Topological naming implementation

Introduction

This page describes an experimental approach to solve the topological naming problem. In general it can be described as following:

> Enable the identification of subshapes within a shape and its modifications in a stable manner.

This means a vertex shall have the same identifier in its original shape as well as in e.g. a fused version of the shape, no matter if they are the same vertex in memory or just geometrical equal.
Next to the problem definitions the use cases for such identifiers are important for deriving a design. There are two for general identifiers:

PythonAPI: One often has subshapes in a certain state of a script (e.g. a vertex created from a Vector) and later, after some additional modeling steps (e.g. extrude vertex to edge, edge to face) one wants to find the exact same vertex in the created face. It shall be possible to use the initial vertex‘s identifier to retrieve it from the face

DocumentObject: When used in a document scope the identifiers must not only be the same for shapes and its modification, but also between all complete recomputes of the shape. Hence when rebuild multiple times, even if the build history is slightly different for other parts of the shape. Identifiers must stay consistent.

Finally one can make an additional use case for identifiers which is not related to topological naming, but which would be highly valuable: Make the identifiers intuitive constructable. This translates to a third use case:

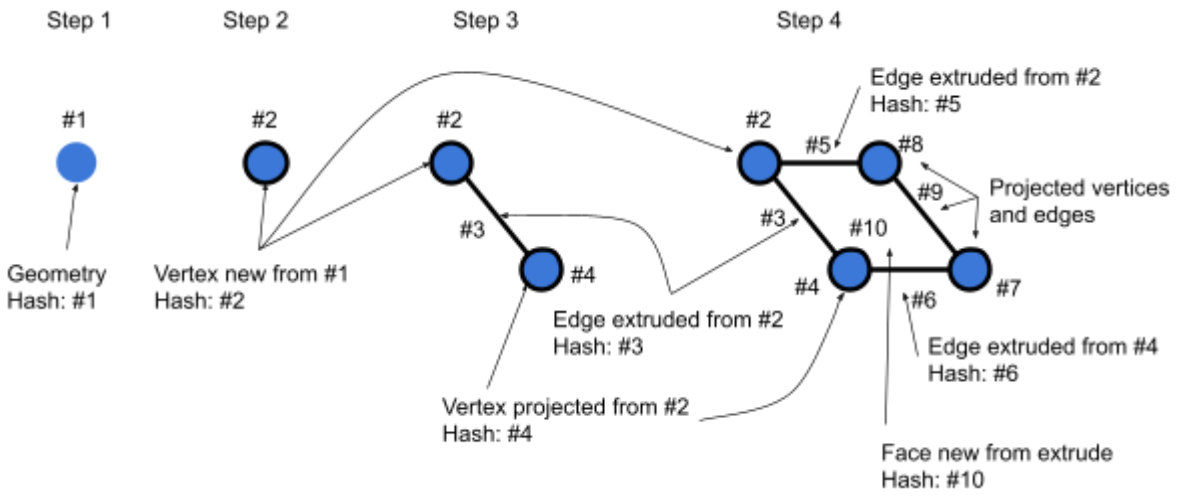
Shape finding: During scripting finding specific subshapes is hard, even thought the user exactly knows in his mind which subshape he wants. For example extruding a edge, and you know you want to have the edge that resulted from the extrusion of vertex1. This is easily imagined, but not usable to find the edge within the current API. To solve this one could make the identifiers represent the users intend in some way so that one could build an identifier by intuitive criterias and hence find subshapes this way.

Lets summarize the main criterias for a implementation:

- Create unique identifiers for subshapes which stay stable during all modifications of a shape
- The identifiers shall be stable for full rebuilds of the shape and all its modifications
- The identifiers shall be usable to identify subshapes by design intend

The chosen approach

The chosen approach is based on identifying subshapes by their creation history. The general idea can be roughly depicted by the following sequence of creating a geometry, making a vertex from it and than extrude it two times to get a face:



So if each subshape has an unique identifier, here called hash, than each newly generated subshape can be named individually too. The information to make up the identifier is the following:

Shape: Is the identified subshape a vertex, edge, face? This information may be redundant as it can also be extracted from the subshape the identifier is added to. However, it seemed convenient to have this information in the identifier to have one more distinguisher. For example it could be that an operation creates new subshapes that have no base. Than having the Shape increases the chance to make the identifier unique.

Type: How is the subshape created. Is it “New”, e.g. created from no base, is it “Generated” from the base, “Merged” from other subshapes? The information of creation is stored here. This is the place to store info generated from OCCs naming algorithms in topological operations.

Base: All identifiers, the new one is based on. In the example above the vertex depends on the geometry. This part of the identifier builds the tree structure of dependencies between the dependend subshapes. Note that Base is a vector, as there can be multiple ones. For example when fusing two lines there may be too different vertices merged into one. The resulting vertex identifier has two bases, the ones that have been merged .

