Interactive Procedural CAD Tool

Randy Fan
EECS Department
UC Berkeley
randyfan@berkeley.edu

ABSTRACT

The procedural CAD tool developed in this project can be used to construct complex geometries via scripts, interactively modify the rendered geometries in the graphical user interface (GUI), and save key modifications back into the file as reusable code. After conducting a need-finding study, it became clear that this tool may be beneficial to graphics programmers and, more generally, people who are interested in 3D modelling but would prefer to stay in a programing environment. There is an opportunity to build a CAD tool that is both procedural and interactive; the prototype tool developed in this project attempts to bridge these two modelling approaches by implementing a new CAD language and an innovative GUI that can handle interactive changes.

Author Keywords

CAD; Graphics; AST; OpenSCAD; Interactive GUI; Script; ANTLR; Scene Graph; Scene Tree; Procedural Tool

MOTIVATION

There are many existing CAD tools out there in the market, such as Blender, OpenSCAD, Maya, and SolidWorks; however, these tools do not strike a good balance between procedural shape creation and interactive GUI editing capabilities. For example, Maya and Blender rely heavily on a click and drag graphical user interface, which is imprecise compared to a definitive modeling method. Requiring a modeler to move objects and vertices around on the screen often makes it difficult to create geometric free-form shapes.

OpenSCAD, an open source script-based 3D modelling tool, allows users to generate shapes and define relationships among various shapes (e.g. how they are modified via intersection, difference, and envelope combination functions). However, OpenSCAD has several limitations that make the tool insufficient for complex artistic tasks; for example, the tool has a limited set of 2D and 3D shape generators, containing only basic primitives such as a circle, square, cube, and cylinder generators [1]. This limitation makes it difficult for users to construct more complex shapes such as a torus knot or a snow crystal, both shapes which could have been easily created if there were pre-existing shape generators of their kind or a language construct that saves symmetrical components.

OpenSCAD's GUI is inflexible for users who want to modify shapes interactively using the mouse cursor. In fact, the mouse cursor can only be used for navigating the scene in OpenSCAD. This is problematic for non-programmer designers who want to customize and configure their models after deployment. Another potential interactive flaw of OpenSCAD is its variables are kept constant during the entire life cycle, which can be confusing for the user if they want to increment variables [2].

Blender, a free and open-source 3D modelling tool used in many animated films, has recently added in Python scripting as an option to automate certain design tasks, but the scripting does not preserve the scene hierarchy when the objects are rendered. 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.

A programmable CAD tool allows for precise placement of shapes in the scene and easily modifiable objects with the simple change of defined parameter value. For example, if a user wants to adjust the size or number of wheels on a truck model, these two tasks could be as trivial as changing a parameter's value when using a programmable CAD tool. In programmable CAD, the code itself is text-readable and can be reused easily by other designers.

Despite these strengths, creating organic-looking, subdivided shapes is difficult with OpenSCAD and other programmable CAD tools in general because these shapes are difficult to create by just piecing together simple, rough-edged primitive objects via code. Thus, the constructive solid geometry (CSG) engine OpenSCAD is built upon is not sufficient for all artistic tasks, and there is opportunity for designing a programmable CAD tool that incorporates both programmatic and interactive GUI capabilities – a tool that allows designers to create accurate 3D models while still being able to modify it freely in the GUI. This is the idea my tool attempts to implement.

A successful outcome would be a fully working shape description language that allows users to create advanced mathematical shapes via scripts, combined with a flexible interface that gives users to ability to modify shapes in the GUI like one could do using more traditional 3D modelling tools such as Maya. To differentiate this project further, the prototype tool should allow users to commit and save their GUI modifications back into the code file, so they can avoid redoing manual changes each time the scene is loaded. The overarching goal of this project is to design a tool to help artists construct mathematically complex sculptures with highly symmetrical components more easily.

RELATED WORK

OpenSCAD is a widely used scripting tool for parametric CAD modelling and is most similar to this project's prototype tool. Despite the similarities, OpenSCAD attempts to solve a slightly different problem compared to this project's topic; instead of focusing on finding a balance between procedural CAD and interactive modeling, OpenSCAD focuses more on the CAD aspects of creating machine parts [3]. Thus, OpenSCAD is primarily useful as a scientific tool for education and research, but not so much for 3D artistic design, which is what my tool attempts to assist with. OpenSCAD also lacks many features such as a limited selection of advanced shape generators and an interface that does not allow shapes to be interactively modified by a mouse [1].

OpenSCAD's solution to procedural shape generation has several features worth emulating, many of which I have adopted or plan to add into my tool. For example, its syntax is familiar to most programmers, with variables created by a statement with an identifier and assignment with an expression and semicolon to denote the end. It also has a familiar "include filename" construct that allows users to import code from external files, and a variety of interpretable mathematical expressions and iterators. There are a set of special variables that can help control object rendering, such as the $t time variable, which is useful for rendering simple animations.

OpenSCAD also has for loops and standard if statements, with a key difference being their for loops do not iterate using an incremented variable, but rather can only directly iterate through the elements of the vector [2]. This is different from most programming languages which give users both iteration options. OpenSCAD's if statements work similar to most programming languages, with the first return value being the output for the true condition and the second being for the else or false condition. OpenSCAD uses Constructive Solid Geometry (CSG), which allows a modeler to use various Boolean operators to merge together shapes and design advanced mechanical parts.

