

Joining Interactive Graphics and Procedural Modeling for Precise Free-Form Designs

*Randy Fan
Carlo H. Séquin, Ed.
Ren Ng, Ed.*



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-125

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-125.html>

May 14, 2021

Copyright © 2021, 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 like to express my deepest gratitude to my advisor, Professor Carlo H. Séquin. His support and mentorship over the past few years have helped me grow into a well-rounded engineer. His passion for 3D modeling is contagious and has made me appreciate the beautiful connection between art and mathematics. Most of all, I would like to thank him for the life values that he has instilled in me through our wonderful discussions - among many are pursuing things in life that interest me and leading teams skillfully.

I would also like to thank Professor Ren Ng for being an amazing role model and helping me reach my potential. He has supported me both academically and professionally on many occasions, and his courses inspired me to delve deeper into the field of computer graphics.

Joining Interactive Graphics and Procedural Modeling for Precise Free-Form Designs

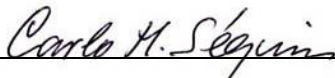
by Randy Fan

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

05/08/2021

(Date)

* * * * *



Professor Ren Ng
Second Reader

05/12/2021

(Date)

Joining Interactive Graphics and Procedural Modeling for Precise Free-Form Designs

Randy Fan

Electrical Engineering and Computer Science, University of California, Berkeley

randyfan@berkeley.edu

Abstract

JIPCAD (Joint-Interactive-Procedural CAD) is a 3D procedural CAD tool used for programmatically creating geometries with a shape description language. Users can interactively modify the scene in a graphical user interface and save modifications back into the corresponding file as reusable code; a reopening of the enhanced file will reproduce the latest graphical state, and the user can continue in either a graphical manner or by textual changes in the *.jip* file.

JIPCAD combines procedural and interactive modeling, making it easier for users to model 2-manifold free-form surfaces of high complexity and inherent regularity. In this report, we will discuss JIPCAD and the development of its shape description language and graphical editing capabilities over the past year. Key contributions include generalized progressive sweeps along arbitrary 3D space curves, dynamic scenes, advanced shape generators for tori, error catching, graphical editing and saving capabilities, and advanced rendering options.

1. Introduction

There are many existing 3D modeling tools out there in the market, such as Blender, OpenSCAD, and Maya; however, these tools do not strike a good balance between procedural shape creation and interactive graphical editing capabilities. For example, Maya and Blender rely heavily on a click and drag user interface, which is imprecise compared to a procedural method.

OpenSCAD is an open-source script-based 3D CAD tool that can be used for precisely placing objects in the scene and easily modifying their parameters [1]; for example, if a user wants to adjust the size or number of wheels on a truck model, this could be as simple as changing the corresponding parameter values in the script used to generate the scene. In OpenSCAD and script-based CAD tools in general, the scene's corresponding code is text-readable and can be reused easily by other designers.

However, OpenSCAD has several limitations that make the tool inflexible for modeling 2-manifold free-form surfaces. OpenSCAD's user interface does not allow shapes to be interactively modified via a mouse

cursor. The cursor cannot be used to select mesh faces and vertices in the window and can only be used for navigating the scene. This is problematic for designers who want to customize and configure their models after deployment. Furthermore, creating subdivided shapes is difficult with OpenSCAD without explicit merge and subdivision functions available.

OpenSCAD also has a limited set of 2D and 3D shape generators, containing only basic primitives such as circle, square, cube, and cylinder generators [1]. This limitation makes it difficult for users to construct more complex shapes such as a torus knot, which could have been easily constructed if there were existing tori generators.

Blender, a free and open-source 3D modeling tool used in many animated films, added Python scripting as an option to automate certain design tasks, but the scripting does not preserve the scene hierarchy when the objects are rendered [2]. This means changes that are made interactively in the GUI cannot be efficiently saved back into the code file when the scene is created using their Python scripts.

Berkeley SLIDE (Scene Language for Interactive Dynamic Environments) is a CAD tool originally developed in the early 2000's and can be used to construct abstract geometrical sculptures with a shape description language [3]. However, it has not been maintained for over a decade and is not compatible with recent versions of operating systems. Implementation-wise, SLIDE represents meshes as two-sided surfaces, so it is limited in its ability to subdivide and offset single-sided surfaces, such as Möbius bands and Klein bottles.

In 2018, NOME (Non-Orientable Manifold Editor) was introduced to handle single-sided, non-orientable surfaces and to add interactive graphical editing capabilities [4]. The initial version of NOME offered only a few of the procedural shape generators that were available in SLIDE, and it was difficult to save the changes that were made graphically in a form compatible with the procedural scene description file. Also, its implementation code was rather "ad-hoc" and made it difficult to enhance NOME's capabilities.

These past efforts and related tools have not found a good balance between procedural mesh generation and interactive GUI modifications. Thus, we have developed a new modeling tool, called JIPCAD (Joint-Interactive-Procedural CAD), that extends NOME by re-implementing it on a more robust, well-structured code base and by enhancing the library of predefined shape generators. Many additional modification modes were added to the graphical user interface, and the means of saving those changes and appending it to the original procedural JIPCAD file were improved. The challenges involved are not only generating the shapes and preserving their hierarchical relationships, but also saving interactive

changes back into the code for reuse. We decided to rebrand NOME as JIPCAD since the tool’s ability to handle non-orientable surfaces is no longer its distinguishing feature.

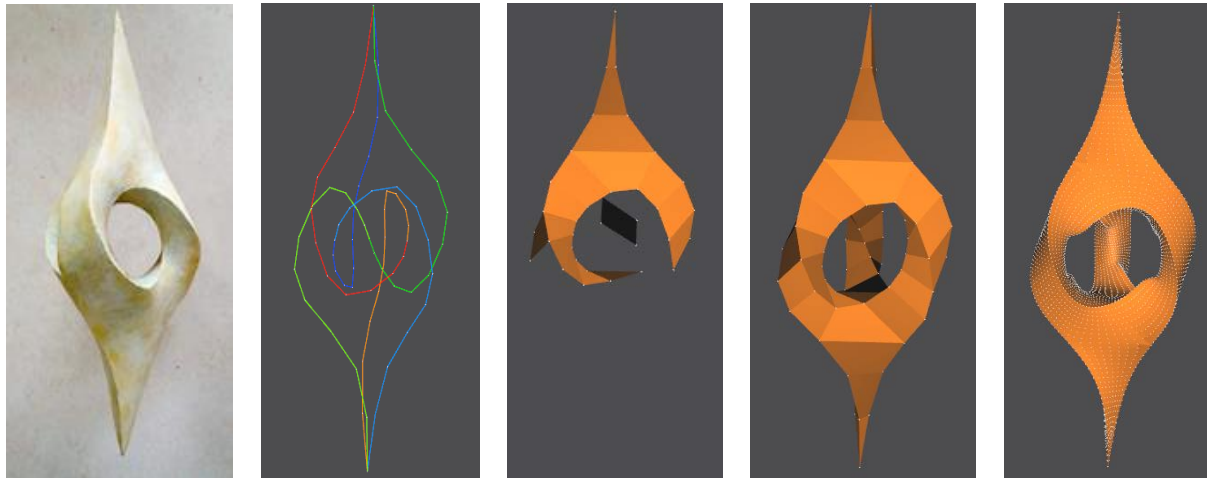


Figure 1: The “3-2-1” Sculpture: (a) Sculpture by Tord Tengstrand; (b) B-splines added; (c) intermediate construction step; (d) completed construction; (e) sharp subdivision applied. (see Appendix A3)

Since I joined the JIPCAD project in 2019, we have made the tool more robust and introduced features such as generalized progressive sweeps along arbitrary 3D space curves, dynamic scenes, advanced shape generators for tori, graphical editing and saving capabilities, error catching, a user-friendly crystal ball interface [5], advanced rendering options, and more. We have also fixed the mesh data structure to handle various non-orientable surfaces while being compatible with our newly added features. JIPCAD users can generate topologically complex 2-manifolds through an iterative workflow, which may start with procedurally generated B-spline curves, to which discrete surface facets are added through a graphical user interface. This is demonstrated in the construction of the “3-2-1” Sculpture by Tord Tengstrand [6] in *Figure 1*.

2. Basic Commands

The three basic entities in JIPCAD are point, face, and polyline. These entities are often assembled to form meshes and are initialized using the following generator commands:

Point: *point id (x y z) endpoint*

Polyline: *polyline id (point_idlist) [closed] [surface surface_id] endpolyline*

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

All generator commands are specified with their command type (e.g., *polyline*), an *id* that can be used to reference the entity, and the entity's parameters. The *id* needs to be unique; JIPCAD uses a single assignment language that does not allow for duplicate names. Command types and parameter names are reserved and cannot be used as *ids*. For example, *face* cannot be used as an *id*.

Parameters enclosed in parentheses are required while parameters enclosed in brackets are optional. In the above commands, the point's parameters are the x, y, z coordinates, while the polyline and face's parameters are the *ids* of their associated points. The list of point *ids* must have length of at least 2 for polylines and at least 3 for faces. The polyline and face can optionally be assigned a color by passing in a surface identifier. The exact coloring convention is described in the *Hierarchical Coloring Scheme* section. Lastly, all commands end with an *end* statement concatenated with their command type (e.g., *endpolyline*).

A user can instantiate any of the above entities as well as other shape generators by using the instance command.

Instance: *instance instance_id target_id [rotate (rx ry rz) {in degrees}] [scale (sx sy sz)] [translate (tx ty tz)] [surface surface_id] [LOD LOD_type] [shading shading_type] endinstance*

The instance command creates an instance of the target geometry, which flags the geometry to be rendered in the scene. One can optionally rotate, scale, translate the instance, and specify the shape's color, level-of-detail (LOD), and rendering mode.

Detailed descriptions for all commands can be found in the JIPCAD language reference [7].

3. JIPCAD Hierarchies

3.1 Hierarchical Constructs

JIPCAD users can assemble shapes in group or mesh commands. This is useful for defining scenes in a hierarchical manner where identical geometries and symmetrical components can be defined once and reused.

Group:

group id

instance id1 object_id1 [instance_parameters] endinstance

...

```

instance idN object_idN [instance_parameters] endinstance
endgroup

```

The group command defines a collection of shape or other group instances and is the most general hierarchical construct designed to introduce hierarchy into the scene description. When a group gets instantiated via an instance command, the group's collection of instances gets added to the scene.

Mesh:

```

mesh id
  point pointId1 id1 (x y z) endpoint
  ...
  point pointIdN (x y z) endpoint
  face faceId1 (point_idlist1) [surface surface_id] endface
  ...
  face faceIdN (point_idlistN) [surface surface_id] endface
endmesh

```

The mesh command defines a collection of points and/or faces. The faces can be optionally colored by referencing an existing surface entity. Faces within mesh commands can be referred to in the *.jip* file with a hierarchical name: *id.faceId*. This is useful if a user wants to perform further operations on the face or reuse the face in another command.

