Computer Aided Design with *Mathematica*

A proposal for extending *Mathematica*'s graphical prototyping into a geometrical one by introducing object topology

by Kristina Livesay

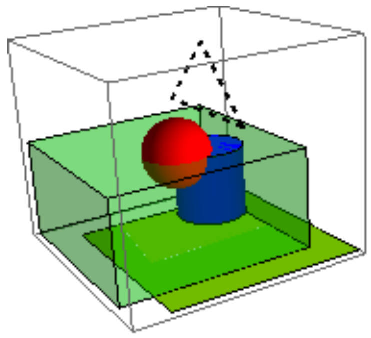
Introduction

A geometric modeling kernel is a 3D solid modeling software component used in computer-aided design packages distributed by companies such as Autodesk, SolidWorks and SolidEdge -- 3 of the leading companies in the CAD market. A geometric modeling kernel's capabilities include modelling and calculating of volumes, surfaces and curves to engineer a geometrical product. Having spent some time reading about how such CAD software components work, and attempting to guess and imagine what could be taking place behind the scenes, I have put together a proposal for a 'geometrical kernel', but the *Mathematica* way. The Wolfram language empowers this method of geometric prototyping I will be proposing to an extent that has not been done before. The final product, aspires to be powerful, multifaceted and easy to use, a feature that not all geometric modeling kernels accomplish.

Object topology and some

Introduction

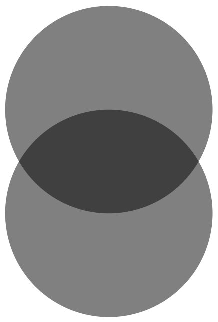
Graphics[] is like a 'window' in *Mathematica*'s set of typesetting objects: you can see through it but you can't access its world to bring it in and manipulate objects to the extent we perceive image manipulation in real life.



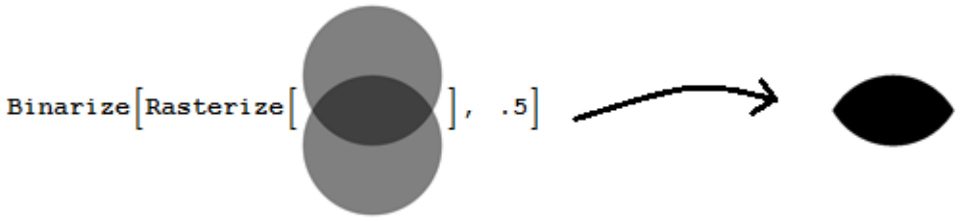
For example, the intersection of two graphics objects should output the planar or volumetric image that is contained in both graphics objects. The darker space below is what one should expect for some *Mathematica* function, say Intersect[GraphicsObj1, GraphicsObj2]:

```
Disk1 = Disk[];
Disk2 = Disk[{0, 1}];
```

```
Graphics[{Opacity[.5], Disk1, Disk2 }]
```



Currently, it is not possible to obtain a graphics object as a result of the intersection of two graphics objects. One may or *may not* work around this by systematically: rasterizing the graphics objects, getting an image data set of pixel information, *transform* them to a new set of points on which we may perform *Mathematica*'s Intersection[] or any other algorithm, inverse transform the result to byte information data, inverse the result to an image, inverse the image to a graphics. Here is an approximate result we want to see for the above image:



Representing graphics window objects as 'topological spaces' will resolve this issue. In addition to the representation of graphics objects as topological elements, more functionality will be proposed, such as construction of graphics objects from unionizing, intersecting, subtracting, boundary definitions, filling, hollowing, assembling...etc.

The topological model

A topological space, in the mathematical sense, is a pair (X, τ) where X is a set and τ is a family of subsets of X, whose elements are called open sets such that

(1) $\varnothing, X \in \tau$
(2) if $\{O_\alpha\}_{\alpha \in A} \subset \tau$ then $\bigcup_{\alpha \in A} O_\alpha \in \tau$ for any set A (the union of any number of open sets is open)
(3) if $\{O_i\}^k{}_{i \in I} \subset \tau$, then $\bigcap^k{}_{i = I} O_i \in \tau$ (the intersection of a finite number of open sets is open)

If $x \in X$, then an open set containing x is said to be an (open) neighborhood of x. To be consistent with the terminology, it is worth mentioning that τ is usually omitted in the notation and the pair is often referred to as just “topological space X”, assuming that the topology has been described.

Topologies are important in unifying ideas of mathematics and similiarly, it could be used to unify the idea of graphics objects in *Mathematica*. Let’s explore.

A set X is given as below:

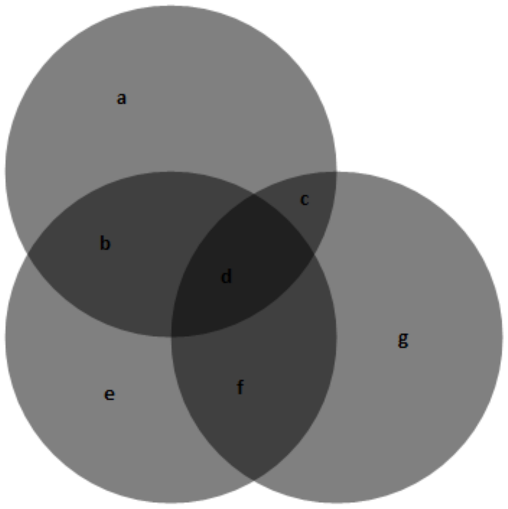
X = { DA, DB, DC } where,

DA = { a, b, c, d };
DB = { e, b, f, d };
DC = { g, c, f, d };

A topological space (X, τ) can be defined for τ described as below:

τ = { ∅, {a, b, c, d}, { e, b, f, d }, { g, c, f, d }, {b, d}, {f, d}, {c, d}, {d}, {a, b, c, d, e, f}, {a, b, c, d, g, f}, {e, b, f, d, g, c}, {c, d, f}, {b, d, f}, {b, d, c}, {b, d, c, f, g}, {c, d, b, e, f}, {a, b, c, d, f}, {a, b, c, d, e, f, g} }

You can easily verify that this indeed consists a topology; here is a better picture to correspond with the problem:



Also, one can rewrite the τ to better correlate with the definition: τ = {∅, {DA}, {DB}, {DC}, {DA ∪ DB}, {DA ∩ DB}, {DA ∪ DC}, {DA ∩ DC}, {DB ∪ DC}, {DB ∩ DC}, {DA ∪ DB ∪ DC}, {DA ∩ DB ∩ DC}, {DA ∪ (DB ∩ DC)}, {DB ∪ (DA ∩ DC)}, {DC ∪ (DA ∩ DB)}, {DA ∩ (DB ∪ DC)}, {DB ∩ (DA ∪ DC)}, {DC ∩ (DA ∪ DB)}}.

