

Interactive CAD Software for the Design of 2-manifold Free-form Surfaces (NOME)

Gauthier Dieppedalle



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-48

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-48.html>

May 10, 2018

Copyright © 2018, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I would first want to say thank you to Professor Carlo H. Séquin for the time he has spent advising me during the 5th year M.S. program and for all of the discussions that we had in developing NOME. I would also like to say thank you to Professor Björn Hartmann for giving me valuable feedback on this report. I am also very grateful for the help that Toby Chen and Beren Oguz have provided in writing code for NOME. I would like to thank Professor Séquin's URAP students for their feedback while using early versions of the NOME programming environment.

**Interactive CAD Software for the Design of 2-manifold Free-form Surfaces
(NOME)**

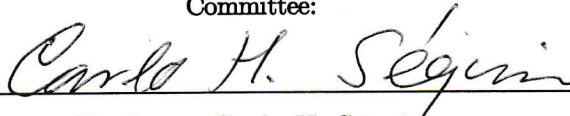
by Gauthier Dieppedalle

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:


Committee:



Professor Carlo H. Séquin
Research Advisor

05/02/2018

* * * * *



Professor Björn Hartmann
Second Reader

05/02/2018

Interactive CAD Software for the Design of 2-manifold Free-form Surfaces (NOME)

Gauthier Dieppedalle
EECS Computer Sciences, University of California, Berkeley
E-mail: gdieppedalle@berkeley.edu

In order to build 2-manifold free-form surfaces of potentially high complexity but with much inherent regularity, we have developed a CAD (Computer-Aided Design) tool called NOME (Non-Orientable Manifold Editor). This tool makes it easier and more precise to build 2-manifold sculptures procedurally through a text editor and interactively through a GUI (Graphical User Interface). We have tested the software by reproducing sculptures by Eva Hild and by Charles O. Perry, which can then be 3D printed.

1. Introduction

Over the past twenty years, Professor Carlo Séquin and his students have been using Berkeley SLIDE (Scene Language for Interactive Dynamic Environments) to create abstract geometrical sculptures [10]. The SLIDE software offers numerous powerful constructs such as sweep generators, several different subdivision techniques, and rendering options. However, SLIDE has some shortcomings. It cannot subdivide and create offsets over non-orientable two-manifolds such as Möbius bands and Klein Bottles. The SLIDE code has also been poorly maintained over the years, as dozens of students have contributed to it without taking into consideration organization and scalability. We have observed that creating smooth, free-form two-manifold, particularly, if they are single sided, is still extremely difficult and tedious with the CAD tools currently on the market. Professor Séquin and Andy Wang, a previous 5th year Masters student at Berkeley, had started the construction of new software that would strike a good balance between procedural shape generation and interactive graphical editing capabilities [12]. Pushing forward in the same direction, the current software is called NOME (Non-Orientable Manifold Editor).

2. NOME Compared to Existing CAD Tools on the Market

We have started the NOME project to create an alternative to existing CAD tools, such as Maya and Blender, which rely too much on a point-and-click Graphical User Interface. This makes it difficult to construct precise geometrical free-form shapes with much inherent regularity. Forcing the designers to generate meshes entirely by clicking and dragging on the screen using the 3D GUI, often produces imprecise sculptures due to the inaccurate nature of physically selecting and moving vertices mapped from

the 2D plane of the screen to a 3D space. In the past several years, Blender has added the support to modify models programmatically, using Python to automate actions from the Blender GUI to speed up the workflow [14]. The scripting add-on to Blender is very powerful to create a first original design or to animate objects in 3D space. However, in Blender, it is not possible to read in a mesh via a script then edit via the graphical interface, and then save the result again as a script. Therefore, Python must be seen as a separate program from Blender that allows users to script their workflow. But it is not a descriptive language that can be used to define a hierarchical scene programmatically. The hierarchy in Blender files (.blend) cannot be accessed directly via a text editor, since the file is in byte format and thus not human readable.

When developing NOME, we have focused on building a shape description language that allows designers to start their designs via code, then fine-tune it and enhance it via a graphical interface, and subsequently capture these edits in a piece of code that can readily be integrated with the original description. In this form, the design process alternates between procedural definitions and graphical edits. So far, NOME has primarily been used to help artists generate precise mathematically defined sculpture that can then be 3D printed [16]. The NOME GUI running on Windows can be seen in *Figure 1* [9]. The displayed sample file shows twelve 4-stub Dyck funnels linked into a symmetrical cluster with the symmetry of the oriented cube (created by Carlo Séquin).

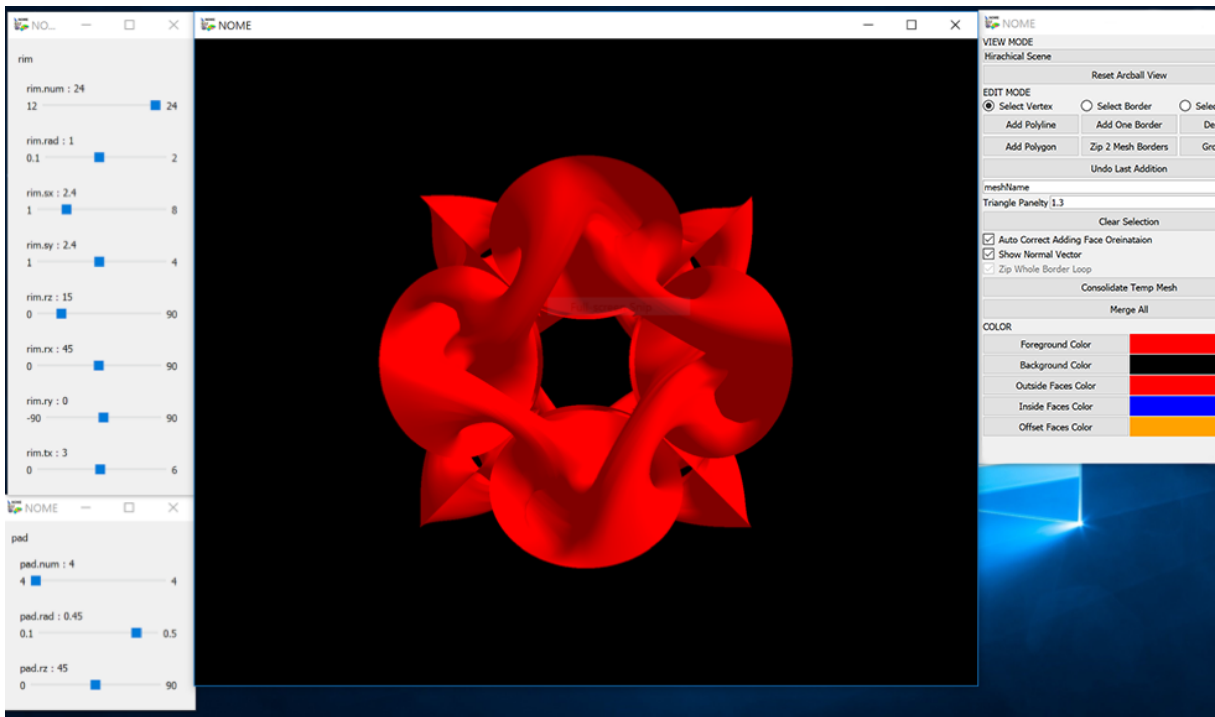


Figure 1: Graphical User Interface of NOME running on Windows.

