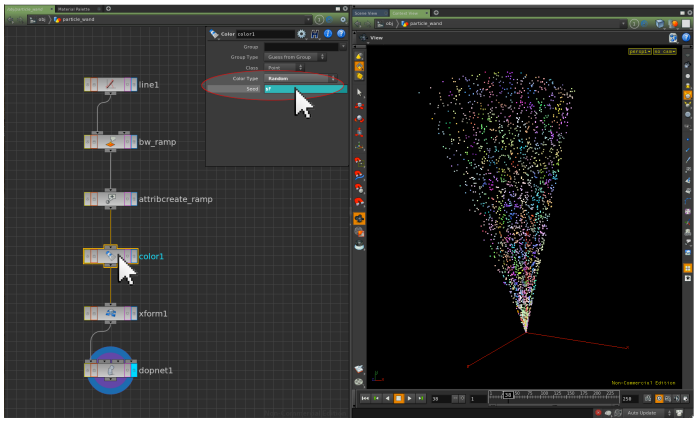
To the **output** of the ramp **Point SOP**, append an **Attribute Create SOP**. In the **parameters** for the **Attribute Create SOP** specify:

Name	**ramp**			
Value	**$CR**	**0**	**0**	**0**

GENERATING RANDOM PARTICLE COLOUR

Now that the ramp colour is stored as a custom attribute, new colour information can be assigned to the line geometry. **Append** to the **Attribute Create SOP** a **Color SOP**. In its **parameters** specify:

Color Type	**Random**
Seed	**$F**



Now when **PLAY** is pressed, the emitting particles are randomly coloured each frame.

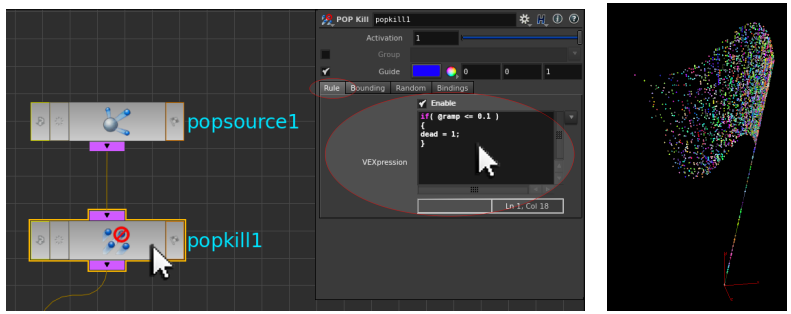
CONTROLING PARTICLE BEHAVIOUR USING CUSTOM ATTRIBUTES

Return back **inside** the **DOP Network**, and to the output of the **POP Source DOP** append a **POP Kill DOP**. The POP Kill DOP can **kill off particles randomly**, by using **Bounding Regions**, or by **VEXpressions**. **VEX** is a **higher-level Scripting Language** in Houdini also realized in node form as **VOPS** (VEX Operators). **VEXpressions** are however are short pieces of VEX code entered into nodes in order to set custom behaviors. **VEX Code differs** from **Houdini's Expression Language** in terms of **syntax** and **application**. It is **more verbose** that Houdini's Expression Language; but delivers many similar functions.

In the **Rule parameters** for the **POP Kill DOP** specify:

☑	**Enable**
VEXpression	**if (@ramp <= 0.1)** **{** **dead = 1;** **}**

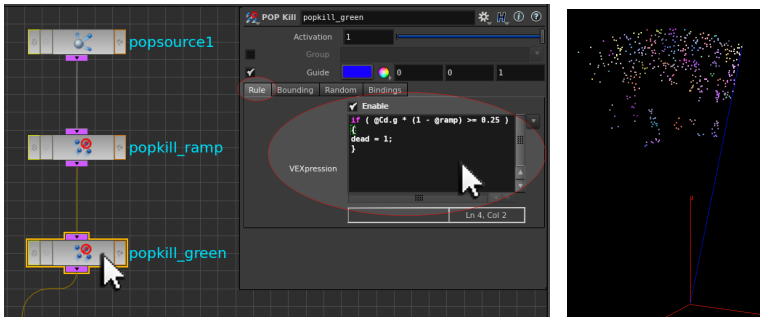
This will kill off particles generated towards the base of the wand, creating a flag of randomly coloured particles.



