

Parametric and Generative Design with Rhino Python and Grasshopper

Exploration in Form Finding and Procedural Generation

—

Isabel Lupiani

Apress®

Parametric and Generative Design with Rhino Python and Grasshopper

Exploration in Form Finding and Procedural Generation

Isabel Lupiani

Apress®

Parametric and Generative Design with Rhino Python and Grasshopper: Exploration in Form Finding and Procedural Generation

Isabel Lupiani
ORLANDO, FL, USA

ISBN-13 (pbk): 979-8-8688-2723-5 ISBN-13 (electronic): 979-8-8688-2724-2
https://doi.org/10.1007/979-8-8688-2724-2

Copyright © 2026 by Isabel Lupiani

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Editorial Assistant: Rachel Zhang

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Pythonand-Grasshopper.

If disposing of this product, please recycle the paper

To James, Zoe, and Caleb.

Table of Contents

About the Author xix

About the Technical Reviewers xxi

Acknowledgments xxiii

Introduction xxv

Part I: Building Blocks 1

Chapter 1: Grasshopper and Rhino Python Quick Start: Part 1—The Basics 3

Running This Chapter's Examples 3

Overview of Scripting in Grasshopper 4

Why Use Grasshopper? 4

Workflow 5

Terminology 5

Launching Grasshopper 5

Grasshopper Editor Interface 6

Locating a Component in the UI 8

Placing a Component on the Canvas 10

The Anatomy of Components 13

General Structure 13

Color 14

Name as Text or Icon 17

Component Outputs 18

Strategies for Creating Input Values 24

Default Values 24

Inheriting Data from Upstream Component(s) 24

Referencing Data from Rhino 24

Setting One or Multiple Values Directly 27

Creating One or More Numbers 28
Using Panels to Create Number(s) 29
Using Params Containers to Create Number(s) 31
Setting Numbers Directly on an Input via Set/Set Multiple 31
Creating Domains with Panels or Construct Domain 31
Number Slider 32
Creating a Boolean Value (True or False) 38
Creating a Point 39
Types of Wires 40
Basic Transforms 42
Move 42
Rotate 44
Scale 45
Creating Variations 46
Summary 46

Chapter 2: Grasshopper and Rhino Python Quick Start: Part 2—Advanced Topics 47

Running This Chapter's Examples 48
Data Conversion 48
Numbers to Booleans and Rounding Numbers 48
Implicit Conversion to Planes 49
Working with Lists 50
Range: Creating a Number Sequence by Dividing a Range (with Fixed End) 51
Series: Creating a Sequence by Skip Counting a Given Number of Times (with the End to be Determined) 53
Removing Items with a Recurring Pattern (Cull) 56
Splitting a List (Partition and Sub List) 57
Dispatch and Weave 59
Data Matching 61
Longest List (Default) 61
Shortest List 65

Cross Reference 66
Data Matching: More Examples 66
Data Trees (Nested Lists) 70
Simplify (Remove Unnecessary Branching) and Flip Matrix (Rows Become Columns) 72
Flatten: Convert a Data Tree into a List 74
Graft (Turn Every Item into a Tree Branch) 75
Reverse 78
Using the Python 3 Script Node to Define a Custom Component 81
Writing Python vs. Using Existing GH Components 81
Writing Your First Python 3 Script 82
Adding Inputs and Outputs 85
Access and Type Hints 87
Saving a Custom Script As a User Object 87
Organizing Grasshopper Definitions 90
Aligning and Distributing Components 90
Grouping Components 92
Commenting with Scribbles and Panels 94
Baking: Turning Grasshopper Preview into Rhino Geometry 94
Summary 96
References 97

Chapter 3: Attractors 99

Running This Chapter's Examples 99
What Is an Attractor? 99
1D Attractors 100
Creating a Linear Array with Custom Spacing 101
1D Attractor 104
1D Repeller 107
Adding Levitation 108
2D Attractors 110
Stage 1: Setting Up a 2D Array of Polygon Curves 110
Stage 2: Creating an Emitter Point 111

