xapkohheh
vimeo , patreon
v 1.0.1
If you want to participate in the editing, please draft

Foreword

The purpose of this article is to expand on explaining the basic principles of the **DOP** context . Here you will not find descriptions of setting up simulations, instructions on how to build **Custom Smoke Solver** , or explanations for each instrument or node.

For a comfortable understanding of this material, it is recommended that you have basic knowledge of the Houdini and **DOP** context and have some experience working with standard presets and **Shelf Tools** . Information came from official documentation on **Houdini** and **HDK** , and white spots were covered by theories supported by experiments.

The article is accompanied by examples. All examples will be presented in the form of **hpaste** links (of the form abcdefg @ HPaste), a guide to using this utility can be found here , the plug-in itself can be downloaded here . If you already have **hpaste** installed , but the links do not work, upgrade to the latest version.

Most examples contain nodes from the **OBJ** context and output data to the **console** . On Linux, this will be the **terminal** window from which Houdini was launched; on Windows, a small **console** window will appear on the first output. For your convenience, **Python Shell is** highly recommended . Upon opening, all output will be automatically redirected there.

To restart the simulation from within the **DOP** network without pressing the Reset Simulation button - select the **Output** node , put the **Bypass** flag on and off , then rewind the simulation to the first frame (even if it is already on the first frame).

DOP

So let's get started. **The DOP** context looks deceptively similar to **SOP** , but it works in a completely different way. A node graph in a dynamic context is an instruction for creating and processing objects and data that exist separately from nodes and are only artificially attached to them.

DOP objects with data attached to them and relayships connecting them are the main working part of the dynamic context. They exist inside the simulation regardless of the nodes; information on their presentation can be found in the **Geometry Spreadsheet** tab within the context.

DOP data

For starters, let's forget about DOP nodes . (In this section, DOP nodes will not be mentioned at all)

Basic concepts

Simulation

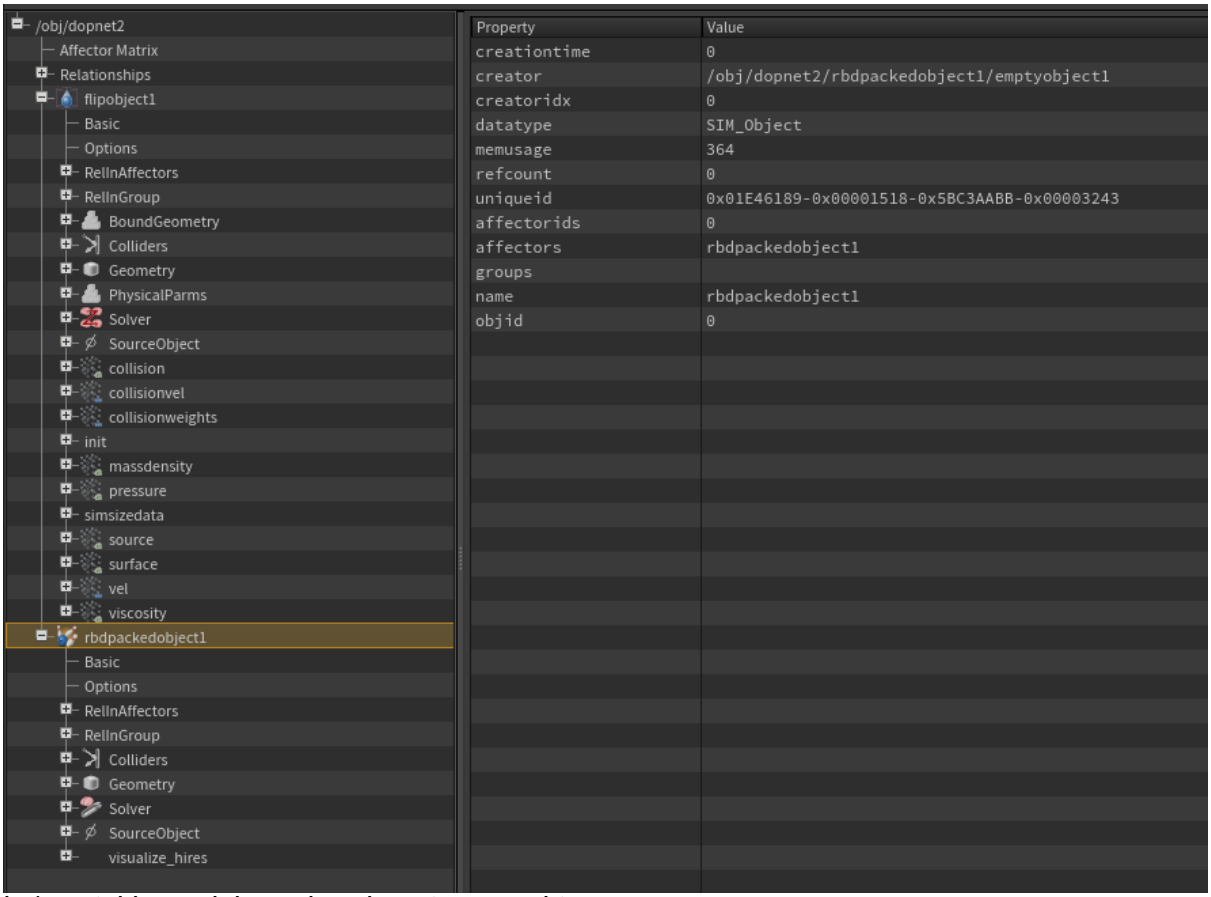
Simulation in a DOP context is an environment in which data exists and is transformed. The simulation is divided into steps, usually with a fixed time step, which we will call timesteps (not frames, so as not to be confused, since the timestep may not be equal to one frame).
Simulation should not be represented as an entity that exists in one timestep, and varies from timestep to timestep, from step to step.
The simulation exists outside the timeline, it is more correct to think that the timeline exists inside the simulation, and each timestep on the timeline is the root for storing links to additional data, these additional data, in turn, exist not related to time, and can participate in data structures attached to different timestepam.
Take, for example, data representing a constant force of gravity, or a static and immutable collider. Objects from different timesteps can have a link to this data, that is, data exists outside of timesteps in a single instance, but different objects of the same timestep, or the "same" object in different timesteps, can refer to this constant gravity or collider.
At first, this may seem like an overcomplicated concept, but then it becomes clear that in this definition, caching simulations in conjunction with date sharing will become easier to understand.

Data

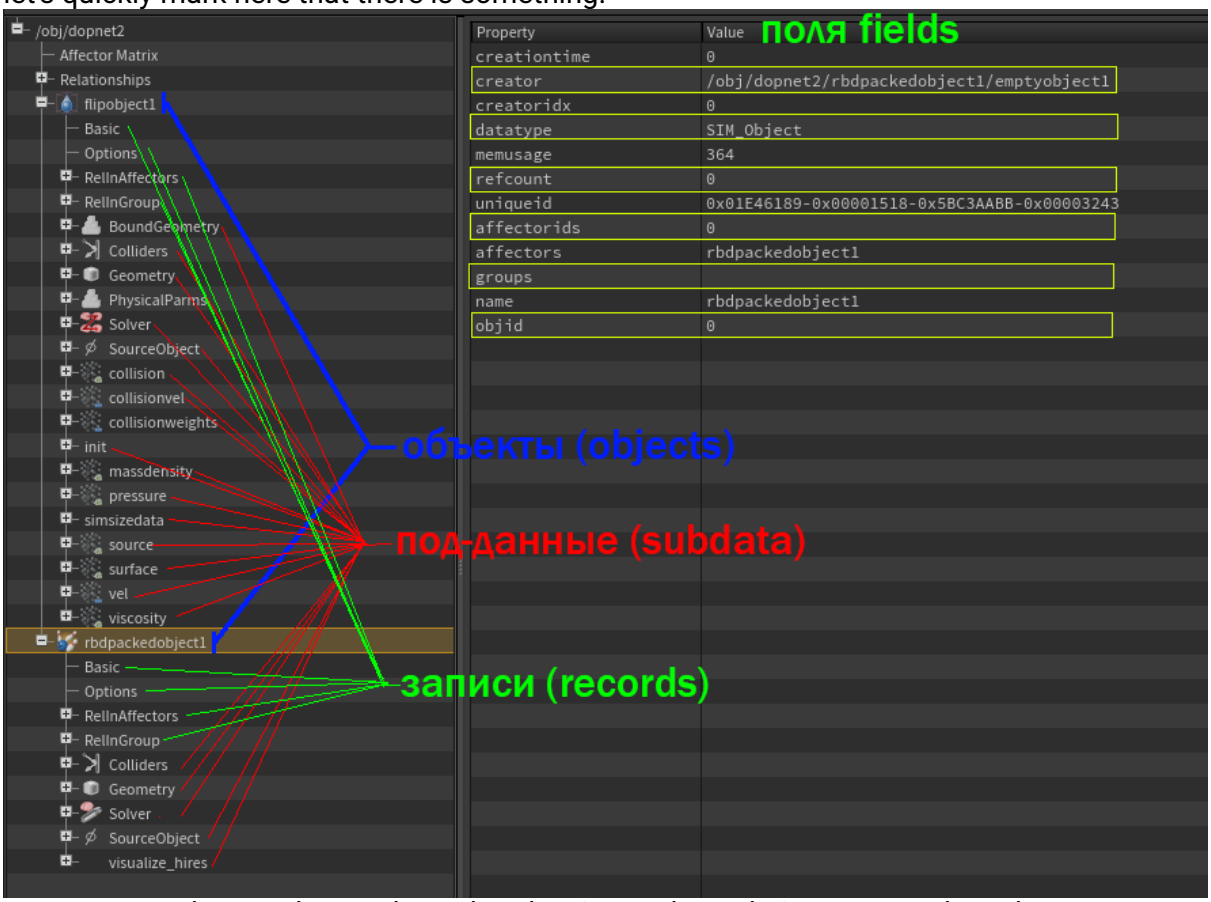
DOP data is the basic unit of DOP simulation. DOP data changes itself and each other according to the rules defined by them. Objects, Relayships, Solvers, Force, Geometry (as understood by DOP), volume fields are all different specialized types of DOP data, and they all have the same functional core.

Any data can store one or more "DOP records", which are a simple key-value table, or a dictionary in which a key-value pair is called a field (yes, a little confusion with volume fields), the keys are ordinary strings, and the values corresponding to them can be any basic Houdini data types (for example, string, int, float, vector, matrix, array of these, etc.).
In addition to records, data can store a link to other data, in which case this data is called subdata . That is, we call data sub-data when we talk about them in the context of having a link to them on other data. It is important to note that the name is not part of the data, the name belongs only to the data link, and the same data can be sub-data of different data under different names.
For example, the same geometry (and I mean not the same geometry, namely the same DOP geometry data in the simulation) can be attached to two different objects, while on one object, say, Geometry, on the second - ReferenceGeometry and used by these objects in different ways, while not duplicating in memory.
(example: **sppexegoqu @ HPaste** *(paste this example into an empty DOP node))*
I note that even though the data as such does not have a name, some special data specializations called root data - objects and relayships - have their own name attached to the object itself. This name is implemented by a special field in the object master record.
However, the object, while still being the basis of the data, can be attached to another object as sub-data with a link name that is not connected in any way with the name of the object. In doing so, keeping your own name in your records.
This is a complicated and useless trick, but still, technically, such an example may be interesting:
(example: **sppuwodalu @ HPaste** *(paste this example into an empty DOP node))*

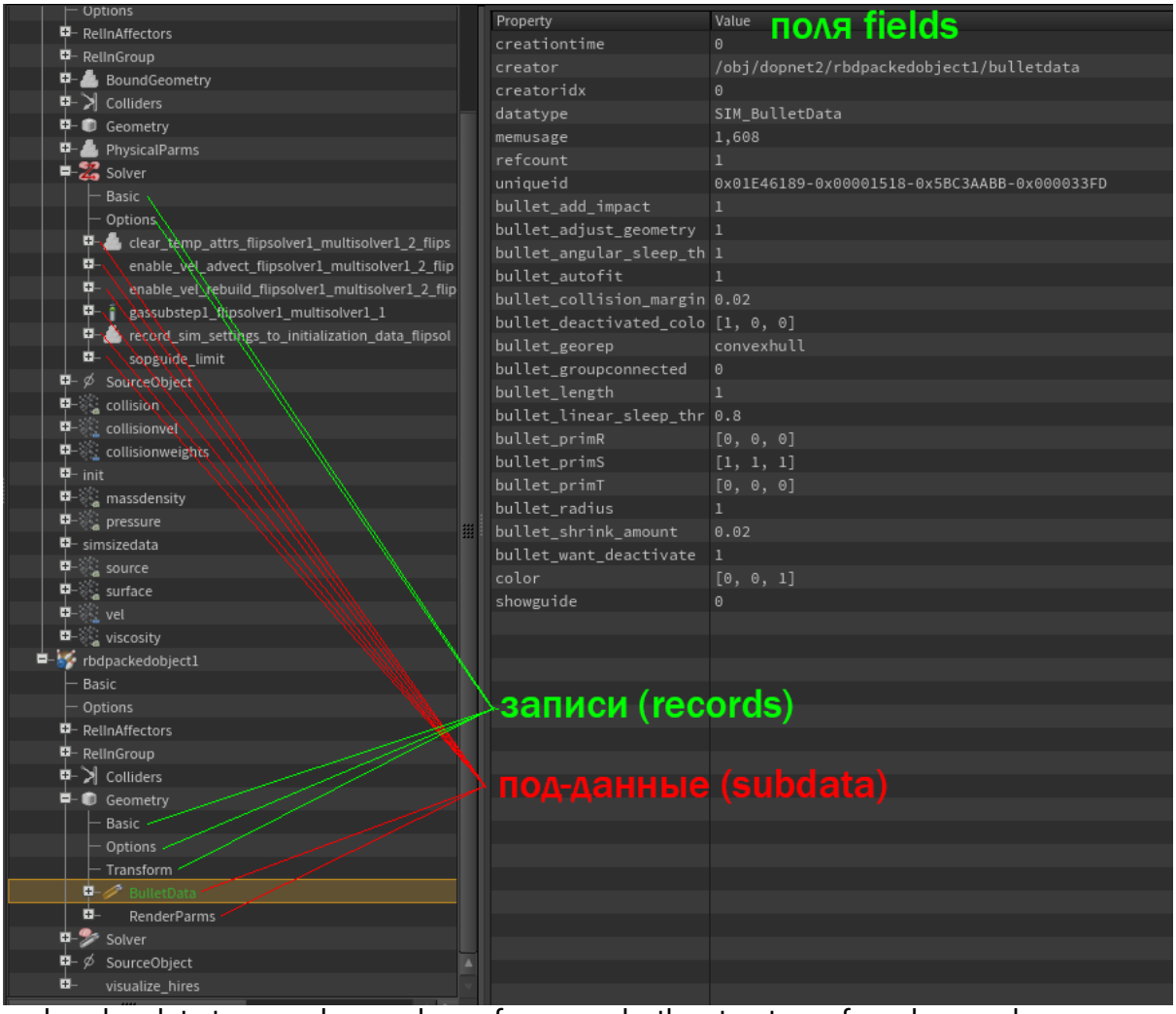
This may look like our usual simulation:



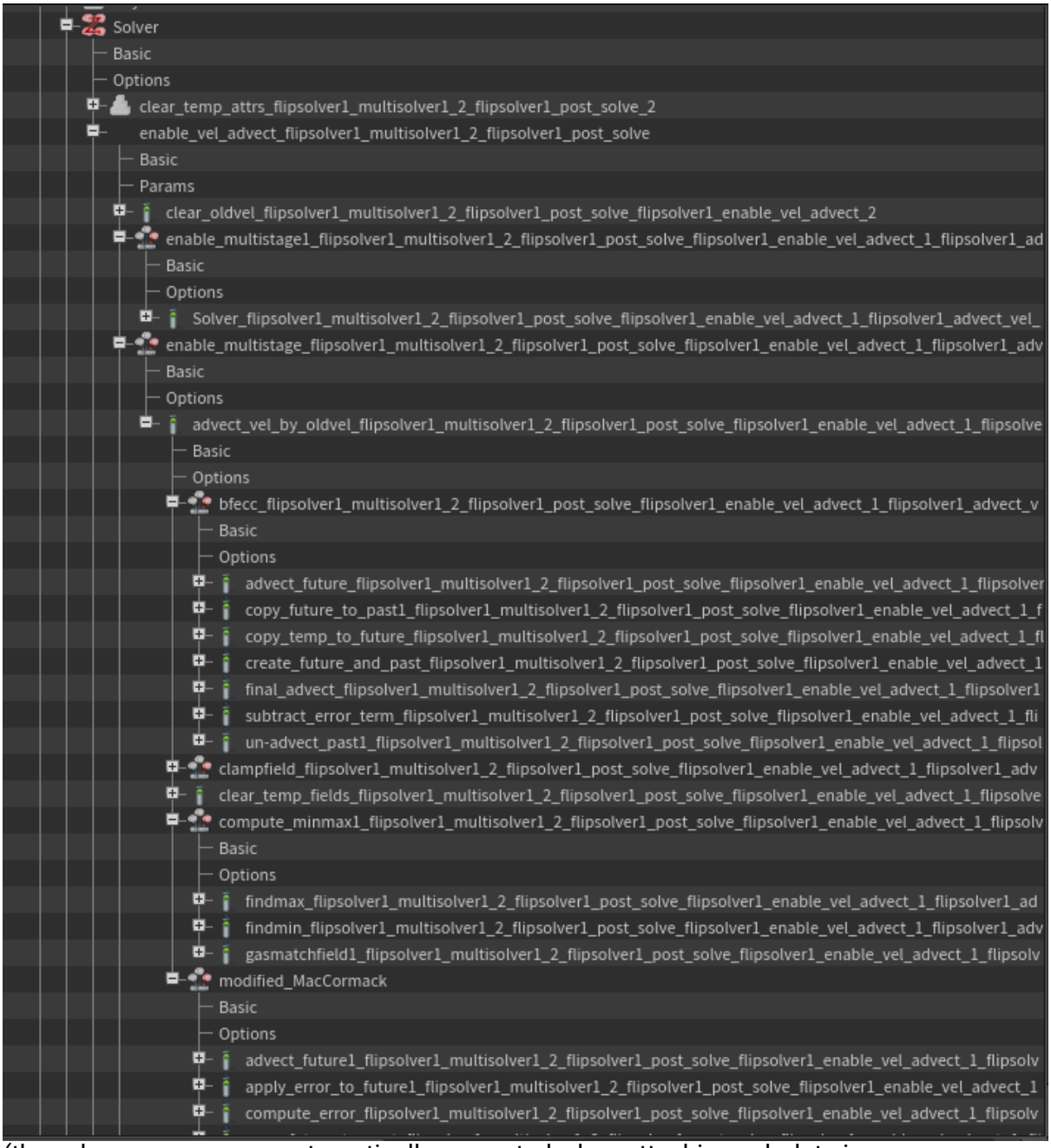
let's quickly mark here that there is something:



we can expand some data and see that they in turn have their own records and sub-data:



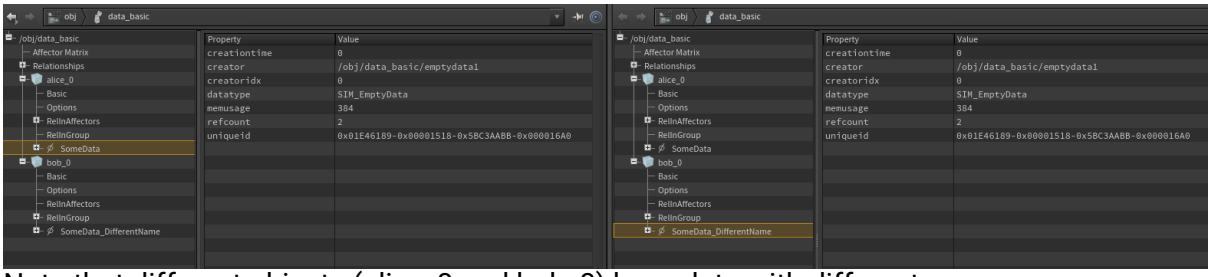
and such a data tree can be very large, for example, the structure of a solver, such as a flip assembled from microsolvers, is nothing more than one big sub-data tree:



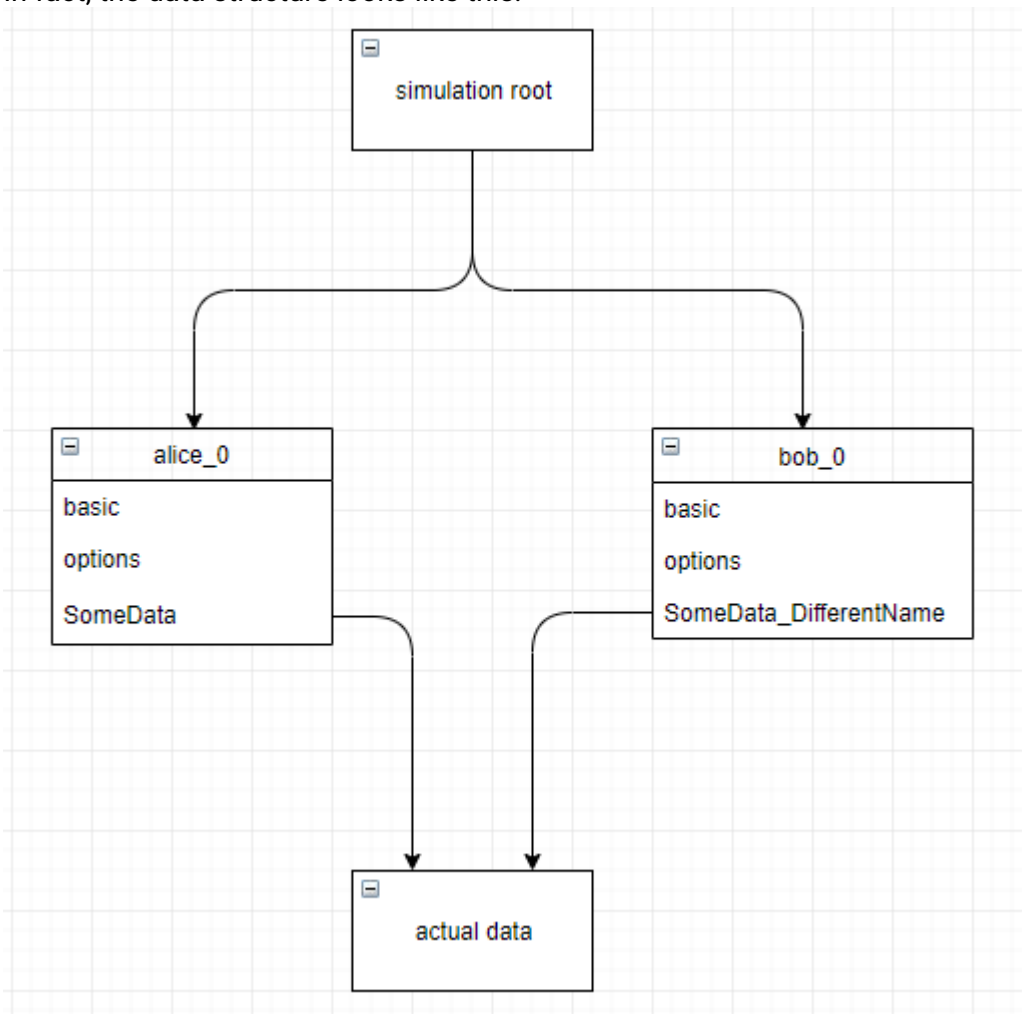
(these long names were automatically generated when attaching sub-data in order to be unique within the object - this is not necessary, it is enough for the names of links to sub-data to be unique only within their parent data)

Unique identificator

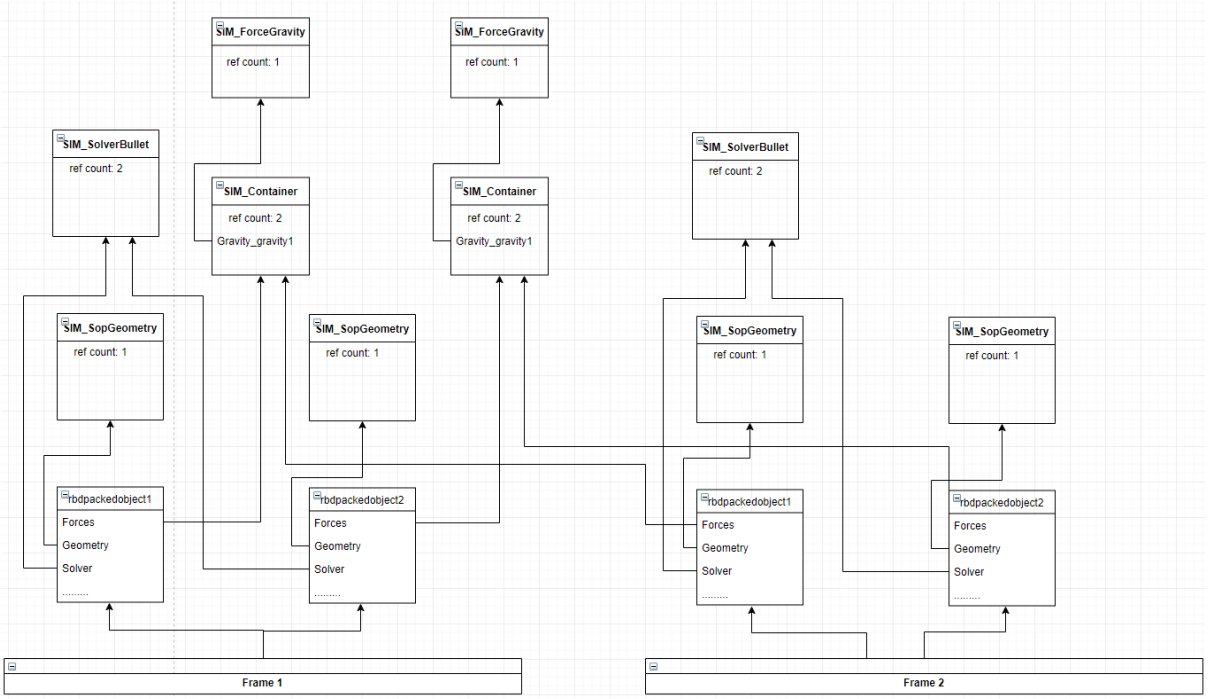
All data, and therefore everything in the DOP simulation, has a unique identifier, uniqueid, it uniquely identifies not only data within the same timestep, but in general throughout the simulation timeline, is unique in different runs of the same simulation, and even among several different computers. Those. uniqueid is really a unique identifier, and it certainly won't be repeated in the framework of one simulation in principle, in addition, it is with the least probability that it will ever be repeated in principle. It can be found in the default entry Basic on any data, and it can be used to judge with certainty whether different sub-data links point to the same data.



Note that different objects (alice_0 and bob_0) have data with different names (SomeData and SomeData_DifferentName), but in fact they are just different links to the same data (this can be seen by the same uniqueid displayed on both links. In fact, the data structure looks like this:



This example may seem abstract, but it is more common than you might think, for example, the default behavior of some solvers, such as a bullet — join objects as one common sub-data (why some solvers join this way by default - it will be clear in the chapter on stages of solving objects):



Reference counter

The data in the Basic record has a ref count field that stores the number of links from different data, i.e. how many times this data is found as sub-data for other data, not only at the current timestep, but generally in the simulation. gravity data can have a reference count of 100, although there is only one smoke object in each timestep, because in fact, all 100 smoks of objects in each frame live in the simulation cache, and each of them refers to the same gravity data.

Data modification, COW principle

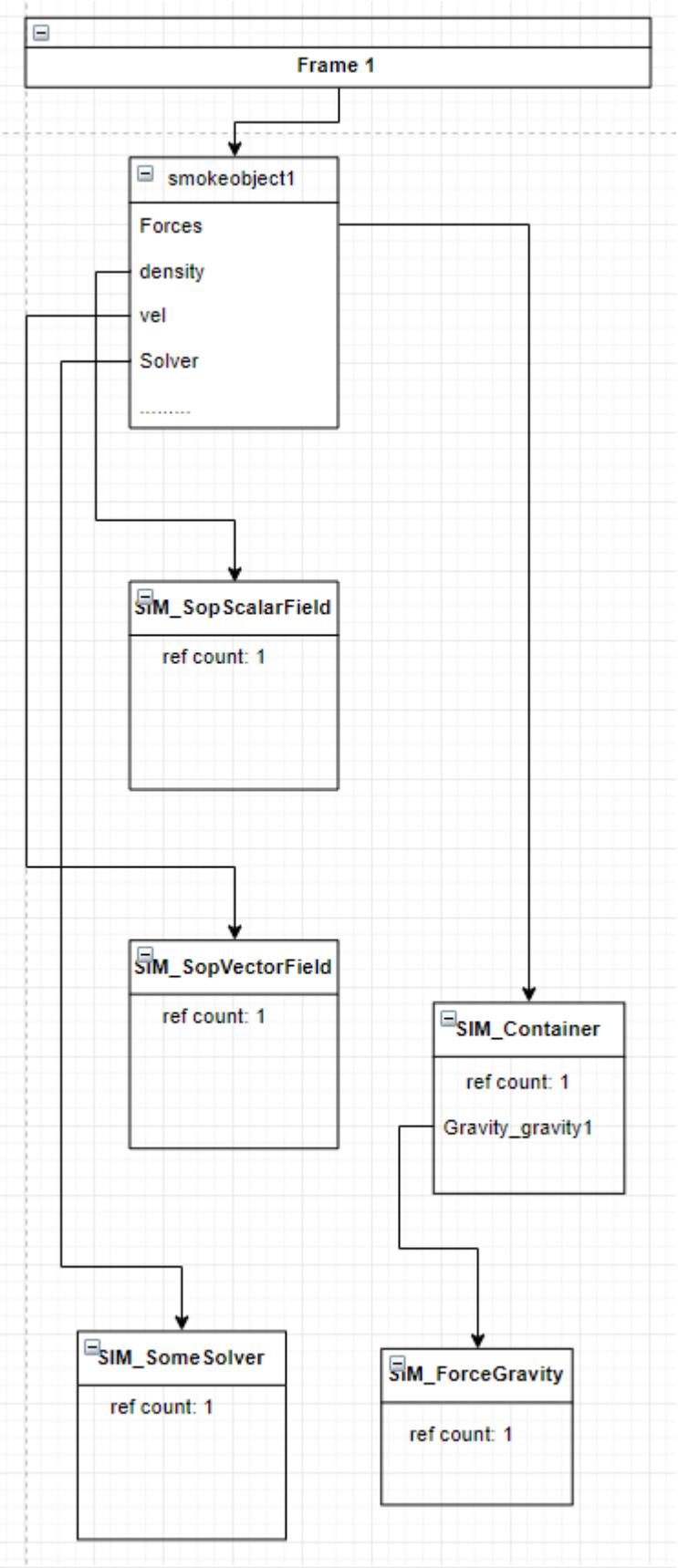
Another important and counterintuitive concept of DOP is copy-on-write (CoW) - if there is **more than one** link to some data , and some node tries to change this data by reference to the sub-data *(in the example below, this modifydata1 node tries to change under -data by reference with the name EmptyData on the bob object)* , - the data will be copied, the link to the sub-data will be changed to indicate a new copy, only then the node will be allowed to change the data. The purpose of this approach is to ensure that one object inadvertently does not change the data of another object, or itself from a previous / other timestep, while maintaining the possibility of multiple references to the same data in memory, saving computer resources.

(example: **sppocufatu @ HPaste** *(OBJ context))*
(also below will be an example of this process in the pictures)

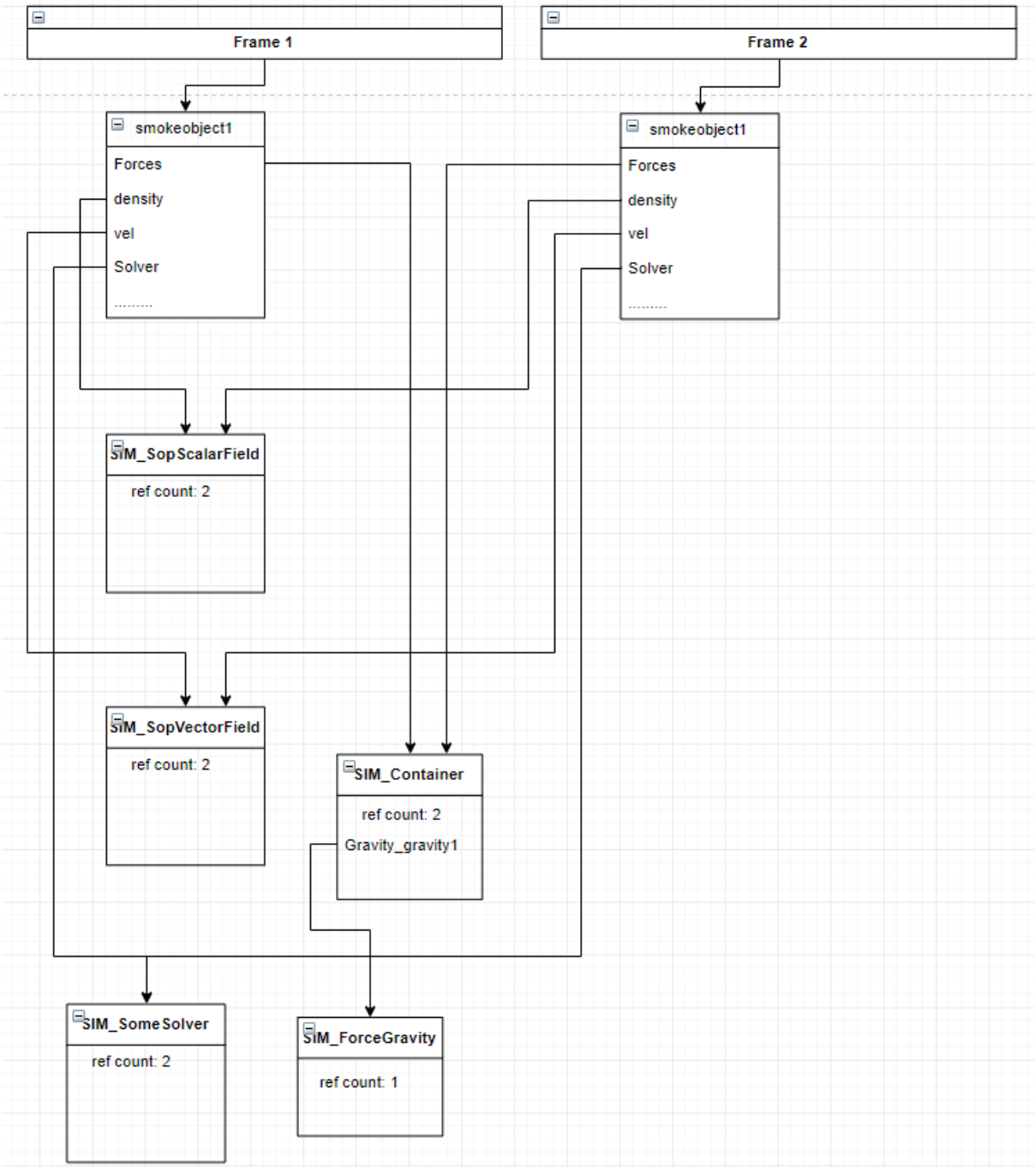
The root data of the simulation (objects and relayships, which may not be attached anywhere, will be discussed in more detail), belong to a certain timestep, and cannot belong to more than one timestep. We can say that timestep stores a unique link to the object data. Thus, in the case of a cached simulation (old timesteps and all their data are stored in the simulation memory), objects and relayships will be copied from timestep to timestep before they undergo any changes.
Relayships are automatically copied to the new timestep until the lists of their objects are empty. Objects must be copied to the new timestep explicitly, which will be described later in the discussion of the object stream.
Each time you change data in a cached simulation (for example, a solver changes geometry, or an animated parameter on a node changes some field) - a new copy of this data is created, so that the old data (from the previous timestep) remains unchanged. Thus, when simulating caching, each timestep will be an invariable

correct state of the simulation, which can be saved, loaded, and from which the simulation can be fully continued.
Note that in non-cached simulation data from previous timesteps is not stored, so there will be no references to objects from previous timesteps, which means there will be no extra links to data from these objects, so objects and data will have a reference count = 1, and data will not be copied when changing, so that solvers can work on objects without copying objects or data. On the one hand, this can help save resources, on the other hand, if any solver needs the previous state of the object, for example, for some kind of interpolation, such a solver will not be able to work. Well, from the point of view of convenience - the user will not be able to quickly interactively navigate the timeline and analyze the state of the simulation .

