50.017 Graphics & Visualisations

2023 January Term 6 Project

Prof. Peng Song

Randomised Pipe Generation

Team 1

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

Table of Contents

1 Introduction & Problem Statement

1.1 Why did we choose this?

Currently, existing Blender add-ons regarding existing pipe generations are niche, usually requiring manual placement of control points for the tubes/pipes to generate, hence not random or automated. And, when it is procedural, the current existing algorithms do not take into account generated meshes colliding with each other.

Additionally, while niche, a fully generalised procedural pipe generator has practical usage in the design community, particularly in urban, sci-fi or steampunk environments, saving time and effort on the artists' part while being able to focus on other aspects of the scene.

1.2 Problem Statement

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

2 Our Approach

We began by exploring Blender and its different functions and libraries. After some research, we settled on geometry nodes and the Python API. We focused on creating basic procedural algorithms to generate pipes and prevent collisions and intersections. With little time to combine geometry nodes and Python API, we decided to try more algorithms using Geometry Node, and use our current Python script to generate different variations of our Pipe Generator. We ended up with 4 algorithms: 3 geometry nodes and 1 Python variation.

Lastly, we included other variables such as material of pipe, and created the user interface to facilitate user interaction.

3 Implementation

The software we used is Blender 3.4.1. It includes the in-built Geometry Nodes and Python API.

All raw data comes from the user via the interface and our algorithms implement error checking and exceptions in the case of any inappropriate input.

We elaborate more on the algorithms below.

3.1 Geometry Nodes

Algorithm 1: Grid Based

For geometry nodes, the first variation (Grid-based) uses a generated grid with X and Y number of vertices. These vertices are converted to points at a random chance based on a Voronoi texture and using a raycast, checks for any points in all directions, before replacing each point with the respective pipe joint.

```
def GridFlat(int X, int Y, scale):
# Create Initial Point Grid
        sizeX = (X - 1) / 0.1 # This is to make sure all cuts are 0.1m apart
        sizeY = (Y - 1) / 0.1 # This is to make sure all cuts are 0.1m apart

        Grid = GridPlanePrimitive(int X no.ofvertex, int Y no.ofvertex, float sizeX, float sizeY)

# Create texture for random deletion, each point's index assigned to col value, value < 0 =
# False, value > 0 = True
        # Dot product to randomise tex further, snap to lock each pixel color to fixed
        # increments
        Tex = Snap( DotProduct( VoronoiTexture(Index), 1.6,0.2,0.5 ), 0.41 )

        # Invert selection, it just somehow looks better this way
        Selection = BooleanNot(Tex)

        # Get 'broken' grid of points due to texture boolean
        Grid = ConvertVertsToPoints(Obj Grid, boolean Selection)
        IndexList = ExtractIndicesList(Grid)
        PositionList = ExtractPositionList(Grid)

# For every point in Grid, check each point and assign correct pipe (16 combinations) to
# each point.

        # Raycast (custom-made node) each point and check if there is another point X/Y
        # away from it, outputs boolean. Raycast(Index,Points,RayInitialPosition,OffsetDirAmount).
        # OffsetDirAmount is a vector [X,Y,Z]. E.g With [-0.1,0,0], ray checks up till -0.1 on
        # the X axis relative to that point.
        for i in IndexList:
                Up = raycast(i, Grid, PositionList(i), [0,0.1,0])
                Down = raycast(i, Grid,PositionList(i), [0,-0.1,0])
                Left = raycast(i, Grid, PositionList(i), [-0.1,0,0])
                Right = raycast(i, Grid, PositionList(i), [0.1,0,0])

                # Binary counting system to assign each point to correct pipe in PipeCollection
                Isel = (Up * 8) + (Down * 4) + (Left * 2) + (Right * 1)

                # PipeCollection is a group in the .blend file with assigned index for each object so that we can
                # call it with integers
                ConvertPointToInstances(points Grid, Instance PipeCollection, InstanceIndex
                Isel)
```

Algorithm 2: Grid Faces (Grid Based but on all faces)

