

# **Randomised Pipe Generation**

**50.017: Graphics & Visualization**  
**Final Presentation**  
Team 1

Leong Keng Hoy 1005164  
Kim Si Eun 1005370  
Jowie Ng 1005494  
Sharryl Seto 1005523

# Problem

- Existing pipe generation algorithms are not random nor automated.
  - require **manual** curve/polyline placements
- Current procedural algorithms do not take into account generated meshes **colliding** with each other.

How can we create a robust, systematic, and randomised **pipe generator** with multiple control variables such as *density, number of layers and type of pipes?*

- ✓ Practical, saves time and effort

# Approach

*Pipeline:*

Explore Blender's **Geometry Nodes** and **Python API**

Create basic algorithms to generate pipes using Geometry Nodes (L-pipes, modular grid)

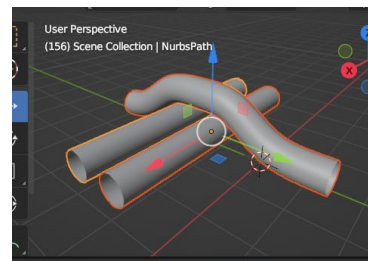
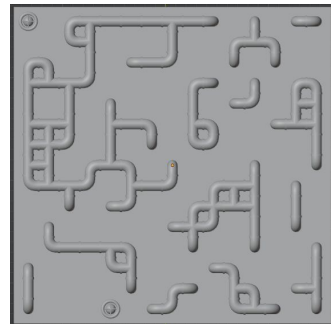
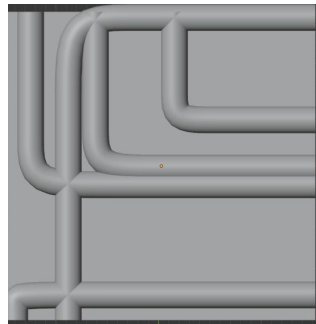
Create Python script to prevent collisions and intersections

Refine our algorithms -> different variations!

Total: 4 **procedural**  
algorithms

Add Blender UI to facilitate user interaction

Include other variables such as material of pipe



# Implementation

## Tools:

- Blender 3.4.1
  - Geometry Nodes (node-based programming)
  - Python API

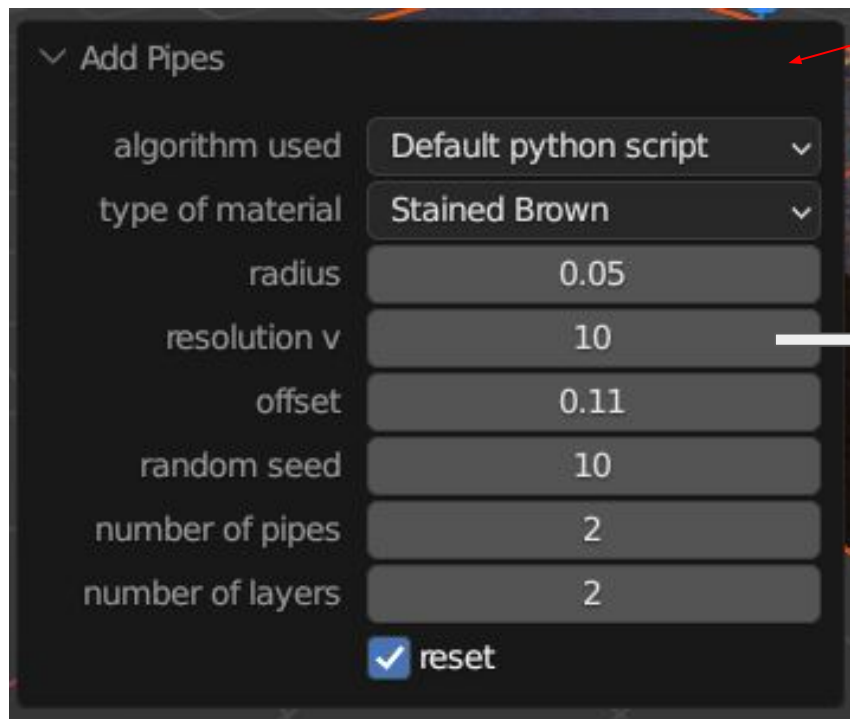


## Input Data and Preparation

- All raw data comes from user input via the interface
- Error checking and exceptions

# Interface

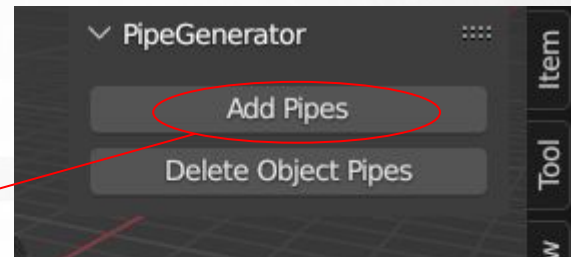
User can input these parameters:



▼ Add Pipes

algorithm used	Default python script ▼
type of material	Stained Brown ▼
radius	0.05
resolution v	10
offset	0.11
random seed	10
number of pipes	2
number of layers	2

☒ reset



*Python / Geometry Nodes* ————— uses UI interface on right panel

*Material from the .blend file*

*E.g., if =4, pipe cross-section = square. > 8 looks circular*

*How far you want the pipes to be from the object*

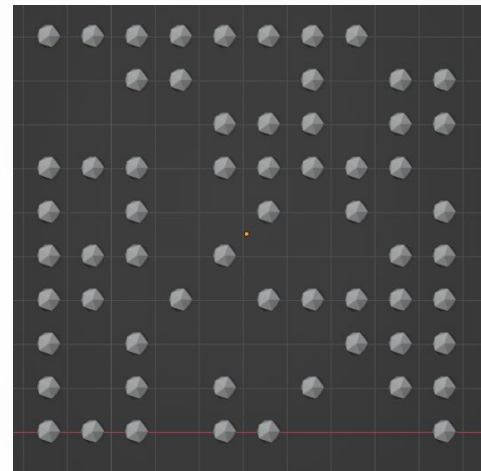
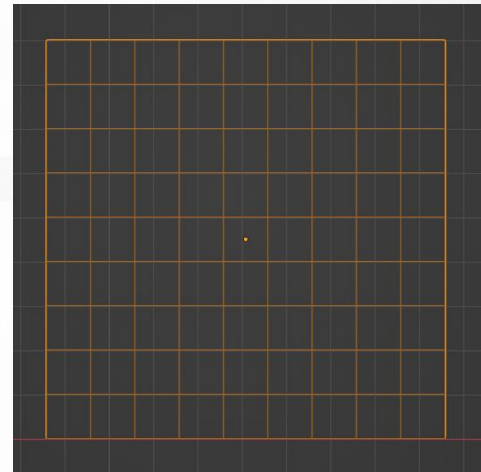
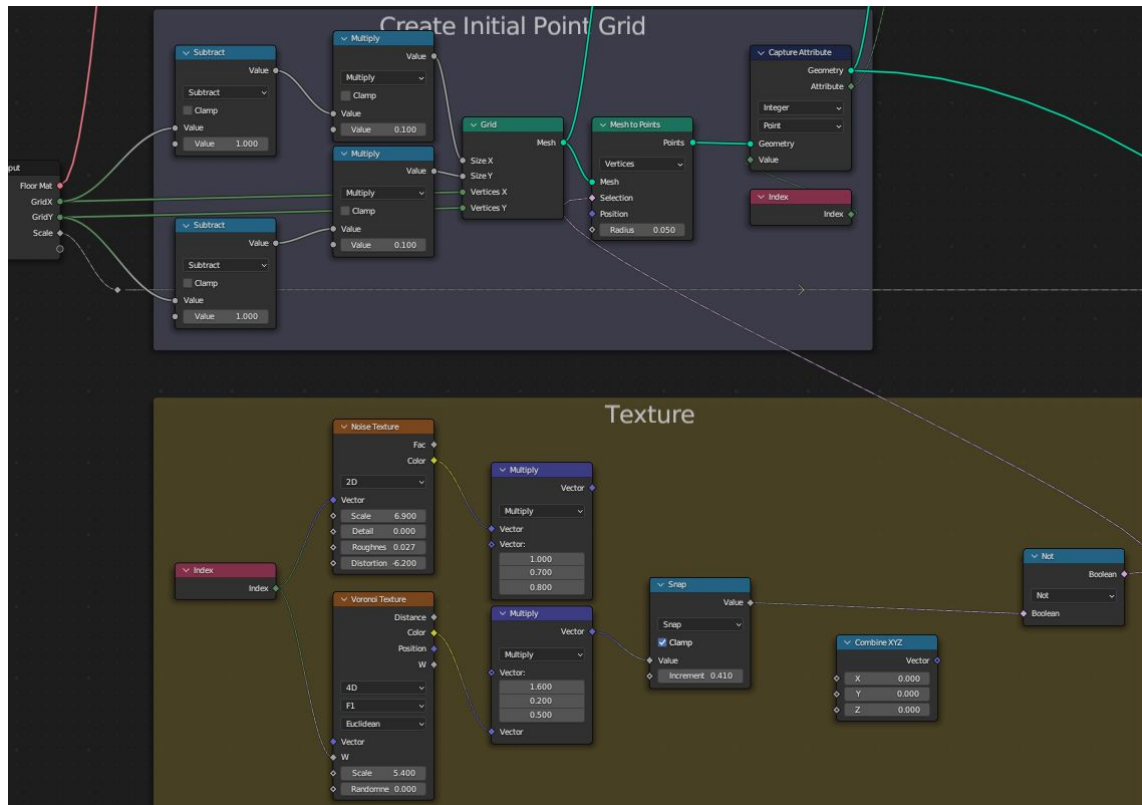
*Randomiser*