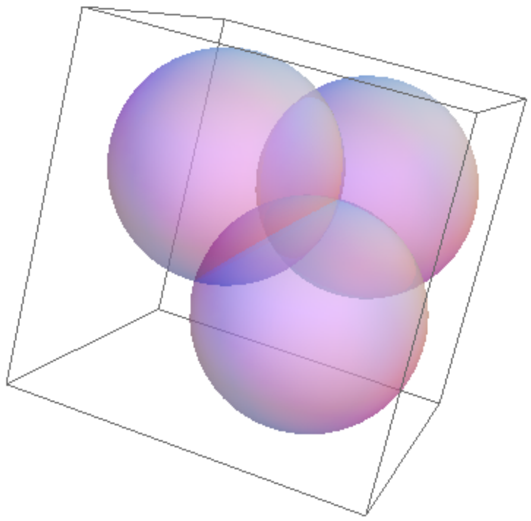
Note: ∪ and ∩ operations from set theory.

Similarly, I want to think of graphics objects as spaces labeled as above: a, b, c, d, e, f, g, as opposed to euclidian spaces. The picture above identifies labels with the color coded space and I want that space to be the ‘grain’ or the building block in my topology. [I will reveal slowly why I want to think of them at this level of abstraction].

3 spheres are defined as below

```
DA = Sphere[];
DB =  Sphere[{0, 1, 1}];
DC =  Sphere[{1, 0, 1}];
```

```
Graphics3D[{Opacity[.5], DA, DB, DC}]
```

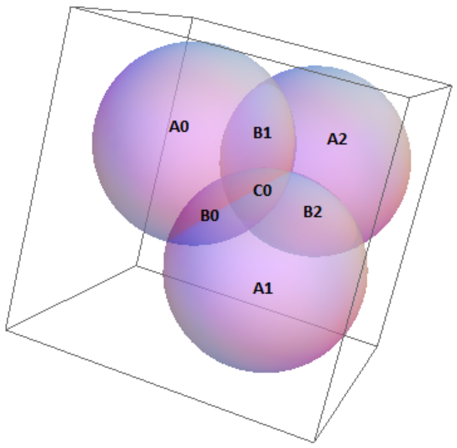


When such a graphics window is rendered, one topology to define is X = {DA, DB, DC}, and τ (same as above):

τ = {∅, {DA ∪ DB}, {DA ∩ DB}, {DA ∪ DC}, {DA ∩ DC}, {DB ∪ DC}, {DB ∩ DC}, {DA ∪ DB ∪ DC}, {DA ∩ DB ∩ DC}, {DA ∪ (DB ∩ DC)}, {DB ∪ (DA ∩ DC)}, {DC ∪ (DA ∩ DB)}, {DA ∩ (DB ∪ DC)}, {DB ∩ (DA ∪ DC)}, {DC ∩ (DA ∪ DB)}}.

A finer topology would be:

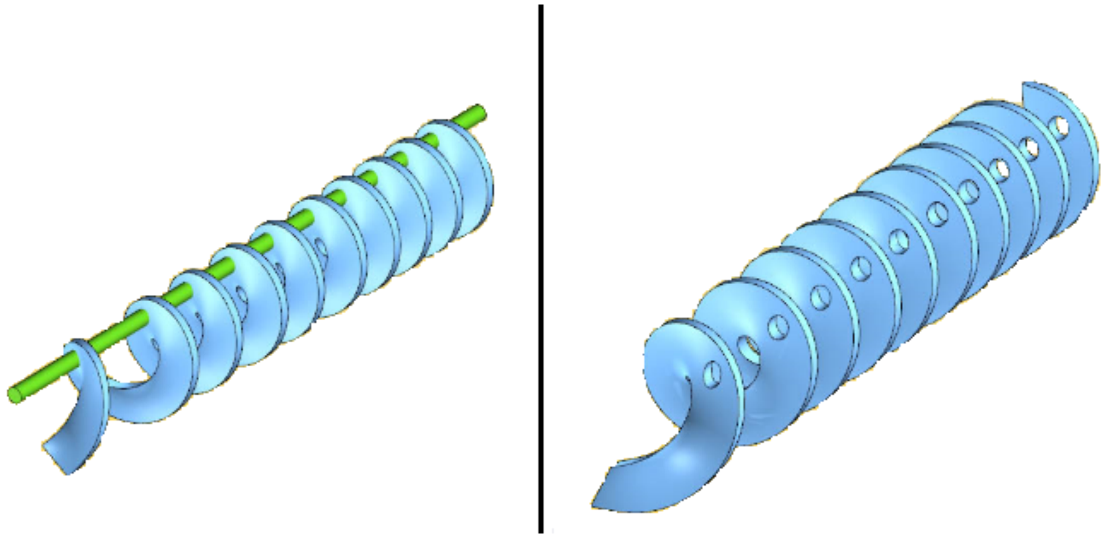
X = {A0, A1, A2, B0, B1, B2, C0}, and τ_2 = ∅ and the union of any x ∊ {1, 2,...7} choices of spaces as labeled below: {A0, A1, A2, B1, B2, B3, C0}. For example, in detail: τ_2 = τ ∪ {{A0}, {A1}, {A2}, {A0 ∪ A1}, {A0 ∪ A2}, {A1 ∪ A2}, {A0 ∪ A1 ∪ A2}, {A0 ∪ B3}, {A0 ∪ C0}, {A0 ∪ B1}, {A0 ∪ B2}, {A0 ∪ B1 ∪ B3}, {A0 ∪ B1 ∪ B3 ∪ B2}, {B1 ∪ B3}, {B2 ∪ B3}, {B1 ∪ B2}, {A1 ∪ B2}, {A1 ∪ C0}, {A1 ∪ B1}, {A1 ∪ B3}, {A0 ∪ B1 ∪ B3} .... etc. What we have added here, in comparison to τ, is elements containing sets of images that propogate as an idea from intersections or/and unions of the subtracted images of DA, DB or DC (think how now we have access to DA \ (DB ∪ DC) as an element of τ_2).