Stage 3: Deriving Scale Factors for Polygons Based on Their Distances to the Emitter..... 111
Stage 4: Creating Surfaces from Curves and Scaling Them..... 112
Tip: Organizing Node Graphs with Relay and Hidden Wires..... 114
Adding Levitation..... 115
3D Attractors..... 117
Creating a Basic 3D Attractor via Extrusion..... 117
Displacement Using a Vector Force Field..... 119
Polar Grid Attractor..... 122
Stage 1: Creating a Polar Grid with Python 3 Script and Polar Array..... 123
Stage 2: Creating an Emitter Point for a Polar Grid..... 130
Stage 3: Computing Scale Factors from Distances Between the Emitter Point and Circle Centers..... 135
Stage 4: Creating Surfaces from Circles and Scale Them..... 136
Curve Emitters..... 139
Summary..... 146
Chapter 4: Tessellations..... 149
Running This Chapter's Examples..... 150
Construction Method #1: Overlapping 2D Arrays of Primitives..... 150
Stage 1: Creating a 2D Array of Polygon Curves..... 150
Stage 2: Creating a Second 2D Array by Offsetting the First Array Horizontally..... 151
Stage 3: Overlaying the Two Arrays to Produce the Tessellation..... 152
Variation #1: Squares with No Filet, No Rotation..... 154
Variation #2: Squares with No Filet, ¼ Pi Radian Rotation..... 155
Variation #3: Hexagons with No Filet, No Rotation..... 156
Variation #4: Hexagons with No Filet, ½ Pi Rotation..... 156
Variation #5: Hexagons with No Filet, ¼ Pi Radian Rotation..... 157
Ideas for Further Experimentation..... 158
Construction Method #2: Sliding Points Along a Cell's Edges and Linking Them thru the Cell's Center to Create a Motif..... 159
Stage 1: Creating a Half Motif by Sliding Points Along the Array Cell's Edges and Linking Them thru Cell Center..... 159

Stage 2: Overlaying a Mirrored Copy of the Half Motif on Top of Itself to Make the Final Motif 160
Stage 3: Tiling the Final Motif Using Rectangular Array 161
Creating Variations by Sliding Edge Points by Different Amounts 164
Composing a Larger Motif from Mirrored Copies of a Single Motif 165
Creating Variations by Sliding Edge Points by Different Amounts 168
Construction Method #3: Sliding Points Along Both an Array Cell's Edges and Diagonals 169
Stage 1: Sliding Points Along the Array Cell's Edges 170
Stage 2: Sliding Points Along Cell Diagonals and Linking Edge Sliders and Diagonal Sliders in Alternating Pattern to Form Single Star Motif 172
Stage 3: Arranging Mirrored Copies of the Single Star into the Final Motif and Tiling the XY Plane 173
Creating Variations by Sliding Edge Points and Points Along Diagonals by Different Amounts 176
Ideas for Further Experimentation 178
Construction Method #4: Using Maelstrom to Swirl the Radial Lines Motif from Method #2 179
Stage 1: Creating a Half Motif by Combining Diagonals with Lines Linking Sliding Points on Cell Edges and "Swirling" the Lines 179
Stage 2: Overlaying Mirrored Copy of the Half Motif on Top of Itself to Create the Final Motif 180
Stage 3: Tiling the Final Motif on the XY Plane with Rectangular Array 181
Creating Variations by Sliding Edge Points by Different Amounts and Using Different R0 and R1 Values for Maelstrom 185
Ideas for Further Experimentation 187
Summary 187

Chapter 5: Loops and Spirals 189

Running This Chapter's Examples 189
Installing the LunchBox Plugin 189
Making Loops: Möbius Strips 190
Construction Method #1: Using the Mobius Surface Node from LunchBox 191
Construction Method #2: Using the Twist Node 193
Construction Method #3: Using Math Formulas 196

Making Spirals: Logarithmic Shells 202
Step 1 of 3: Creating Input Values for the Spiral Curve 204
Step 2 of 3: Using Python to Derive Points for the Spiral Curve and Radii for the Shell 204
Step 3 of 3: Generating the Shell Surface 206
Making Spirals: Staircases 208
Stage 1 of 6: Creating the Center Pole and a Spiral Curve Around the Pole 208
Stage 2 of 6: Forming Tread Rectangles Along the Spiral Curve 211
Stage 3 of 6: Tapering Treads Toward the Center Pole 214
Stage 4 of 6: Extruding Tread Surfaces to Create Thickness 216
Stage 5 of 6: Creating Balusters 218
Stage 6 of 6: Creating the Handrail 221
Creating Variations 225
Summary 226
References 226

Chapter 6: Stacking and Voxelizing 227

Running This Chapter's Examples 227
Making Stacks: Columns 228
Column with Scaling 228
Column with Rotation 232
Column with Variable Block Thickness, Tapering, and Random Horizontal Offsets 234
Making Stacks: Random Blocks Along an Arbitrary Curve 244
Stage 1: Creating a Sine Curve 244
Stage 2: Deriving Thicknesses for the Blocks by Randomly Dividing the Curve 246
Stage 3: Creating Random 2D Outlines for the Blocks 246
Stage 4: Extruding the Blocks Along Curve Direction 247
Making Stacks: Stacking Within a Volume 249
Stage 1: Populating Points in a 3D Volume 251
Stage 2: Creating Blocks with Random Dimensions with Bottoms Centered at Those Points 251
Stage 3: Creating Poles from the Blocks to the Ground 252