In *Figure 2*, a cube is created by defining a mesh consisting of a single square face and then instantiating that mesh six times within a group construct; each instance of the mesh is applied the proper rotation transformation to ensure the faces together form a cube. We can then instantiate the entire group to render all its components and form the cube. Orange is the default surface color.



```

point p1 (1 1 1) endpoint
point p2 (1 -1 1) endpoint
point p3 (-1 -1 1) endpoint
point p4 (-1 1 1) endpoint

mesh one_face
  face f1 (p4 p3 p2 p1) endface
endmesh

group cube
  instance front one_face endinstance
  instance back one_face rotate (0 1 0) (180) endinstance
  instance left one_face rotate (0 1 0) (-90) endinstance
  instance right one_face rotate (0 1 0) (90) endinstance
  instance top one_face rotate (1 0 0) (-90) endinstance
  instance bottom one_face rotate (1 0 0) (90) endinstance
endgroup

instance cube0 cube endinstance

```

Figure 2: Cube and corresponding *.jip* file.

3.2 Hierarchical Scene Graph

A directed scene graph is used to keep track of the hierarchical relationships present in a scene, giving us a method to identify entities located within hierarchical constructs, such as group or mesh commands. Any entity can be referenced by its scene graph path and its own name separated by periods. Vertices are defined once and can be referenced by multiple faces (references are depicted as arrows in *Figure 3*). The green box in the bottom right of *Figure 3* is referencing the vertex *v3* underlined green named by using the identifier *.g5.g3.o3.v3*. This means changes made to the original vertex *v3* (e.g., modifying its position with a slider) are propagated across the hierarchy. The yellow box is referencing the vertex *v4* underlined yellow via the identifier *.g7.g9.mE.v4*.

The root node of the graph is labeled *RENDER WORLD*, and its children are the global instance nodes. Global instances are instances not found within a group. An instance node's child is its target geometry, which can be a group, mesh, or pre-defined shape generator. Each target geometry has its own set of children. For example, a group node's children are its instance commands, and a mesh node's children are its associated faces and points.

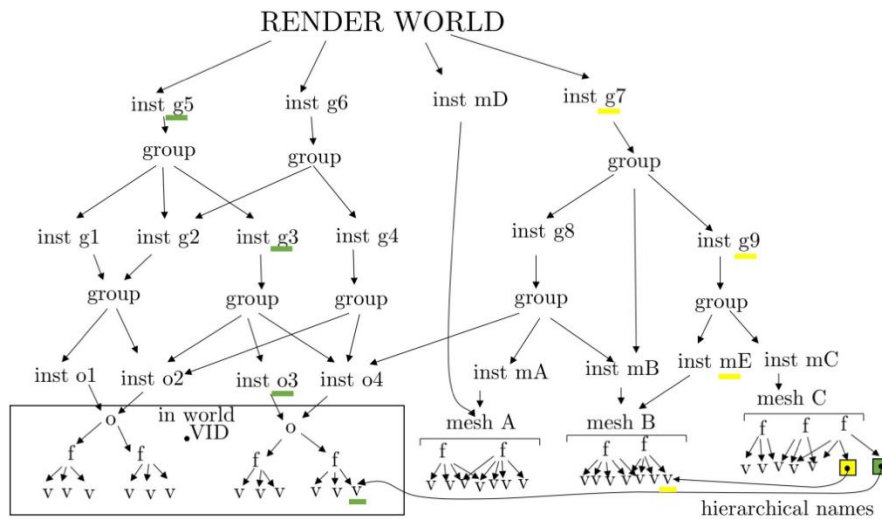


Figure 3: Scene Graph [4].

3.3 Hierarchical Coloring Scheme

The surface command is used to assign colors to objects. It has an *id* and user-defined RGB value in the range [0, 1].

Surface color: *surface id (R G B) endsurface*

Key geometrical constructs such as faces, polyline-related entities (polylines, B-splines, and Bézier curves), and all instances can be assigned a surface color by passing in the *surface surface_id* as an argument.

JIPCAD follows a hierarchical coloring scheme convention. This means that entities that have been assigned a surface color will not change their color when a higher-up group or instance that they are part of gets re-colored. For groups, only elements that have not yet been colored will accept the new group color.

In *Figure 4*, the cube's top face is assigned a green surface color while the entire cube group is assigned blue. Since the top face is within the group, the green surface color overrides blue for the top face. The other cube faces are colored blue because they are not assigned any colors lower in the hierarchy. If we assigned the surface color red to the mesh face *f1*, then the whole cube would become red as *f1* is positioned relatively lower in the scene graph hierarchy than the group and global instance commands *f1* is a part of.

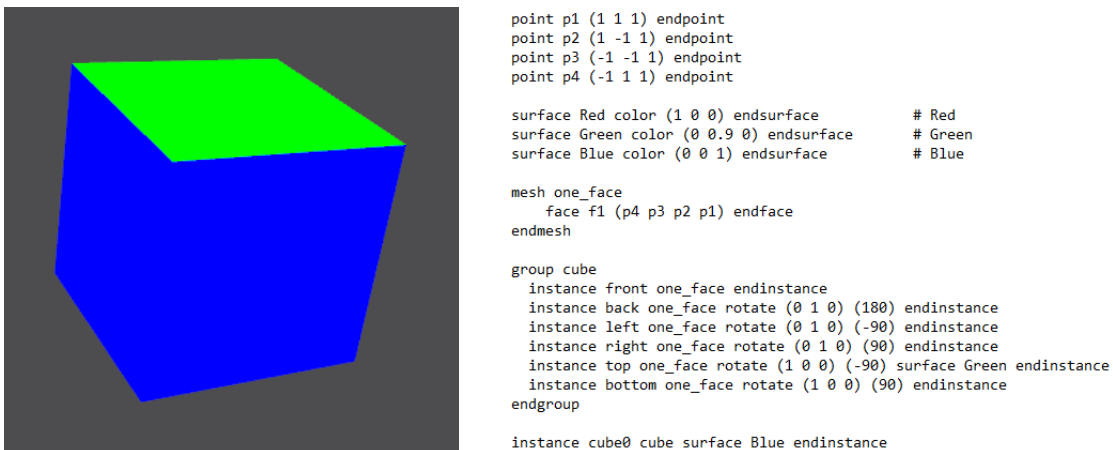


Figure 4: Colored cube and corresponding *.jip* file.

4. Sliders

Sliders can be used to interactively change numerical parameter values in the scene description. A user can simply click and drag a slider in the GUI and alter the slider's associated value. This is useful for interactively modifying a scene and determining which combination of shape parameter values can create the desired output. After the slider has been adjusted, the user can click on the "Commit Changes" button in the GUI as shown in *Figure 5* and then type "Ctrl+S" or click "Save" to save the altered parameter

value as the new default slider value upon reloading. This means the slider's default value in the *.jip* file is updated to reflect this change.

Bank:

bank bankID

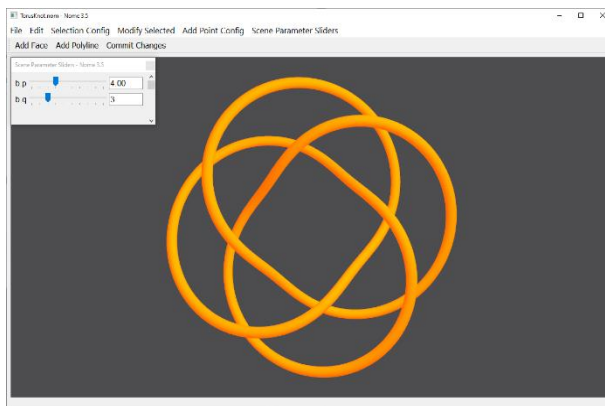
set setID1 default_value1 start1 end1 step_size1

...

set setIDN default_valueN startN endN step_sizeN

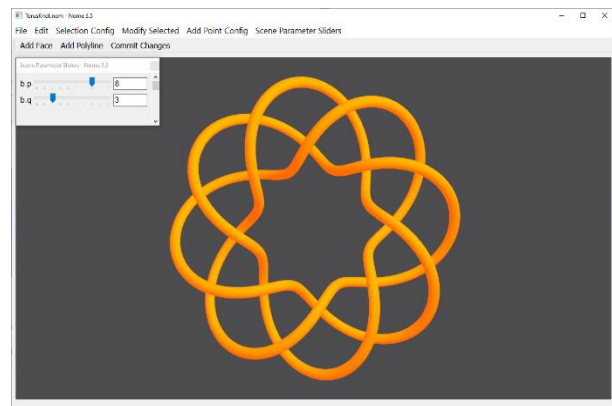
endbank

Banks are a collection of sliders. The bank's sliders can be used in any generator command by setting *\$bankId.setID* as the numerical parameter value. In *Figure 5*, the torus knot's default starting *p* value is four as specified in its corresponding *.jip* file. If we move the slider such that *p* becomes eight, the torus knot in the scene updates accordingly. We can then save the changes back into the *.jip* file so the torus knot's new default starting *p* value is eight.



```
bank b
  set p 4 1 10 1
  set q 3 1 10 1
endbank
```

```
torusknot tk ($b.p $b.q 5 2 0.25 20 500) endtorusknot
instance tk1 tk endinstance
```



```
bank b
  set p 8 1 10 1
  set q 3 1 10 1
endbank
```

```
torusknot tk ($b.p $b.q 5 2 0.25 20 500) endtorusknot
instance tk1 tk endinstance
```

Figure 5: Modifying the *p* parameter via sliders and updating the *.jip* file.

5. JIPCAD Language Development

Over the past year, we have added dozens of new commands and features to the JIPCAD language. The new commands that have been the most impactful to the design and display of non-manifold surfaces are listed below along with a few non-traditional surface generators. The following subsections describe these

commands more in-depth. The full language reference describes all commands including those not listed here [7].