NOTE: In **VEXpressions**, **attributes** are **called** using the **@** symbol rather than $UPPERCASE, and **all commands** must be ended with a semicolon (**;**).

Append a **second POP Kill DOP** to the **POP network**. In it's **Rule parameters** specify:

☑	**Enable**
VEXpression	**if (@Cd.g * (1 - @ramp) >= 0.25)** **{** **dead = 1;** **}**



This VEXpression will compare the green value of each particle with its position on the ramp, removing green particles below a certain value. When **PLAY** is pressed, any green particles are thinned out towards the bottom area of the flag.

See file H14_particle_wand_stage2.hipnc

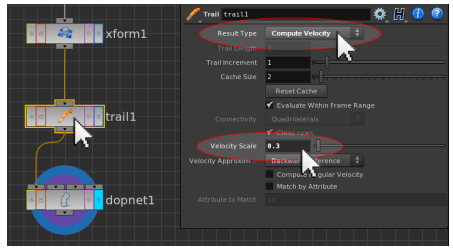
ASSIGNING GEOMETRY VELOCITY TO THE PARTICLES

By default particles will inherit any velocity information found on the incoming geometry. As velocity is not normally present on geometry, it needs to be calculated and activated as a custom attribute. This can be done using a **Trail SOP**.

Return back to **Geometry Level,** and to the **Transform SOP append** a **Trail SOP**. In its **parameters** specify:

Result Type	**Compute Velocity**
Velocity Scale	**0.3**

This will compute the velocity but also scale its values down so it does not adversely affect the overall look of the particle system.



When **PLAY** is pressed again, the movement of the particles is more fluid and gelatinous as a result of the inherited velocity. This velocity movement can also be added to further by specifying on the **POP Source DOP**:

Attributes >

Initial Velocity	**Add to inherited velocity**		
Variance	**0.2**	**0.2**	**0.2**

This will give each particle movement based upon the movement of the wand, plus and additional random direction variance, resulting in a more naturalized aesthetic.

GROUPING PARTICLES

POPS have a number of **existing attributes** that can be called on its operators. A useful list of POP variables is included at the end of these lecture notes. One such variable is **@life**. This returns the percentage of lifespan of each particle as a value between 0 and 1.

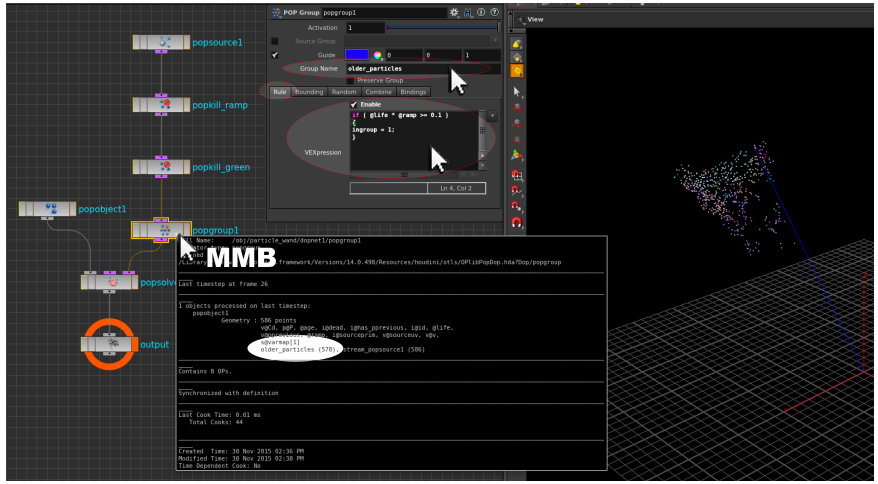
Append a **POP Group DOP** to the particle network, and in its **parameters** specify:

Group Name **older_particles**

Rule >

☑	**Enable**
VEXpression	**if (@life * @ramp >= 0.1)**
	{
	ingroup = 1;
	}

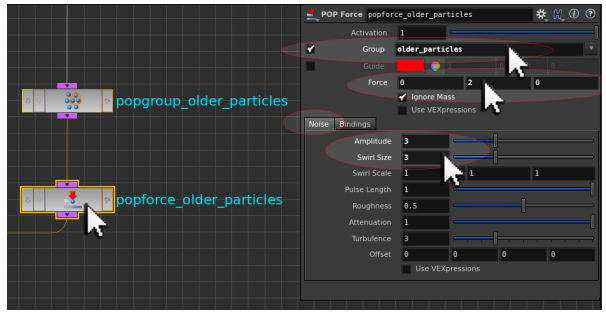
This will group any particles whose **$LIFE** value has exceeded **0.1**, and whose position is higher up the particle wand.