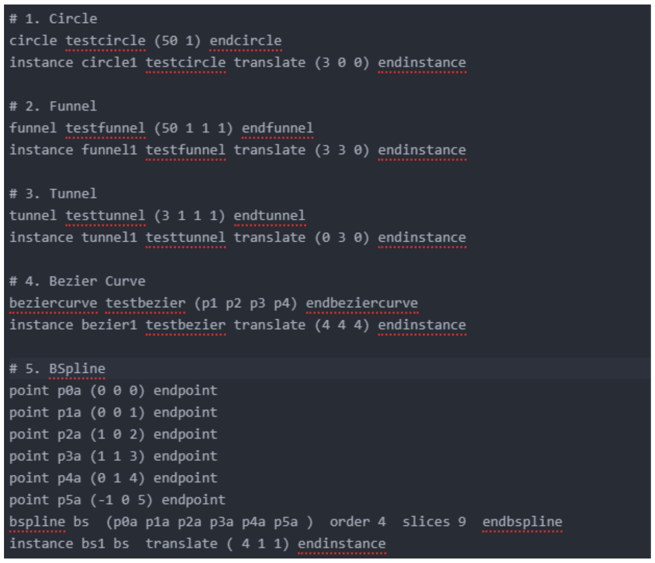
Blender, as briefly mentioned in the previous section, has added scripting support. A Blender user can use Python to automate tasks and animate objects in the GUI. However, it is impossible to construct a mesh using Blender's Python script, make changes interactively in the generated scene, and then save the modifications back into the Python code. Therefore, Blender's Python is not a fully descriptive language that can be utilized to create and modify a hierarchical scene interactively.

These past efforts and related tools have not found a good balance between procedural mesh generation and interactive GUI modifications.

MOTIVATING TASKS

There were three main motivating tasks for my prototype tool. To provide context, these tasks were designed for novice CAD users, specifically users that are interested in constructing shapes without utilizing 3D modelling software that require a lot of manual shape modifications, such as Maya and Blender. The intended users are not expert Maya/Blender users, but rather those that are interested in 3D modelling and programming, and their primary goal is to construct complex shapes while staying in a programming environment.

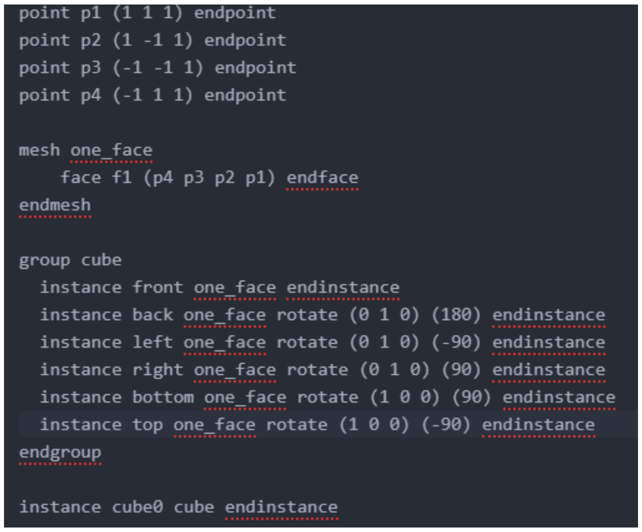
The first motivating task is to be able to generate shapes using pre-defined shape generators from my shape description language. The language should be similar to OpenSCAD's but contain additional complex shape generators beyond basic 3D primitives, such as a torus knot generator. Starting with a blank file and a language reference, the user in this first task should be able to generate a wide variety (greater than the 6 3D primitives OpenSCAD has as default) of primitives and complex shapes using a newly defined procedural CAD language, and then load the corresponding scene by simply opening the file in the GUI. For example, Figure 1 shows 5 shape generators the user should be able to make with my prototype tool and language.



```
# 1. Circle
circle testcircle (50 1) endcircle
instance circle1 testcircle translate (3 0 0) endinstance

# 2. Funnel
funnel testfunnel (50 1 1 1) endfunnel
instance funnel1 testfunnel translate (3 3 0) endinstance

# 3. Tunnel
tunnel testtunnel (3 1 1 1) endtunnel
instance tunnel1 testtunnel translate (0 3 0) endinstance

# 4. Bezier Curve
beziercurve testbezier (p1 p2 p3 p4) endbeziercurve
instance bezier1 testbezier translate (4 4 4) endinstance

# 5. BSpline
point p0a (0 0 0) endpoint
point p1a (0 0 1) endpoint
point p2a (1 0 2) endpoint
point p3a (1 1 3) endpoint
point p4a (0 1 4) endpoint
point p5a (-1 0 5) endpoint
bspline bs  (p0a p1a p2a p3a p4a p5a )  order 4  slices 9  endbspline
instance bs1 bs  translate ( 4 1 1) endinstance
```

Figure 1. Circle, Funnel, Bezier, and B-Spline generators

The second motivating task is to have the user reuse components in meshes to create new shapes and avoid reimplementation. The goal of this task is to implement shapes without needing to define all the individual points or faces; for example, making it so a user does not have to define all 8 points of cube. This task would involve first defining point entities, and using those points to define a face (e.g. a cube face). Then, the user should be able to reuse that face to build complex meshes. For example, Figure 2 shows how a cube could be constructed using a group construct.



```
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 bottom one_face rotate (1 0 0) (90) endinstance
  instance top one_face rotate (1 0 0) (-90) endinstance
endgroup

instance cube0 cube endinstance
```

Figure 2. Creating a cube

This second task is particularly important because in existing 3D procedural CAD modelling tools, there is not a robust scene hierarchy in place to keep track of object relationships and allow objects to be reused in many instances.