Frame: *\$frame*

Time: *\$time*

Bézier Curve: *beziercurve id (point_idlist) segs endbeziercurve*

B-Spline: *bspline id order (point_idlist) segs endbspline*

Sweeps: *sweep id crosssection id [reverse] [begincap] [endcap] endcrosssection path id [mintorsion] [azimuth a_angle] [twist t_angle] endpath [brep brep-type] endsweep*

Control Point: *controlpoint id point id scale (sx sy sz) rotate (rx ry rz) [startreverse or endreverse] cross id endcontrolpoint*

General Cartesian Surface: *gencartesiansurf id func (x_min x_max y_min y_max x_segs y_segs) endgencartesiansurf*

General Implicit Surface: *genimplicitsurf id func (x_min x_max y_min y_max z_min z_max x_segs y_segs z_segs) endgenimplicitsurf*

General Parametric Surface: *genparametricsurf id func (u_min u_max v_min v_max u_segs, v_segs) endgenparametricsurf*

Subdivision: *subdivision id [sd_type sd_flag] sd_level [instances] endsubdivision*

Offset: *offset id [offset_type offset_flag] height width [instances] endoffset*

Torus: *torus id (rad_maj rad_min theta_max phi_min phi_max segs_theta segs_phi) endtorus*

Torus Knot: *torusknot id (symm turns rad_maj rad_min rad_tube segs_circ segs_sweep) endtorusknot*

We have also implemented generator commands for a disk, cylinder, cone, ellipsoid, sphere, Möbius strip, and Dupin cyclide. The command descriptions can be found in the JIPCAD language reference.

Light: *light id type (JIP_AMBIENT or JIP_DIRECTIONAL) color (R G B) endlight*

Camera: *camera id projection (JIP_PARALLEL or JIP_PERSPECTIVE) frustum (min_triple) (max_triple) endcamera*

Window: *window Window (xmin ymin) (xsize ysize) background (R G B) endwindow*

Viewports: *viewport vp (xmin ymin) (xmax ymax) background (R G B) endviewport*

Include: *include file_name.jip endinclude*

5.1 Dynamic Scenes

We have introduced two global variables, *\$time* and *\$frame*, to handle time-varying models. These variables can be used in any expression that evaluates a numerical value in JIPCAD. *\$frame* is an integer

value that gets incremented by +1 after every rendering of the scene. All expressions comprising $\$frame$ get updated before the scene is rendered again. $\$time$ is a real value that keeps track of elapsed time in seconds. After a rendering, the system-clock is interrogated and compared to the remembered system clock value after the previous rendering. The time difference is then added to the $\$time$ variable, and all expressions comprising $\$time$ get updated before the scene is rendered again.

5.2 Path Entities

Polylines, B-splines, and Bézier-curves are three path entities available in JIPCAD. These entities can be instantiated and rendered as piecewise linear segments or used as a sweep path in the sweep command.

Polyline: *polyline id (point_idlist) [closed] [surface surface_id] endpolyline*

B-spline: *bspline id order (point_idlist) segs [closed] [surface surface_id] endbspline*

Bézier curve: *beziercurve id (point_idlist) segs [closed] [surface surface_id] endbeziercurve*

A B-spline has an *order* parameter, which is an integer that sets the B-spline's degree to be *order-1*. The *point_idlist* is a list of point ids, and the referenced points behave as the B-spline's control points. Note the number of control points has to be greater than or equal to *order*. For closed curves, there must be at least *order-1* control points.

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 function:

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

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

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

For $n \neq 0$:

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

B-splines are useful for constructing the edges of curved surfaces as shown in *Figure 6*.

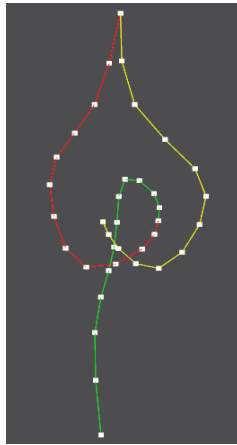
A Bézier curve's *point_idlist* references a list of point entities that are used as the curve's control points. According to De Casteljaeu's algorithm, a Bézier curve with $n+1$ control points (c_0, c_1, \dots, c_n) can be evaluated at a point using the following function:

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

Where $t \in [0, n + (\{order\} - 1)]$ and b is defined as the basis defined as following for $n = 0$:

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

In both the B-spline and Bézier curve commands, *segs* is the number of segments into which they are sampled.



```

surface R color (1 0 0) endsurface          # Red
surface G color (0 0.9 0) endsurface        # Green
surface Y color (1 1 0) endsurface          # Yellow

point eaR (0.3 6 0) endpoint
point eb (0 4 0) endpoint
point ecR (-0.3 2 0) endpoint

point wd (-0.5 1 1.5) endpoint
point we (-0.4 0.0 1.5) endpoint
point wf (-0.2 -1.2 0.8) endpoint

point wg (0.8 -0.5 0.2) endpoint
point wh (0.8 0 0) endpoint
point wi (0.8 0.5 -0.2) endpoint

bspline bs order 4 (eaR eb ecR wd we wf wg wh wi) segs 12 endspline

instance bsR bs surface R endinstance
instance bsRu bs surface G rotate(1 0 0)(180) endinstance
instance bsG bs surface Y rotate(0 1 0)(120) endinstance

```

Figure 6: Three B-splines defining part of the edge structure of the in the Tord Tengstrand sculpture [6], each shown as a 12-segment polyline, with highlighted vertices, through which new surface facets may be defined.

5.3 Sweeps

We have introduced a sophisticated sweep procedure that can sweep an arbitrary 2D cross-section along a sweep path. The cross-section and the sweep path must be a polyline, Bézier curve, B-spline, or circle.

Sweep:

sweep id

crosssection id [reverse] [begincap] [endcap] endcrosssection
path id [mintorsion] [azimuth a_angle] [twist t_angle] endpath

[brep brep-type]
endsweep

Each path has a set of Frenet frames (tangents, normals, and binormals) that are used to determine how the *crosssection* will twist along the path. The user can control the twist in four ways: *mintorsion* minimizes the twisting of the intrinsic Frenet frame, *azimuth* is the angle about the tangent through which all Frenet frames will be rotated, *twist* is the angle about the tangent that specifies the overall amount of twist from the first Frenet frame to the last, and *warp* sets each twist angle explicitly at specified *controlpoints* in the path.

Control Point: *controlpoint id point id scale (sx sy sz) rotate (rx ry rz) [startreverse or endreverse] cross id endcontrolpoint*

The *controlpoints* also permit rotating and non-uniformly scaling of the *crosssection* at their locations. At any sample points between adjacent *controlpoints*, the transformation variables are interpolated in the same way that the x, y, z coordinates are being interpolated (e.g., by a cubic polynomial for the cubic B-spline). Given a list of control points, we can tag a control point as *startreverse* and another as *endreverse*; then, the orientation of all sweep faces between these two control points are reversed, turning their generated brep-surface inside out. Control points can also reference a cross section by assigning a path-related entity *id* to *cross*. Regular path points act like control points with no additional transformations.

begincap and *endcap* can be used to draw the starting and ending faces of sweeps along an open path (with outward normal) respectively.

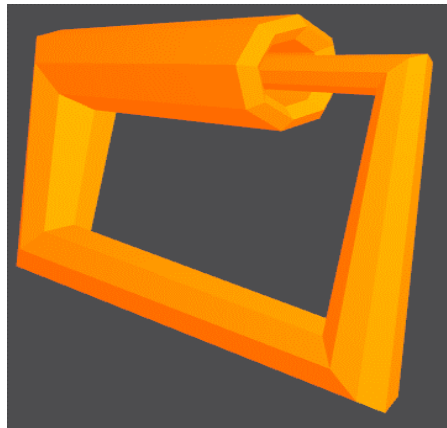


Figure 7: A sweep with local profile inversions. (see Appendix A1)

5.4 Surfaces Generated from Mathematical Expressions

JIPCAD users can also define surfaces in 3D space via mathematical expressions that define points in general Cartesian or in parametric coordinate spaces, or in an implicit form [8].

General Cartesian Surface: `gencartesiansurf id func (x_min x_max y_min y_max x_segs y_segs)`
`endgencartesiansurf`

General Implicit Surface: `genimplicitsurf id func (x_min x_max y_min y_max z_min z_max x_segs y_segs z_segs)`
`endgenimplicitsurf`

General Parametric Surface: `genparametricsurf id func (u_min u_max v_min v_max u_segs, v_segs)`
`endgenparametricsurf`

The Cartesian generator's *func* parameter takes in an equation of the form $z = f(x,y)$. The generator has parameters for the minimum and maximum x and y values and the number of segments in the x and y ranges.

The implicit surface's *func* is the equation $f(x,y,z) = b$, where b is the isolevel; a user can define the upper and lower bounds of x, y, and z and the number of segments in each axis range.

The parametric generator's *func* is based on the three expressions: $x(u,v) \mid y(u,v) \mid z(u,v)$. It also has parameters to define the lower and upper bounds for the u, v domain and for the number of segments in both the u and v ranges.

Basic operators and mathematical functions are supported in the *func* parameters. *Figure 8* shows examples of each type of the three general shape generators.

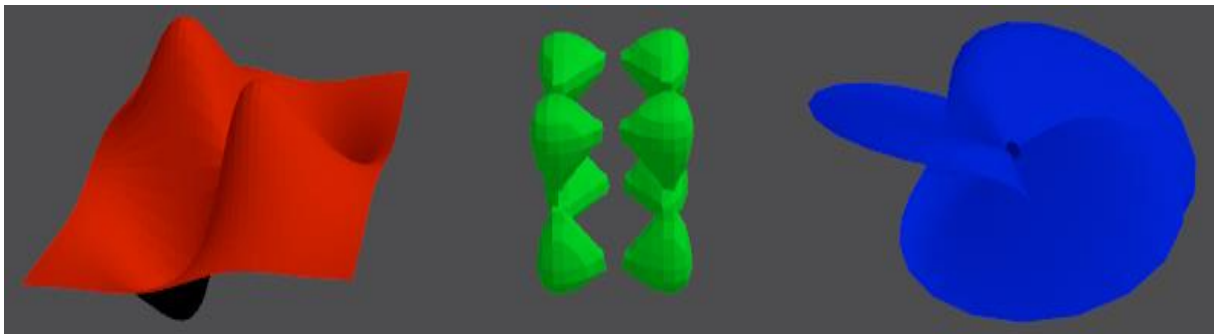


Figure 8: General shape generators: (a) Cartesian “humps”, (b) implicit “blobs”, (c) parametric “leaves”
(See Appendix A2)

5.5 Tori generators

Among the various shape generators we have introduced are the torus and torus knot generators.

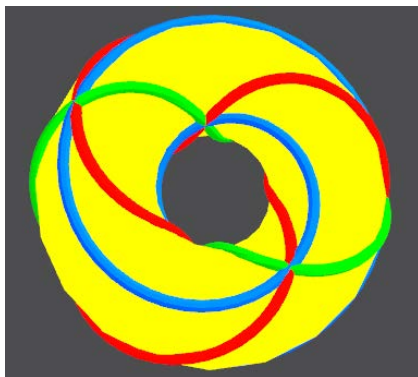
Torus: `torus id (rad_maj rad_min theta_max phi_min phi_max segs_theta segs_phi) endtorus`

Torus Knot: `torusknot id (symm turns rad_maj rad_min rad_tube segs_circ segs_sweep) endtorusknot`

The *rad_maj* and *rad_min* refer to the major and minor radii for both the torus and torus knot. The torus generator has the parameter *theta_max*, which is specified in degrees and determines how far the minor circle cross section is swept starting at the x-axis and circling the z-axis by the angle *theta* until the *theta_max* is reached. The torus generator also has a *phi_min* and *phi_max* which determine the starting and terminating angle in degrees around the minor circle. *segs_theta* and *segs_phi* correspond to the number of segments along the major radius and minor radius, respectively.