Let's take a simple everyday example to look at how data is copied during the simulation.
Initial simulation state:

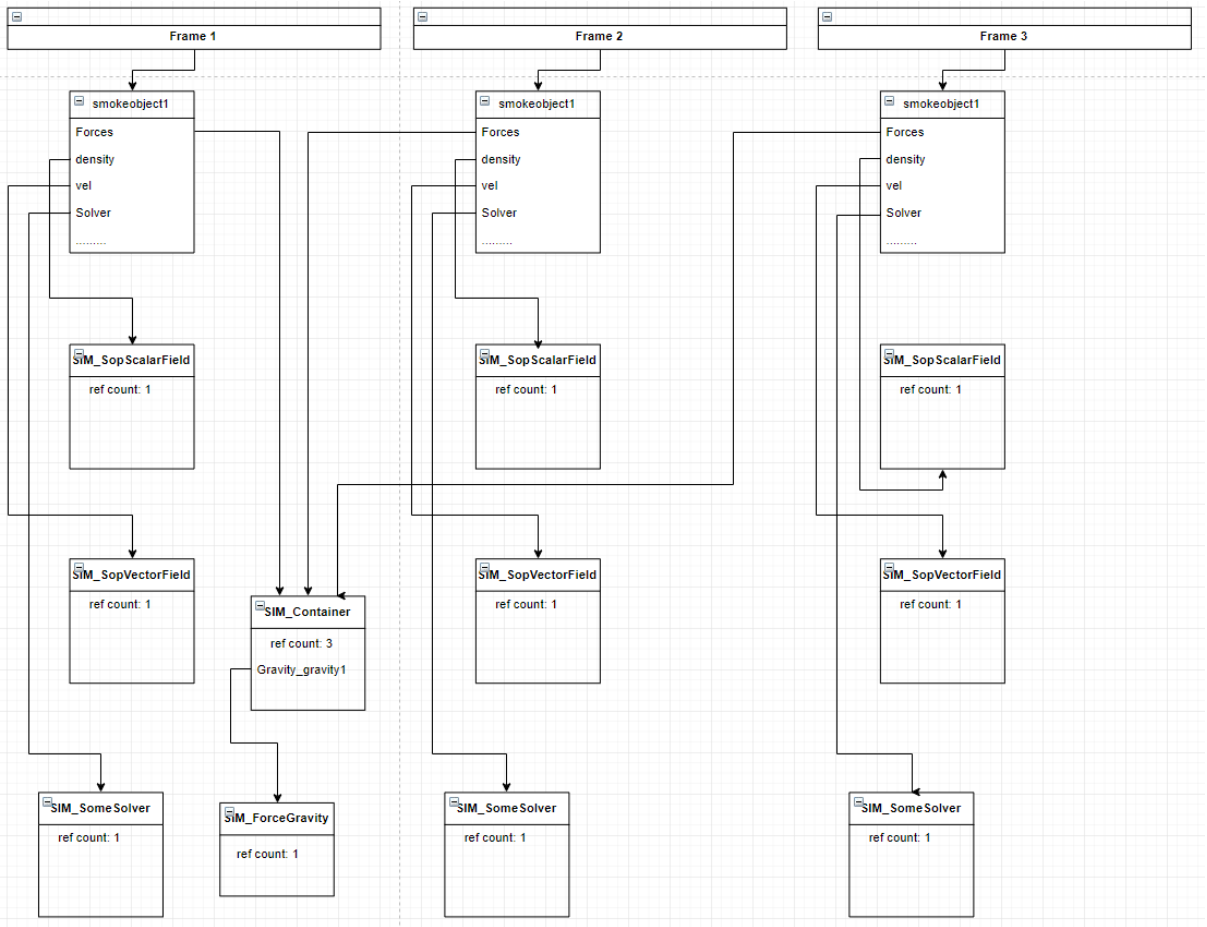


Now, at the next timestep, the following will happen: the object will be copied (cached simulation), preserving links to all sub-data from the previous frame

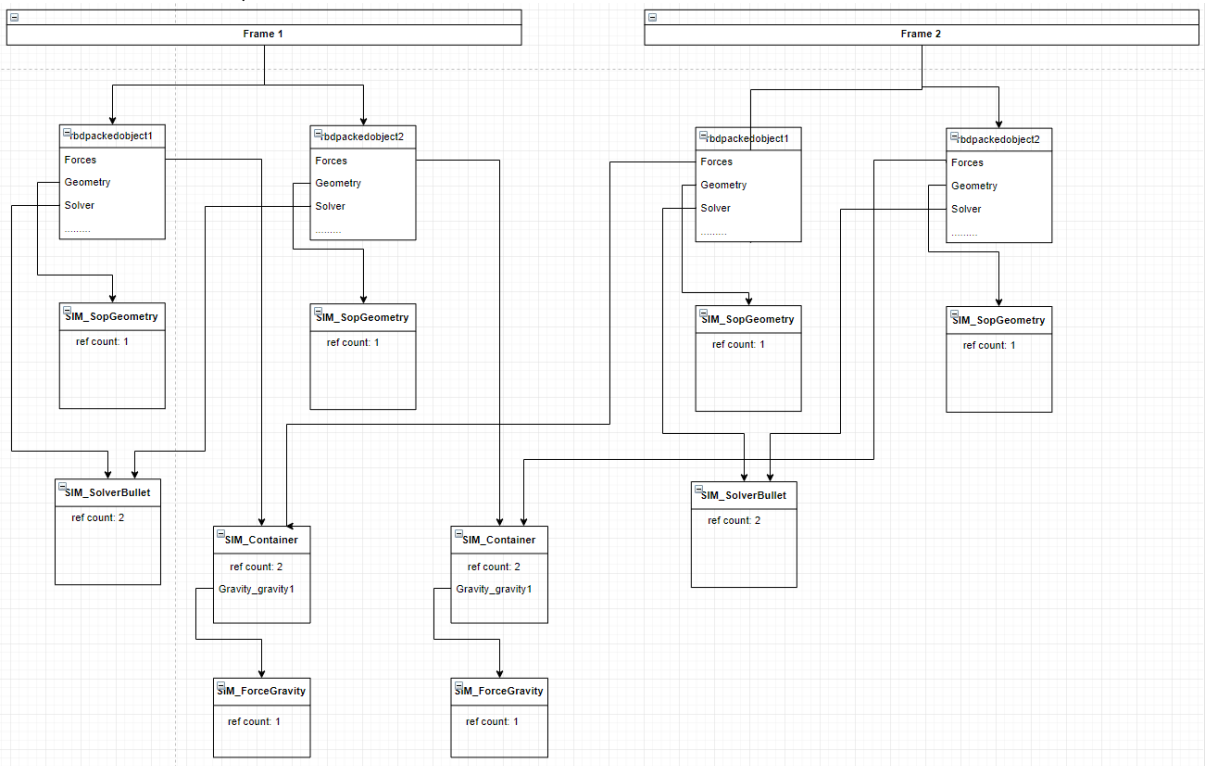


further, depending on the logic of the simulation, if something changes this data, for example, the smoke solver changes all the fields, the changed fields will be written to the new data, and the link to them will be updated on the object. (launching the functionality of the solver in the predominant number of cases (except for the statics of the solver, which does nothing at all) is considered a change in the data of the solver, therefore they are copied)

Data that has not changed, for example, the force of gravity, which we left constant by default, will remain the same, they will not be copied, the link to them will not change



In the case of one solver for several objects, as, for example, in the default setup with a bullet solver, the data scheme will look like this:



(Note that despite the fact that the same force joins the objects, in the default setup for each of the objects separate force data will still be created)

So, what are the data, what do they personify and do? It is difficult to answer this question, since in addition to some specific types of data *(for example, objects, solvers, relayships), the* simulator of the Houdini does not know and does not want to know what exactly is for the data, why and who needs it, it does not distinguish between them.
Data specializes in tasks by programmers. for example, for simulations of particles and RDBs (and for much more), geometric data (geometry) were written (in fact, this is a wrapper for ordinary SOP geometry), for simulation of gases and liquids - data for vector and scalar grids, all kinds of visualization data, Constrains, and, of course, hundreds of different primary and secondary solvers

Basic data types

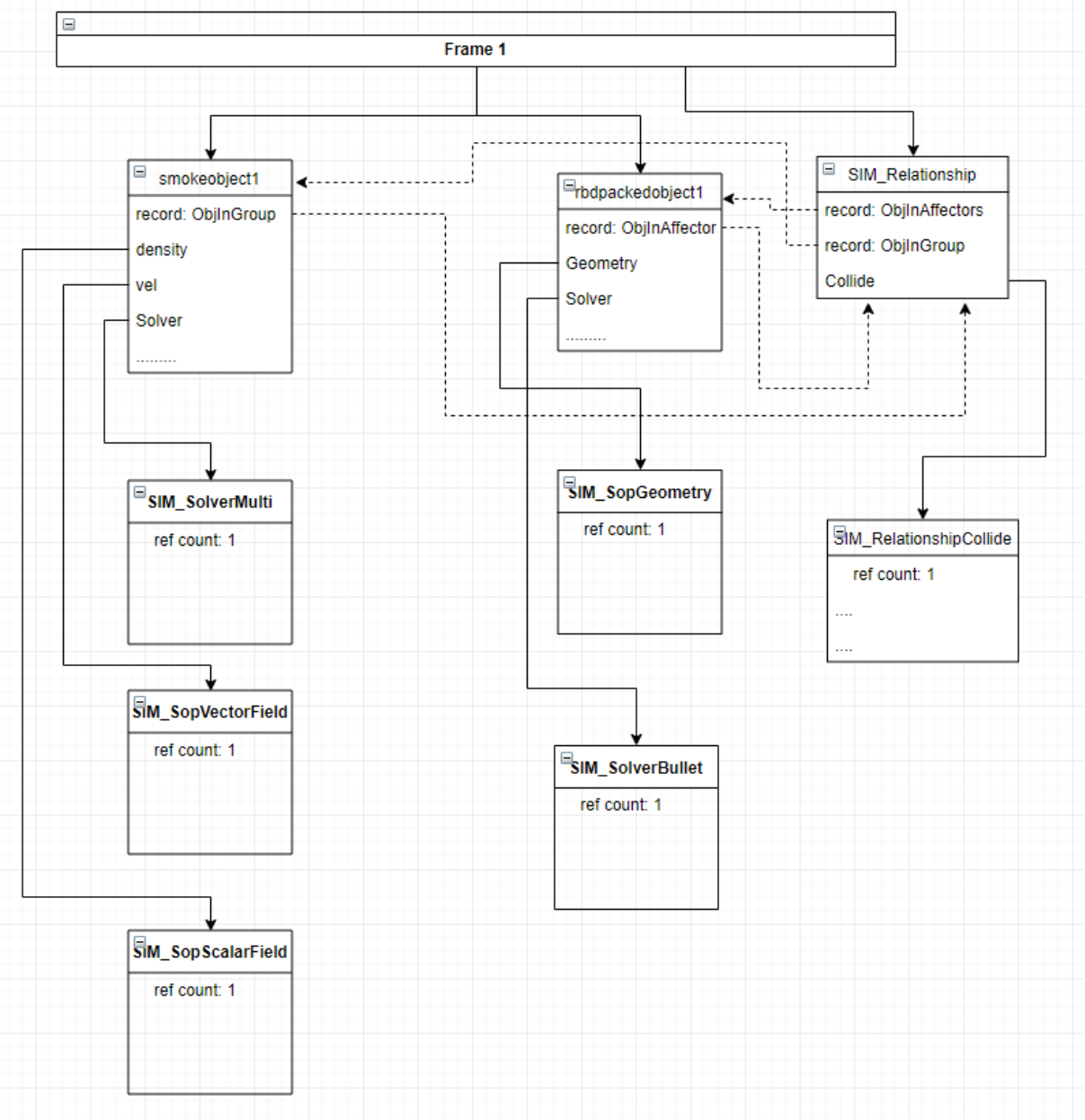
We'll talk separately about objects, solvers and relayships.
We briefly touched on the features of the objects:

The objects

Objects are a central data type. Only objects and relayships can be root, and exist not attached to other data. Therefore, an object can be considered as a frame on which data of various types are hung, and since all specialization goes into data types attached to an object, it makes no sense to specialize the object itself. Therefore , having looked at any simulation , you will see the same data type of the object everywhere .
An object belongs to one single timestep, so in a cached simulation, a Hoodini will copy objects every timestep, despite the fact that no changes have occurred on the object itself.
Houdini recognizes and uses the name field in their Basic record to display and search for an object by name, in addition, each Houdini timestep updates the lists of relays in which the object participates and in a special way processes the data with the name of the Solver link.

Relayships

Relayships are a special type of data that, like an object, can be root. The main goal of relayships is to tell the goodies what objects influence which so that during the execution of solvers the goodies can group objects and choose the correct order for their solving.
Relayship stores 2 lists of objects: those that affect and those that affect. As soon as objects are added to one of these lists, the Houdini tracks this and adds an entry to the object with the name of the given relay for the convenience of tracking. It is worth noting that the specializations of relayships, as well as objects, do not occur through inheritance of the relayship class itself, but through the attachment of specialized data, subclasses of auxiliary data of relayshipDates, which are attached to the data of the standard relayship and determine its behavior. For simplicity, in the future I will talk about the specialization of relayships, referring to the specialization of precisely the sub-data on relayships.
There are many specializations of relayships, for example: collide relayshire, sors relayship, pump, constant, group, etc. - their goal is to supplement the main functionality of relayships with special ones and to separate this type of relayships from others. So, for example, solvers can look to see if the object is precisely in the collide relay with another object, and get a list of these objects, and already react to it as he wants. A solver does not have to look for a collide relayship to find a list of collision objects, it is up to the programmer to write this solver - he can use at least pump relayship, write his own relayship, or not support relayships at all, but have a parameter with a list of object names.



P about the totality of all the Houdini relayships builds a dependency graph and decides which objects to solve in which order. (more detail on the procedure solvinga later) H ome representation of the dependencies of objects you can see in the Geometry Spreadsheet tab in the DOP simulation, pseudo plate Affector Matrix, where the color is marked as the object of the column affects the object from the line, place the cursor on a cell of the table you will see hint about it.

Solvers

Solvers are another special core data type. Data like Solver, in addition to actually storing information, has a functionality in itself that the Houdini knows how to run.
One of the stages of the simulation at each timestep is the solving process . G Udine, according to the construction of the graph depending on the objects (details will be in the section on Stage Solvinga) looking at each of these data with the name of Solver (yes, the search is on behalf of links to sub-data), and if these data are of type SIM_Solver or derivatives , then the Houdini simulation engine will launch the functional (we say “launches the solver”, “solvates the object”), programmed in this solver. What this functionality will do is one programmer knows, the possibilities are, in principle, unlimited . T raditsionno,

Solver fed the objects to which it is attached, whereupon he finds or creates the data he needs on them, and changes them.
So, for example, bullet and pop solvers will look for data with the name of the Geometry link on their objects, the smoke solver will look for a number of data, such as the scalar density field and vector field vel, etc. There is also a whole set of microsolvers and auxiliary solvers, which will be discussed in the chapter Solving Stage.

Less Important Base Data Types

Forces

Forces (force) are one of the frequently encountered data types. All forces have a convenient general functionality that allows you to request force to calculate the force used for a number of the most common situations, which makes force a convenient general mechanism for introducing physical forces into a variety of simulations.
solvers may request and use force for their calculations, but may not do so. just as they may follow the recommendations of the selected sampling mode, or they may not follow.

Visualizers

such as ScalarFieldVisualization, VectorFieldVisualization, ConstraintNetworkVisualization and, in general, many others, mostly having the word Visualization in the data type name. This data should usually be sub-data of the data being visualized, which is logical for visualizing individual data, however, composite visualizers visualizing the combined data will be located somewhere else, for example, on the same parent as the set of visualized data. All this is individual, and depends on the particular visualizer, since visualizers are united only by a common semantic goal, no strict rules.
DOP has a mechanism that allows arbitrary data to build mappings for the viewport, for example, using this mechanism, the geometry from SIM_Geometry is displayed in the viewport. However, visualization is often separated from the actual important data in order to conveniently and easily change the visualization at the request of the user or according to the idea of simulation, and not to overload the data carrying the payload with a bunch of additional repeating functionality.
So, for example, there is a scalar field SIM_SopScalarField, which is not displayed in the viewport, however, we can attach SIM_ScalarFieldVisualization data to it and visualize data from SIM_SopScalarField using different methods implemented in SIM_ScalarFieldVisualization. Or instead of individual visualization of the field, we can have one data SIM_MultiFieldVisualization, which, given the data from several fields at once, can display a more adequate data display in the viewport. for example, it would not be informative to look at density, temperature, heat separately, each rendered with its own visualizer, but it’s convenient for us to see multivisualization, where the density of the smoke is taken from density and heat, and temperature is responsible for color.

Empty Data

The simplest basic data type (SIM_Container is simpler, serving only as data for storing links to other data, not even having an options record)
It is convenient to use empty data to create your own fields and store arbitrary information in them. it can be fleets, thongs, arrays, which, for example, are needed for your special solver script

Now that we are more or less aware of the general principles of the DOP contest working with data, let's return to the actual simulation, and finally see how the

nodes correspond to the data.

General structure of simulation calculation

Simulation in a DOP context is an environment in which data exists and is transformed. The simulation is divided into steps, usually with a fixed time step (timestep), which can be set on the DOP simulation node; by default, the timestep is equal to one frame.
Each step of the DOP simulation takes place in 2 stages:

- Stage calculation of the node graph
- Solver calculation stage (solving)

It is important to remember that these two stages occur strictly sequentially and do not intersect, therefore, at the stage of calculating the nodes, no information is available for the Sloing of this timestep, even if the solver was created by the currently calculated node, just like at the time of solving, no intermediate stage of computing the nodes is available - they are all have already been calculated, all new data has been created, the order of solving objects has already been built and cannot be changed for the current timestep.
Consider these stages:

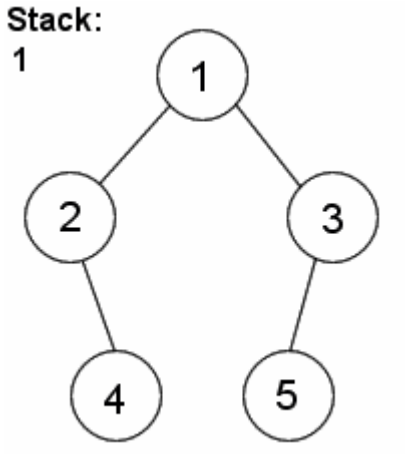
Node Graph Computation Stage

Further, speaking of nodes, we will separate objects from other types of data, and using the term “data” we will mean only non-object data and non-relayships, object data will be called just objects, and relayship data - relayships.
we will also consider the inputs of nodes from the **first** , **NOT from zero** . usually, the inputs of the nodes will be divided simply into the first and the rest.
The goal of DOP nodes is to create, delete, or modify objects, relayships, and other data in a DOP simulation.
It is worth noting that there are 2 types of additional node connections: the stream of objects, and the stream of data, they differ in color in the network editor.

Node Count Bypass

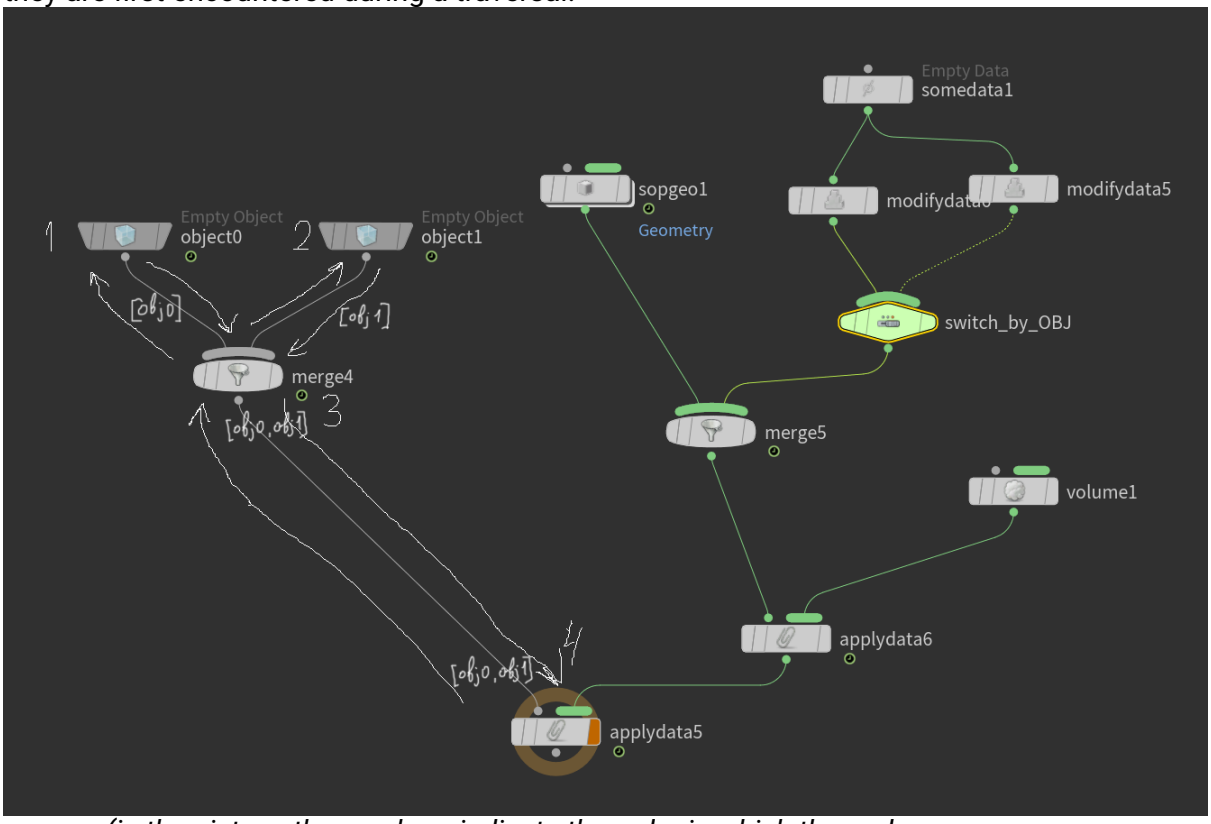
Attention, examples of the graph traversal stage contain scripted parameters on the nodes calculated at the graph traversal stage, however, if any of the nodes is selected in the viewport, its parameters will be additionally calculated for the display after the graph traversal, creating additional output to the console, which can be confusing . Therefore, always either deselect all the nodes during the test, or select the Output node (there are never any expressions on it), or watch the console output exclusively from outside the additional network

The calculation of DOP nodes occurs in a strict order, determined by traversing the graph of nodes by object connections in depth (from bottom to top in the hood) from the node marked with the Output flag.



(in the picture above, the node with the output flag is node 1, note that this graph is drawn "down" from the root, while the additional graph is drawn up from the root. The functionality of the node occurs when the node turns green)

However, there are a number of nodes that determine the structure of the graph itself, such as switch (**not** switch solver), merge, apply data, they are executed according to a special rule, not according to the rule of deep going, usually when they are first encountered during a traversal.



(in the picture, the numbers indicate the order in which the nodes are executed. Note that while the merge4 node's creation of the relay functionality will be the third, its activation parameter will be calculated at the first meeting of the node, i.e., before the function object0 is executed, as we already said, this the parameter defines the graph traversal structure itself, as well as switch nodes, if activatino evaluates to 0 - the graph traversal does not go to the object1 branch)

Through object connections, a list of links to simulation objects is transmitted from node to node (not the object data itself is "transmitted", but only links, the data itself exists outside the nodes, in the simulation itself) through data connections , a list of links to other, nonobjective data *usually* flows but their calculation is a little different

By "Execution" or "Work" of a node is meant the execution of the functionality laid down in the node on the list of objects supplied to it, directly (list of the stream of

objects) or indirectly (current data, local list of data (see data stream below))

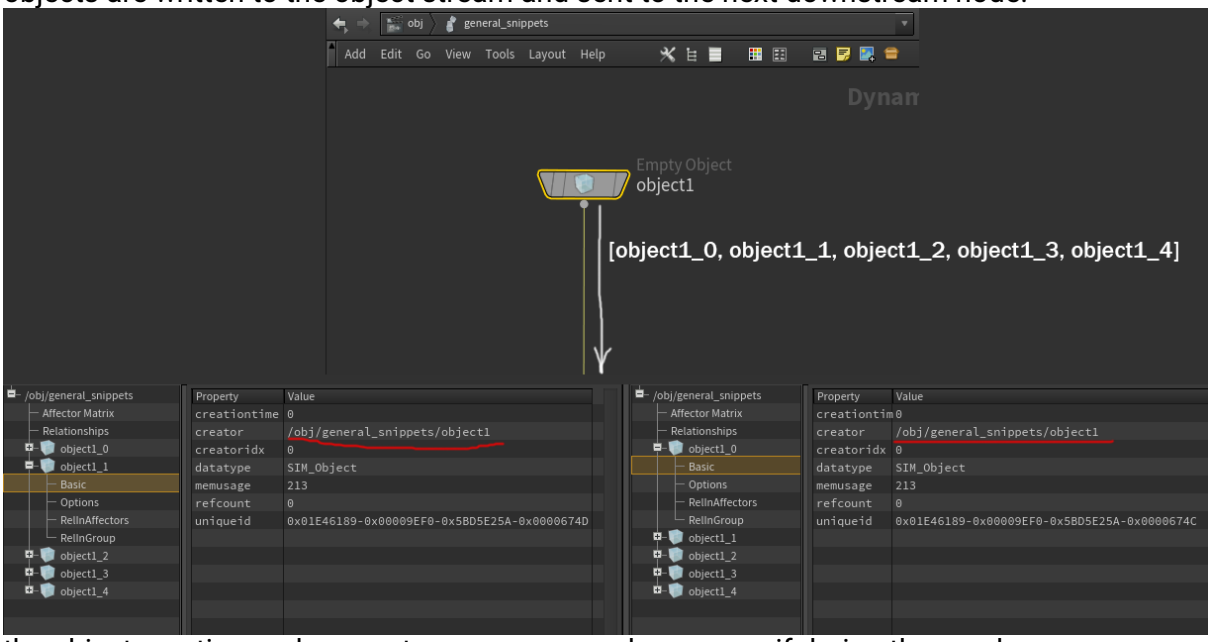
Stream of objects

First, let's talk about the object stream exclusively.
You may notice that any chain of object connections starts with the empty object node.

The object node (empty object) calculates its activation parameter, and if it is not zero, it repeats the number of times equal to the number of objects parameter, does the following: creates a new object, calculates the object name parameter and the remaining 2 daws, and applies these parameters to the created one object. Note that for newly created objects in the Basic record, the path to the given node that created the object is written in the creator field - this is one of the key ways how data is bound to the nodes .
Next, all objects in the simulation that belong to the previous timestep are scanned , and those whose creator field matches the path of the given empty object node are copied with all their links to sub-data (according to the CoW principle described earlier) into the current timestep (as mentioned earlier, each the instance of the object data belongs to one specific timestep, so “copying to the current timestep” means copying an object and writing a link to it in the current timestep). It is the empty object node that is responsible for copying objects from the past timestep to the current one, the process that was mentioned earlier.
If the node registered with an object in the creator field of the Basic record was not reached bypassing the node graph in the current timestep, this object will not be copied to the current timestep from the previous one.

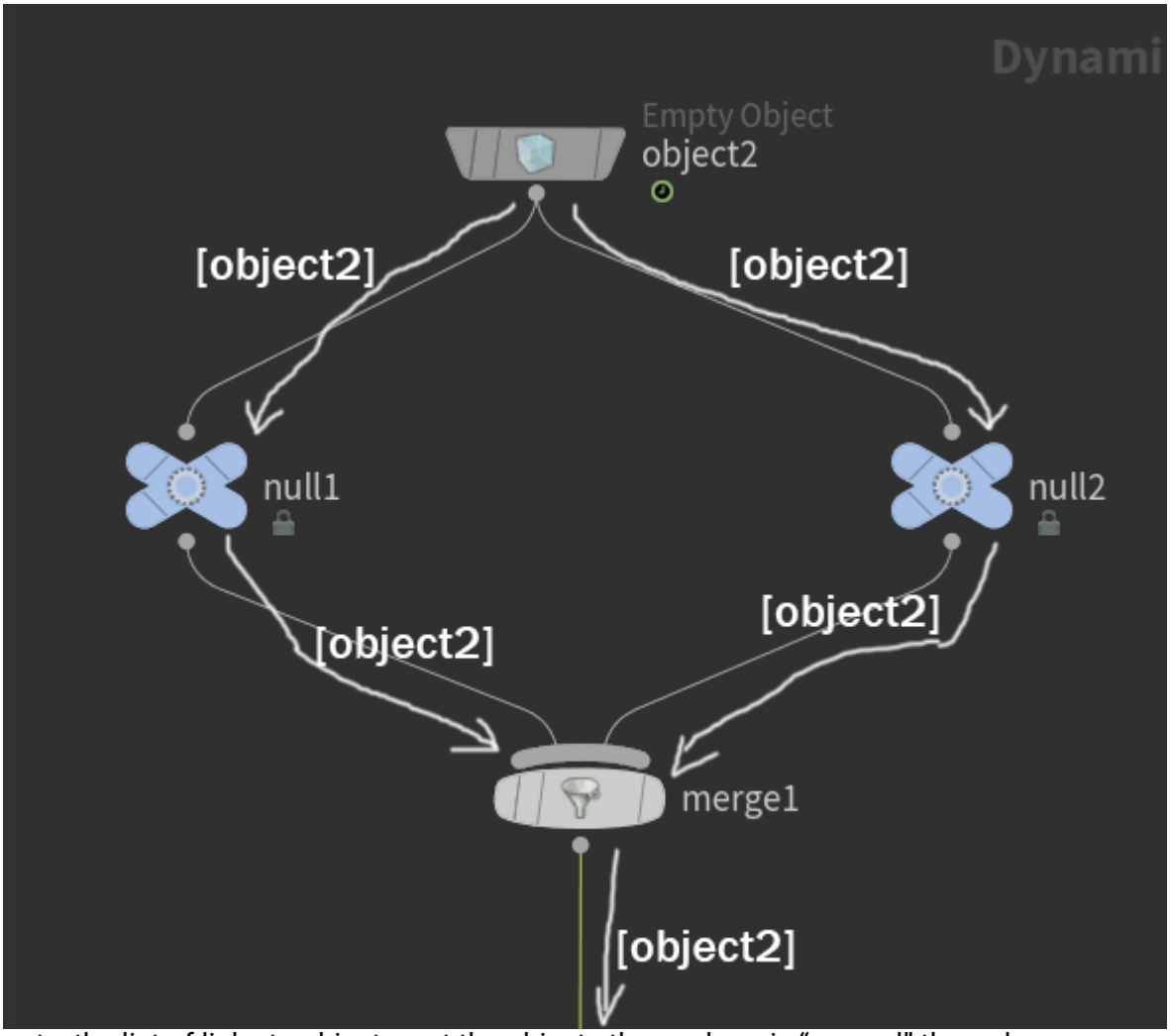
(example: **sppehewuqe @ HPaste** *(OBJ context))*

Now links to the objects copied from the previous timestep and newly created objects are written to the object stream and sent to the next downstream node.



the object creation node executes once per graph passage, if during the graph traversal we come to this node again - it will skip the stage of calculating the parameters and creating objects and immediately return the list of objects, the

same one that it returned earlier in another connection



note, the list of links to objects, not the objects themselves, is “passed” through the object stream, so the object is not duplicated in the screenshot above, duplicate links in the list will be merged, so below merge1 object2 will NOT be processed twice.

Further along the object stream, nodes such as:
auxiliary: zero, merge, switch
data modification nodes: modify data, delete
data nodes or apply data

we will analyze in order.