3. Formal Language Definition

NOME follows the general scheme of a graphical scene description. It is a single assignment language requiring unique identifiers in each of its hierarchical contexts. Points, edges, and faces can be assembled into meshes, and these, in turn, can be assembled into groups that form a hierarchy. The following constructs are available in NOME:

Point: `point id (point_triple) endpoint`

Polyline: `polyline id (point_idlist) [closed] endpolyline`

Face: `face id (point_idlist) [surface surface_id] endface`

Object: `object id (face_or_polyline_idlist) endobject`

Mesh: `mesh id`

`face faceId1 (point_idlist1) [surface surface_id1] endface`

`...`

`face faceIdN (point_idlistN) [surface surface_idN] endface`

`endmesh`

Group: `group id`

`instance id1 object_id1 [surface surface_id] endinstance`

`...`

`instance idN object_idN [surface surface_id] endinstance`

`endgroup`

Circle: `circle id (n ro) endcircle`

Funnel: `funnel id (n ro ratio h) endfunnel`

Tunnel: `tunnel id (n ro ratio h) endtunnel`

Bézier Curve: `beziercurve id (point_idlist) slices numSlices endbeziercurve`

B-Spline: `bspline{order} id (point_idlist) [closed] slices numSlices endbspline{order}`

Instance: `instance id mesh_id [transformations] endinstance`

Surface: `surface id (color_triple) endsurface`

Background: `background surface surface_id endbackground`

Foreground: `foreground surface surface_id endforeground`

Front Faces: `frontfaces surface surface_id endfrontfaces`

Back Faces: `backfaces surface surface_id endbackfaces`

Rim Faces: `rimfaces surface surface_id endrimfaces`

```

Bank: bank bankID

    set setID1 value1 start1 end1 stepSize1
    ...
    set setIDN valueN startN endN stepSizeN
endbank

```

```

Delete: delete

    face faceId1 endface
    ...
    face faceIdN endface
enddelete

```

```

Subdivision: subdivision id

    type subdiv_type
    max subdivision level
endsubdivision

```

```

Offset: offset name

    type typeOffset
    instance instanceName
    max maxValue step stepValue
endoffset

```

Within a mesh, variable names must be unique. Sets within a bank and instances within a group must also have unique names. Most designers will spend a considerable amount of time modifying a .NOM file. It is convenient to be able to quickly disable some small or large portions of the code: # turns remainder of a line into a comment, (* and *) bracket a larger section of code spanning multiple lines.

3.1 Workflow of NOME

A NOME input file is analyzed in three phases: a lexer, which reads the program and creates the structure of the input file, a parser that structures these tokens, and lastly an evaluator that interprets the meaning of the program. Flex is used to create the lexer, and Bison is used for the compiler (these are newer versions of the tools called Lex and Yacc that were developed originally for UNIX) [3]. The workflow of NOME is shown in *Figure 2*. The user first opens a .NOM file. A .NOM file contains a hierarchical NOME description. Once the file has been opened, the lexer tokenizes the file into a set of

recognized tokens. Since NOME is a single-assignment language, the parser keeps track of variables that have been defined in the file, and duplicate names within the same context would trigger an error. These tokens are parsed using regular expressions. The parser organizes these tokens into sets that build up the scene graph. The evaluator takes these sets and constructs the meshes that define the geometry of the scene. These meshes can be modified and enhanced interactively through the graphical user interface. Finally, the user can subdivide and offset the mesh and generate a .STL file than can be sent to a 3D printer.

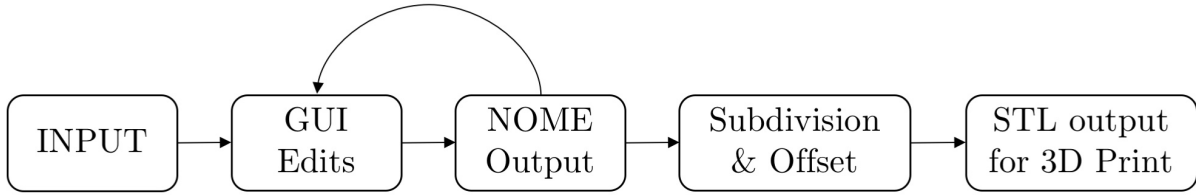


Figure 2: General Workflow of NOME.

3.2 Formal Naming for Variables and Reserved Names

Variable names must begin with an uppercase or lowercase letter of the alphabet, or with an underscore. After the first character, the variable name can contain either letters or numbers. The NOME language differentiates between lower case and uppercase characters. Reserved keywords cannot be used as variables names, such as `expr`, `bspline`, `endbspline`, `closed`, `slices`, `beziercurve`, `endbeziercurve`, `offset`, `endoffset`, `min`, `max`, `step`, `surface`, `endsurface`, `point`, `color`, `endpoint`, `bank`, `endbank`, `set`, `face`, `endface`, `object`, `endobject`, `mesh`, `endmesh`, `tunnel`, `endtunnel`, `funnel`, `endfunnel`, `polyline`, `endpolyline`, `circle`, `endcircle`, `instance`, `endinstance`, `scale`, `translate`, `rotate`, `reverse`, `foreground`, `endforeground`, `background`, `endbackground`, `insidefaces`, `endinsidefaces`, `outsidefaces`, `endoutsidefaces`, `offsetfaces`, `endoffsetfaces`, `delete`, `enddelete`, `group`, `endgroup`, `subdivision`, `endsubdivision`, `subdivisions`, `type`, `{`, `}`, `(`, and `)`. Throughout the compiler pipeline, we have added multiple checks to ensure that the naming of variables is consistent throughout the file.

3.3 Geometrical Elements and Data Structures

We have defined three basic constructs generators in NOME: `point`, `face`, and `polyline`. All of them are stored using the two-winged edge data structure [1]. Each construct in NOME must have a name as a

string and a generated index is automatically created in order to be able to efficiently compare if two construct references the same object.

Point: A point statement is used to define a vertex in 3D space, which can then be referenced by its id.

The *point_triple* defines the 3D position of the point via its x, y, and z coordinates. The individual values in such a list must be separated by white spaces, which can be spaces, tabs, new lines, or return characters.

Polyline: A polyline statement generates a 3D chain of piecewise linear segments. The *point_idlist* is a list of points. The list of point ids must be of length at least 2 (which would create a single edge) and they must be separated by whitespace.

Face: A face statement defines a contour which may be non-planar and non-convex, that is panned by a “membrane” composed of triangles. The front side or outside of the face is defined as the direction from which it is seen in a counterclockwise manner when seen from the “outside”, i.e. the direction against the face normal. The *point_idlist* is a list of points *ids*. Optionally, the *surface_id* allows to assign predefined surface properties to this face.

3.4 Hierarchical Constructs

We have created three hierarchical constructs in NOME to group the basic constructs together: object, mesh, and group.

object: An object statement is used to define a collection of faces and polylines, which can then be referenced by its id. These elements need not be connected. The *face_or_polylineidlist* is a list of face or polyline names.

mesh: A mesh statement also creates a collection of faces. Faces in a mesh can then be referred in the rest of the program via a hierarchical name: *id.faceId*, where *id* is the id of the mesh and *faceId* is the id of the face in the mesh.

group: A group is a collection of instances of primitive objects or other groups. Groups are the most general construct to introduce hierarchy into the shape description. All constructs so far have been procedural definitions. No geometry is added into the scene until an instance of one of these definitions is called.

3.5 Geometry Generators

The NOME language also supports various geometry generators that makes modeling geometrical sculptures simpler. Currently implemented are `circle`, `tunnel`, `funnel`, `bspline`, and `beziercurve`. A generated tunnel, funnel, and circle are shown in *Figure 3*. Future work could add spheres and tori.