Voxelizing: Divided Surface (Hollow) 253
Stage 1: Evenly Dividing a Surface into a Grid of Points 254
Stage 2: Creating Blocks of Random Dimensions with Bottoms Centered at Those Points 254
Variation: Voxelizing a Cylindrical Surface 256
Voxelizing: Mesh or Solid 258
Attribution to the Stanford Bunny 258
Stage 1: Reducing Poly Count and Converting to SubD 260
Stage 2: Deriving an Enlarged Bounding Box from the SubD 260
Stage 3: Creating a 3D Voronoi Bordered by the Bounding Box 261
Stage 4: Creating Voxelized Model from the Voronoi Cells' Bounding Boxes 261
Variation: Using Spheres As Voxels 262
Summary 263
References 264

Chapter 7: Cracks and Wireframes 265

Running This Chapter's Examples 265
Installing the LunchBox Plugin 265
Attribution to the Stanford Bunny 266
Making Wireframes: LunchBox Star Dome 266
Stage 1: Creating a Half Cube to Enclose the Bottom Half of the Sphere 268
Stage 2: Paneling the Sphere with Skewed Quads 268
Stage 3: Creating the Wireframe Dome from the Top Half of the Paneling 269
Making Wireframes: Beehive Dome 270
Stage 1: Creating a Half Cube to Enclose the Bottom Half of the Sphere 272
Stage 2: Creating a Hemisphere with an Open Bottom by Trimming a Sphere with the Half Cube 272
Stage 3: Projecting a Hexagonal Grid onto the Open Hemisphere and Thickening the Edges to Create the Dome 273
Making Wireframes: Floral Tower 274
Stage 1: Creating Vertical Division Lines Around a Cone 276
Stage 2: Sliding Points Around the Cone's Base to Create a Second Set of Lines 276
Stage 3: Using Maelstrom to Swirl the Lines and Combining the Results with a Mirrored Copy 277

Making Wireframes: From a Mesh 279
Quad Wireframe 279
Triangular Wireframe 281
Hex Wireframe 283
Making Wireframes: By Cracking a Surface 285
Cracked Cube Wireframe 285
Cracked Sphere Wireframe 288
Making Cracks: Exploded Cube 290
Stage 1: Creating a 3D Voronoi Diagram Inside a Cube to Emulate Cracked Chunks 292
Stage 2: Displacing Chunks Radially Outward from the Cube's Center to Create Explosion 292
Variation: Exploded Sphere 295
Making Cracks: Exploded Mesh 297
Stage 0: Simplifying + Converting the Mesh to a SubD and Finding the Mesh's Enlarged Bounding Box 299
Stage 1: Creating a 3D Voronoi Inside the Mesh's Enlarged Bounding Box and Intersecting It with the Bunny SubD 300
Stage 2: Displacing the Voronoi Cells Away from the Box's Center and Trimming the Cells with the Bunny SubD 301
Summary 303
References 304

Part II: Applications 305

Chapter 8: Weaving 307

Running This Chapter's Examples 308
Installing the Weaverbird Plugin 309
Attribution to the Stanford Bunny 309
Basket Weave from a Rectangular Grid 309
Stage 1: Create a 2D Rectangular Grid As a Template for the Weave 310
Stage 2: Finding Midpoints of the Grid Cell Edges 311
Stage 3: Deriving a Second Set of Cell Edge Midpoints, Elevated Along +Z 314
Stage 4: Linking the Two Sets of Cell Edge Midpoints into a Basket Weave and MultiPipe 315
Using Python to Form the Basket Weave Structure 319

Basket Weaving an Input Mesh 323
Double, Triple, and Twisted Variations of Basket Weave 327
Double Weave 328
Triple Weave 333
Twisted Double Weave 334
Triaxial Weave from a Triangular Grid 336
Using Python to Form the Triaxial Weave Structure 339
Triaxial Weaving an Input Mesh 340
Summary 345
Reference 346

Chapter 9: Sliceforms 347

Running This Chapter's Examples 347
Installing the Weaverbird Plugin 347
Attribution to the Stanford Bunny 348
What Is a Sliceform? 348
Generating a Sliceform from an Arbitrary Input Mesh 350
Stage 1: Computing the Bounding Radius of the Input Mesh 351
Stage 2: Creating a Stack of Planes for Slicing the Mesh 353
Stage 3: Forming Slices in One Direction 355
Stage 4: Extruding Slices for Thickness 357
Extending Stage 2: Slicing in Two Directions 360
Adjusting Inter-Slice Distances and Angles to Create Variations 363
Using Breps As Input Instead of Meshes for Sliceform Generation 367
Generating Hollow Sliceforms 368
Hollow Sliceforms from Input Meshes 369
Hollow Sliceforms from Input Breps 372
Make Mock Hollow Sliceforms from Interlocking Spiral Ribbons 374
Interlocking Spherical Ribbons 375
Interlocking Polygonal Ribbons 378
Summary 381
References 381

Chapter 10: Generating Models from Images 383

Running This Chapter's Examples 383

Attribution to Image Used As Inputs for Node Graphs 383

Generating Tiles from Images Using the "Image Sampler" Node 384

Stage 1: Creating Relief Surface from Points with Heights Proportional to Pixel Intensities 385