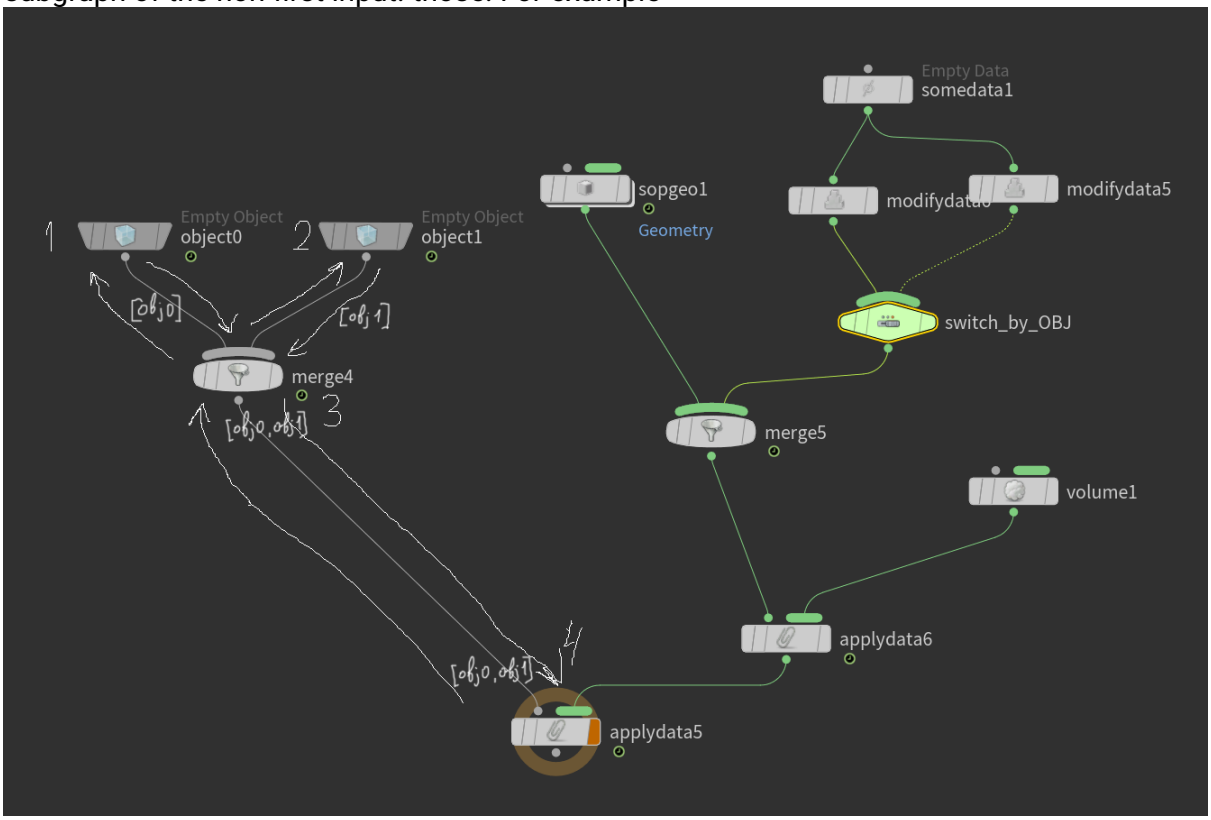
- The switch node (not to be confused with the switch solver) serves only to determine the structure of the bypassed graph, their parameters are calculated at the first meeting (i.e., when moving from the bottom up), determining how the graph will go around further. except for this, the node does not affect the simulation in any way.
- Noda Null (null) - does nothing
- Noda merge (merge) - combines the lists of objects that go into it (duplicate entries are excluded), also, for the sake of convenience, it includes the ability to create relayships between object flows that enter it: in mutual mode, a relayship with a given type will be created, including all input objects as affectors , and in affecting lists (all objects mutually influence each other), in the left inputs affect right mode one or more relayships will be created, reflecting that all objects from the first input of the merge affect the objects of all inputs to the right, then the objects of the second input and merzh Affect objects of all inputs to the right of the second, and so on. The activation parameter of the merge will also be calculated at the first meeting (i.e., when going from the bottom up), and if the

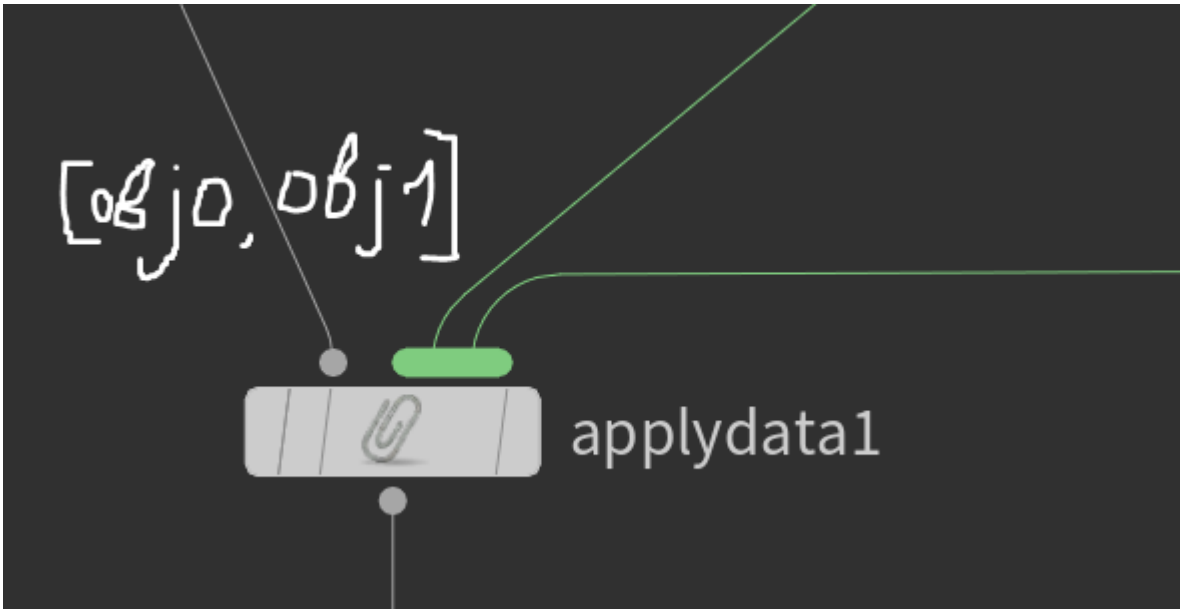
parameter is calculated to zero, the node will not do anything, it will only transfer the list of objects from the first input further down,

● Nodes of data modification work on the same principle: for each of the objects, one of the input list will calculate the parameters of the node and perform the corresponding operations determined by the node. For example, for modify data, it is to check whether the object falls into the group, find the given data on it, on which to make a series of modifications with the fields in the Options record, for delete - delete the given data from the object, or remove the object from the simulation (respectively from the list of the object stream, it will also be deleted, of course) (note, only a copy of the object / data of the current timestep is deleted, with copies of the object belonging to other timesteps everything will be okay)

● **apply data** is a very special node in the graph structure. it may seem that it has more than one input, according to this rule of depth going around, you must first get around and execute a subgraph of the first input, then the second, etc., and only then the apply data node itself should be executed. But this is not so. in fact, the apply data node connects several independent graphs of the nodes: the graph of the first input, and each individual graph from the rest of the inputs. It is only part of the graph from its first input, all the graphs of the other inputs are, as it were, the parameters of the node. So as soon as the bypass of the first entry subgraph is completed, and we return to the apply data node, it will be executed. in the previous example - node applydata5 will really performed fourth in a row, sub-boxes, starting with node applydata6 is so to say the parameter node applydata5, and calculating it is part of the executable functional nodes applydaya5

functional node type apply data is to perform each of its non-primary subgraphs for each DATA at the first entrance (more will be later). Those. in the picture above, 2 objects come in the list by the first input, so one of its non-primary subgraphs starting with applydata6 will be executed 2 times, once for each of the objects. It is also worth noting that the activation attribute of the apply data node is calculated for each object / data from the first input list, for each subgraph of the non-first input. those. For example





the activation parameter of the applydata1 node will be calculated 4 times: for each of the objects obj0 and obj1 for each of the second and third inputs.
We only note that the order of the subgraphs is as follows: each graph of the non-first input is executed for each object / data of the first input. those. in the example above:

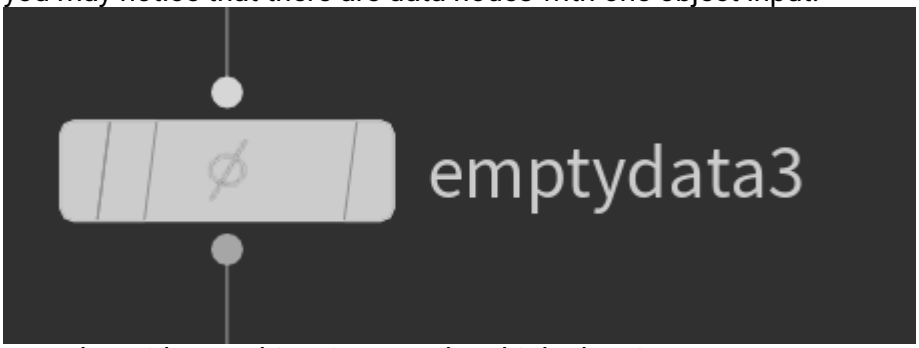
1) subgraph of input 2 for obj0
2) subgraph of input 2 for obj1
3) subgraph of input 3 for obj0
4) subgraph of input 3 for obj1

more details on how the apply data will work will be discussed later in the data stream

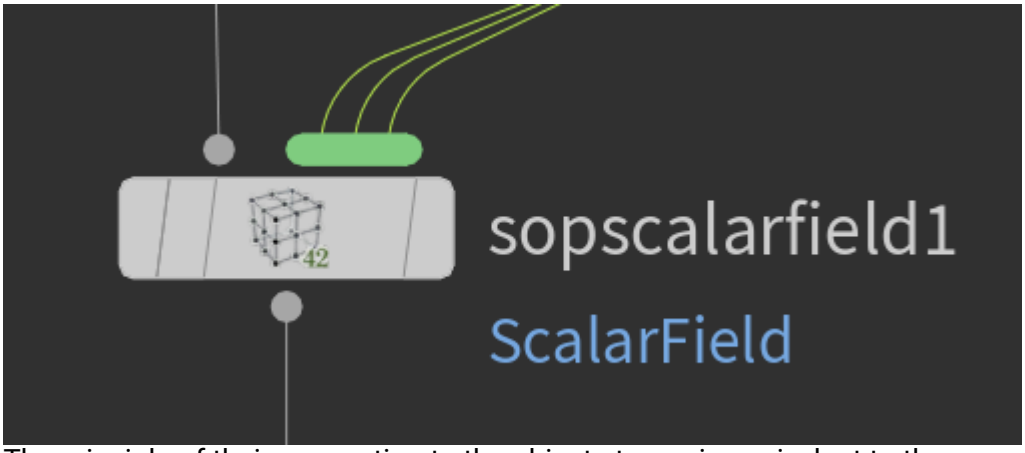
(example: **sppodazeki @ HPaste** *(OBJ context))*

Data nodes

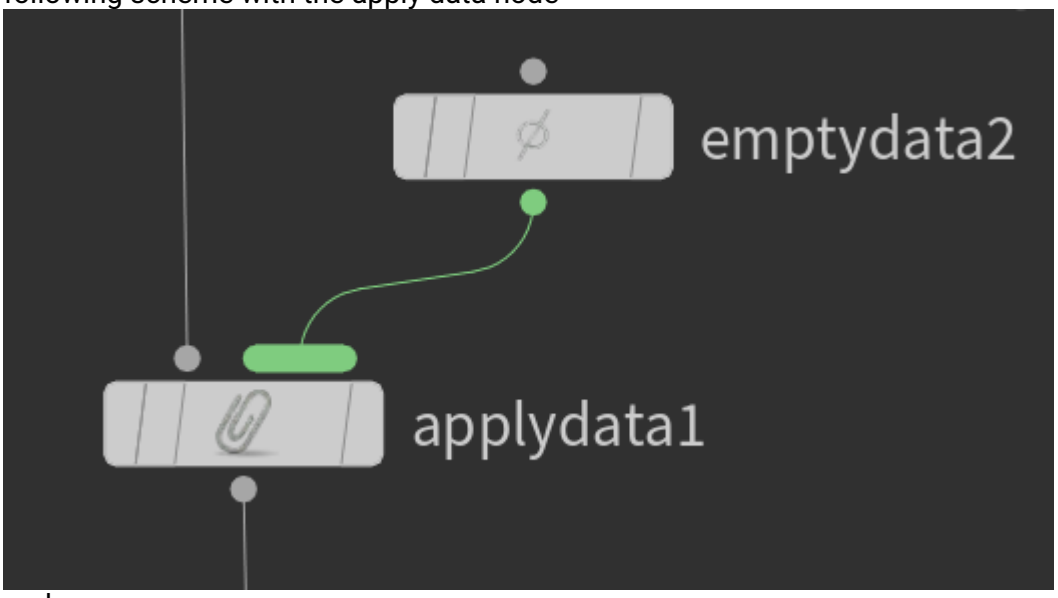
As a rule (this is not a strict DOP rule, but rather a guideline) for each data type, there is a node whose job is to create this data type.
(e.g. Sop Geometry, Sop Vector Field, Bullet Data, all solvers). These nodes can be included directly in the object stream, or connected via the apply data node - both of these methods are simply different options for recording the same process.
you may notice that there are data nodes with one object input:



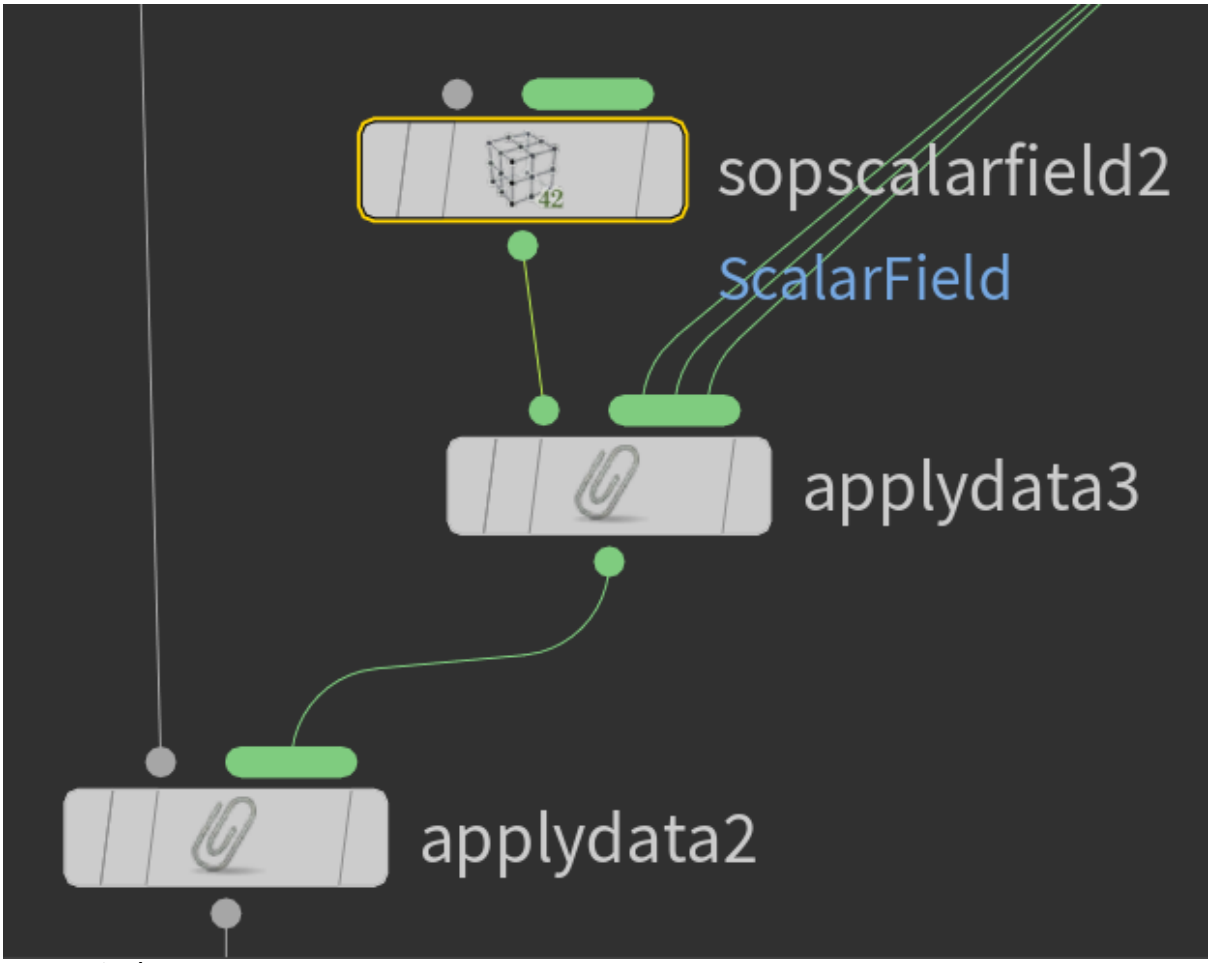
or nodes with one object input and multiple data input



The principle of their connection to the object stream is equivalent to the following scheme with the apply data node



and



respectively.
We will not pay special attention to the multiple input of data on data nodes, because they can be brought to the general scheme through apply data, as shown in the picture. (The only difference between connecting data through apply data in the last and pre-pre-last pictures is that in the case of pre-pre-last, the data can tell the graph crawler what types of subjects they expect to see on themselves, and give out a vorning in case of an unexpected type sub-data, however, this will not affect the process of the graph and the structure of the generated data)

Data stream

We will consider only the method of attaching data through the apply data node in view of the equivalence of the methods of attachment (described just above).

I tried to come up with the easiest way to describe the process of calculating and adding data, while minimally sacrificing details, so that the further description will be somewhat overcomplicated.
Unfortunately, in the process of calculating the node graph of the DOP context, there are a number of special cases that make a simple and clear description of its work in detail difficult.

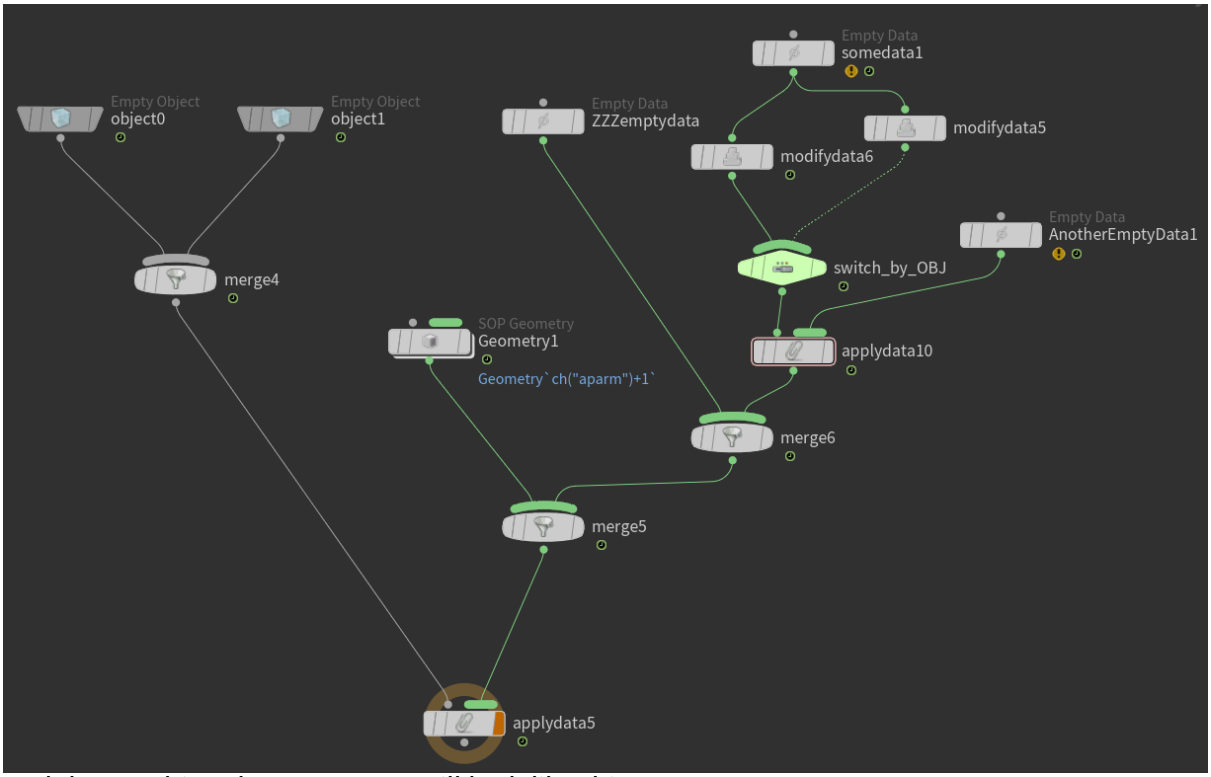
So, the apply data node (both the object stream and the data stream can enter its first input) works in a similar way to the copy stamp node in the SOP context: the subgraphs included in the data inputs are calculated consecutively for each data or objects coming into list from the first entry, sequentially. That is, from the list of objects or data coming to the first input, one by one the next element in order is selected. We will say that this next element (data or object) is set as the current parent data for calculating data subgraphs (not the first input) on the apply data node.
If the current data is an object, then, as usual for an object stream, local variables $ OBJ $ OBJID $ OBJNAME are set (set "globally", just like for any other

node in the object stream) and other variables associated with the object. we will say that the reference to the object is saved as the current object , that is, the object will be both current data and the current object .
If the current data is non - object data, nothing but the setting of the current data will occur, that is, the set local variables of the type $ OBJ $ OBJID $ OBNAME and the current object will remain untouched.
Once again, for each calculation of each subgraph, you will be given:

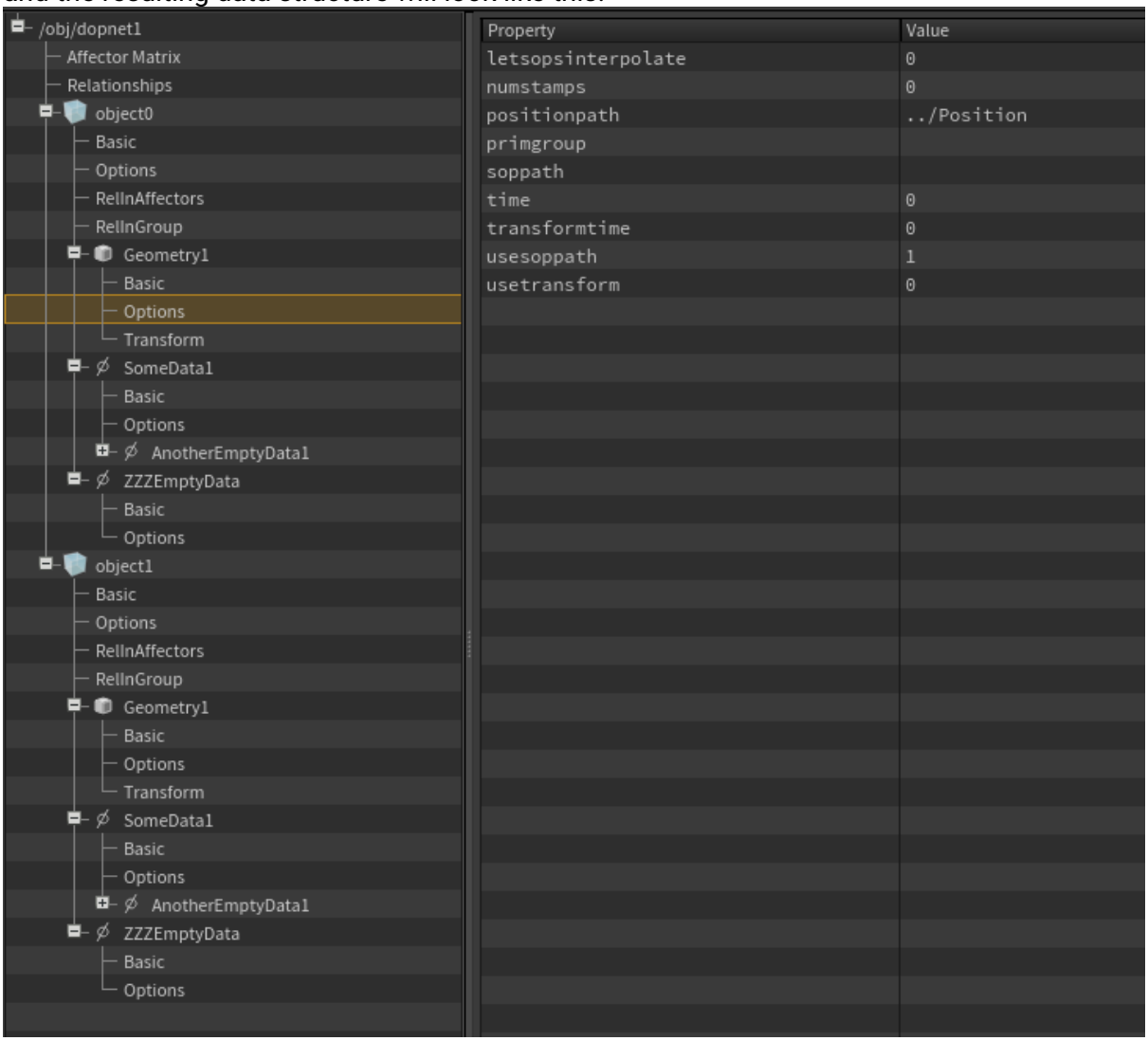
- current data
- local variables associated with the object ($ OBJ $ OBJID $ OBJNAME $ OBJCT $ OBJCF)
- the current object is initially equal to the current data, but not changed by subsequent apply data nodes in the data stream.

You can also combine the last 2 points, meaning that when traversing non-primary subgraphs, information about the object for which the subgraph of the data stream is calculated is always stored.
Note: In this section we will not talk about attaching data to relayships, so as not to confuse the already confusing explanation. in general, the logic will be the same, but other local variables will be set for relayships, and the current object will not be set.

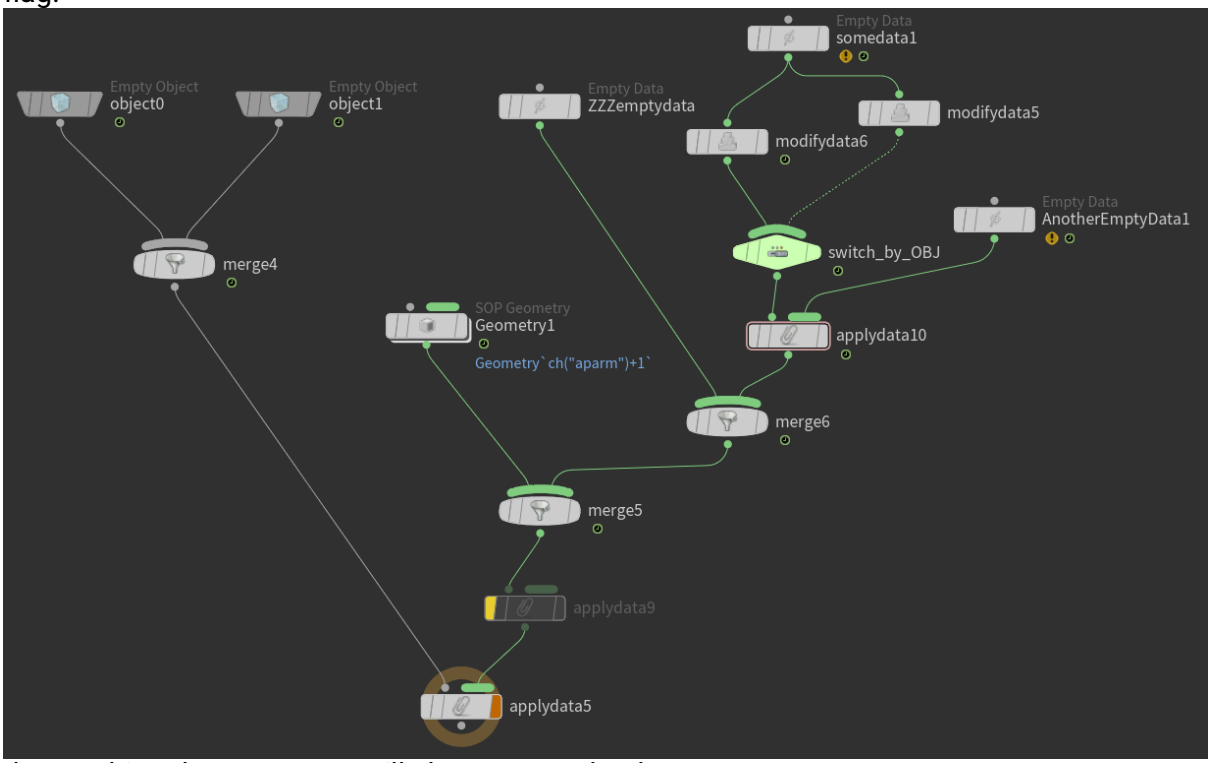
T EPER with given current data and the rest of the information, all nodes in the subgraphs of the non-first input apply data are calculated in depth, in the same way as the bypass of the object stream was calculated, also with the calculation of all switches along the way (when passing from the bottom up). Further, during a recursive walk in depth, there are 2 options for graph behavior, very different. My personal opinion is that this is a bug, and no one has been fixing it for years, because such complex structures in the DOP graph are rare, and can always be converted into simpler ones. So, if one more apply data node is not encountered when traversing a subgraph, then the calculation takes place in a similar way to the object part by a method: the graph is traversed in depth, all switch nodes calculate their input the first time they meet the graph, the nodes are executed on the data list coming into them,the current data as sub-data with the desired link name is added to the input list and transferred to the next nodes downstream, the merge node combines link lists from its flows (no relayships are created and cannot be for non-object data). We will call this option work - Normal
However, if another apply data node is encountered recursively going deeper into the stream, calculating the graph of **its first entry** will go in a very unexpected way. in this case, the calculation of the second and subsequent inputs will occur in the same way as in Normal mode . But what about the first entry: the execution order remains the same, but the data stream completely breaks now, despite the branching structure of the graph data connections, there is no data stream at all. instead, the data list is common to all subgraph nodes from now on. Each subgraph node is executed on this general list, and links to newly created data are added to this list. This list exists at the time of one crawl of the subgraph, the next crawl the list will be created anew, we will call this list -a local list of subgraph data , and this option is Broken . because of this brokenness, one can stumble upon very unobvious, illogical, and unexpected results of calculating a graph. eg: with this structure of nodes, the data subgraph will be calculated by the Normal method:



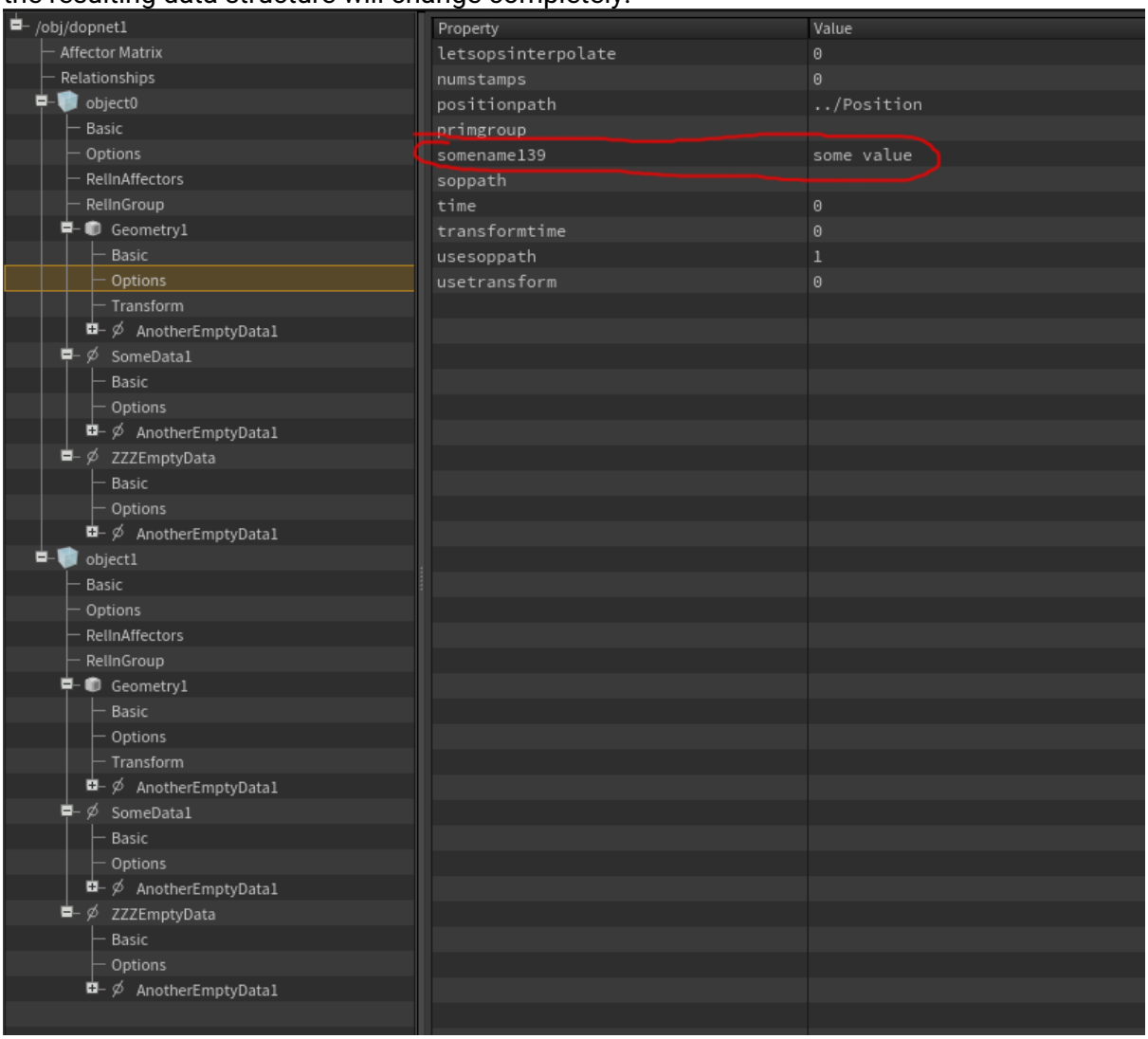
and the resulting data structure will look like this:



However, if we include an apply data node in the graph, even if it has a bypass flag:



the resulting data structure will change completely:



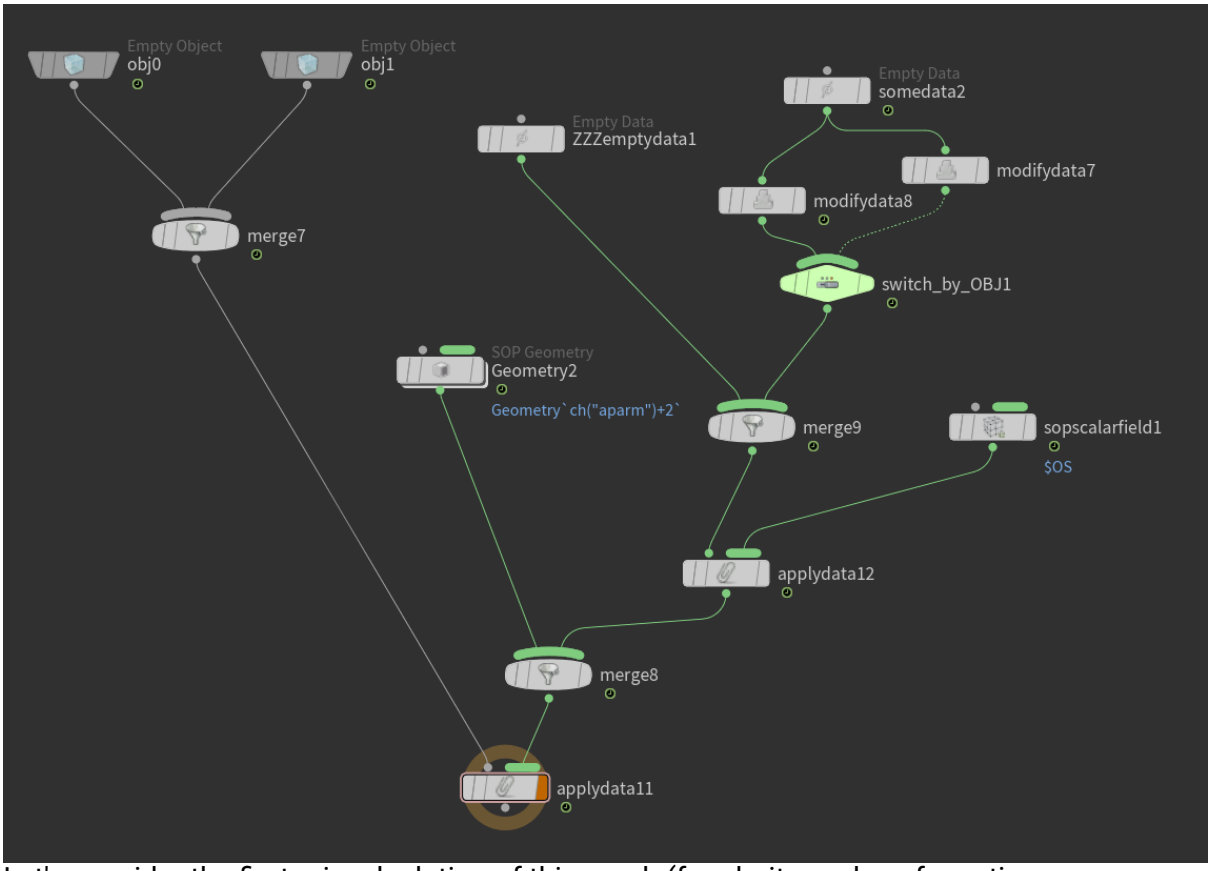
note that the data with the link name AnotherEmptyData1 was created on all the data, and the modifydata6 node unexpectedly changed the record field on the Geometry1 data (somename139 = "some value")

(example spptidamop @ HPaste (OBJ context))

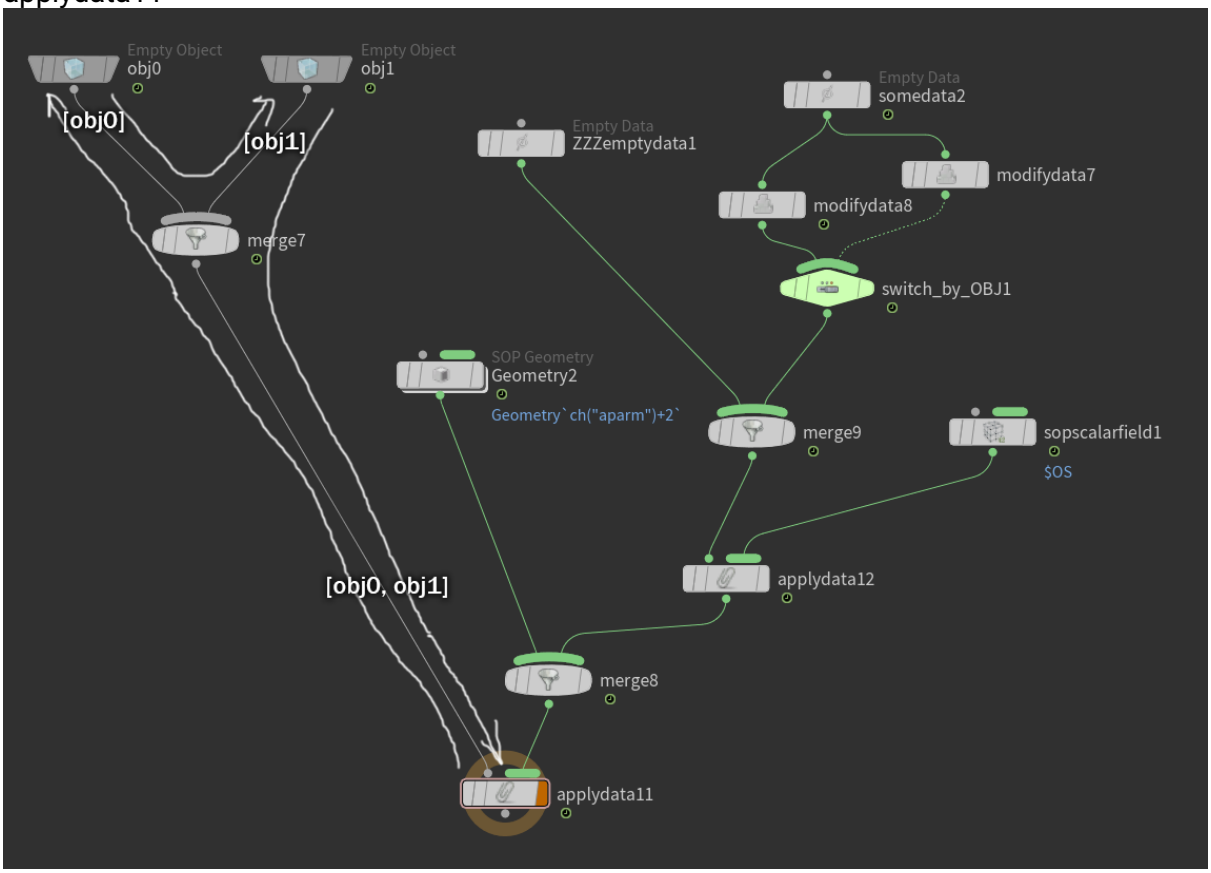
From the general implementation of data nodes:
Each data node, in order of going deeper, will first calculate its main parameters, such as activation, group and data name, and generally decide whether it needs to work, whether the current object is suitable for the given group (if not, the node is simply skipped), if there is no sub-data on the current data with the name calculated by the node, then new data is created and added to the current data as sub-data, with the calculated name of the link.
If the link with the same name already existed on the current data , then the data from this link will be transformed into the data type created by this node, instead of creating new data. After that, the remaining parameters from the node will be applied to this data, as before. (about the transformation process later). a link to the created / transformed data will be added to the data stream in Normal operation mode, or to the local data list of the current calculation of the subgraph in Broken mode.

Nodes such as modify data in the data stream will be applied to each data in the incoming list in Normal mode, or to data in the local list of the current calculation of the subgraph in Broken mode.
merge node - combines data streams (without repetitions) in Normal mode, and does nothing at all in Broken mode ; there it serves only to determine the structure of the graph. The relayship field on it is also inactive in the data stream, because relayships make sense only for objects.
If the apply data node is encountered on the way, exactly the same thing as described earlier will happen: the current data at the time before entering the apply data will be remembered and delayed, now every data from the local list *(if we went down the graph we came across apply data node, entering its first input, we already know in the Broken mode of operation)* will be set as current data (since this is not an object, the variables $ OBJ $ OBJID $ OBJNAME etc. will not be changed), and all subgraphs will be calculated for them , from left to right, looking in depth, recursively, according to the same principle that we describe, ince in Normal mode. For each calculation of each subgraph. Upon completion, the apply data nodes stored the current data will be restored before the processing of apply data, and the calculation of the graph will continue.

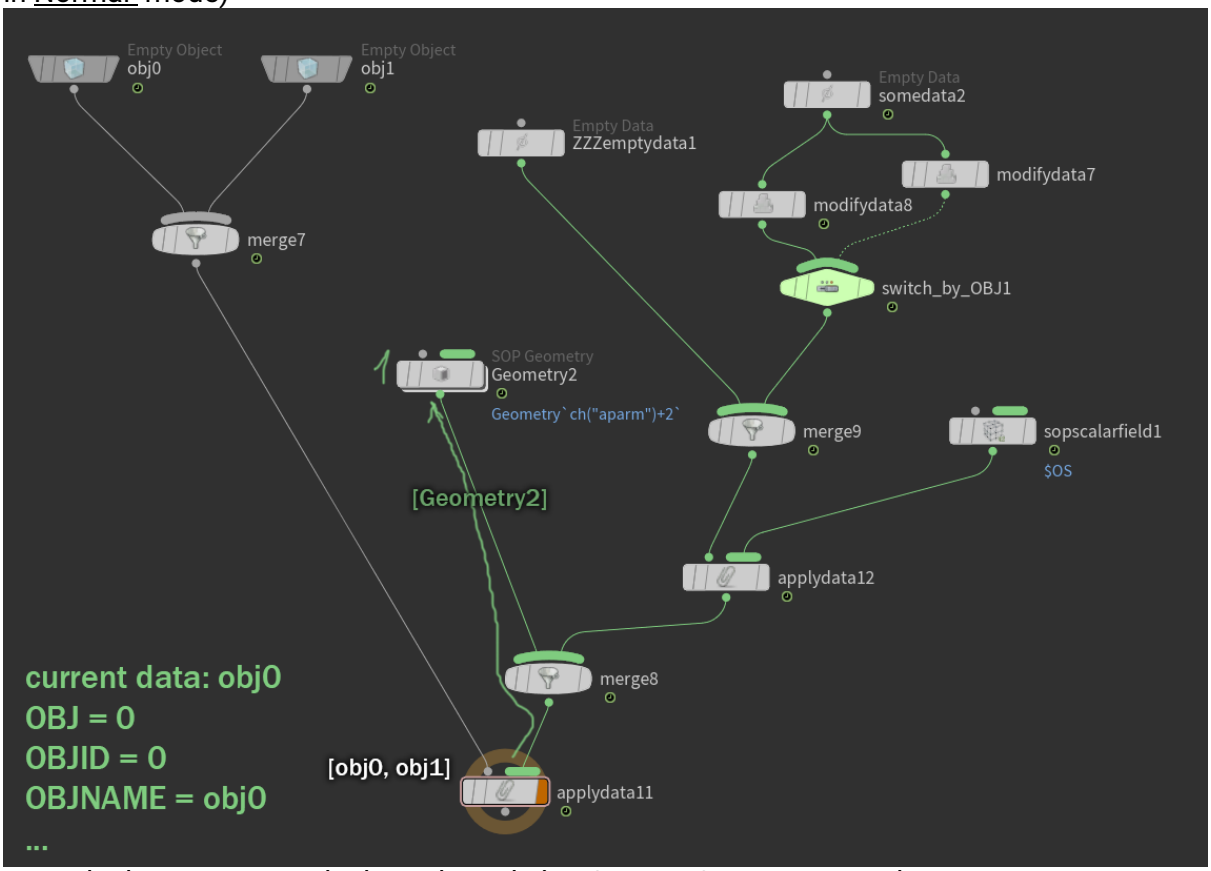
Consider an example of traversing a node graph:
We have not yet talked about the Data Sharing parameter, which is present on all data nodes, so far we assume it is always in the default value.



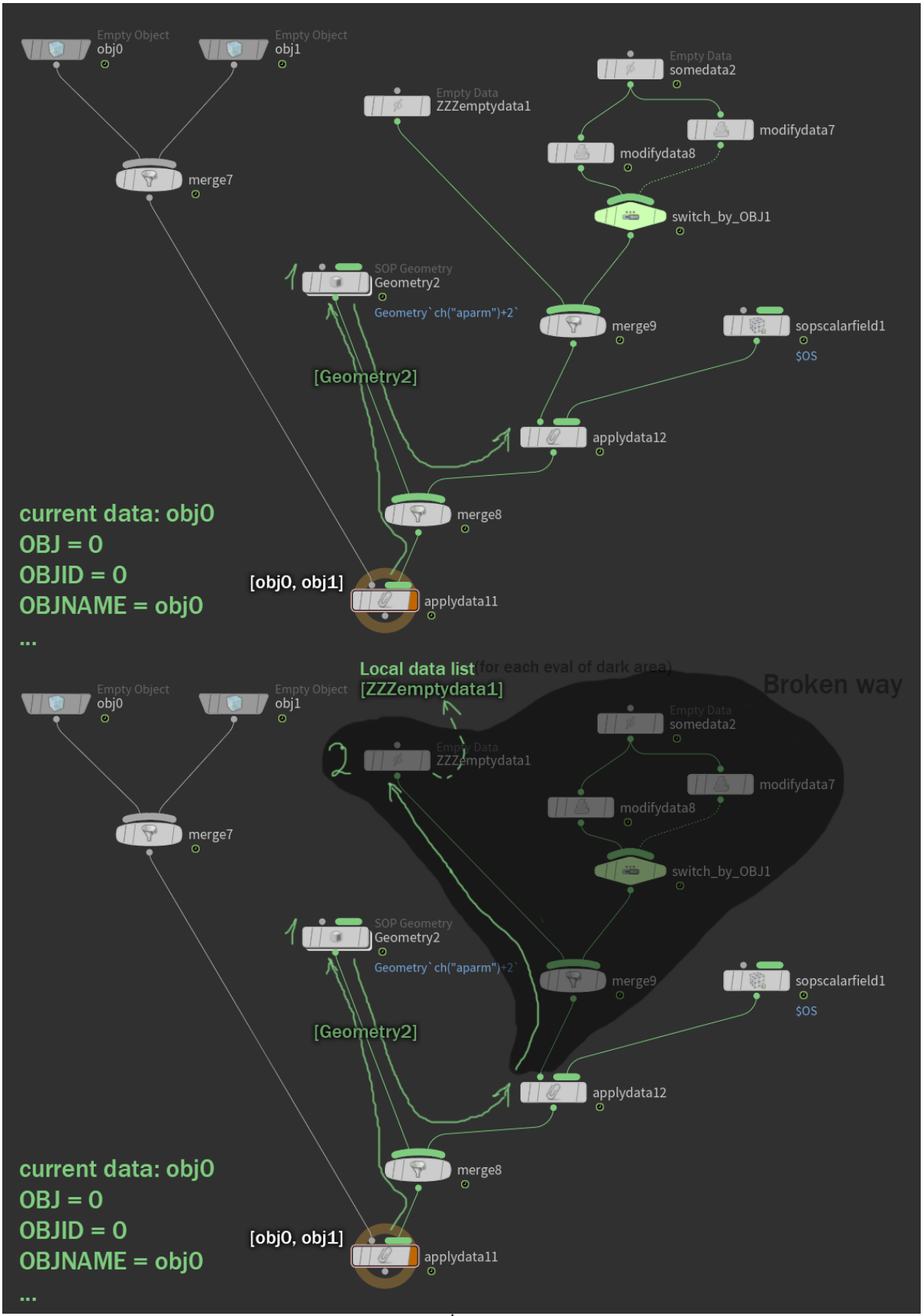
Let's consider the first miscalculation of this graph (for clarity, nodes of creating objects and data create objects and links to data with the same names as the nodes themselves)
First, the obj0 and obj1 nodes will create new objects and pass references to them, which merge7 will combine into a list of two objects and pass down to applydata11



Now, for each of the objects obj0 and obj1, all subgraphs of not the first applydata11 input will be calculated (only one in this case). The current data will be set first on the obj0 object, the variables $ OBJ * and other associated with the object will be set in accordance with the current object. First, the Geometry2 node is calculated, creates data of the SIM_SopGeometry type and immediately attaches a link to them with the name Geometry2 to the current obj0 data, the data link will also be sent downstream of the graph (since the graph is calculated in Normal mode)

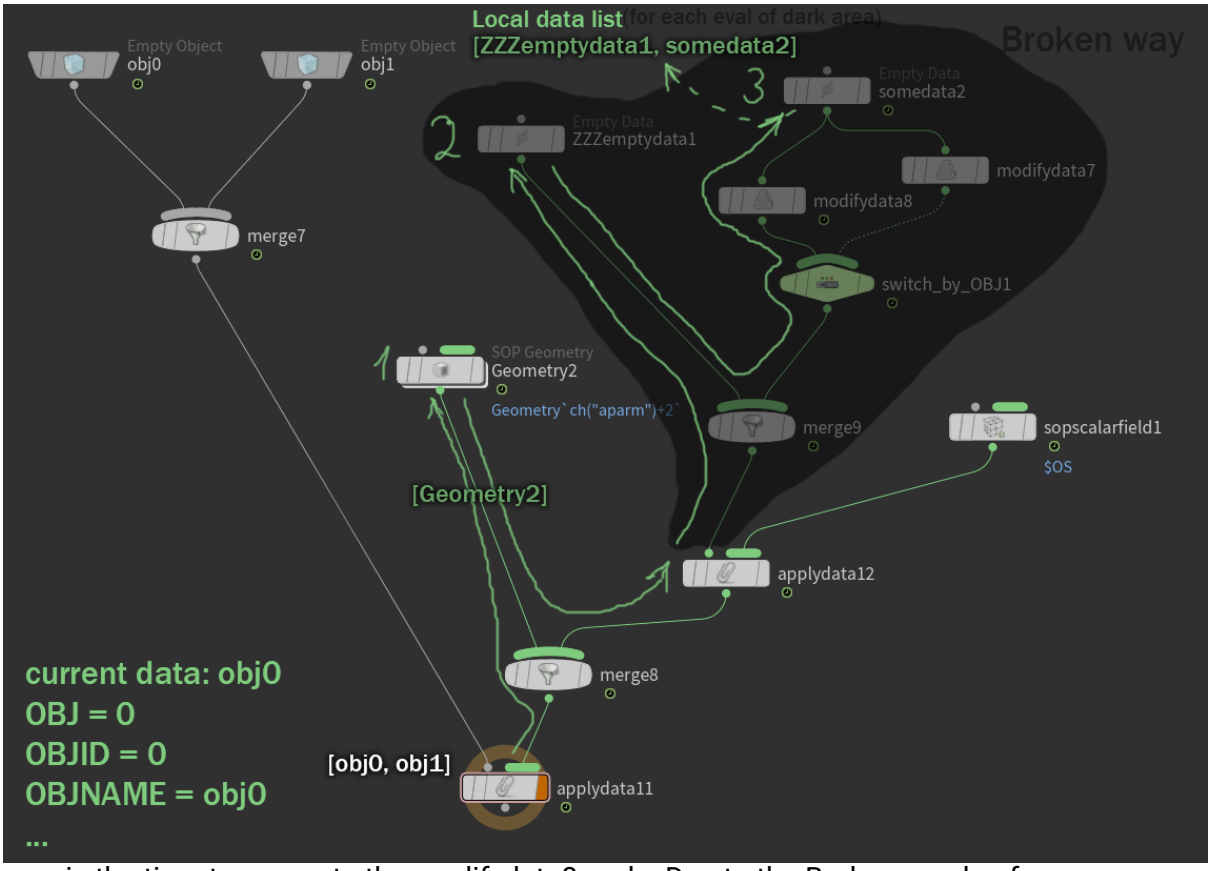


Next, the bypass proceeds through applydata2 merge9 to ZZZemptydata1.
Note further that the dark subgraph is included in the first input of the apply data node in the data stream, which means that its calculation will take place in Broken mode, however, the procedure for calculating the nodes will remain the same.
The ZZZemptydata1 node will create data of the SIM_EmptyData type and immediately attach a link to them to the current data (still an obj0 object).
However, the link to this data will not be transferred downstream due to the Broken mode, it will be added to the local data list of the current calculation of the broken subgraph)

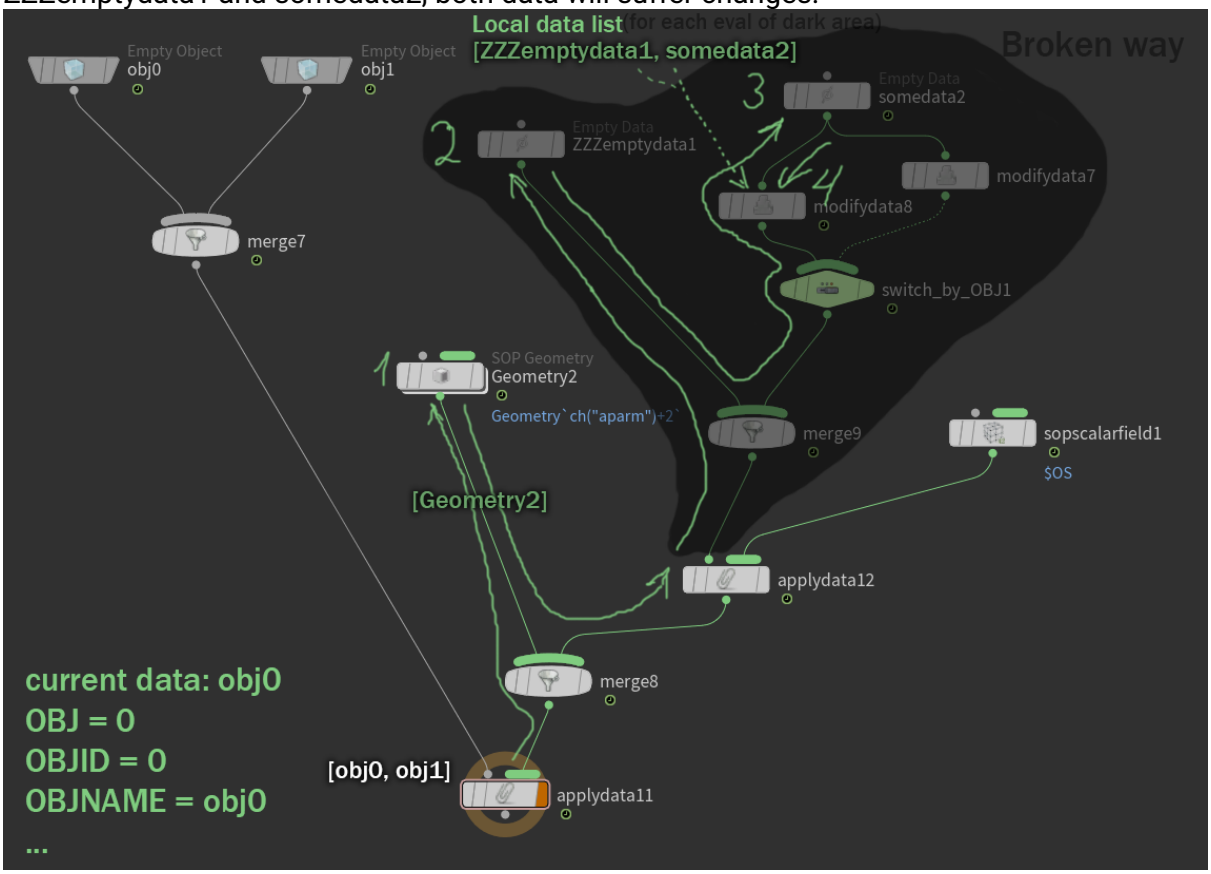


The switch_by_OBJ1 switch has the variable $ OBJ in the value, so as soon as it is met as the graph is crawled (at the first meeting for each crawl), the value of $ OBJ will be calculated to 0, and this will direct the further crawl along the left branch.

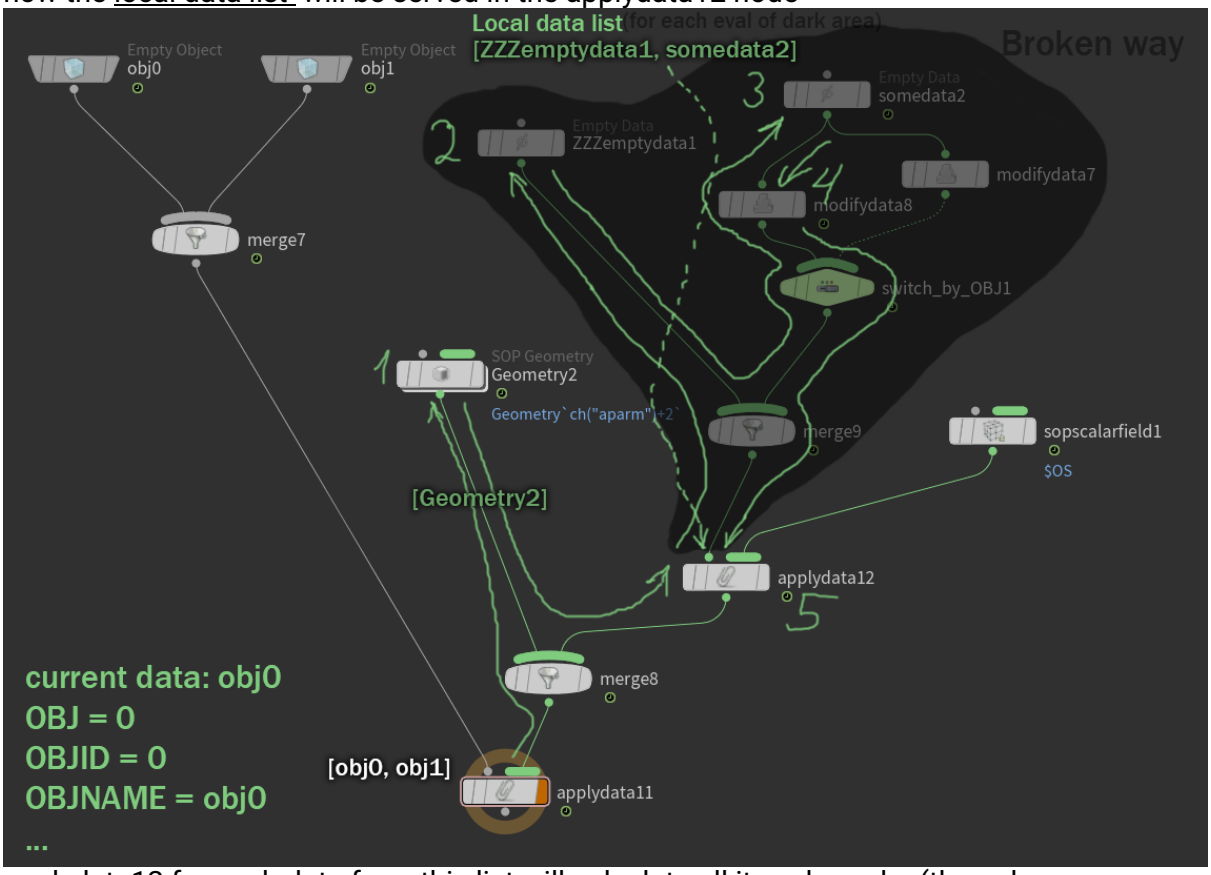
Now the somedata2 node is executed - it will create data again of the SIM_EmptyData type, a link to them under the name somedata2 will be added to the current data (obj0), and to the local data list



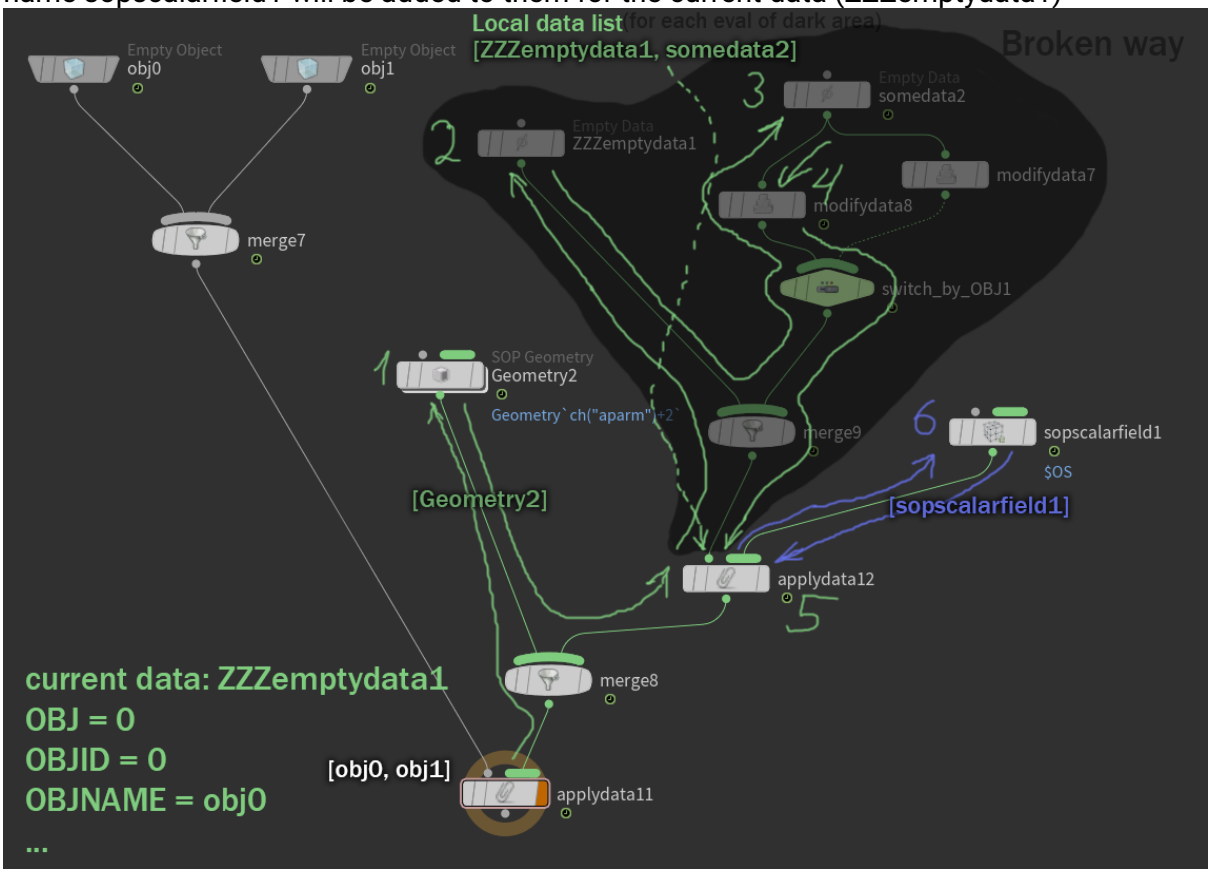
now is the time to compute the modifydata8 node. Due to the Broken mode of operation, the modifydata8 node will work on the entire local list, on both ZZZemptydata1 and somedata2, both data will suffer changes.