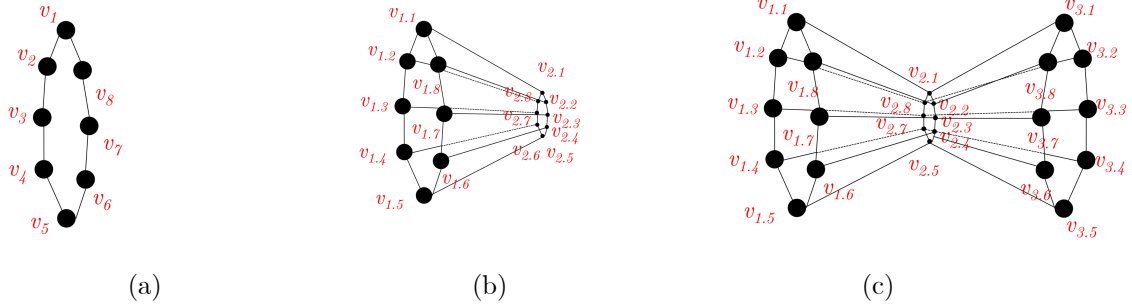


Figure 3: Examples of generated in geometry: (a) circle (b) funnel, and (c) tunnel.

circle: A circle statement generates a set of edges and vertices forming a regular n -gon with a given a radius and with n sides.

funnel: This statement generates a mesh in the shape of a truncated pyramid. n represents the number of vertices along the rim of the funnel. ro represents the radius of that rim. $ratio$ represents the size of the secondary rim compared to the original rim. h is the height of the funnel from the first rim to the second rim.

tunnel: This statement generates a “cylindrical” surface that can be modeled by the inner part of a torus. It is defined by 3 circles. n represents the number of vertices along each circle. ro represents the radius of the central circle. $ratio$ represents the size of the two outer rims compared to the central circle. h is the height of the tunnel from the middle circle to the outside rims.

beziercurve: This defines a Bézier curve that interpolates the two end points and approximates the rest of the points from a list of control points. The `point_idlist` defines a list of control points. The `numSlices` parameter defines the number of slices by which the Bezier curve is sampled.

NOME builds the Bezier curves based on the De Casteljaeu’s algorithm. According to the algorithm, a Bezier curve with $n+1$ control points (c_0, c_1, \dots, c_n) can be evaluated at a point using the following function:

$$B(t) = \sum_{i=0}^n c_i b_{i,n}(t)$$

where $t \in [0, n + (\{order\} - 1)]$ and b is the Bernstein basis polynomial defined as:

$$b_{i,n}(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

Since NOME only supports uniform Bezier curves, if we have *numSlices* segments for a given Bezier curve then we need to create a for loop that calls $B(t)$ with t incrementing by $\frac{1}{numSlices}$ at each step.

B-spline: The statement `bspline{order}` defines a B-spline of degree $\{order\}-1$ that approximates a list of sequence of $n+1$ points. The *point_idlist* defines a list of control points. The closed parameter defines whether the curve is closed. This automatically repeat the first $\{order\}-1$ control points at the end of the sequence. NOME uses the De Boor's algorithm to compute the points on the b-spline curve. Given a degree $\{order\}-1$ with $n+1$ control points (c_0, c_1, \dots, c_n) the points of the b-spline can be obtained using the following equation:

$$B(t) = \sum_{i=0}^n c_i B_{i,n}(t)$$

where $t \in [0, 1]$ and $B_{i,n}(t)$ is defined as the basis defined as following for $n = 0$:

$$B_{i,0}(t) = \begin{cases} 1, & t_i \leq t < t_{i+1} \\ 0, & otherwise \end{cases}$$

For $n \neq 0$:

$$B_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+k-1}} B_{i+1,k-1}(t)$$

3.6 Instances

All constructs so far have been procedural definitions. No geometry is added into the scene until an instance of one of these definitions is called.

instance: `instance id mesh_id [transformations] endinstance`

An instance creates an instance of geometry. Each instance that is created outside of a group at the top level of the hierarchical description, is rendered in the scene. The *mesh_id* is the name of the primitive object or group that will be instantiated. The user can optionally translate, rotate, and/or mirror that instance. Multiple transformations can be used together in any arbitrary order. Color transformations can also be specified by giving a *surface_id*.

Rotations are specified by: `rotate (axis_triple) (angle_float)`. The *axis_triple* defines the arbitrary axis of rotation $u=(u_x, u_y, u_z)$ through the origin. The *angle_float* representing the arbitrary angle of rotation is given in degrees.

Non-uniform scaling can be obtained with: `scale (scale_triple)`. *scale_triple* is the scale vector $s=(s_x, s_y, s_z)$ representing by how much each of the three coordinates must be scaled. The operation performed to scale a point $p=(p_x, p_y, p_z)$ is $(p_x \cdot s_x, p_y \cdot s_y, p_z \cdot s_z)$.

Translations are applied by using: `translate (translate_triple)`. The *translate_triple* specifies the three components of a translation vector $t=(t_x, t_y, t_z)$. This operation transforms all points $p=(p_x, p_y, p_z)$ as $(p_x + t_x, p_y + t_y, p_z + t_z)$.

To reverse the order of the vertices in a face the user can use the reverse transformation. The normal vector calculated for the face will then point in the opposite direction.

For debugging purposes and to understand the geometry rendered in the scene, the user can set colors to every element in the scene. Color transformations are defined by: `surface surface_id`. The *surface_id* is the name of the surface to be applied on the instance. The surface construct defines a color as following:

```
surface id ( color_triple ) endsurface
```

The *color_triple* defines the RGB color of the surface. The values of each of the three parameters must lie between 0 and 1.

A user may apply multiple colors to the same object in the scene and in that case the closest to the leaves of the tree color applied on the object will take precedence. Therefore, if an instance references to a vertex from another instance the color of the other instance will color the vertex. From a hierarchical standpoint, the color that is in the lowest node in the hierarchical tree representing the NOME file will color the object in the scene.

Multiple transformations are composed left to right in world coordinates.

3.7 Display Colors

The background of the window containing the geometry is set by default to black but can be changed by using the following statement:

```
background surface surface_id endbackground
```

Geometry that does not get assigned an explicit color in the .NOM file will be rendered with some default display colors which can be set as follows:

```
foreground surface surface_id endforeground
```


Once an offset has been applied on a geometry, the color of the front, back, and rim faces can be changed using the following statements:

```
backfaces surface surface_id endbackfaces  
frontfaces surface surface_id endfrontfaces  
rimfaces surface surface_id endrimfaces
```

These colors can also be set from the control panel in the GUI. Once the mesh has been merged and an offset is applied, the inside, outside, and offset faces corresponds to the colors of the faces.

4. Hierarchy stored as a Graph

4.1 Nodes of the Directed Scene Graph

The NOME hierarchy corresponds to a directed scene graph in order to be able to instantiate useful constructs multiple times without having to copy the entire data into a new node. When an instance references a vertex in the scene, that vertex keeps a pointer to the original, to ensure that any modification is propagated across the hierarchy in case a parameter relating to that vertex is changed.

The root node of the tree is always a Session object, representing the file that has been opened and parsed. This root node contains pointers to the different constructs and to the parameters available in the scene. In *Figure 7*, RENDER WORLD is the root node. A Session also contains a list of instances that are at the top level of the scene. An instance can reference either a mesh or a group. It contains a pointer to the original mesh or group but also makes a copy of all of the vertices, edges, and faces in order to ensure that if two instances reference the same mesh, they will be independent from each other. They still maintain a reference to the original mesh or group, in case the software needs to reference to the original definition. Similarly, a group also keeps a pointer to its list of instances. These instances may then reference either meshes or objects. Meshes or objects can contain faces, vertices, or edges. Polylines, funnels, and tunnels are stored as meshes.

Every element (point, face, mesh, group...) has a unique hierarchical name so that it can be referenced unambiguously. Any construct can then be reused to create the scene graph.