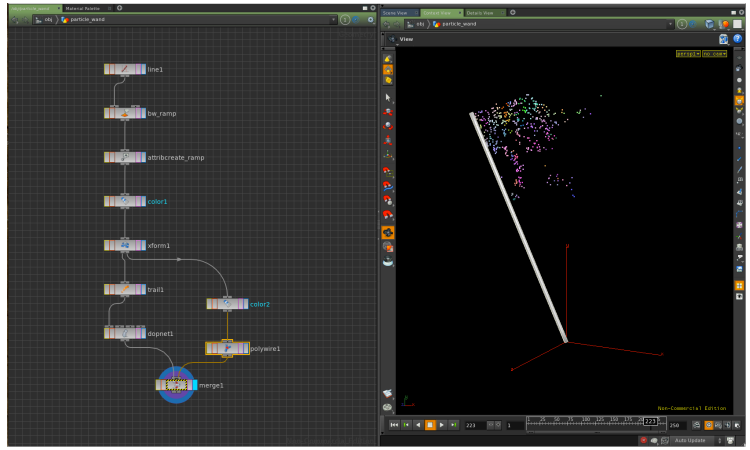
This grouping can be verified by **MMB** on the **POP Group DOP** after **PLAY** has been pressed. This grouping can be used to drive secondary particle animation effects on older particles before they die off.

To the **output** of the **POP Group DOP** append a **POP Force DOP**. In its **parameters** specify:

☑	**Group**	**older_particles**		
Force		**0**	**2**	**0**
Noise >				
Amplitude		**3**		
Swirl Size		**3**		



When **PLAY** is pressed again, the older particles have additional oscillation to their movement.



As a final step, a **Color SOP** and **Polywire SOP** can be used to create a piece of wand geometry from the original animated line, and merged with the final wand particle system.
See file H14_particle_wand_end.hipnc

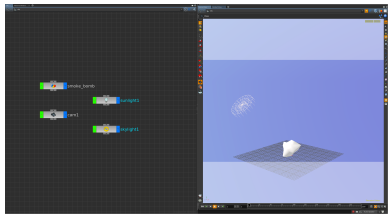
A SIMPLE SMOKE BOMB (TAKEN FROM LECTURE – DYNAMICS #2)

The **Pyro FX Shelf Tools** can create a number of presets for smoke, fire and explosion effects. Selected geometry can be assigned one of these presets, and Houdini will configure the associated networks automatically.

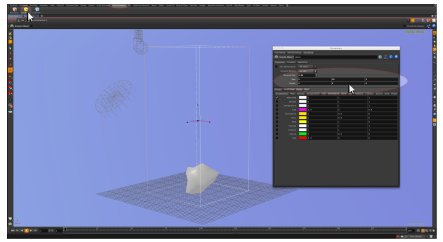
While these presets are very useful for quick effects setup; it is also useful to know a bit more about the components of dynamic volumetrics by creating a similar setup manually. This can be facilitated using the **Fluid Containers** and **Populate Containers** Shelf Tools.

The **Fluid Containers Shelf** has three items: The **Pyro**, **Smoke** and **Liquid** Containers. These will create the digital equivalent of an Acquisition Water Tank to hold the fluid effect. Each of these tanks is a box with user-definable divisions to fill the box with voxels (3D pixels). As volumetric data is passed through these voxels, they are shaded accordingly. This means for example that volumetric smoke travelling through its container is actually static voxels being procedurally activated and deactivated to give the impression of moving smoke. A greater resolution of voxels will increase the visual quality of the fluid, but will result in longer render times.

Open the scene H13_smoke_bomb_begin.hipnc. This scene contains an animated sphere (the smoke bomb) and a simple camera and Sky Light setup. A Mantra PBR node has also been activated allowing for physically based rendering of the smoke bomb effect.