The second variation (Grid Faces) is similar but takes the normal of each polygon, generates the grid-based pipes on each polygon and rotates it to face the normals. It also deletes any of the generated polygons that are too far away from the mesh.

```
def GridFaces(obj Obj, float Scale, float Seed):
    # Get Max Length of obj
    Maxlen = BoundingBox(Obj).max.round

# Convert obj faces to points, save normal of each face to be copied onto point
    # make sure the polygon is planar, otherwise remove it, 0.01 is the threshold
    ObjEdit = FaceIsPlanar( seperateEdges(Obj).selectTrue, 0.01)
    NrmList = ExtractNrmList(ObjEdit)

    # rotate all points to be same as their face nrms so that generated grids are all facing correct direction
    Points = ConvertFacesToPoints(obj ObjEdit)

    for p in Points:
        setRotation( p, alignment Z, NrmList(p.index) )
        # offsets all points by some amount in the normal direction (so pipes wont be inside the face)
        Translate( p, NrmList(p.index), [0.08,0.08,0.08] )

    # Convert all points to GridFlat function outputs
    Instances = ConvertPointToInstances(points Points, Instance GridFlat( Maxlen,
    Maxlen, Scale ) )

    NrmList = ExtractIndicesList(Instances.mesh.vertices)
    PositionList = ExtractPositionList(Instances.mesh.vertices)

    # Raycast and check ray x amount away from the normal of each face, if mesh is too far away from the
    # normal, delete mesh.
    for v in Instances.mesh.vertices:
        Hit = DotProduct( Raycast( Index(v), Points, PositionList(v), NrmList(v) ),
        PositionList(v) )

        if Hit == False:
            Delete(v)
```

Algorithm 3: Wall

The third variation (Wall) generates a straight curve, locked onto the Y axis and randomly offsets some points based on a texture.

```
def GridWall(obj Obj, float MinDist, float Rad1, float Rad2, float SecondaryPipeOffset, float Seed):
# GENERATE MAIN PIPES W. NOISE OFFSETS
    # Get Max Length of obj
    Maxlen = BoundingBox(Obj).max.round

    NrmPointList = ExtractNrmPointList(Obj)
    NrmFaceList = ExtractNrmFaceList(Obj)

    # Distribute pts on all faces, merge them if points are < 0.5m apart
    Points = DistributePointsOnFaces(Obj, MinDist, Seed)
    Points = MergeByDistance(Points, 0.5)

    Instances = ConvertPointToInstances(points Points, Instance
    MeshLinePrimitive(length Maxlen) )

    # Lock all pipes generated into the Y axis. It is a restriction but also makes building the algo a lot easier.
    RotateInstances(Instances, [0, 90, 0] )
    OffsetInstances(Instances, [0.05, 0, 0] )
    ScaleInstances(Instances, [1.5, 1.5, 1.5] )

    # Resample curves means the curve will have the same shape, but subdiv/unsubdiv to have X number
    # of points
    ConvertMeshtoCurve(Instances)
    ResampleCurve(Instances, 175)
    MeshEdit = ConvertCurveToMesh(Instances)

    For f in MeshEdit.faces:
        dist = GeometryProximity(source MeshEdit, target Obj, type Faces)
        # if face distance between meshline and original object is > 0.09, delete face
        if dist > 0.09:
            Delete(f)
    # Set each index of mesh to combined texture, perform some math to offset and
    # make pipes look better
    Tex1 = Snap( NoiseTexture(Index+2, scale 0.03), 0.46 )
    Tex2 = Snap( NoiseTexture(Index, scale 0.009), 0.36 )
    NewTex = ( Tex1 * Tex2 - [0.4, 0.4, 0.4] ) * -0.8

    # Offset instances by Nrm, then set offset for each index via the texture by each index's tangent
    OffsetInstances(MeshEdit, [0.01, 0.01, 0.01] *  NrmFaceList)
    for p in MeshEdit.points:
        Translate(p, CrossProduct( NrmFaceList(p), NewTex ) )

    # Convert meshline back to curve, fillet and resample to smoothen curve and give it thickness
    Curves = ConvertMeshToCurve(MeshEdit)
    FilletCurve(Curves, count 3, radius 0.15)
```

```
        ResampleCurve(Curves, length 0.1) # Each point is 0.1m away from each other
        CurveCircle(Curves, radius Rad1)

# Add random Extrusions via index comparing to random noise value
        Threshold = Normalise(value NoiseTexture(Index, scale 0.05), from 0.3-1.1, to
        -5000000-5000000)

        ExtMesh = MeshEdit.copy
        for f in ExtMesh.faces:
                if f.index < Threshold:
                        Delete(f)
        Extrude(ExtMesh, offset 0.02)

# Add secondary Pipes using main Pipes as reference
        secMesh = MeshEdit.copy

        # Use main pipe as geometry, convert each point into an meshline instance with 0.04 probability
        SecInstances = ConvertPointToInstances(points secMesh, Instance
        MeshLinePrimitive(length SecondaryPipeOffset), selection
        RandBoolValue(probability 0.04) )

        # Delete every 2nd instance (to leave gaps so secondary pipes won't intersect easily)
        DeleteInstance(instances SecInstances, selection Modulo(Index, 2) )

        ResampleCurve(instances SecInstances, countnum 16)

        # Translate instances a random amount tangent of the normal
        Offset =  CrossProduct((Snap( NoiseTexture(scale 0.4), 0.3) - 0.5)*0.21 *
        NrmFaceList, [-2.6, 0, 0] )
        TranslateInstances(instances SecInstances, offset Offset )

        # Start&EndSelection selects the start and end indices of each instance. GeometryProximity gets the
        # position of the closest target mesh from the main curve.
        SetPosition(instances SecInstances, selection Start&EndSelection(SecInstances),
        position GeometryProximity(type Points, target MeshEdit)

        # give secondary curves thickness
        CurveCircle(instances SecInstances, radius Rad2)
```