The torus knot generator has a *symm* parameter that determines the number of sweeps through the donut hole, which is equivalent to the rotational symmetry of the knot. *turns* indicates the number of turns around the donut hole. The *rad_tube* specifies the radius of the swept circle. If the *rad_tube* is equal to 0, the torus knot takes the form of a polyline, which can be used as a sweep path. *segs_circ* specifies the number of segments on the circular cross section and *segs_sweep* is the number of segments along the sweep path.

Behind the scenes, the parametric equations used to create a torus with handle radius a and large radius c are $x = (c + a \cos v) \cos u$, $y = (c + a \cos v) \sin u$, and $z = a \sin v$, where u and v are substituted with t , as t runs from 0 to *theta_max*. The torus knot has additional p and q parameters: $x = (c + a \cos(p*v)) \cos(q*u)$, $y = (c + a \cos(p*v)) \sin(q*u)$, and $z = a \sin(p*v)$. Figure 9 depicts three intersecting torus knots on a torus, creating the embedded graph K7.



```

surface Y color (1 1 0) endsurface      # Yellow
surface R color (1 0.1 0) endsurface    # Red
surface G color (0 0.9 0) endsurface    # Green
surface U color (0 0.6 1) endsurface    # Uniform

torus tor1 (2 1 360 0 360 20 25) endtorus
instance handlebody tor1 surface Y endinstance

torusknot path23 (2 3 2 1 0.1 8 140) endtorusknot
instance tk23 path23 surface R endinstance

torusknot path31 (3 1 2 1 0.1 8 140) endtorusknot
instance t31 path31 surface U endinstance

torusknot path12 (1 -2 2 1 0.1 8 140) endtorusknot
instance tk12 path12 surface G endinstance

```

Figure 9: Three different torus knots on a torus, creating the embedded graph K7.

6. Mesh Operations

Well-formed meshes are the starting point for operations such as subdivision (to form smoother surfaces) and offsetting (to thicken mathematical surfaces into slabs that can be fabricated on a 3D printer). Composite meshes that comprise assemblies of sub-meshes or of individual facets need to be merged first.

6.1 Merge

The merge operation combines overlapping meshes into a single mesh. It can be activated through a pull-down menu in the GUI as shown in *Figure 10*. Two meshes are considered overlapping if there exists at least one mesh vertex located within 0.001 units of the other mesh's vertices. Merge is useful for subdividing or offsetting overlapping shapes that are defined separately but should be combined and operated on together.

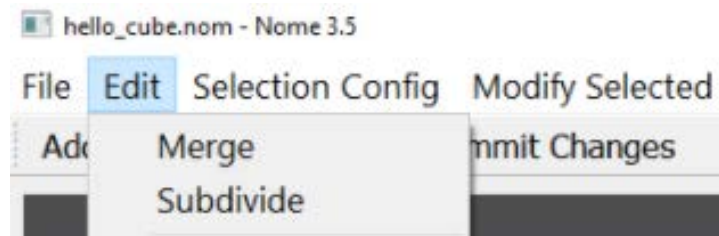


Figure 10: Merge and subdivide buttons in the GUI.

In the scene shown in *Figure 2*, the cube faces appear connected, but each face is its own separate mesh instance due to the usage of the group construct. If we were to subdivide this group of faces before merging, we would get six disjoint, rounded faces. To treat the six mesh instances as a single mesh instance, we need to click on the merge button. When we subsequently click on the subdivide button and specify the subdivision level in the pop-up window, the cube will get subdivided and output a spherical shape shown in *Figure 11*.

6.2 Subdivision

In addition to interactively subdividing shape in the user interface, smaller parts of the scene can also be subdivided procedurally in the *.jip* file by using the subdivision command. The subdivision command automatically merges all instances in its collection, so there is not a need for an explicit merge command.

Subdivision:

subdivision id

[sd_type sd_flag] sd_level

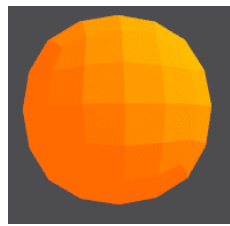
instance id1 object_id1 [xform & color] endinstance

...

instance idN object_idN [xform & color] endinstance

endsubdivision

The *sd_flag* denotes the type of subdivision algorithm used: *JIP_SD_CatmullClark* is the plain and simple Catmull-Clark subdivision and *JIP_SD_CC_sharp* is Catmull-Clark subdivision respecting “sharp” flags in the mesh. Sharp subdivision is an extension of Catmull-Clark subdivision and models sharp creases on a surface [9]. *sd_level* is the integer number of iterated subdivision steps, and *[xform & color]* are the optional parameters for instance transformations and coloring.



```
include hello_cube.nom endinclude  
  
subdivision subdiv_example  
  sd_type NOME_SD_CatmullClark  
  sd_level 2  
  instance cube0 cube endinstance  
endsubdivision
```

Figure 11: Subdivided cube resulting from the merged mesh shown in *Figure 1*.

For sharp subdivision, each edge in the underlying mesh data structure carries an integer tag denoting for how many more subdivision steps this edge should not be smoothed. This tag is decremented in each subdivision step. When the integer reaches zero, the edge receives no further special treatment and is smoothed with simple Catmull-Clark subdivision.

The preparatory process that constructs the above mesh data structure sets the appropriate flags for all edges. If an edge does not have a user-defined sharpness tag, then the tag is set to zero. Therefore, it is useful to have a *SD_type* flag that says whether we want to do sharp subdivision or just simple Catmull-Clark subdivision. In the latter case, all user-introduced sharpness tags are ignored, and the tags on all the winged-edge structures are set to zero. In a *.jip* file with several objects, which may be subdivided differently with different goals, the flag *JIP_SD_CatmullClark* can then save the effort of looking for sharpness tags; all edges are simply treated as if they had a tag “zero”.

There are two ways that a user can set some non-zero sharpness tags. In an interactive mode, the user selects some edges and sets their tags to the desired value in the range [1,9]. An alternative method is specifying sharpness information in the *.jip* file. For example, at the end of a mesh that contains some

sharp edges, we could add an explicit list of edges with their sharpness tags:

```
sharp flagValue vertex1 vertex2 endsharp
```

This can be specified for a whole chain of edges or for a closed loop:

```
sharp flagValue vertex1 vertex2 vertex3 ... vertexN endsharp
```

```
sharp flagValue vertex1 vertex2 vertex3 ... vertexN vertex1 endsharp
```

6.3 Offset

A surface mesh of zero thickness can be thickened into a thicker sheet surrounded by a water-tight B-rep with the following offset command.

Offset:

```
offset id
```

```
  [offset_type offset_flag] height width
```

```
  instance id1 object_id1 [xform & color] endinstance
```

```
  ...
```

```
  instance idN object_idN [xform & color] endinstance
```

```
endoffset
```

The *offset_flag* defines the type of thickened surface generated: *JIP_OFFSET_DEFAULT* offsets the starting mesh inward and outward symmetrically. Each facet in the original mesh maps to two facets in the offset mesh; the inner one with reversed orientation. Each boundary edge of an open polyhedron will map to a quadrilateral facet.

In default offset, every vertex is duplicated, and these vertices are shifted in opposite directions along an Averaged Vertex Normal (AVN). The AVN is found by averaging the face normals of all the faces that share that vertex as shown in *Figure 11*. The weight by which each face contributes to the AVN is the angle that the face has at the shared vertex. Vertices that only have small sliver angles count less in this averaging process.

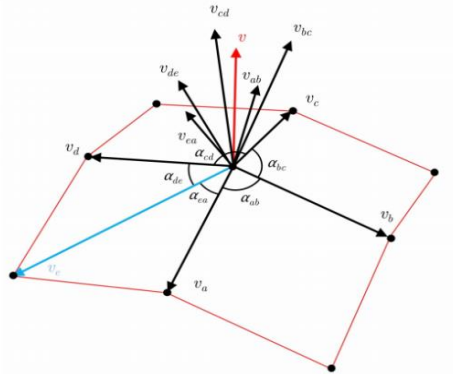


Figure 11: Vertex normal calculation using weighted face normals [4].

A gridded sheet of some specified thickness can be produced when the edges of a polyhedron are converted into prismatic structures. However, we are not actually thickening the edge themselves, but instead create first a thin gridded surface by cutting out the inner portions of all the facets and then applying a normal offset operation to thicken this gridded surface. The inner cutout in every face is defined to produce bands of exactly the specified *width*. This can be done by creating a new inner vertex at every face-vertex that lies on the angle bisector at that vertex and shifted by the desired width divided by the cosine of the half-angle. These new vertices are connected to form the inner cutout contours. All the open edges in this polyhedral surface are replaced with quadrilaterals that connect the inner and outer surfaces. [*xform & color*] are the optional parameters relating to transformations and coloring.

7. Rendering Control

JIPCAD has rendering control commands to specify lights and cameras. The light command can be used to create ambient or directional light sources. Ambient light is a colored light source that affects all surfaces regardless of orientation. A directional light is located at infinity and shines in the direction of the negative z-axis; it can be re-directed by applying a rotation to any instances of it.

Ambient or Directional light: *light id type [JIP_AMBIENT or JIP_DIRECTIONAL] color (R G B)*
endlight

The *camera* command creates a virtual camera object that can be positioned in the scene via an instance command.

Camera:

```
camera id  
  projection (JIP_PARALLEL or JIP_PERSPECTIVE)  
  frustum (min_triple) (max_triple)  
endcamera
```

The camera projection can be either JIP_PARALLEL or JIP_PERSPECTIVE. Parallel projection models a viewer that is infinitely far away from the scene; it is often useful to check the alignment of different geometrical elements in a complicated scene. Perspective projection models an ideal pinhole camera, mimicking the functionality of an eye located a finite distance from the point of interest.

The *frustum* is used for defining the geometry of the viewing volume. Specifically, the x and y components of both the *min_triple* and *max_triple* are used to define the rectangular window on the projection plane, and the z components define the near and far planes.