Maximize the **Scene View** and activate the **Shelves**. Ensure that the **Scene View's Tool Options** are set to **Create at Object Level**. From the **Fluid Containers Shelf** activate a **Smoke Container**. Press **ENTER** to **automatically place it** at the scene origin.

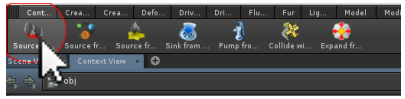


This will create an **AutoDopNetwork** with a **Smoke Object DOP** inside it. Pressing **p** with the mouse over the viewer can activate the parameters for the **Smoke Object DOP**. In the **parameters** for the **Smoke Object DOP** specify:

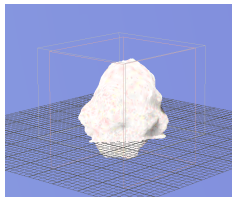
Division Size	**0.05**		
Size	**5**	**10**	**5**
Center	**0**	**5**	**0**

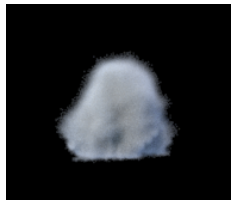
This will move and resize the smoke container so it surrounds the smoke bomb geometry, as well as increasing the voxel resolution of the container.

At **Object Level**, **select** the **smoke_bomb sphere** and activate the **Source from Surface** shelf button found on the **Populate Containers** Shelf. **When prompted**, **select** the **Smoke Object Container** and press **ENTER** to complete the setup. The **Source from… shelf tools** will automatically generate a fluid from an object, based upon the type of fluid container selected.



When **PLAY** is pressed, smoke emits from and surrounds the animated sphere.





NOTE: The boundaries of the smoke container automatically resize relative to the smoke effect generated. This behavior can be deactivated if required.

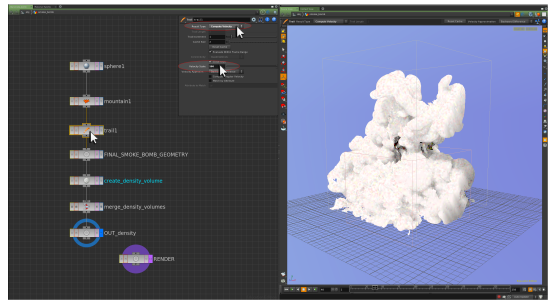
NOTE: Only objects within a container will respond to any fluid properties assigned to them. Once an emitter object moves outside of its container, its ability to produce fluid is culled.

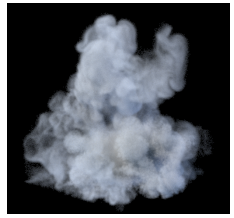
CREATING MORE INTERESTING SMOKE

Currently the smoke effect is somewhat tepid. This is because any source geometry for the smoke must first be primed in an interesting way to generate interesting smoke effects.

Inside the **smoke_bomb object**, insert a **Trail SOP** after the Mountain SOP animating the sphere. In the **parameters** for the **Trail SOP** specify:

Result Type	**Compute Velocity**
Velocity Scale	**500**

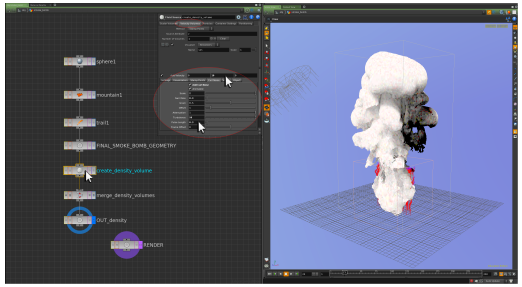




When **PLAY** is pressed, a more dramatic smoke effect is created.

Modifying the **Fluid Source SOP** creating the **density volume** from the smoke bomb geometry can further enhance this smoke effect. In the **parameters** for the **Fluid Source SOP** specify:

Velocity Volumes >				
☑	**Add Velocity**	**0**	**10**	**0**
Velocity Volumes > Curl Noise >				
☑	**Add Curl Noise**			
	Scale	**2**		
	Swirl Size	**0.3**		
	Turbulence	**50**		
	Pulse Length	**0.3**		



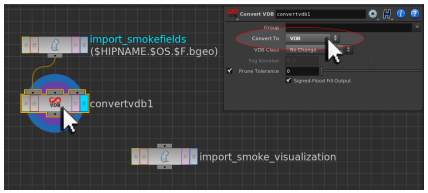
The **AutoDopNetwork's Smoke Solver DOP** can also adjust the overall fluidity of the smoke. Increasing **Viscosity** for example will create more **gooey smoke** aesthetics. The **Resize Container** node can determine which **sides of the fluid container** are deemed as **solid** or which of them are **holes** for the smoke fluid to escape out of the container.