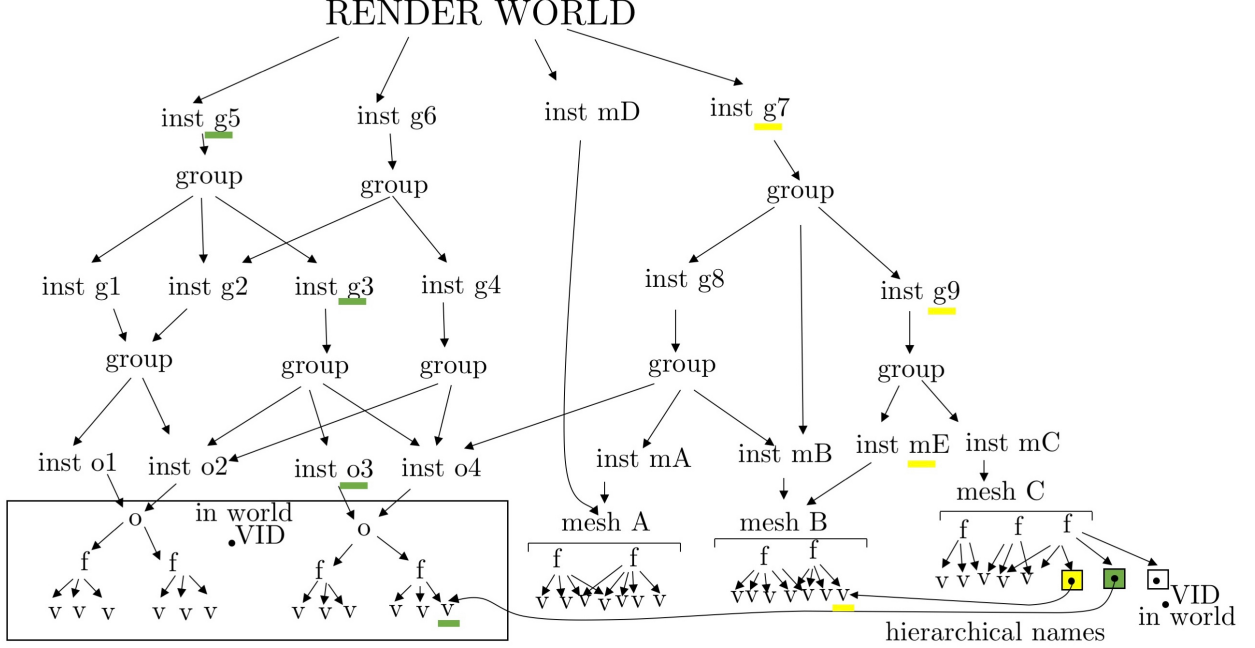


Figure 4: Hierarchical graph of the geometrical constructs of a NOME file.

4.2 Referencing to Vertices, Edges, and Faces

If a user needs to reference a node in the tree from another node, he/she needs to provide the entire path to that node either from the root of the tree or from the current node. Thus, the hierarchical name of a node is a sequence of node ids separated by dots “.” following the path through the scene graph. For example, in *Figure 4*, if the face from Mesh C references to a vertex in inst o3, the path must be .g5.g3.o3.v3. The initial dot is needed, since the path starts at the root. If inst g9 wants to reference to a vertex from mesh B, then the user can use mE.B.v3 without using the initial . character, since the path starts at the current node of the tree and not at the root.

4.3 Numerical Parameters and Sliders

After having defined the proper topology of a scene via the scene graph, a designer after would like to fine-tune the geometry by changing some parameters interactively while observing the results on the graphical display (example: make a ribbon wider, or a tunnel larger). NOME provides sliders that allows the user to change any numerical value in the NOME file through a slider in the GUI. Multiple sliders can be combined in a bank:

bank *bankID*

set *setID1 value1 start1 end1 stepSize1*

```

...
set setIDN valueN startN endN stepSizeN
endbank

```

Each *setID* must be a string representing the name of the variable to be changed. *value* represents the initial value of the slider. *start* represents the minimum value of the slider. *end* represents the maximum value of the slider. *stepSize* represents the size of the incremental steps. Each slider contains a pointer to a double value. In order for an object to reference a value from a slider, the user uses the following syntax *{expr \$bankID.setID}*. The user can then use this value to change the z-value of a point:

```
point p1 (0 0 {expr $lefttunP.n}) endpoint
```

That point will maintain a pointer to the double value in the set.

A bank is displayed in a window containing slider for every set. A user may use a set or a bank anywhere where a number is required. In order to add more flexibility every field requiring a numerical value can take *{expr...}* expression, which may contain complex mathematical expressions. These expressions are parsed using a lexer and parser developed in Flex and Bison. For example, the user may map two sliders to the position of an object such as in the following point definition:

```
point p1 (0 0 {expr sqrt($lefttunP.n * cos($lefttunP.r + 5 * sin(0)))}) endpoint
```

In order to parse these expressions, we have created a separate parser in Lex and Yacc that returns the calculated values. In our lexer, a bank/set combination must have the following regular expression syntax $\backslash\$[a-zA-Z][a-zA-Z0-9]^*.[a-zA-Z][a-zA-Z0-9]^*$, in which a bank or a set must start with any lowercase or uppercase letter and then may contain numbers in the name. The following mathematical operations are currently supported in NOME: *sqrt*, *cos*, *sin*, *tan*, *sec*, *cot*, *csc*, *arccos*, *arcsin*, *arctan*, *arcsec*, *arccot*, *arccsc*, *log*, *e*, *ln*, *+*, *-*, ***, */*, *^*, *(*, *)*. They may include integers or floating-point numbers

5. Graphical Editing

When opening a file in NOME, the user can visualize it and edit it via the graphical interface. These changes can then later be saved back into the original NOME file in a convenient hierarchical manner.

5.1 Editing via the User Interface

After having read in the initial geometry described in the NOME file and displayed on the screen, the user can edit and fine tune the geometry while visually observing the results. First, sliders can be adjusted to obtain overall best dimensions and proportions. In addition, topological changes can be made. The user

may delete faces or add new ones. By clicking on vertices in the display, a user may select a sequence of points, which can then be turned into a polyline or into a face contour.

5.2 Selecting Borders and Zippering

Often an extrusion or bridge between lengthy contours needs to be constructed. To make adding faces in a well-structured manner less tedious and more efficient, a high-level “zippering” command has been implemented. It is possible to zipper together two closed or open borders. The user first needs to click on the “Select border” radio button, then needs to select the border by selecting one of the vertices on that border. The code works by iterating through all of the edges connected to that vertex and recursively returns a list of all the paths that connects back to the initial vertex. When all of the closed borders have been returned, NOME returns the path that has the lowest number of vertices, i.e. the shortest path. By clicking on the “Add one border” button, the user saves that border for zippering. After the user has selected a second border, he/she can then zipper them together using the “Zip two borders” button. The program first iterates through all pairs of vertices between the two borders and finds the closest vertex to each one by simply using the equation to get the distance between two points:

$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$. NOME creates a dictionary containing a vertex as a key and the closest vertex from the other border as a value. The program simply iterates through all of the vertices, creating edges between the vertex itself and the closest vertex by looking through all of the keys in the hash table. Then the code iterates through the edges just created for each vertex and creates a face between the current vertex the next vertex and the edge connecting them to the closest vertex from the other border. The zippering of borders is done by adding triangles or quads between the faces. *Figure 5* shows 3 examples of zippering borders.

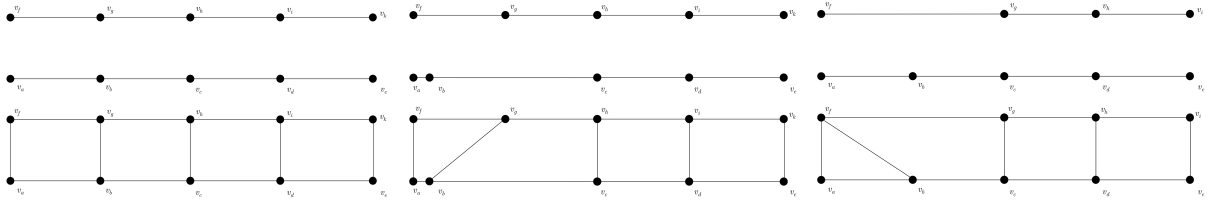


Figure 5: Three example cases of zippering of borders in NOME

5.3 Saving the Changes

Once the user has finished editing the file in the interface, he/she can save the changes back to the NOME file. Saving in the .NOM format will update the current value of each slider and append the

meshes added to the saved file. For example, a user may have defined two instances of a tunnel and created faces between them through the graphical interface. If the user wants to make further edits to the base geometry, such as creating multiple symmetrically placed copies of the faces just generated, that user needs to save the edits and make changes to the .NOM file. A user may also delete faces via the interface by selecting the face and clicking on the delete face button. Deleting a face only deletes the face itself without removing any of the edges or vertices. When saving the edited geometry, NOME also appends a list of deleted faces to the .NOM file.