Name: Sometimes shapes can get special names that describe their purpose. For example the OCC box creation allows to identify “Top”, “Bottom” etc. faces. But also things like “Start”

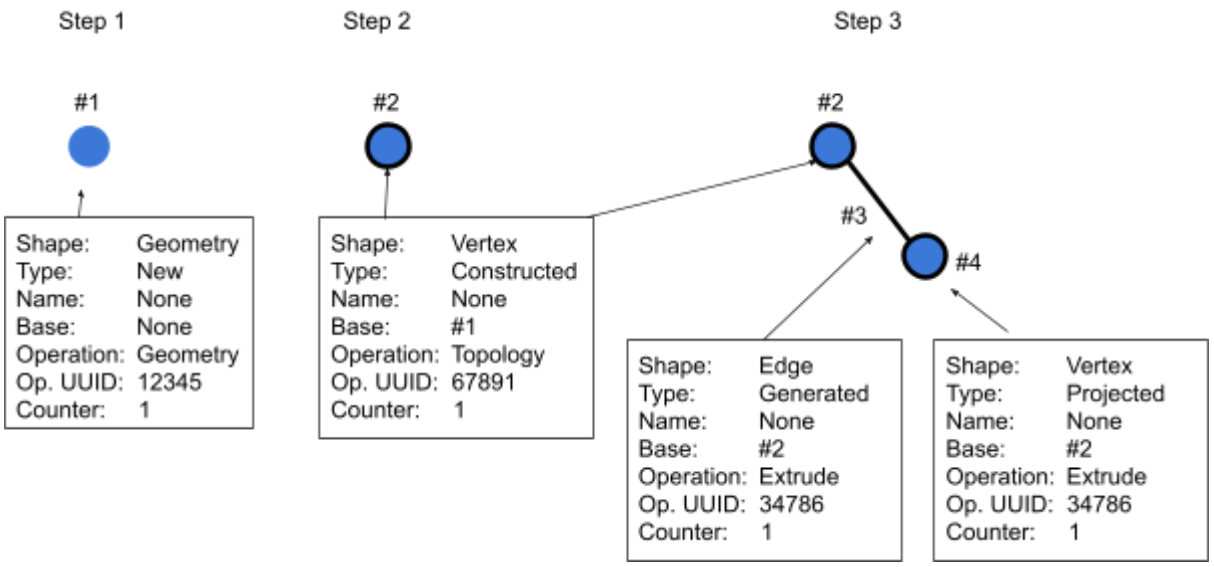
and “End” vertices of an edge can be used to create unique identifiers. This is especially important for shape creation methods. When rebuilding one must ensure that the same subshape gets the same identifier again, and there things like “Top” fce is very welcome.

Operation: The operation used to create a subshape is an important information to capture the design intent of it. It allows not only to improve uniqueness but also is highly usefull for finding subshapes.

Operation UUID: As seen in the example above it could happen that there are identical identifiers, e.g. two times Edge extruded from Vertex #2. To overcome this problem the opertions get not only a name, but also an universal unique identifier, a UUID. Than it becomes clear thet those two edges extruded from #2 are different, as they are created from two different operations. However, this also creates an issue for DocumentObject use. When rebuilding everything there the Operation UUID must stay the same for each rebuild to ensure that the resulting identifiers are the same as before. Hence it must be stored in the DocumentObject itself and be applied to the identifier.

Counter: This is a rescue information. Some operations may create two entirely new vertices that cannot be named individual. Than, to distinguish both, they get a different count. For rebuild scenarios one must hope that the count will be assigned the same, but that is pure luck. Hence better to avoid to use this.

For the given example above this would like the following:



Notes

- This approach implies that every subshape needs to have an identifier, not only the ones you are interested in. This is because even for document object use you never know which one may be later used for selection, and t make everything well defined you need everything to have a identifier.
- It is easily possible to use unique shapes to define others, that have no unique property. For example one can identify all edges of a cube with the named faces, as the edge is fully defined by the two faces it is shared by.

Hashes

When using identifiers it i not practical to carry around the whole data structure every time. Hence hashes are used to allow to reduce an identifier to a string. This would also allow to use identifiers within the current FreeCAD link system which is string based. The question is how one could derive a hash from the identifier information. The chosen approach is to create a unique string from the given identifier information. This string is then cryptographically hashed.

The current implementation allows to print information for an identifier as a history:

```
p = Part.Point(App.Vector(0,0,0))
v = Part.Vertex(p)
v.printHistory()
```

The console output is (Hash and operation UUID are shortened):

```
Constructed Vertex build from operation Topology (01e19… 48) based on
        387808...77646{ New Geometry build from operation Geometry (60...979)}
```

This string holds all information that can be used identify the subshape (excluding default values for name etc, as those don't help in the individualisation). One can see that also the base identifiers are printed fully behind their hash. This is only the case for printHistory().

