Python Toolpathing for 3DP from mathematical transformations

Luis Pacheco
luigipach@gmail.com
Neobrutal Studio

Abstract

This is a design workflow method for developing 3DP toolpaths (gcode) with Python Scripting Node inside Blender's addon Sverchok. It involves using trigonometric functions and transformations described by Joseph Choma in the book Morphing for generating the geometry with python and expanding the customization of the geometry with python flow control. @choma_morphing_2015.

Introduction

Current 3DP toolpath generation (gcode) is generally achieved by "Slicing" software that while it has improved substantially in the last years still limits the control on the final toolpath. Another alternative involves using visual programming languages (Grasshopper) to parse polylines or curves into GCODE as described in the book Advanced 3D Printing with Grasshopper@cuevas_advanced_2020. A more recent method involves programming the gcode and setting "path" operations in order with Full Control Gcode Designer.@gleadall_fullcontrol_2021.

This workflow method aims to be the starting point for a tool used to generate toolpaths combining a visual-based programming language (Blender Nodes) with traditonal text-based programming (Python) for logic operations and control flow, to simplify and facilitate code readability. Including machine controll and interaction.

Blender comes with some integrated tools that allows users to experiment this method without the need to download any external addons, complicated installation or pay any licences. It provides a fully functional and extensible python interpreter and multiple simulation tools for vizualizing the toolpath.

State of gcode inside Blender.

Currently there are some pre-existing open source tools for generating Gcode withing Blender:

- BlenderCAM by Vilém Duha (external addon)
    - Traditional CNC toolpath generator for Blender.
- Gcode exporter by Alessandro Zomparelli (external addon)
    - Export edges, paths or polylines as gcode.

- Sverchok gcode export node also by Alessandro Zomparelli (included as part of Sverchok)
    - Same as the stand alone addon but integrated in Sverchok, receives a list of vertices to parse them into gcode.
- Nozzleboss by Heinz Loepmeier
    - Uses weightmaps and vertex colors to cutomize flow and speed for texturizing the print.
- nCNC by Nuh Ayvaz (external addon, discontinued)
    - Converts curves to gcode, includes a machine controller and vizualizing machine position in Blender.

Blender Layout

Blender's main interface is not optimized for the this workflow.

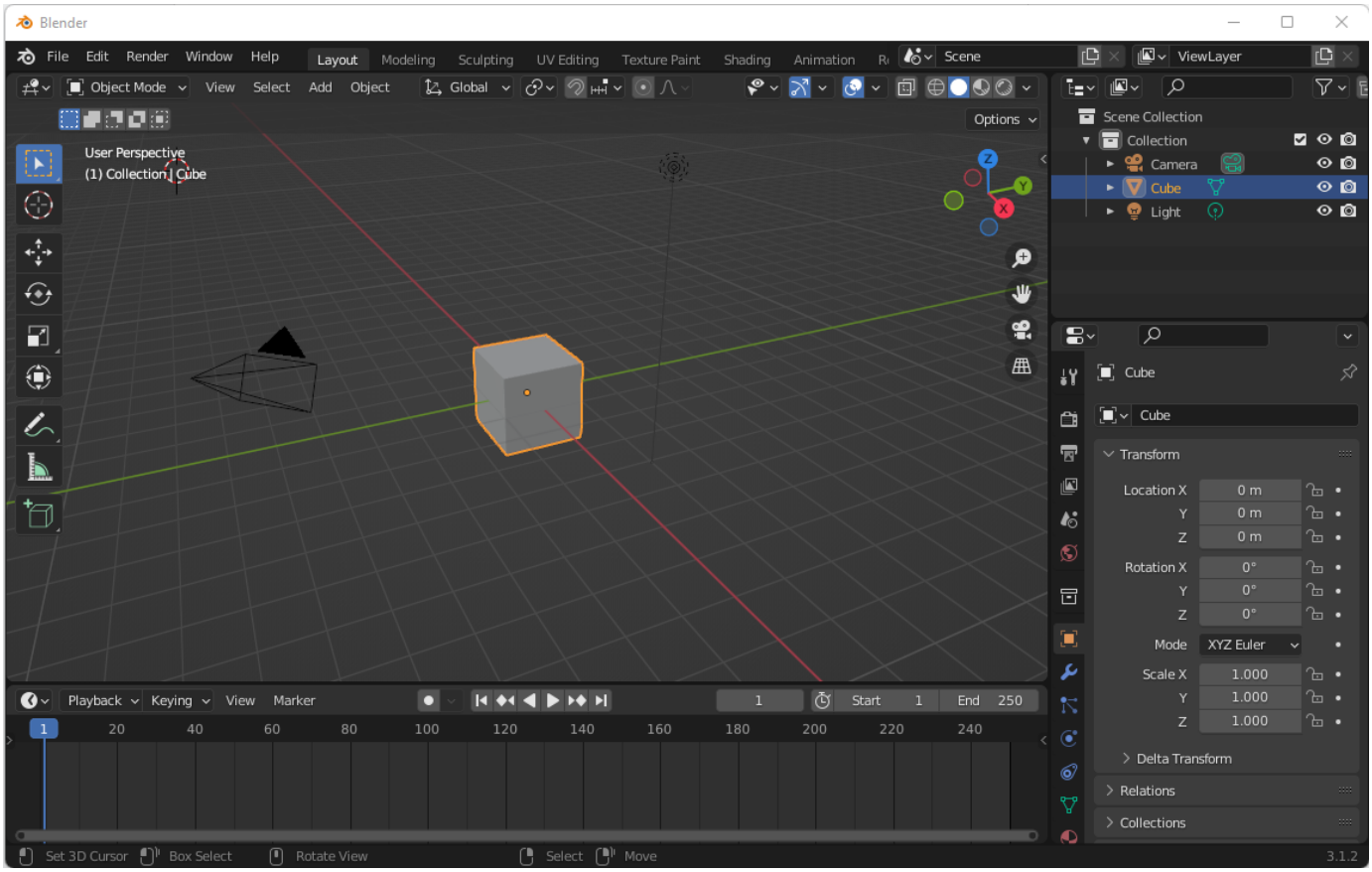


Figure 001 Blender's default workspace

First it's important to activate the included addon Sverchok following the next steps:

1.-Go to edit ---> preferences

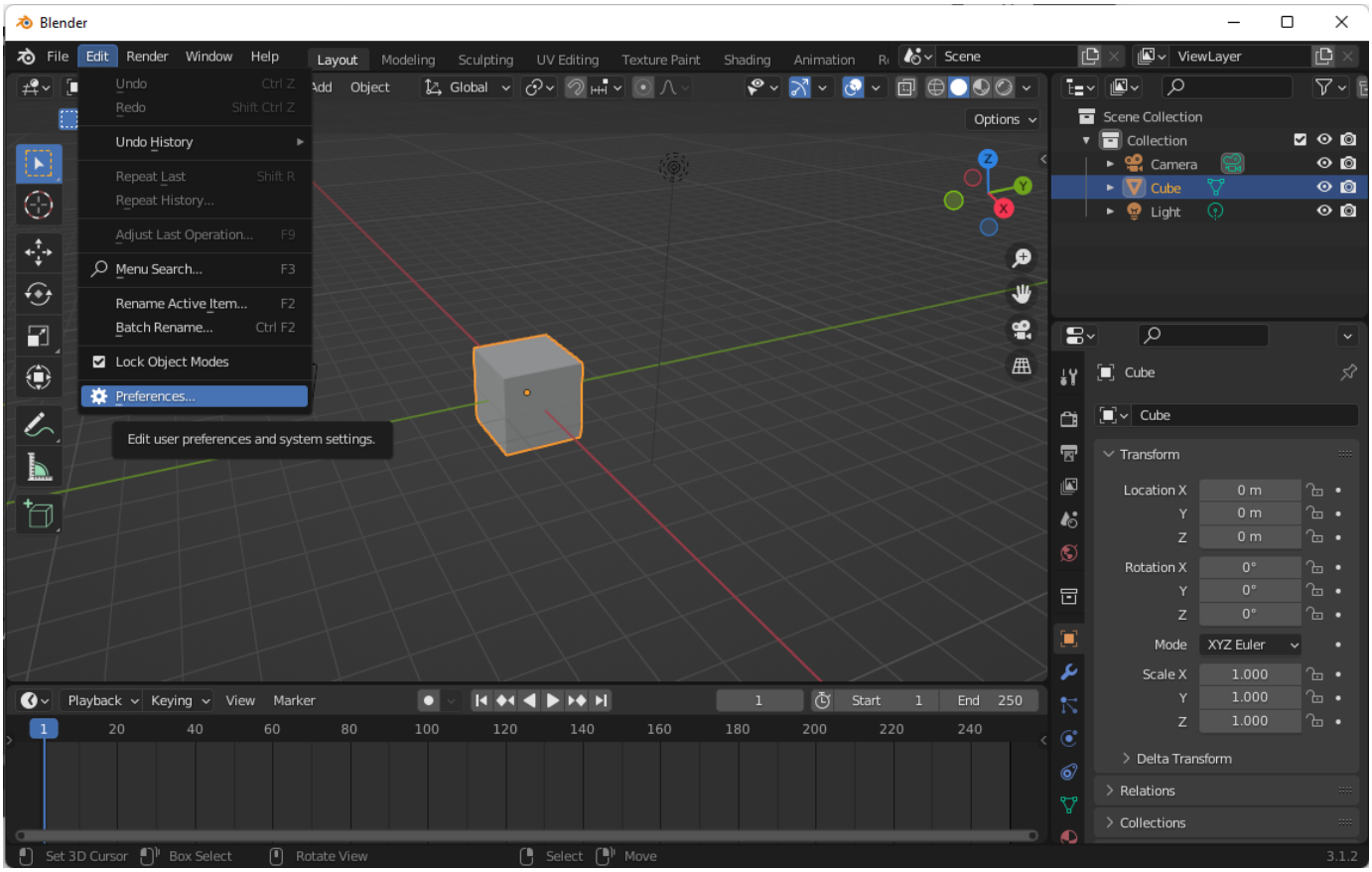


Figure 002 Selecting preferences

2.- Select add-ons

3.- Type sverchok on the search box

4.- Click on the checkbox next to Node: Sverchok

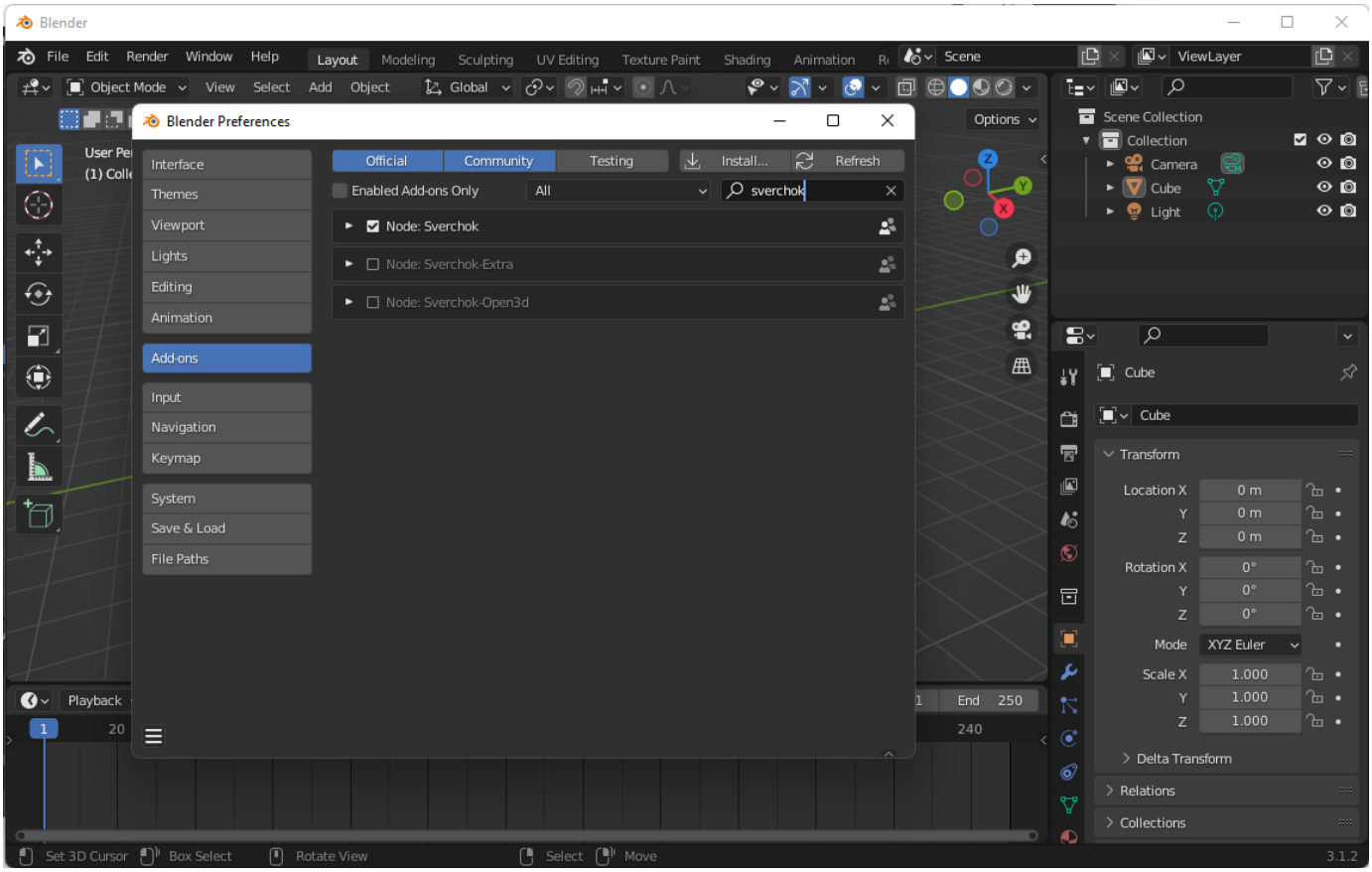


Figure 003 Blender add-ons preferences

Second we will rearrange the workspace.