# Results

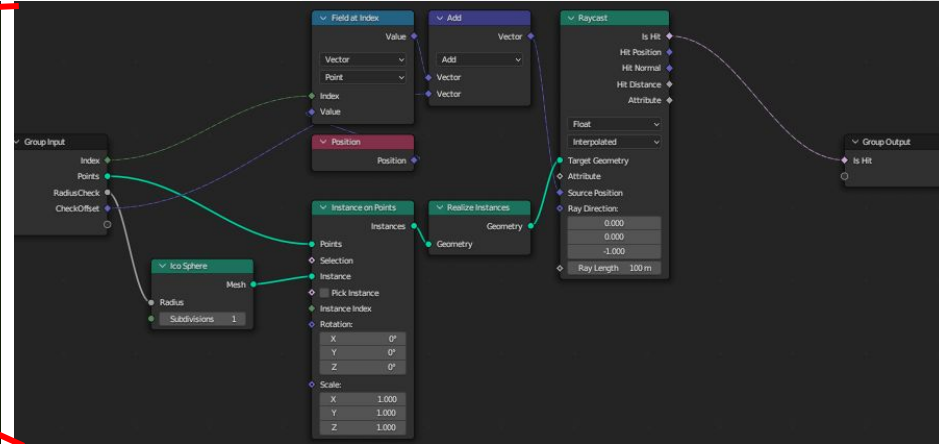
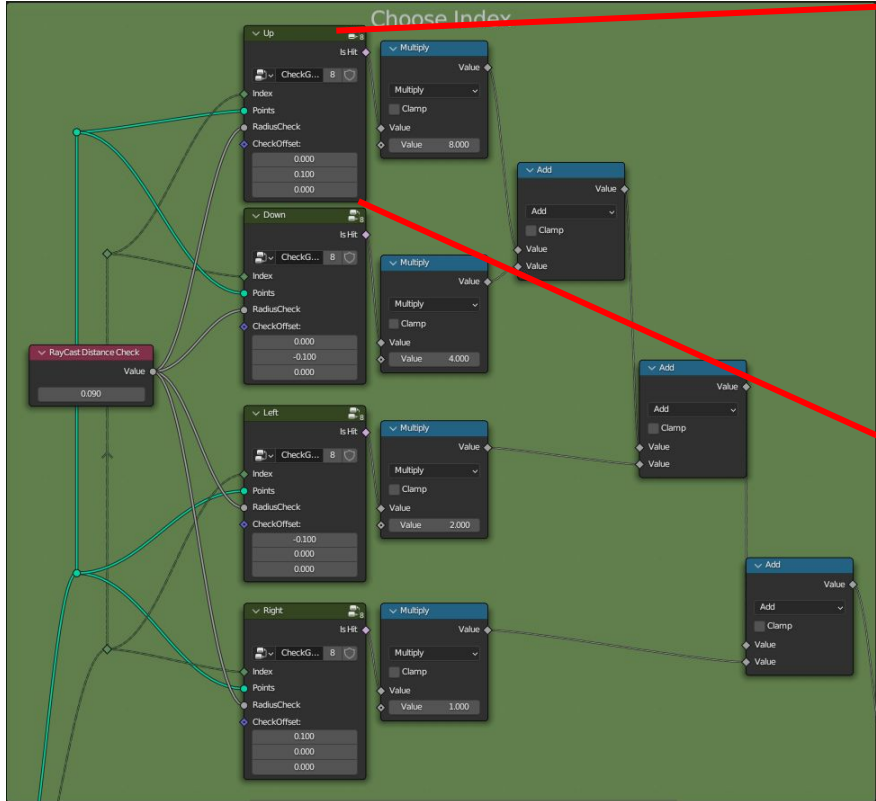
## Geometry Nodes - Grid