3.2 Python API

Algorithm 4: Python Script - Generalised Generation

The Python script, the last variation, uses the concept of graphs and path-finding. A fixed graph is created, and for every pipe generated, it gets a start and end vertex, and uses them to get the shortest path. If successful, used vertices are removed from the graph. This prevents any collisions.

```
def get_faces_from_obj_polygons(obj SelObj):
# Gets List of faces given an object mesh
        Faces = []
        # Gets the mesh's polygons
        Polygons = Selobj.polygons
        For each polygon:
                Faces.add(poly.location, poly.normal, poly.edges)
        Return Faces

def create_mesh_to_pathfind(List Faces, int layers):
# Creates vertices and edges for graph usage
        # Creates graph vertices
        For each face:
                For each layer:
                        Create vertex with face.location offsetted and face.normal
                Create vertex pair between adjacent layers
                Assign index to face
        # Creates graph edges
        For each face:
                For each other face that shares an edge:
                        For each layer:
                                Create vertex pair if it does not already exist
        # Fixes pipes clipping through mesh
        For each vertex pair:
                If vertex normals are not identical:
                        Extrapolate planes based on normals
                        Find line of intersection between planes
                        Find closest point to vertices on line of intersection
                        Create new vertex using closest point
                        For each vertex in vertex pair:
                                Create vertex pair with new vertex
                        Remove original vertex pair
        Set Class Attribute Vertices and Vertex Pairs

def create_pipe_skeletons(Face start, Face end):
# Identifies vertices used in pipe
        # Gets the face vertices
        start_vertex = Base layer vertex of start
        end_vertex = Base layer vertex of end
        If either vertex is occupied or has no free neighbours:
                Return occupied = True, occupied state of vertices
        Else:
                Get unoccupied vertices and vertex pairs
```

```
            Attempt to find path from start_vertex to end_vertex
            If path found:
                Mark used vertices as occupied
                Return occupied = False, list of used vertices
            Else:
                Return “PATH NOT FOUND”

def render_curve(List pipe_skeletons, float radius):
# Renders a curve given certain parameters
    Curve.radius = radius
    Set some other curve parameters
    Add vertices in pipe_skeletons to Curve
    Create Curve object in Blender

def create_pipes(List Faces, int max_pipes, int limit, float radius, other parameters):
# Given list of faces and other parameters, create pipes
    While number of pipes < max_pipes and current_tries < limit and free faces >= 2:
        Start_face, End_face = 2 different random faces
        Occupied, Vertices = create_pipe_skeletons(Start_face, End_face)
        If Occupied:
            Remove occupied faces
        Else if PATH NOT FOUND:
            current_tries <- current_tries +1
            Return to start of while loop
        Else:
            render_curve(Vertices, radius)
            Number of pipes <- Number of pipes + 1
            Remove faces
    Beautify pipes
```

3.3 User Interface

PipeGenerator Panel: This is where we can access the operators to add new pipes or delete existing pipes.
When “Add Pipes” is pressed, the corresponding operator will open.
When “Delete Object Pipes” is pressed while selecting a mesh with pipes generated on it, it will delete all pipes connected to that mesh.
*Delete Object Pipes only works with Python script algorithm, not with geometry nodes algorithms.

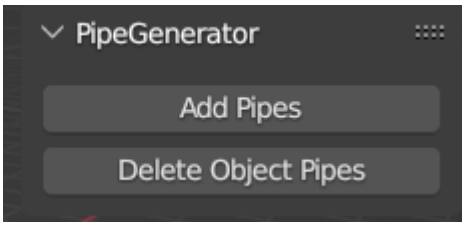

Figure 3.1. PipeGenerator UI Panel

Add Pipes: This operator opens once “Add Pipes” is pressed on the panel. The user can adjust variables and switch between our different algorithms here.