The advantage of treating graphics windows as the above topological spaces easies with object selection and manipulation.

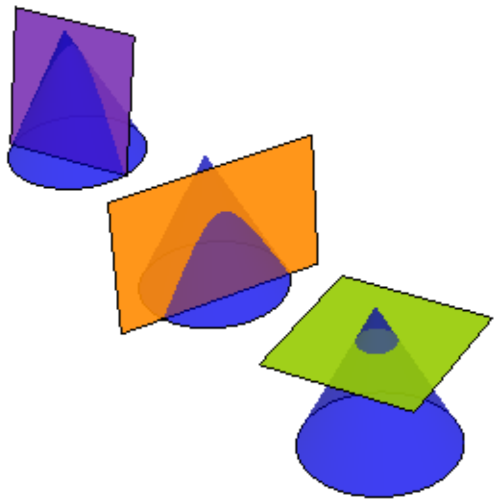
Here is an example of a subtraction of an object from another graphics object, where X = {spiral, rod}, and τ:

τ = {∅, spiral, rod, spiral ∩ rod, spiral ∪ rod, spiral \ (spiral ∩ rod), rod \ (spiral ∩ rod), (rod \ (spiral ∩ rod)) ∪ (spiral \ (spiral ∩ rod))} is the topology and the ‘subtraction’ operation is removal of elements from τ in such a way that we result with a new topological space X_2 = {spiral \ (spiral ∩ rod)}, and the trivial case τ_2 = {∅, spiral \ (spiral ∩ rod)}:

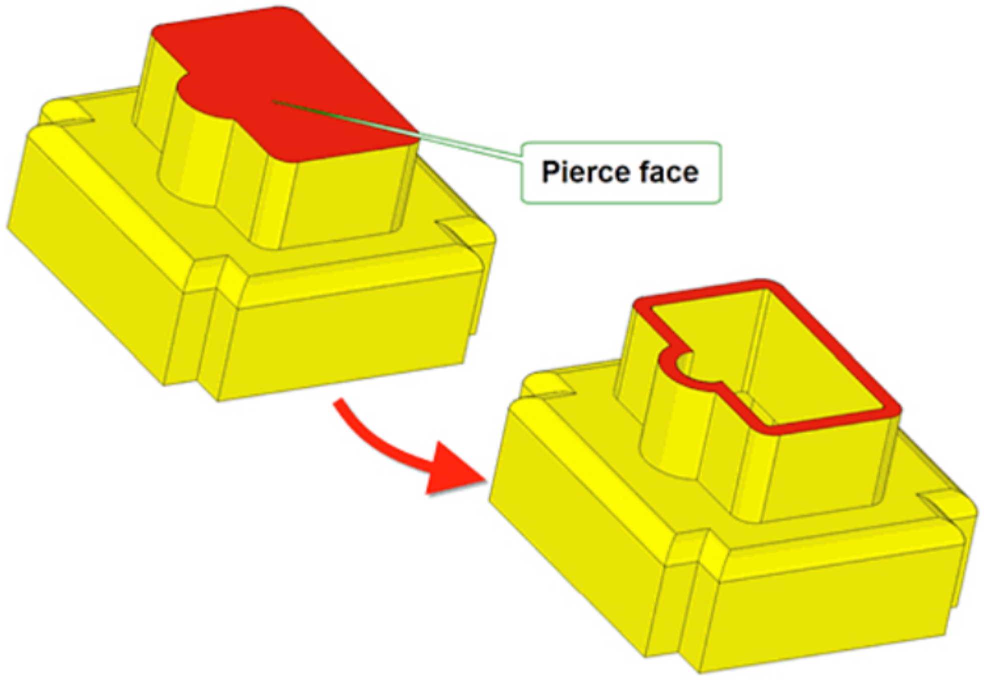


A better refinement of the above topology can take place if we want the boundary and the interior of the shape to be elements for individual manipulation (think of hollowing objects or partial texturing).

Another advantage of seeing images at this level of granularity is image slicing, to result in definition of new images and a new topology:



It may also be easier through this concept to define fillings and hollowings of complex objects:



Elements to aid in the generation of a topology we need

The refinement of the topology boils down to the functional operations we want to perform. What I propose that we provide to perform on graphics objects (note that some of the followings are not going to be feasible on the topological space selection alone):

- Unionize objects: in return, the result should give us one graphics object.

- Intersect objects: the result should give us the intersected object as one graphics object.

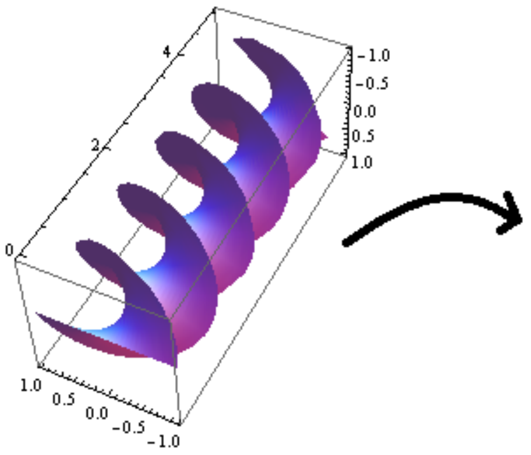
- Subtract objects: the result should give us the volumetrically bigger object with a hollow or "bite" of smaller or same size as the subtracted object.

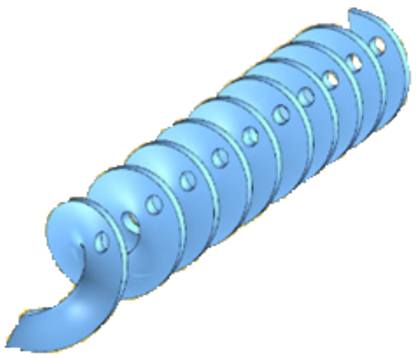
- Select objects: one should be able to access one part of a rendered object, such as an intersection of two objects, for operations such as texturing and coloring of the surface.