# Results - Geometry Nodes (Grid)

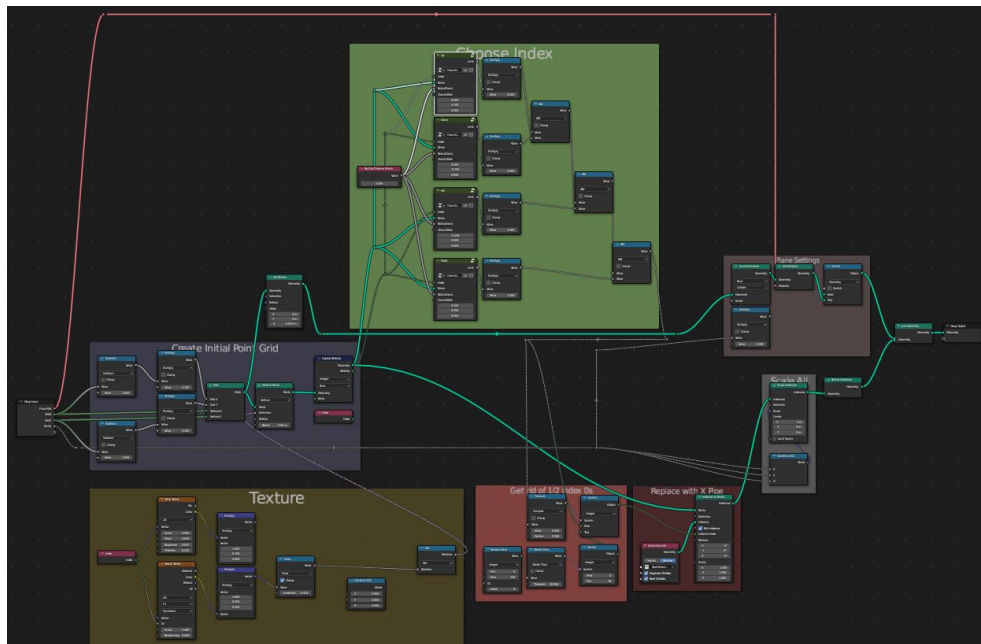


# Results - Geometry Nodes (Grid)





# Results - Geometry Nodes (Grid)



Get Max Length of Obi

Bounding Box

Bounding Box

Min

Max

Geometry

Subtract

Vector

Subtract

Vector

Vector

Separate XYZ

X

Y

Z

Vector

Maximum

Value

Maximum

Clamp

Value

Value

Maximum

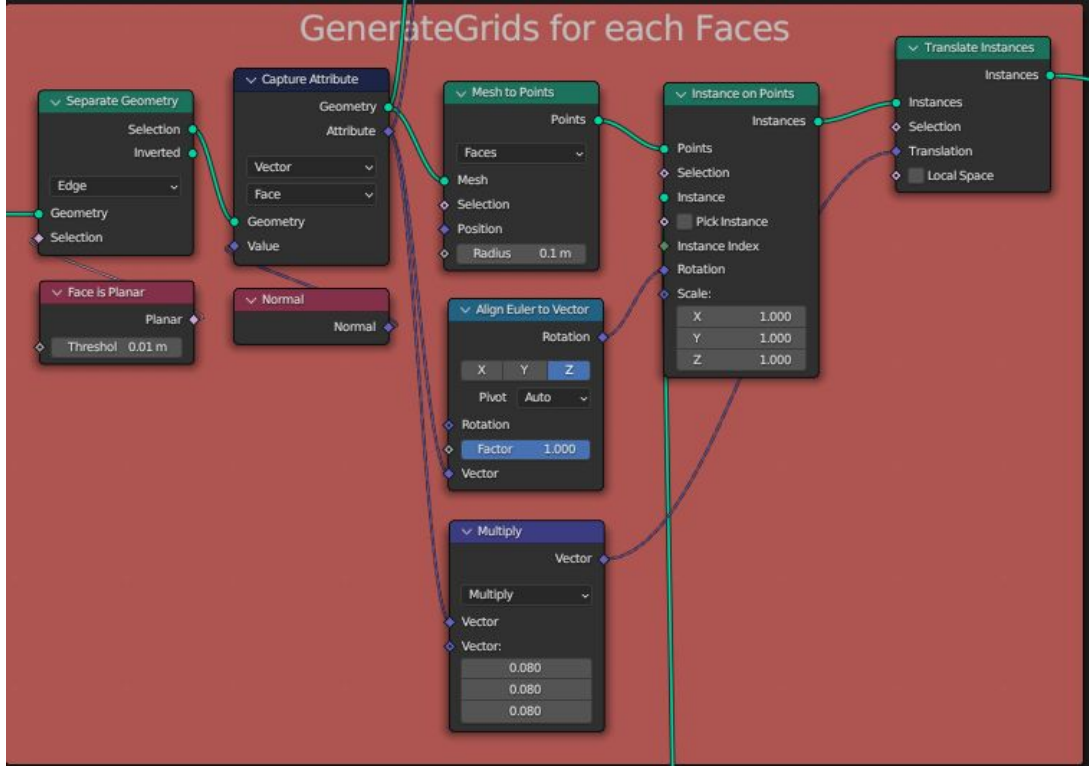
Value

Maximum

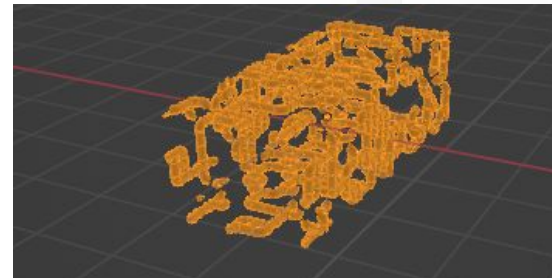
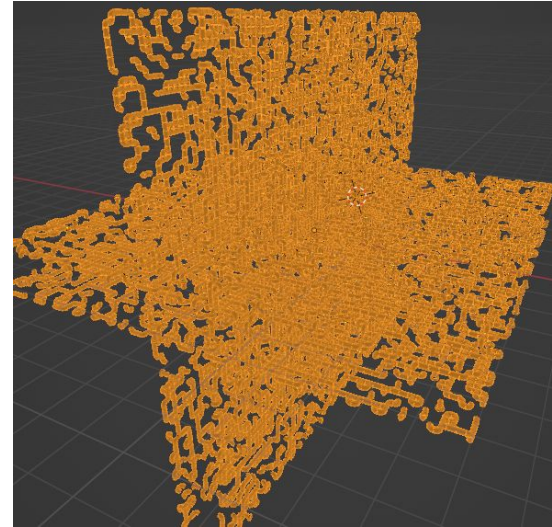
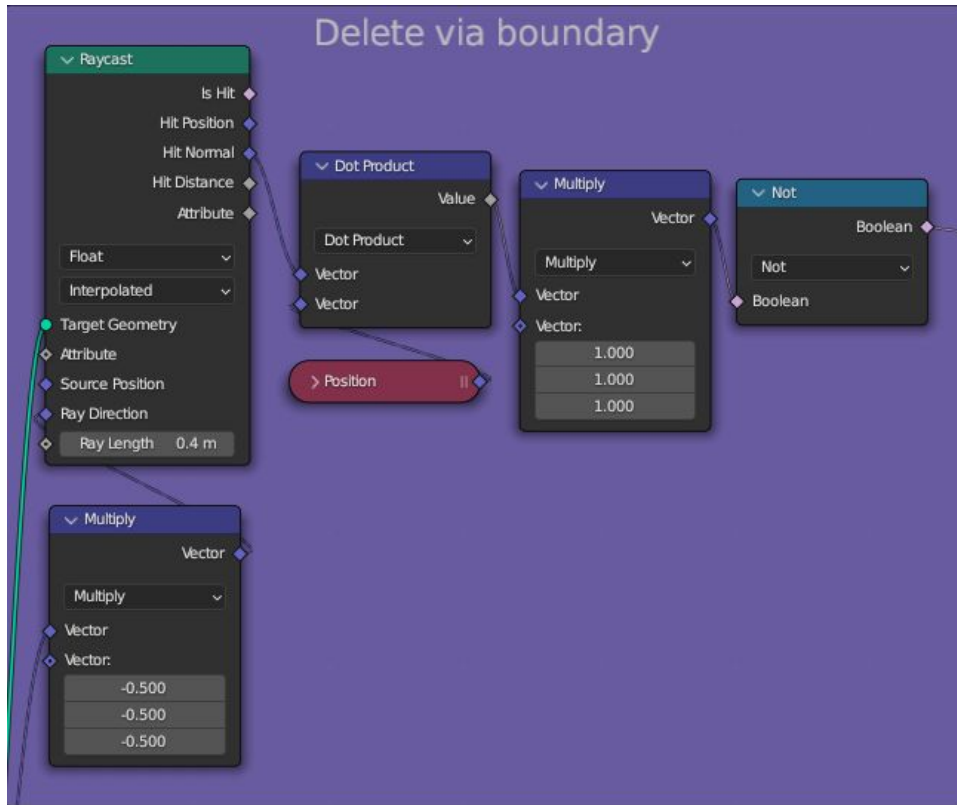
Clamp

Value

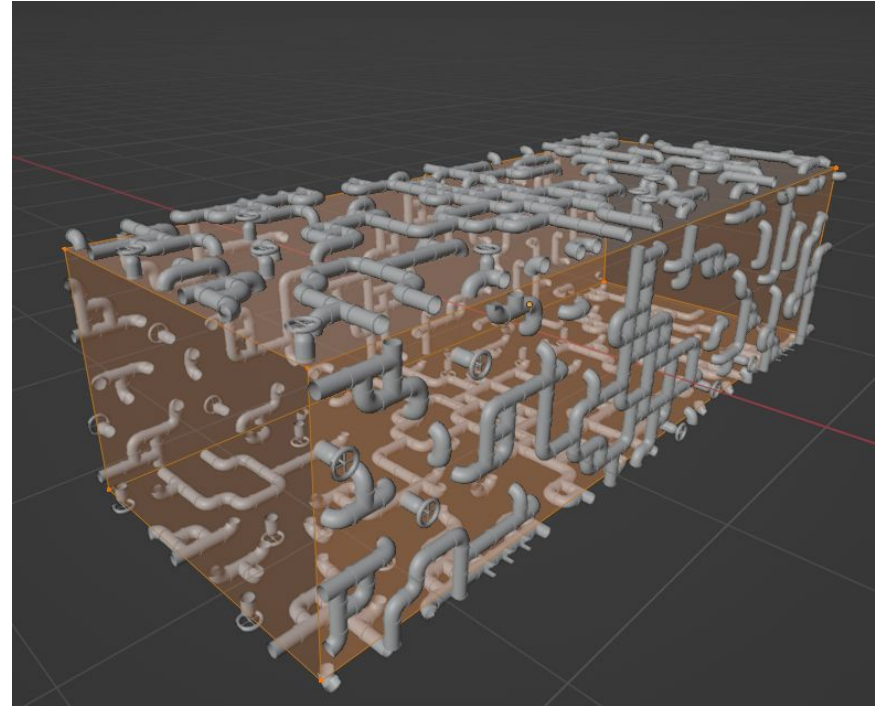
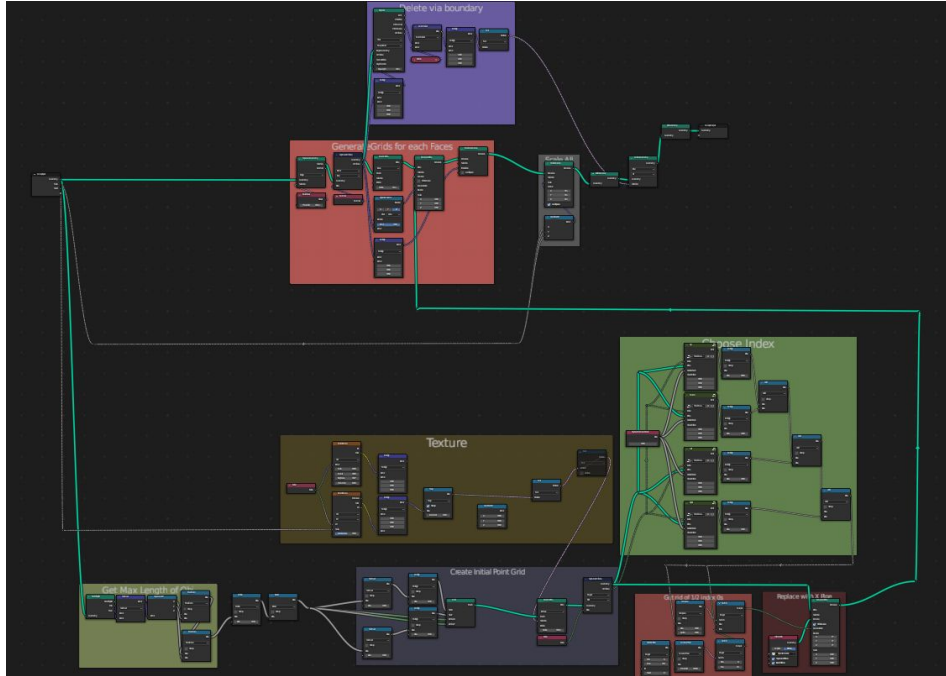
Value