JIPCAD specifies a *window* into which all displays are mapped. *xmin* and *ymin* define the origin in the Normalized Device Coordinates (NDC). *xsize* and *ysize* determine the horizontal and vertical scale of the window.

```
Window: window Window (xmin ymin) (xsize ysize) background (R G B) endwindow
```

Inside the display window, users can define one or more rectangular *viewports*, which are specified as fractions of the display window. For instance, two separate, side-by-side viewports could be specified to generate a cross-eye stereo display of a 3D object. The extent of the viewport in the window is defined by the *xmin*, *ymin* and *xmax*, *ymax* parameters.

Viewport:

```
viewport id  
  (xmin ymin) (xmax ymax)  
  background (R G B)  
endviewport
```

All scene geometry and possibly some lights are gathered in a group called *world*. The render command is then used to specify a camera, perhaps some additional lights to further customize a specific viewport's illumination, and a viewport into which the results are projected. If we want three cameras in the world (perhaps to render "top", "front", and "side" views of an object), we simply use three render commands, each with its own camera instance and destination viewport properly placed in the display window.

```

world world_id
  instance id1 geometry ... endinstance
  instance id2 lightId_1 [xform] endinstance
  instance id3 lightId_2 [xform] endinstance
endworld

render
  instance id1 world_id [xform] endinstance
  instance id2 camera_id [xform] endinstance
  instance id3 lightId_1 [xform] endinstance
  instance id4 lightId_2 [xform] endinstance
  ...
  viewport id (xmin,ymin) (xmax,ymax) background (R G B) endviewport
endrender

```

7.1 Include Files

Sometimes users may want to reuse their personally developed rendering controls in different *.jip* files. To avoid lengthy monolithic file structures, users can import partial *.jip* files with an include command.

Include: *include file_name.jip endinclude*

This is useful for users to create their preferred *.jip* files that define surface colors, important geometry, coordinates axes, collections of camera, lights, and window/viewport specifications, etc., and then simply include these files in several of their active *.jip* files. *Figure 12* depicts a generic coordinate axis file that is often imported into active *.jip* files.

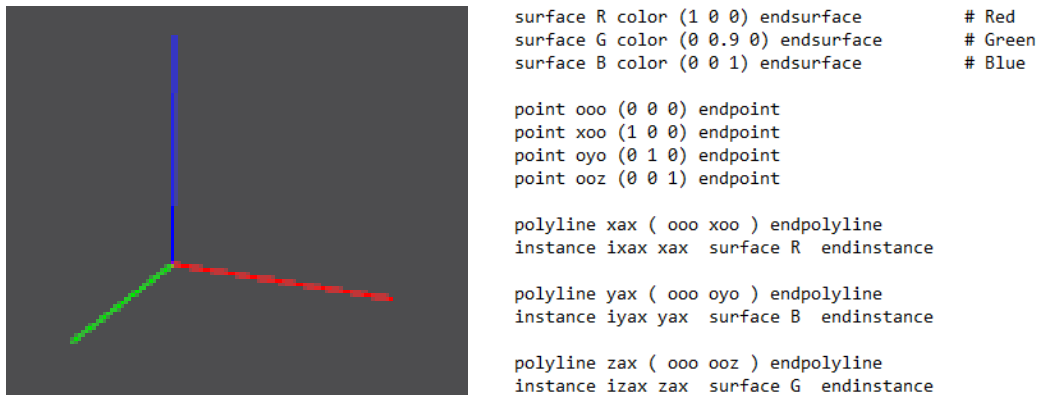


Figure 12: Generic coordinate axes at the world origin.

8. Graphical Editing

When JPCAD displays the contents of an active *.jip* file, interactive graphical editing features can be toggled on in the toolbar as shown in *Figure 13*. Toggling on these features allow the user to make changes interactively by primarily using the mouse cursor. The core graphical editing options are described below.

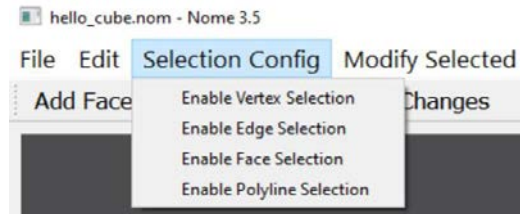


Figure 13: The drop-down selection menu.

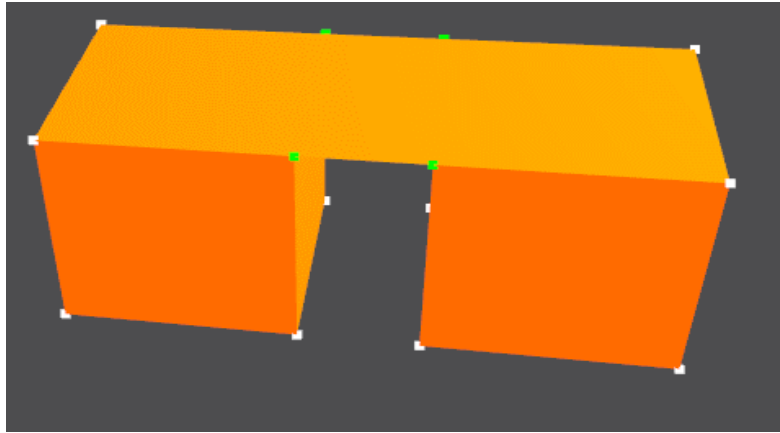
8.1 Interactive Addition of Faces and Polylines

Users can add faces and polylines interactively into the scene and save them back into a corresponding *.jip* file as reusable code, which will then display the same graphical state that was on the screen just before the save operation, so that additional graphical editing can continue.

Surface patches can be constructed by selecting existing vertices in the scene to form a clockwise polygonal outline of a desired face and clicking the “Add Face” button as shown in *Figure 13*. Similarly, users can add a polyline by selecting vertices and clicking on the “Add Polyline” button. Vertices are selected based on their closeness to the current cursor ray; they are then identified by their hierarchical name in the scene graph and entered correspondingly into the emerging JPCAD scene.

The added entities can be saved by first clicking on the “Commit Changes” button and then clicking on “Save” to add all committed entities into the chosen *.jip* file as regular JPCAD commands.

Corresponding instance commands are created along with the needed generator commands so that these entities can be properly rendered when reloading the file. Each newly added face and/or polyline is assigned a placeholder name based on the order in which they were added. An example of adding a face and saving the face as JPCAD code is shown in *Figure 14*. The code is inserted at the end of the *.jip* file. The user is encouraged after each such save operation to edit the saved *.jip* file to substitute their own preferred names for the new faces of polylines and to move these new entities into their most appropriate places in the hierarchically structured *.jip* file.



```

mesh TempMesh0
  face new0 ( .cube0.top.p1 .cube0.top.p2 .cube1.top.p3 .cube1.top.p4 ) endface
endmesh
instance instTempMesh0 TempMesh0 endinstance

```

Figure 14: Adding a face between two cubes and saving it as JIPCAD code.

Interactively adding faces into a design is useful because often only a small fraction of the surface needs to be constructed as a mesh. Multiple instances of those faces can then be grouped together with the proper symmetry operations to form the whole symmetrical surface. The resulting composite polyhedral surface can then be merged and smoothed with a subdivision operation.

8.2 Interactively Add Points

Users can also add new points into the scene while in the interactive graphical editing mode. From a user experience perspective, it would not make sense to manually specify x, y, z coordinates of the newly added point while in the interactive graphics mode, because the user could just type the point specification in the *.jip* file instead. What is needed are ways to add points graphically based on what can be seen on the screen, without explicit reference to any coordinate values.

After toggling on the “Add Point” option in the GUI, the user can first put the cursor in the desired location and simply click. This casts a physical line in space from your eye/camera to the cursor on which the point must lie. The user then rotates the scene, so that this temporary line can be seen from the side. Another cursor-click somewhere on that line then defines a 3D point location; the original cursor line can then be turned off. Since two lines in 3D space are unlikely to intersect each other directly, we look for the closest intersection point by finding the shortest line segment connecting the initial cursor ray and the second “select point” ray [10]. The shortest line segment between these lines is guaranteed to be perpendicular to both lines as shown in *Figure 15*.

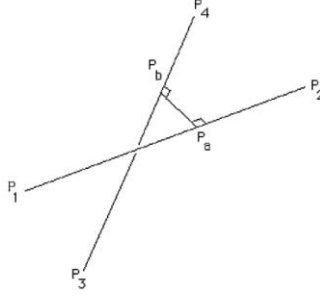


Figure 15: Shortest line segment connecting two skewed lines in 3D space.

Let the points P_a and P_b be written as points along the rays, represented as two finite line segments in *Figure 15*:

$$P_a = P_1 + mu_a(P_2 - P_1)$$

$$P_b = P_3 + mu_b(P_4 - P_3)$$

mu_a and mu_b range from negative to positive infinity and represent how far along the ray the intersection points are. We can then write the following equations using the perpendicular property of the shortest line segment.

$$(P_a - P_b) \cdot (P_2 - P_1) = 0$$

$$(P_a - P_b) \cdot (P_4 - P_3) = 0$$

Expanding the above equations using the line equations:

$$(P_1 - P_3 + mu_a(P_2 - P_1) - mu_b(P_4 - P_3)) \cdot (P_2 - P_1) = 0$$

$$(P_1 - P_3 + mu_a(P_2 - P_1) - mu_b(P_4 - P_3)) \cdot (P_4 - P_3) = 0$$

Expanding the equations in terms of the x, y, z coordinates results in the following:

$$d_{1321} + mu_a d_{2121} - mu_b d_{4321} = 0$$

$$d_{1343} + mu_a d_{4321} - mu_b d_{4343} = 0$$

where

$$d_{mnop} = (x_m - x_n)(x_o - x_p) + (y_m - y_n)(y_o - y_p) + (z_m - z_n)(z_o - z_p)$$

Lastly, we can solve for mu_a and mu_b , which we can then use to determine the coordinates of P_a and P_b :

$$mu_a = \frac{(d_{1343}d_{4321} - d_{1321}d_{4343})}{(d_{2121}d_{4343} - d_{4321}d_{4321})}$$

$$mu_b = \frac{(d_{1343} + mu_a d_{4321})}{(d_{4343})}$$

To make the point visible and to capture it in the JPCAD file, we simply click “Commit Changes” and “Save” to save the added point as a point command. Multiple added points can be saved at once.

If we add the point directly onto a mesh’s surface, our newly added point will be declared in the mesh construct as follows.

```
mesh TEMP
  point id (x y z) endpoint
endmesh
```

8.3 Vertex Selection and Movement

Once a vertex has been selected, the user is able to move it by clicking on the “Move Vertex” button. A right-mouse-down would “grab” the vertex and move it in a plane parallel to the window plane, following the cursor movement. Another mode “duplicates” a selected vertex and creates a new vertex that can then be moved to a new location.