- Assemble objects through boundaries: shells, edges and verteces: eg. two sheets can be joined at an angle and share an edge to create a new graphics object. The assembling of objects should give us one graphics object and maybe it should initiate a redefinition of a topology. The later may be neccessary when assembled objects intersect with other objects and we would like to continue to provide the functionality on above bullets, for example, to subtract the intersected space.

- Hollow and fill objects: it would be very convenient to the user to draw images using an exact equation and later requesting to turn it into a volumetric shape which he later may continue to 'drill' it with other graphics objects to obtain a new image, well, here is what I am picturing:

```
ParametricPlot3D[{u Sin[t], u Cos[t], t / 3}, {t, 0, 15}, {u, -1, 1}]
```





Starting image   Desired result

Now all these components: object bodies, shells, edges and verteces should be entities to be included in our topological space. We are not in shape to generate the topology criteria yet, but I am going to need to identify some elements.

It may seem difficult to view an image and not think of it as an object in the euclidean space; after all, we are using samples of points to render them on the screen. So, to remove that thought, I will refer to the topology I want to create as the topology of the 'total' arbitrary unions of the set of elements, generated from a rendered object on the criteria of partitions (see labels on those 3 spheres above), plus, the whole set of tags on verteces, edges and shells, as well as all unions of bodies with the boundaries (be it verteces, edges and shells).

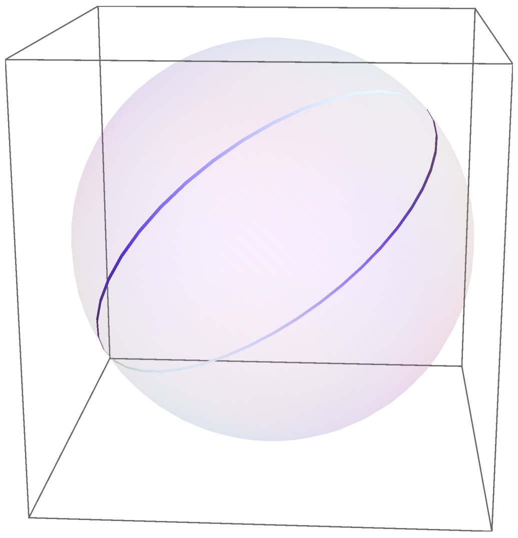
Identifying these elements with unique tags makes for a better mental picture (think of how a spherical object is identified in *Mathematica*: Sphere[]; in our case the string name will be dynamically defined for an arbitrary object). One should logically identify tags with a bag of points that render that part of the image but not vice versa: a rendered section of an image may not be a unique tag, instead it may be sharing its points among multiple tags. It is essential for the objects to be recognized and singled out as entities in a model, so I propose that we do use unique identification to represent an element of the topology. These tags should exist throughout the *Mathematica* notebook and be automatically generated (when possible, after the user has requested rendering of a set of objects), as well as be defined by the user.

The refinement of the sets of **body'**s topology should resemble some of the above τ and τ_2 tags. Now, the **edges** should identify with tags of curves on the boundary; the **shells** should identify with tags of the surfaces of an object; the **vertices** should identify with tags on the boundary points.

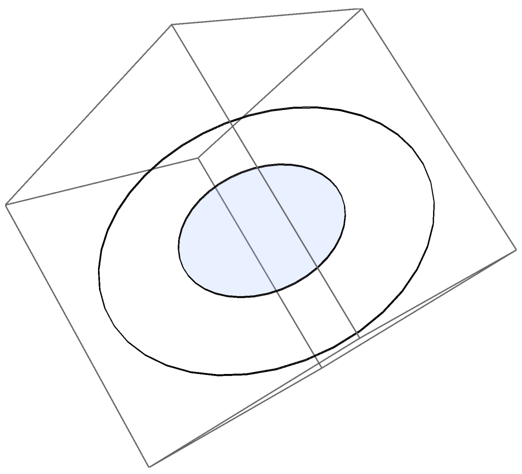
As we already suggested, identification of tags may be automated or user defined. In an automated event, the classification of tags can be a challenge. However, we are capable to define some criterion in cases where we are given a number of bodies, shells, edges and verteces. To define more body tags, we attempt to find the intersections of the given bodies by locating sets of points lying in the interior of a body. Sometimes, the intersection may be a shell, vertex or edge which should also be identified with a tag; if there is a subtractable image, the subtracted bodies, edges and verteces should also be identified with unique tags.
For user-requested assemblies, auto-defining tags is impossible. Therefore, providing the user with the option to define tags is ideal. In *Mathematica*, it is fairly easy to programmatically graph spaces, so defining tags should be allowed to be defined the same way one is able to render it; here is a thought:

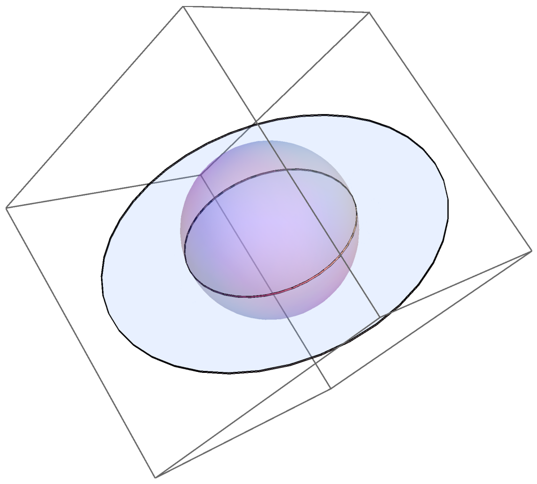
DefineTag[“Edge1Sphere1”, On[Sphere[{0,0,0},.1]], CapForm[None],Tube[{{0,0,0},{.001,.001,.001}},.1]]

Mental picture (the tag, which is an edge, is the band around the sphere):



Then, one may also define the same edge tag (literally, same name) on another object, and later, they can join the the two graphics objects into one, through some defined *Mathematica* function (internally would be a series of isometric transformations):


The band enclosing the light blue space is the edge


The halo can now be joined with the sphere

To summarize this section:
for what we want to achieve, as we observed, there is no standard formula on how to generate our topology. But, we can set some criteria:
- generate our topology after some objects are rendered, such as primitives or a set of equations;
- aim to generate a finite topology;
- and, be able to regenerate it after a user has requested actions that may be changing meaning of the rendering, for example, assembling of objects.