# Results - Geometry Nodes (Grid on Faces)

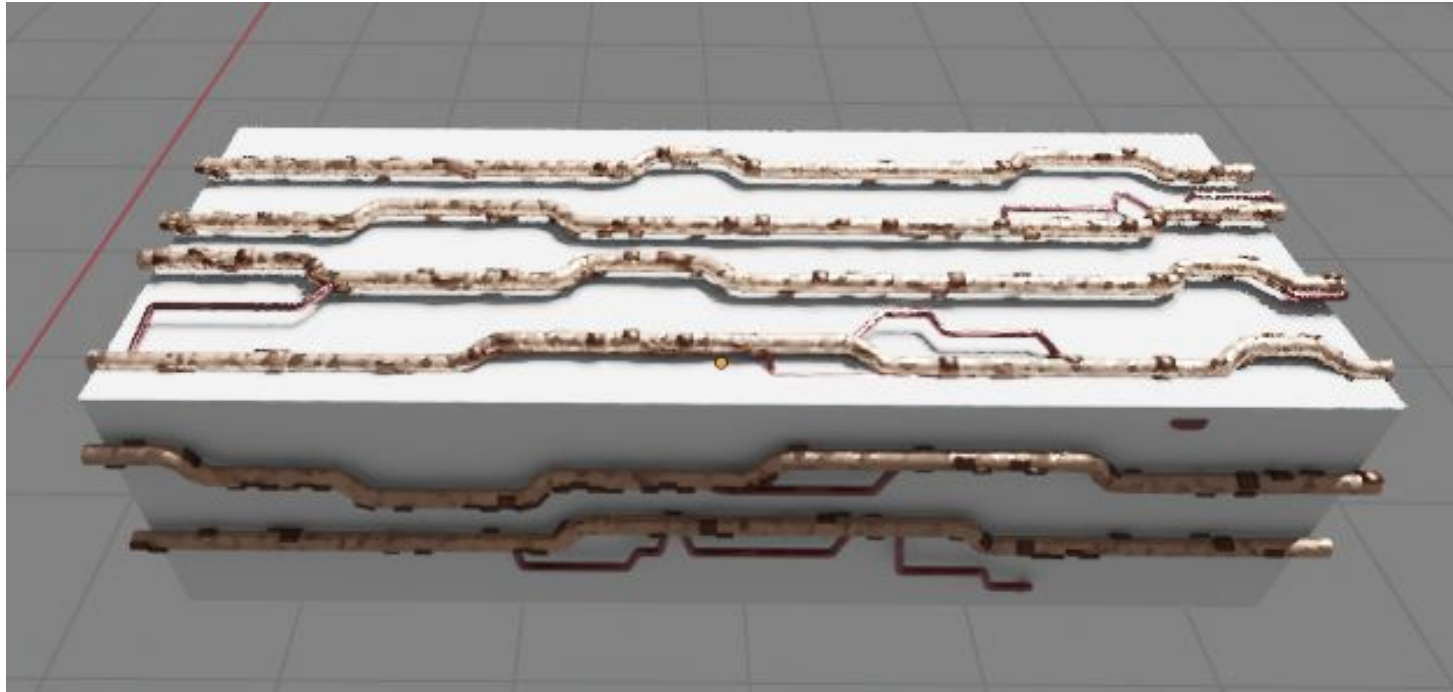


# Results - Geometry Nodes (Grid on Faces)



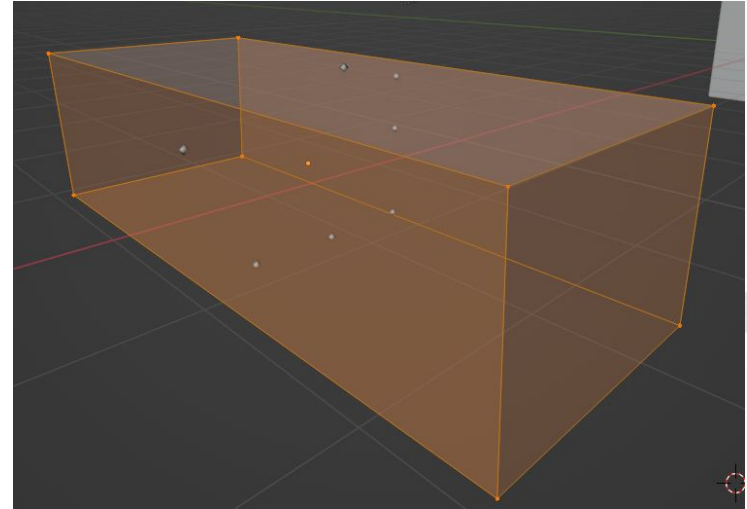
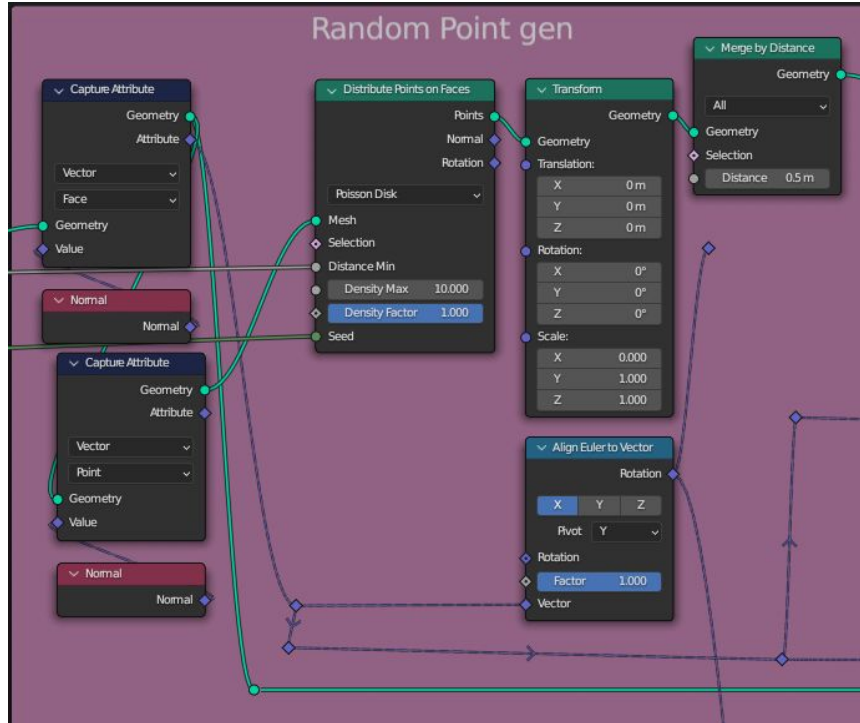
# Results

## Geometry Nodes - Wall Pipes

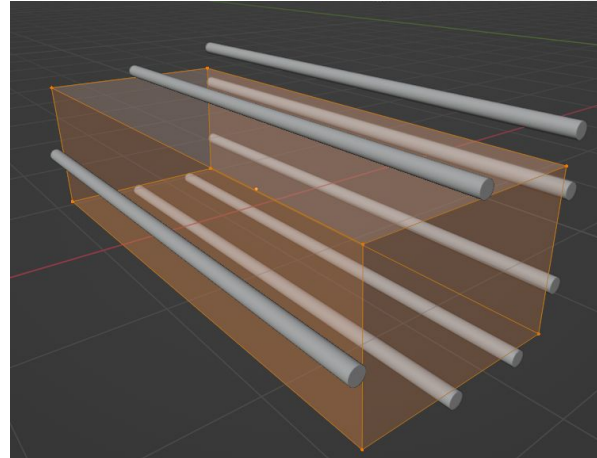
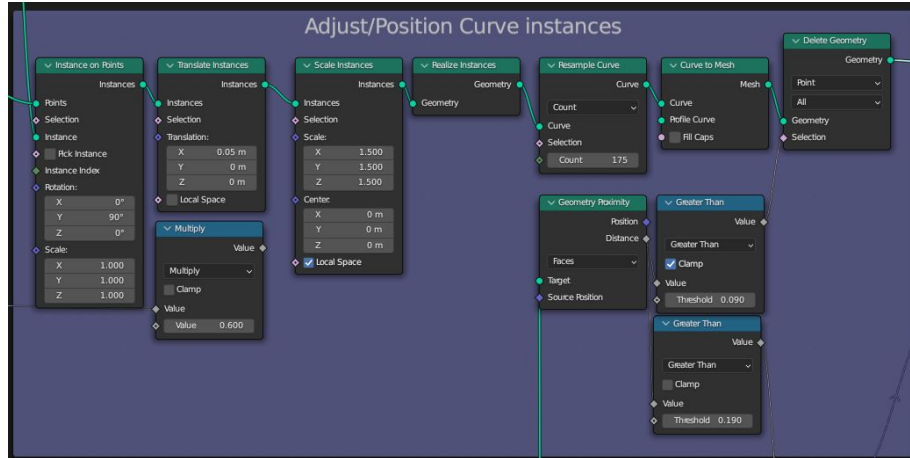
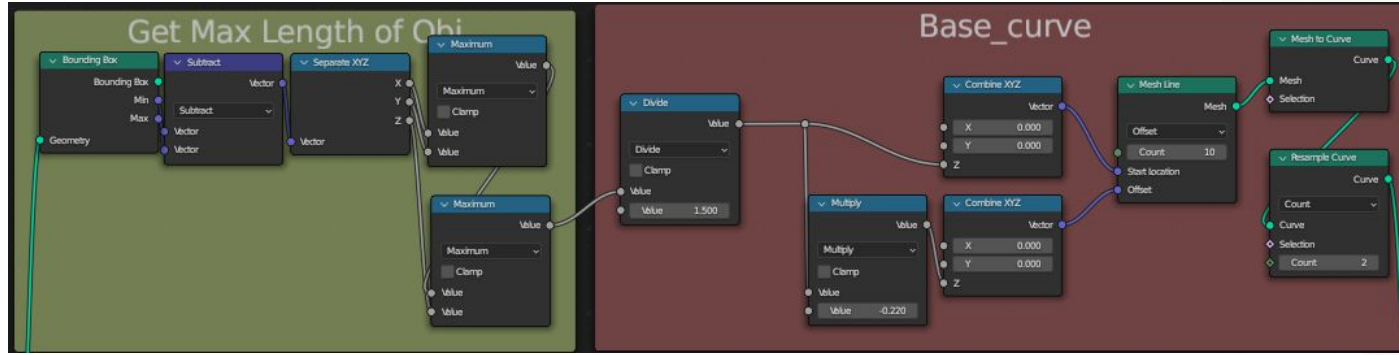