If the shape is defined procedurally with a "group" command as shown in Figure 2, it is difficult for existing CAD tools to identify the hierarchical relationships present within the group and which parameter values to alter when the scene is interactively modified in the GUI. This is because in many tools, a consequence of rendering the scene is the scene gets converted into a flat mesh description, making it difficult to save modifications back into the hierarchical code. Thus, it would be useful to create a language that preserves the scene hierarchy when its rendered, and that is what my tool aims to do.

The third motivating task is to save changes made interactively in the GUI back into code, without disturbing the code's readability. This task is differentiating because existing 3D procedural CAD tools do not provide the functionality to save changes back into a file in real-time. Not being able to easily save the changes slows down the workflow and makes it difficult to edit programs as the user would have to constantly alternate between the GUI screen and the code file.

The program at the start of the interaction for this third task would be the GUI with a scene loaded. This task makes most sense in the context of constructing shapes programmatically, modifying the shapes interactively in the GUI, and then trying to save the changes back in the code. Specifically, the user should be able to modify the scene either using sliders or other features, such as by clicking on an "Add Face" button. Then, by clicking a "Commit Changes" button, the user could then save the modifications into the file. For example, if we removed a face interactively in the GUI, it would ideally remove the appropriate face initialization from the code

USER DATA

For user data, I was primarily focused on collecting data that could provide insights on if this procedural CAD tool could reduce the amount of effort and time required to generate shapes while still allowing users to modify the shapes in real-time.

As discussed in class and in the readings, solution viscosity is important for providing flexibility for the user. A cumbersome tool would be one that has few solutions to a problem. My tool aims to reduce viscosity by making it quick and simple for users to group together shapes via a "group" command and create highly complex, symmetrical shapes. I attempted to evaluate this by measuring the time required for users to complete specific CAD tasks.

One of the purposes of my tool is to make 3D modelling tasks less viscous and more flexible for the user (in regards to easily creating the shape and modifying it afterwards), so evaluating the amount of effort/time saved when implementing shapes and evaluating the effectiveness of the interface will allow me to answer my research questions.

I recruited 2 participants who have some experience using 3D modelling tools (e.g. Maya, Blender, OpenSCAD, Solidworks, etc.) and 1 that had zero experience. Participants were recruited from my personal and professional network. Among those that have experience using 3D modelling tools, I recruited one expert designer (someone who has taken Berkeley's UCBUGG: 3D Modelling and Animation course) and a novice user (< 1 year of experience w/ CAD tools). The participant with no prior 3D modelling experience did have some programming experience, but no 3D graphics-related experience. I believe these participants were somewhat representative of users with varying levels of CAD experience. To keep it focused, I prepared a list of question before the interviews and took in-depth notes, which were useful for analysis.

I collected data on the time required to complete a simple CAD task and a complex CAD task using my prototype tool vs OpenSCAD. I also gathered qualitative data and opinions on the different interface/language designs, asking them what they found most exciting about the tools and why.

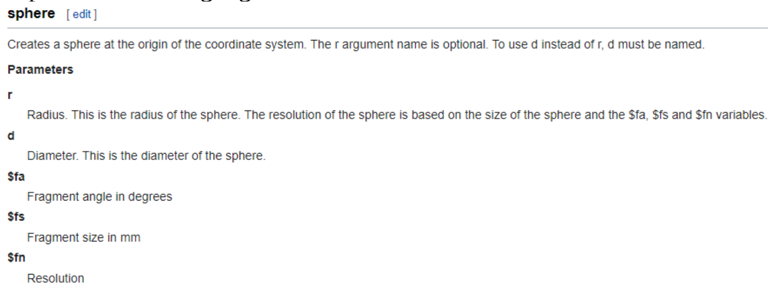
Before the interviews, I had the participants install OpenSCAD and provided them with a precompiled executable of my prototype tool, so they wouldn't need to go through the trouble of building the codebase. Afterwards, we met in one-on-one Zoom calls. I started the interviews by asking them questions about their background and prior experience, collecting relevant demographic variables and making sure the participant is a fit for my intended population. Then, I provided a 15-minute training demo on how to use the tools by showing a dummy example of creating a cube using OpenSCAD and my tool. I shared my screen and clearly indicated the different language syntax used to generate cubes in each respective tool. I also provided links and clearly indicated where the language documentation is located, so they could easily reference them as they worked on their assigned tasks. After the demo, I asked them to share their screen, and I assigned them a simple task and complex task.

The simple task was adding a sphere into the scene, and the complex task was creating a square pyramid. I asked them to complete both tasks using OpenSCAD first, and then complete the same task using the prototype tool. Completing both tasks using a single tool first was to hopefully avoid language syntax confusion, which may have arisen from switching back and forth between tools and languages.