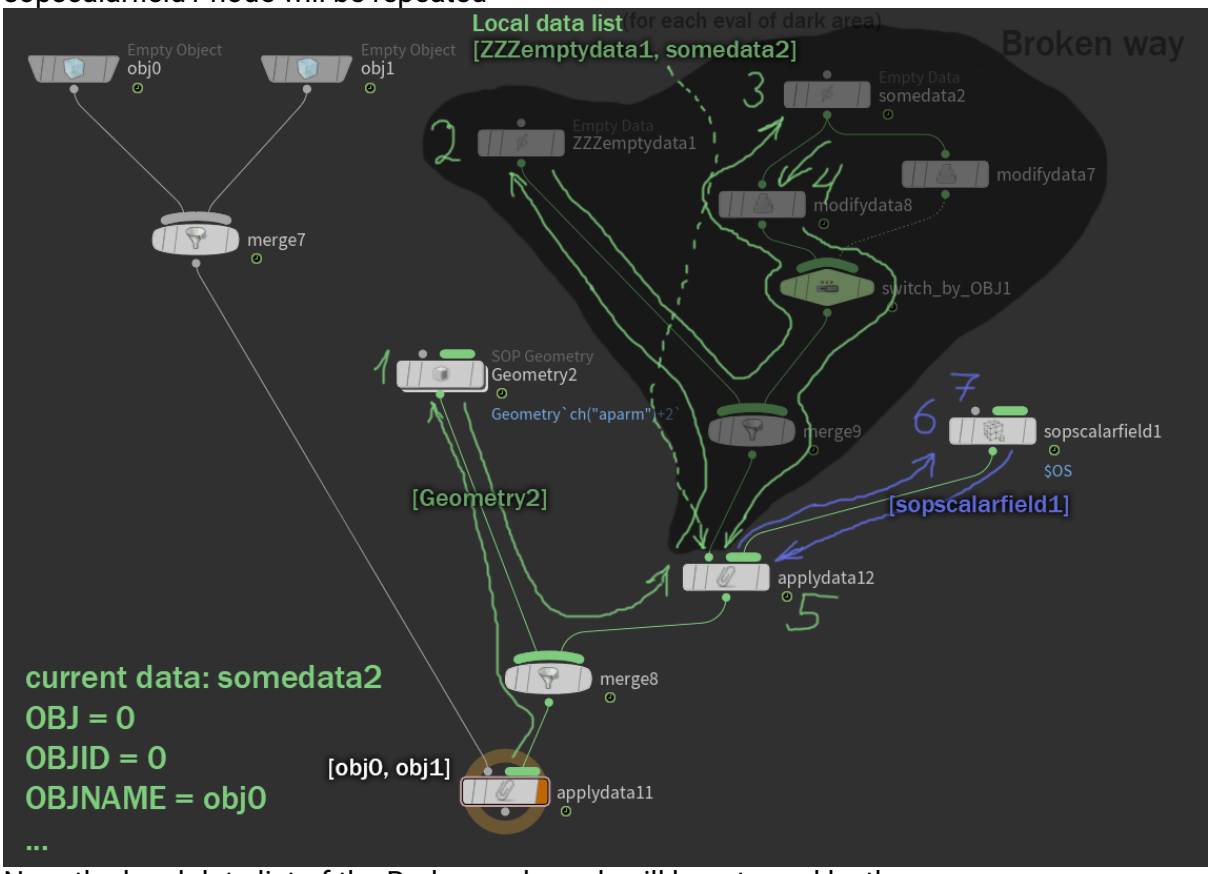
now the local data list will be served in the applydata12 node



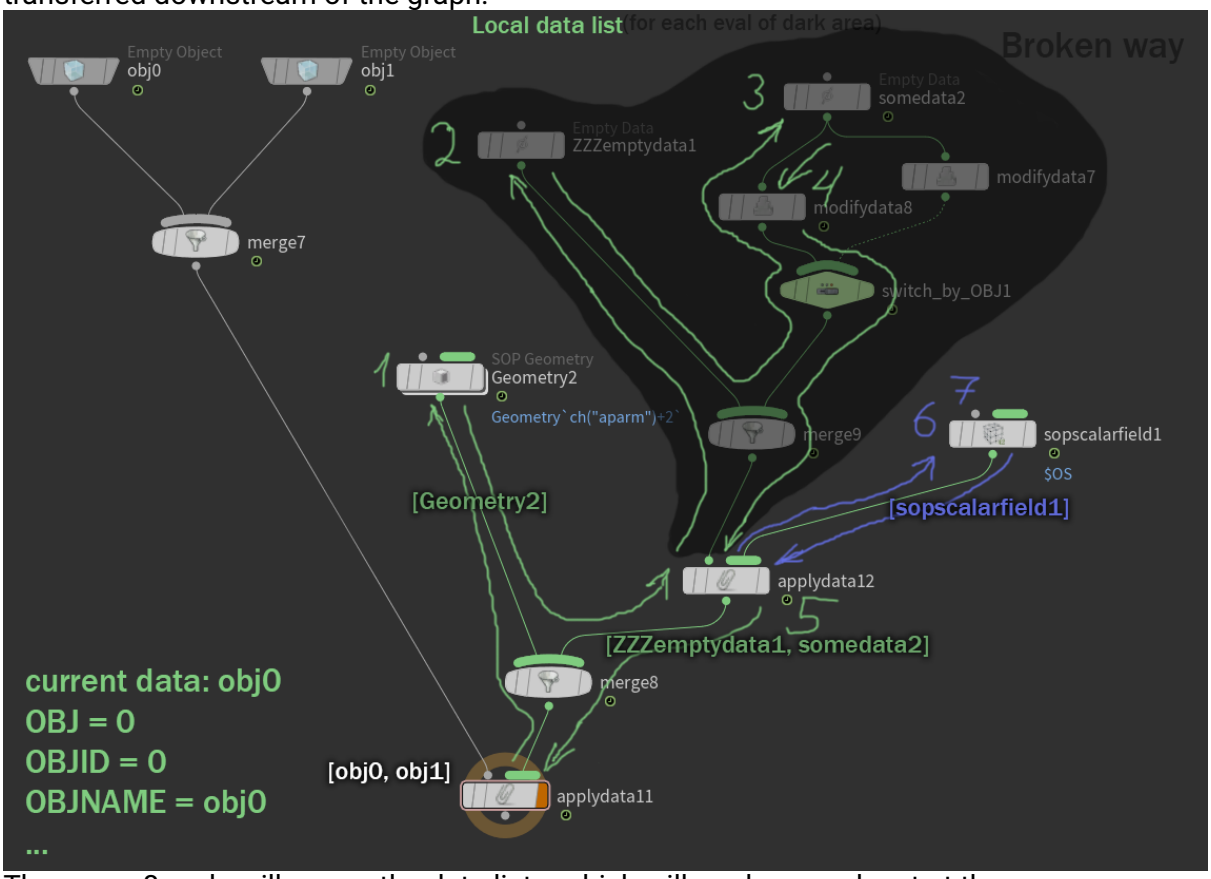
applydata12 for each data from this list will calculate all its subgraphs (the only one in this example) of the first input (in Normal mode). current data will be assigned the first data from the list (ZZZemptydata1). The sopscalarfiled1 node will be calculated, data of the SIM_ScalarField type will be created, a link with the name sopscalarfield1 will be added to them for the current data (ZZZemptydata1)



then somedata2 on obj0 will become the current data, and the calculation of the sopscalarfield1 node will be repeated

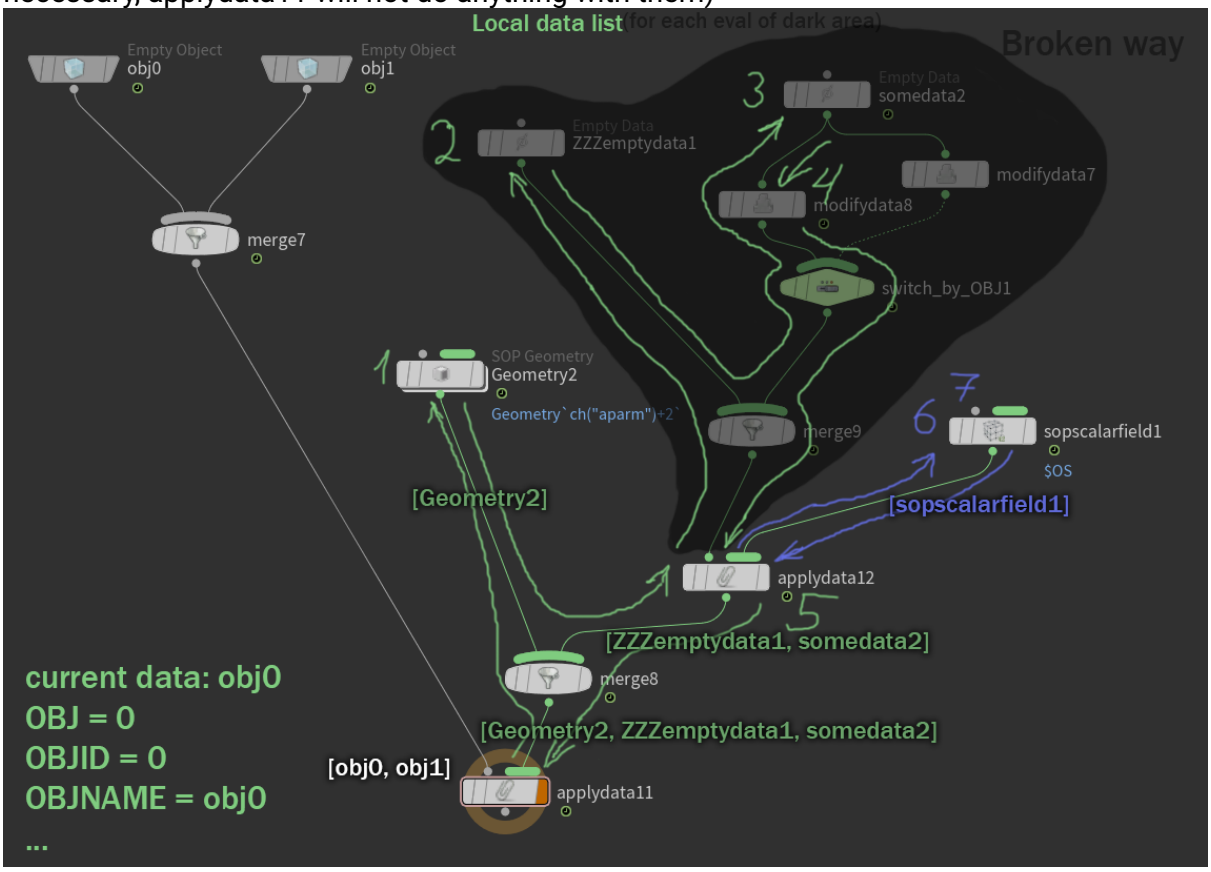


Now, the local data list of the Broken subgraph will be returned by the applydata12 node to the subgraph that runs in Normal mode, so it will be transferred downstream of the graph.

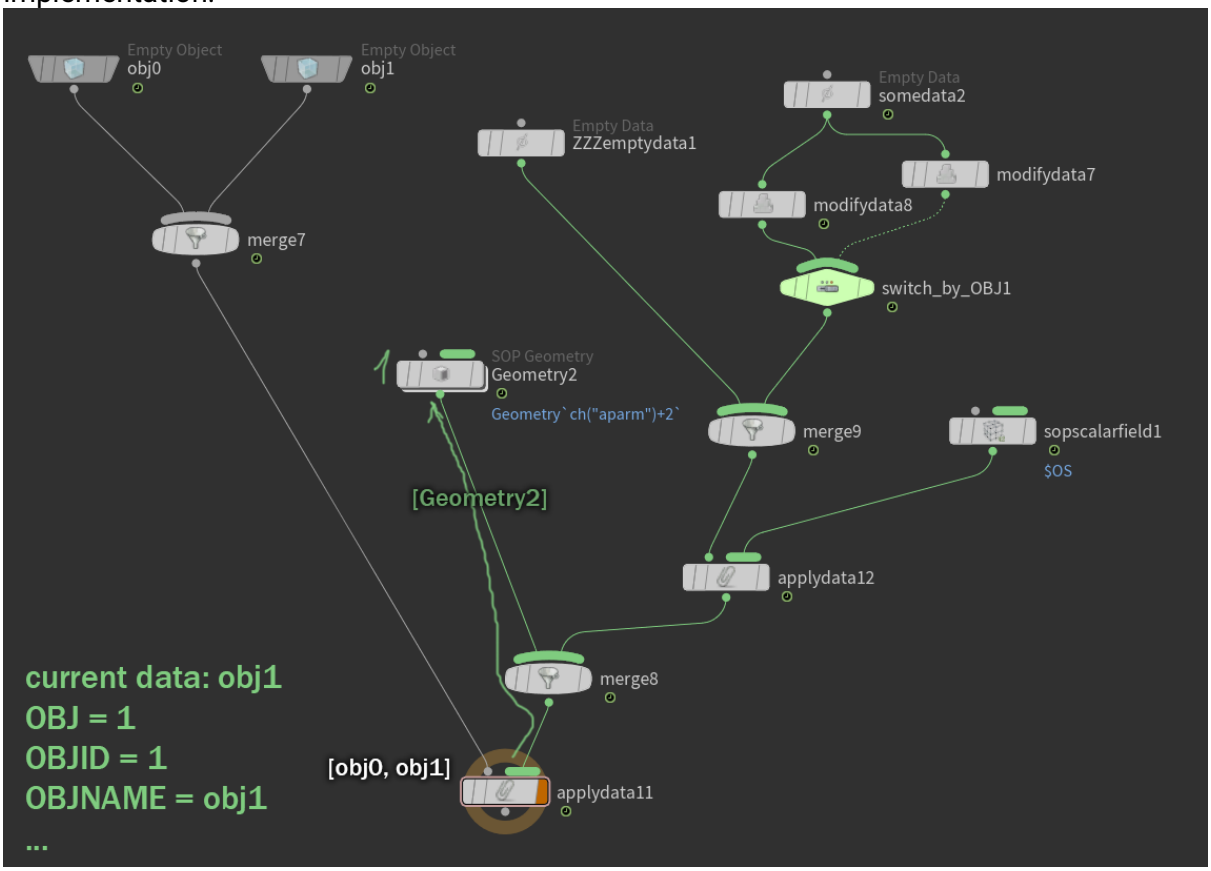


The merge8 node will merge the data lists, which will go down and rest at the beginning of the graph, this will finish the calculation of the second inputdata11

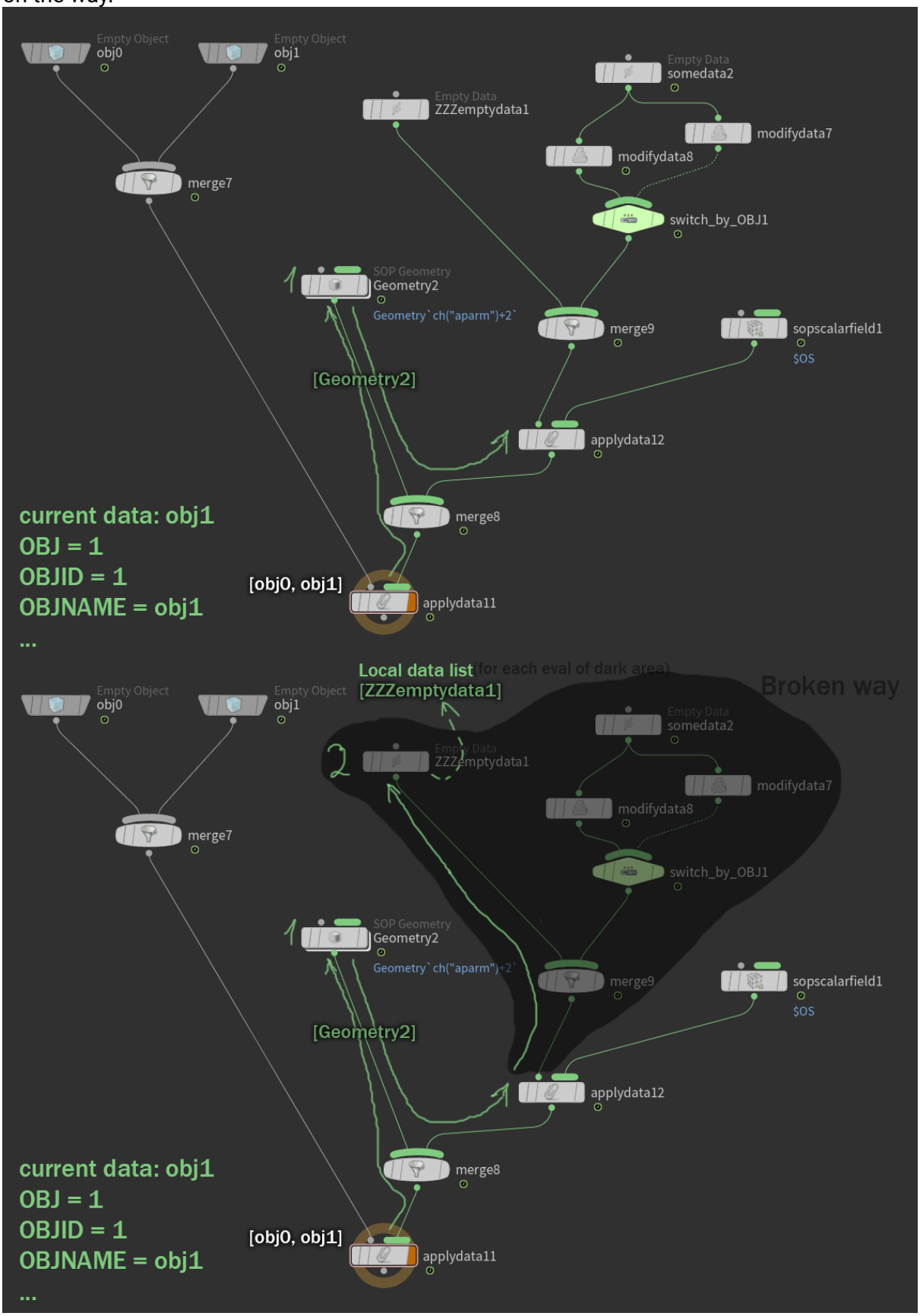
subgraph for the obj0 object (remember that the data is already fixed where necessary, applydata11 will not do anything with them)

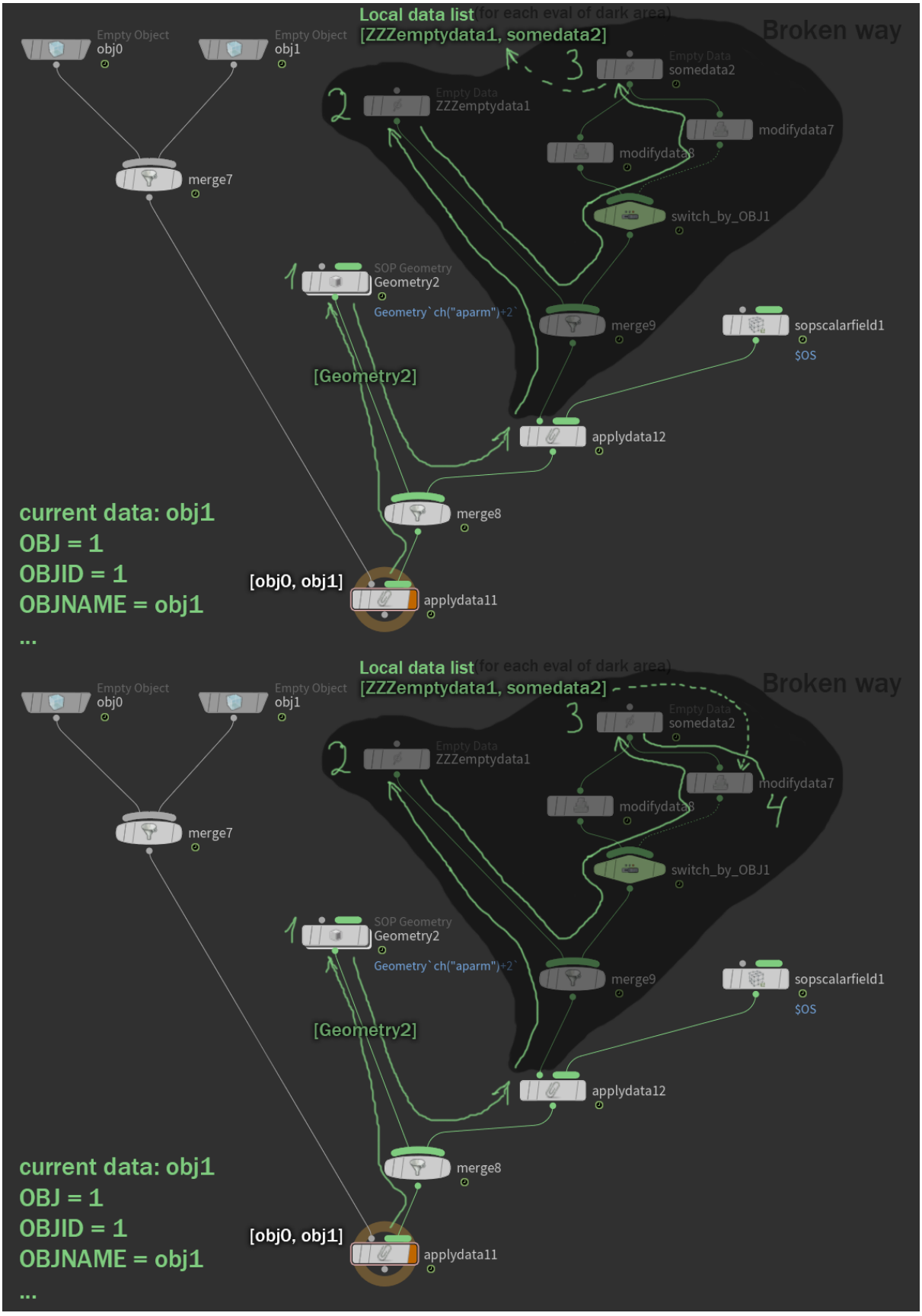


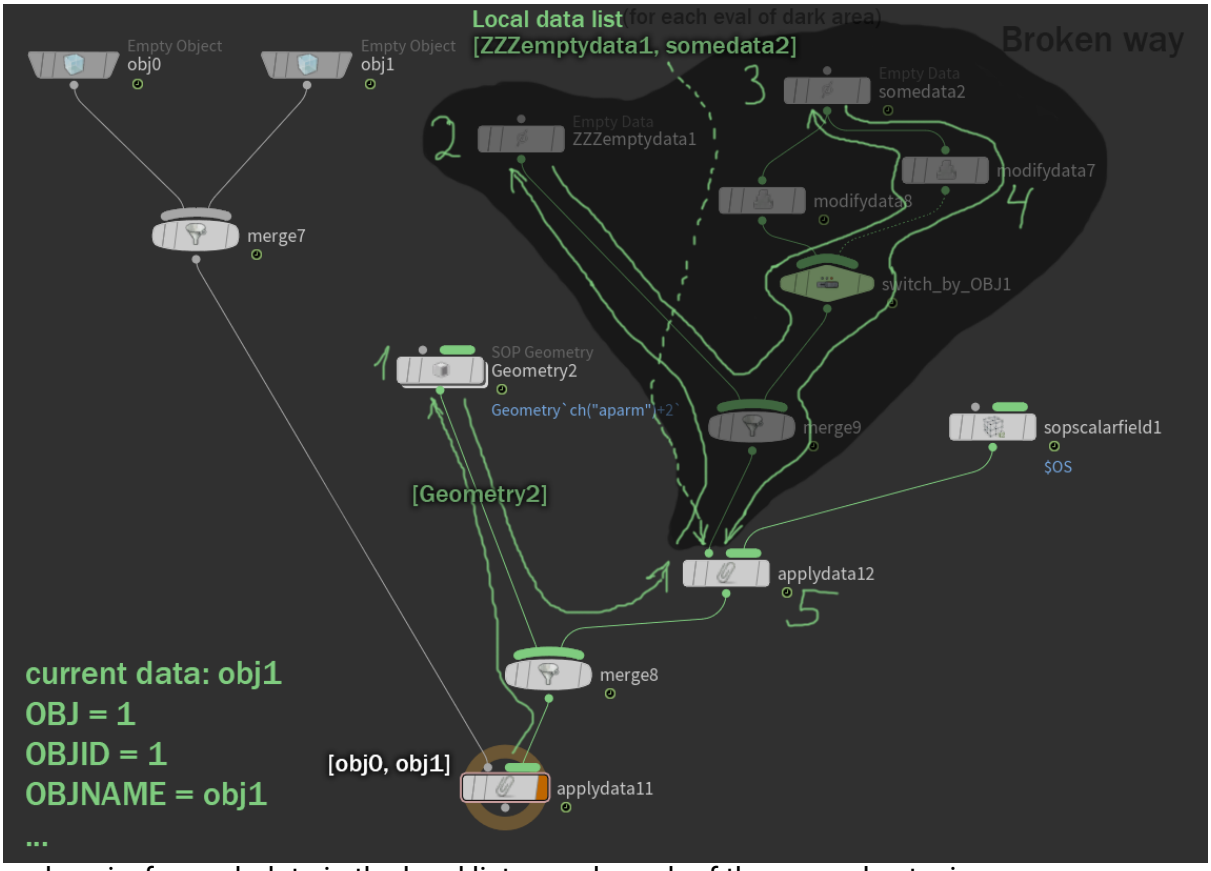
Now the obj1 object will be set with the current data and the subgraph of the second applydata11 input will be re-computed for it. Let's quickly go over this implementation:



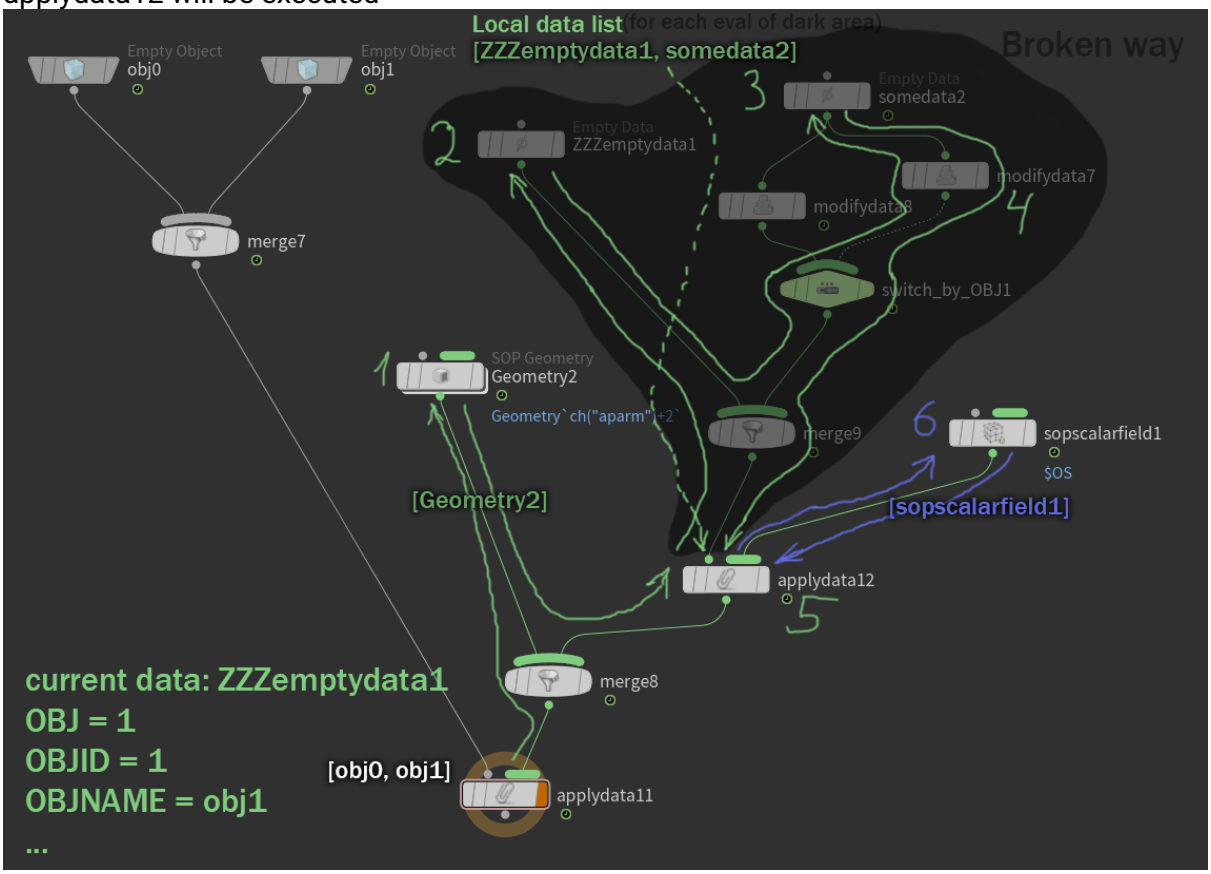
The entire description above is also true for the execution of the subgraph for the object obj1, only the switch switch_by_OBJ1 switch, which has $ OBJ as its value, will direct the graph traversal along another branch as soon as it is encountered on the way.

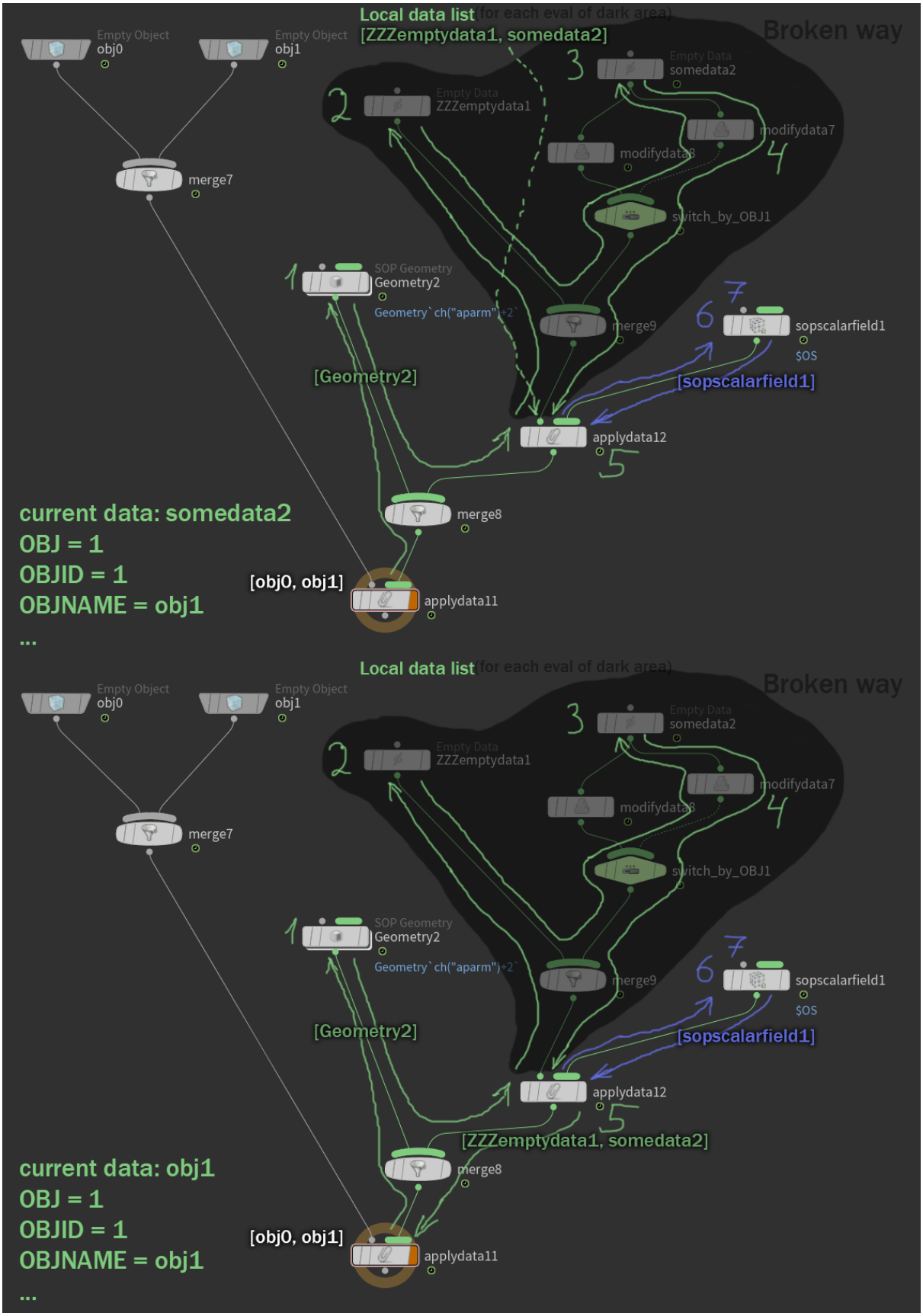


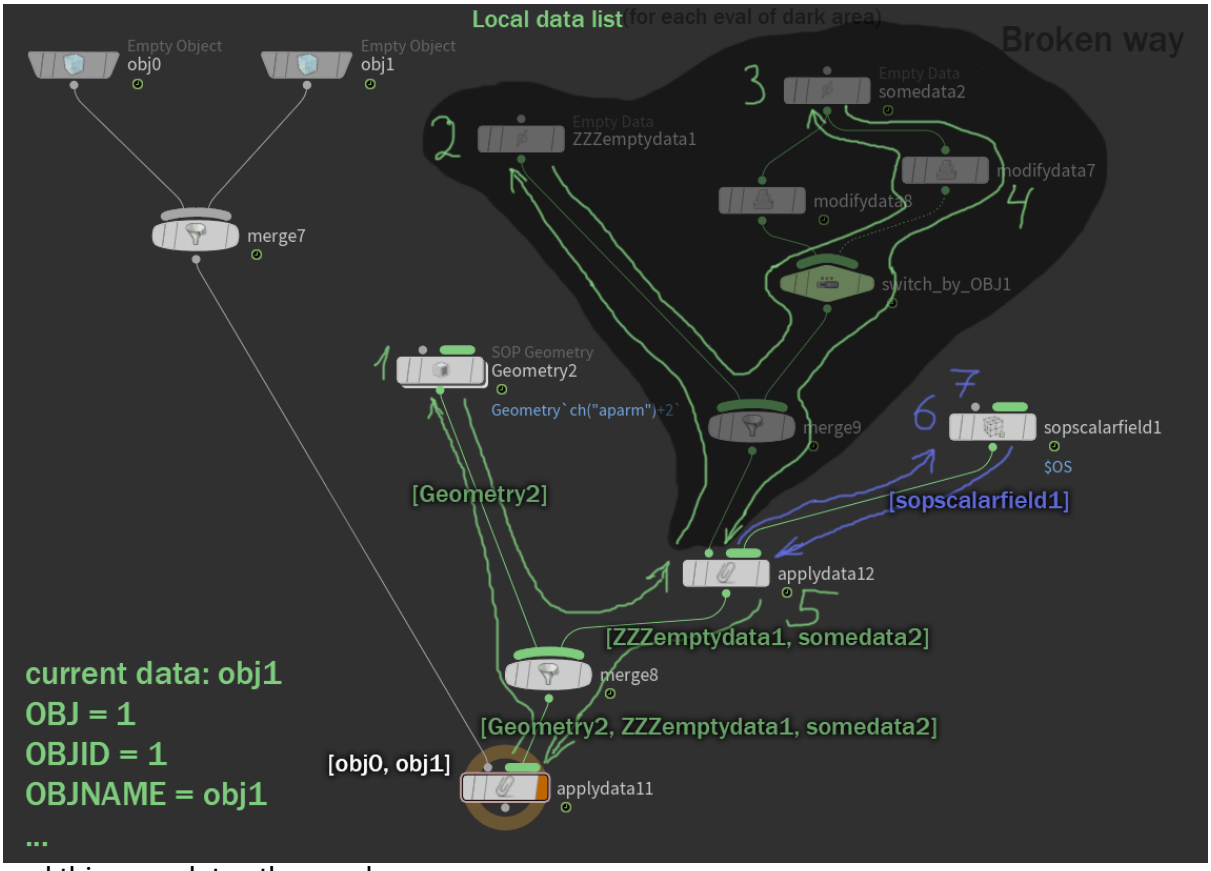

Local data list
[ZZZemptydata1, somedata2]
Broken way
Empty Object
obj0
Empty Object
obj1
Empty Data
somedata2
Empty Data
ZZZemptydata1
modifydata7
modifydata8
switch_by_OBJ1
merge7
SOP Geometry
Geometry2
Geometry`ch("aparm")+2`
merge9
sopscalarfield1
$OS
[Geometry2]
applydata12
merge8
[obj0, obj1]
applydata11
current data: obj1
OBJ = 1
OBJID = 1
OBJNAME = obj1
...
Local data list
[ZZZemptydata1, somedata2]
Broken way
Empty Object
obj0
Empty Object
obj1
Empty Data
somedata2
Empty Data
ZZZemptydata1
modifydata7
modifydata8
switch_by_OBJ1
merge7
SOP Geometry
Geometry2
Geometry`ch("aparm")+2`
merge9
sopscalarfield1
$OS
[Geometry2]
applydata12
merge8
[obj0, obj1]
applydata11
current data: obj1
OBJ = 1
OBJID = 1
OBJNAME = obj1
...