1.- Locate the mouse pointer on the top corner right between the 3D view and the outliner, the pointer will change to a cross.

2.- Click and drag the new "workspace" into the 3D view.

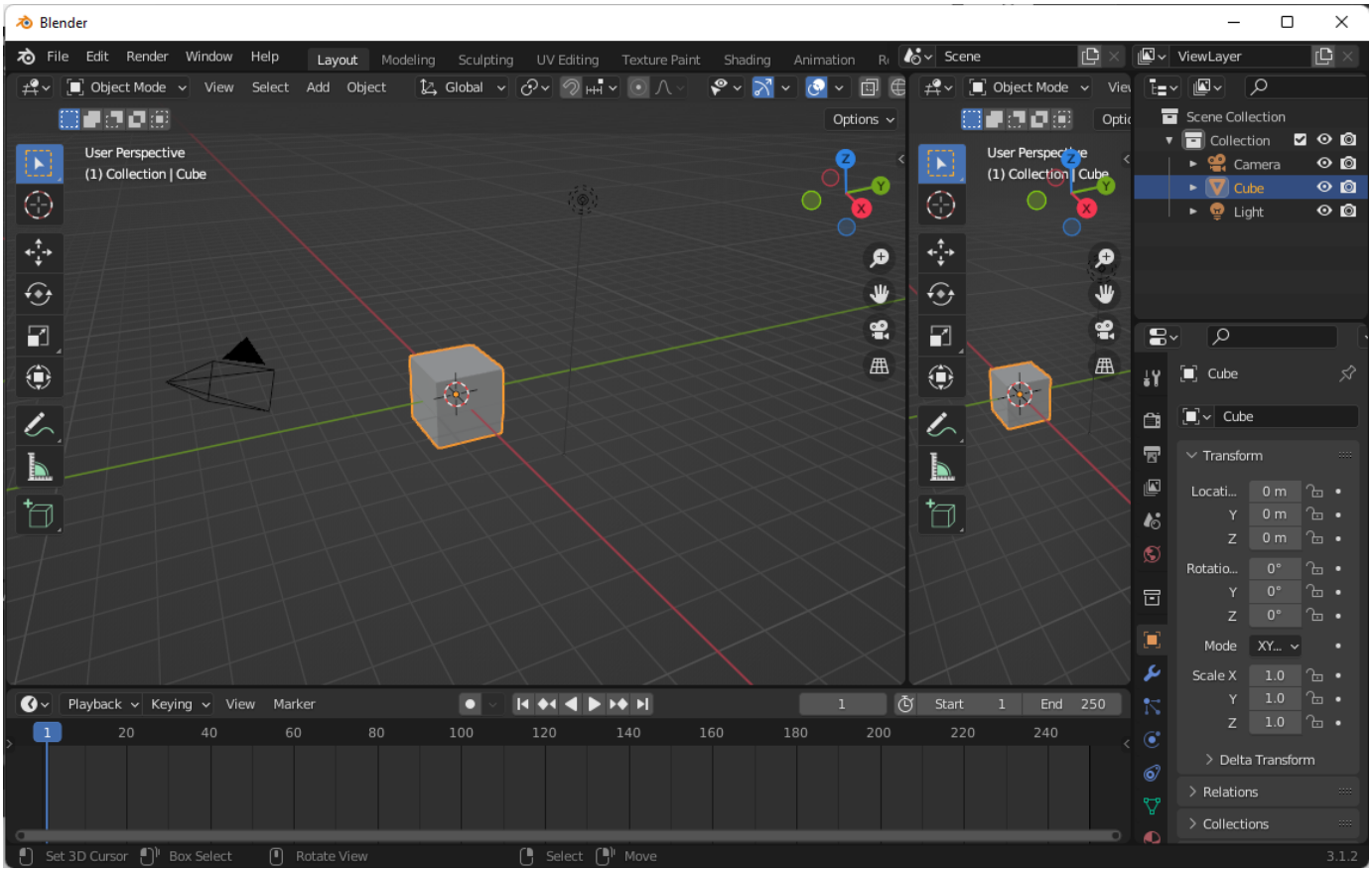


Figure 004 Creating a new workspace window

3.- Click on the left top icon on the new "workspace" and change its type to Text editor.

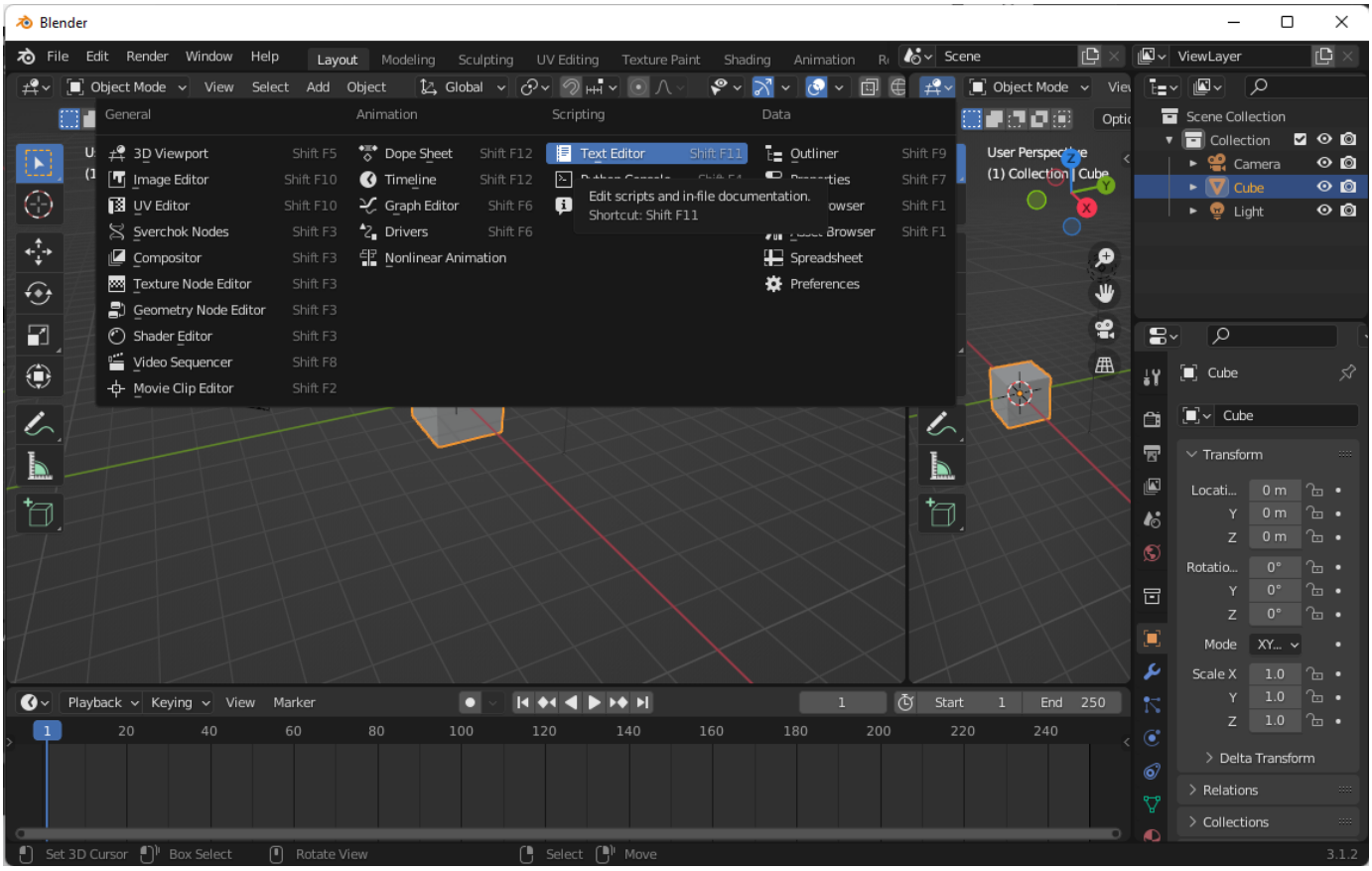


Figure 005 changing the new workspace window to a text editor

4.- Click on the left top icon on the "timeline workspace" and change its type to Sverchok Nodes .

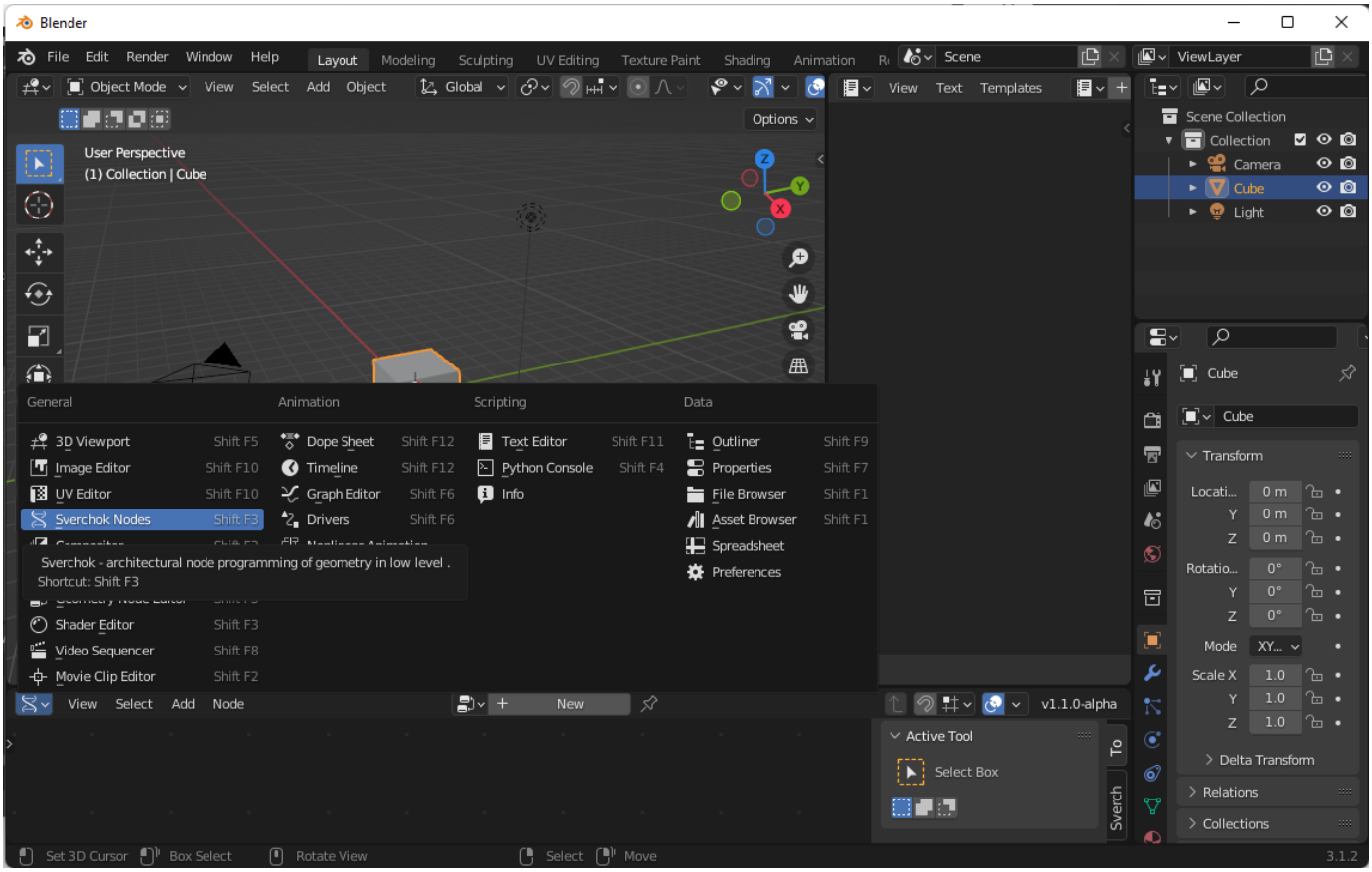


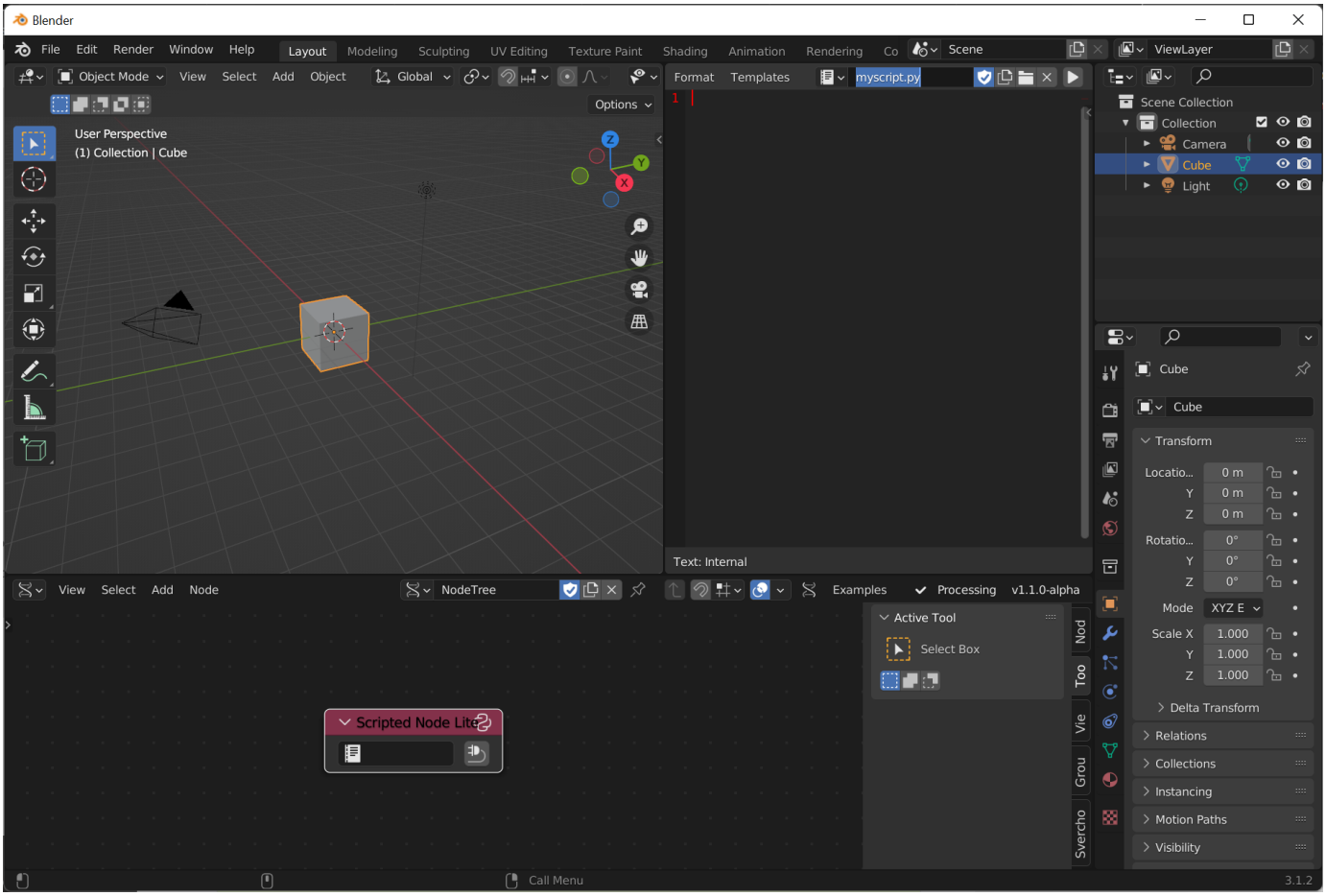
Figure 006 changing the timeline workspace window to a Sverchok nodes editor