6. Exporting the File to Be 3D Printed

Once a file has been opened in NOME and edited via the user interface, the user can then merge, subdivide, and add an offset to 3D print the geometry created.

6.1 Merging

The merging step is required before subdividing and offsetting a mesh and allows different instances to be connected with each other. Elements within an individual mesh are connected to each other, however constructs between two different instances will not share their vertices and edges. A merging step is therefore required in order to ensure that during subdivision the mesh is treated as a single mesh and individual mesh parts will not pull apart. During the merging procedure, only border edges are joined so that no non-manifold edges are being generated. Merging between border edges are done within the lowest subtree first. In other words, when multiple border edges can potentially be merged, NOME will take a hierarchical approach in which edges lowest in the scene graph will be merged together first.

Once the merged button has been clicked, NOME creates a new merged mesh that will contain the mesh generated from the merging operation. Merging is implemented by checking whether the vertices of two border edges are within an ε value of each other. If this is true then the two edges are merged by following the rules illustrated in *Figure 6*. In NOME that ε value is set to 0.2 (assuming designs have a diameter or about 1 to 10).

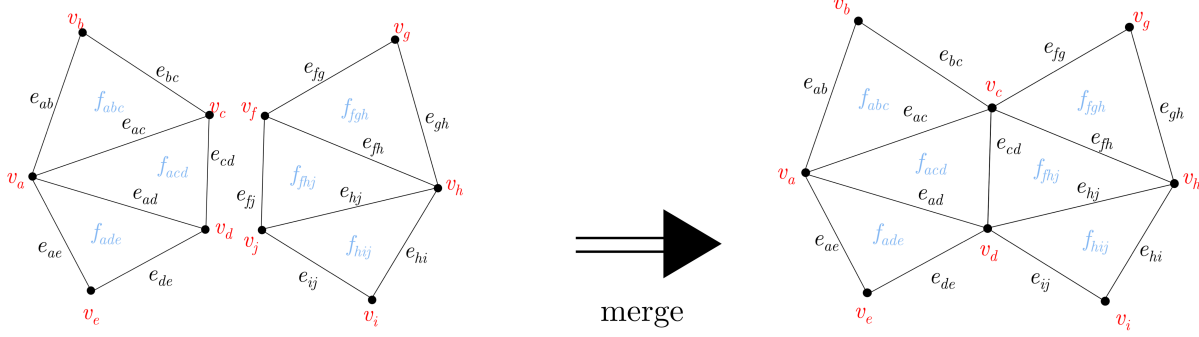


Figure 6: Merging an edge in a mesh.

When merging a pair of coinciding edges, we need to delete one edge and two vertices (*Figure 6*). In *Figure 6*, we need to remove the edge e_{fj} and vertices v_f and v_j . The edges connected to v_f are connected to v_c . The faces adjacent to v_f become adjacent to v_c . Similarly, edges connected to v_j are connected to v_d . The faces adjacent to v_j now comprise v_d . Every reference to v_f and v_j is replaced by references to v_c and v_d .

Once a merged mesh has been created, it is not possible to make edits in the merged view that get transferred back to the regular hierarchical description. Mergers may change the topology and hierarchy dramatically and the new description may no longer be meaningful to the designer. This decision also has consequences for any slider action on a merged mesh. Sliders can still affect the geometry of a mesh. However, when a slider translates one of the meshes that contains one of the edges that have been merged, that merger is no longer possible within the given tolerances (*Figure 7*).

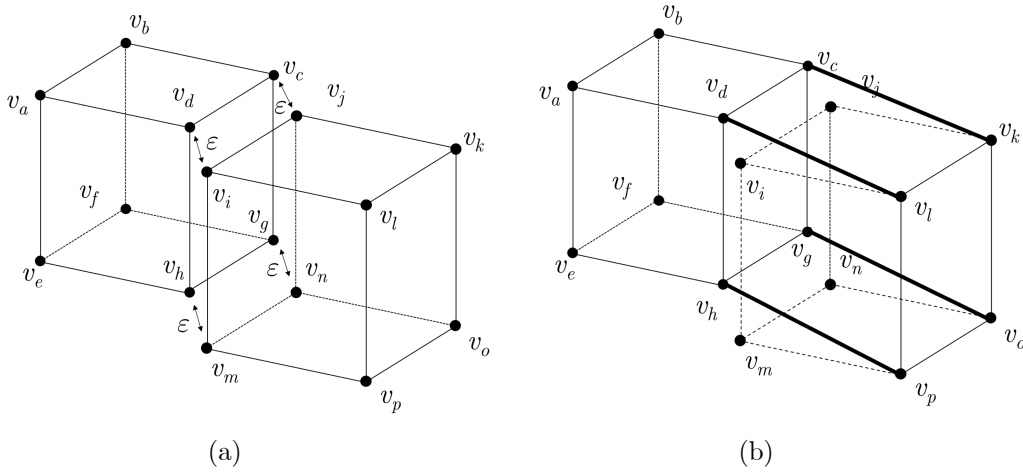


Figure 7: Two merged funnels (a) before and (b) after merging.

If a connection between two meshes must be maintained when one of the edges is moved, then the designer should explicitly insert one or more (“rubber”) faces between the two meshes. Every time a slider is changed, NOME goes back to the hierarchical mesh and creates new mergers between every pair of border edges that coincide within ε .

6.2 Subdivision

After a merging process, the geometry should consist of one or more 2-manifolds plus some polylines. The 2-manifolds can then be subdivided in order to create smooth surfaces. The subdivision code module provides the needed parameters. The *subdiv_type* represents the type of subdivision used. Two types are currently available: *SLF_CATMULL_CLARK* (Regular Catmull-Clark Subdivision) and *WEIGHTED_FACEPOINT_SLF_CATMULL_CLARK* (a modified version of Catmull-Clark weighting the edges of a face according to their length). The advantage of using Catmull-Clark subdivision compared to other types of subdivision, such as Loop subdivision [4], is that it can subdivide faces with any number of vertices. Loop subdivision only supports subdivision for triangle faces. The subdivision module also displays a slider that allows the user to change the current level of subdivision. The *level* parameter (a pure integer) represents the maximum number of subdivisions allowed for the current .NOM file.

6.3 Offset

In order to be 3D printed, the 2-manifold geometry needs to be thickened [7]. The thickening is achieved by adding two offset surfaces: one that is displaced outwards along the surface normal vector and another one the opposite orientation that is moved inwards. In addition, rim faces are generated along the borders of the original 2-manifold to connect the inner and outer offset surfaces into a water tight boundary representation. The offset code module displays a slider that allows to set the offset distance between zero and a maximum value in incremental steps specified by *step* in the offset module.

6.4 STL Output

When the design of some 3D shape has been completed, the user can export it as a .STL file. The .STL file is generated by iterating through the vertices and faces of the hierarchically flattened mesh generated during the merging, subdividing, and offsetting operations. The quadrilateral faces of this mesh are split into two triangles and are output in STL format.

6.5 Other Input and Output Options

To connect the NOME design environment to other programs that generate or modify geometry, NOME can currently open and save files in SIF, STL, and OBJ format. A user may import any generated mesh from a separate CAD tool into NOME supporting these file formats.

7. Implementation Details

7.1 Frameworks Used

NOME has been developed using Qt [15], a cross-platform application framework used to make software running on different operating systems with minimal changes, it uses OpenGL [13] to render the 3D graphics. We have chosen to use these two frameworks in order to maintain and develop the application on multiple operating systems. NOME currently can be run on all major operating systems (Windows, macOS, and Linux), the code simply needs to be compiled separately on each OS.

7.2 Getting References to Constructs

Whenever a hierarchical NOME construct is reference via another instance in the NOME file, NOME traverses the scene graph to find the pointer to that construct. NOME never stores the hierarchical name in the construct itself, because an object in a scene may have multiple paths through the object may be used via multiple instances. Whenever a name is referenced in a NOME file, the program first checks if the first character of the path is a dot, and if it is the case, then NOME will start the search from the root node. After checking for the presence of the initial dot, NOME separates the path in a list of tokens delimited by the dot character. The program then uses breath first search to iterate through each of the constructs in the tree until it sees the correct id at each level. In the past we have considered using a dictionary at the root level of the tree, that would map entire string path names to pointers to the object. Such a dictionary would prevent the user from having to iterate through each object in the tree and the lookup operation would be more efficient. However, if an instance or a face is deleted updating the table would be inefficient as we would have to iterate through each construct using that instance or face and update the path name in that table.