In addition, it is feasible to automate some generation on the criteria of object intersections as well as enable the user to define some of the grains in the topological space and, later, extend it to an actual topological space. A finer automated topology can be decided when looking further into this problem. For now, most importantly, the combination of both (user-defined and automated) should be expected.

Suggested functionality

This section lists some functionality that one should be able to execute on graphics objects. The following functionality are meant to draw a picture for the reader and encourage thoughts.

CreateGraphicsObject[objectname, recipe]: shall output the empty set or another object as created through the ‘recipe’ and named after the objectname tag. CreateObject shall be able to create a function: objectname[], the same way Sphere[] or Disk[] appears to Graphics[]. The ‘recipe’ should be recognized in many forms:

An equation:
```
Parametric3D[{Sin[u] Sin[v] + 0.05 Cos[20 v], Cos[u] Sin[v] + 0.05 Cos[20 u], Cos[v]},
 {u, -π, π}, {v, -π, π}, MaxRecursion → 4]
```

A set of graphics primitives:

```
{Cylinder[], Sphere[{0, 0, 2}], Line[{{-2, 0, 2}, {2, 0, 2}, {0, 0, 4}, {-2, 0, 2}}],
 Polygon[{{-3, -3, -2}, {-3, 3, -2}, {3, 3, -2}, {3, -3, -2}}],
 Opacity[.3], Cuboid[{-2, -2, -2}, {2, 2, -1}]}
```

A set of NURBS:

```
BSplineCurve[{{0, 0, 0}, {1, 1, 1}, {2, -1, 1}, {3, 0, 2}, {4, 1, 1}}],
 {0.1, 0.2, 0.5, 0.2, 0.1}]

Tube[BSplineCurve[{{0, 0, 0}, {1, 1, 1}, {2, -1, 1}, {3, 0, 2}, {4, 1, 1}}], 0.2]

{BSplineSurface[pts], Point /@ pts},
for this definition :  pts = Table[{i, j, (-1)^(i+j)}, {i, 5}, {j, 5}];
```

The resulting object can be a body, a boundary or both, which can be conveniently defined through *Mathematica* options. Another option can be provided to allow for the topology to be generated automatically. Upon object creation, tags and the object's topology is neccessary to define, if it can be, automatically. If we have a set of graphics primitives or a set of spline curves, this may be achieved by identifiying points of intersection between the primitives. Algorithms to automatically define such space divisions should be selected carefully.

Intersect[object1, object2] : shall output the empty set or another object, which is the 'natural' intersection between the two objects; options may be defined to include or exclude boundary (shells, verteces and edges) and body in the output image. For example, if the user defines an intersection on the bodies and boundary, if the objects contain shells and vertices, we expect an output of an object with a body and boundary. Further, new tags and topology should be generated for the new body and boundary. The user should be allowed to rename tags.

Slice[object, sheetEquation1, sheetEquation2...]: shall output the object itself or two or more objects (additionally, new tags and new topology), which are the results of object's slicing with the sheet (provided through the sheetEquationx).

Subtract[objectname, object1, object2]: shall output the empty set or another object, which is object1 with a hollow or a bite on the location of object2; options as mentioned above shall be provided: if the user defines a boundary to remain after subtraction, part of object2's boundary shell shall remain intact on object1.
So far, per functionality definition, one may execute: Subtract[object1, Intersect[object1, object2] to obtain a new type of object.

Union[objectname, object1, object2]: shall output the empty set (if object1 and object2 are the empty set) or another topology: the union of two objects. A new topology of the object shall be defined: tags for union of shells, verteces, edges and bodies. When redefining the topology, if the union of two objects results in a smaller space occupation than the sum of the two bodies, we could be dealing with an intersection, in which case redefinition of the topology shall take into account the intersection body, shell or edges and any union with other objects. [Note: if the two objects are disconnected, we will be resulting in a disconnected topology].

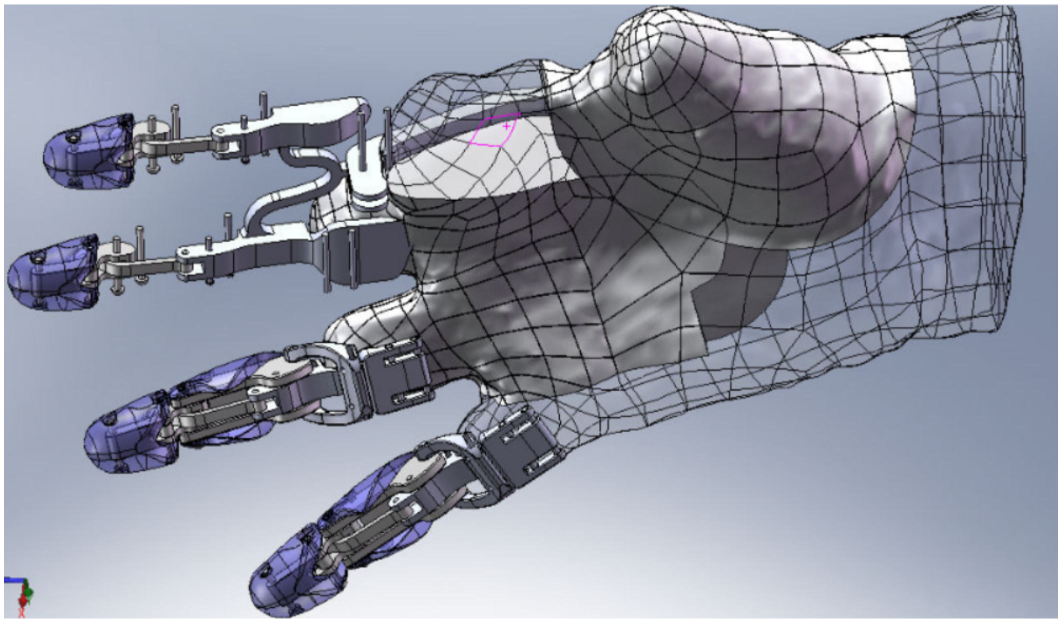
ShowTags[objectname]: shall output a list of tag names, which logically constitutes all subsets of the object identified with the objectname. For example, if we have two spheres with bodies and shells:

ShowTags[Union[“UnionTwoSpheres”, Sphere[], Sphere[{0,0,1}, Include->Bodies, Include->Shells]]] may give us: SphereBody_000, SphereBody_001, SphereBody_000_Or_SphereBody_001 (union of two sphere bodies), SphereBody_000_And_SphereBody_001 (intersection of two sphere bodies), SphereBody_000_Minus_SphereBody_001, SphereBody_001_Minus_SphereBody_000 (subtractions of sphere bodies), and the same for shells, then, the same for bodies and shells, such as Sphere-BodyAndShell_000. Again, the amount of the subsets relates directly to the refinement level of the topology. If one excutes: Graphics[SphereBodyAndShell_000_And_SphereBodyAndShell_001[]] one should obtain an image of the intersection of the bodies and the shell surrounding it.

RenameTags[newobjectname, objectname]: this shall allow the user to rename tag names for an existing tag, observed as above. The user can be a better judge on what these spaces mean to him.

DefineTag[objectname, On[object], recipe]: shall allow the user to define a tag name as well as an element of the topology on the object as specified through the recipe. An option, such as ‘type’, may identify the type of tag: body, shell, edge or vertex. If the option does no correspond with the recipe, an error should be generated. Identifying the elemental type of the topology: is it an edge, vertex or body, could be a challenge. We expect that the recipe provide a formula or set of points that exist on the object. Through options, one should define whether the recipe is meant for bod(y)(ies), edge(s), shell(s) or verte(x)(ces). A careful selection on how to define the inputform of the recipe can be important. One should have the capability to throw in the recipe input section an algorithm, which generates a table of, say, edges in a finite form (since we have chosen to only generate a finite topology). For clarification, one should be able to define an edge as sin(x) between 0 and π but one should be limitted to a finite set of edges, eg: set of edges = {edge[c], c between 0 and 15 with 5 partitions} where, edge[c] := {sin(c x), x between 0 and π}.

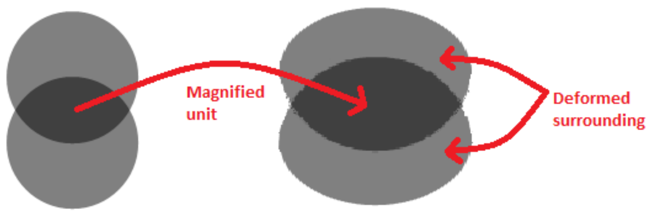
Assemble[object1, object2...., Isometry[On[object1], equation], Isometry[On[object2], ...],... At[shell1, shell2,...]]
Assemble[object1, object2...., Isometry[On[object1], equation], Isometry[On[object2], ...],... At[edge1, edge2,...]]
Assemble[object1, object2...., Isometry[object1, equation], Isometry[On[object2], ...],... At[vertex1, vertex2,...]]... (or other overloaded functions): shall enable the user to ‘glue’ objects on defined boundaries, such as shell, edge or vertex after a set of isometric transformation (as described by the equation), when possible. Two finite boundaries (I am referring to verteces, edges and shells) are equivalent if there exists a euclidian isometry between the two, such as, a reflection and translation operation. Such boundaries should be allowed to be glued. An isometric transformation on one of the edges, to glue it to the desired edge, should result on the isometric transformation of all points in the subsets that share a set with the edge: lifting a cup of coffee from the table and rotating it should not result in deforming of the cup. Therefore, verifying whether the user-input equation is an isometric transformation is very important. However, one can simplify this issue by providing the user with a set of functions they can choose from, such as Translate[], Rotate[] or Reflect[]. The assembling shall result in redefinition of tags and the topology (taking into account new unions of objects resulting from sharing of edges). Tags of glued boundaries should be merged to represent one tag or element, since they will become logically the same bag of points. Overall, assembling can play a major role in putting together complex designs:



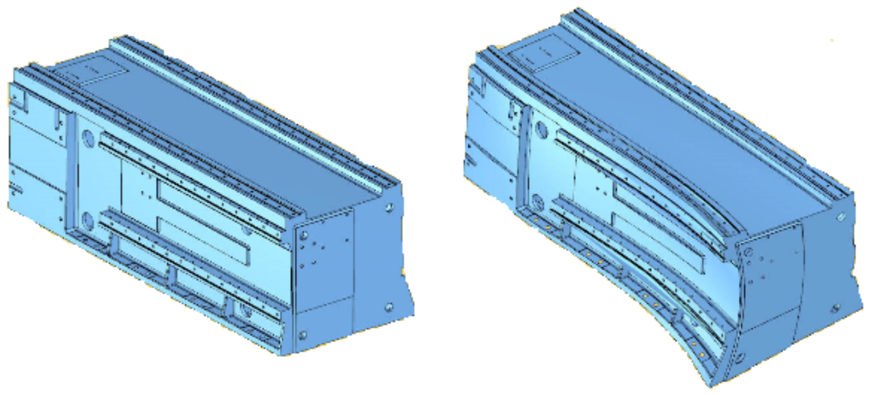
Fill[object, boundary]: should ‘fill’ the interior of the object enclosed by the boundary (see filled part of the hand above), or the ‘thickening’ of the object in the case the boundary is not connected. Options may define a direction of the thickening and the magnitute of the thickness for disconnected boundaries. Any other objects that lie in the interior of the boundary should seize existing. In ‘boolean terms’, for a connected boundary, the filling is an intersection between the object and a body that encloses the object. A filling should initiate a redefinition of the topology. To verify algorithmically whether the boundary is connected or not is a callenge. The next section will demonstrate some algorithms to a set of problems we will come accross.

Hollow[object, boundary]: should hollow the interior of the object enclosed by the boundary. The thickness of the hollowness or boundary should be defined through an option otherwise an automatic value will be used. If the boundary is disconnected the hollowing cannot take place, hence, an error message is appropriate.