Scripting and visualizing the geometry

Now we will setup a node tree that allows us to get started:

1.- Click on + New on the Sverchok node editor
2.- Press Shift + A while on the Svechok window
3.- Go to Script --> Scripted Node Lite

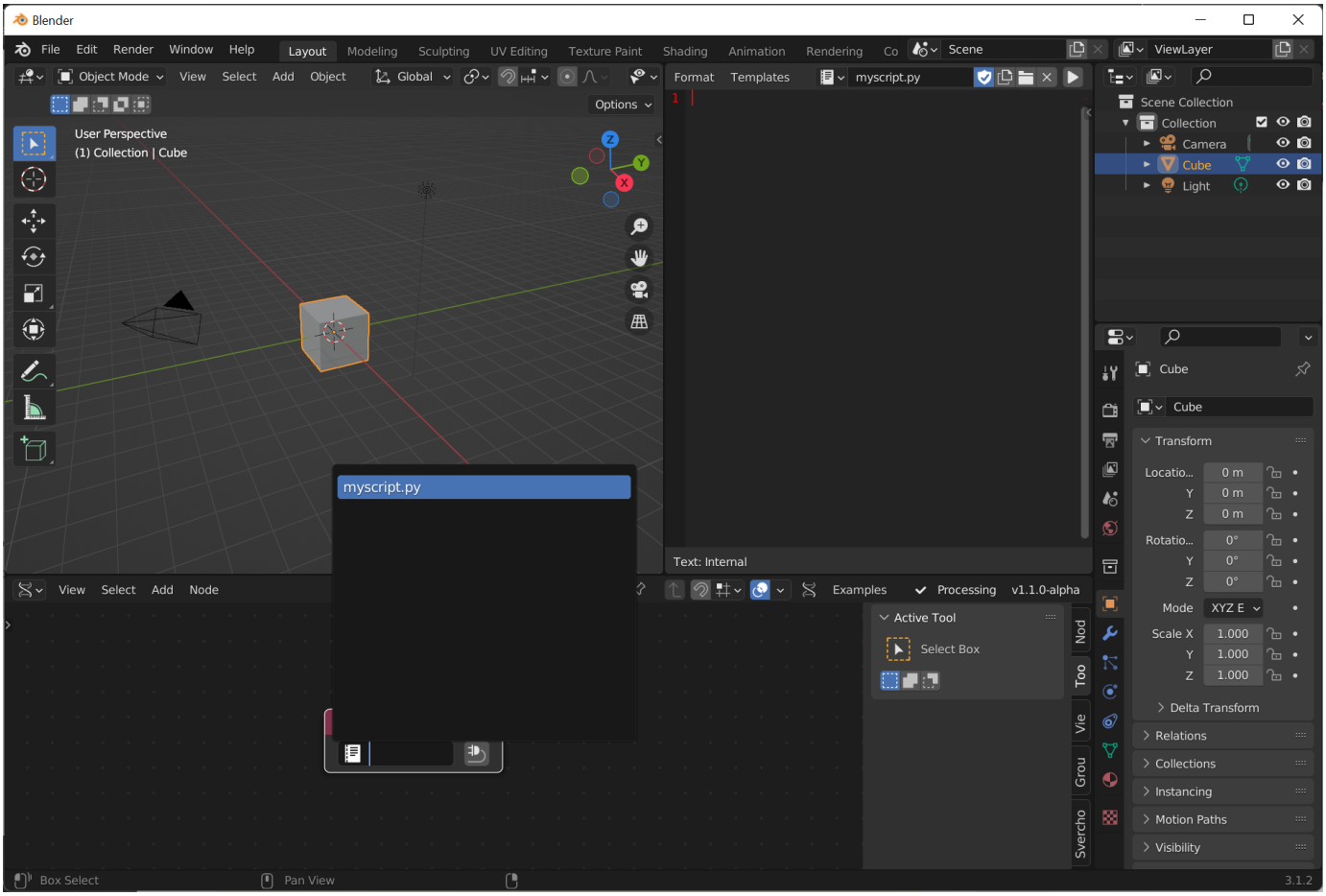
If you click on the notebook icon, you'll see the scripted node lite is looking for a text file, in this case the text file will come from our text editor, but now its empty.



4.- Click on + New on the Text editor

5.- Rename the file on the top of the text editor to myscript.py

6.- Click again on the notebook icon in the Sverchok scripted node lite and select myscript.py



From the Sverchok Documentation @sverchok-team_script_nodate:

Now in order to initialize our scripted node, input and output sockets we will need to define our directive on the text editor, wich is wrapped with triple quote marks """, this has to be the first thing on the text editor:

```
"""
in socketname  type  default=x nested=n
in socketname2 type  default=x nested=n
out socketname type  # (optional)
"""
< python code >
```

For each input and output we define:

- direction (in or out)
- socket name wich will automatically become available in the script as a local variable or empty list
- socket type:
  - Vertices (`v`)
  - Strings/Lists (`s`)
  - Matrices (`m`)
  - Curves (`C`)
  - Surfaces (`S`)
  - Solids (`So`)
  - Scalar fields (`SF`)
  - Vector fields (`VF`)
  - Objects (`o`)
  - File Path (`FP`)
- Default value
- Nestedness level.

In this case we will start with the following directive:

```
"""
in rv s d=0.1 n=2
in ru s d=0.1 n=2
in su s d=0.0 n=2
in nu s d=2.0 n=2
in sv s d=0.0 n=2
in nv s d=1.0 n=2
```

```
out vertices v
out edges s
out polylines v
out polyedges s
"""
```

The input sockets stand for:

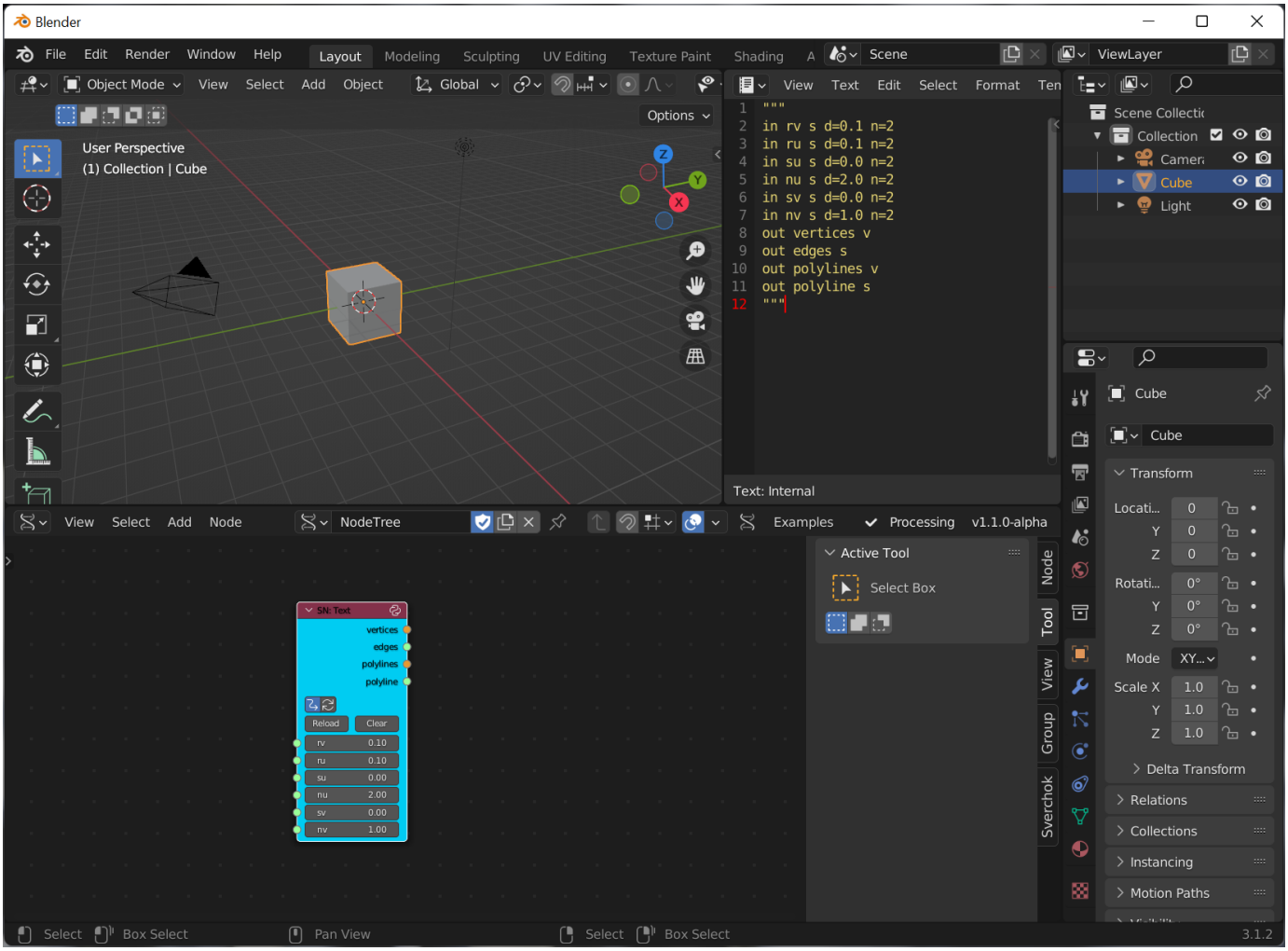
- rv -resolution on V
- ru -resolution on U
- sv - cut start on V
- nv-cut end on V
- su -cut start on U
- nu -cut end on U

The letters "U" and "V" denote the axes of the 2D texture because "X", "Y", and "Z" are already used to denote the axes of the 3D object in model space @noauthor_uv_2022

The output sockets stand for:

- vertices - multidimensional list of vertices
- edges - multidimensional list of edges
- polyline - one list of all vertices
- polyedge - one list of all edges

Now if we click on the Plug button on the Scripted Node Lite node in the Sverchok editor the node will load our sockets:



Sverchok scripted node lite allows us to import any python library installed on our interpreter, we will be using python's math library for this workflow:

`import math`

Following we will calculate the steps, start and end UV cutting:

```
startV = math.pi * sv  #cut v at start
stopV = math.pi * nv  #cut v at end
stepSizeV = rv #redundat but helps code redability
stepsV = int(abs(stopV-startV)//stepSizeV) #total v steps
startSkipV=int(abs(startV)/stepSizeV) #steps that get cut from start v
endSkipV=int(abs(stopV)/stepSizeV)#steps that get cut from end v
#print(startSkipV) # prints value for debugging
#print(endSkipV) # prints value for debugging

startU =  math.pi * su      #cut u at start
stopU = math.pi * nu  #cut u at end
stepSizeU = ru #redundat but helps code redability
stepsU = int(abs(stopU-startU)/stepSizeU)#total u steps
startSkipU=int(abs(startU)/stepSizeU) #steps that get cut from start u
endSkipU=int(abs(stopU)/stepSizeU) #steps that get cut from end v
```

```
#print(startSkipU) # prints value for debugging
#print(endSkipU) # prints value for debugging
```

Now we will define our the main control flow to test our geometry functions:

```
for stepV in range(startSkipV,endSkipV+1: #counts the V steps between the start and end cut parameters and starts iterating for each one
    print("stepV " + str(stepV)) #prints the current iteration of V
    v = stepV*stepSizeV #multiplies the current iteration times the step size on V
    print("v "+ str(v)) #optinal for debugging
    pointList = [] #creates a temporal point list for current V iteration
    edgeList =[]  #creates a temporal edge list for current V iteration
    for stepU in range(startSkipU,endSkipU+1):  #counts the U steps between the start and end cut parameters and starts iterating for each one
        u=stepU*stepSizeU #multiplies the current iteration times the step size on U

        x= math.sin(u) #Here we set a function for the X value
        y= math.cos(u)#Here we set a function for the Y value
        z= v #Here we set a function for the Z value

        p = [x,y,z] #creates a point
```

```
        pointList.append(p)#appends the point to our current V iteration list
        polyline.append(p) #appends the point to the polyline
    vertices.append(pointList) #appends our current V iteration list to our multidimensional vertices list

    for i in range(len(pointList)+1): #this  creates our edges list for our multidimensional vertices list
        v1 = i
        v2 = i + 1
        ed = [v1,v2]
        edgeList.append(ed)
    for i in range(len(polyline)+1):#this  creates our edges list for our polyline vertices list
        v1 = i
        v2 = i + 1
        ed = [v1,v2]
        polyedges.append(ed)
    edges.append(edgeList)
```