Selection of vertices or faces in the graphical interface is implemented using an octree-based approach [6]. Clicking the cursor on a vertex will return its index. From that index, NOME then iterates through the constructs of the scene graph to return a pointer to that element.

7.3 Updating the Scene when Moving Sliders

Each construct that requires a number contains a double value corresponding to the current value and a string that contains the mathematical expression extracted by the parser. Whenever a slider is changed the NOME file is reinterpreted. All $\{expr \dots\}$ fields are evaluated first. The change of a slider may lead to inappropriate references in the scene if it is mapped to a parameter that deletes or adds vertices in the scene. It is recommended that the user only changes the value mapping to the number of vertices in a funnel, tunnel, or circle, when no vertices in these constructs have been connected to any additional meshes.

7.4 Subdivision Techniques

The subdivided mesh is stored as a separate subdivision mesh at first when the subdivision level is at 0, it simply duplicates the merged mesh. The default subdivision technique used is Catmull Clark subdivision [2] and can be called using the *subdiv_type* called *SLF_CATMULL_CLARK*. In Catmull Clark subdivision, for each level of subdivision, we iterate through all of the faces and create a new face point that is at the position that averages the position of all of the contour vertices. For each edge in the mesh, we then add an edge mid-point that averages the position of the two new neighboring face points and the two endpoints of the edge. Then for each face point we add an edge that connects the new face point to each edge midpoint of the face. Lastly, for each vertex in the mesh, we create a new vertex point, where the new position of the vertex is defined according to the following formula:

$$\frac{F + 2R + (n - 3)P}{n}$$

where P is the original point, n is the number of face points touching P , R is the average of all edge midpoints, F is the average of all face points. We then connect each new vertex point to its adjacent edge points and define new faces between the vertex points, edge points, and face points as in the figure below:

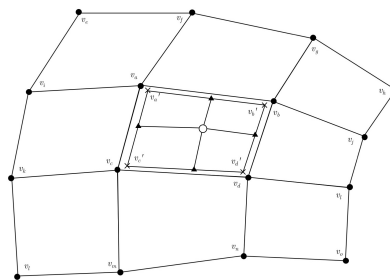


Figure 8: Catmull-Clark subdivision.

If the subdivision slider is changed, then the program checks if the subdivision has already been calculated. If this is the case, it switches a pointer to the previously cached mesh. Each previous level of subdivision only takes $\frac{1}{4}$ of space for quad faces as each quad face is separated into 4 faces. Storing previous levels of subdivision doesn't require significant space compared to the current level of subdivision.

After doing some user testing, we noticed that the default Catmull-Clark subdivision technique is not ideal for every type of meshes. For example, in *Figure 9 (a)*, we can see that subdividing a face with multiple vertices close to each other along an explicitly specified smooth border curve will not create a face point that is at the geometrical center of the face. The vertices $v_a, v_b, v_c, v_d, v_e,$ and v_f on the border curve are pulling the face point towards the top of the face.

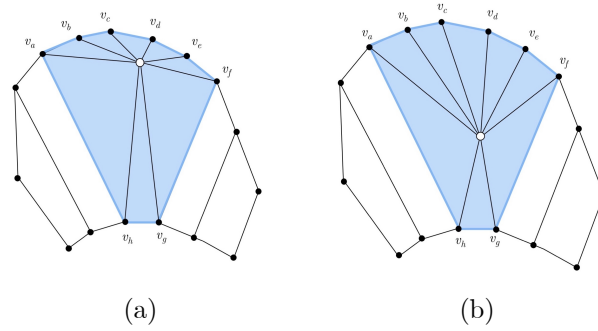


Figure 9: Regular vs. Modified Catmull-Clark subdivision technique.

For this situation, we have introduced a modified Catmull-Clark subdivision technique called *WEIGHTED_FACEPOINT_SLF_CATMULL_CLARK*. Here the weight of contour vertices around a face is given by the length of the edges attached to them. In *Figure 9*, the vertices $v_h, v_g, v_a,$ and v_f would therefore have a greater weight compared to the other vertices as they are connected to longer edges. The face point will therefore be shifted slightly downwards in the face and would produce a smoother surface in the subdivision process.

7.5 Offset Techniques

NOME calculates the normal vector at each vertex by scaling each of the face normal vectors by the angle made between the two edges of the face sharing that vertex. The advantage of this method is that when one of the quads sharing a vertex is split into two triangles, the direction of the vertex normal does not change (*Figure 10*), because the weights of the two triangle normal is scaled down appropriately.

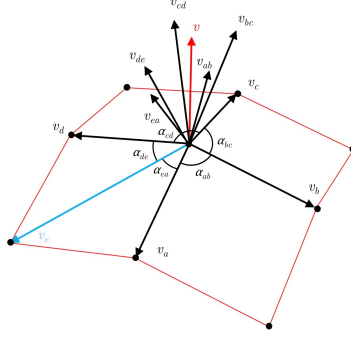


Figure 10: Calculation of a vertex normal using with weighted face normals.

Two vertices are then created at distances equal to \pm half of the offset distance along the vertex normal vector $+\vec{n}_v$. The vertices along the vector \vec{n}_v are then connected together to create the outward offset faces, and the vertices along the vector $-\vec{n}_v$ are connected to create the inward offset surface.

The offset calculation described above assumes that the faces in a mesh form a 2-sided 2-manifold and all have mutually consistent the same orientations. On single-sided surfaces such as Möbius strips or Klein Bottles more analysis is needed, because the normal vectors of some adjacent faces on the surface will be pointing in opposite directions. Such misaligned normal vectors have to be reversed. A “Möbius edge” results when the vertices of one of the faces are defined in a clockwise order while the vertices of the other face are in a counter-clockwise order. Therefore, before calculating the normal vector for each vertex, NOME iterates through each edge and flags all the Möbius edges. For each vertex normal calculation, NOME iterates through all of the adjacent faces and temporarily reverses the direction of the normal when that face is contained between two Möbius edges.

In order to link the front and back offset faces into a watertight boundary surface, NOME needs to create faces along the borders of the offset surfaces. These rim faces are generated by adding faces along the vertices of the outward and inward offset faces.

8. Modeling Sculptures Using NOME

8.1 “Daniela” by Charles Perry

Professor Séquin, several URAPs students, and I have used the NOME program to model some of Charles Perry’s sculptures in a parametrized procedural manner. Charles Perry was an American sculptor who created many 2-manifold sculptures that he called “topological sculptures”. Over his lifespan, he built

hundreds of mathematically inspired sculptures for public spaces in the United States. He is mostly known through sculptures like “Continuum”, a single sided ribbon sculpture in front of the Smithsonian Institution's National Air and Space Museum in Washington, or “Eclipse” in the Hyatt Regency Hotel in San Francisco. As an example of how NOME can be used to model such 2-manifold sculptures, we focused on a simple sculpture titled “Daniela” [5] made by Charles Perry and placed in Carter Residence in Westport, CT, USA (*Figure 11*). The first step was to gather images of “Daniela” (we found only two). “Daniela” can be modeled by first creating an assembly of two tunnels and 5 circles which are then connected by adding faces.



Figure 11: “Daniela” by Charles O. Perry

We started by defining two interlocked tunnels in a .NOM text file. The exact dimensions, position, and tilt-angles are controlled by variable parameters, which are adjusted in an interactive session to best reflect the geometry seen in the photos (*Figure 12 (a)*). In that first interactive session we also remove half of the faces in each tunnel. We then saved and edited the file again in order to add circles to form the outer rims of “Daniela” and portions of the inner border curves that extend the rims of the half-tunnels (*Figure 12 (b)*). The graphical interface is then used to form the faces of “Daniela” by clicking manually on the vertices that needs to be connected (*Figure 12 (c)*).