See file H14_smoke_bomb_stage1.hipnc

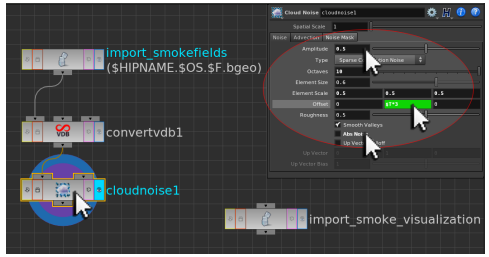
REFINING FLUID AESTHETICS

Once the overall shape of the smoke fluid has been established, its aesthetic can be refined. Inside the **smoke_import1 object**, append a **Convert VDB SOP** to the **import_smokefields SOP**. In the **parameters** for the **Convert VDB SOP** specify:

Convert To **VDB**



Append a **Cloud Noise SOP** to the Convert VDB SOP. This will allow cloud noise to be added to the smoke simulation, creating a greater sense of turbulence to the smoke aesthetic.



In its **parameters** specify:

Noise >			
Amplitude	**0.5**		
Octaves	**10**		
Element Scale	**0.5**	**0.5**	**0.5**
Offset	**0**	**$T*3**	**0**
■ **Abs Noise**			

NOTE: This operator only works on VDB volumes, not standard Volumes; hence the use of the Convert VDB SOP.

When the simulation is played again or rendered, the plume of smoke now has greater sense of visual scale due to the additional surface definition created by the Cloud Noise SOP.

The aesthetic of the smoke can further be refined by **modifying the parameters** of the **Billowy Smoke Material** automatically created when the smoke_bomb object was sourced by the smoke fluid container.

At **SHOP Level**, locate the **Billowy Smoke Material** and in its **parameters** specify:

Smoke Intensity	**4**		
Smoke Color	**0.25**	**0.25**	**0.25**
Density >			
Smoke Density	**4**		
Noise >			
☑ **Do Noise**			
Frequency	**0.5**	**0.5**	**0.5**
Amplitude	**0.2**		
Offset	**0**	**$T**	**0**

When the simulation is rendered again, and much denser plume of smoke is generated. Modification of the Billowy Smoke Material can also help increase the sense of scale to the smoke.

Adjusting parameters of the **Mantra PBR ROP** can also refine the render aesthetic of the smoke. A key parameter for rendering volumes is the **Volume Limit parameter**. This parameter controls **how many times light can bounce internally through a volume** giving internal volume illumination. This can be useful when creating more naturalistic volume effects (for example sunlight permeating through a cloud). At **OUTPUTS Level**, specify in the **Mantra PBR ROP**:

Rendering > Sampling	
Volume Quality	**1**
Rendering > Limits	
Volume Limit	**3**

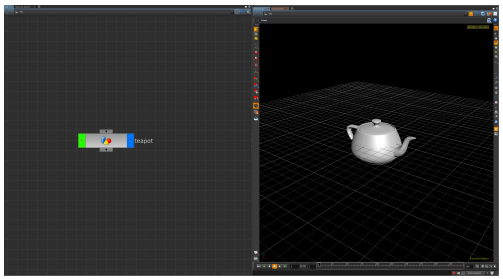
When the scene is rendered again, the smoke plume has a greater sense of internal lighting and overall quality; however render times have increased.