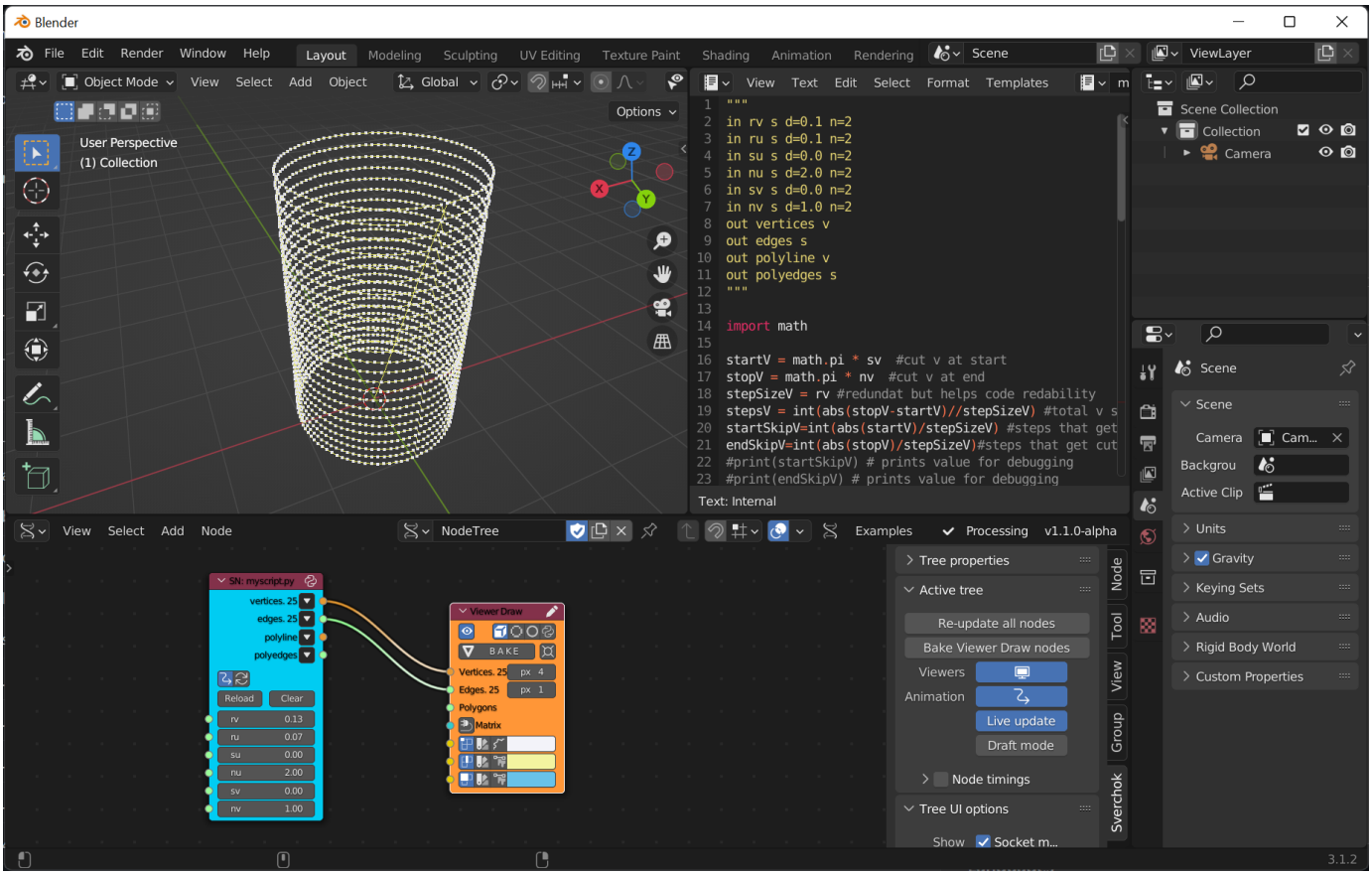
```
polyline=[polyline] #avoids wraping the output socket in the  node editor
print("done") #prints done to the console to notify that everything worked as expecteds done to the console to notify that everything worked as expected
```

Visualizing our toolpath

Now we will setup a node tree that allows us to Visualize our toolpath:

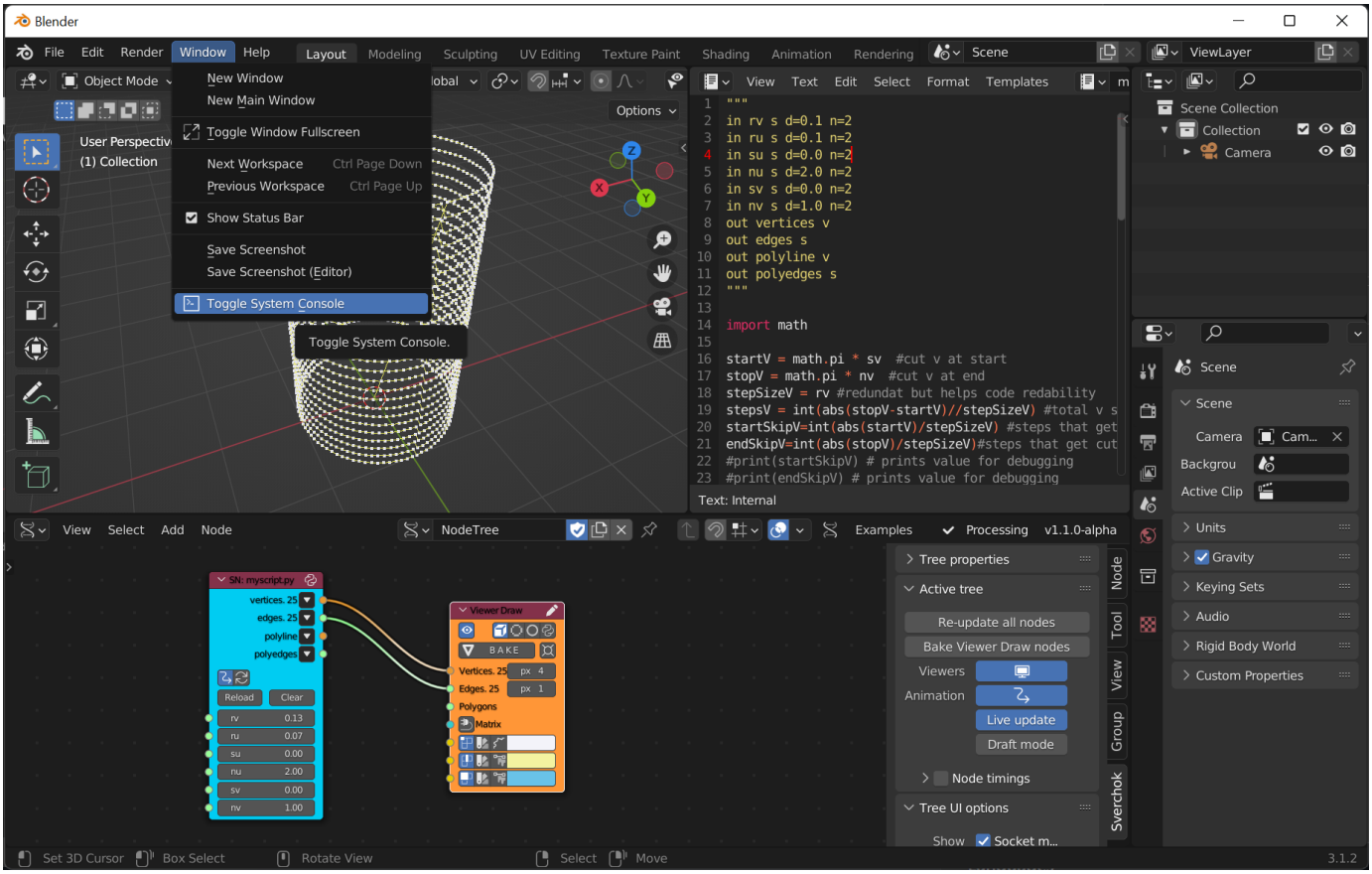
1.- Press Shift + A while on the Sverchok window
2.- Go to Viz --> Viewer Draw
3.- plug in the vertices and edges output to the vertices and edges input on the Viewer draw

If everything is working you should be able o view the vertices and edges on the 3D viewer



For debbuging you can enable the system console:

1- Go to Window--->Toggle System Console



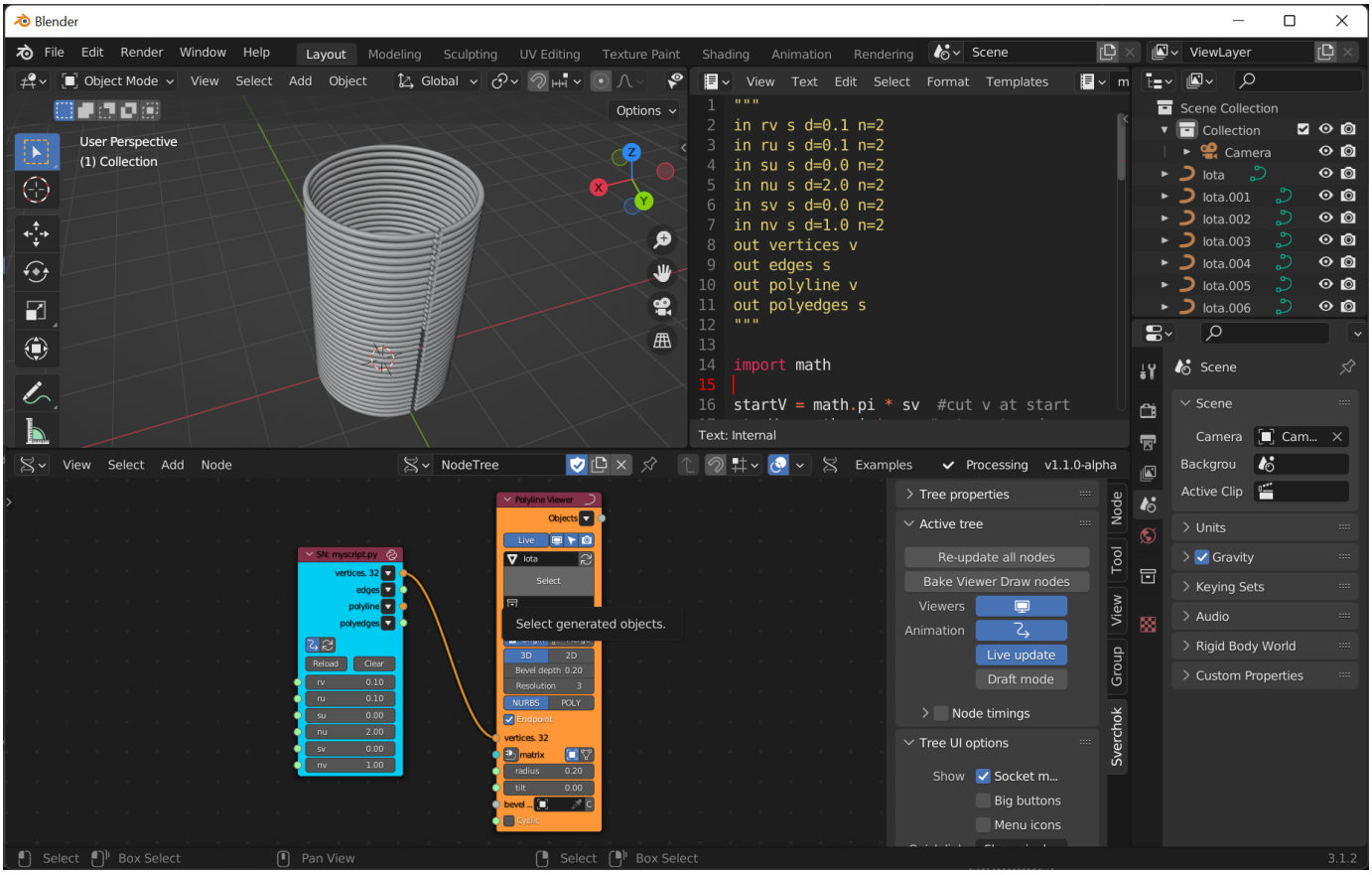
The System Console will print any print statements on your code and display any errors when executing your code.

Test the different parameters to understand how each one affects the geometry.