CONCLUSION FROM USER DATA

The participants had OpenSCAD installed and opened (specifically, the preview window, toolbar, and editor window were opened) and used https://www.openscad.org/cheatsheet/ as a language reference.

The two experienced participants were able to complete the first sphere creation task easily, while the inexperienced user felt a bit overwhelmed by the sphere generator syntax depicted in the language reference.



sphere [edit]

Creates a sphere at the origin of the coordinate system. The r argument name is optional. To use d instead of r, d must be named.

Parameters

r
Radius. This is the radius of the sphere. The resolution of the sphere is based on the size of the sphere and the $fa, $fs and $fn variables.

d
Diameter. This is the diameter of the sphere.

$fa
Fragment angle in degrees

$fs
Fragment size in mm

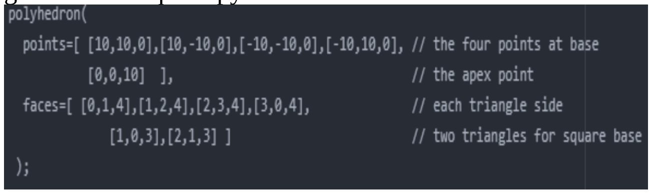
$fn
Resolution

Figure 3. OpenSCAD Sphere Generator Syntax

The zero experience user said, "I'm not sure which parameters are optional and which ones aren't". This was the case because the OpenSCAD sphere generator has 5 parameters: radius, diameter, fragment angle in degrees, fragment size in mm, and resolution. He mentioned he was confused by the definition of the latter 3 parameters, and wished the specs "did a better job with parameter definitions". He said, "it's confusing that [the parameters] are not clearly marked as optional". These findings affected how I implemented optional parameters in my CAD language, typically prioritizing simplicity over complexity or, if the optional parameter is deemed helpful, clearly indicating they are optional in the new language reference.

All users opted to create a sphere using the radius parameter, and not using the remaining optional parameters (e.g. sphere(r=1) was used by the novice user). I recorded the amount of time it took each of these participants to complete the task with no outside resources except the language reference and the text editor, with the recorded time beginning when they share their screen and ending when the sphere is rendered. The experienced participant with > 1 year of experience took ~30 seconds, while the novice user took ~30 seconds as well. The user with no experience took ~2 minutes due to the amount of time spent trying to decipher the sphere syntax.

The second task was to implement a square pyramid on OpenSCAD. The experienced participant completed the task in 3 minutes, the novice user took 7 minutes, and the beginner took 15 minutes (rounded to the nearest minute). This task was relatively more difficult than the initial sphere task because the participants had to use the polyhedron generator and creatively combine points and faces. Figure 4 shows an example of a code that would generate the square pyramid.

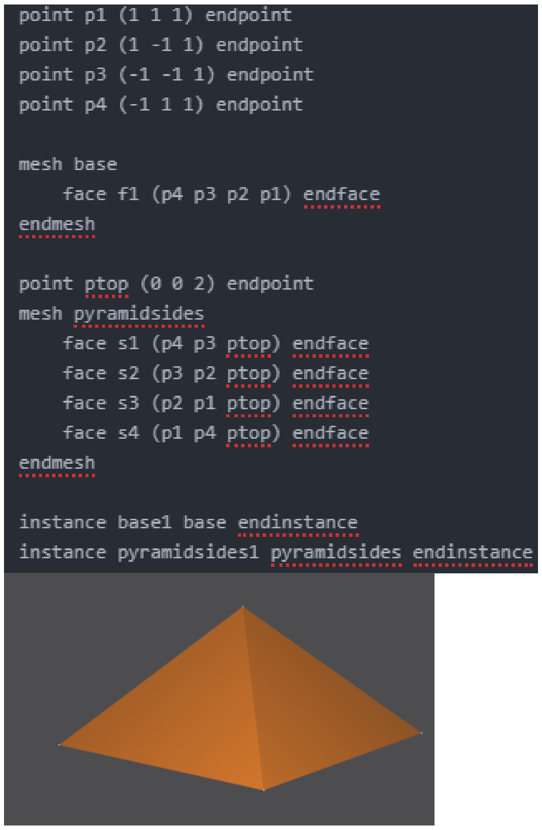


```
polyhedron(
  points=[ [10,10,0],[10,-10,0],[-10,-10,0],[-10,10,0], // the four points at base
         [0,0,10]  ],                                  // the apex point
  faces=[ [0,1,4],[1,2,4],[2,3,4],[3,0,4],              // each triangle side
           [1,0,3],[2,1,3] ]                            // two triangles for square base
 );
```

Figure 4. OpenSCAD Square Pyramid

I then asked the participants to redo the cube and square pyramid tasks using my prototype tool. Creating a sphere using my interactive CAD tool and language took only a few seconds for all users. I believe this is because my sphere generator only has one parameter, radius, so there are no optional parameters that may cause confusion. The most experienced participant stated, “the language reference is easy to understand”. A potential confounding factor is the fact that they had first completed tasks using OpenSCAD, which may have made them more comfortable with the programmable CAD environment and thus more prepared to re-implement the sphere in my prototype environment.