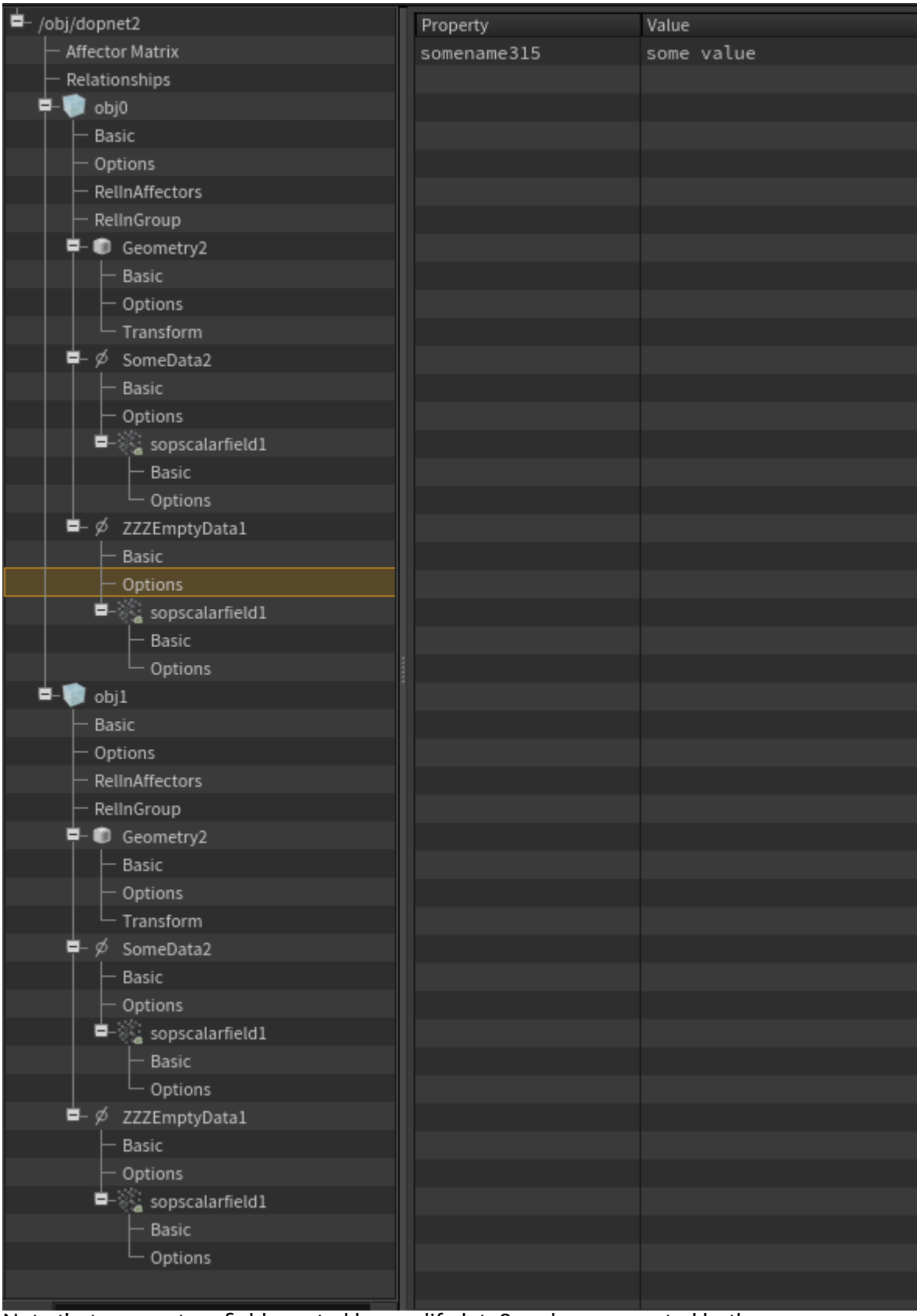
and again, for each data in the local list , a subgraph of the second entry in applydata12 will be executed



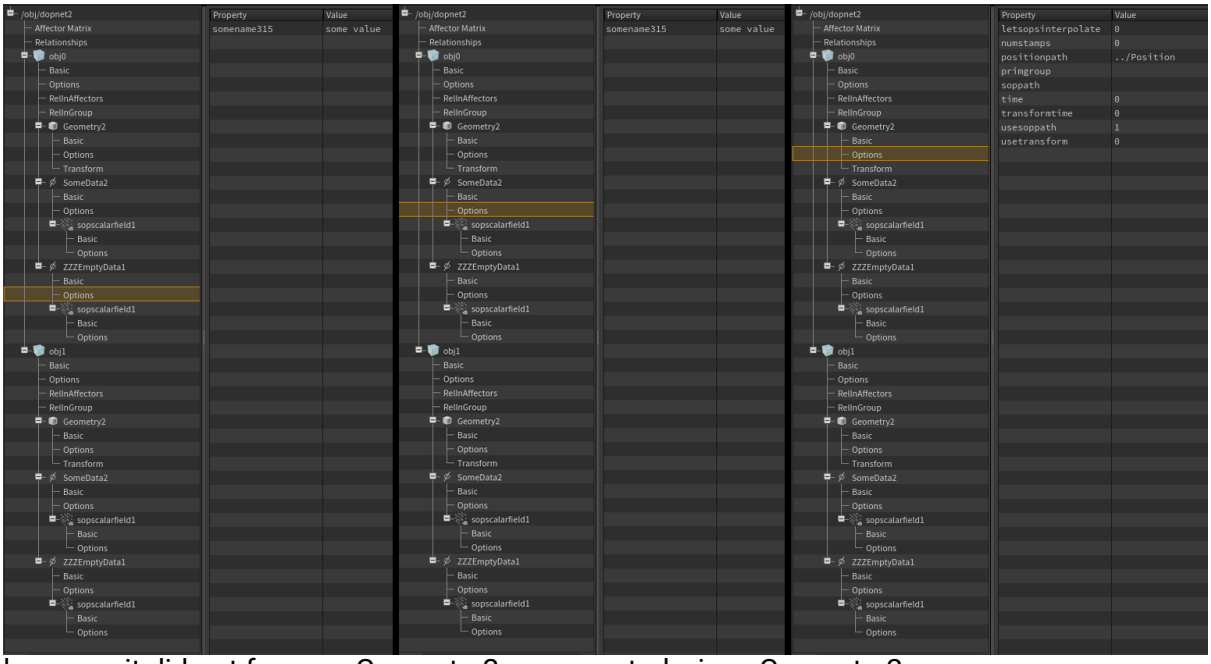

Local data list (for each eval of dark area)
[ZZZemptydata1, somedata2]
Broken way
Empty Object
obj0
Empty Object
obj1
Empty Data
somedata2
Empty Data
ZZZemptydata1
modifydata7
modifydata8
switch_by_OBJ1
merge7
SOP Geometry
Geometry2
Geometry`ch("aparm")+2`
merge9
sopscalarfield1
$OS
[Geometry2]
[sopscalarfield1]
applydata12
merge8
[obj0, obj1]
applydata11
current data: somedata2
OBJ = 1
OBJID = 1
OBJNAME = obj1
...
Local data list (for each eval of dark area)
Broken way
[ZZZemptydata1, somedata2]
current data: obj1
OBJ = 1
OBJID = 1
OBJNAME = obj1
...



and this completes the graph
As a result, the following data structure will be created:



Note that our custom field created by modifydata8 node was created both on SomeData2 and ZZZEmptyData1



however, it did not form on Geometry2, as expected, since Geometry2 was created in the Normal graph traversal mode, and SomeData2 and ZZZEmptyData1 were created in the Broken single-subgraph traversal mode

(scene described example: **sppoqituka @ HPaste** *(OBJ context))*

In the example above, the scheme for computing the graph of nodes at the first timestep of the simulation was described, when the simulation did not yet have data. What will happen at the next timestep? on an arbitrary timestep with an arbitrary data structure that already exists in the simulation?
As described earlier, the traversal and execution of the nodes will occur in exactly the same way, it will only be necessary to add that the obj0 and obj1 nodes will calculate their activation and number of objects parameters and decide whether they will create new objects or not, and into the lists produced by them objects that already exist in the simulation will be added, in which the creator field contains the path to any of these nodes, respectively.
Data will be used in the same way, only instead of creating data of a certain type, if any data with a link with the given name already exists on the current data, the existing ones will be transformed to the requested data type. If the data already has the specified type, as in the vast majority of cases when data of the same type is simply updated from the timestep to the timestep of the simulation, a link to them is simply returned, no processing occurs. How data can be transformed from one type to another is determined by the programmer at the stage of creating the data type. the maximum that Houdini can do automatically is to create new data from scratch and copy the sub-data links and records from the transformed data onto them, the Houdini's clear business in itself, without the help of a programmer, cannot know anything about the internal data structure and how to transform it into another. So, in our case, on the second frame of the simulation, all data nodes instead of creating new data will simply return links to existing ones that they created earlier, nothing new will be created.

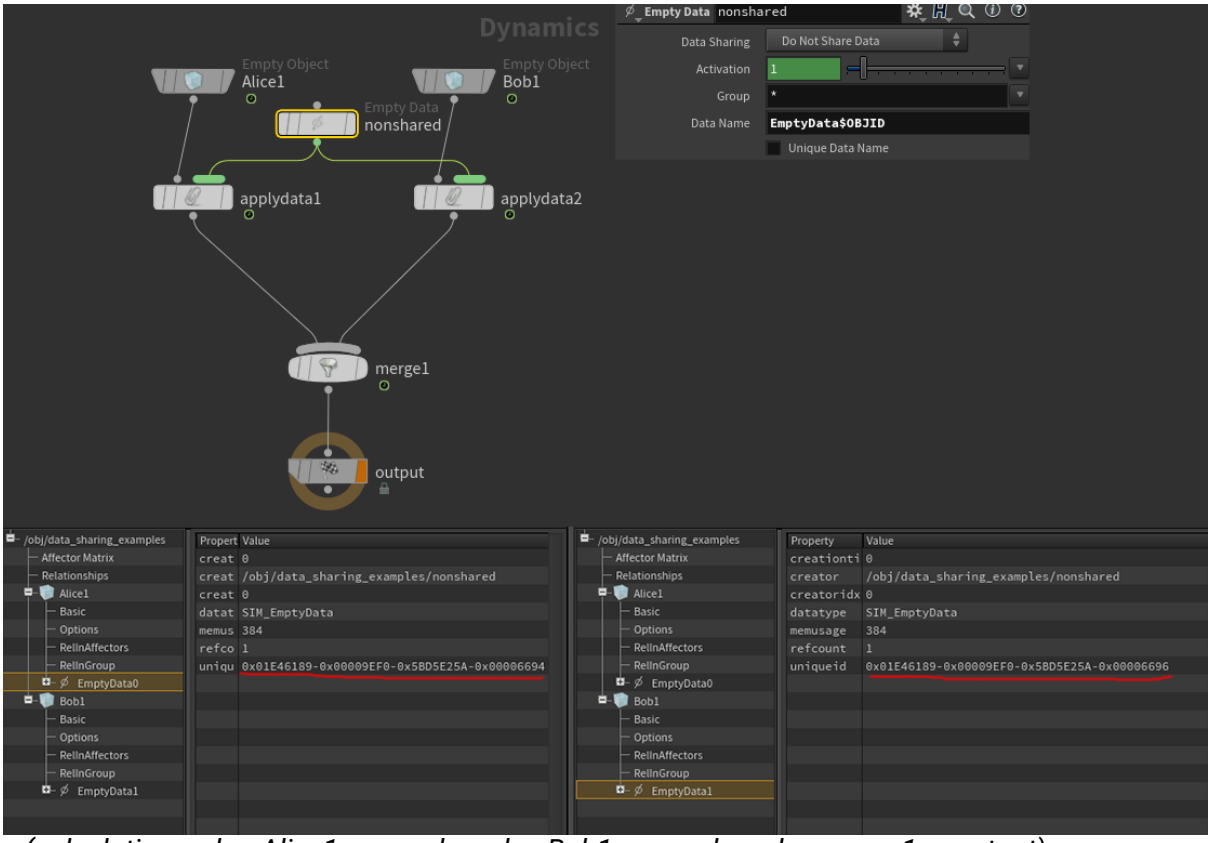
DOP groups. Groups in a DOP context are similar to groups in a SOP intuitively. Unlike SOP, only DOP objects and nothing more can be in DOP groups. For many DOP nodes, you will see the Group field containing the group pattern perceived by this node - this pattern applies not only to the names of the additional groups the object is in, but also to the names of the objects and their objid. If an object from the list being processed (in the object stream) or the current object (in the data stream) does not fit the pattern specified in the node parameter group, the node for this object is not executed. DOP groups are implemented through group relayship without affectors, i.e. not affecting the determination of the calculation order of solvers.

Data Sharing Parameter

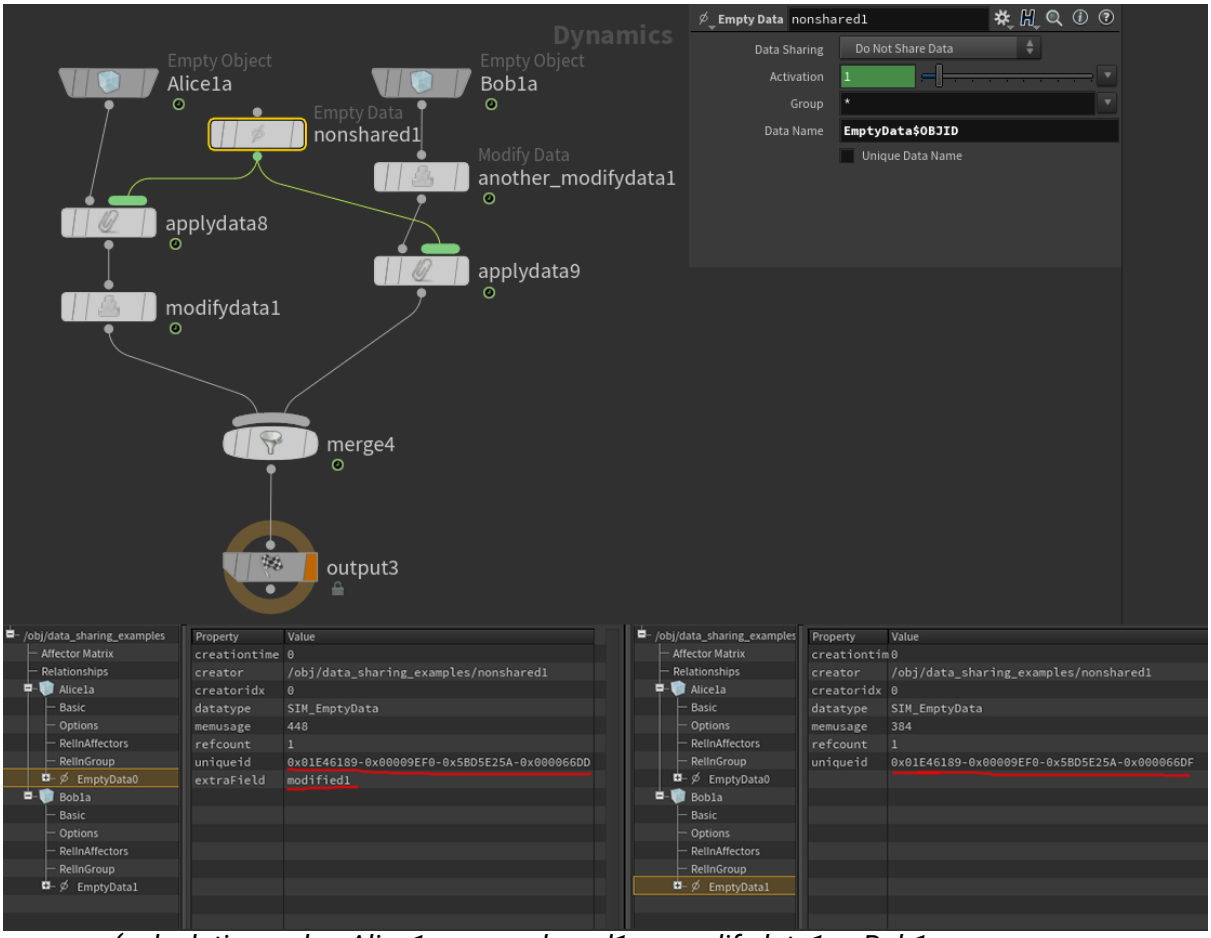
We already know how data can have several opposite links to different data, in this case the data is called shared. the default interface of all nodes producing and connecting data includes one parameter, the purpose of which is to give the user at least some control over how to create shared data on different objects or data, this is the Data Sharing parameter.
Despite the likely noble idea, the final implementation introduces even more confusion into the already relatively broken calculation of additional nodes.

Do not share data

Everything described above is true for the default value of data sharing - " **Do not share data** ".
Each time a data node is executed, it is executed separately, regardless of its previous executions in a given timestep or past.
using a simple setup as an example, we can make sure that the data is created different

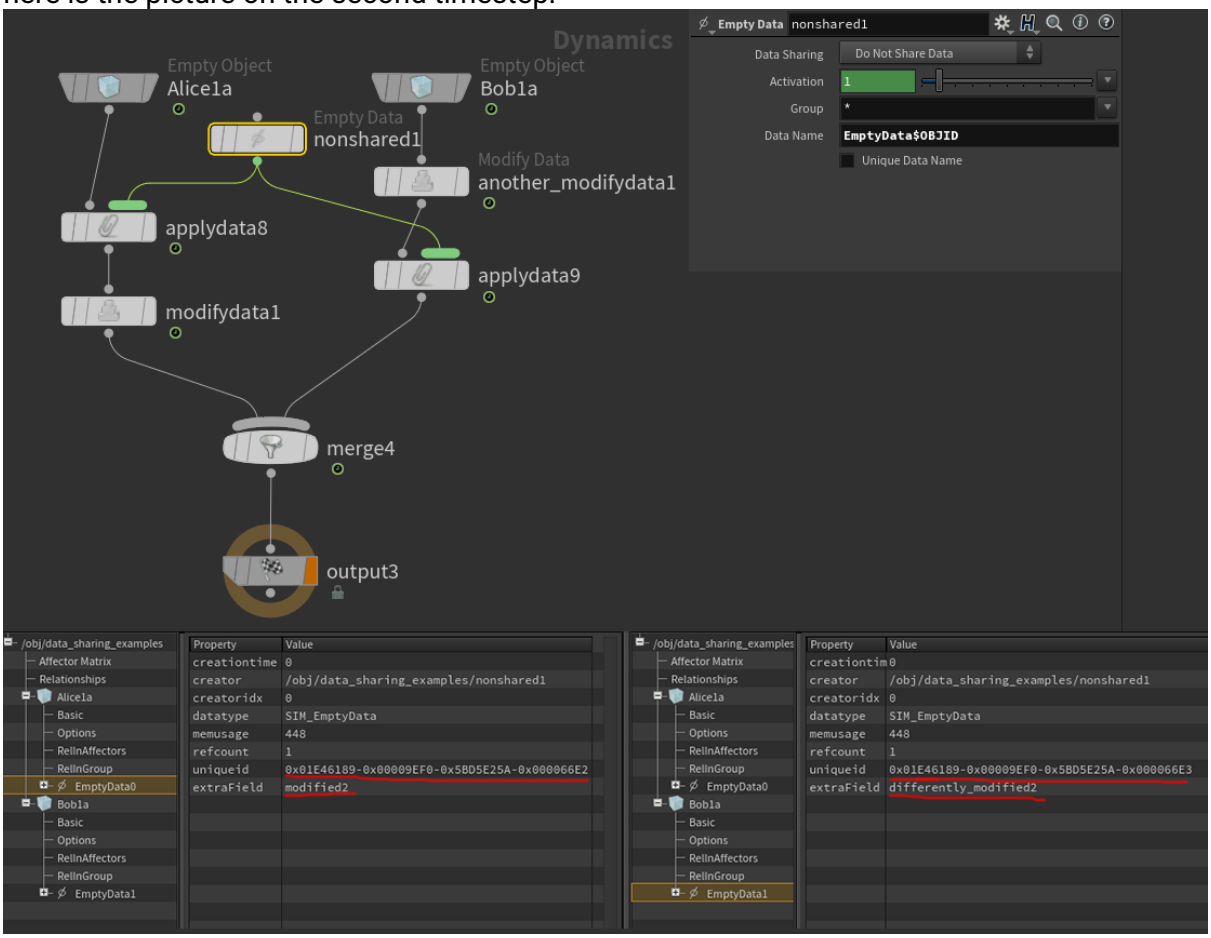


(calculation order: Alice1 -> nonshared -> Bob1 -> nonshared -> merge1 -> output)
we'll complicate the setup a bit: note on the first timestep, the data is still completely different, modifydata1 only affects the data of Alice1a, another_modifydata1 does not work so far, because the data is created only after it is executed when the graph goes deeper)



(calculation order: Alice1a -> nonshared1 -> modifydata1 -> Bob1 -> another_modifydata1 -> nonshared -> merge1 -> output)

here is the picture on the second timestep:



(the calculation procedure is the same)

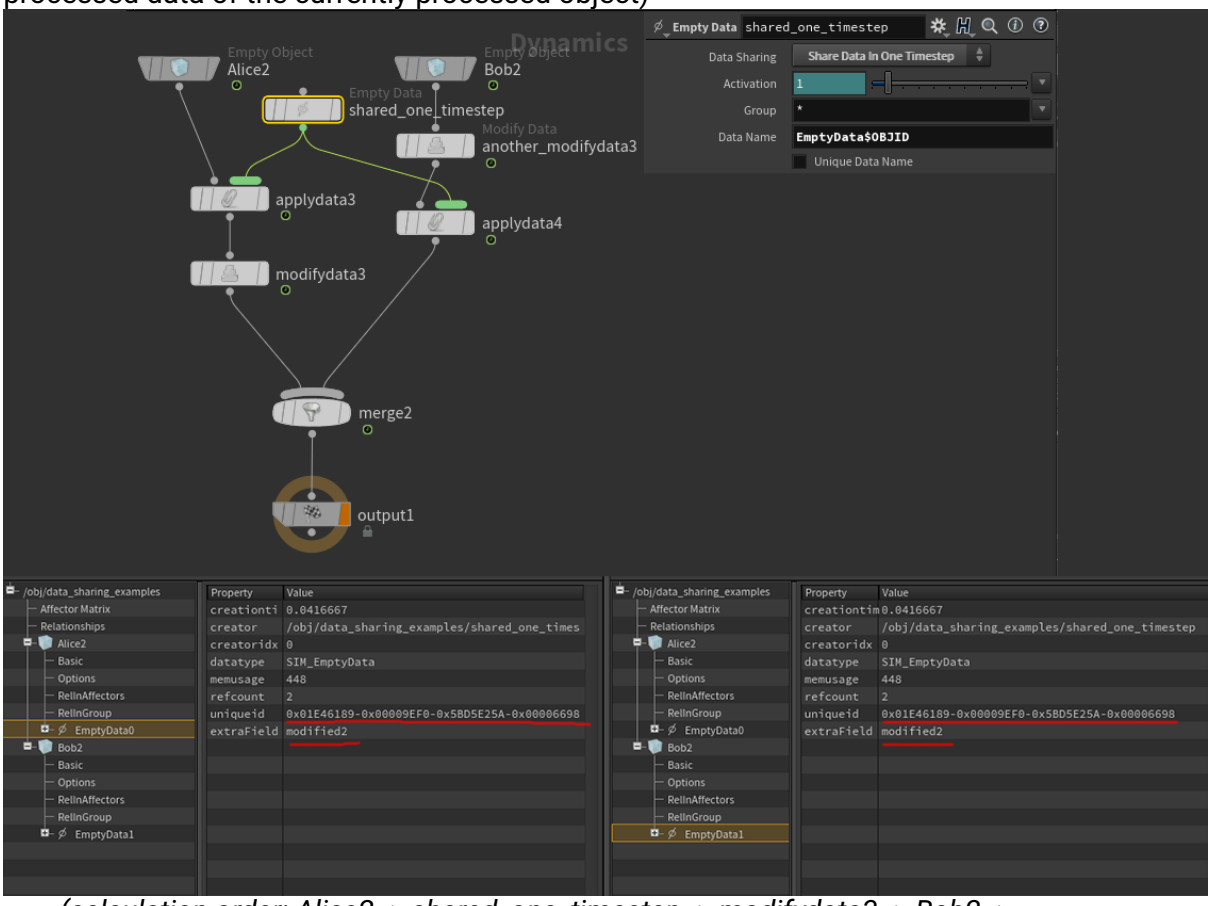
note, the data on Alice1a and Bob1a are processed completely independently, according to the logic already described above.

(example: **sppsorenuc @ HPaste** *(OBJ context))*

Other values of the Data Sharing parameter change the logic of creating a data node:

Share Data In One Timestep

In this mode, each timestep during the first processing in this timestep, this node will remember the object number and the name of the link on this object to the processed data (not the data link itself) (with the names of the links to their parent data, if the link is in the root of the object, on other sub-data, for example, object: smokeobject1, data link name: density / Visualization). And during all subsequent executions in the current timestep, return the stored data instead of any standard processing. That is, this data could be deleted, re-created by another node and changed as you like, but the aforementioned node will still return them, instead of creating or processing the data in a classical way, because the search takes place by the object number and link name. (This is true for reducible data types, moreover, an unreduced type will be returned, that is, the GeometryCopy node can thus return data of the SIM_SopGeometry type, which can lead to even more confusion than all of the above. For irreducible data types, the node will forget the stored object number and link name, and will work as if for the first time in this timestep - that is, to recall the object number and link name of the processed data of the currently processed object)



(calculation order: Alice2 -> shared_one_timestep -> modifydata3 -> Bob2 -> another_modifydata3 -> shared_one_timestep -> applydata4 -> merge2 -> output1)

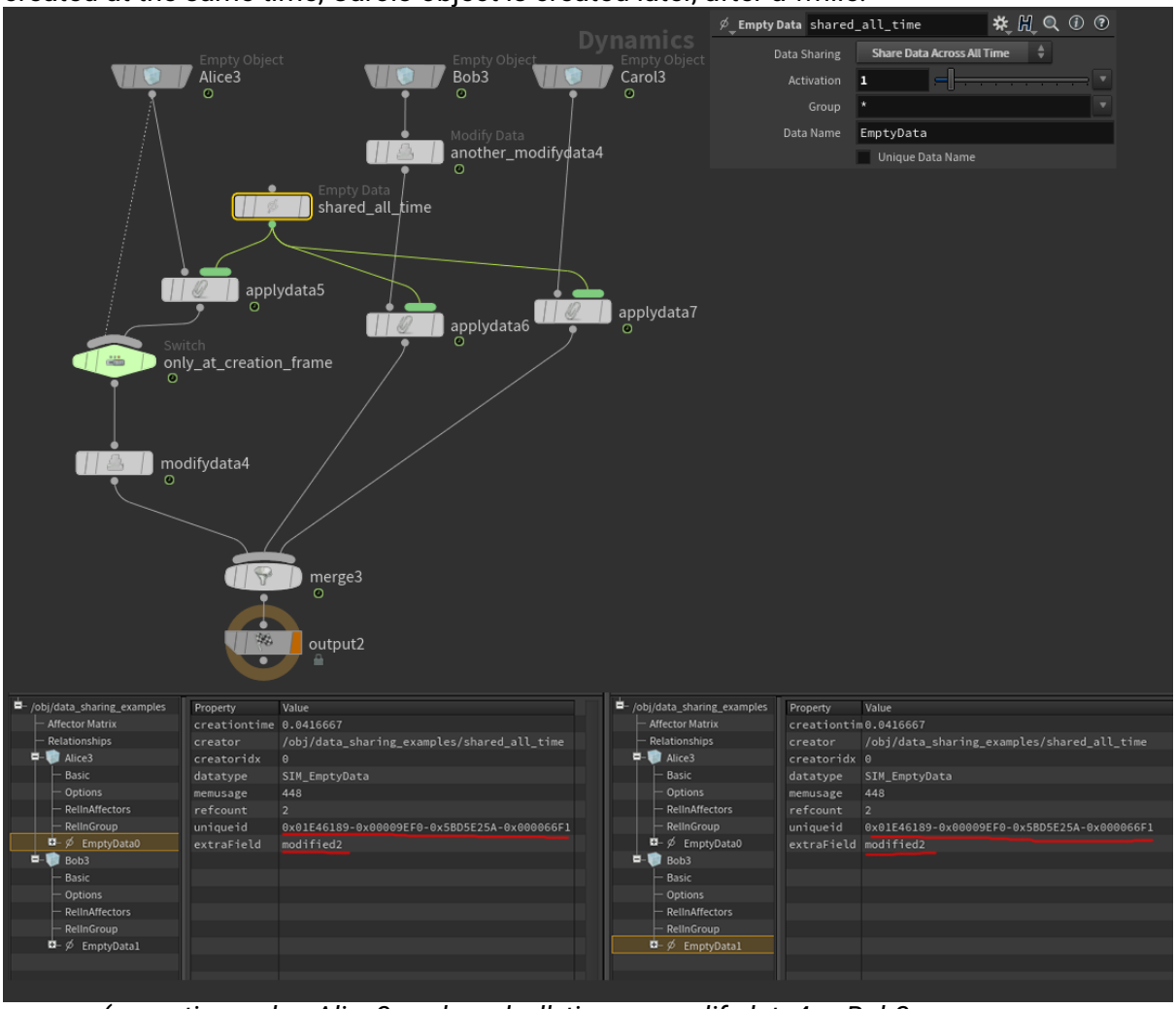
In this example, it is seen that the shared_one_timestep node creates data first on the Alice2 object, a reference name for the created data and the object number, so when shared_one_timestep is processed for the current Bob2 data - it does not look for data on Bob2, it collects data from Alice2 by the stored name, therefore, even if the original data created by shared_one_timestep on Alice2 was copied and replaced, it is the newly attached data to Alice2 that will be taken by name,

and if their type is the same or reduced to SIM_EmptyData (in this case, in most cases, the types will match exactly but) - are attached to the current Bob2 data with a link with a new name calculated for Bob2 (note in the screenshot that the link names of the objects are different for the same data)
And each frame, despite the fact that another_modifydata3 node will copy and modify the EmptyData1 data, the subsequent operation of the shared_one_timestep node for Bob2 will overwrite this link to the EmptyData0 pointing to the data with Alice2

(example **sppagojege @ HPaste** *(OBJ context))*

Share Data Across All Time

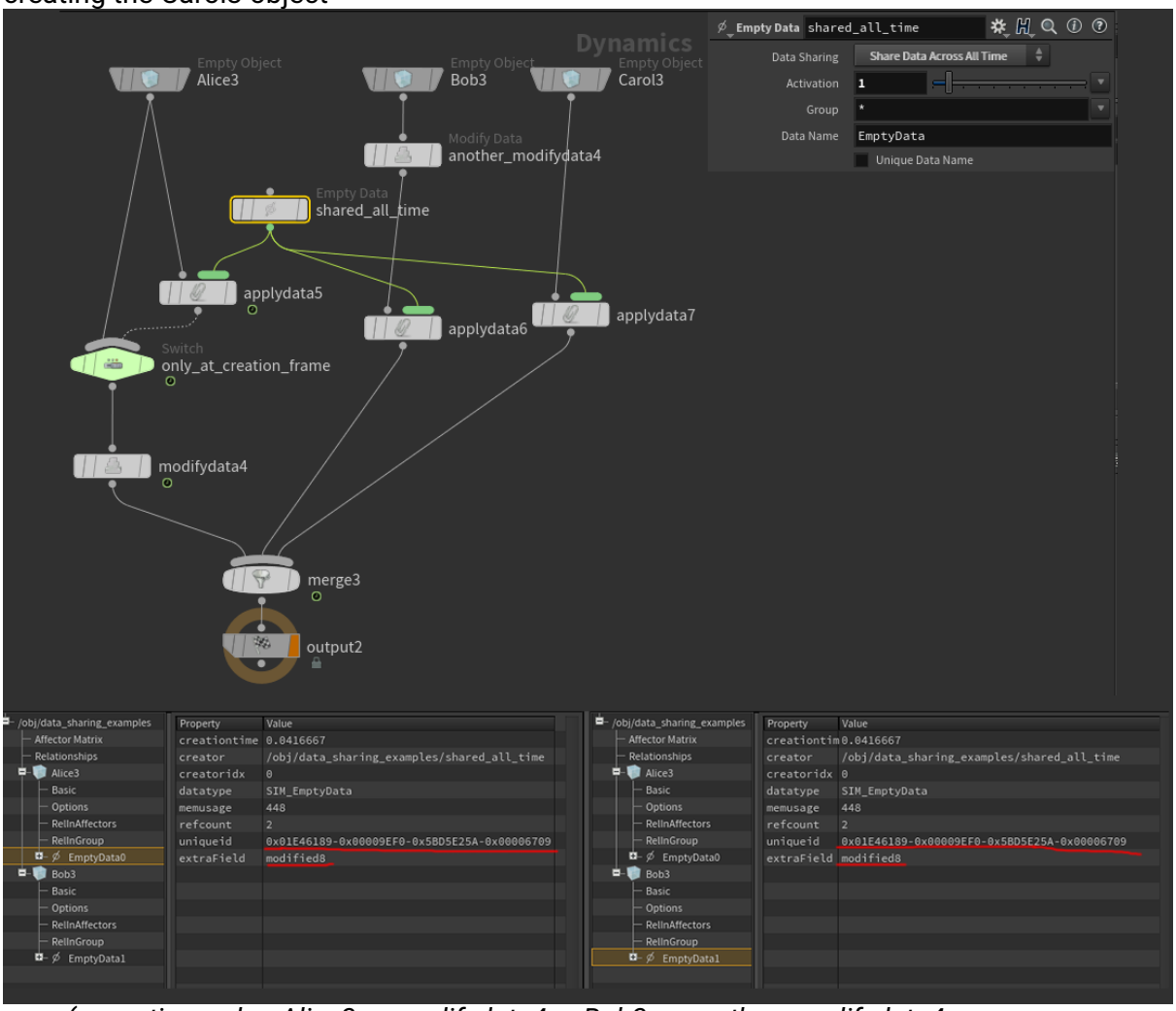
This mode is similar to the previous one, all the logic of the previous mode is correct, with the correction that instead of localizing to one timestep, the behavior extends to the entire simulation, that is, the stored object number and link name are not forgotten when switching from a timestep to a timestep. There is one stupid noticed bug that I will describe after the example, so as not to be confused.
As in the case for Share Data In One Timestep, if the data is not found by the stored object number and data link name, or if the data is irreducible to the data type produced by this node, the “memory” of the data creation node will be reset, the number will be forgotten the object and the name of the link, and the node will be executed as if the first time.
Let's look at an example that is a bit more complicated in order to show the peculiarity of this Data Sharing mode. meaning - a branch with applydata5 works only in one timestep of creating an Alice3 object, after that the switch only_at_creation_frame always gives a zero path. Alice3 and Bob3 objects are created at the same time, Carol3 object is created later, after a while.