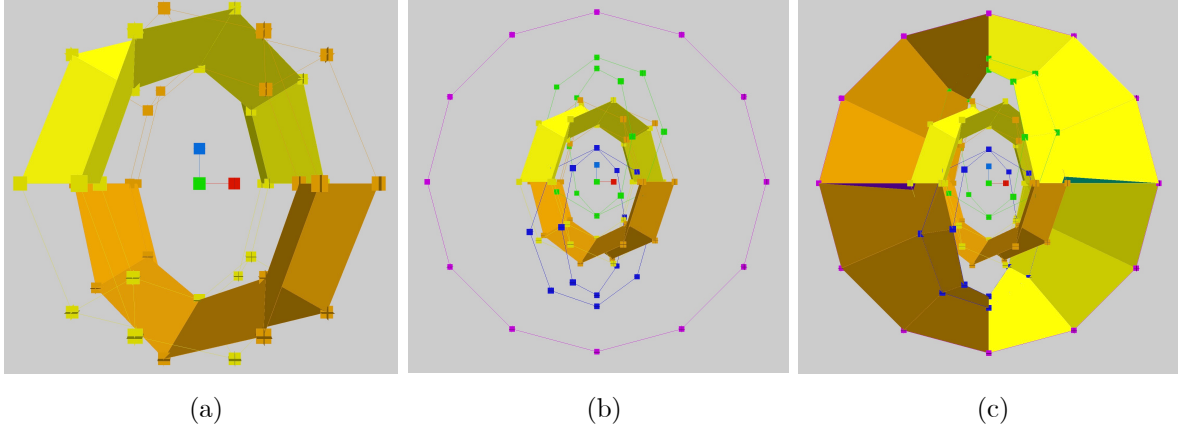


Figure 12: Intermediate steps in constructing “Daniela” by Charles O. Perry in NOME

The placement and radii of the various circles are also parametrized so that they can be lined up against each other as needed (*Figure 13 (a)*). Moreover, after subdividing and offsetting the surface, we can fine-tune the resulting shape by making small changes to the circle parameters (*Figure 13 (b)*). “Daniela” is a single-sided 2-manifold and therefore some Möbius edges (where the front and back face colors come together) are unavoidable. By saving the file in an STL format the geometry can then be 3D printed (*Figure 14*).

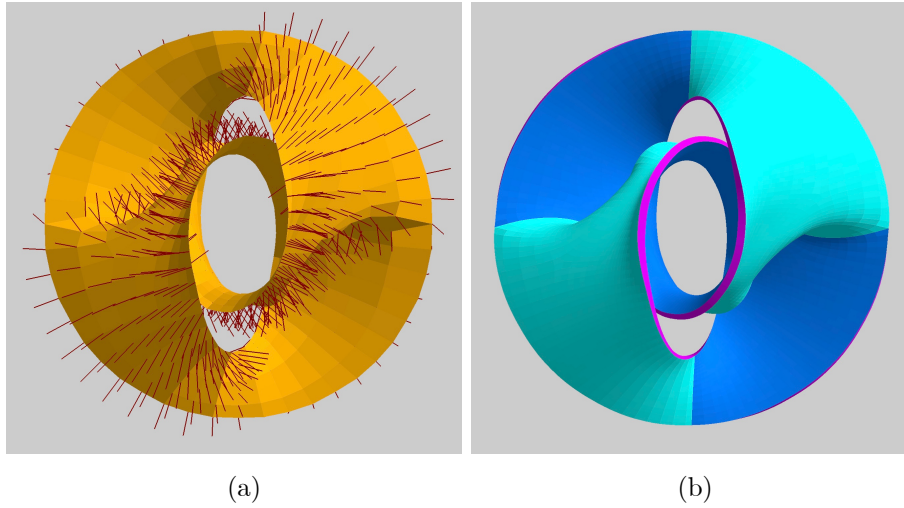
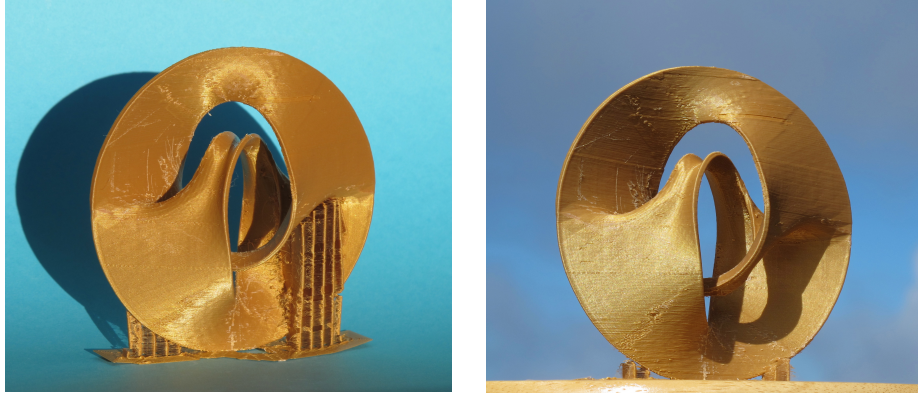


Figure 13: “Daniela” modeled in NOME (a) after subdividing and (b) after offsetting.



(a)

(b)

Figure 14: 3D Printed “Daniela”

Conclusion

We believe that the NOME program will help designers build precise mathematically inspired sculptures in a parametrized manner. The ability to edit the ASCII NOME file and then continue to modify the geometry via a GUI speeds up the workflow of designers and permits the user to build sculpture in a clear hierarchical manner. Precise 2-manifold sculptures can be designed through a NOME script and adjusted via the interface. Small 3D prints allow sculptors and designers to obtain better results since the physical 3D models typically reveal geometric details that were not readily discernable on a 2D computer screen.

Acknowledgement

I would first want to say thank you to Professor Carlo H. Séquin for the time he has spent advising me during the 5th year M.S. program and for all of the discussions that we had in developing NOME. I would also like to say thank you to Professor Björn Hartmann for giving me valuable feedback on this report. I am also very grateful for the help that Toby Chen and Beren Oguz have provided in writing code for NOME. I would like to thank Professor Séquin’s URAP students for their feedback while using early versions of the NOME programming environment.

References

- [1] Baumgart, Bruce G. *Winged edge polyhedron representation*. No. STAN-CS-320. STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1972.

- [2] E. Catmull, J. Clark. *Recursively generated B-spline surfaces on arbitrary topological meshes*. Computer-Aided Design, Vol. 10, Iss. 6, pp. 350-355, 1978. - <https://www.sciencedirect.com/science/article/pii/0010448578901100>
- [3] Levine, John. *Flex & Bison: Text Processing Tools*. “O'Reilly Media, Inc.”, 2009.
- [4] Loop, Charles. “Smooth subdivision surfaces based on triangles.” (1987).
- [5] C. Perry. “Daniela”, 2011. - <http://www.charlesperry.com/sculpture/daniela>
- [6] Peters, Stefan. “Quadtree-and octree-based approach for point data selection in 2D or 3D.” *Annals of GIS* 19.1 (2013): 37-44.
- [7] X. Qu, B. Stucker. *A 3D surface offset method for STL-format models*. Rapid Prototyping Journal, Vol. 9, Iss. 3, pp. 133-141, 2003. - <https://www.emeraldinsight.com/doi/abs/10.1108/13552540310477436>
- [8] C. Séquin. *Homage to Eva Hild*. 2017. - https://people.eecs.berkeley.edu/~sequin/PAPERS/2017_Bridges_HILD.pdf
- [9] C. Séquin. *Multiple 4-stub Dyck funnels linked into symmetrical clusters based on Platonic and Archimedean polyhedra*. 2017. <http://people.eecs.berkeley.edu/%7Esequin/MATH-MODELS/Dyck4-Clusters/Dyck4-Clusters.pdf>
- [10] J. Smith, SLIDE design environment, (2003). - <http://www.cs.berkeley.edu/~ug/slide/>
- [11] J. P. Smith, S. A. McMains, C. H. Séquin *SIF: A Solid Interchangeable Format for Web-based Prototyping*
- [12] Y. Wang, Robust Geometry Kernel and UI for Handling Non-orientable 2-Manifolds (EECS-2016-65). - <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-65.html>
- [13] OpenGL, *OpenGL Homepage*. - <https://www.opengl.org>
- [14] Python Blender Documentation Contents, <https://docs.blender.org/api/current/>
- [15] Qt, *Qt Homepage*. - <https://www.qt.io/>
- [16] Type A machines. - <https://www.typeamachines.com/>
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.25.3714&rep=rep1&type=pdf>