8.4 Crystal Ball Interface

Selecting objects in the scene requires a user-friendly interface. To make navigating the scene easier, we implemented a crystal ball interface [5] that simulates a world in which a viewer is looking into a transparent crystal sphere from the outside-in. The crystal ball is centered around the world origin with a diameter of 90% of the height of the graphics window. The cursor controlled by the left mouse button is used to rotate this ball by changing the orientation part of the viewing matrix correspondingly.

In the general case, the 3D rotation axis always goes through the world origin as shown in *Figure 16*. We assume, the cursor is in contact with the crystal ball of radius R . Moving a surface point by distance D , rotates the ball through angle α . With the left mouse button down, a small incremental cursor movement in the display window is converted to a simple small rotation. If the cursor falls outside the projection of

the crystal ball, we perform a pure Z-rotation. If the cursor moves inside the projection of the crystal ball, we do a rotation around a properly slanted rotation axis.

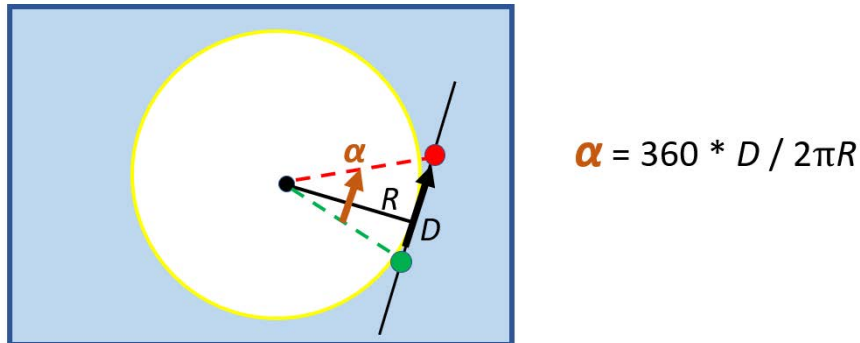


Figure 16: Scene rotation based on crystal ball interface.

The right mouse button is used for positioning as the cursor drags the object parallel to the x-y-plane. The left mouse button is used for adjusting the view orientation as the cursor rotates the world around its origin with the crystal ball GUI, and the mouse wheel is used for zooming in and out of the scene by changing the scale of the object. Moving the wheel upwards increases the scale and results in zooming into the scene.

As long as the cursor is active in the crystal ball display window, the scene geometry does not change. Thus, the hierarchically flat display-list remains unchanged and can be reused in subsequent frames.

9. Data Structures and Implementation Details

We considered using OpenMesh's half-edge data structure [11] for storing vertices, edges, faces, and connectivity information in a highly efficient manner. However, the half-edge data structure is most efficient when adjacent mesh entities (edges and faces) share the same orientation; it is less convenient to handle single-sided, non-orientable surfaces. It is technically possible to modify the half-edge data structure to handle non-orientable surfaces with a variety of complicated conditional statements and attributes, but we found this approach to be too unwieldy and convoluted. Therefore, we decided to use the winged edge data structure as shown in *Figure 17*, since it can represent single-sided surfaces with a few simple adjustments.

Each edge is assigned an edge type based on the orientation of its adjacent face(s). If the edge is shared by two faces with the same orientation, it is classified as a regular edge. If the two adjacent faces have opposite orientations, it is a Möbius edge. Lastly, if the edge is only used by one face, it is considered a

boundary edge. These edge types are useful in determining which face properties to use when performing mesh operations on a single-sided surface. For example, when JIPCAD is computing vertex normals via weighted face normals, JIPCAD iterates through the vertex’s adjacent faces and simply reverses the direction of the face normal when encountering a face located between two Möbius edges. Thus, by using the winged edge data structure, JIPCAD can apply mesh operations to non-orientable surfaces.

The first iteration of the NOME prototype in 2018 had also used the winged edge data structure [4]. In continuing with this data structure, we have made its implementation more robust as well as compatible with our new mesh operations (e.g., sharp subdivision) and interactive graphical editing capabilities.

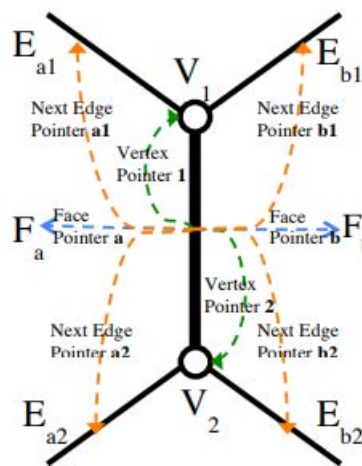


Figure 17: Winged edge data structure [13].

9.1 Implementation Details

JIPCAD is a Qt-based application [12] that uses OpenGL [13] and Qt 3D modules to easily build and interact with 3D generated scenes. ANTLR [14] is used to walk parse trees constructed from a custom .g4 grammar file designed for the JIPCAD language. We used Git [15] for version control and AppVeyor [16] as a continuous integration service for automated builds; these tools were important in ensuring our JIPCAD development efforts were scalable and well synchronized.

10. JIPCAD Design Examples

“3-2-1” is a sculpture created by Tord Tengstrand (Fig.17a), showcased in the Bridges 2020 Art Exhibition [6]. The sculpture has 3 edges, 2 vertices, and a single face. It is of genus-2 and has a 3-fold rotational symmetry around the long axis. Given its high degree of symmetry, JIPCAD is the ideal tool to construct a complete model of this sculpture. As a start, we can create a JIPCAD file that captures the

three edge curves that emerge from the vertex. Then, the first manual task is to add some facets between a pair of these curves. Three copies of this, rotated around the z-axis in steps of 120° leads to *Figure 18b*, and flipping this trio upside down produces *Figure 18c*; with overall geometry properly captured in a crude polyhedral shape, we can now use subdivision to produce a smooth surface. To keep the edges sharp and clean we label those edges as “sharp” before we apply the sharp subdivision process. The final result is shown in *Figure 18d*.

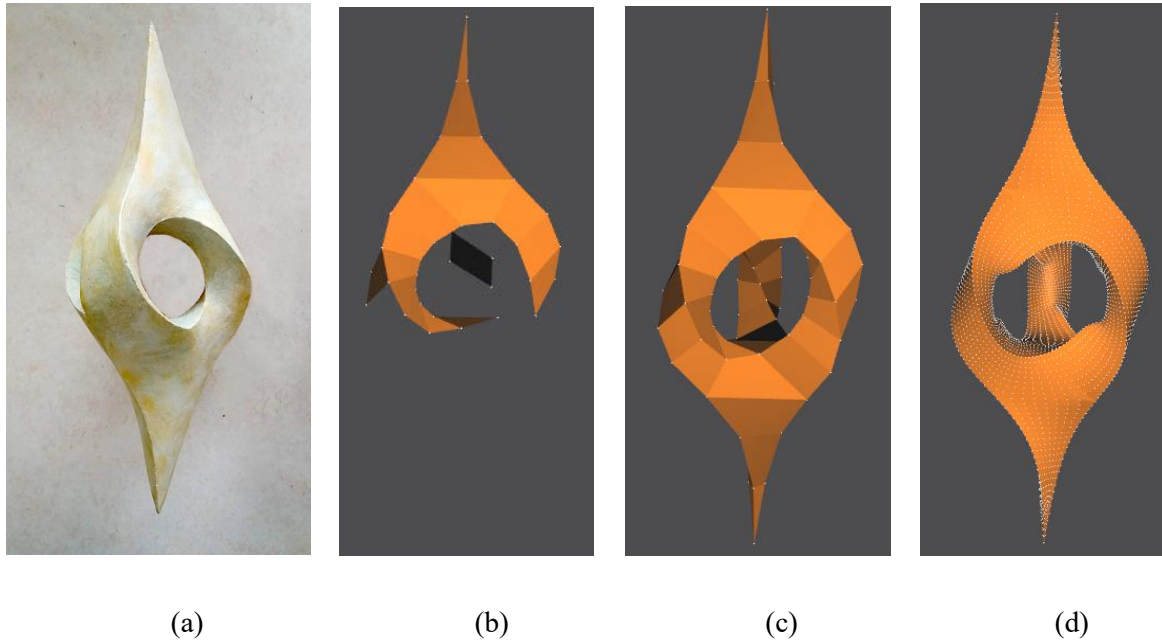


Figure 18: The “3-2-1” Sculpture: (a) Sculpture by Tord Tengstrand; (b) intermediate construction step; (c) completed construction; (d) sharp subdivision applied. (see Appendix A3)

Figure 19 depicts a (3, 4) torus knot sculpture created by Carlo H. Séquin. The sculpture was constructed using JIPCAD sweeps with B-spline paths.



Figure 19: (3, 4) Torus knot via sweeps. (see Appendix A4)

11. Interactive Workflow

The typical JIPCAD workflow begins with the user creating a procedural scene description in the form of a *.jip* file. Users may have the Language Reference [7] open to assist with the language syntax. Upon opening the *.jip* file in JIPCAD.exe, a text window will notify the user if there were any syntactic or semantic errors in the input file. If there were no errors, the scene will display on the graphics screen, and the user can then navigate the scene through the crystal ball interface and modify the scene geometry with the cursor.

A commonly used interactive modeling operation is adding faces between curves or other previously defined vertices. Adding faces is enabled by toggling on “Enable Vertex Selection” as shown in *Figure 13*, selecting the target vertices, and clicking on the “Add Face” button. Users may add individual faces in small batches and save those faces back into the file. The saving mechanism is activated by clicking on the “Commit Changes” button in the GUI as shown in *Figure 10* and then typing “Ctrl+S” or clicking on “Save”.

The JIPCAD code produced for the newly added faces is inserted at the bottom of the *.jip* file. For added faces, the code consists of a mesh command containing the various faces along with an instance command to ensure the faces get re-rendered upon reloading the file, as shown in *Figure 14*. Note the added mesh and face(s) are named with a “Temp” prefix. This is intended to be a placeholder name, and the user can replace it with a more descriptive name later. The faces’ point names are obtained and saved based on their location within the scene graph hierarchy and adhere to JIPCAD’s point naming convention as described in the *Hierarchical Scene Graph* section.

The user may move the new face definitions into a larger, pre-existing mesh construct that contains a more complete version of the unique surface swath that the user aims to construct. This whole surface swath may be instantiated more than once if the envisioned shape has some symmetry. By placing the new faces into the proper hierarchical context, the user can verify that the latest batch indeed fulfills their intended role in the overall CAD model. The user can then return to the graphics display to add some more faces or to eliminate faces that may overlap inadvertently. This iterative process is illustrated in *Figure 18*, where the intermediate construction step depicts several faces connected to the B-spline edge curves.

Adding polylines is similar to the process of adding faces. The key difference is added polylines are saved as individual polyline commands rather than grouped together in a hierarchical mesh construct. Polylines, individual edges, or vertices can be marked as “sharp”, so that surfaces with some sharp creases can be

designed as described in the *Subdivision* section. This can be done through the graphical interface or with appropriate textual annotations in the *.jip* file.