I then asked the 3 participants to recreate the square pyramid using my tool. This task took them significantly longer than when they had done it using OpenSCAD. The experienced user was able to complete the task in 6 minutes, the novice user took 9 minutes, and the beginner user took 13 minutes. The reason this task took relatively longer is because I haven’t implemented the capability to define a vector of point coordinates, so the participants had to define each point individually. Despite this, the novice user mentioned, “he could see how the mesh construct could be used for more complicated shapes”. Figure 5 is an example of the code needed to generate the square pyramid in my prototype tool.



```
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 base
    face f1 (p4 p3 p2 p1) endface
endmesh

point ptop (0 0 2) endpoint
mesh pyramidsides
    face s1 (p4 p3 ptop) endface
    face s2 (p3 p2 ptop) endface
    face s3 (p2 p1 ptop) endface
    face s4 (p1 p4 ptop) endface
endmesh

instance base1 base endinstance
instance pyramidsides1 pyramidsides endinstance
```

Figure 5. Prototype Pyramid Code and Generated Pyramid

To collect more qualitative data, I asked the participants about their experience with Maya, Blender, OpenSCAD, and SolidWorks, if any; specifically asking them what features they found most exciting about the tools. The two experienced participants both prefaced by saying they had experience using Maya and Blender, but no experience with OpenSCAD. The more experienced participant mentioned he was aware of the Blender’s Python scripting functionality, but found the feature to be “too confusing and difficult to set up”. This was surprising given the participant came from CS and Computer Graphics background, so had extensive experience coding geometry. This insight further drove my desire to create a procedural CAD language that requires no prior coding experience, which meant reducing the amount of complex functions and parameters.

What else should I know about your project?

The prototype tool uses event-based programming. There are states and when the program detects an event (e.g., a user opens the code file in the GUI or selects a vertex), the application responds to that event by altering states. The three state elements in the codebase are Document, Scene, and Renderer as shown in Figure 6.

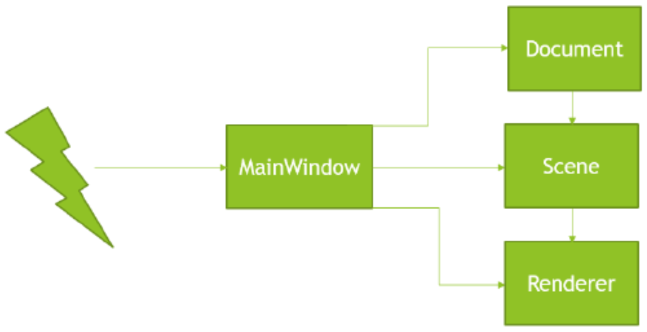
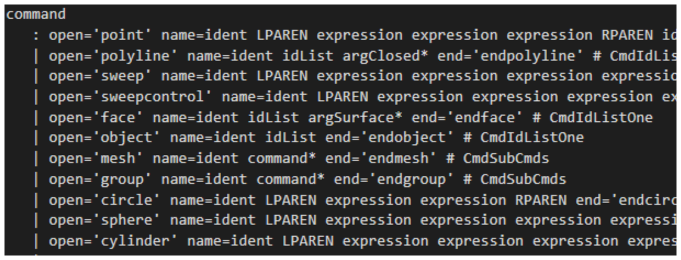


Figure 6. Codebase Design

The Document files are in charge of defining the language grammar, parsing the code file, and converting the code file into an Abstract Syntax Tree (AST). The AST can then be used to build the Scene. I used the ANTLR4 library to build the grammar with its convenient .g4 file implementation.

Figure 7 contains a screenshot of the .g4 file used in the prototype tool.



```
command
    : open='point' name=ident LPAREN expression expression expression RPAREN id
    | open='polyline' name=ident idList argClosed* end='endpolyline' # CmdIdLis
    | open='sweep' name=ident LPAREN expression expression expression expressio
    | open='sweepcontrol' name=ident LPAREN expression expression expression ex
    | open='face' name=ident idList argSurface* end='endface' # CmdIdListOne
    | open='object' name=ident idList end='endobject' # CmdIdListOne
    | open='mesh' name=ident command* end='endmesh' # CmdSubCmds
    | open='group' name=ident command* end='endgroup' # CmdSubCmds
    | open='circle' name=ident LPAREN expression expression RPAREN end='endcirc
    | open='sphere' name=ident LPAREN expression expression expression expressi
    | open='cylinder' name=ident LPAREN expression expression expres
```

Figure 7. ANTLR4 Commands for Prototype Tool

A key component of the tool is parsing the file. After we have constructed the AST using the Document files, we can construct the scene using the AST. The code for this AST-to-Scene conversion first loops over all the commands in that file. It visits all the bank and sets first to create all the sliders. After it creates all the sliders, it goes through all the commands. For each command, it calls VisitCommandSyncScene(). This function classifies the command into the following 4 types and syncs it with the scene:

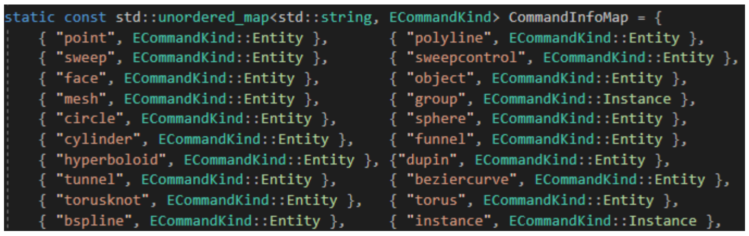
1. Dummy (not handled – just a placeholder name)