(execution order: Alice3 -> shared_all_time -> modifydata4 -> Bob3 -> another_modifydata4 -> shared_all_time -> Carol3 -> shared_all_time -> merge3 ->

output2)

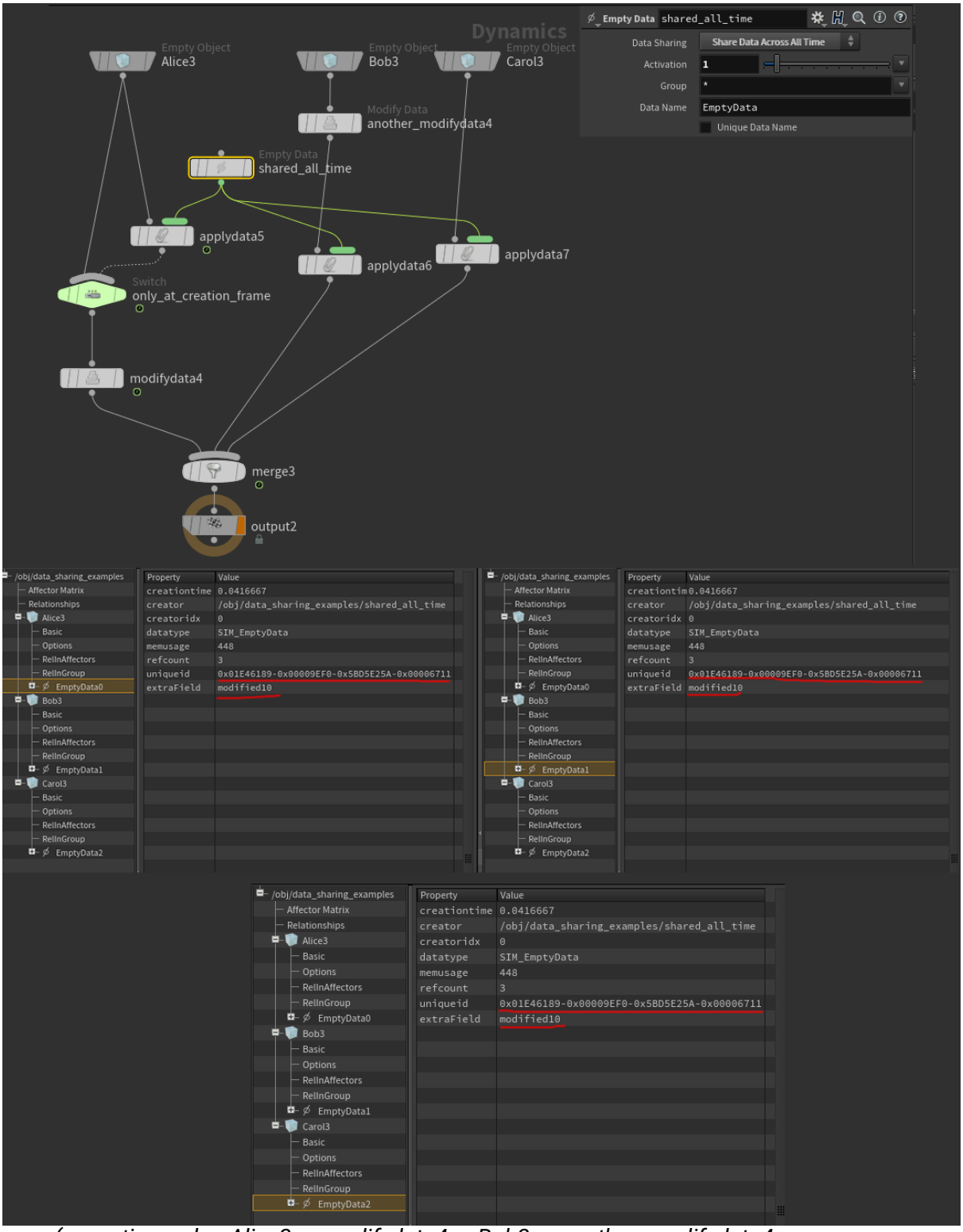
At the first timestep, we see a picture identical to that from the previous example, the share_all_time node remembered the Alice3 object number and the data link name EmptyData0, and they are also attached to the Bob3 object with the link name EmptyData1. Now let's look at one of the following timesteps, but before creating the Carol3 object



(execution order: Alice3 -> modifydata4 -> Bob3 -> another_modifydata4 -> shared_all_time -> Carol3 -> shared_all_time -> merge3 -> output2)

now applydata5 will never be calculated, modifydata4 on Alice3 will change the EmptyData0 data created on the first timestep to Alice3, but the share_all_time node will never be processed with the Alice3 object again due to the switch. However, we see that despite the action of another_modifydata4 node, which copies the EmptyData1 data to Bob3 and changes extraField to differently_modified $ F, the subsequent operation of the share_all_time node for the Bob3 object will overwrite the link to the data found by the name stored on the number at the first execution in the first timestep at the Alice3 facility. Note that it is important that share_all_time does not remember the data reference, but rather the object number and data name. Therefore, it returns no link to the data stored in the first timestep,

Now let's look at the timestep of creating a Carol3 object



(execution order: Alice3 -> modifydata4 -> Bob3 -> another_modifydata4 -> shared_all_time -> Carol3 -> shared_all_time -> merge3 -> output2)

on this timestep, everything happens the same as on the previous one, only in addition a new Carol3 object is created, and a link is attached to it with the same data obtained by the object number and data name with Alice3, the same as for Bob3, but with link personal name EmptyData3

(example **sppnocejox @ HPaste** *(OBJ context))*

Let's get back to the bug that I mentioned. If you open the example for Share Data Across All Time, you will see that the Alice3 and Bob3 objects are created not in the first timestep of the simulation, but in the second. the reason is a detected bug, resetting all remembered links for nodes in the Share Data Across All Time mode exclusively after the first timestep. That is, at the first timestep of the

simulation, the nodes in the Share Data Across All Time mode actually work in the Share Data In One Timestep mode. Starting from the second timestep - everything works correctly, according to the described logic. Maybe there is some hidden meaning in this, but it looks more like a bug, since there is little logic behind this behavior.

Solvers and Jackdaw Solver Per Object

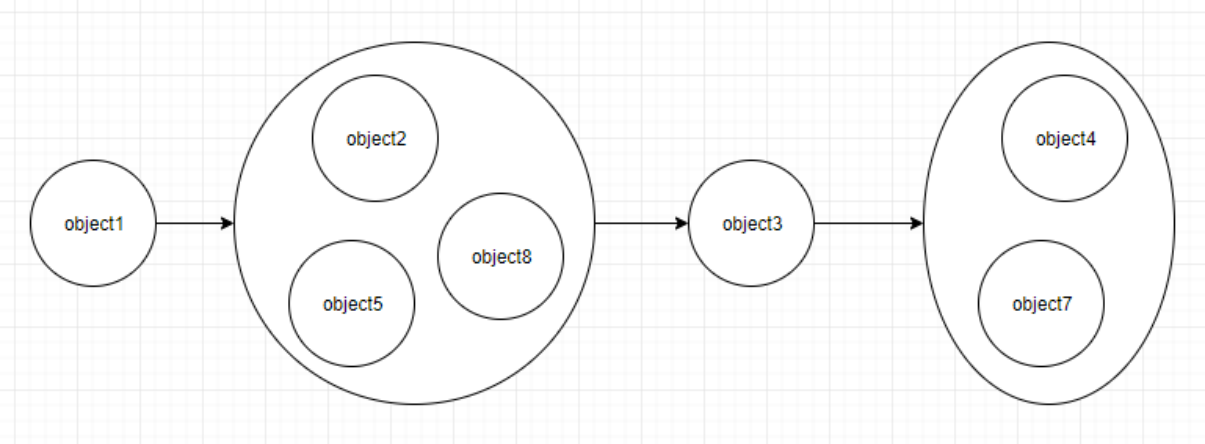
You may notice that the data nodes of the solvers usually do not have the Data Sharing parameter, however they have the Solver Per Object checkbox. Since any call to the solver functionality is by definition considered a change in the solver data, and the solver functionality is called very often (solving stage), the Share Data Across All Time mode is not very reasonable for the solvers, so the user is given a choice between the Share Data In One Timestep mode - equivalent to the unchecked checkbox in the Solver Per Object parameter, and Do Not Share Data mode - equivalent to the checkbox set in the Solver Per Object parameter.

Solving Stage

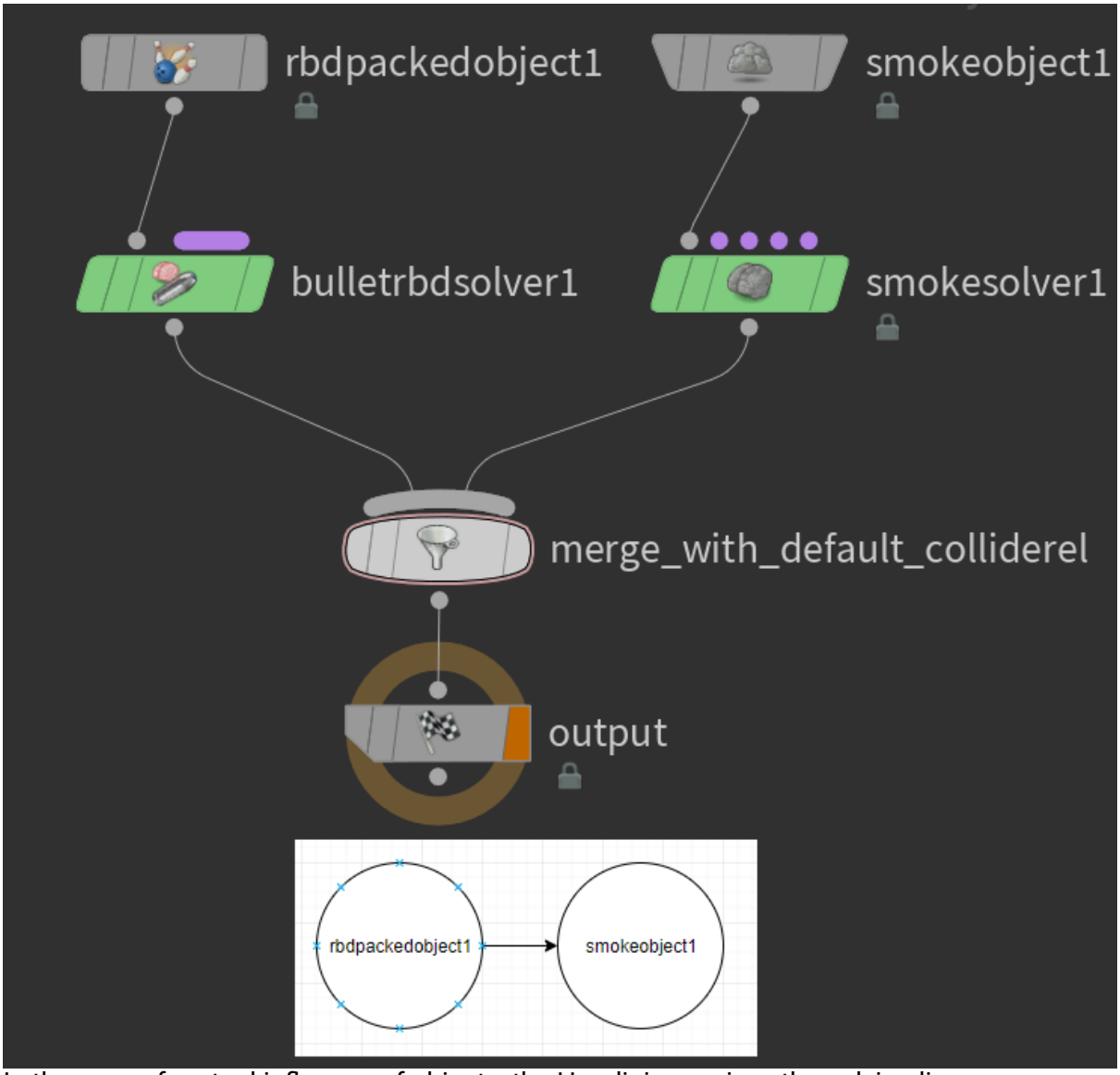
The solving stage is much more familiar to many than the stage of computing the node graph. It occurs no longer tied to the node graph, purely on simulation data.

Solving objects

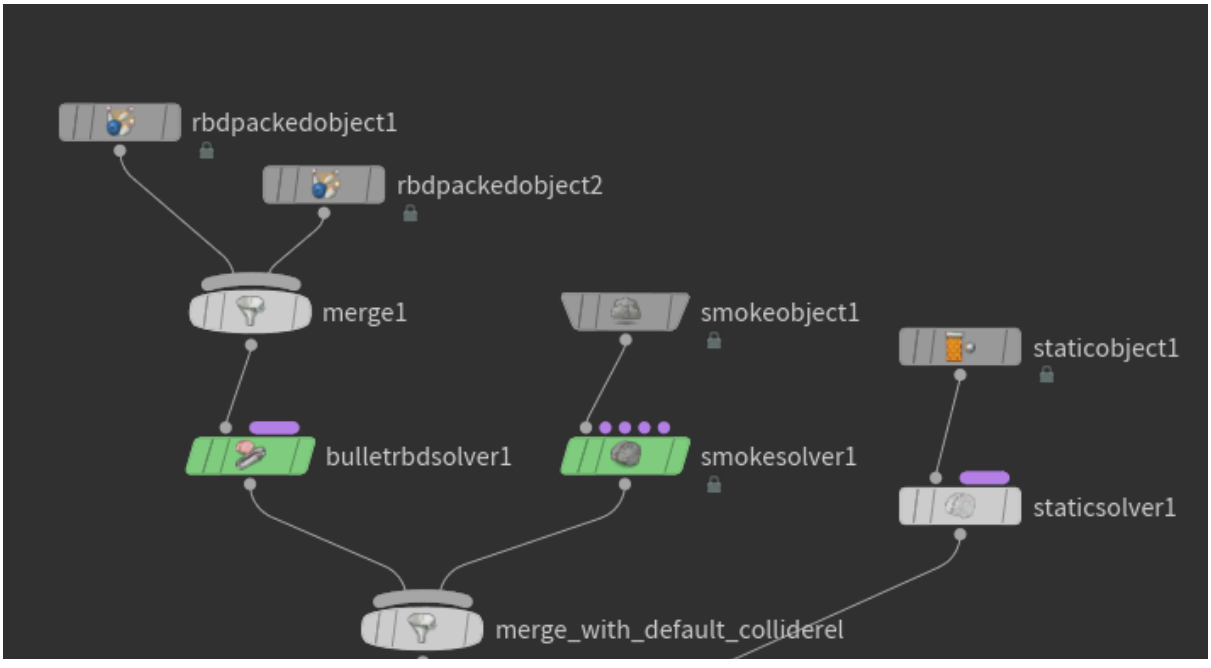
After going through the node stage and all the data changes associated with it, the Houdini takes all the relayships associated with the objects of the current timestep, and, analyzing which objects affect which ones, selects the calculation order that best suits all the relayships, building a queue of objects, elements of this queue can be both individual objects and groups of objects connected by mutual relayships.

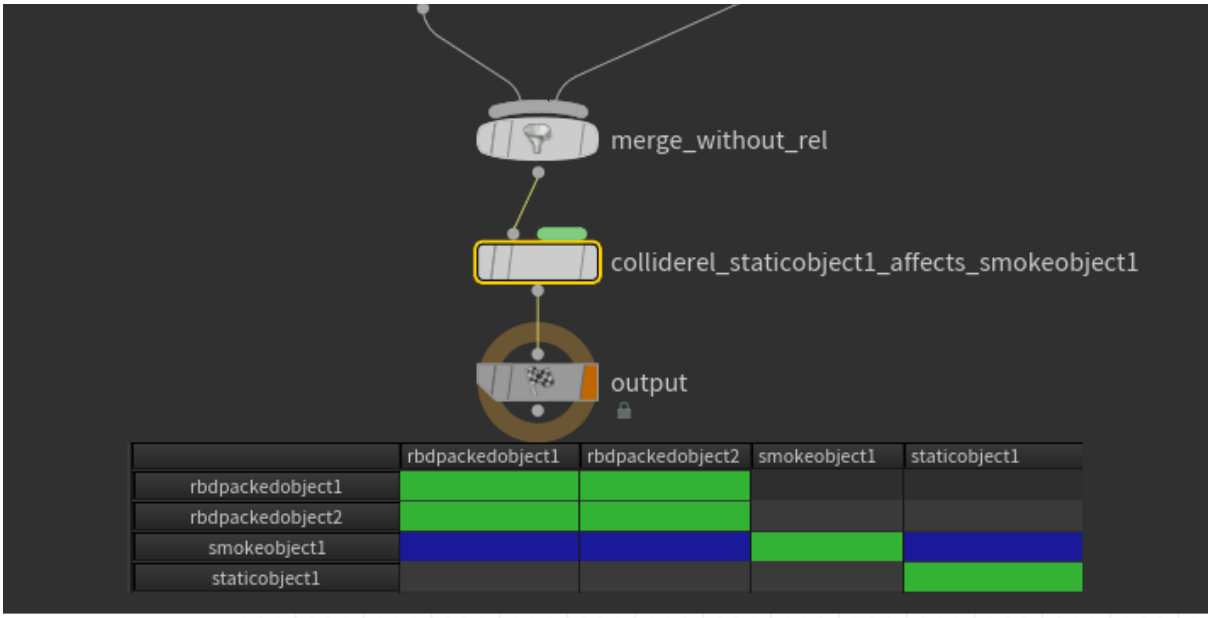


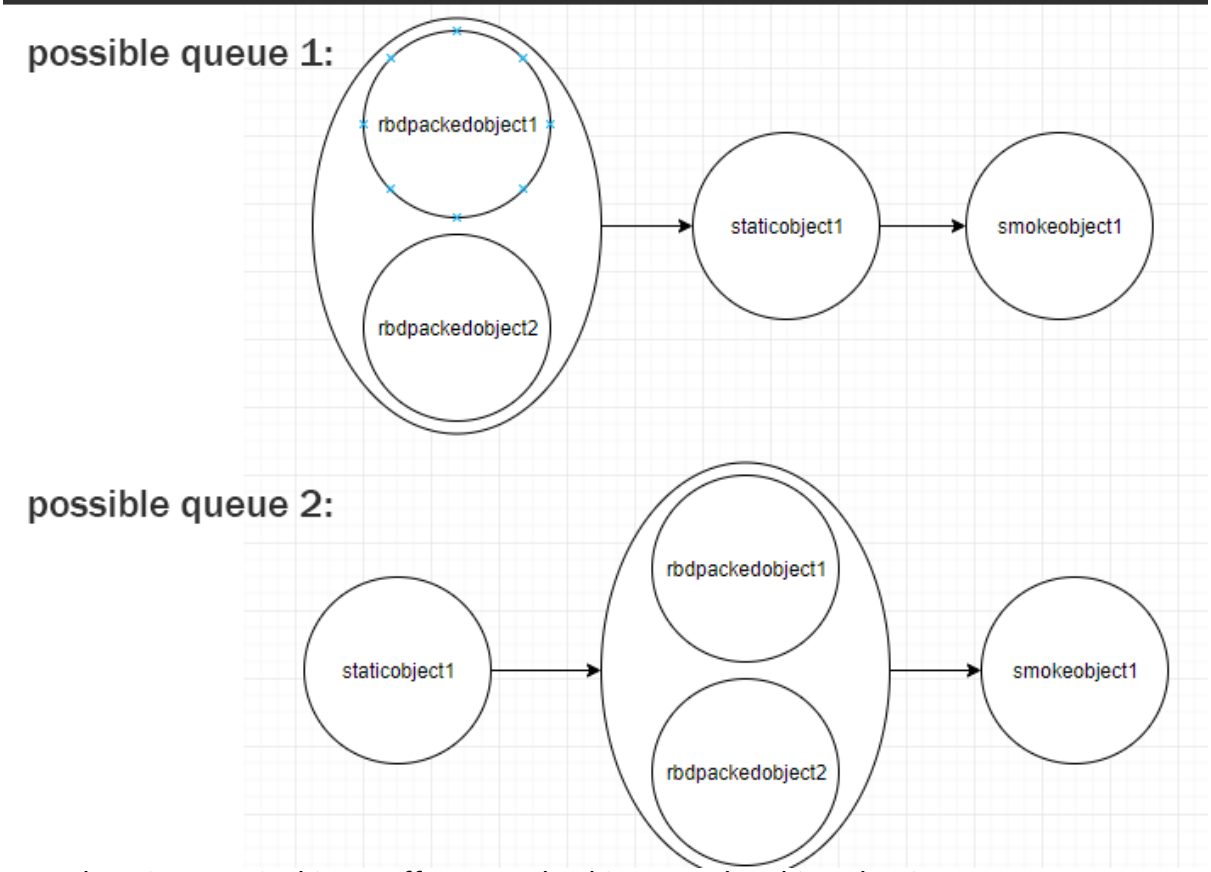
for example, in the classical case, the object’s statics and smokes, which enter the merge one after the other, with collide relay and in the “objects on the left, affect objects on the right” mode, the order of solving objects is obvious.



In the case of mutual influence of objects, the Houdini organizes the solving line as follows. (merge_with_default_colliderel node - the merge node in the default version, which creates a collide relayship, where the left objects affect the right, the merge without_rel node - does not create relayships)







Note that since staticobject1 affects smokeobject1, and nothing else, it can appear in the queue of objects in different places. However, the Houdini will selectively choose one of these options. Different queues (at different timesteps, for example) for the same simulation structure cannot appear.

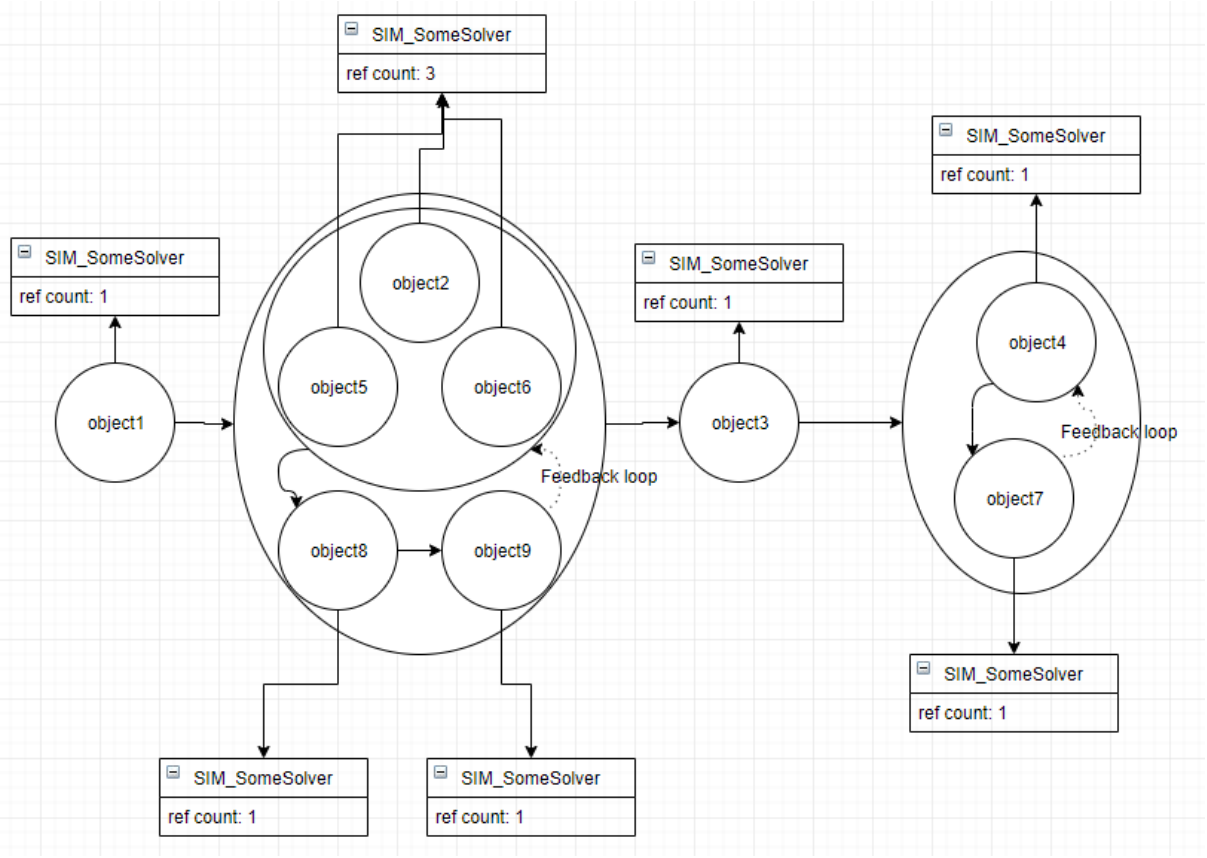
(example: **sppogijehu @ HPaste** *(OBJ context))*

Feedback loops

If all objects from the mutual interaction group have a link with the name Solver to the same data (we will say: they have the same solver, or refer to the same solver), the functionality of this solver will be launched once with the list passed to it all objects from the group of mutual interaction. If in the group of mutual interaction there were objects with different solvers (note that we are talking about different data, not different types of solvers, that is, objects can have the same type of solver, for example, bullet, but these solvers can be different data in

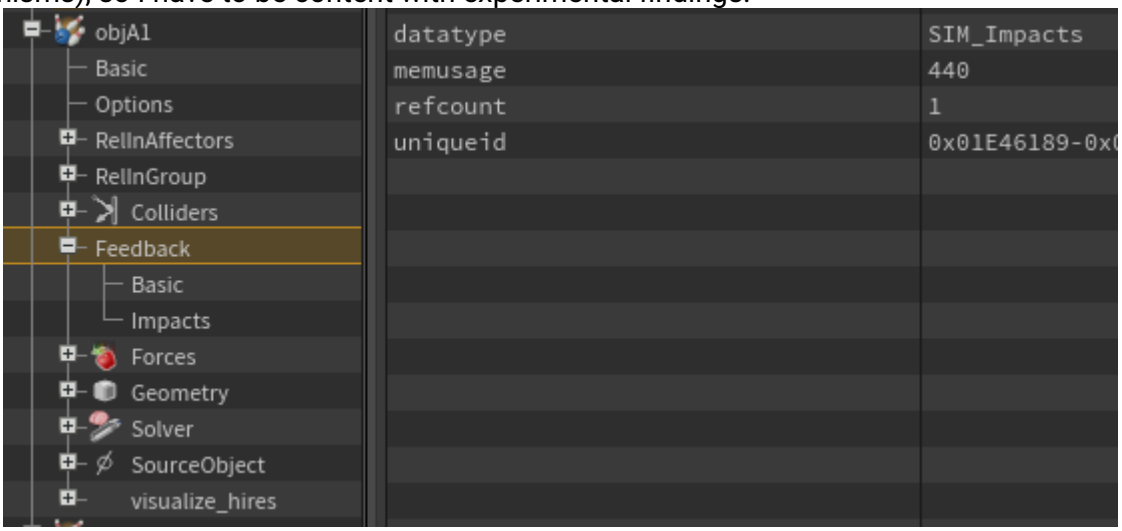
memory), the objects from the group will be serialized, and their solvers are launched sequentially, potentially several times, forming a feedback loop.
So, after the stage of computing the graph of nodes, we have a data structure in the current timestep. The order of attachment of sub-data to the data is strictly defined, and the data knows it, however, if you look through the geometry spreadsheet, the sub-data can be displayed there in a different order, often simply sorted alphabetically, if you change the sorting method to sort in the context menu of the parent data none - data will be displayed in the order of their attachment.
After the queue is drawn up, the solvation of objects is carried out by a much simpler and more stable method than the calculation of the node graph; it can be described literally like this:

1) if the calculation queue is empty, end the stage
2) take another item from the queue
    a) if the element is an object, it searches for data with the name Solver of a derived type from SIM_Solver, and starts for the current object
    b) if the element is a group of objects of mutual influence, then the objects of this group will be divided into subgroups that have a link to the same solver data. The feedback loop counter will be reset to zero, all data with the Feedback link name will be deleted from all objects in the group .
        i) A common solver will be launched for each subgroup in a deterministic order, with a list of all objects of the subgroup transferred to it.
        ii) If the feedback loop counter reaches the maximum allowed value set on the DOP node, this item is skipped. Otherwise, the triggers for the callback mechanism will be checked:
        If data of the SIM_Impacts type with the name of the Feedback link is detected on one of the objects of the current mutual influence group, moreover, during this feedback loop (points i and ii) if there were no such ones on the same object data, or the number of impacts was strictly less than now, then the feedback trigger is considered to have worked. When the feedback trigger is triggered, the simulation will be returned to the state before i) with the Feedback data saved, the feedback loop counter will be increased, and we will return to i)
3) return to 1)



(example: **sppawimeka @ HPaste** *(OBJ context))*

Perhaps there are other feedback loopback triggers, but, unfortunately, I could not find any documentation on this subject (even less than on other undocumented mechanisms), so I have to be content with experimental findings.



A few words about common microsolvers

Houdini microsolvers are called solvers whose functionality performs some narrow subtask, so it makes no sense to use them as a solver for an object. Microsolvers are made to collect hierarchical trees from them, which together would solve one large problem.

We know that at the solving stage, the Houdini searches for only single data on objects with the name Solver and launches their functionality, therefore there are a number of auxiliary solvers whose purpose is to launch their sub-data solvers under certain conditions

Multiple solver

Solver container, its functionality consists only in launching the functionality of its sub-data such as solver sequentially in the order of their attachment. through the use of this solver, a series of successively launched solvers can be attached to the object

Switch solver

A solver that starts one of its sub-data solvers depending on the value of its field, or a field on other data - allows you to customize the behavior of the solver, run different sub-solvers depending on the results of calculations of previous microsolvers or on the object being calculated as such. If a switch is supplied with a solver for calculating a list of objects, for each of them it will check the conditions, group the objects according to the selected conditions, and start the corresponding sub-solvers with filtered lists of objects

Enable solver

It is very similar to a switch with a solver, it allows you to start or not to start the solvers attached to it, depending on the condition calculated during the solving. Similarly to a switch to a solver, if an enable solver receives a list of objects for calculation, it will filter out those for which its condition is fulfilled and start its sub-solvers in a row for this filtered list of objects

Blend solver

The Blend Solver starts the solvers attached to it and interpolates the given data from the results of the attached solvers. Interpolation weights are set by special SIM_BlendFactor data, one factor for each attached solver. It is described in more detail in a help, I do not think that this solver is especially common and used, so I see no reason to dwell on it longer.

Static solver

Solver who does nothing. Useful when you need the formal presence of a solver, but you do not need any effect from the solving. For example, it may come in handy in conjunction with a switch solver to switch solving to void, or a blend solver to control the influence of another solver.
A separate topic is why they like to create a static solver on static objects (static object) - a static solver is not needed there. Perhaps there are historical reasons why it is still created in the presets of standard shelf instruments, but everything will work the same without it. The only thing that he does is by default creates a mutual relayship to all objects included in it. However, given that it does not affect objects in any way, the meaning of this mutual reliance is not clear. If someone knows a good example of a situation where a static solver is needed (apart from the cases with switch and blend solvers), write in the comments.

Solving Relayships

As you remember, relayships do not specialize through subclasses of the SIM_Relationship class - all relayships are specified by the SIM_Relationship class data, as all objects are specified by SIM_Object data - relayships specialize by the data referenced by the SIM_Relationship framework.
This data, and not the SIM_Relationship base framework, can have subjects with the Solver link name of the SIM_Solver subclass - then the buzzwords, just like for objects, will execute the functionality defined in these solvers.
The general outline of the startup order, however, is simpler.
All solvers found on the given relayships will be launched BEFORE solving objects .
The launch order is arbitrary, even for relayships, which could appear in the object queue in a strictly sequential order, the order of launch of their solvers can be any, not even determined by the order of creating the relayships, but by alphabetical sorting of the names of the relayships.

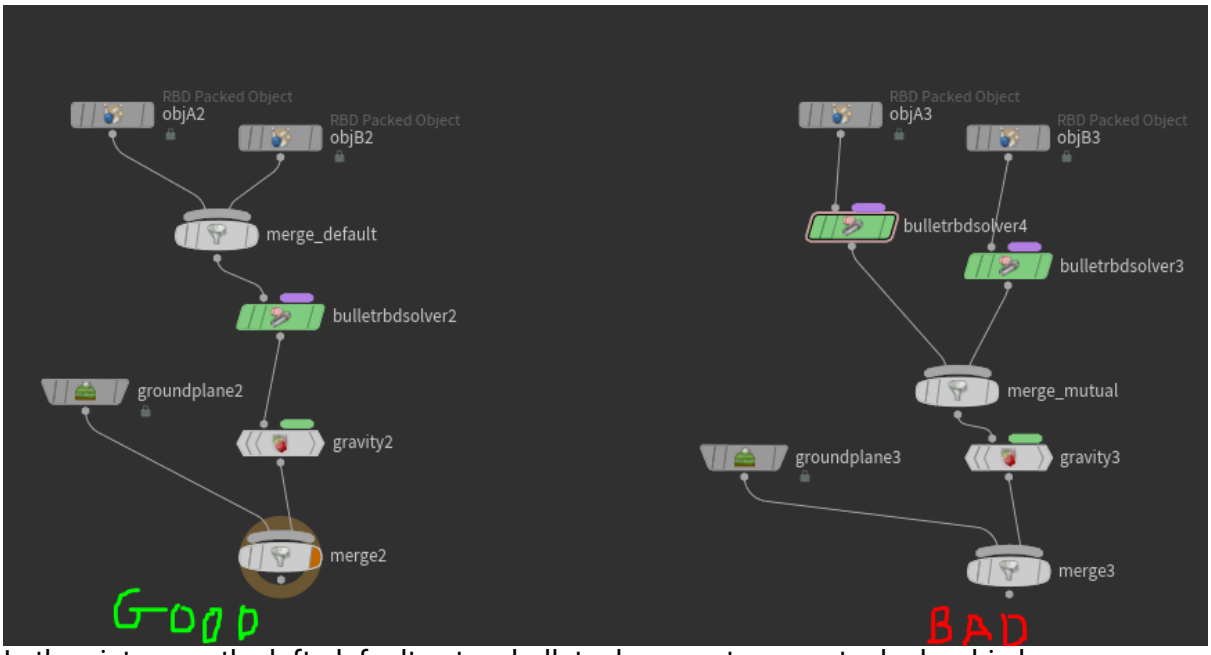
It is worth noting that DOP distinguishes between the functional of solvers for objects and for relayships, that is, the same solver can theoretically behave completely differently, whether it is subject to the object or relayship.
It is also worth noting that the functionality for relayships does not imply a call for a list of several relayships (in contrast to the functionality for objects, as described previously), i.e. if several relayship sub-data are referenced to one solver, the solver data will be copied to unique for each of the relayships when the solver starts.

(example: **spplatituy @ HPaste** *(OBJ context))*

Tips and Tricks

Common Solvers

As can be seen from the above, common solvers on several objects that create mutual relayship of these objects are sometimes a key structure for the correct and optimal solving. So, for example, a default bullet solver is created exactly so that it can solve all objects together in one call and without feedback loops. As we remember, the functionality of the solver is called once for a list of objects - these objects must be simultaneously 1) mutually affecting; 2) have a link to the same solver data.