To create a hash the the following is used:

```
Constructed Vertex build from operation Topology (01e19… 48) based on
387808...77646
```

This results in the hash '17490584643799743444', which can be accessed in the current implementation via

```
v.Reference
```

As eachs identifier hash uses the hash of its base identifier it works like a blockchain.

How to utilize the information

I has been shown which and how information is stored and how it is used as a reference (with the hash). Now for handling topological naming there are a few more involved issues when try to find a shape in a later modeling step. Let's see how to use this in the intialy defined use cases:

Python API

To access the information of the subshape identifiers the following useful functions are provided. Comprehensive information can be found in the functions documentation with python help command, e.g. help(v.findSubshapes) :

```
printHistory(...)
findSubshapes(...)
subshape(...)
getModificationsOf(...)
getGeneratedFrom(...)
getMergedFrom(...)
getConstructedFrom(...)
```

Those functions allow to get a subshape by a known reference or query subshapes by their creation method. This allows the user to translate the intuitive subshape description he has in his mind into a way to find those shapes.

```
p = Part.Point(App.Vector(0,0,0))
v1 = Part.Vertex(p)
v2 = Part.Vertex(2,2,0)
e1 = Part.Edge(v1, v2)
e1.subshape(v2.Reference)  #get the vertex subshape in edge that represents v2
e1.getConstructedFrom(p.Reference) #get the vertex constructed from point p
```

One special problem are fusions. When two subshapes are at the very same position they get merged into one, for example the corner vertices of two edges merged into a wire. Now if the user wants to find the vertex of edge1 back in the fused shape he will not find it. But he can use the information we store in our identifiers:

```
l = Part.LineSegment(App.Vector(2,2,0), App.Vector(2,0,0))
e2 = Part.Edge(l)
w = Part.Wire([e1,e2])
w.getMergedFrom(v2.Reference)
```

Document Object

Note that this has not been implemented. But one has to think of the special problems arising with the document object use. For one the mentioned Rebuild stability, but this is easily solved with the presented approach and an stable operation UUID within the document object. Annother major one is how to handle the reference retrieval. As shown before it is simple to get a reference that still exist, however, it may be more complex. On rebuild the once existing subshape, that has been used for selection, can be

- Merged into annother shape
- Split into multiple subshapes

Both cases are handable by the presented data structure, and a function which check all those cases when retrieving a reference can easily be added.

Implementation Details

Let's have a few words about implementation. Note that in the text above the word identifier has been used. However, the class to handle all the described functionality is called “Reference”. Sorry for the inconvenience.

Reference class

One critical point is how to get the information for building the identifiers. For this mostly the algorithm information must be utilized. For example currently nearly all creation constructors of vertex, edge and face have been ported. There one knows many things just from the fact that it is a constructor and hence can fill up the info. In general it can be said that only the shape creation algorithm has enough information to set up identifiers. Hence one can only provide helping functions to make this easier.

The class “Reference” provides static functions to easily setup everything individual References, prefixed with “build”:

```
static Reference buildNew(Shape sh, Operation op, Name = Name::None);
static Reference buildGenerated(Shape sh, Operation op, const Reference& base, Name =Name::None);
static Reference buildMerged(Shape sh, Operation op, const std::vector<Reference>& base,  Name=...);
static Reference buildModified(Shape sh, Operation op, const Reference& base, Name = Name::None);
static Reference buildConstructed(Shape sh, Operation op, const Reference& base);
static Reference buildConstructed(Shape sh, Operation op, const std::vector<Reference>& bases);
```

The references can than be stored inside the TopoShape class in a map, whcih links subshapes to Refernces.

Instead of building the Refernces one by one a whole TopoShape can be processed. For this functions with the prefix “populate” exist. This for one provides a mechanism to copy over References of unchanged subshapes

```
static void	populateSubshape(TopoShape* base, TopoShape* subshape);
```

as wella s from annother important source of information, OCC algorithms with their “Modified” and “Generated” methods. Other algorithms have special functions to access special generated subshapes. Those can be utilized with

```
static void	populateOperation(BRepBuilderAPI_MakeShape* builder, TopoShape* base,
                TopoShape* created, Operation op, Base::Uuid opID = Base::Uuid());
```

etc.

Annother importat way of naming subshapes is to use their relation to other named shaps. For example, if all faces in a cube are named, than the edges are perfectly defined by the faces it is shared from. And also all vertices can be defined like that. Those functions are prefixed with “name”

```
static void	nameVerticesFromFaces(TopoShape* shape, Operation op, Base::Uuid opID =
Base::Uuid(),  bool onlyUnnamed = true);
```

TopoShape

The toposhape class has been extended with functions to query subshapes by reference or by construction. It can fully handle the new Reference class. The link between Reference and subshape is achieved with a map. Note that saving and loading of the references is not implemented yet.

Phyton goodies

A function for comparing shapes for equal geometry has been added:

```
myShape.isGeometricalEqual(annoterShape)
```