During all these design phases, the overall geometry of the emerging shape can be fine-tuned by adjusting some slider values that might move some vertices, or which stretch or warp a whole section of the overall sculpture. With any “save” operation the latest values of all the sliders will get entered into the *.jip* file as the default settings in the corresponding parameter banks.

Typically, these manually constructed surfaces are coarse polyhedral forms. Optionally, such surface can be subjected to one or more subdivision steps to smooth out the dihedral edges between adjacent faces. Users can activate the subdivision process through the GUI’s “Merge” and “Subdivide” buttons as shown in *Figure 10*. Alternatively, subdivision of specific parts of the overall scene can be specified textually in the JPCAD file with the subdivision command.

Similarly, mathematically thin 2-manifolds can be thickened with the offset operation. This operation can also be launched through the GUI, or it can be specified in the JPCAD file itself.

A final step in the design process may be to orient the sculpture into a more informative or aesthetically satisfying orientation and to adjust the lights that illuminate the whole scene – and then to save a snapshot of the display.

12. Conclusion

We have developed a dual-mode design environment for geometrical shapes by extending and robustifying the previous NOME system [4]. One part is a procedural shape description language that makes it easy to define precise geometry as well as combining geometrical elements in a hierarchical manner. The other part is an interactive graphical display that allows the user to enter geometrical modification based on cursor controlled selections and movements. The key achievement is to commit and save any such changes with suitable annotations to the original *.jip* file, so that a reopening of the enhanced file will reproduce the latest graphical state and thus allow the user to continue in either a graphical manner or by textual changes in the *.jip* file.

This system has allowed users with limited computer graphics experience to make highly complex, precise geometrical models.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Carlo H. Séquin. His support and mentorship over the past few years have helped me grow into a well-rounded engineer. His passion for 3D modeling is contagious and has made me appreciate the beautiful connection between art and mathematics. Most of all, I would like to thank him for the life values that he has instilled in me through our wonderful discussions - among many are pursuing things in life that interest me and leading teams skillfully.

I would also like to thank Professor Ren Ng for being an amazing role model and helping me reach my potential. He has supported me both academically and professionally on many occasions and has been pivotal in my personal development. His courses inspired me to delve deeper into the field of computer graphics, which ultimately led me to work on the JIPCAD project.

I am also very grateful for all the URAP members who participated in the JIPCAD project in 2020-2021. In alphabetical order: Brandon Lee, Brian Kim, Monica Tang, Rohan Sood, Toby Chen, Vishnu Pamula, Xinyu Zhang, Zachary Yin. These are exceptional students and their assistance with the development and testing of JIPCAD was crucial.

References

- [1] OpenSCAD Documentation. - <https://openscad.org/documentation.html>
- [2] Python Blender Documentation. - <https://docs.blender.org/api/current/>
- [3] J. Smith. SLIDE design environment. 2003. - <http://www.cs.berkeley.edu/~ug/slide/>
- [4] G. Dieppedalle. *Interactive CAD Software for the Design of 2-manifold Free-form Surfaces*. 2018. - <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-48.pdf>
- [5] C. Sequin. Crystal Ball Interface. 2004. - <https://people.eecs.berkeley.edu/~sequin/CS184/LECT04/L13.htm>
- [6] T. Tengstrand. *3-2-1 Sculpture*. 2020. - <http://gallery.bridgesmathart.org/exhibitions/2020-bridges-conference/tordtengstrandtelia.com>
- [7] JIPCAD Language Reference. - <https://tinyurl.com/9ds62ee2>
- [8] General Shape Generator Description. - <https://brandonyli.github.io/>
- [9] T. DeRose, M. Kass, and T. Truong. *Subdivision Surfaces in Character Animation*. 1998. - <https://graphics.pixar.com/library/Geri/paper.pdf>
- [10] P. Bourke. *Points, Lines, and Planes*. 1998. - <http://paulbourke.net/geometry/pointlineplane/>
- [11] OpenMesh Documentation. - https://www.graphics.rwth-aachen.de/media/openmesh_static/Documentations/OpenMesh-5.2-Documentation/a00016.html
- [12] Qt, *Qt Homepage*. - <https://www.qt.io/>
- [13] OpenGL, *OpenGL Homepage*. - <https://www.opengl.org>
- [14] ANTLR, *ANTLR Homepage*. - <https://www.antlr.org/>
- [15] Github, *Github Homepage*. - <https://github.com/>
- [16] AppVeyor, *AppVeyor Homepage*. - <https://www.appveyor.com/>
- [17] Y. Wang. *Robust Geometry Kernel and UI for Handling Non-orientable 2-Manifolds*. 2016. - <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-65.html>

Appendix: Some Example JIPCAD Files

This appendix contains the complete JIPCAD files that produce some of the figures shown earlier in the main text. It shows the use of some of the special generators, of hierarchical constructs, and of general mesh-refinement processes.

A1: Sweep with Profile Inversion (Figure 7)

```
# Sliders
bank p
  set radius 1.2 1 5 0.1
  set twist 0 0.1 360 1
  set azimuth 0 0.1 360 1
endbank

#### Cross Section
point cp0 ({expr $p.radius * cos(0)} {expr $p.radius * sin(0)} 0) endpoint
point cp1 ({expr $p.radius * cos(0.0174533 * 45)} {expr $p.radius * sin(0.0174533 * 45)} 0) endpoint
point cp2 ({expr $p.radius * cos(0.0174533 * 90)} {expr $p.radius * sin(0.0174533 * 90)} 0) endpoint
point cp3 ({expr $p.radius * cos(0.0174533 * 135)} {expr $p.radius * sin(0.0174533 * 135)} 0) endpoint
point cp4 ({expr $p.radius * cos(0.0174533 * 180)} {expr $p.radius * sin(0.0174533 * 180)} 0) endpoint
point cp5 ({expr $p.radius * cos(0.0174533 * 225)} {expr $p.radius * sin(0.0174533 * 225)} 0) endpoint
point cp6 ({expr $p.radius * cos(0.0174533 * 270)} {expr $p.radius * sin(0.0174533 * 270)} 0) endpoint
point cp7 ({expr $p.radius * cos(0.0174533 * 315)} {expr $p.radius * sin(0.0174533 * 315)} 0) endpoint

polyline profile (cp7 cp6 cp5 cp4 cp3 cp2 cp1 cp0) closed endpolyline

#### Sweep Path
point point0 (-5.5 4 0) endpoint
point point1 (-5 4 0) endpoint
point point2 (5 4 0) endpoint
point point3 (5.5 4 0) endpoint
point point4 (5 4 0) endpoint
point point5 (-10 4 0) endpoint
point point6 (-10 -6 0) endpoint
point point7 (10 -6 0) endpoint
point point8 (10 4 0) endpoint
point point9 (-5 4 0) endpoint

#### Control Points
controlpoint sc0 point point0 scale (1.5 1.5 0) rotate(0 0 0) endcontrolpoint
controlpoint sc1 point point1 scale (1.1 1.1 0) rotate(0 0 0) endcontrolpoint
controlpoint sc2 point point2 scale (1.1 1.1 0) rotate(0 0 0) endcontrolpoint
controlpoint sc3 point point3 scale (0.8 0.8 0) rotate(0 0 0) endcontrolpoint
controlpoint sc4 point point4 scale (0.5 0.5 0) rotate(0 0 0) startreverse endcontrolpoint
controlpoint sc5 point point5 scale (0.6 0.6 0) rotate(0 0 0) endcontrolpoint
controlpoint sc6 point point6 scale (1.0 1.0 0) rotate(0 0 0) endcontrolpoint
controlpoint sc7 point point7 scale (1.4 1.4 0) rotate(0 0 0) endcontrolpoint
controlpoint sc8 point point8 scale (1.8 1.8 0) rotate(0 0 0) endcontrolpoint
controlpoint sc9 point point9 scale (1.9 1.9 0) rotate(0 0 0) endreverse endcontrolpoint
```

polyline sweeppath (sc0 sc1 sc2 sc3 sc4 sc5 sc6 sc7 sc8 sc9) closed endpolyline

```
### Sweep
sweep s0
  crosssection profile endcrosssection
  path sweeppath mintorsion azimuth {expr $p.azimuth} twist {expr $p.twist} endpath
endsweep
```

instance sweepinst s0 endinstance

A2: Examples of General Shape Generators (Figure 8)

```
gencartesiassurf humps (7*x*y)/exp((x^2)+(y^2)) (-2 2 -2 2 40 40) endgencartesiassurf
surface R color (0.7 0.2 0 ) endsurface # Red
instance ihump humps surface R endinstance
```

```
genimplicitsurf blobs (x^4+y^4+z^4-x^2-y^2-z^2+0.5) (-2 2 -2 2 -2 2 30) endgenimplicitsurf
surface G color (0 0.7 0.2) endsurface # Green
instance iblob blobs translate (0 5 0) surface G endinstance
```

```
genparametricsurf leaves ( sin(v)*(2+cos(3*u))*cos(2*u) | sin(v)*(2+cos(3*u))*sin(2*u)
| sin(v)*cos(v)*(sin(3*u)) (0 6.28318 0 1.0472 60 20) endgenparametricsurf
surface B color (0 0.2 0.7) endsurface # Blue
instance ileaf leaves rotate (0 1 0)(180) translate (0 10 0) surface B endinstance
```

A3: 3-2-1 Sculpture Using Sharp Subdivision (Figure 18)

```
## Tord_SD.jip
##
## Reconstructing the Tord_sculpture from his original 3-2-1 Sculpture at Bridges2020.
## A different way of structuring the facets into 6 coherent surface pieces.
## Consolidated with added sharpness specifications.
##
## CHS 2020/12/4
```

Some Surface colors

```
surface M color (0.9 0 1 ) endsurface # Magenta
surface Z color (1 0 0.6) endsurface # Zinnober
surface R color (1 0.1 0 ) endsurface # Red
surface O color (1 0.6 0 ) endsurface # Orange
surface Y color (1 1 0 ) endsurface # Yellow
surface L color (0.5 1 0 ) endsurface # Lime
surface G color (0 0.9 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 0.6 1 ) endsurface # Uniform
surface B color (0 0.3 1 ) endsurface # Blue
surface V color (0.3 0 1 ) endsurface # Violet
```

```

surface P color (0.6 0 1 ) endsurface #Purple
surface W color (1 1 1 ) endsurface #White
surface S color (0.7 0.7 0.7) endsurface #Snow
surface D color (0.4 0.4 0.4) endsurface #Dark
surface K color (0 0 0 ) endsurface #Black

```

```
##### coordinate system #####
```

```

point ooo (0 0 0) endpoint
point xoo (1 0 0) endpoint
point oyo (0 1 0) endpoint
point ooz (0 0 1) endpoint