2. Entity (the shape generators, including “mesh”)

3. Instance (group or instance commands)

4. BankSet (Slider command)

Figure 8 contains a screenshot of the command type of the various commands.



```
static const std::unordered_map<std::string, ECommandKind> CommandInfoMap = {
    { "point", ECommandKind::Entity },        { "polyline", ECommandKind::Entity },
    { "sweep", ECommandKind::Entity },        { "sweepcontrol", ECommandKind::Entity },
    { "face", ECommandKind::Entity },         { "object", ECommandKind::Entity },
    { "mesh", ECommandKind::Entity },         { "group", ECommandKind::Instance },
    { "circle", ECommandKind::Entity },       { "sphere", ECommandKind::Entity },
    { "cylinder", ECommandKind::Entity },     { "funnel", ECommandKind::Entity },
    { "hyperboloid", ECommandKind::Entity }, {"dupin", ECommandKind::Entity },
    { "tunnel", ECommandKind::Entity },       { "beziercurve", ECommandKind::Entity },
    { "torusknot", ECommandKind::Entity },    { "torus", ECommandKind::Entity },
    { "bspline", ECommandKind::Entity },      { "instance", ECommandKind::Instance },
```

Figure 8. Prototype Command Type

If the command is an Entity command, we create the corresponding entity object. For example, if the entity is a polyline, we create a new polyline object that has the entity name given. Then, the program adds the entity into the scene and connects faces as needed, which is required for a mesh entity. Whenever we add an entity, the code inserts the entity object into an EntityLibrary dictionary for future reference. We only need to reference the entity if it gets instantiated.

The dictionary key is the entity’s name and the value is the entity object, which means providing unique names for each entity is crucially important to prevent overlap. EntityLibrary is useful when we need to retrieve vertex data from certain entities. More importantly, the dictionary is crucial when we instanciate the entity and need to attach the entity to a scene node; this occurs when we need to find the entity from the dictionary and then set the scene node’s entity to be it.

The “mesh” command is kind of a special type of entity because it is allowed to have subcommands. It is similar to “group”; the key difference being “group” is a collection of instances while “mesh” is a collection of entities, specifically faces. Thus, the “group” command is considered an Instance command, while “mesh” is considered an Entity command.

If the command is an Instance command, we make a scene node because every instance needs to have a scene node that is part of the scene graph and the scene tree. The instance can be either an instance of an entity or an instance of a group. To determine if it’s an entity vs group, the code grabs the second identifier name and tries to find an entity. If it finds the entity in the EntityLibrary dictionary, it calls sceneNode- >SetEntity(entity), storing the entity as an InstanceEntity attribute for the scene node (specifically, within each of the scene node’s corresponding scene tree nodes).

If the program doesn’t find an entity that matches the identifier, it tries to find a group. If it finds a group, then it makes the instance scene node a parent of the existing group scene node.

If the command is a group command, it will create a scene node for the group. Importantly, this scene node is created, but not connected to the actual scene graph immediately. In order for it to be added into the scene graph, the user needs to make an instance of the group (similar to how you had to make an instance of an entity in order for the entity to be attached to a scene node). When you make instance of the group (e.g. instance instofG1 G1 endinstance), the code finds the group called G1 in the Group dictionary and then it makes the instance scene node (named instofG1) its parent, and lastly makes SceneRoot the parent of instofG1. Thus, now G1 is linked into the scene graph directly.

Whenever we create a group scene node, what we're actually doing is putting a group scene node into the Groups dictionary. Then, we visit each of the group's subcommands. For example, in Figure 9, there are two subcommands that are both themselves instance commands. Those two subcommands would each create an instance scene node which would be instanciated under a group scene node.

A rather innovative component of my tool is the Scene Graph and Scene Tree data structures I implemented to capture the geometry in the scene. We are, on the fly, using these scene nodes to create scene tree nodes that form a more useful data structure for rendering. This Scene Tree is then fed into the Rendering files to display the scene. To illustrate the difference between these two data structures, let's define an instance of a group called G1. This group contains a mesh and polyline instance as seen in Figure 9.

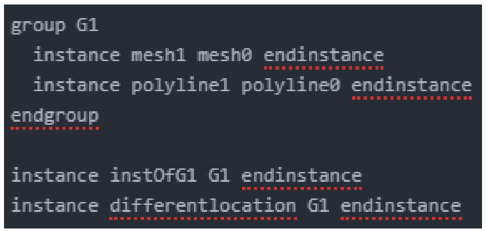

```
group G1
  instance mesh1 mesh0 endinstance
  instance polyline1 polyline0 endinstance
endgroup

instance instOfG1 G1 endinstance
instance differentlocation G1 endinstance
```

Figure 9. Group with Mesh and Polyline

Then, the left graph in Figure 10 would be the corresponding Scene Graph built (which contains scene nodes) and the right graph would be the Scene Tree (which contains scene tree nodes). As you can see the Scene Tree allows each mesh (e.g. polyline1) to have a unique path for each time it's been instanciated. This is not the case with the Scene Graph. The Scene Graph does not have two unique paths to polyline1 for example. There is just one path. This is problematic if we only used the Scene Graph data structure as you can imagine the renderer would not be able to figure out what objects to alter in the scene if a slider is moved. For example, using just a Scene Graph representation, a user may wonder if moving a parameter's slider would alter just instOfG1's polyline1 or another instance of G1's polyline1. Unfortunately, the slider would incorrectly alter both, and that is why we need to construct a Scene Tree and use it for rendering.