- Algorithm used - Switch between “Default python script”, “GeoNode - GridFlat”,“GeoNode - GridFaces”, “GeoNode - GridWall”
- Type of material - material of pipes
- Radius - radius of pipes
- Resolution v - number of sides for cross-section of pipe
- Offset - offset distance from face of mesh
- Random seed - seed for basis of random pipe generation
- Number of pipes - maximum number of pipes to be generated. If no more available vertices, will not generate up to maximum
- Number of layers - number of layers for pipe generation area

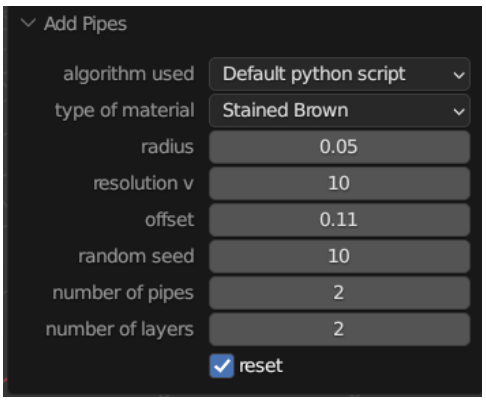

Figure 3.2. Add Pipes UI Panel

Geometry Nodes: For GeoNode algorithms, the right-hand modifiers are used instead of the Add Pipes operator.

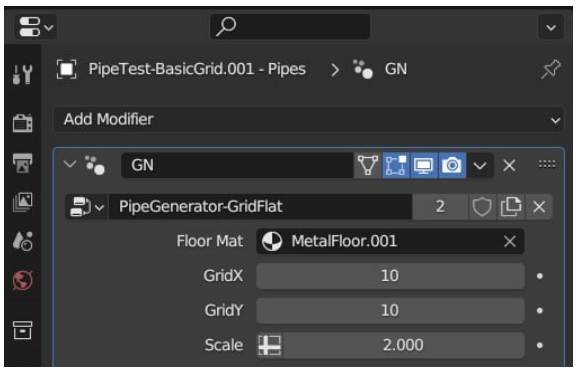

Figure 3.3. Geometry Nodes Algorithm 1 (Grid)

GridFlat

- Floor Mat - material of pipes
- GridX - X axis length of generation
- GridY - Y axis length of generation
- Scale - scale of pipes

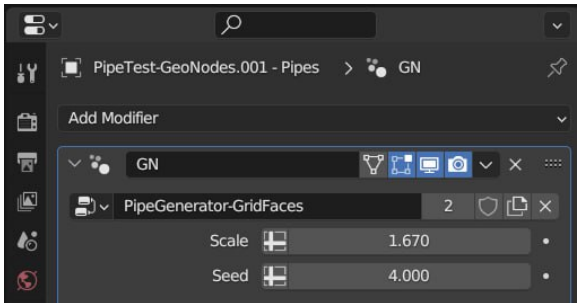

Figure 3.4. Geometry Nodes Algorithm 2 (Grid Faces)

GridFaces

- Scale - scale of pipes
- Seed - seed for basis of random noise texture to generate pipes on

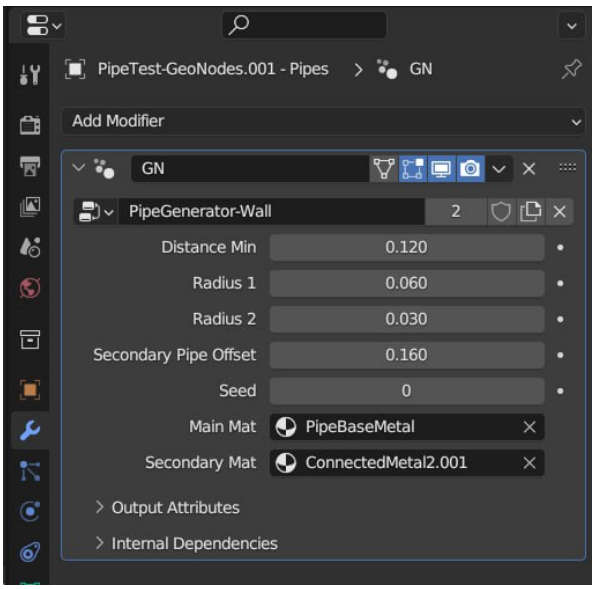

Figure 3.5. Geometry Nodes Algorithm 3 (Wall)

GridWall

- Distance Min - minimum distance between each pipe
- Radius 1 - radius of primary pipes
- Radius 2 - radius of secondary pipes
- Secondary pipe offset - maximum distance secondary pipes offset from primary pipes
- Seed - seed for basis of random noise texture to generate pipes on
- Main mat - primary pipe material
- Secondary mat - secondary pipe material