Stage 2: Creating a Flat Bottom for the Tile 391

Stage 3: Creating the Sidewalls of the Tile 393

Stage 4: Combining the Top, Sides, and the Bottom of the Tile 394

Generating Tiles from Images Using the "Import Image" Node 397

Stage 1: Creating a Relief Mesh from Pixel Intensities with the Import Image Node 398

Stage 2: Generating the Sidewalls of the Tile with the Help of Python 3 400

Stage 3: Creating a Flat Bottom for the Tile 406

Stage 4: Combining the Top, Sides, and Bottom of the Tile and Converting It into a Mesh 407

Comparing Tiles Generated Using Image Sampler vs. Import Image 407

Projecting Image Tiles onto Other Surfaces As Decorative Elements 410

Projecting an Image Tile onto a Sphere 410

Projecting Image Tile onto a Cylinder 413

Generating Mold Boxes from Image Tiles 417

Inverting Heights of the Relief Surface for the Mold 419

3D-Printed Lithophane (Darker Pixels Are Taller) 421

Summary 423

Chapter 11: L-System 425

Running This Chapter's Examples 425

What Is an L-System? 425

Rewriting Strings 427

Using Pen and Paper to Rewrite the String for a Koch Curve Variation 427

Using Python to Rewrite Strings 432

Implementing a Turtle Graphics Object to Create 2D Polylines (Continuous Line Drawing) Based on Expansion Strings 436

Two-Dimensional Continuous Line Variations 442

Adding Ability to the 2D Turtle to Lift the Pen Off the Page 446
Adding Ability to the 2D Turtle to Create Branches 449
Two-Dimensional Branching Tree Variations 454
Base Tree 454
Variation #1: Longer Branches 455
Variation #2: Left- or Right-Leaning Branches 457
Variation #3: More Branching on One Side 458
Variation #4: Bulb-Like (Dandelion) Trees 459
Taking the Turtle to 3D 461
Exploring 3D Tree Variations 463
Basic Tree with Random-ish Branching 463
Trees with Fourfold Symmetry 465
Generating a Canopy for the Fourfold Symmetric 3D Tree 469
Step 1: Isolating the Points Defining the Branches (Minus the Trunk) 471
Step 2: Finding the Convex Hull of the Branches 473
Step 3: Using “Shrink Wrap” to Create a Shell Hugging the Branches 473
Step 4: Using “Mesh Difference” to Carve the Convex Hull from the Shrink-Wrapped Shell to Create the Canopy 474
Step 5: Combining the Tree with the Canopy 476
Summary 476
References 478

Chapter 12: Generating City Skyline Models from Building Footprint (GIS) Data ... 479

Running This Chapter’s Examples 479
Installing Required Python Packages 480
Installing QGIS 482
Attribution to the Provided Data Sets 482
What Is Building Footprint Data? 483
Sourcing Building Footprint Data Online 483
Example: Downloading Building Footprint Data from NYC OpenData 483

Post-Processing Building Footprint Data in QGIS 485
Step 1: Importing the *.csv File in QGIS As a Delimited Text Layer 485
Step 2: Selecting an Area of Interest 488
Step 3: Exporting the Current Selection to a Spreadsheet 490
Step 4: Exporting the Current Selection to a Shapefile 493
Parsing and Previewing a Building Footprint Data Set with Third-Party Python Packages in Python 3 495
Step 1: Configuring Input and Output Ports for the Python 3 Script Node 495
Step 2: Creating Input and Output Values 497
Step 3: Checking References to Packages and Adding Import Statements 498
Step 4: Reading Contents of a Shapefile with Python 3 Using `pyshp` 499
Step 5: Plotting Shape Polygons with Python 3 Using `matplotlib` 500
Step 6: Reading the Contents of a Spreadsheet with Python 3 Using `pandas` 503
Generating Buildings 507
Stage 1: Parsing the Spreadsheet and Filling in Missing Values 507
Stage 2: Turning the Polylines Outlining Building Footprints into Trimmed Surfaces 508
Stage 3: Moving the Footprint Surfaces to Center at the Origin (0, 0, 0) 510
Stage 4: Extruding Footprint Surfaces into Buildings and Offset Them Along Z Based on Ground Elevations 511
Generating the Havana Cityscape 515
Summary 518
Reference 518

Chapter 13: Packaging Rhino Python and Grasshopper Scripts As Plugins for Publication 519

Running This Chapter's Examples 519
Building a Grasshopper Plugin 520
Creating a ScriptEditor Project 520
Creating SVG Files As Icons 523
Scripts As Individual Components 526
Creating Installers and Packages 536