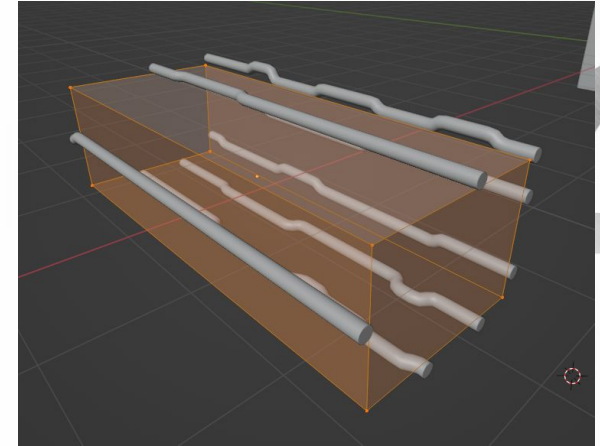
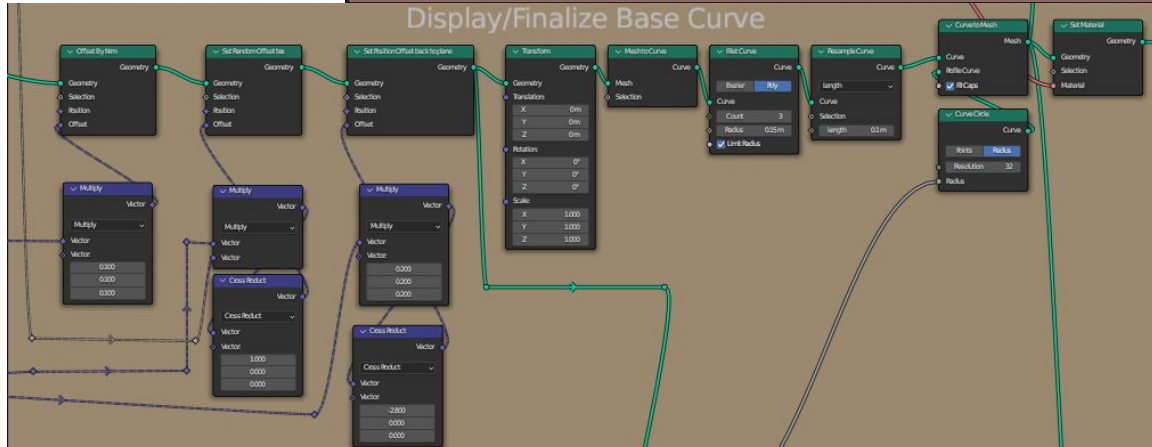
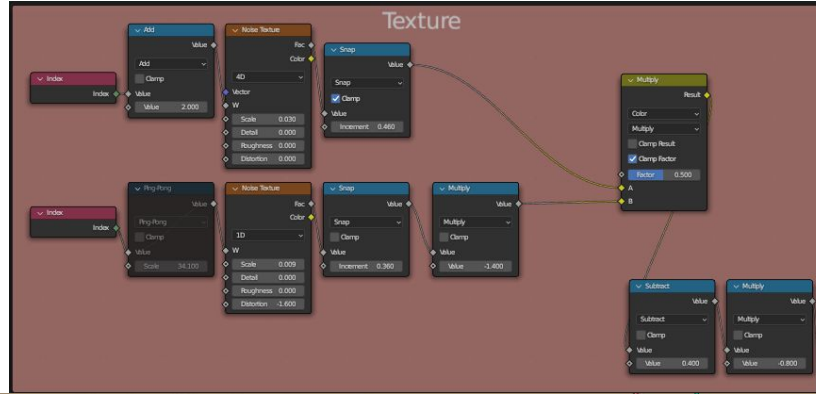
# Results - Geometry Nodes (Wall Pipes) *[points]*



# Results - Geometry Nodes (Wall Pipes) *[straight curve]*

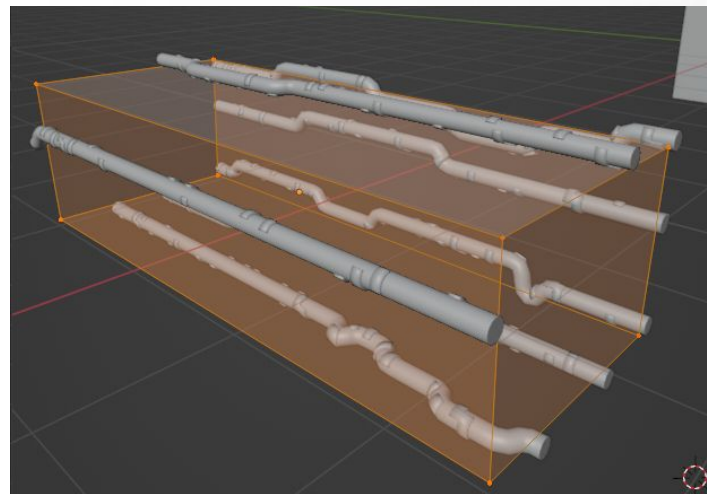
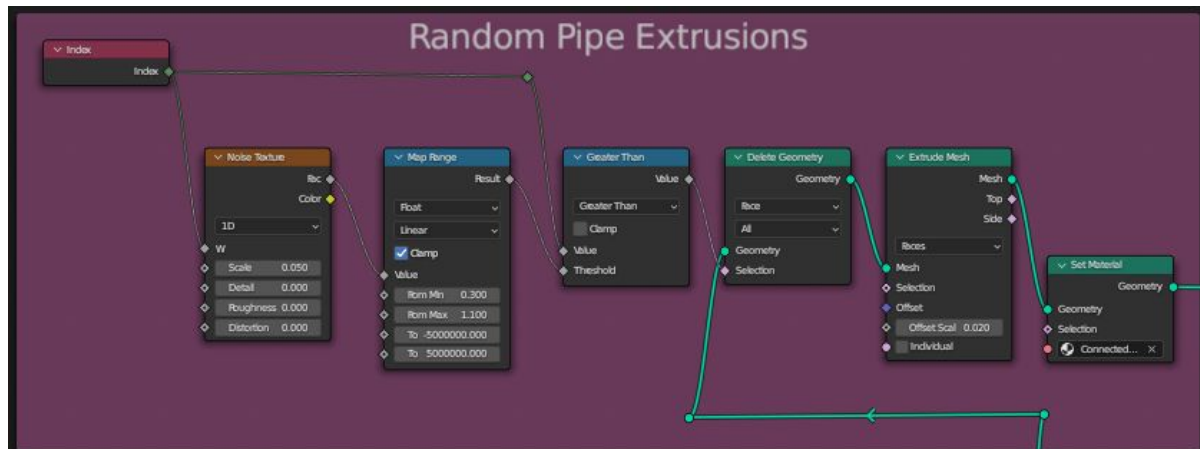