4 Results

4.1 Geometry Nodes

Algorithm 1: Grid Based

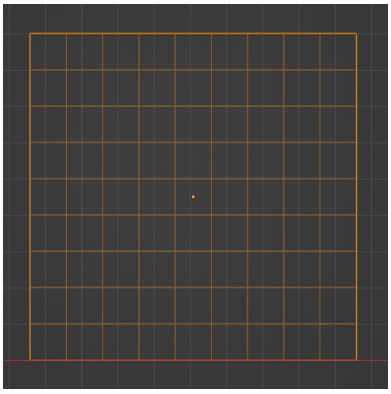

Figure 4.1.1 Grid created

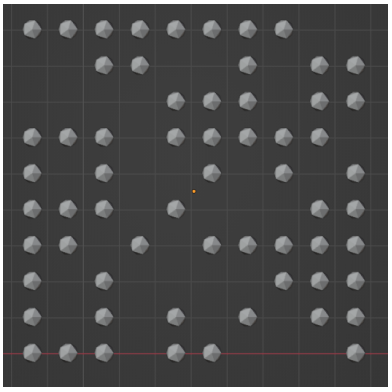

Figure 4.1.2 Grid with points

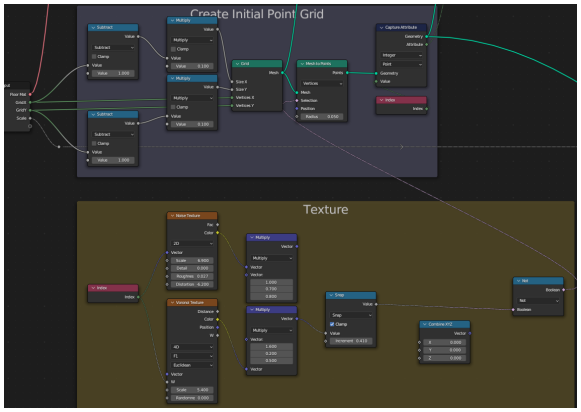

Figure 4.1.5 Geometry nodes to achieve Figure 4.1.2

Figure 4.1.4 Modular pipe joints

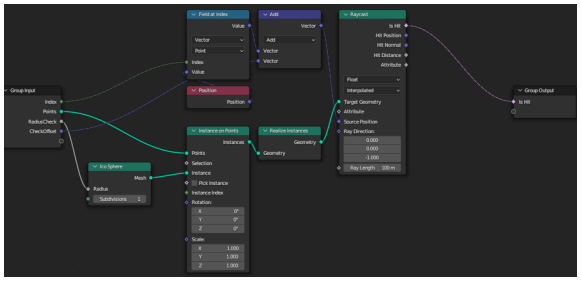

Figure 4.1.3 Geometry node to replace points with pipe joint

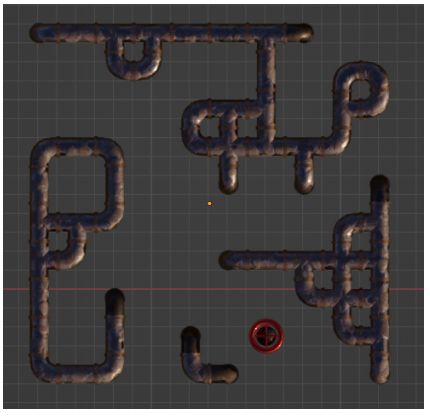

Figure 4.1.6 Grid algorithm example

Algorithm 2: Grid Faces (Grid Based but on all faces)