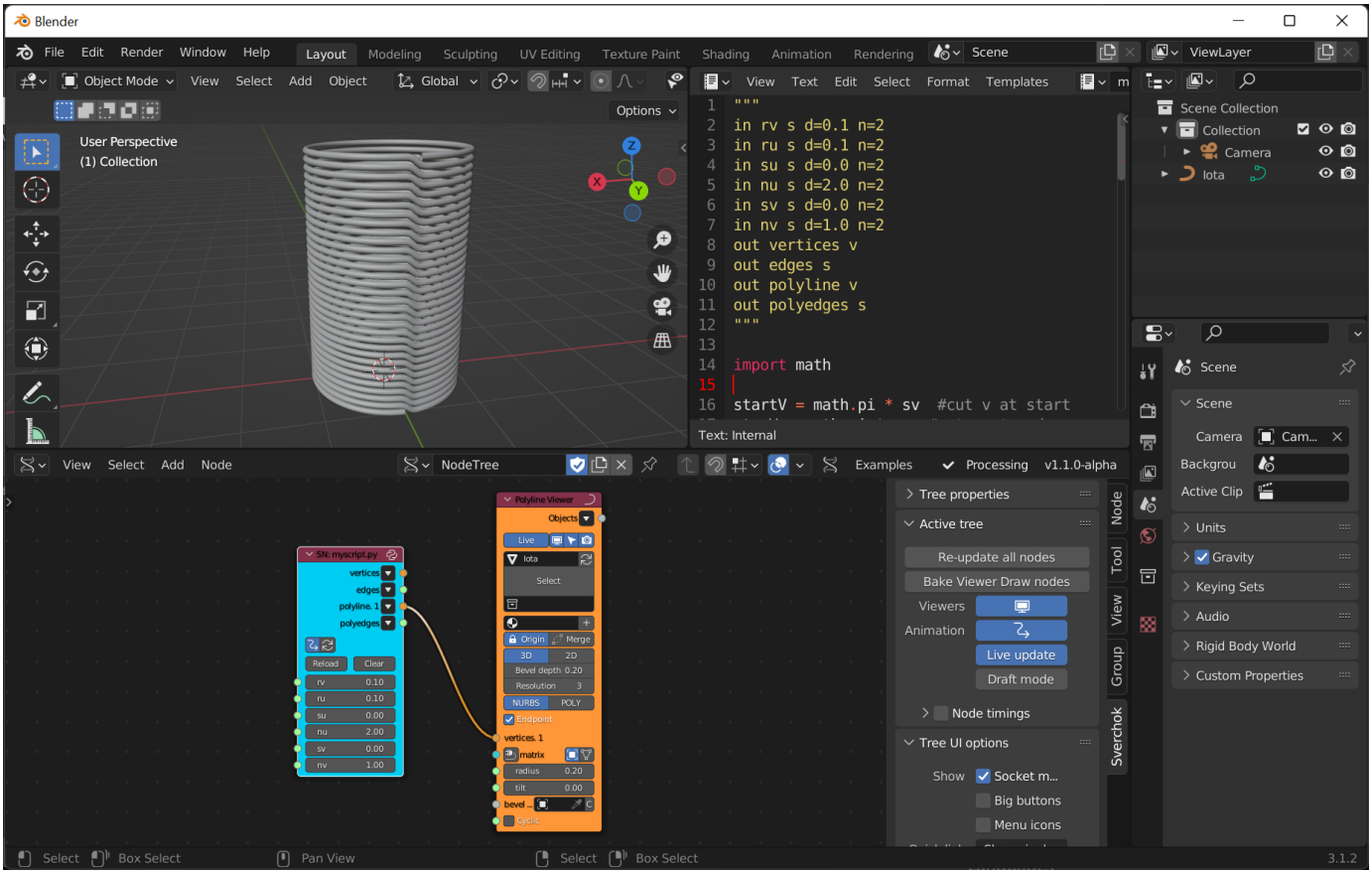
A better way to visualize our toolpath is using the polyline viewer:

1.- Press Shift + A while on the Sverchok window
2.- Go to Viz --> Polyline Viewer
3.- plug in the vertices output to the vertices input on the Polyline Viewer

4.- Activate the endpoint checkbox.



The key difference between the vertices and polyline output from our script can be better observed by plugging the polyline output from our scripted node to the vertices input of the polyline viewer node.

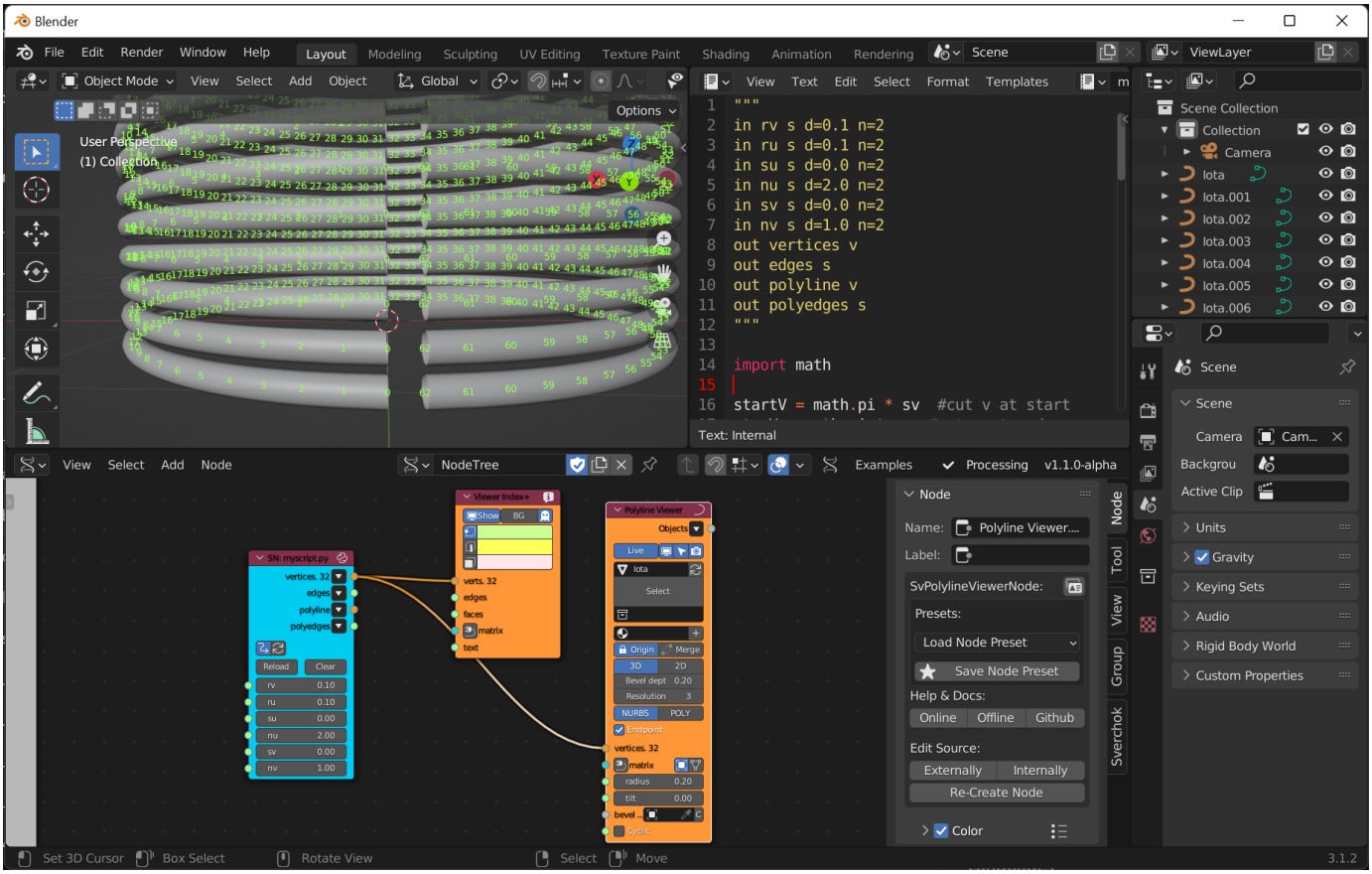


Notice the vertices output gives us a list of lists with points for each polyline, while the polyline output returns one list containing a list with all our vertices as part of one polyline connecting the last vertex of the previous polyline to the first vertex of the following polyline.

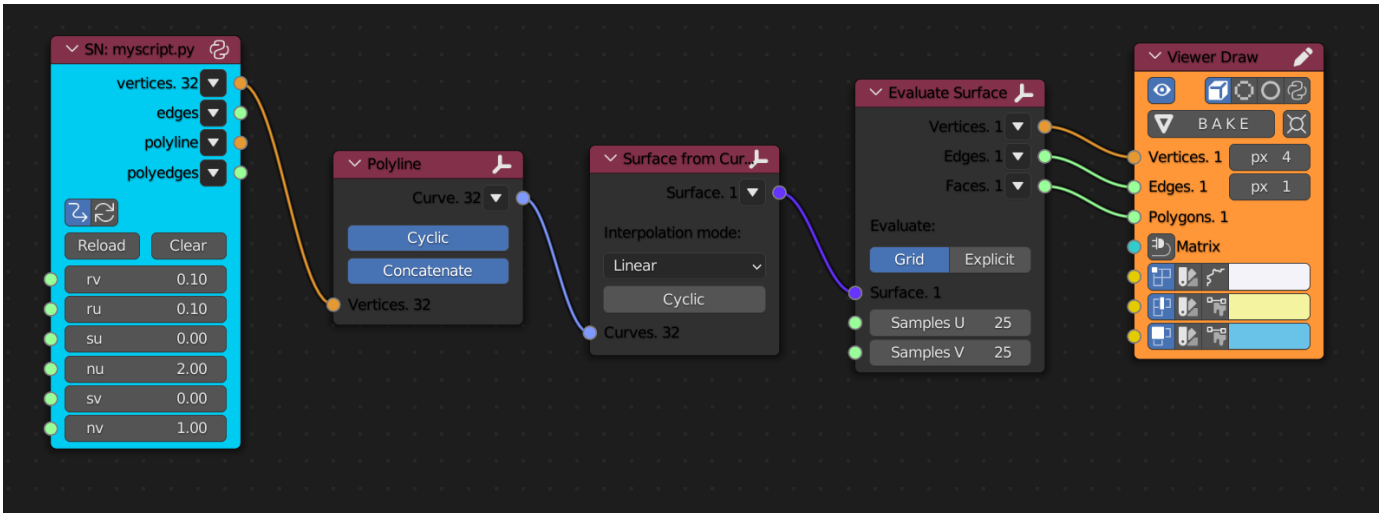
We can take a look at the data structure for each of this outputs by:

1.- Press Shift + A while on the Sverchok window
2.- Go to Viz --> Viewer Index +
3.- Plug in the vertices output to the vertices input on the Viewer Index +

By alternating the vertices output with the polyline output you can observe how the index number resets for every item in the vertices list, while they add up on the polyline output. It is possible to test different data structures for different Visualization methods.



A surface can be created from the vertices with the following node tree, from now on it will be assumed that the reader can add new nodes to the node tree:



A final step is to "spiralize" our polyline to avoid seams when 3D printing:

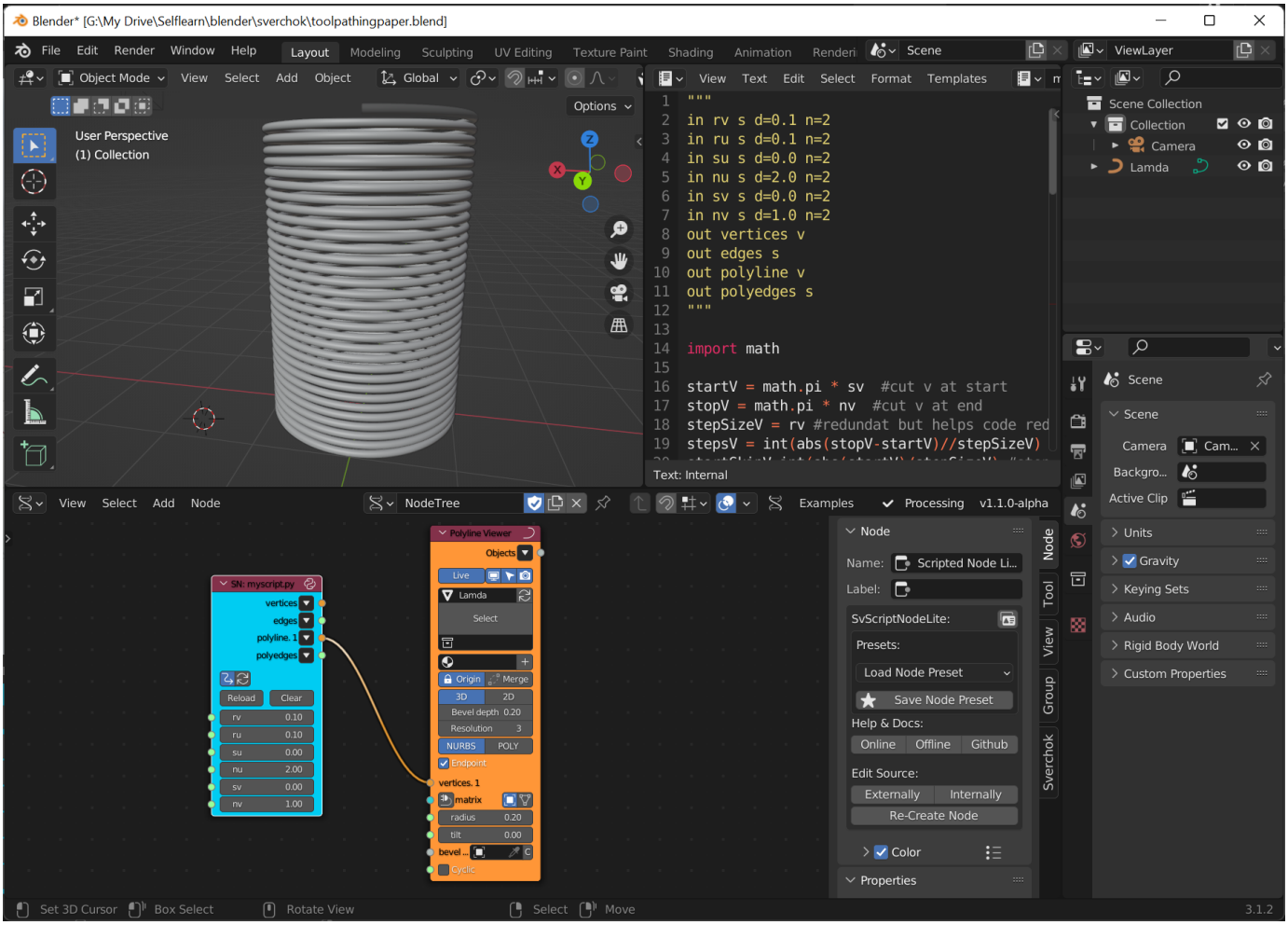
replace this line of code:

```
        z= v #Here we set a function for the Z value
```

with:

```
        if stepV == 0:
            z = v + (stepSizeV/2) #first layer will stay flat
        else:
            z = v + ((u*stepSizeV)/6.2831)-(stepSizeV/2) #divides the layer
height by the steps in the polyline on subsecuent layers
```

Now both outputs sockets vertices and polylines seem like the same output, but on the first one you will be able to select each "loop" separately, the second one will be a continuous polyline. Also note the different object quantity on the outliner.