# Results - Geometry Nodes (Wall Pipes) *[rand offset]*

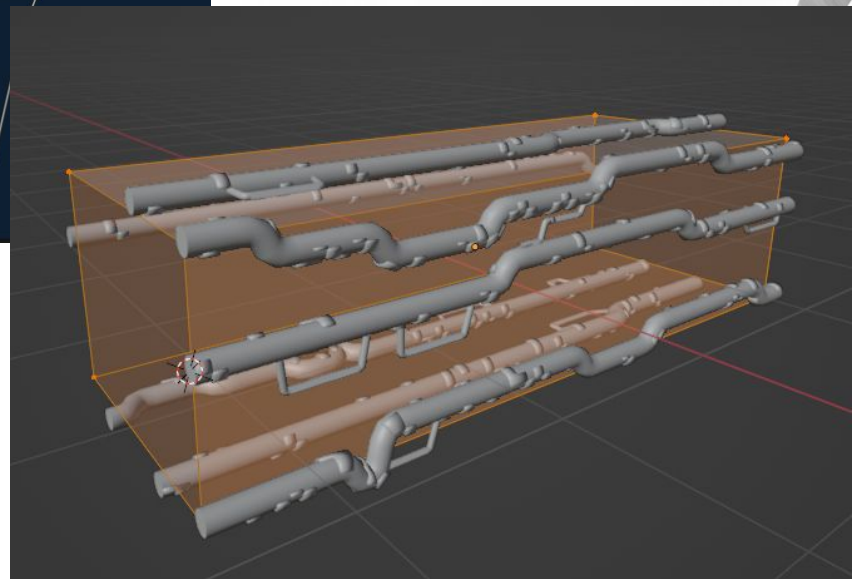
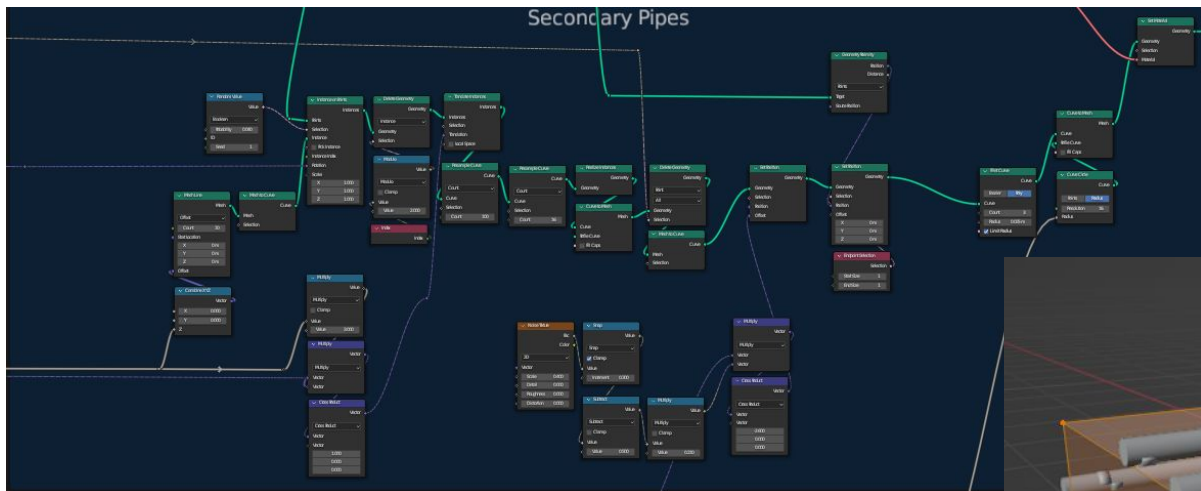




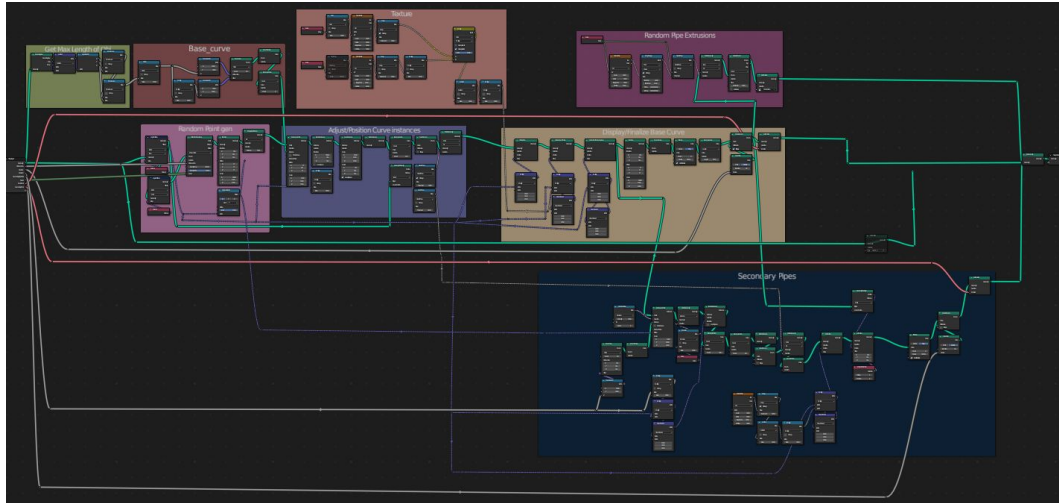
# Results - Geometry Nodes (Wall Pipes) *[rand extrude]*



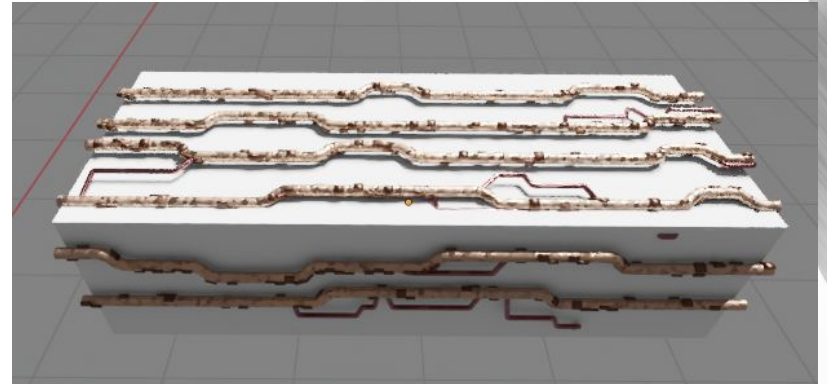
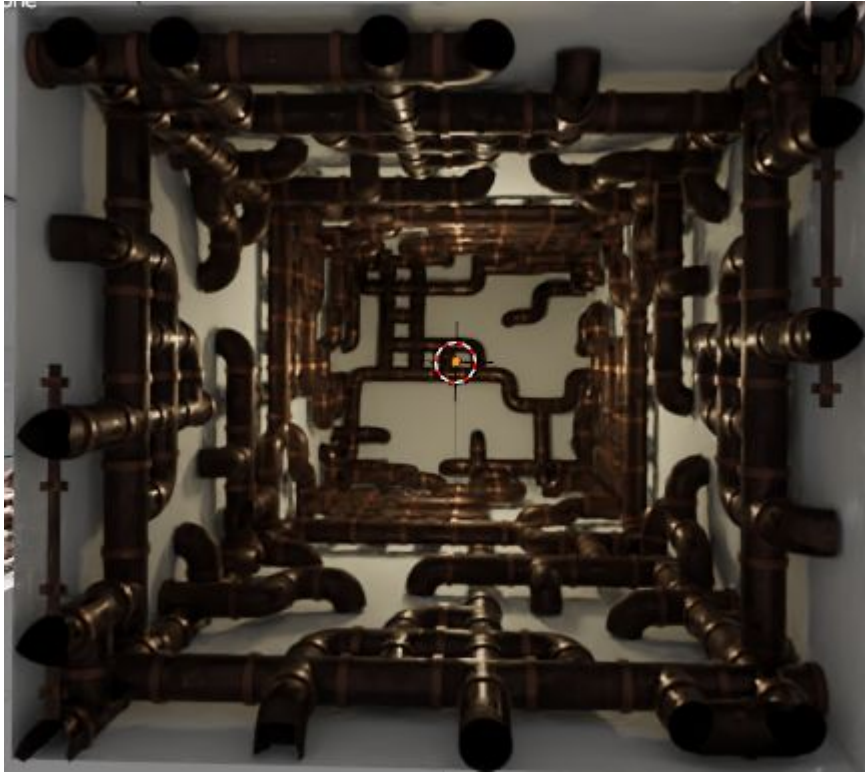
## Results - Geometry Nodes (Wall Pipes) *[secondary pipes]*