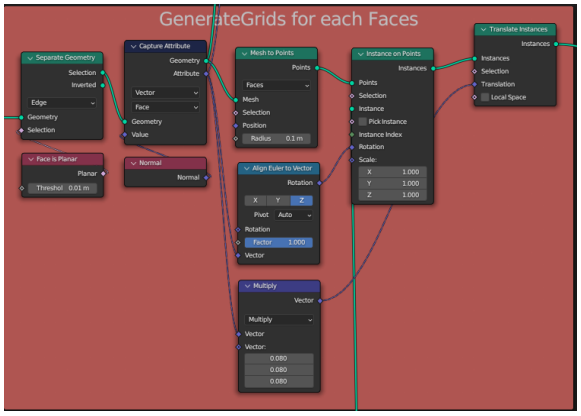

Figure 4.2.1 Geometry nodes to generate grids for each face

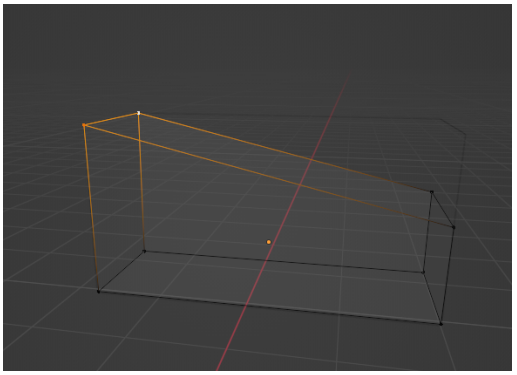

Figure 4.2.2 Object example

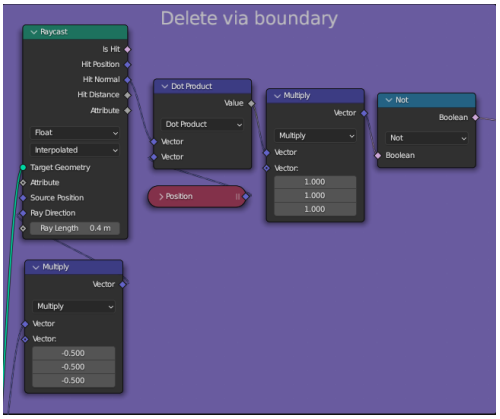

Figure 4.2.3 Geometry nodes to delete pipes via boundary

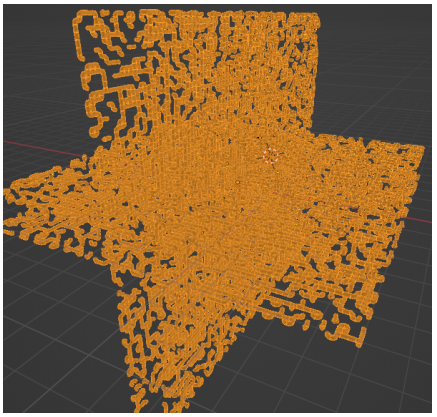

Figure 4.2.4 Pipes before 4.2.3

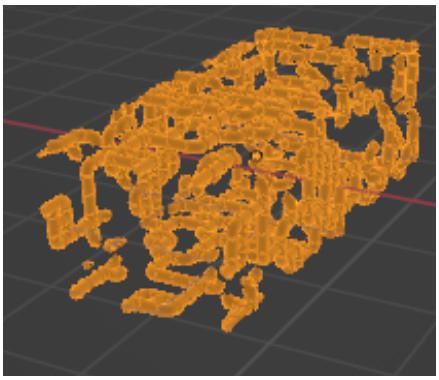

Figure 4.2.5 Pipes after 4.2.3

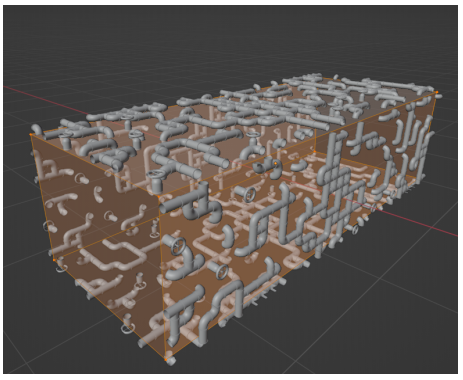

Figure 4.2.4 Grid Faces algorithm example

Algorithm 3: Wall

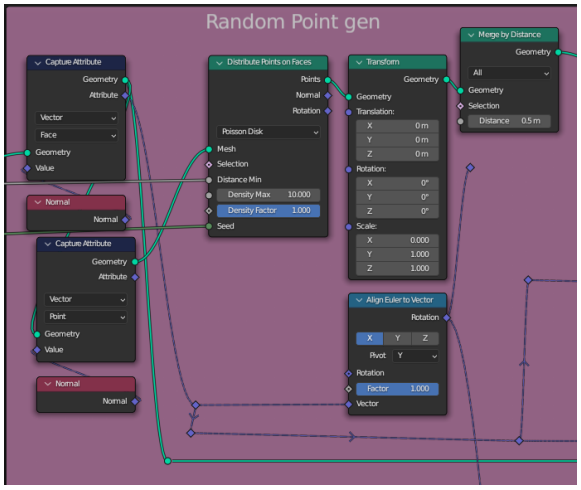

Figure 4.3.1 Geometry nodes to generate random points

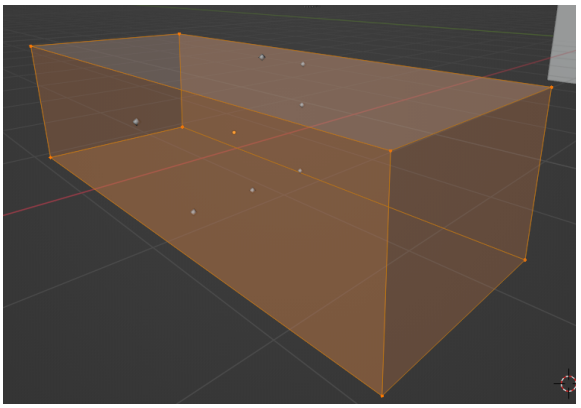