Installing the Grasshopper Plugin You've Just Created 538
Verifying the Grasshopper Plugin 540
Combining Multiple Script Nodes into One Component 542
Pushing to the Rhino Package Server (Yak) 552
Pushing to the Yak Test Server 554
Pushing (for Real) to the Yak Package Server 555
Hosting on Food4Rhino 555
Building a Rhino Command 555
Converting a Node Graph to a Python 3 Script Node 555
Adding Contextual Inputs and Output(s) 558
Linking the GH Definition Under Commands/ in a ScriptEditor Project 560
Installing the Rhino Command You've Just Built 561
Verifying the Rhino Command 561
Publishing the Rhino Command 564
Remote Control Panel 564
Adding Input Widgets to the Remote Control Panel 565
Customizing the Remote Control Panel 568
Juggling Multiple Grasshopper Definitions with Remote Control Panels 572
Summary 573
References 573

Index 575

About the Author



Isabel Lupiani is a software engineer by day and maker by night, who enjoys handcrafting 3D models as much as procedurally generating them. She received her MS in Computer Science from Georgia Tech and has worked at several game studios in the past as an AI engineer for PC/Xbox games. Isabel most recently worked as a lead AI engineer in the simulation industry.

About the Technical Reviewers



Simon Jackson is a long-time software engineer and architect with many years of game development experience, as well as an author of several game development titles. He loves to both create game projects and lend a hand to help educate others, whether it's via a blog, vlog, user group, or major speaking event.

His primary focus at the moment is with the Reality Toolkit project, which is aimed at building a cross-platform Mixed Reality framework to enable both VR and AR developers to build efficient solutions in Unity and then build/distribute them to as many platforms as possible. He is also a board member of the MonoGame Foundation, aiming to secure and promote open source game development for all developers.



Guilherme Jardim is a senior software engineer and computational design specialist based in Brazil, with professional experience spanning full-stack software engineering, real-time 3D visualization, and digital twin platforms. He develops large-scale digital twin solutions involving Rhino, Grasshopper, React, Python, Three.js, GIS data, LiDAR processing, and real-time spatial analytics for universities and enterprise property portfolios across international projects.

Guilherme has extensive experience with computational modeling, parametric design, and digital fabrication workflows and has spent more than a decade working with 3D design, additive manufacturing, and generative modeling across the architecture, product design, and jewelry industries.

In addition to his industry work, he has taught modeling and digital fabrication techniques to students, professionals, and businesses for over nine years through consulting, private instruction, and university lectures.

Acknowledgments

First and foremost, a big thank you to all the readers out there—your support inspires and motivates me to keep writing. I am also grateful to Spandana Chatterjee, Abirami D S, Joseph Quatela, Gryffin Winkler, Sowmya Thodur, and the rest of the Apress and Springer Nature team for guiding me through the publication process. A special thanks to the technical reviewers for testing everything and making the book the best it can be. Finally, thanks to my husband James, my daughter Zoe, and my son Caleb for your continued love and support.

Introduction

It's been many years since I created a Grasshopper (GH) script for the first time, and I still remember the excitement—building a node graph with intuitive drag-and-drops, and with just a few sliders here and there, an explosion of forms appears in the Rhino viewport. This is what I want to show you in this book—how to start with an idea, an interesting silhouette or arrangement of shapes, and translate it into a parametric model in GH that allows the most diverse and aesthetic variations.

The book is organized into two parts. The first part (Chapters 1–7), gets you up to speed with Grasshopper and Python 3 basics, and shows you how to generate models like attractors, tessellations, spirals, voxels, cracks, and wireframes. The second part covers Chapters 8–13 and provides five advanced application workshops, including weaving meshes, creating sliceforms, generating Breps from images, making fractals and tree-like canopies from L-systems, and creating skylines from building footprint (Geographic Information System, GIS) data. The final chapter takes you through packaging GH scripts into apps suitable for online publication.

I've intentionally structured the book to utilize built-in GH components as much as possible, with a few custom Python 3 Script nodes—although there is a plethora of GH apps on food4rhino.com that will do seemingly anything, there is value in learning how to do things yourself, for when your idea is so unique that no app will do precisely what you want. In addition, apps often become out of date with new Rhino releases, with no guarantee when or if they will be updated. That said, we'll still take advantage of a few popular GH plugins such as LunchBox and Weaverbird throughout the book.

Chapter Outline

In Chapter 1, you'll learn essential basics for scripting in Grasshopper, beginning with an overview of the editor interface and how to locate and place objects, followed by a deep dive on wires and components, the meaning of their structures and colors, and ways to preview and customize their inputs and outputs. You'll also pick up strategies for quickly creating commonly used types like numbers, Booleans, points, curves, and surfaces. In

addition, you'll learn how to perform basic transforms like move, rotate, and scale on geometry and combine these operations in creative ways. Along the way, you'll discover how to access data from Rhino and edit it in Grasshopper and vice versa.