For further learning on Sverchok nodes and data structure, the free e-book Learning Sverchok is a great resource. [@giachino_learning_2019]

Geometry from math

At this point we can start testing different mathematical functions in our script. The following parametric equations are described in the book Morphing by Joseph Choma:

" A parametric ecuation is one way of defining values of coordenates (x,y,z) for shapes with Parameters (u,v,w)."(Joseph Choma, 2015)[@choma_morphing_2015]

Replace the following lines for each test you make:

```
        x= math.sin(u) *(math.sin(u))
        y= math.cos(u)*(v)
                if stepV == 0:
            z = v + (stepSizeV/2) #first layer will stay flat
        else:
            z = v + ((u*stepSizeV)/6.2831)-(stepSizeV/2)
```

Make sure to keep the indetation or your script will return an error.

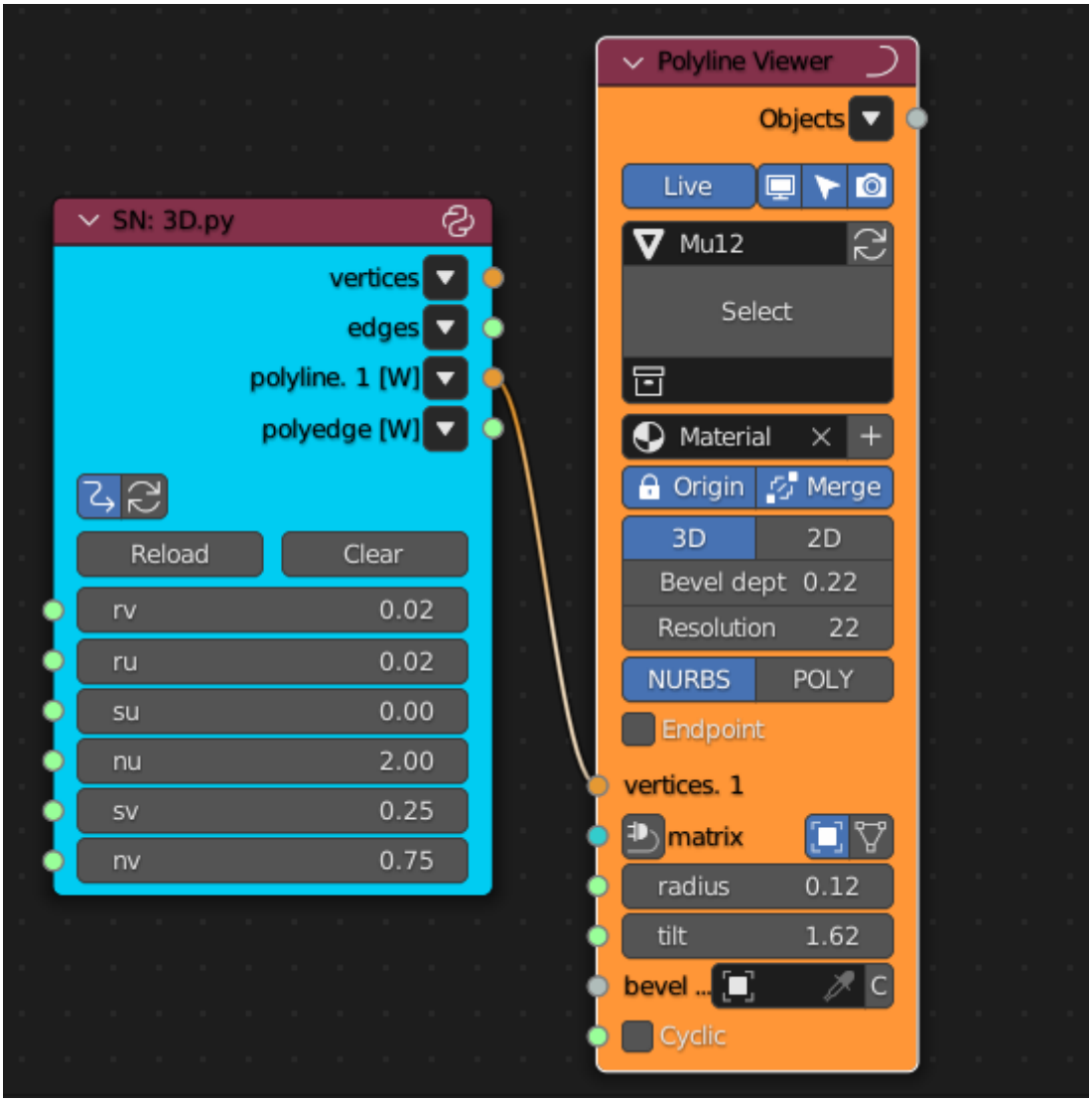
By pressing the shortcut `shift` + `enter` you can update the 3D view.

Test 001

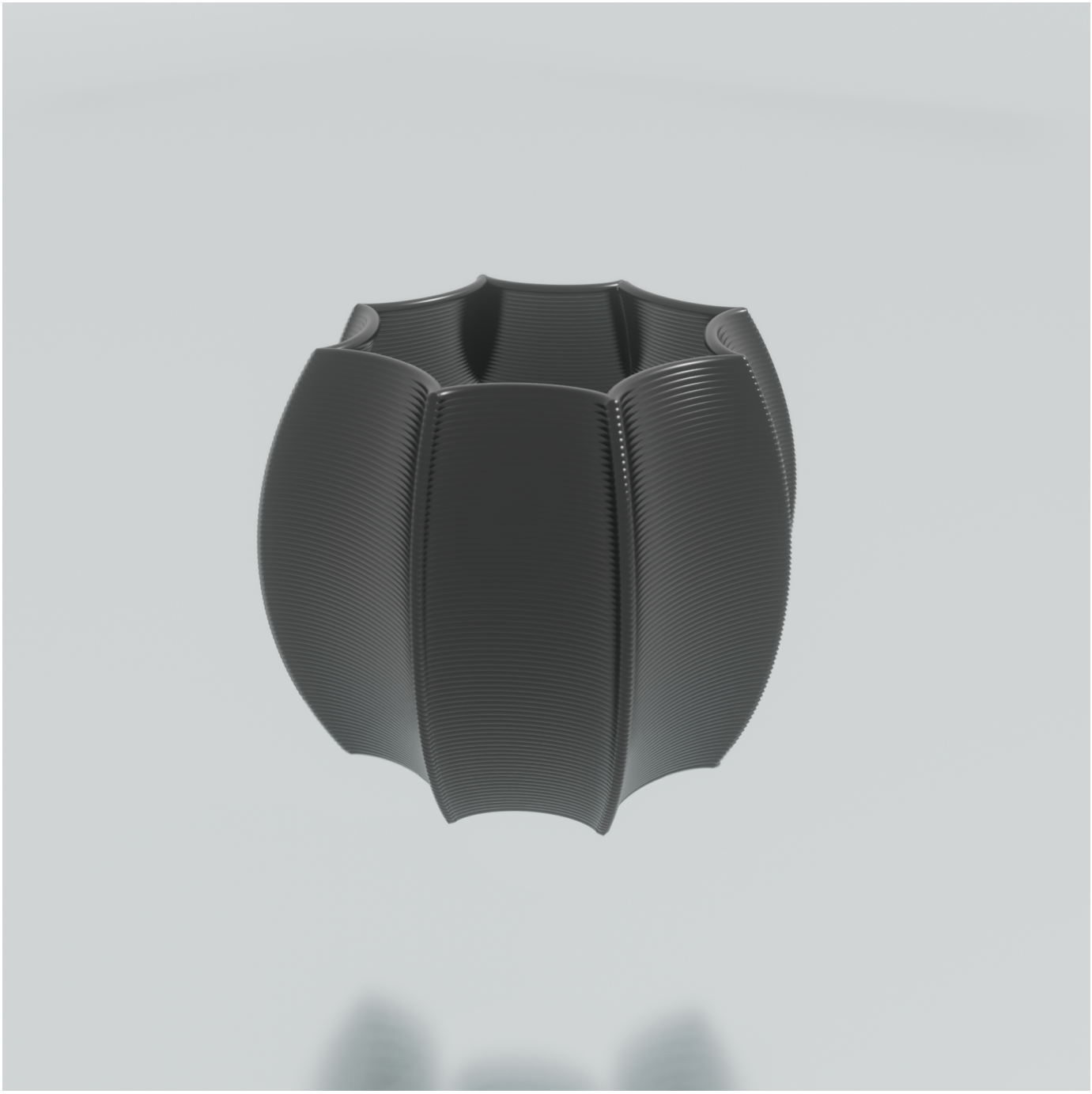
code:

```
        x=math.sin(math.sin(v) *  (math.cos(u) +
math.sin(7*u)/7))
        y= math.sin(math.sin(v) *  (math.sin(u) + math.cos(7*u)/7))
        if stepV == startSkipV:
            z= v + (stepSizeV/2)
        else:
            z= v + ((u*stepSizeV)/6.2831)
```

parameters:



render:

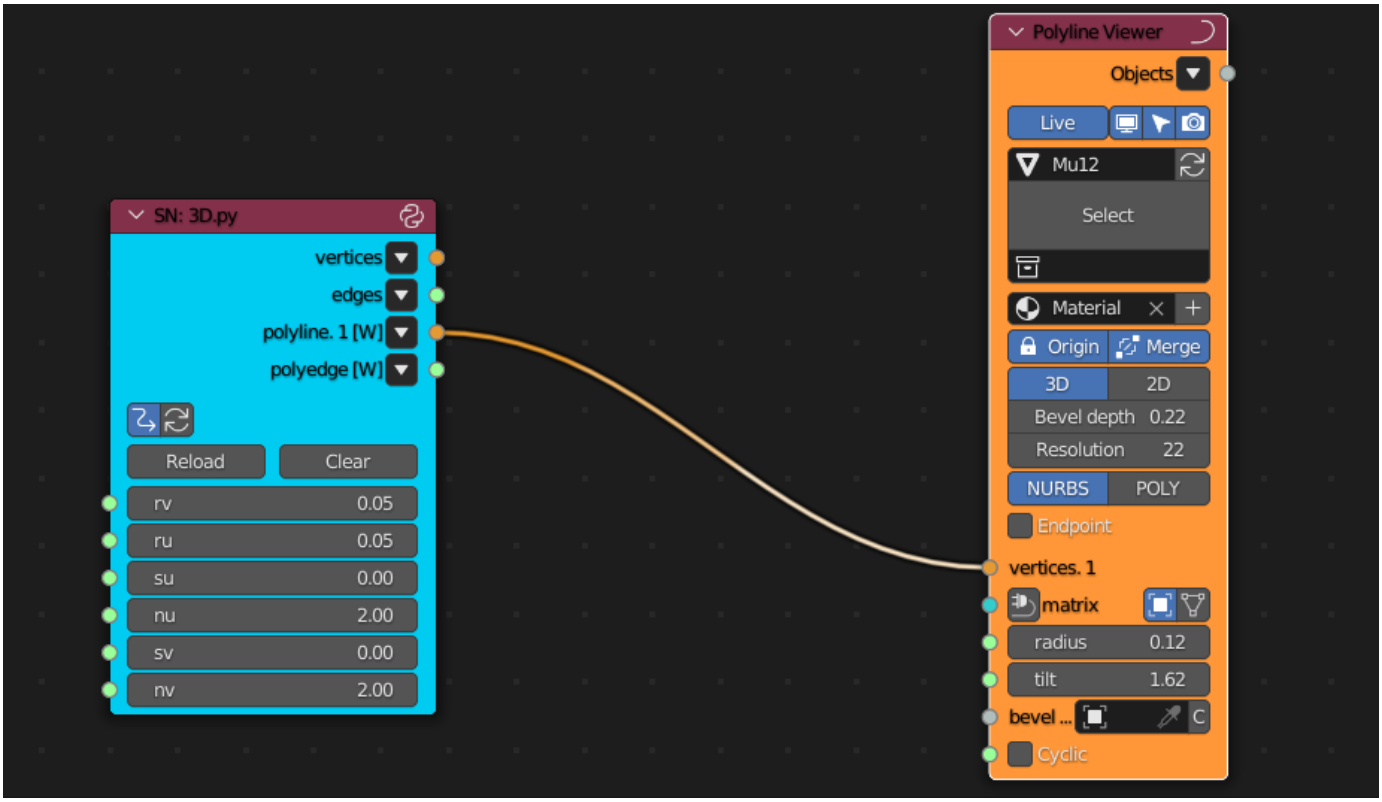


Test 002:

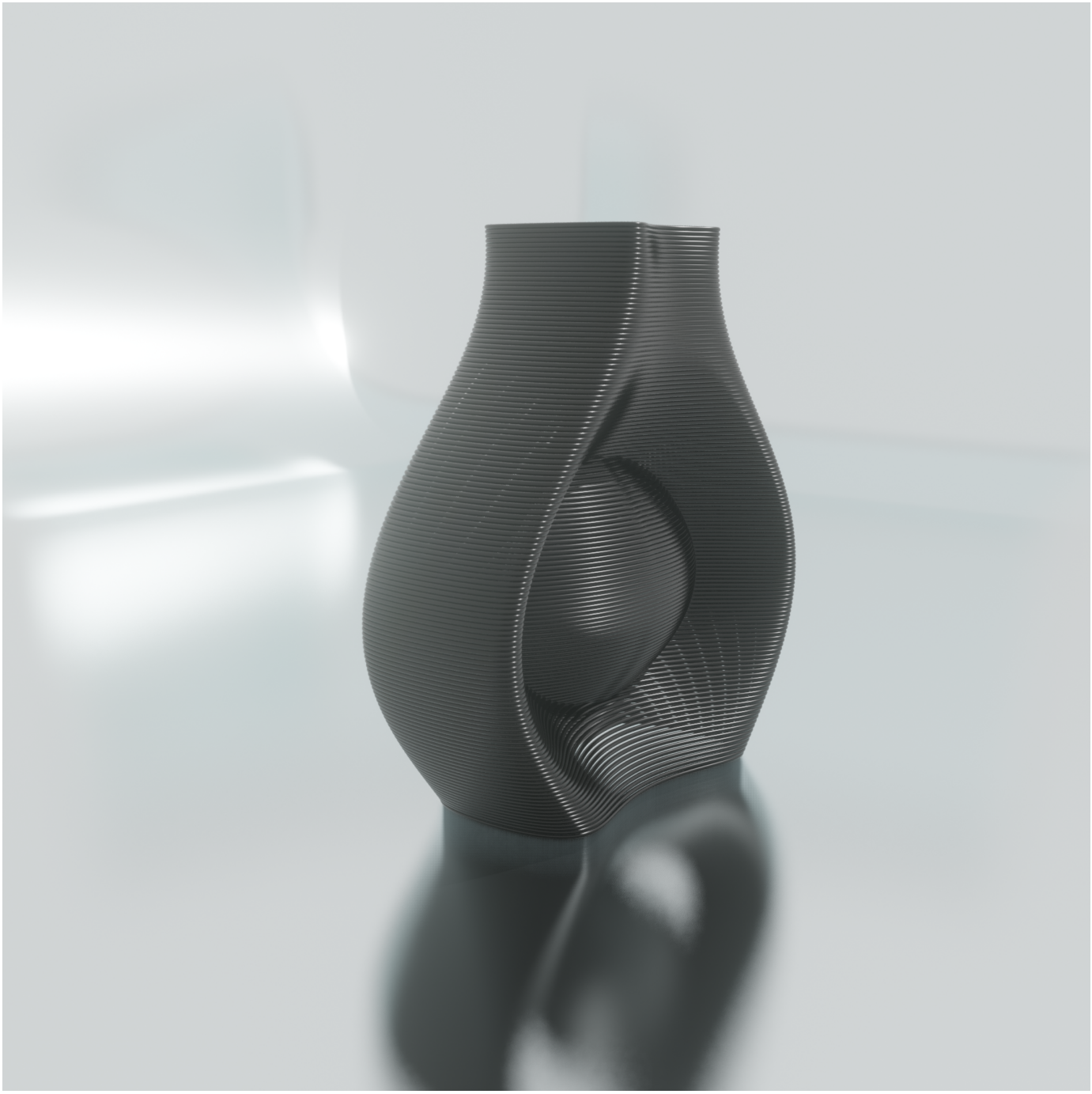
Code:

```
        x= ( 2+ math.sin(v*.75)) * (math.sin(u))
        y= math.sin((3+ math.sin(v-.75)) * (math.cos(u)))
        if stepV == 0:
            z= v + (stepSizeV/2)
        else:
            z= v + ((u*stepSizeV)/6.2831)
```

Parameters:


SN: 3D.py
vertices
edges
polyline. 1 [W]
polyedge [W]
Reload
Clear
rv 0.05
ru 0.05
su 0.00
nu 2.00
sv 0.00
nv 2.00
Polyline Viewer
Objects
Live
Mu12
Select
Material
Origin
Merge
3D
2D
Bevel depth 0.22
Resolution 22
NURBS
POLY
Endpoint
vertices. 1
matrix
radius 0.12
tilt 1.62
bevel ...
C
Cyclic

Render:

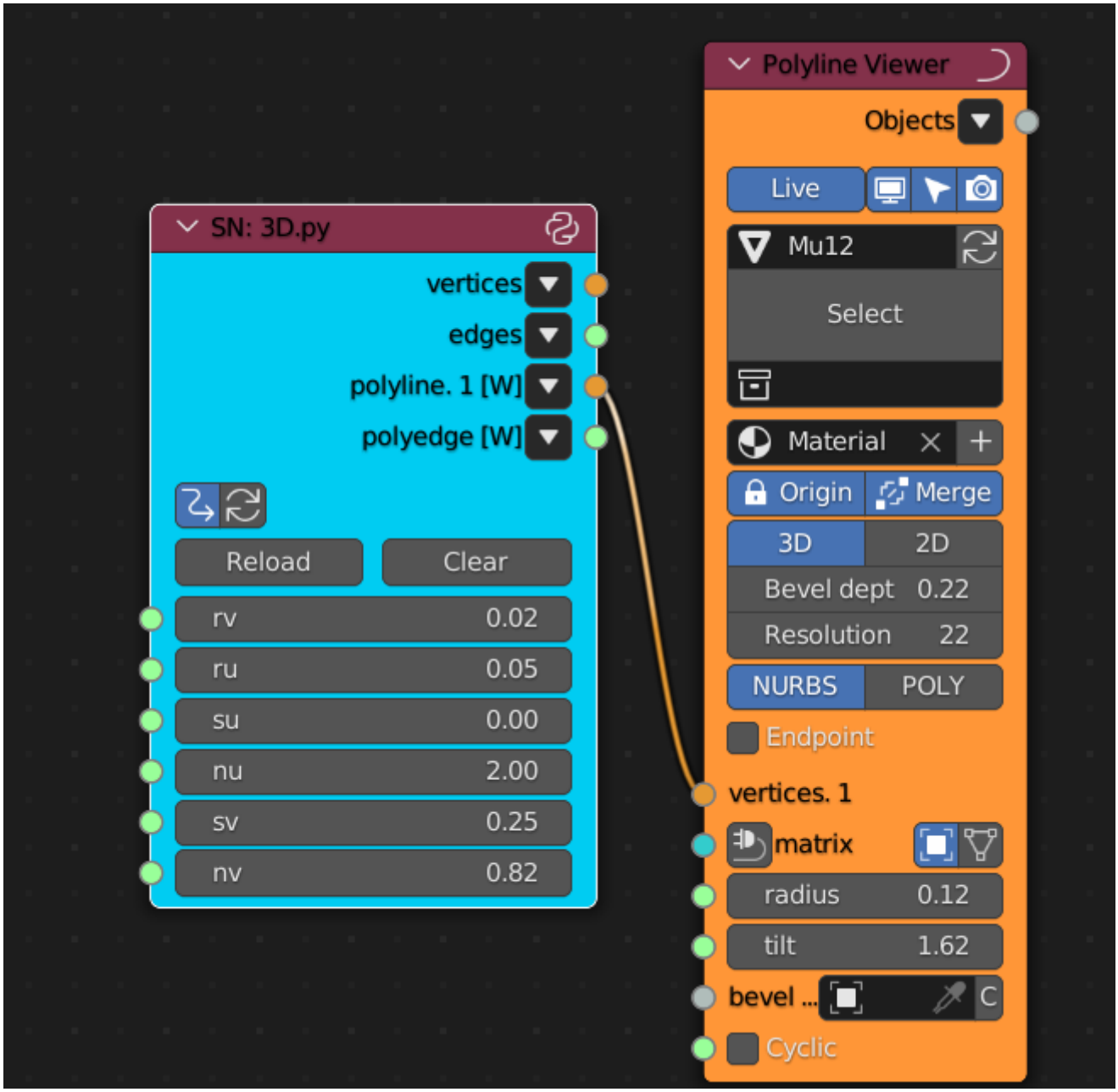


Test 003:

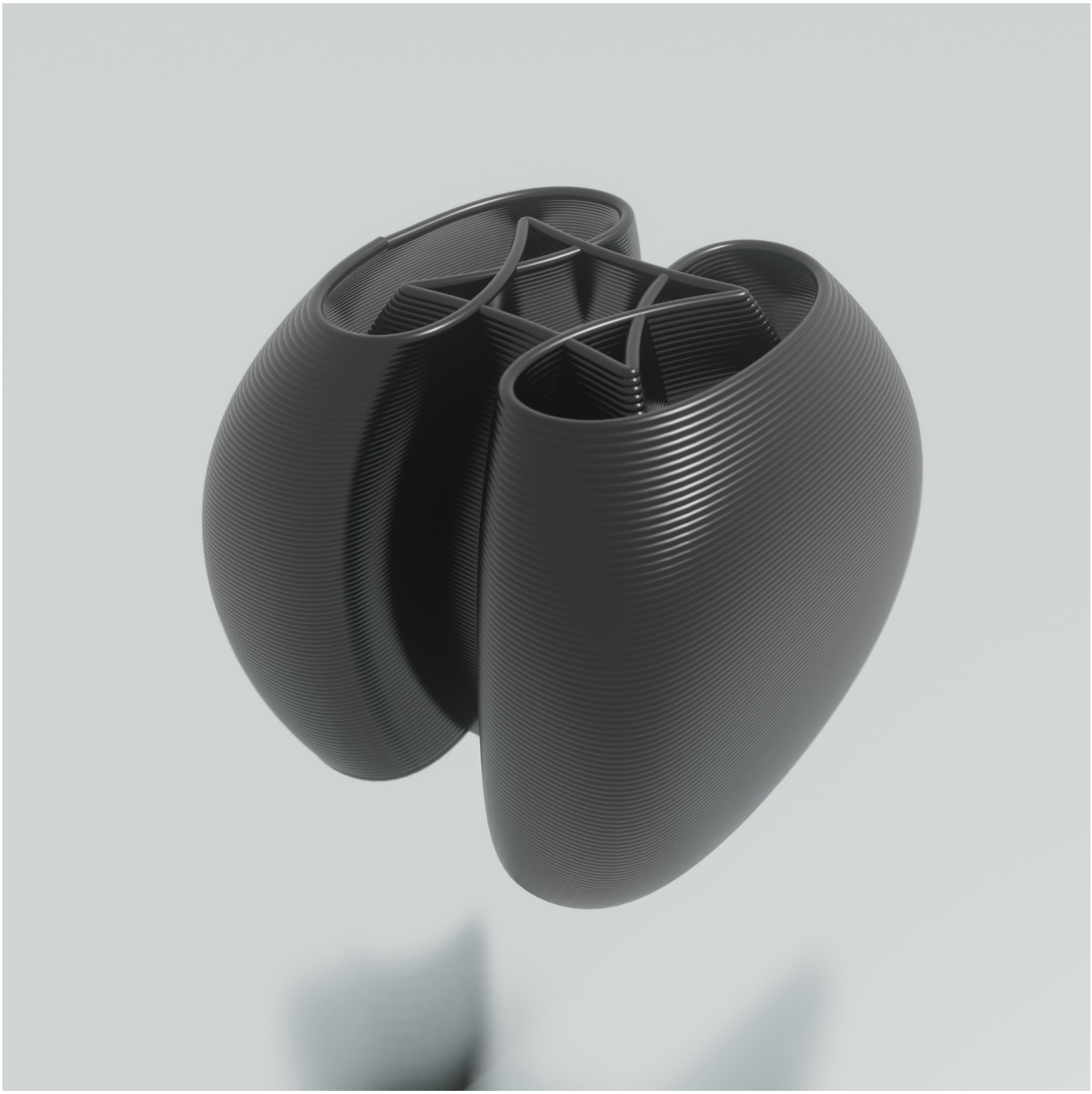
Code:

```
        x=math.sin(math.sin(v) *  (math.sin(u*3) + math.sin(5*u)/2))
        y= math.sin(math.sin(v) *  (math.cos(u) + math.cos(5*u)/2))
        if stepV == startSkipV:
            z= v + (stepSizeV/2)
        else:
            z= v + ((u*stepSizeV)/6.2831)
```

parameters:


Polyline Viewer
Objects
Live
Mu12
Select
Material
Origin
Merge
3D
2D
Bevel dept 0.22
Resolution 22
NURBS
POLY
Endpoint
vertices. 1
matrix
radius 0.12
tilt 1.62
bevel ...
C
Cyclic
SN: 3D.py
vertices
edges
polyline. 1 [W]
polyedge [W]
Reload
Clear
rv 0.02
ru 0.05
su 0.00
nu 2.00
sv 0.25
nv 0.82

Render:



Extending our script with python control flow

By implementing some control flow tools we are able extend control over the geometry, in this case we use an IF statement to change the value of Z based on our V iteration and the parameter r (radius), to change the direction of the loops. To achieve this:

1.- Add the following line to the directive:

```
in r s d=1.0 n=2 # this creates the radius parameter input socket
```

2.-Add the following line after the variable declaration and before our main loop starts:

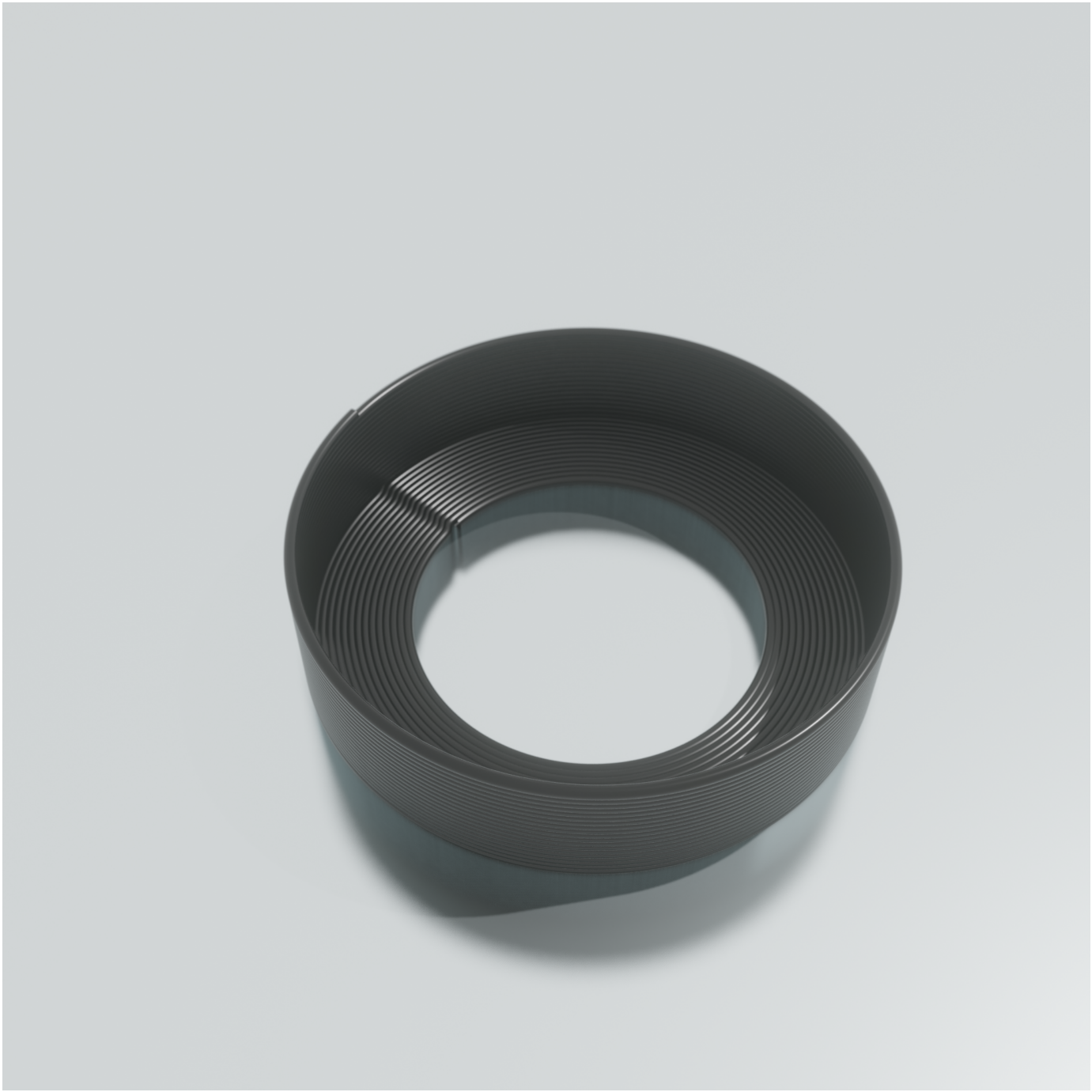
```
baseLoops= r/stepSizeV # this calculates the number of v loops before
changing direction with the z value
```

3.-Change the x,y and z declaration lines to the following code:

```
        if stepV < baseLoops:
            x= math.sin(u)*v #Here we set a function for the X value #scales
the x value times the V loop
            y= math.cos(u)*v#Here we set a function for the Y value #scales
the y value times the V loop
            z=0 #makes a flat base for our cup
        else:
            x= math.sin(u)*r #Here we set a function for the X value
            y= math.cos(u)*r#Here we set a function for the Y value
            z = (v  + ((u*stepSizeV)/6.2831)-(stepSizeV/2)) -
stepSizeV*baseLoops #this changes the direction on Z
```

Notice that we can control parametrically:

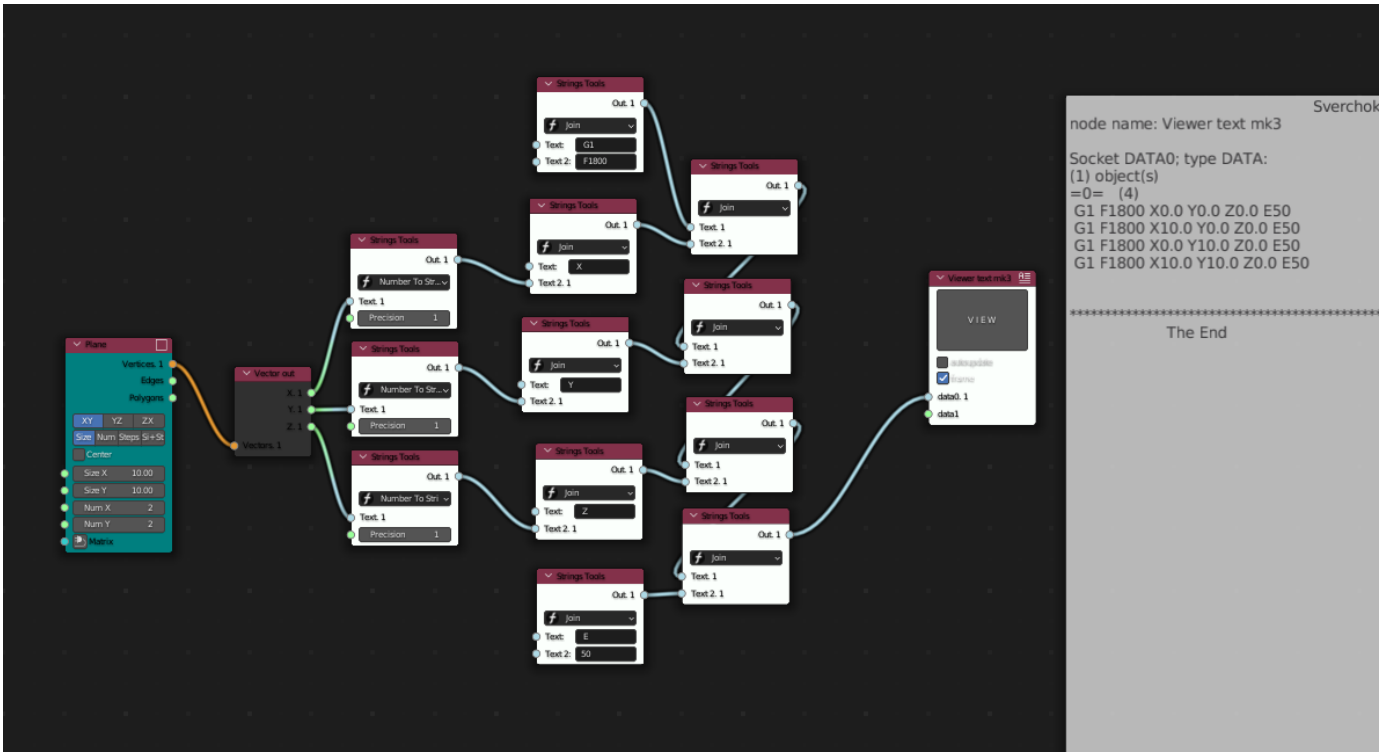
- Radius of our object (`r`)
- Distance between "loops" (`rv`)
- Resolution of the cylinder (`ru`)
- Inner hole radius (`sv`)
- Heigh(`nv`)
- "Cut" our cylinder (`su`,`nu`)



Bu using control flow tools like loops, conditionals, etc. Its possible to further cutomize toolpathing. Some ideas to explore further are patterning, attractors, custom subdivisions, recursion, etc.

Parsing into GCODE

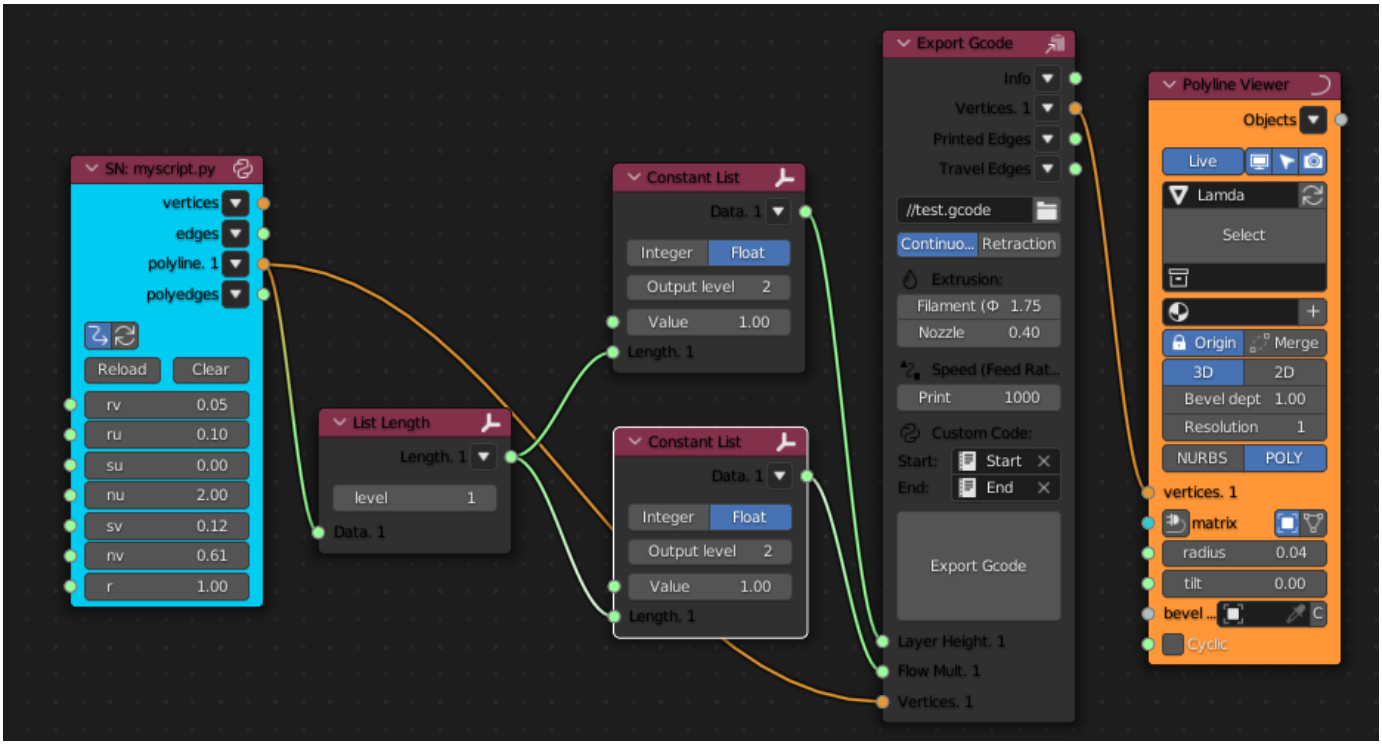
A recurrent issue with visual programming languages is the visual chaos that can arise from complex node trees. In this case the following node tree allows us to make a simple gcode file from vertices :



The same can be acheved with just a couple lines of Python:

```
for i in v:
    print("G1 F1800 x{x} y{y} z{z} e50".format(x = i[0], y = i[1],z = i[2]))
```

Of course this can be expanded to include speed, extrusion, start and end gcode, with parametric inputs but Sverchok provides a Gcode export node that does this for us:

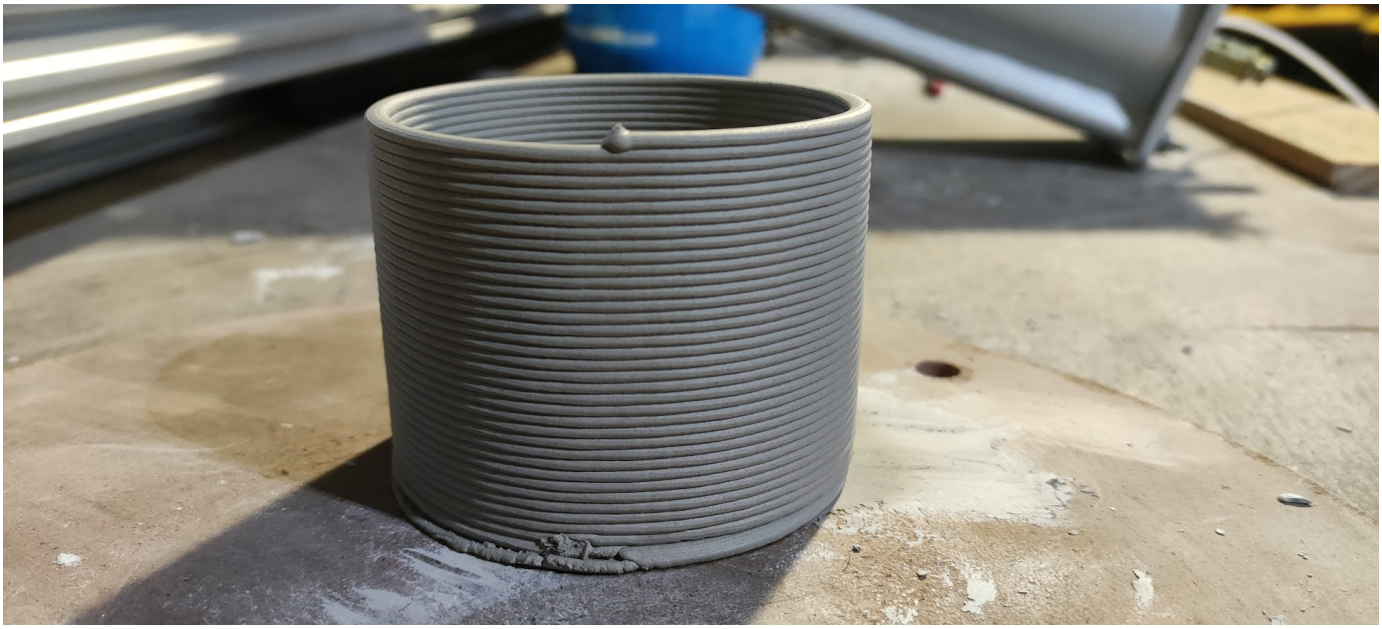


Note that the list length and constant list nodes can be avoided by generating the output sockets from our scripted node. In this case we are sending a list of a constant value.

With three Nodes we can achieve to generate geometry, parse the gcode and visualize our object.

Results

Some of the Objects 3D printed with this method:



Next Steps

- Extend toolpath output sockets for variable flow and speed.
- Connectivity and remote control.
- Implement 4th,5th and 6th axis rotation for generating more complex toolpaths including tangential printing and parsing robotic arm languages.
- Extend post processing capabilities.

Conclusion

While this workflow provides a flexible and creative approach to develop toolpaths , it is not intended to replace current workflows. It is a tool for exploring and iterating "non-traditional" toolpaths. It can provide benefits over traditional methods for creative applications.