See file H14_smoke_bomb_end.hipnc

NOTE: ALL ARCHIVE LECTURE NOTES CAN BE FOUND IN: /public/mapublic/Phil/Archive/OLD_1415

SEQUENCE ANIMATION IN CHOPS

It is possible to utilise CHOPs as a sequence animation editor. This allows for clips of animation to be ordered relative to each other. Open the scene **teapot _begin.hipnc**

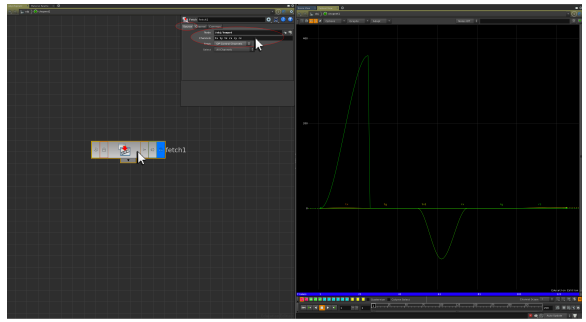


This scene contains a key-framed teapot. When **PLAY** is pressed, there are **three teapot animation sets**, a **back flip** (**frames 1 – 25**); a **wiggle** (**frames 50 – 75**); and a **jump forward** (**frames 100 – 125**). These animation sets can be passed into **CHOPs**, **re-ordered** and **re-sequenced** with the new animation data passed back onto the teapot.

Alongside the teapot object, create a custom **CHOP Network**. Inside this network create a **Fetch CHOP**. In the **Parameters** for the **Fetch CHOP** specify under the **Source** section:

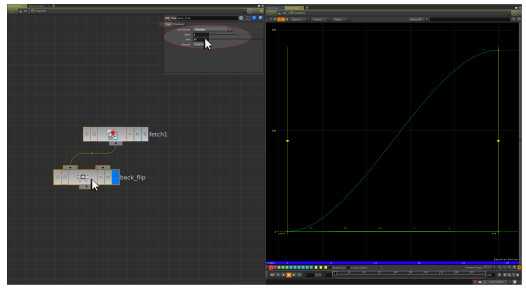
Node	**/obj/teapot**
Channels	**tx ty tz rx ry rz**

This will read in both the translational and rotational data for the teapot's animation. These data channels can now be edited into their three component animation parts by using a Trim CHOP.



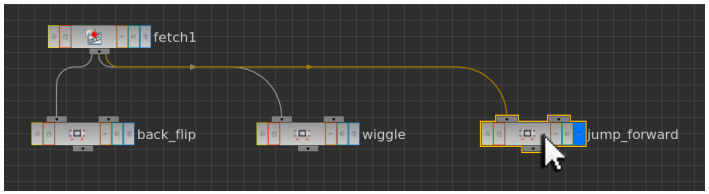
To the **Fetch CHOP** append a **Trim CHOP** and rename it to **back_flip**. In the **Parameters** for the Trim CHOP specify:

Trim >	
Unit Values	**Absolute**
Start	**1**
End	**25**
Common >	
Units	**Frames**



This will trim the animation data from the original channels creating a standalone clip of the back flip section.

Repeat this operation for the other two animation sequences; the wiggle (frames 50-75); and the jump forward (frames 100-125) branching each of them as separate node chains from the Fetch CHOP.

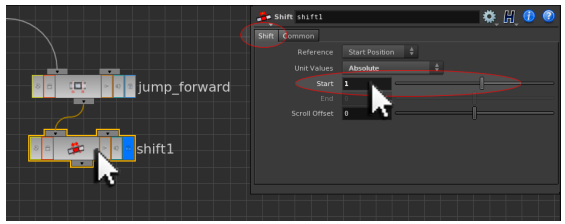


See teapot_ stage1.hipnc

CREATING A SEQUENCE OF CLIPS

Starting with the **jump_forward clip**, append a **Shift CHOP**. Under the **Common** section of the **Parameters** set the **Units** to **Frames**. Under the main **Shift** Section specify:

Unit Values	**Absolute**
Start	**1**



This will move the clip to the start of the timeline.

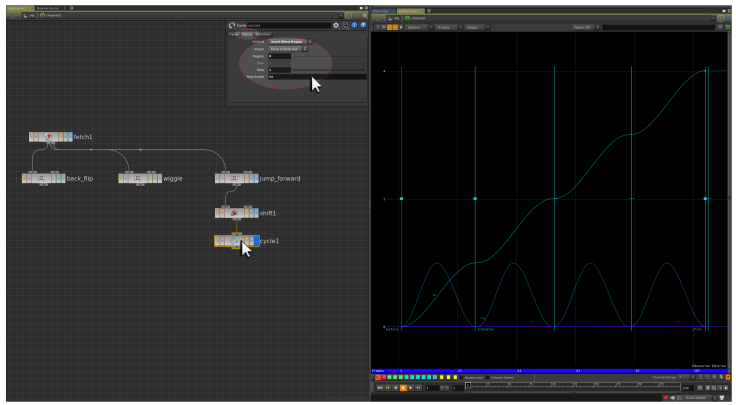
To the **Shift CHOP** append a **Cycle CHOP**. In the **Parameters** Cycle CHOP specify:

Cycle >

Cycles After	**3**
☐	**Blend Start to End**

Blend >

Method	**Insert Blend Region**
Region	**0**
Step	**1** (second)
Step Scope	**tx**

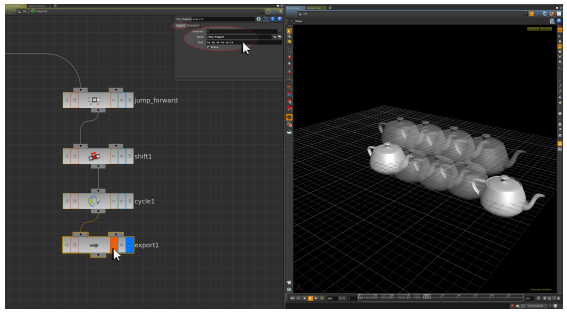


This will create four cycles of animation where each leap forward (the tx value) is added to relative to its previous position. Without specifying a Step and a Step Scope Parameter the result would be a teapot that would reset itself to its original tx position after each leap forward.

To test the result of this animation sequencing, append to the **Cycle CHOP** an **Export CHOP**. In the **Parameters** for the **Export CHOP** specify:

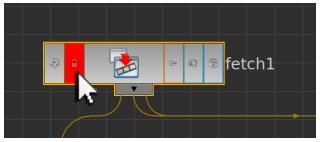
Node	**/obj/teapot**
Path	**tx ty tz rx ry rz**

Ensure that the **Orange Export Flag** is **activated** on the Export CHOP. Switch the **Viewer** over to a **Scene View**, and press **PLAY**. The **teapot** will **jump forward four times**.



PREVENTING RECURSION ERRORS

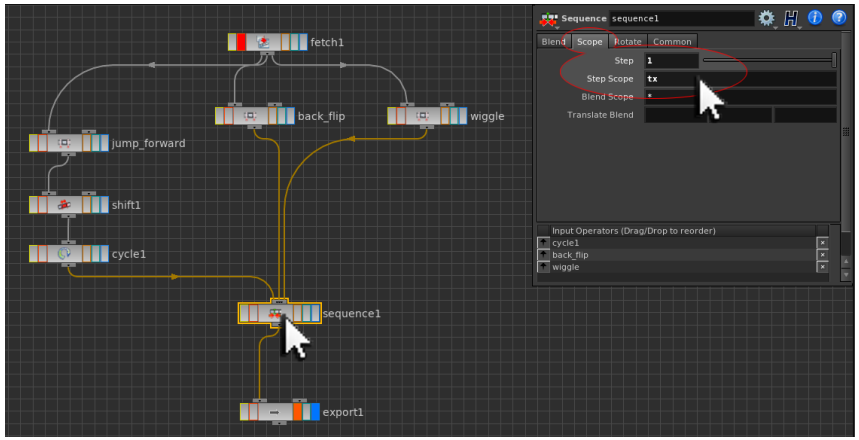
While everything works as it should, it is good practice to **lock** the **Fetch CHOP** reading in the original animation sets. This will **prevent** any **recursion errors** from the exported CHOP animation data being re-passed back into CHOPs as an infinite cycle. **Ensure the timeline is set to frame 1** and **lock** the **Fetch CHOP**.



THE SEQUENCE CHOP

The cycled animation sequence can be appended with a **Sequence CHOP** inserted before the Export CHOP. A **Sequence CHOP** will accept multiple inputs allowing for all of the clips to be ordered and re-ordered as the user desires. In the **Parameters** for the **Sequence CHOP** specify:

Scope >	
Step	**1** (second)
Step Scope	**tx**



Now the back_flip and the wiggle can be wired into the Sequence CHOP. The visual end result is a teapot that will leap forward four times, perform a back flip followed by a wiggle.

See file **teapot _stage2.hipnc**