```

```

polyline xax ( ooo xoo ) endpolyline
polyline yax ( ooo oyo ) endpolyline
polyline zax ( ooo ooz ) endpolyline

```

```

group coord_axes
instance ixax xax surface R endinstance
instance iyax yax surface B endinstance
instance izax zax surface G endinstance
endgroup

```

```
##### Some test B-spline #####
```

```

bank p
set mrd 0.8 0 2 0.01
set msl 0.2 0 2 0.01
set wdx -0.5 -1 2 0.1
set wdz 1.5 0 3 0.1
set wex -0.4 -1 2 0.1
set wey 0.0 -1 2 0.1
set wez 1.5 0 3 0.1
set wfx -0.2 -2 2 0.1
set wfy -1.2 -2 2 0.1
set w fz 0.8 -2 2 0.1
set yrot 0 -180 180 1
set slicesN 9 0 20 1
endbank

```

```

point eaR (0.3 6 0) endpoint
point eb ( 0 4 0) endpoint
point ecR (-0.3 2 0) endpoint

```

```

point wd ( {expr $p.wdx} 1 {expr $p.wdz} ) endpoint
point we ( {expr $p.wex} {expr $p.wey} {expr $p.wez} ) endpoint
point wf ( {expr $p.wfx} {expr $p.wfy} {expr $p.w fz} ) endpoint

```

```

point wg ( {expr $p.mrd} -0.5 {expr $p.msl} ) endpoint
point wh ( {expr $p.mrd} 0 0 ) endpoint
point wi ( {expr $p.mrd} 0.5 {expr -$p.msl} ) endpoint

```

```

#####
#
## These are the 3 edges connecting the two vertices in Trord Tengstrand's sculpture.
## Some segments have been selectively de-activated to reduce clutter when adding faces between the
curves.
## It is only necessary to create 1/6 of the hole surface; the rest is generated by applying D3 symmetry.

bspline bs order 4 ( eaR eb ecR wd we wf wg wh wi ) segs 12 endbspline

instance bsR bs surface R rotate(0 1 0){(expr $p.yrot)} endinstance
instance bsRu bs surface O rotate(1 0 0)(180) rotate(0 1 0){(expr $p.yrot)} endinstance

instance bsG bs surface G rotate(0 1 0){(expr $p.yrot+120)} endinstance
instance bsGu bs surface L rotate(1 0 0)(180) rotate(0 1 0){(expr $p.yrot+120)} endinstance

instance bsB bs surface B rotate(0 1 0){(expr $p.yrot-120)} endinstance
instance bsBu bs surface U rotate(1 0 0)(180) rotate(0 1 0){(expr $p.yrot-120)} endinstance

#####
## >>> The task now is to manually add quads and triangles between adjacent edge curves and
combine them into a mesh.
## Six copies of that mesh will then make the complete sculpture surface.

## interactively added faces, cleaned up, combined into one mesh:
mesh surf
face fa ( .bsR.v0 .bsR.v1 .bsG.v1 ) endface
face fb ( .bsR.v1 .bsR.v2 .bsG.v2 .bsG.v1 ) endface
face fc ( .bsR.v2 .bsR.v3 .bsG.v3 .bsG.v2 ) endface
face fd ( .bsR.v3 .bsBu.v9 .bsBu.v8 .bsG.v3 ) endface
face fe ( .bsR.v3 .bsR.v4 .bsBu.v10 .bsBu.v9 ) endface
face ff ( .bsR.v4 .bsR.v5 .bsBu.v11 .bsBu.v10 ) endface
face fg ( .bsR.v5 .bsR.v6 .bsB.v12 .bsBu.v11 ) endface
face fh ( .bsR.v6 .bsR.v7 .bsB.v11 .bsBu.v12 ) endface
face fi ( .bsR.v7 .bsR.v8 .bsB.v10 .bsB.v11 ) endface
face fj ( .bsR.v8 .bsR.v9 .bsB.v9 .bsB.v10 ) endface
face fm ( .bsBu.v8 .bsBu.v7 .bsG.v4 .bsG.v3 ) endface
face fn ( .bsBu.v7 .bsBu.v6 .bsG.v5 .bsG.v4 ) endface

sharp 9 ( .bsR.v0 .bsR.v1 .bsR.v2 .bsR.v3 .bsR.v4 .bsR.v5 .bsR.v6 .bsR.v7 .bsR.v8 .bsR.v9 ) endsharp
##<<< sEa
sharp 9 ( .bsG.v0 .bsG.v1 .bsG.v2 .bsG.v3 .bsG.v4 .bsG.v5 ) endsharp ##<<< sEb
sharp 9 ( .bsB.v9 .bsB.v10 .bsB.v11 .bsB.v12 ) endsharp ##<<< sEc
sharp 9 ( .bsBu.v6 .bsBu.v7 .bsBu.v8 .bsBu.v9 .bsBu.v10 .bsBu.v11 .bsBu.v12 ) endsharp ##<<< sEd

endmesh

mesh rimfill
face fp ( .bsBu.v6 .bsBu.v5 .bsG.v6 .bsG.v5 ) endface
sharp 9 ( .bsBu.v6 .bsBu.v5 ) ( .bsG.v6 .bsG.v5 ) endsharp

```

endmesh

mesh polecaps

face fq (.bsBu.v9 .bsGu.v9 .bsRu.v9) endface

face fr (.bsG.v9 .bsB.v9 .bsR.v9) endface

endmesh

inner polar caps:

instance ipolecaps polecaps surface W endinstance

missing patches in the 3 arms:

instance i0fp rimfill surface S endinstance

instance i1fp rimfill surface S rotate(0 1 0)(120) endinstance

instance i2fp rimfill surface S rotate(0 1 0)(240) endinstance

instance surf1 surf surface Y endinstance

instance surf2 surf surface O rotate(0 1 0)(120) endinstance

instance surf3 surf surface C rotate(0 1 0)(240) endinstance

instance surf4 surf surface G rotate(1 0 0)(180) endinstance

instance surf5 surf surface R rotate(1 0 0)(180) rotate(0 1 0)(120) endinstance

instance surf6 surf surface B rotate(1 0 0)(180) rotate(0 1 0)(240) endinstance

END

A4: (3, 4) Torus knot (Figure 19)

TK_3-4_simple.jip

a C-shaped cross section swept along a "triangular" torus-knot path

#

CHS 2021/04/26

#####

surface O color (1 0.5 0.2) endsurface

Define the C-shaped profile

point pa (0.010168 -0.522839 0) endpoint

point pb (0.149077 -0.435194 0) endpoint

point pc (0.257372 -0.311707 0) endpoint

point pd (0.326136 -0.162547 0) endpoint

point pe (0.349704 0.000000 0) endpoint

point pf (0.326136 0.162547 0) endpoint

point pg (0.257372 0.311707 0) endpoint

point ph (0.149077 0.435194 0) endpoint

point pi (0.010168 0.522839 0) endpoint

point pj (-0.021651 0.512500 0) endpoint

point pk (-0.014695 0.479775 0) endpoint

point pl (0.087675 0.385145 0) endpoint

point pm (0.165126 0.269232 0) endpoint

```

point pn ( 0.213377 0.138441 0) endpoint
point po ( 0.229763 0.000000 0) endpoint
point pp ( 0.213377 -0.138441 0) endpoint
point pq ( 0.165126 -0.269232 0) endpoint
point pr ( 0.087675 -0.385145 0) endpoint
point ps (-0.014695 -0.479775 0) endpoint
point pt (-0.021651 -0.512500 0) endpoint

```

```
polyline pProfile ( pa pb pc pd pe pf pg ph pi pj pk pl pm pn po pp pq pr ps pt) closed endpolyline
```

```
##### The sweep path #####
```

```

point C ( {expr 3.0*cos(-100*0.01745)} 3.0 {expr 3.0*sin(-100*0.01745)} ) endpoint # -60
point D ( {expr 7.0*cos( 0*0.01745)} 0.0 {expr 7.0*sin( 0*0.01745)} ) endpoint # 0
point C2 ( {expr 3.0*cos( 100*0.01745)} -3.0 {expr 3.0*sin( 100*0.01745)} ) endpoint # 60
point B2 ( {expr 2.0*cos(-200*0.01745)} -1.4 {expr 2.0*sin(-200*0.01745)} ) endpoint # -180
point A2 ( {expr 1.5*cos(-120*0.01745)} 0.0 {expr 1.5*sin(-120*0.01745)} ) endpoint # -120
point B3 ( {expr 2.0*cos(-40*0.01745)} 1.4 {expr 2.0*sin(-40*0.01745)} ) endpoint # -60
point C3 ( {expr 3.0*cos(-340*0.01745)} 3.0 {expr 3.0*sin(-340*0.01745)} ) endpoint # -300
point D3 ( {expr 7.0*cos(-240*0.01745)} 0.0 {expr 7.0*sin(-240*0.01745)} ) endpoint # -240
point C4 ( {expr 3.0*cos(-140*0.01745)} -3.0 {expr 3.0*sin(-140*0.01745)} ) endpoint # -180

```

```

controlpoint cpC point C scale(0.2 0.2 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpD point D scale(3.0 3.0 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpC2 point C2 scale(0.2 0.2 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpB2 point B2 scale(1.4 1.4 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpA2 point A2 scale(1.0 1.0 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpB3 point B3 scale(1.4 1.4 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpC3 point C3 scale(0.2 0.2 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpD3 point D3 scale(3.0 3.0 1.0) rotate(0 0 0) endcontrolpoint
controlpoint cpC4 point C4 scale(0.2 0.2 1.0) rotate(0 0 0) endcontrolpoint

```

```

bspline sweeppath
order 4
(cpC cpD cpC2 cpB2 cpA2 cpB3 cpC3 cpD3 cpC4)
segs 90
endbspline

```

```
##### DEFINING THE SWEEPED RIBBON #####
```

```

#crosssection crossX
# type polyline pProfile
#endcrosssection
bank p
set s_twist 55.10 0.1 360 1
set s_azimuth 0 0.1 360 1
endbank

```

```

sweep ribbon
crosssection pProfile endcrosssection
path sweeppath mintorsion azimuth {expr $p.s_azimuth} twist {expr $p.s_twist} endpath

```


endsweep

ASSEMBLY

group knot

instance r0 ribbon rotate(0 1 0) (120) endinstance

instance r1 ribbon endinstance

instance r2 ribbon rotate(0 1 0) (-120) endinstance

endgroup

group assembly

instance k0 knot surface O rotate(1 0 0) (90) rotate(0 0 1) (-90) endinstance

endgroup

RENDERING

instance a0 assembly scale (0.4 0.4 0.4) endinstance

#####