Deform[object1, object2,.. On[Shell1,Shell2,...], typeOfDeformation]
Deform[object1, object2,.. On[Edge1,Edge2,...], typeOfDeformation]
Deform[object1, object2,... On[Vertex1,Vertex2,...], typeOfDeformation]: attaches a type of deformation or plasticity to the objects on a boundary. The Deform[] function shall allow a ‘natural’ deformation of the glued bodies and boundaries. The treatment of objects as elements of a topological space comes in handy now that we want to apply the deformation only on specific parts of the object. For example, when attempting to magnify an object which shares a boundary with another object (who was not signed up to be magnified as a whole), and you want to keep it that way, allowing a local deformation around the boundary the two objects share may enable one to create a new set of graphics manipulations. Here is a picture:



Also, here is another example of a type of manipulation that can now be achieved for more complex deformations (a quick note: Deform[] is a hypothetical function to demonstrate this idea; it is likely that more than one function will be needed to incorporate ‘Deform’):



LineModel[object]: should return an equation or a table of points that identify the model of the object, if it is a an edge.

SurfaceModel[object]: should return an equation or a table of points that identify the model of the object, if it is a shell.

SpaceModel[object]: should return an equation or a table of points that identify the model of the object, if it is a body.

SurfaceArea[object]: should return the surface area of the object.

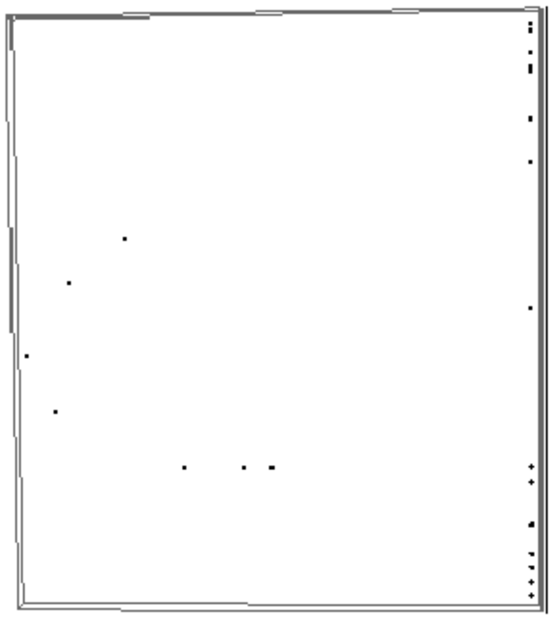
Volume[object]: should return the volume of the object, if it encloses a volume.

And to wrap it up, here is a an additional set of relevant self-explanatory functionality that we may want to accomplish:

Magnify[object, magnification], **Fold**[object, crease, angle], **Bend**[object, On[edge], angle], **Rotate**[object, axis, angle], **Translate**[object, affine equation]...etc.

All functions shall output appropriate error messages.

A note:
An existing functionality that can play an interesting role in object generation is the *Mathematica* Tube[] function. Tube[] enables the user to generate graphics through rotations around axes points:



Axis of rotation of the pot

Final product using Tube[] and BSpline[]

A set of problems in computational geometry

Some problems that we may encounter are complex problems in computational geometry.

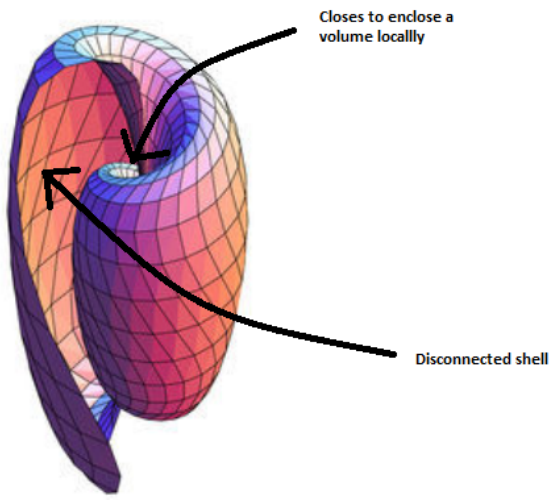
Intersections of volumetric objects

Algorithms to generate topologies on a granular criteria (as suggested so far) demand for the selecting of a set of known objects, the assessment whether the objects have connected or disconnected boundaries, and the finding of intersection of the selected volumetric objects (for the purpose to label that bag of points as a grain of the topological space to be constructed). Finding the actual volume requires just as much thought too. The computation of the volume of the unions or intersections of high-dimensional geometric objects (such as 3D objects) is an #P-hard problem. Karl Bringmann and Tobias Friedrich[1] give a very nice polynomial-time approximation scheme to solve such problems, where one can (1) test whether a given point lies inside the object (classify it as a point of the intersection space if lies inside another object as well), (2) sample a point uniformly, and (3) calculate the volume of the object in polynomial time; and it suffices to be able to answer all three questions approximately. For a complete understanding, check out the link in the references section.

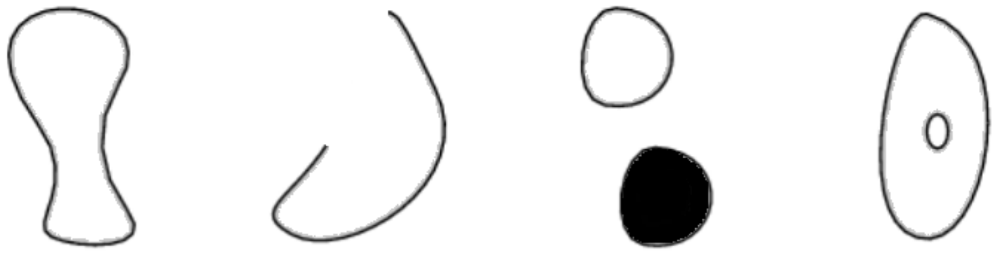
Gluing objects

Gluing objects at a boundary requires solving the problem of identification of isometric transformations between two boundaries of two objects: if the boundaries are isometrically equivalent then the objects may be glued. Assume we enable the user to provide us with two boundaries to be glued, and a given transformation that does that. How does one verify that the transformation really is an isometry and the boundaries are equivalent? As a quick thought, one can verify that two smooth curves are equivalent if their curvature and torsion are equivalent, by the Fundamental Theorem of Space Curves[2]. However, how should one go about identify smooth curves in the first place? Or how should one identify an isometric transformation on shells? Should we even allow the user to provide an equation that moves the objects, or should we restrict them to use *Mathematica* functionality, such as Transform[] and Rotate[], which are sufficient to achieve the objective of aligning two boundaries that we want to glue? Such questions may guide us to pick through the correct functionality to provide.