# Results - Geometry Nodes (Wall Pipes)

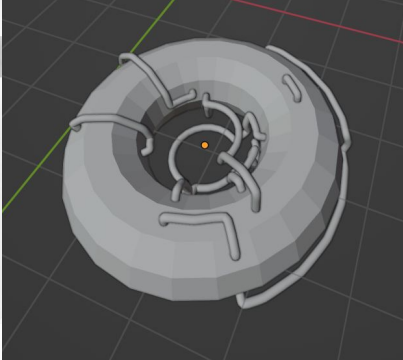


# Results - Geometry Nodes

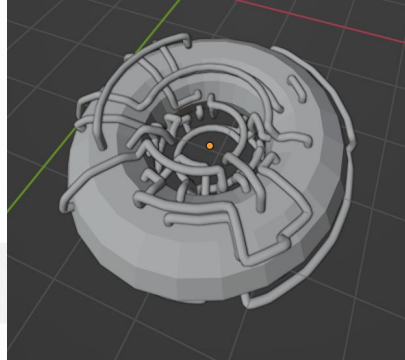


# Results

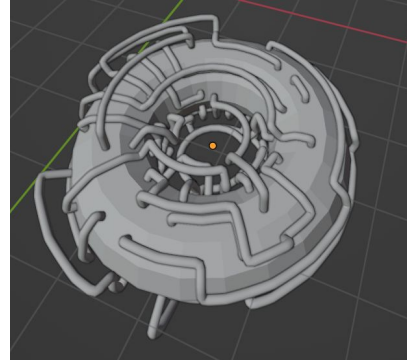
## Python Script



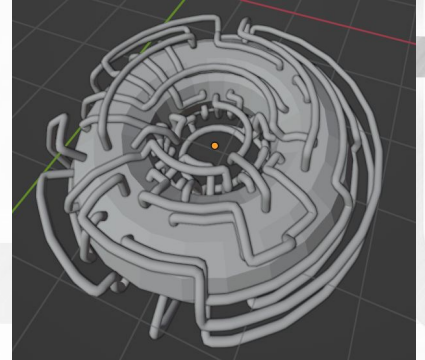
10 pipes



20 pipes



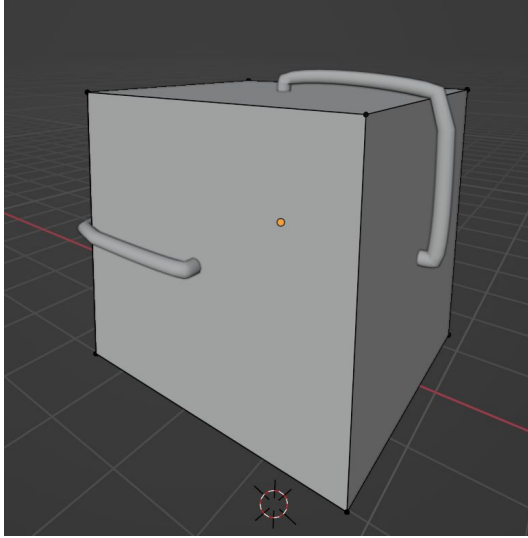
30 pipes



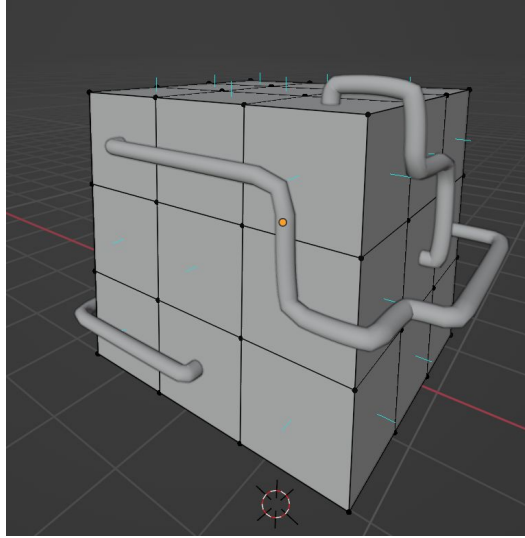
40 pipes



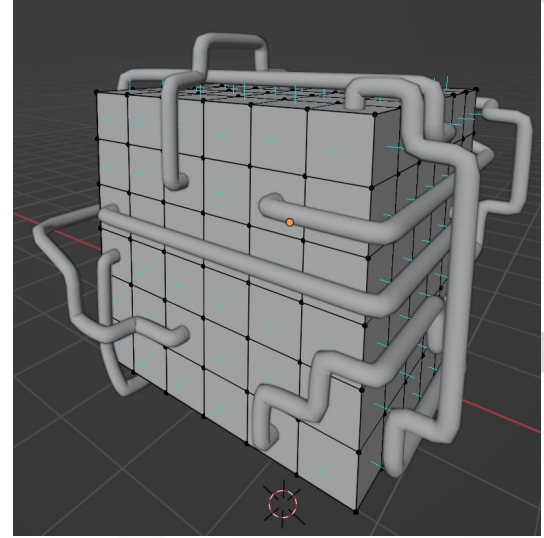
# Results - Python Script



1 face per side  
2 pipes



9 faces per side  
7 pipes



36 faces per side  
9 pipes

# Blender Python API: Main Functions

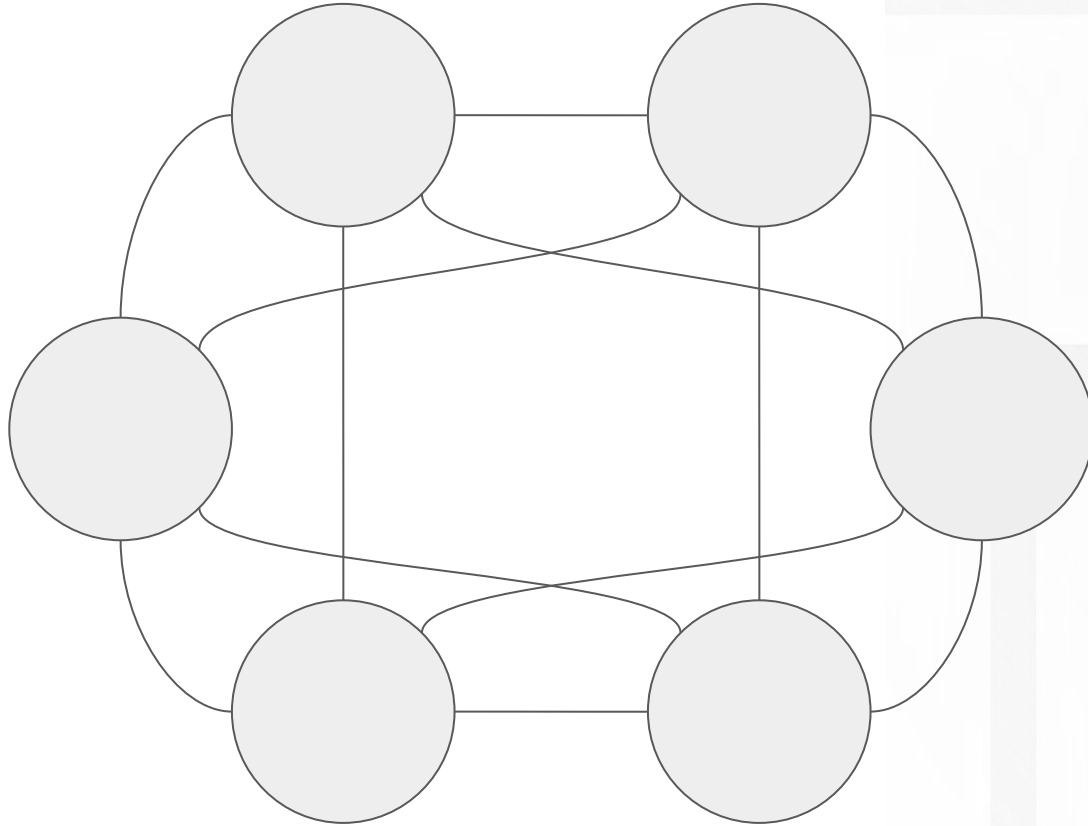
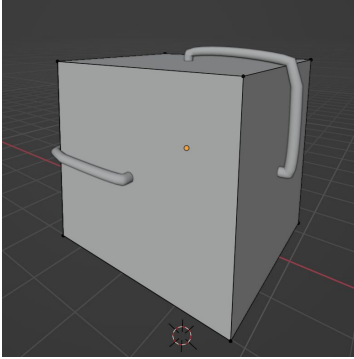
- **get\_faces\_from\_obj\_polygons(self, sel\_obj) {reading mesh}**
  - For each face on mesh, save location of center, normal, and indices of edges
- **create\_mesh\_to\_pathfind(self, faces, layers=2) {creating main graph}**
  - For each face, create a graph vertex corresponding to its center, once for each layer
    - Vertices are protruded from the face center using face normal and a user-definable offset magnitude
  - For each face, identify neighbours based on shared edge indices, create a graph edge for each corresponding vertex neighbour pair
    - These edges are added for higher layers by using vertex indices
  - Graph edges are added for connections between layers
  - For each graph edge, if the face normals for both vertices are not identical, extrapolate planes for normals, and add a vertex for closest point on the line intersection of the planes
    - Break the existing edge, add edges between new vertex and initial vertices