Chapter 2 introduces more advanced scripting topics, starting with *data conversion*, which is how Grasshopper interoperates between compatible but non-identical types, followed by editing *lists* using nodes like Range, Series, Cull Pattern, Dispatch, and Weave. After that we'll go over *data matching,* which is the way GH deals with inputs of mismatched sizes. You'll then find out how to access and edit *data trees*, a nested data container in GH, followed by developing a custom Python 3 Script node that generates a seashell curve. Finally, we'll conclude the chapter with tips on how to group and align components and wires to keep things tidy.

In Chapter 3, you'll create a series of *attractors,* which are arrays of objects transformed based on their proximities to one or more objects called *emitters*. Starting with the simplest case, a 1D attractor with one row of geometry influenced by one emitter point, you'll quickly generalize the idea to 2D rectangular and polar grids, before extruding along Z to add a third dimension. Along the way, you'll experiment with scaling the array objects in different ways and contracting and expanding them with vector fields.

Chapter 4 shows you several methods for creating tessellations. The first is to overlay 2D grids of primitive shapes on top of one another and then offset and rotate the shapes to create variations. The second is to slide points along the edges of a shape and then link them together to build motifs. The third method builds on the second, where points glide along edges as well as diagonals of a shape before connecting to form star-like motifs. By varying how much each point is moved and composing larger motifs from mirrored copies of smaller ones, many surprising tessellations emerge ranging in style from geometric to Islamic.

In Chapter 5 you'll discover how to generate spiral and looping curves and then use them as the bases for creating more complex geometry like Möbius strips, seashells, spiral stairs, etc.

Chapter 6 explores abstract structures reminiscent of Habitat 67 through stacking 3D primitives. We'll also explore the concept from the opposite angle, by *voxelizing* an organic model by making it pixelated in the 3D sense.

Chapter 7 shows you how to virtually "smash" surfaces to produce cracked pieces and stylistically remove faces from meshes to create a skeletal or wireframe effect.

Chapter 8 takes you on a journey of creating "weaved" versions of meshes, inspired by baskets and textiles.

In Chapter 9 you'll create *sliceforms* by cutting interlocking cross-sections of meshes at various angles and intervals.

Chapter 10 takes you through generating Breps from images, which could then be edited into tiles, lithophanes, and molds and projected onto the surfaces of other objects as decorative elements.

In Chapter 11 you'll develop custom Python 3 Scripts nodes generate fractals and tree-like canopies using L-systems. Starting with a pen-and-paper example, you'll learn how L-systems work from the ground up and then develop a turtle graphics–based system in Python 3 to interpret and generate curves and surfaces based on L-system grammar.

In Chapter 12 you'll create city skyline models from building footprints, a form of Geographic Information System (GIS) data. You'll learn how to source data sets online and prepare them in the open source software QGIS to extract shape files (*.shp) containing the buildings' cross-sectional curves and Excel spreadsheets with their heights, which you'll then use to generate skyline models in Grasshopper.

In Chapter 13 you'll learn how to package Rhino Python and Grasshopper scripts into apps suitable for sharing with other users and publishing online.

Suggested Roadmap

If you are new to Grasshopper or Python 3 or are in need of a refresher, a linear read through from Chapters 1 to 7 is suggested. You can then pick and choose between Chapters 8 to 12 based on which application areas interest you, as they are standalone workshops that can be read in any order. Chapter 13 can be skipped if you do not plan to package or publish your scripts as apps.

Who the Book Is For

This book is for Rhino users who want to use Grasshopper and Python 3 to procedurally generate 2D and 3D content for parametric design and form finding. These users include architects, landscape designers, software developers, industrial designers, product designers, jewelry designers, naval architects, as well as makers and hobbyists. The targeted audience either already use Rhino or are interested in learning the basics on their own. In addition, the ideal readers will already know some Python or are motivated to pick it up along the way.

PART I

Building Blocks

CHAPTER 1

Grasshopper and Rhino Python Quick Start: Part 1—The Basics

In this chapter, you'll quickly pick up essential basics for building parametric models in Grasshopper. Beginning with an overview of the scripting workflow, including terminology, the editor interface, and how to locate and place objects, we progress to learning about components—the fundamental building blocks of Grasshopper—starting from their general structure and appearance, how to preview and customize inputs and outputs, to strategies for creating common input values such as numbers, Booleans, points, curves, and surfaces. Along the way, we'll discuss how to reference data from Rhino and edit it further in Grasshopper. To wrap up the chapter, we'll go over how to perform basic transforms like move, rotate, and scale and combine them to create interesting variations in a parametric way.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch1` folder, where you'll find `*.gh` files named after the figures in which they appear along with a short description, for example, `Figure1-8_component_structure.gh`. The sample files have been tested in Rhino 9 on Windows. As you read the chapter, you can build your own node graphs by following the steps described in each section or start with the downloaded `*.gh` files and experiment by modifying their parameters.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_1

Overview of Scripting in Grasshopper

Grasshopper (GH) is Rhino's visual scripting environment. As shown in Figure 1-1, you create a program ("definition") to describe the steps for constructing a parametric model, by connecting a series of "components" with "wires" on a "canvas" (the GH editor background). A component may be a container that holds data like number(s), piece(s) of geometry, etc. or an operation for editing or analyzing data, like rotating a Brep or finding the length of a curve.