Figure 4.3.2 Object after 4.3.1

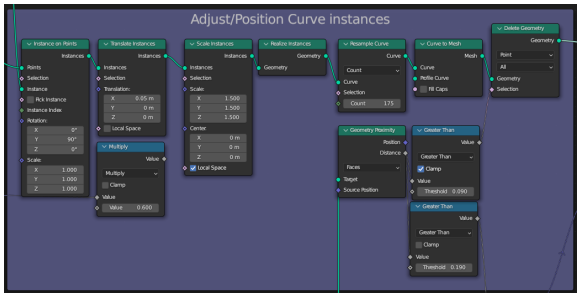

Figure 4.3.3 Geometry nodes to adjust curve instances

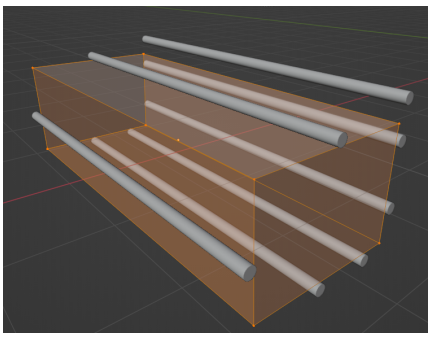

Figure 4.3.4 Object after 4.3.3

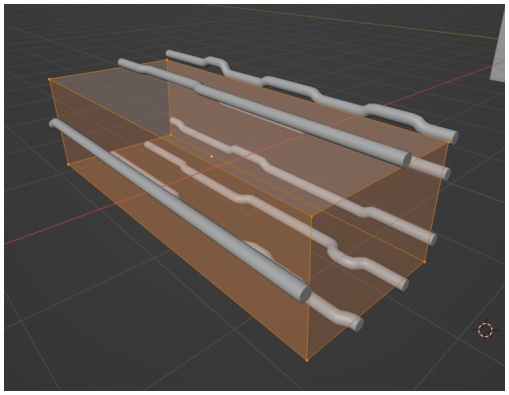

Figure 4.3.5 Object after random offset (based on texture)

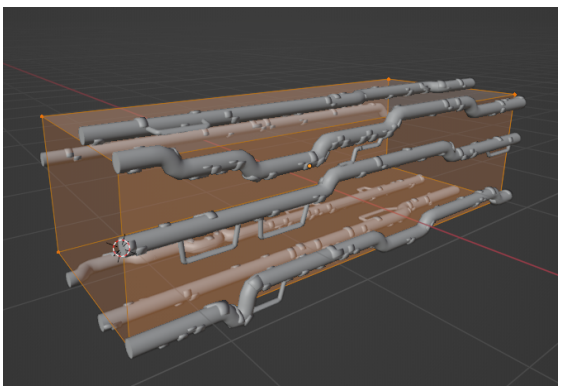

Figure 4.3.6 Object after adding secondary pipes

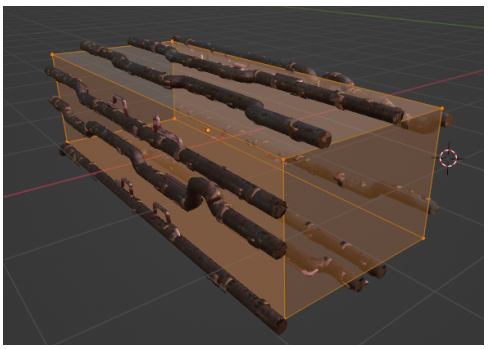

Figure 4.3.7 Object after adding material

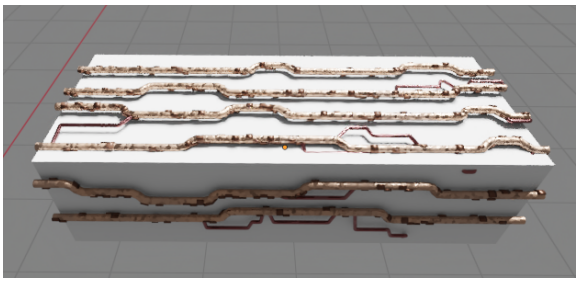

Figure 4.3.8 Wall algorithm example

4.2 Python API

The figures below show the pipe generation on a torus.