Appendix A: Input NOME file for the “Daniela”

```
### Daniela.nom
#
# Another Perry sculpture in a disk
# A fists modeling approach, starting with 2 half-tunnels ...
# ... add a secondary arc to allow for a more concave disk surface.
# ... fine-tune the saddles between the ends of the half-tunnels.
# More fine tuning after first 3D print. Also try different orientations for printing.

### CHS 2018/4/20

#### Some Surface colors ####
surface R color (1 0.1 0) endsurface # Red
surface O color (1 0.7 0) endsurface # Orange
surface Y color (1 1 0) endsurface # Yellow
surface br color (0.4 0.4 0) endsurface # brown
surface L color (0.5 0.8 0) endsurface # Lime
surface G color (0.1 1 0) endsurface # Green
surface A color (0 0.9 0.7) endsurface # Aqua
surface C color (0 1 1) endsurface # Cyan
surface U color (0 0.5 1) endsurface # Uniform
surface B color (0.1 0.1 1) endsurface # Blue
surface P color (0.6 0 1) endsurface # Purple
surface M color (0.9 0 1) endsurface # Magenta
surface Z color (1 0 0.5) endsurface # Zinnober
surface S color (0.8 0.8 0.8) endsurface # Snow(dirty)
surface W color (1 1 1) endsurface # White

#### Display settings ####
foreground surface O endforeground
background surface S endbackground
insidefaces surface U endinsidefaces ## really "outside"
outsidefaces surface C endoutsidefaces ## really "inside"
offsetfaces surface M endoffsetfaces

#### Some elementary reference geometry #####

## Coordinate axes:
point ooo ( 0 0 0 ) endpoint
point xoo ( 0.3 0 0 ) endpoint
point oyo ( 0 0.3 0 ) endpoint
point ooz ( 0 0 0.3 ) endpoint

polyline xaxis (ooo xoo) endpolyline
polyline yaxis (ooo oyo) endpolyline
```



```
polyline zaxis (ooo ooz) endpolyline
```

```
instance xax xaxis surface R endinstance
instance yax yaxis surface U endinstance
instance zax zaxis surface G endinstance
```

```
##### Defining the two half-tunnels #####
```

```
bank ft
```

```
set      tn      8      3      10      1
set      tro      1      0.1    2.0    0.1
set      ratio    0.4    -0.5    0.5    0.1
set      th      0.4     0.0    1.0    0.1
```

```
set      tsep     0.0    -1.0    1.0    0.1  ## displacement
set      ttilt    0      -30     30.0    1    ## angle between half-tunnels
```

```
endbank
```

```
tunnel tun ( {expr $ft.tn} {expr $ft.tro} {expr $ft.ratio} {expr $ft.th} ) endtunnel
```

```
instance tump tun surface Y rotate (0 1 0)({expr 45+$ft.ttilt}) translate ( 0
{expr $ft.tsep} 0 ) endinstance
instance tunn tun surface 0 rotate (0 1 0)({expr -45-$ft.ttilt}) translate ( 0
{expr -$ft.tsep} 0 ) endinstance
```

```
## Get rid of unwanted half-tunnels:
```

```
delete
```

```
face tump.f1_4 endface
face tump.f1_5 endface
face tump.f1_6 endface
face tump.f1_7 endface
face tump.f2_7 endface
face tump.f2_6 endface
face tump.f2_5 endface
face tump.f2_4 endface
```

```
face tunn.f1_0 endface
face tunn.f1_1 endface
face tunn.f1_2 endface
face tunn.f1_3 endface
face tunn.f2_3 endface
face tunn.f2_2 endface
face tunn.f2_1 endface
face tunn.f2_0 endface
```

```
enddelete
```

```
##### Defining the rim and two additional arch-curves #####
```

```
bank cb
  set   ocn      12      3      24      1
  set   ocrad     3      1      5.0    0.1  ## outer disk rim

  set   icn       8      3      10      1
  set   icrad    1.3    0.1    3.0    0.1  ## inner arc curve
  set   icsep    0.7    0.1    3.0    0.1
  set   atilt    20    -30    30.0    1

  set   mcn       8      3      10      1
  set   mcrad    1.4    0.1    3.0    0.1  ## mid arc curve
  set   mcsep    0.8    0.1    3.0    0.1
  set   mtilt    -5    -30    30.0    1
endbank
```

```
circle ocirc ( {expr $cb.ocn} {expr $cb.ocrad} ) endcircle
```

```
instance oc ocirc surface M endinstance
```

```
circle iarc ( {expr $cb.icn} {expr $cb.icrad} ) endcircle
```

```
instance arp iarc surface G rotate (0 1 0)({expr -45-$ft.ttilt-$cb.atilt})
translate ( 0 {expr $cb.icsep} 0 ) endinstance
instance arn iarc surface B rotate (0 1 0)({expr 45+$ft.ttilt+$cb.atilt})
translate ( 0 {expr -$cb.icsep} 0 ) endinstance
```

```
## >>> An additional intermediate arc between the above one and the rim,
##      so that the disc surface cab be nade concave to blend more naturally into the
##      half-tunnels.
```

```
circle marc ( {expr $cb.mcn} {expr $cb.mcrad} ) endcircle
```

```
instance marp marc surface G rotate (0 1 0)({expr -45-$ft.ttilt-$cb.mtilt})
translate ( 0 {expr $cb.mcsep} 0 ) endinstance
instance marn marc surface B rotate (0 1 0)({expr 45+$ft.ttilt+$cb.mtilt})
translate ( 0 {expr -$cb.mcsep} 0 ) endinstance
```

```
##### Starting to fill in the outer disk surface #####
```

```
face rf0 (marp.v2 marp.v1 oc.v2 oc.v3 ) endface
face rf1 (marp.v1 marp.v0 oc.v1 oc.v2 ) endface
```

```

face mf0 (arp.v2 arp.v1 marp.v1 marp.v2 ) endface
face mf1 (arp.v1 arp.v0 marp.v0 marp.v1 ) endface
face ff0 (marp.v0 arp.v0 tunn.f1_7.v1_0 tunn.f1_7.v2_0 ) endface
face ff1 (oc.v0 oc.v1 marp.v0 tunn.f1_7.v2_0 tunn.f2_7.v3_0 ) endface

object rim (rf0 rf1 mf0 mf1 ff0 ff1 ) endobject

instance ri0 rim surface Y endinstance
instance ri1 rim surface 0 rotate (0 1 0)(180) endinstance
instance ri2 rim surface Y rotate (1 0 0)(180) endinstance
instance ri3 rim surface 0 rotate (0 1 0)(180) rotate (1 0 0)(180) endinstance

## >>> The challenging puzzle is how to complete the surface ...
##      ... particularly, making nice saddles between the ends of the half-tunnels.

mesh mm
face f1 (tunp.f1_0.v1_1 tunp.f1_0.v1_0 oc.v0 tunn.f2_7.v3_0 tunn.f2_6.v3_7 )
surface A endface
face f3 (tunn.f1_4.v1_5 tunn.f1_4.v1_4 oc.v6 tunp.f2_3.v3_4 tunp.f2_2.v3_3 )
surface P endface
endmesh

instance im mm endinstance

#####

subdivision subdiv type SLF_CATMULL_CLARK subdivisions 6 endsubdivision

offset fatty type WEIGHTED min 0.0 max 0.4 step 0.02 endoffset

#####

```