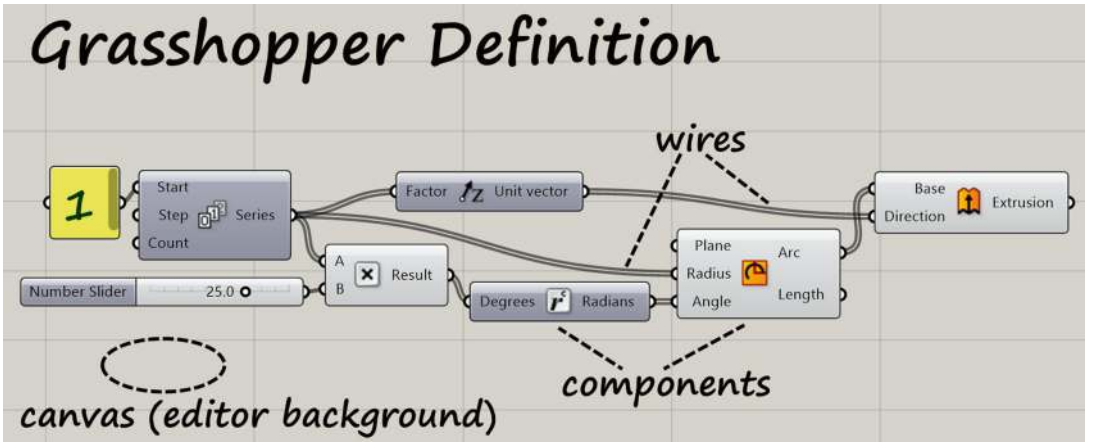


Figure 1-1. *You create a Grasshopper definition by connecting a series of components with wires on the canvas (editor background)*

Why Use Grasshopper?

Chaining components in Grasshopper is like performing a series of edits or executing a series of commands in the Rhino viewport—except you "write down" the steps in a GH definition so they can be run over and over. You also compose your definition in such a way that parameter values for intermediate steps are changeable via sliders and knobs. As you drag the sliders, Grasshopper instantly updates the preview for the geometry output by the definition in the Rhino viewport, hence allowing you to create *variations for form finding quickly and interactively* that are *easily sharable with others*.

There is a GH component for almost every Rhino command, which lets you translate any Rhino design into a GH definition in a straightforward way. Moreover, the power of scripting means *you can create models in Grasshopper that would otherwise*

be too complex or tedious to produce by hand. The fact that you can visualize all the intermediate steps of a GH definition in the Rhino viewport also makes it a great tool for communicating your design process or teaching others how a model is constructed. In addition, you can reuse parts of a GH definition as jump-off points for new designs via copying and pasting or share them with others for creative collaboration.

Workflow

The workflow in Grasshopper entails the following: You first come up with an overall design for a model, for example, a surface lofted from a stacked column of hexagons, and then derive the sequence of edits that will lead to that design. After that you translate the sequence into Grasshopper components and identify key parameters that when varied produce the most meaningful and aesthetic variations.

Terminology

In this book, I will use the following terms interchangeably. You'll often find these terms used in online Grasshopper communities as well:

- **GH** as a shorthand for Grasshopper
- Grasshopper **definition**, or Grasshopper file, script, node graph, graph, program
- **Component**, or node
- **Data tree**, or tree

Don't worry if you're unfamiliar with some of these terms—we'll discuss them in detail throughout this chapter and the next.

Launching Grasshopper

You can launch the Grasshopper editor with the Rhino command "Grasshopper", as shown in Figure 1-2.

Note Grasshopper has been bundled with Rhino since V6. You can download the installer for Grasshopper as a plugin for Rhino 4 and 5 at `https://www.grasshopper3d.com/page/download-1`.

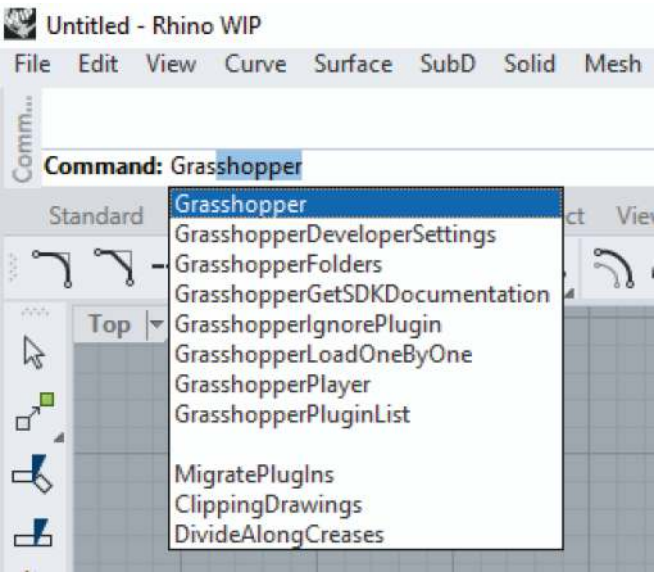


Figure 1-2. *You can use the Rhino command Grasshopper to launch the Grasshopper editor*