To summarize, Scene Tree has a unique path to each object in the scene, and this is the key difference between the Scene Graph and the Scene Tree; the unique path allows us to reuse the same objects without confusing the renderer.

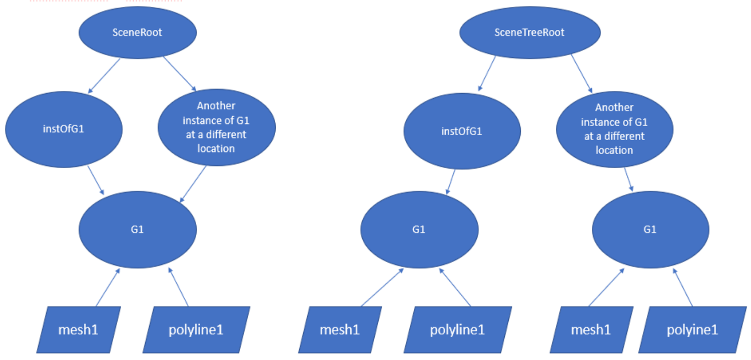

SceneRoot
instOfG1
Another instance of G1 at a different location
G1
mesh1
polyline1
SceneTreeRoot
instOfG1
Another instance of G1 at a different location
G1
G1
mesh1
polyline1
mesh1
polyine1

Figure 10. Scene Graph and Scene Tree for Figure 9 code

I will now discuss the workflow needed to complete the three motivating tasks described in a previous section. The first motivating task was to generate a set of shapes using pre-defined generators.
Here are the generators I have implemented for this project:

Include Files
Syntax:
include "file_name.nom"

Description:
Allows to combine frequently used statements, such as specification of surface colors or generally useful geometry, such as a triplet of coordinate axes, in special files that can then be included with a single-line command. Another example is: a collection of camera, light, and window/viewport specifications for the rendering process.

Numerical Parameters and Sliders
Bank
Syntax:
bank bankID
 set setID1 value1 start1 end1 step_size1 [0,1]
 ...
 set setIDN valueN startN endN step_sizeN [0,1]
endbank

Description:
Allows the user to change any numerical value in the file through an interactive slider in the GUI.
setID: the variable to be parameterized.
value: the initial value of the slider.
start: the lower bound of the slider.
end: the upper bound of the slider.
step_size: the incremental step size of the slider.
[0,1]: an optional flag to show this slider (1) or skip it (0) in the displayed bank.

Generators
Point
Syntax:
point id (x y z) endpoint

Description:
Defines a point at the specified x, y, and z coordinates.

Polyline
Syntax:
polyline id (point_idlist) [closed] [surface surface_id]
endpolyline
Description:
Defines a polyline, a chain of piecewise linear segments. You can optionally make it closed, i.e., the last point connects back to the first.
point_idlist: a list of points of the form point1 point2 ...

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

Description:
Defines a face from a list of points. Front face uses counter-clockwise winding.
point_idlist: a list of points of the form point1 point2 ...

Bezier Curve
Syntax:
beziercurve id (point_idlist) segs endbeziercurve

Description:
Defines a Bezier curve.
point_idlist: a list of control points of the form: point1 point2 ...
segs: the number of segments into which the Bezier curve is sampled.

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

Description:
Defines a B-spline.
{order}: integer that sets the B-spline's DEGREE to be {order}-1.
point_idlist: a list of control points of the form: point1 point2 ...
segs: the number of segments into which the B-spline is sampled.
The number of control points must be greater than or equal to {order}
For closed curves, there must be at least {order}-1 control points.

Mesh
Syntax:
```
mesh id
 face faceId1 ( point_idlist1 ) endface
 ...
 face faceIdN ( point_idlistN ) endface
endmesh
```

Description:
Also creates a collection of faces, which can optionally be colored. Faces in a mesh can then be referred to in the rest of the program via a hierarchical name: id.faceId. Variable names must be unique within a mesh.
faceId: the name of the face
point_idlist: a list of points of the form: point1 point2 ...

Circle
Syntax:
circle id (radius segs) endcircle

Description:
Defines a circle.
radius: the radius.
segs: the number of line segments.
"botcap": if present, draw the bottom face (with downward normal).
"topcap": if present, draw the top face on the cylinder..

Sphere
Syntax:
sphere id (radius) endsphere

Description:
Defines a sphere.
radius: the radius of the sphere..

Torus
Syntax:
torus id (maj_rad min_rad theta_max phi_min phi_max theta_segs phi_segs) endtorus

Description:
Defines a torus.
maj_rad: the major radius.
min_rad: the minor radius of the outer ring.
theta_max: specified in degrees. The minor cross-section circle is swept starting at the x-axis and circles the z-axis by the angle theta until thetamax is reached (=< 360).
phi_min: starting angle in degrees around the minor circle.
phi_max: terminating angle in degrees around the minor circle.
0 <= phi_min < phi_max <= 360 (degrees).
theta_segs: the number of segments along the major radius.
phi_segs: the number of segments around the minor radius.