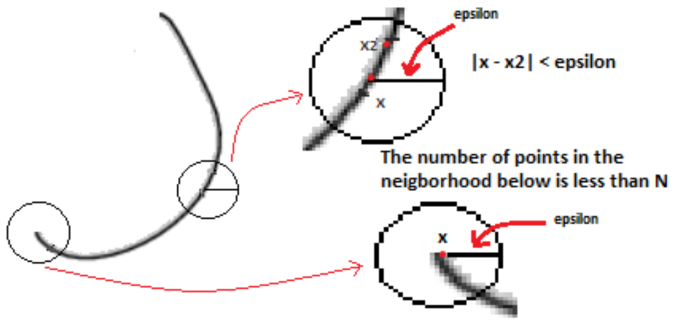
Identifying connected vs disconnected boundaries
Assessing whether a given object has a connected or disconnected boundary (for example, a shell may enclose a volume or may just be a bend sheet) can be difficult to determine, if not impossible. And what about objects that enclose a volume for some boundary and otherwise for a disconnected boundary (eg. a sea shell like object)?



Some problems may not be deterministic and thus not solvable algorithmically. In fact, I have not been able to find a research paper on this problem yet. However, here is a theoretical thought for a more 'regular' volume. Skip to the last paragraph if not interested, since my argument is not supported with a proof. One may project the boundary of an object on an arbitrary number of unique planes that cross cut the object arbitrarily (using randomized methods). Resolving where they cut, is a problem of intersection of two objects; see above. The result of a cross-section would be a curve or a set of curves (or other combinations of: point, for tangent planes and surfaces, for coplanar cutting sheets; for now, we will skip these cases).

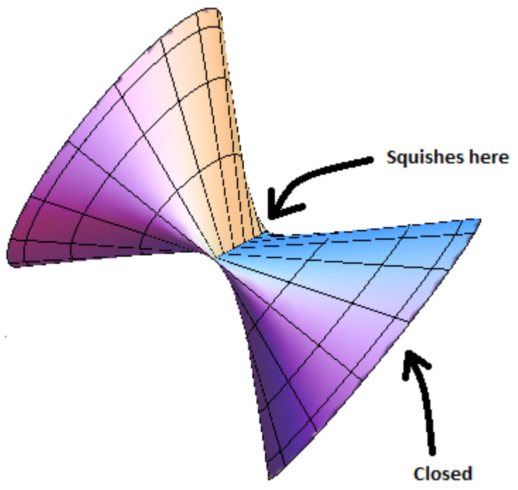


Now, the problem has been simplified to determining whether there exists a disconnected curve on the plane. Using randomized methods for intersections expressed as equations (otherwise, we have a set of points), one may generate a random sample of points on the curves (the sample can be as refined as the number iterations we choose to run), then for a fixed ϵ (carefully chosen), a fixed N (depending on the number of iterations and ϵ), and an arbitrary point x_0, one should expect an ϵ-neigborhood with n number of points greater than N. If we find a neigborhood at an aribtrary point with m < N number of points, then we have found a disconnection.


epsilon
x2
x
|x - x2| < epsilon
The number of points in the neigborhood below is less than N
epsilon
x

Note: if the intersection is not curves but surfaces, for choices of ϵ and N as above, the surfaces would pass the test of a closed curve; and this is what we want.

Now, finding whether there exists an open curve does not solve the problem yet: one may have an open curve, and still have an enclosed space by a boundary, such as volumes with ‘squishes’ at a dimension (or points for tangent planes):



The middle of this venturi-like shape closes,
to not allow a flow between the volumes

Then, once we have found a disconnected curve on a given plane P, we may generate a new set of ordered random planes parallel with P which cross-cut the volume on a series of locations. Now, we proceed to run the same algorithm for finding a disconnection on these new curves, as above. If we find a series of ordered disconnections, that means we may be dealing with a disconnected shell and so, no enclosed volume exists between the existing disconnections. Because my method is crude, incomplete and not really proved for these cases, I cannot verify, at least at this time, that there will not be any false positives in this decision. In fact, we still have not decided how we will treat surfaces that enclose volumes at one part and are disconnected at others. In any case, this demonstrates the amount of thought and computation that is needed to make such decisions.

To not continue getting into too much detail, a range of topics are open for discussion when attempting to solve the problem of division of objects into building blocks of a topological space and more: optimization, memory storage and representation, calculus on spaces, neccessary computational power, multi-threading etc... Pinpointing some of these problems, can guide us through a good selection of functionality to provide to the user.

Conclusion

I hope that I have built my case that to provide the user with the ability to select and manipulate graphics objects, treating them as elements of a topological space is the way to go. While software engineering such an idea can be a lot of work, overall, the amount of functionality that can become accessible exceeds imagination. Certainly, problems that derive while attempting to put this together can be complex, but a careful design can permit one to determine a better criteria for selection of front end functionality.

References

[1] Karl Bringmann and Tobias Friedrich. Approximating the volume of unions and intersections of high volumes. http://www.mpi-inf.mpg.de/~tfried/paper/CGTA1.pdf
[2] Chuu-Lian Terng. Parametrized curves. Lecture notes on Curves and Surfaces by Chuu-Lian Terng. Department of Mathematics, University of California at Irvine. http://math.uci.edu/~cterng/162A_Lecture_Notes.pdf.
[3] Russian 3D-kernel RGK: Functionality, Advantages and Integration. 24 May 2013. isicad.net/articles.php?article_num=16135
[4] Parasolid: Desktop Visualization & Modeling and Toolkit Products. Technical Overview. Introduction to Booleans. http://www.techsoft3d.com/developers/getting-started/parasolid/introduction-to-booleans/
[5] Josh Mings. [SolidWorks image of a prosthetic hand]. Retrieved 11 November 2013. SolidSmack.com