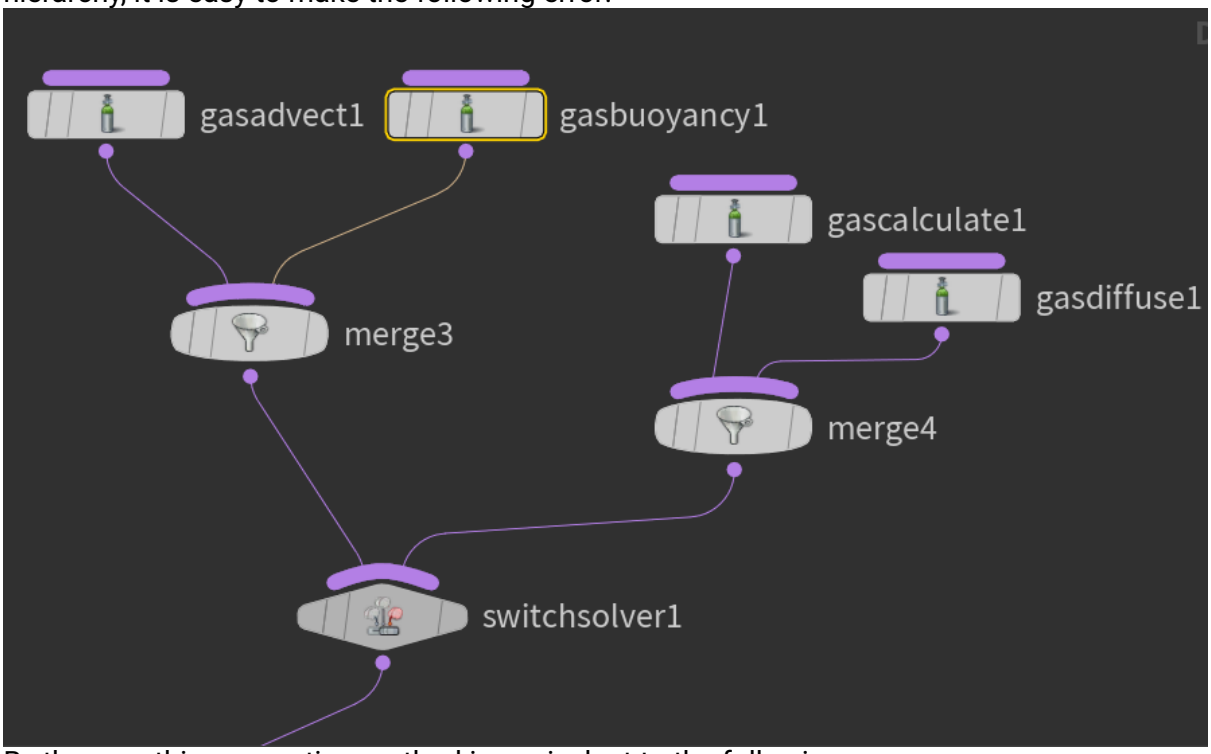
In the picture on the left: default setup: bulletsolver creates a mutual relayship by default, and applies a link to the same solver to both objects. As a result, one solver solves both objA2 and objB2 objects in one call, which means that he can internally deal with the mutual interaction of these objects within his internal substeps.
In the picture on the right: a less likely, but still meeting setup. (A similar effect can be easily achieved from the left setup by simply turning on the Solver Per Object checkbox). In such a setup, mutual reliance will still exist between objA3 and objB3 objects, but objects will have links different instances of the bullet solver. This, as we know, leads to the use of feedback loops mechanics, and we can observe the appearance of Feedback data on objA3 created by the bullet solver of the objB3 object. Now, instead of solving two objects at a time, each instance of the solver bullet will be executed with only one of the objects. First, one obA3 will be calculated, then one ojbB3 with calculated objA3 as a static collider, it will calculate the necessary impulse, which must be applied back to objA3 and attach this data to objA3 with the name Feedback. After that, the Houdini will return the objects to the state before the solv, retaining Feedback and

go in a circle: launch the objA3 solvo, etc. It is easy to see what terrible results this approach leads to: in fact, we forbid a bullet to use its own mechanism for calculating the interaction of objects, and make 2 bullet solvers communicate through the common mechanisms of buzz.
And with the minuses of a setup with a common solver, one can only say that when traversing a node graph, the parameters on the node will be calculated only the first time during the traversal in one timestep, each subsequent calculation will simply return this stored data (as in the Share Data In mode One Timestep, as discussed in Data Sharing), so that varying any parameters of the solver for each individual object in this way will not work. However, it is still possible to customize a common solver for different objects using auxiliary microsolvers, such as switch solver and enable solver.

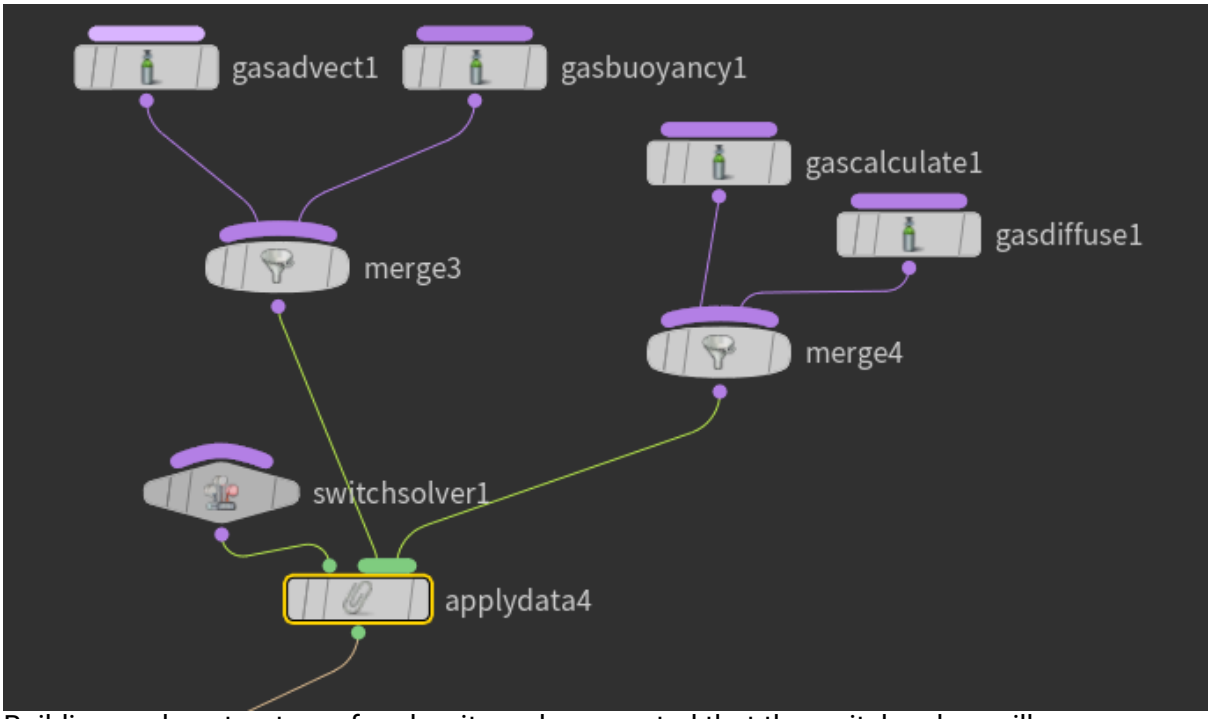
(example: **sppnaqoban @ HPaste** *(OBJ context))*

Switch solver

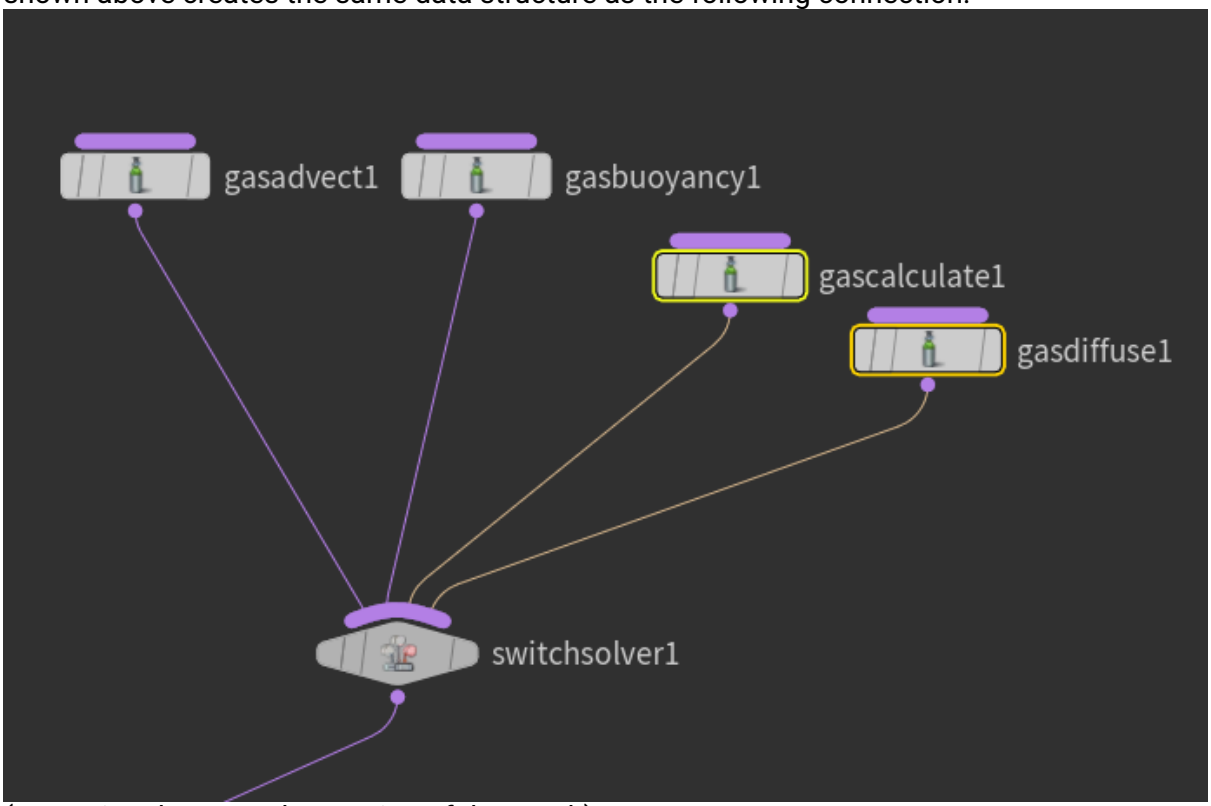
It is important not to confuse Switch and Switch Solver. We already know that Switch is a node that is necessary exclusively to determine the structure of a graph during a traversal. Switch Solver is a solver, and it, as it should be, is triggered in the solving stage. The functionality of this solver is to run the functionality of one of its solver sub-data. Sub-data are considered according to the account of addition, the desired number can be specified either as a field on special switch data, or on the switch solver itself.
Due to the fact that the node graph does not correspond to one in one data hierarchy, it is easy to make the following error:



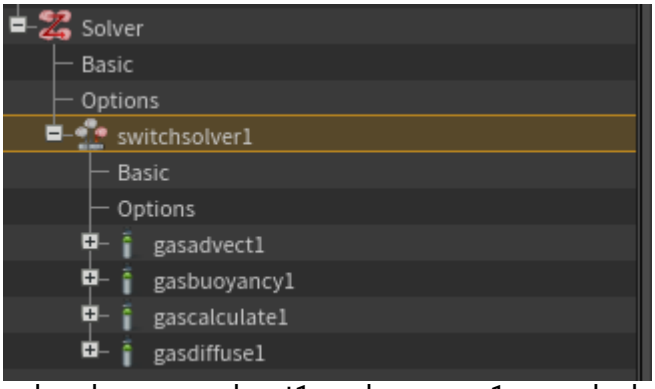
By the way, this connection method is equivalent to the following, as we discussed earlier:



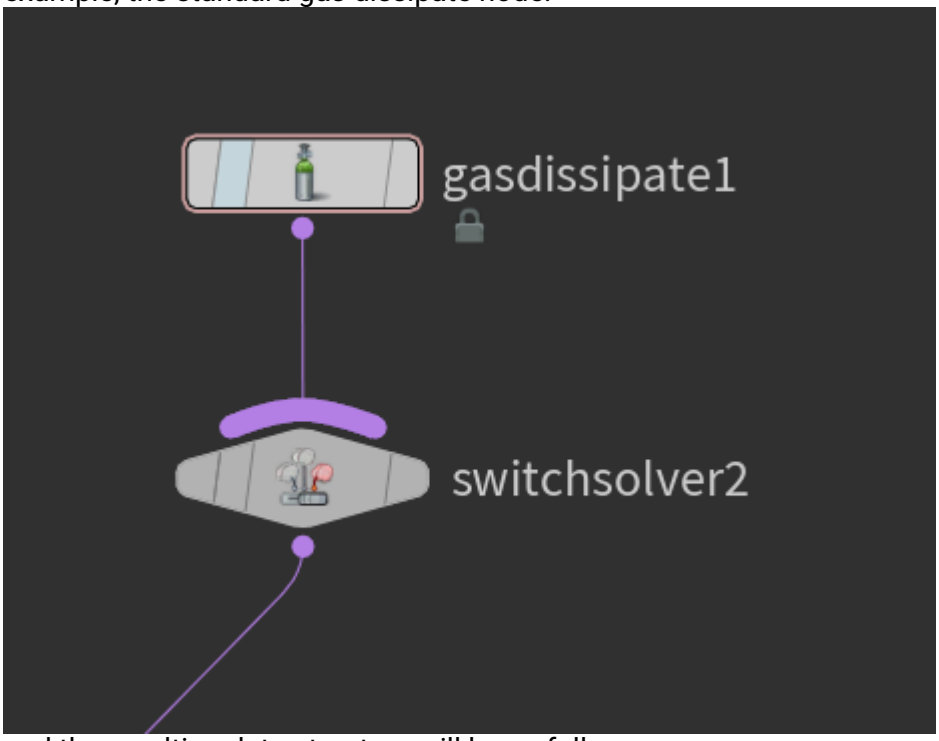
Building such a structure of nodes, it can be expected that the switch solver will switch between groups of solvers included in it. However, we now know how the evaluation of the node graph occurs, and we know that the connection method shown above creates the same data structure as the following connection:



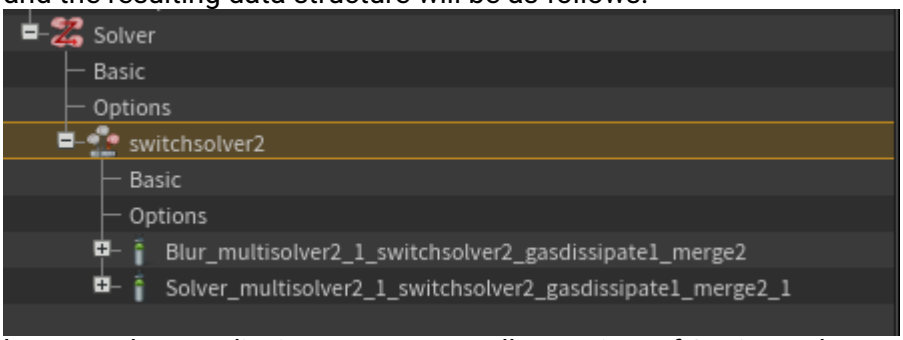
(assuming the normal operation of the graph).
Here is the structure they created



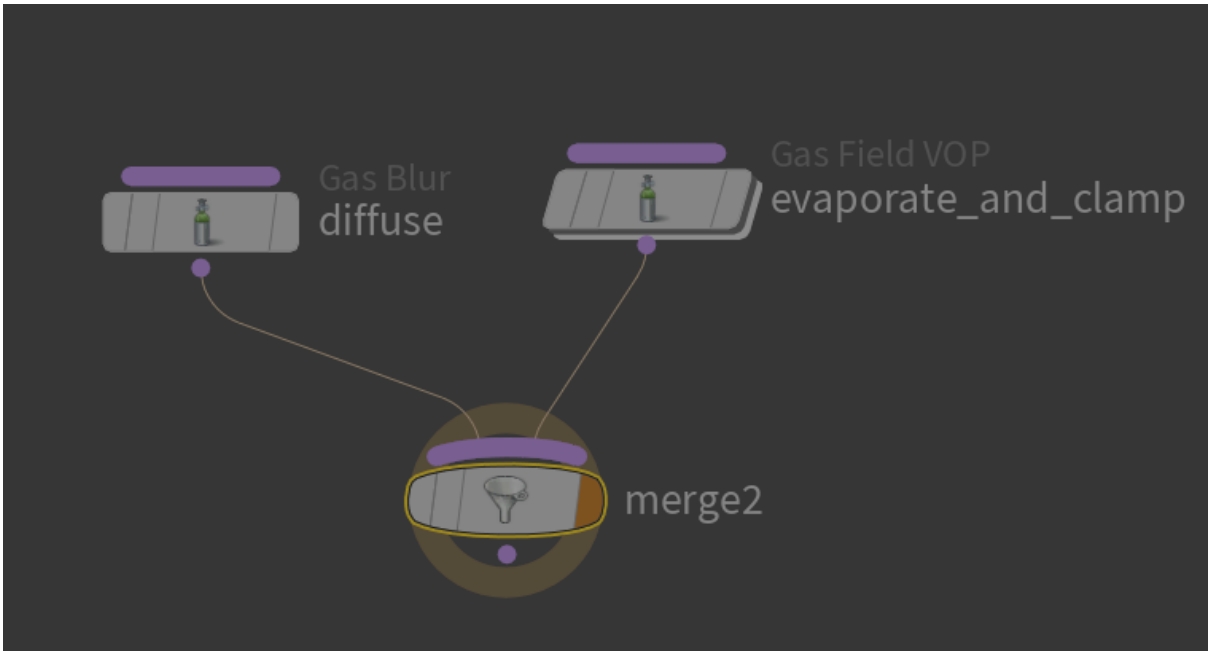
sub-solvers gasadvect1 gasbuoyancy1 gascalculate1 and gasdiffuse1 will be sub-data of the solver switch, therefore it is from them that the switch will choose whose functionality to run, depending on its switch value.
it's even easier to make such a mistake when working with assets, such as, for example, the standard gas dissipate node:



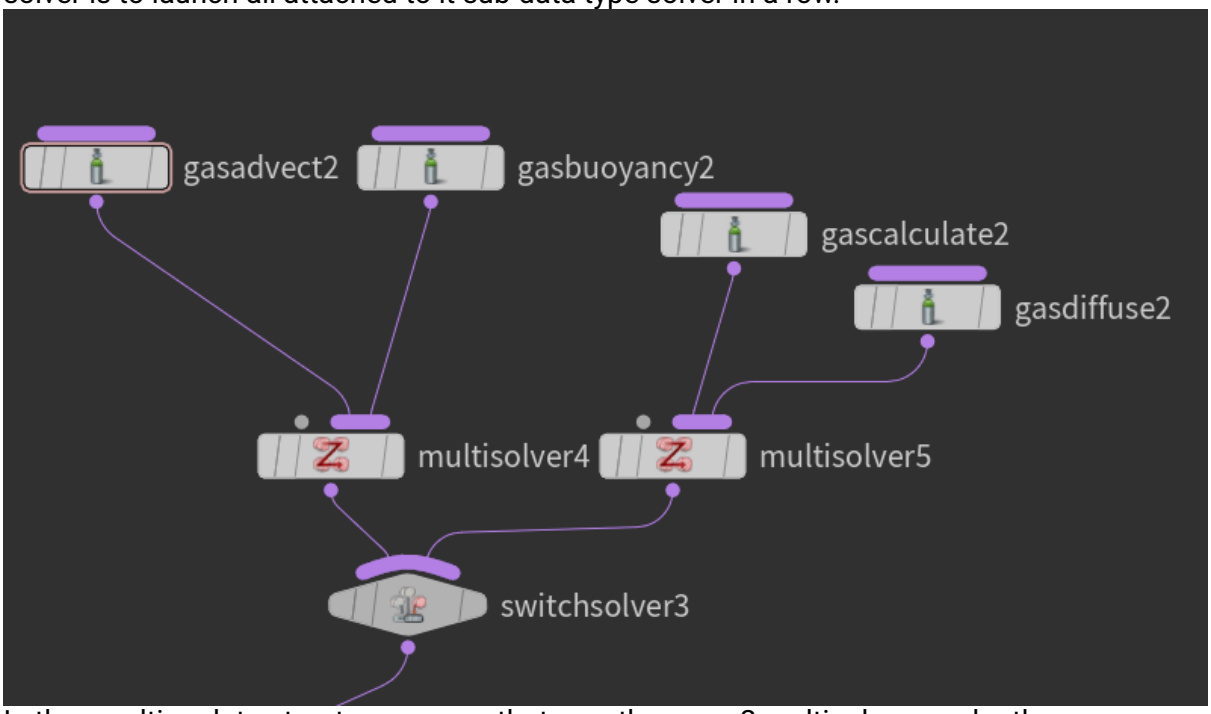
and the resulting data structure will be as follows:



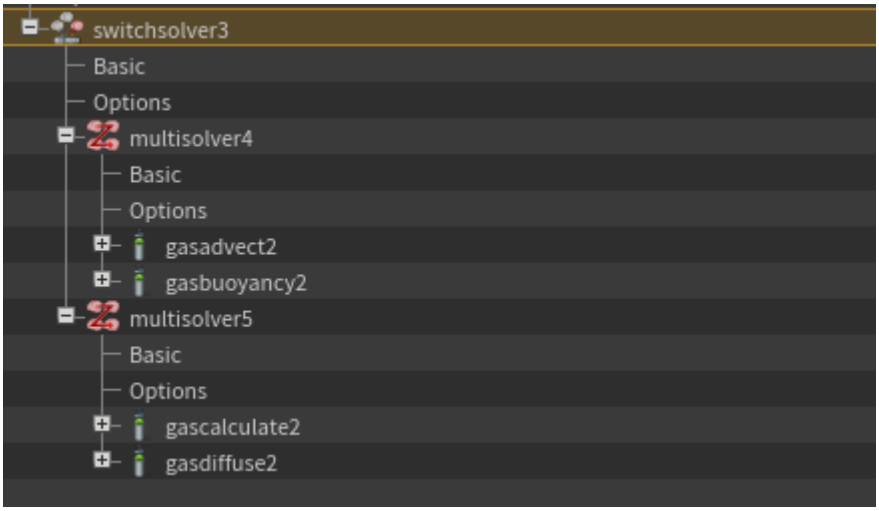
because the gas dissipate asset actually consists of 2 microsolvers united by merge:



.
If we want to group the sub-solvers of the switch solver in a similar way to the first picture, we need to use some intermediate sub-solver, for example multiple solver, which, as we already know, is a solver container, whose function as a solver is to launch all attached to it sub-data type solver in a row.

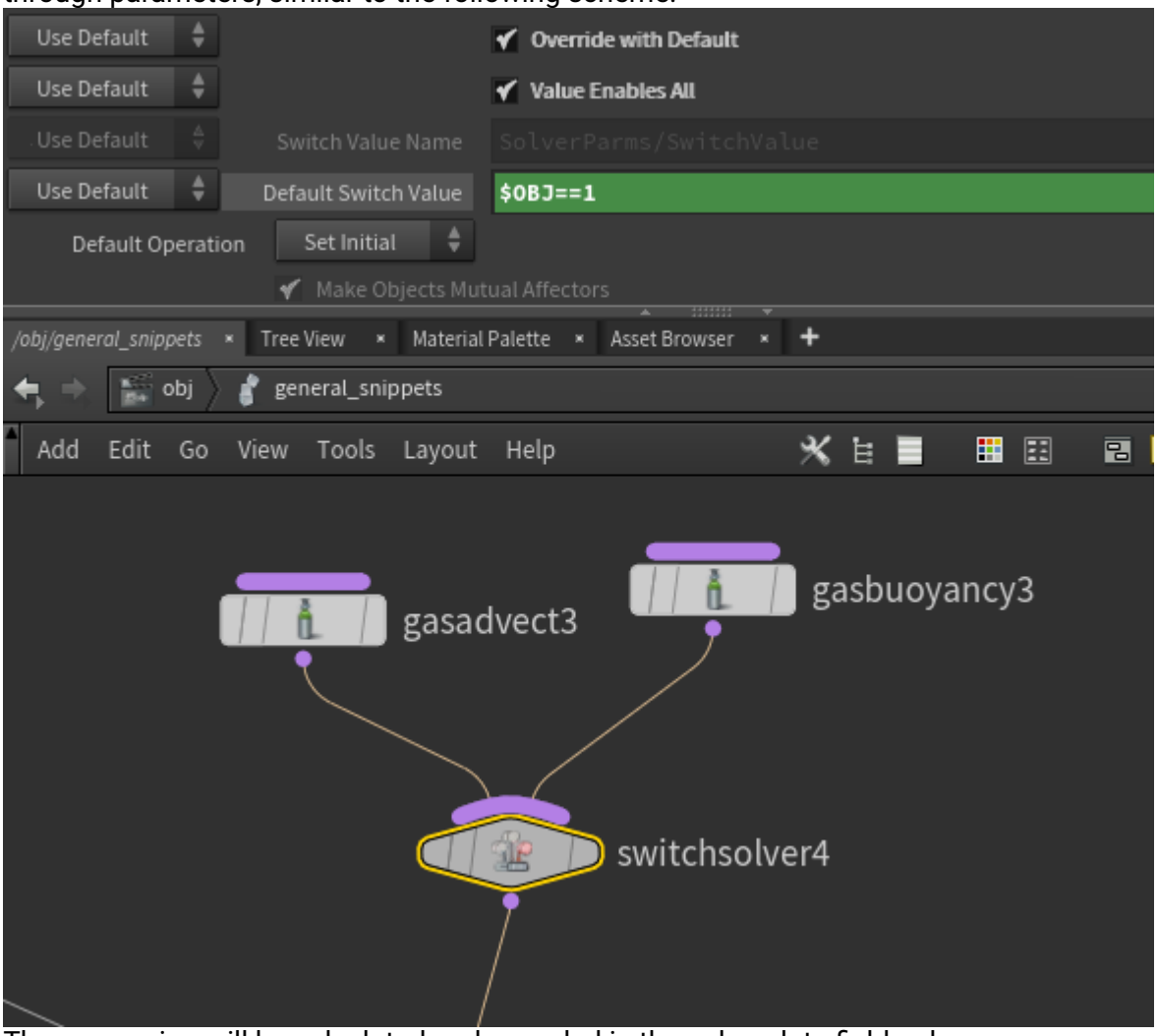


In the resulting data structure, we see that now there are 2 multisolvers under the switchsolver3 subdata, and the switch will occur as expected between them



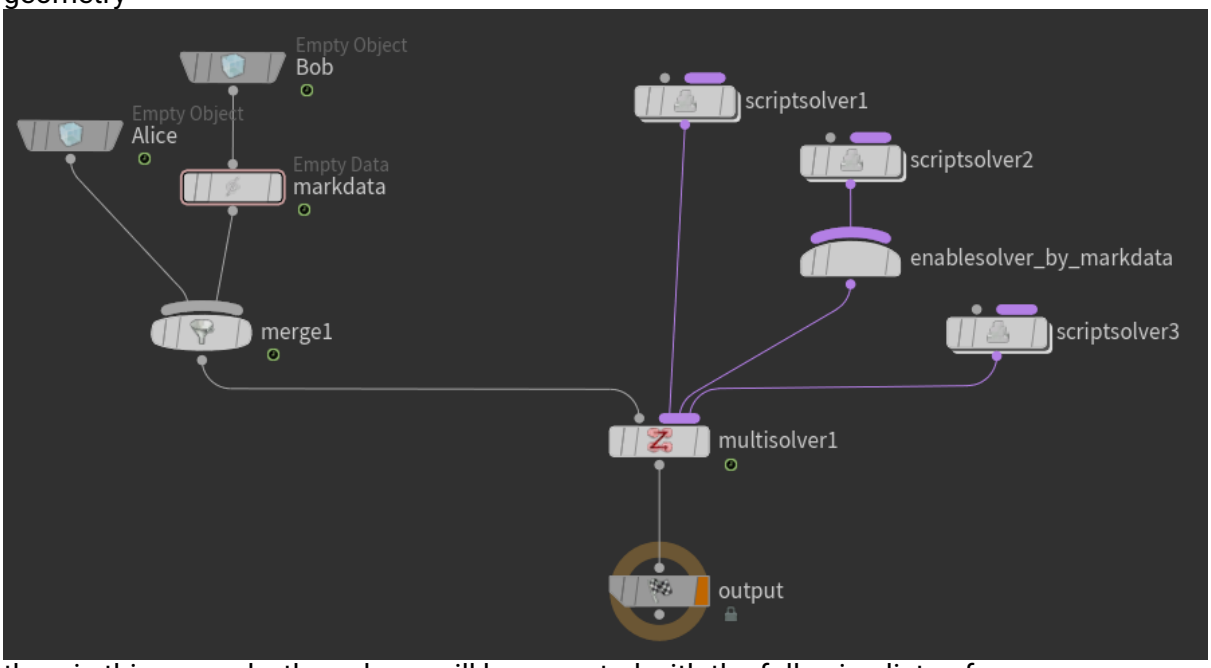
Enable Solver, Switch Solver. customization of a common solver without the need for duplication of the main solver on objects

we now know how in some cases it is important to have a common solver on objects connected by mutual reliance, which means that for all objects the same solver will be executed, without the possibility of changing and customizing through parameters, similar to the following scheme:

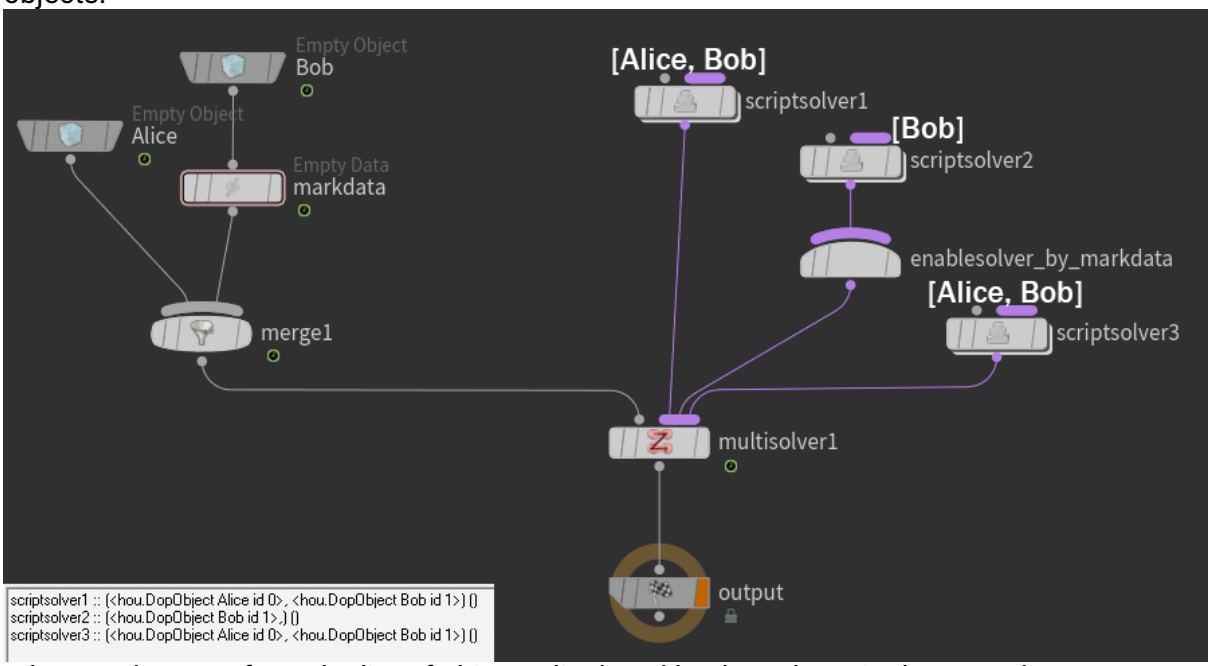


The expression will be calculated and recorded in the solver data field only once during the stage of computing the node graph. (as we recall, a solver node with the daw solver per object removed behaves like a data node in the Share Data In One Timestep mode)

However, we can still customize the solver for different groups of simultaneously processed objects. Solvers Enable Solver and Switch Solver, in addition to the most commonly used functionality for launching or selecting subjects according to the default parameter, which is recorded in the field solver from the node parameter, can also operate with data.
For example, the switch solver can be switched on the value of some third-party data on the object, not on the solver, and the enviable solver can be turned on both by the presence / absence of data with a specific name on the object, even by the presence / absence of geometric attributes in this data, if they are geometry



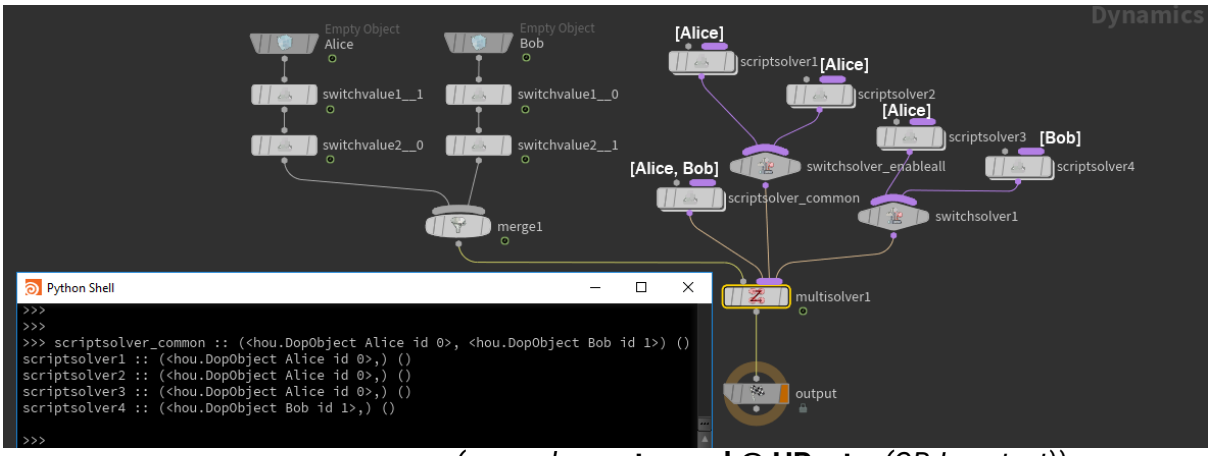
thus, in this example, the solvers will be executed with the following lists of objects:



What can be seen from the list of objects displayed by the solvers in the console
(example: **sppnuyedot @ HPaste** *(OBJ context))*

Similarly, you can use the switch solver
The standard switch solver uses data of a special type SIM_SwitchValue that stores the switch field in the Options record. Switch Solver in default mode of operation looks for a link to the switch data with the given name on the objects being processed, and interprets the switch field as the serial number of the sub-

data of the solver type that should be run for each of the objects, groups the objects for each of its sub-solvers and starts them for each corresponding group of objects
Switch solver in value enables all mode does not interpret the switch field as the serial number of the sub-solver, but instead starts all the subsolvers for objects with the switch field on the switch data non-zero.



(example: **spptezanol @ HPaste** *(OBJ context))*

However, these methods of specializing the behavior of a common solver only work for customizing solvers assembled from microsolvers, for customizing the parameters of monolithic solvers, such as, say, a bullet solver, this method is not applicable. However, in the case of a bullet, all parameters that can vary from object to object can be set both at the object level and at the level of geometry attributes.