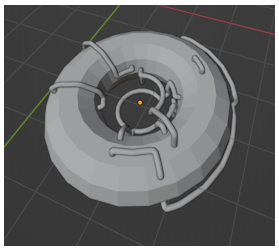

Figure 4.4.1
10 pipes

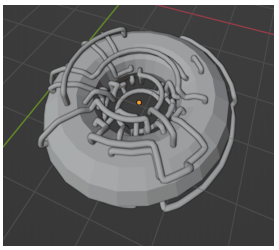

Figure 4.4.2
20 pipes

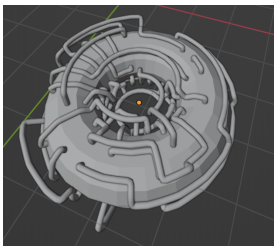

Figure 4.4.3
30 pipes

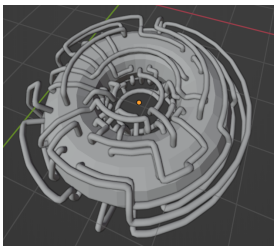

Figure 4.4.4
40 pipes

The figures below show the pipe generation on a cube. The pipes are able to generate from and through the faces. With more subdivisions, the pipes generated have more available areas to move.

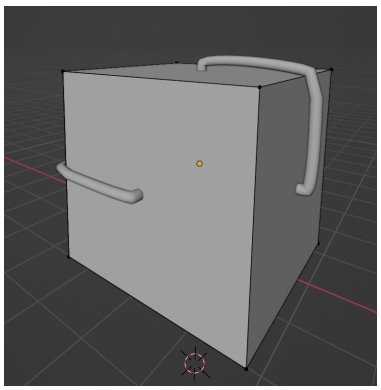

Figure 4.4.5
1 face per side, 2 pipes

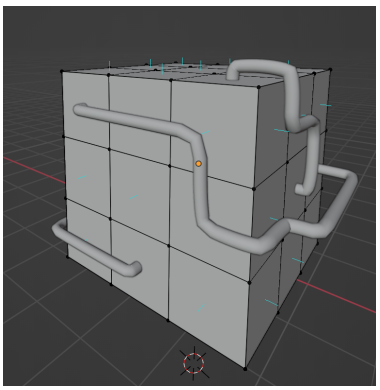

Figure 4.4.5
9 face per side, 7 pipes

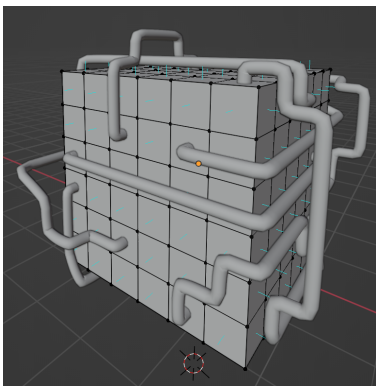

Figure 4.4.5
36 face per side, 9 pipes

5 Discussion

Advantages

- The layout of the pipes can be randomised through seeds.
- Both procedural (script and wall) and modular (grid based) generations are available for different needs.
- Users can adjust different variables such as radius, material, offset, and layers which makes our generator more interactable.

Limitations

- This generator is only usable within Blender.
- Grid based and wall algorithms do not work with very complex/irregular meshes.
- Depending on CPU specs, the generator might lag when generating too many pipes at once since the generation algorithm is a bit slow.

6 Conclusion & Future Work

We managed to combine our knowledge of Python and Blender to create 4 different algorithms for our pipe generator. The script algorithm is coded entirely through Python, and generates pipes by finding the shortest path between 2 points on the mesh. This is the most flexible algorithm out of the 4. The grid-based algorithm uses modular components and random noise textures to generate pipes on one or more faces of a mesh. The wall algorithm generates primary and secondary pipes along one axis on the mesh. This project helped the members learn more about both Python and Blender, as well as new libraries and modules.

One improvement can be allowing the user to choose more materials for the script algorithm. Although our grid-based algorithms allow the user to select any material in Blender and even upload their own, the materials for the Python script are hard-coded. In the future, we can allow the Python script algorithm to pull materials from Blender's material library instead to make the material selection more dynamic.

In the future, we can also try to combine our existing algorithms into one universal algorithm that can be applied to different types of meshes. This will also make our generator more intuitive for users.

7 References

T.M. Meschede. “Introducing Piperator” (2020). [Online]. gitlab.io. Accessed: 19 April, 2023. Available: https://yeus.gitlab.io/post/2019-12-03-piperator0.91/

Blender 3.5 Python API Documentation. [Online]. Accessed: 19 April, 2023. Available: https://docs.blender.org/api/current/index.html

J. Krumm. “Intersection of Two Planes” (2000). [Online]. Microsoft.com. Accessed: 19 April, 2023. Available: https://www.microsoft.com/en-us/research/publication/intersection-of-two-planes/