Torus Knot
Syntax:
torusknot id (symm turns maj_rad min_rad tube_rad circ-segs sweep_segs) endtorusknot

Description:
Defines a torus knot.
symm: sweeps through the donut hole = rotational symmetry of knot
turns: turns around the donut hole
maj_rad: the major radius of the donut.
min_rad: the minor donut radius (tube radius).
tube_rad: radius of swept circle. For tube_rad =0, only the sweep path is output.
circ_segs: the number of segments on the circular cross section.
sweep_segs: the number of segments along the sweep path.

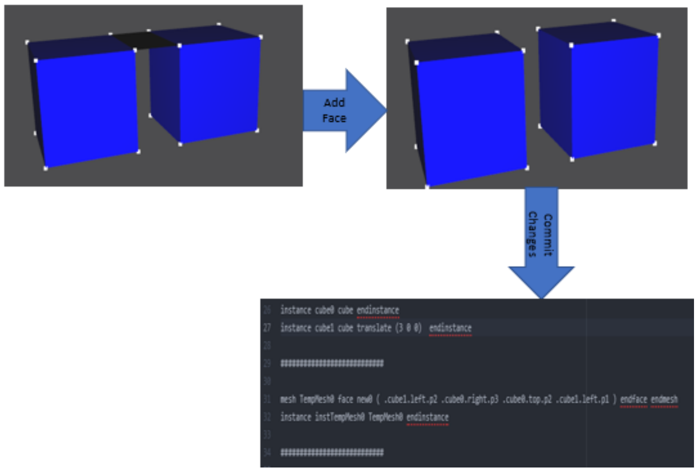
Scene Graph
Instance
Syntax:
instance name object [rotate (rx ry rz){in degrees}] [scale (sx sy sz)] [translate (tx ty tz)] [surface surface_id] endinstance

Description:
Creates an instance of geometry. One can optionally rotate, scale, translate the instance and specify its color.
object: the name of the primitive, generator, or group that will be instantiated.
surface_id: a specified (RGB) surface color

Group
Syntax:
group id
  instance id1 object_id1 [instance_parameters] endinstance
  ...
  instance idN object_idN [instance_parameters] endinstance
endgroup

Description:
Defines a collection of instances of primitive objects or other groups. Groups are the most general construct to introduce hierarchy into a model description.
id: the name of the instance.
object_id: the name of the object to be instantiated.
[instance_parameters]: all the optional parameters discussed above.

The second task was to reuse components in mesh to create new shapes. **Figure 2** shows the "group" command being used. Users could reuse the single face and rotate it to form a cube.

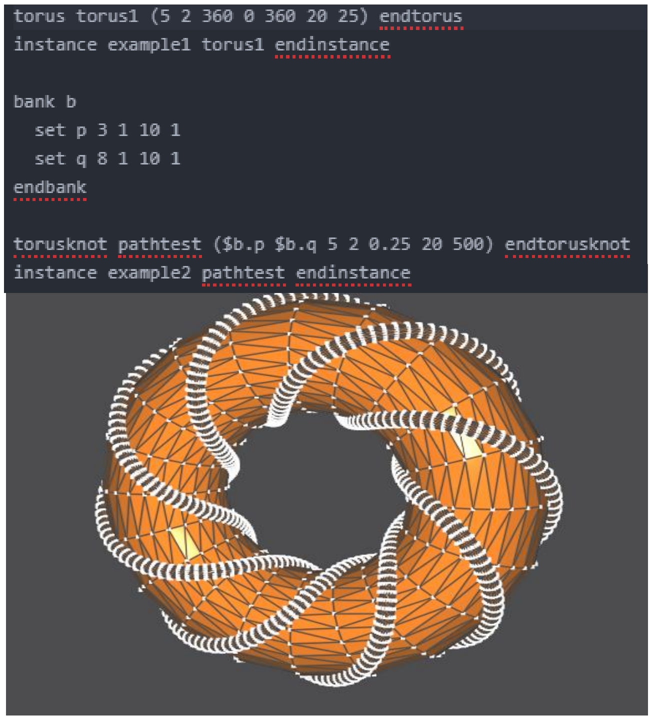


```
instance cube0 cube endinstance
instance cube1 cube translate (3 0 0)  endinstance

##########################

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

##########################
```

Figure 11. Add Face and Committing the Changes

The third motivating task was to be able to save changes back into the code file. In **Figure 11,** we see an example of "Add Face" in action as well as commit changes. As you can see, the added face is appended to the bottom of the file as a mesh and an instance.

Figure 12 shows a torus knot wrapped around the torus surface.



```
torus torus1 (5 2 360 0 360 20 25) endtorus
instance example1 torus1 endinstance

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

torusknot pathtest ($b.p $b.q 5 2 0.25 20 500) endtorusknot
instance example2 pathtest endinstance
```

Figure 12. Torus and Torus Knot Code and Scene

REFERENCES

[1] https://www.openscad.org/cheatsheet/

[2] https://computationalmodelling.bitbucket.io/tools/CSG.html

[3] https://www.openscad.org/about.html