# Blender Python API: Main Functions

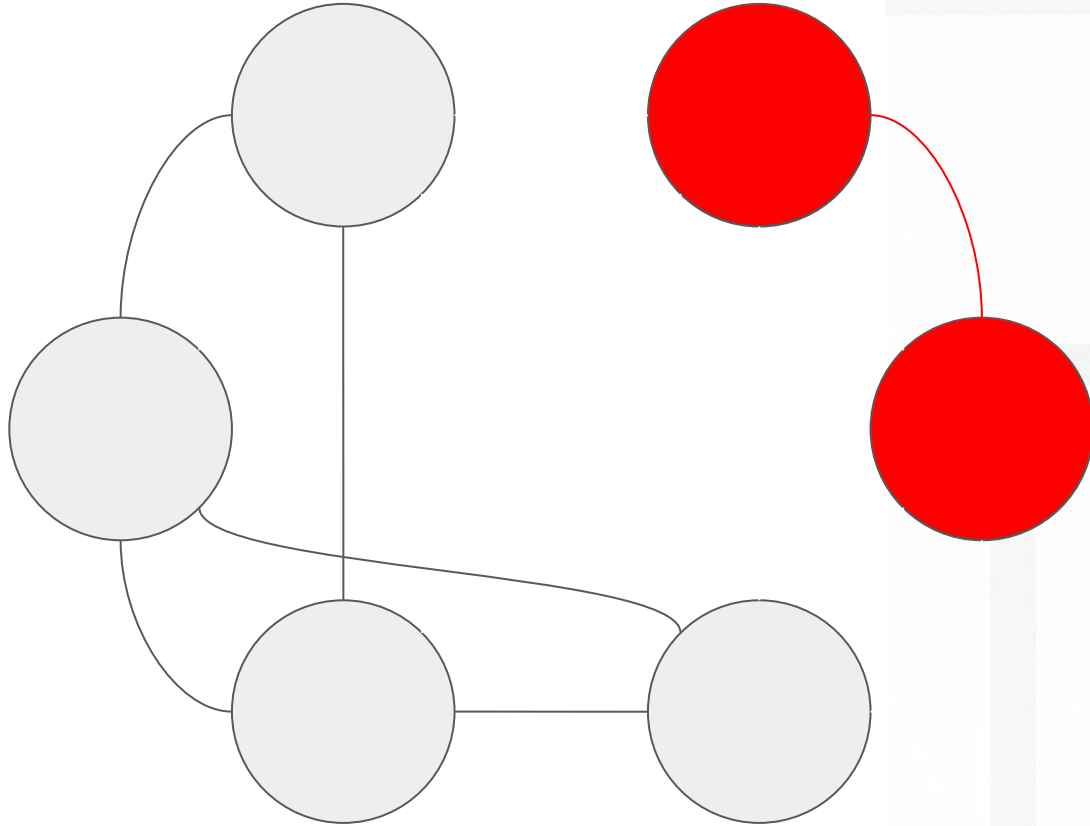
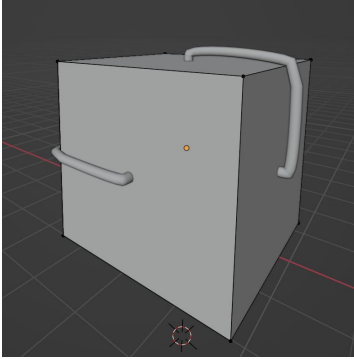
- **get\_usable\_mesh(self)** {creating usable graph}
    - Disregards occupied vertices and edges with any occupied vertex, then returns the rest
  - **find\_path(self, usable\_verts, usable\_edges, start, end, vert\_mapping)** {traversing graph}
    - Creates graph based on usable vertices and edges, then attempts to find shortest path from start vertex to end vertex
    - If valid path found, records used vertices as occupied
  - **create\_pipes(self, faces, max\_paths, radius, resolution, material, seed = 1, mat\_idx = 0)**
    - Tries to create up to {max\_paths} pipes based on input faces and existing graph.
    - Pipe creation: randomly selects 2 faces and calls a processing function that calls find\_path
      - If one of the faces is occupied or has no available neighbours, remove from list
      - If no paths found, randomly select 2 faces and try again (limit exists to prevent infinite looping)
      - If path found, create pipe object using vertices in path
    - All pipes are subdivided and smoothed to create better feel
- ✓ **Collision-prevention**
- As path for pipes are created from traversing unoccupied areas on a graph, there will be **no collisions**.



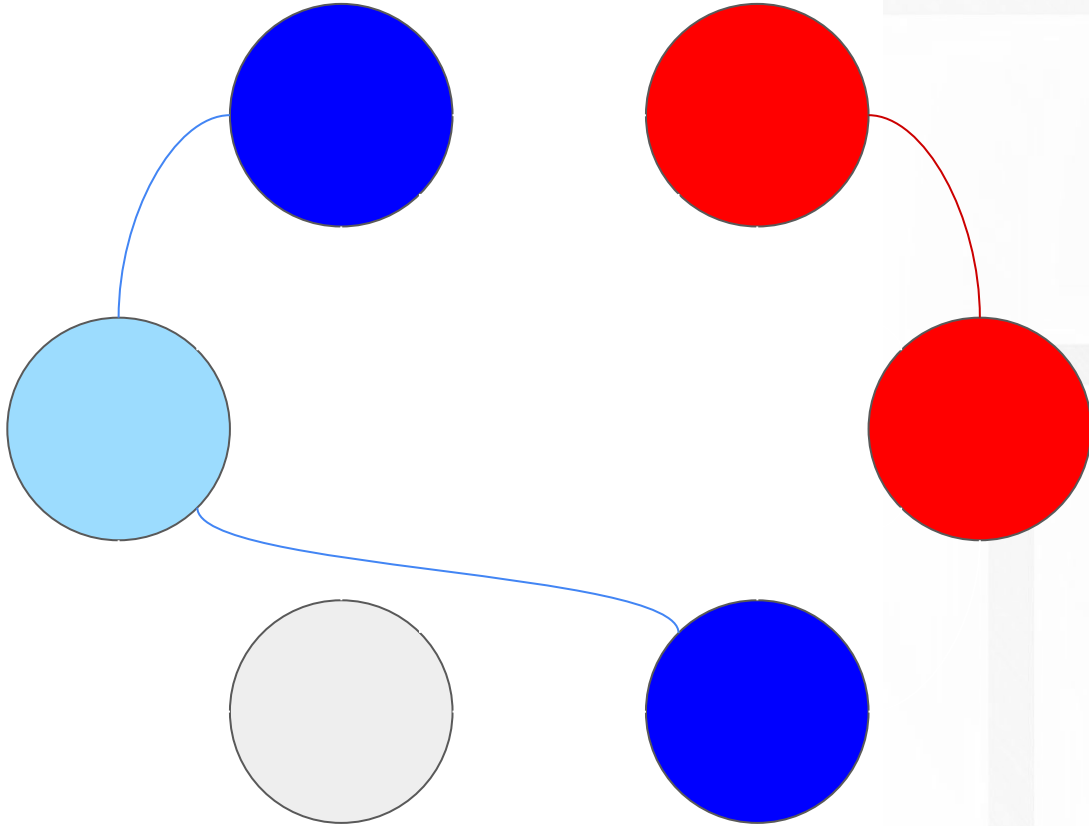
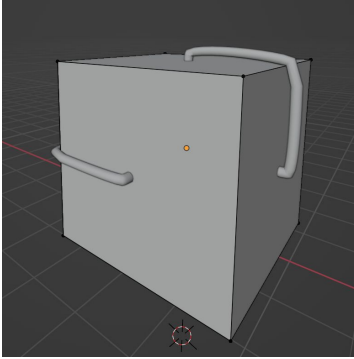
# Blender Python API: Visualizing a Simple Cube Graph



# Blender Python API: Visualizing a Simple Cube Graph

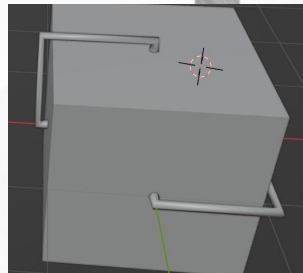
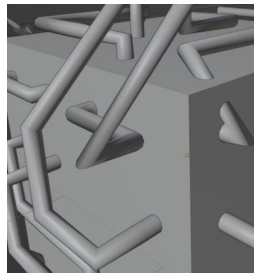


# Blender Python API: Visualizing a Simple Cube Graph



# Blender Python API: Solved issues

1. Initially looked weird, with forced bends near the ends of pipes
  - Algorithm initially used the existing vertices and edges on the mesh for the graph, but used the center of faces as start/end points for pipes
  - Pipes would exit from a face but immediately bend towards a vertex
  - **Using the faces as graph vertices** fixed this problem
2. Pipe could clip through mesh when going between edges
  - Caused by graph vertices being on non-parallel planes
  - Fix was described earlier



**Demo Video**

**<https://youtu.be/x9o-IIcBsCc>**

# Discussion

## Advantages

- Random seed-based pipe layouts
- Implemented both procedural and modular generation algorithms
- User-friendly interface
- Interactable and dynamic variables

## Limitations

- Only within Blender
- Grid-based and wall algorithms don't work well with complex/irregular meshes
- Generator might lag when generating large number of pipes

# Future Improvements

## *Python Script*

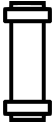
- Code can be rewritten to have lower time complexity.
- Convert to downloadable addon on Blender instead of running a script.

## *Geometry Nodes*

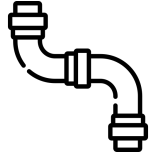
- Algorithms can be improved to be more generalized.

**The 2 algorithms can be combined to create a more comprehensive pipe generation experience.**

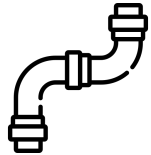
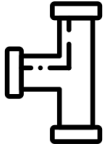
# Conclusion



Class lectures regarding different types of curves and meshes helped understand and implement our generator better.



Using Blender and Python, we were able to randomise pipe generation, controlling their density, number of layers, and type of pipes.



We learned more about Python libraries and Blender interactions though this project.







**Thank you!**

# References

Python generation algorithm and blender UI

<https://yeus.gitlab.io/post/2019-12-03-piperator0.91/>

Documentation for Blender as a Python module

<https://docs.blender.org/api/current/index.html>

Intersection of Two Planes

<https://www.microsoft.com/en-us/research/publication/intersection-of-two-planes/>