Grasshopper Editor Interface

Figure 1-3 shows an overview of the Grasshopper editor interface. The upper left of the window displays the name of the Grasshopper definition currently in view. Below that is the main menu bar, with options for saving files, customizing display of components, and so on, followed by component tabs. Each tab corresponds to one category of components (e.g., Curve, Surface, etc.), which is further divided into panels.

The gridded area with a beige background at the bottom is called the "canvas," which is where you deploy components to and edit the definition that is currently active. Note that there is no "run" button, since the active GH definition is automatically refreshed upon each edit and previewed in real time in the Rhino viewport, unless otherwise disabled. You can have multiple GH definitions open at the same time, although only one will be currently active and displayed on the canvas. You can click the file name at the upper-right corner of the editor to bring up the list of open files and then click to jump to a different file.

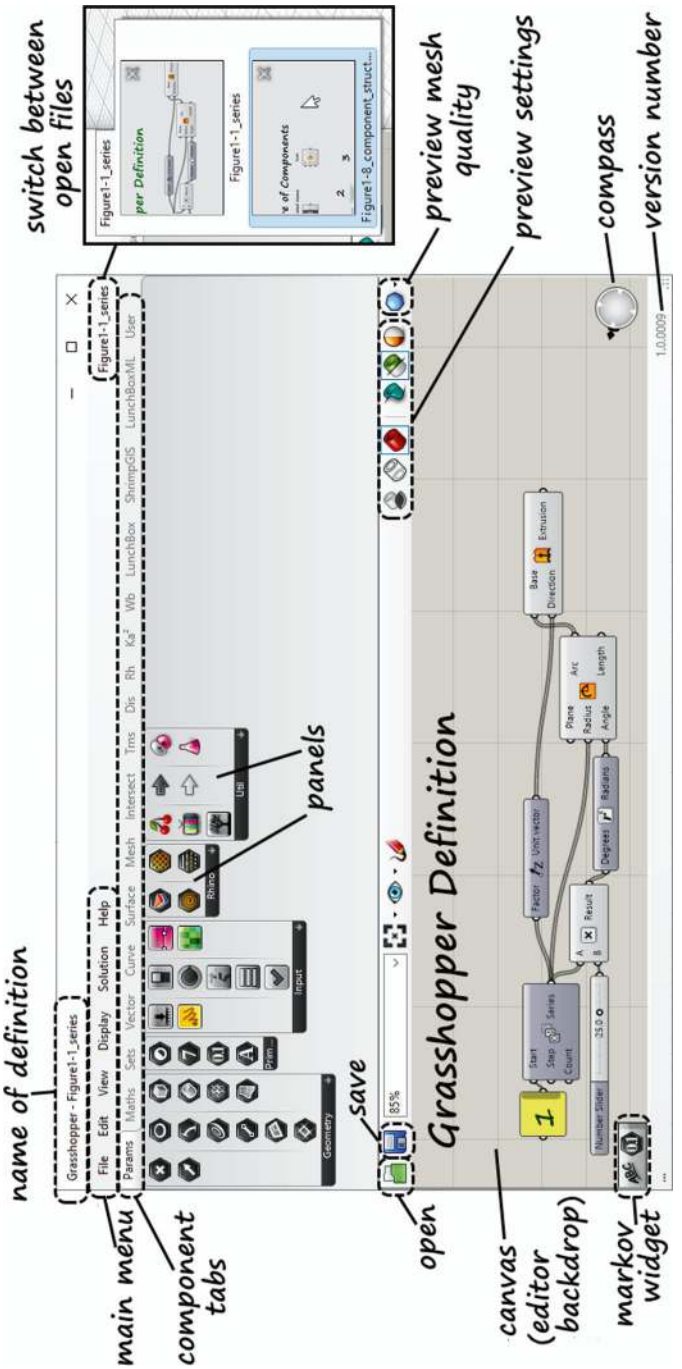


Figure 1-3. *Overview of the Grasshopper editor interface*

At the upper right of the canvas is a row of buttons for configuring preview options and mesh preview quality, which we'll go over in more detail later in this chapter. After interacting with Grasshopper for a while, you'll see a widget with a row of component icons appear at the lower-left corner of the canvas—this is called the "Markov widget," which records the history of your edits and tries to predict what component you might use next. The circular widget at the lower-right corner of the canvas is called the "compass," which points you back toward the node graph in case you get lost panning and zooming around the editor. Finally, the Grasshopper version number is shown at the extreme bottom right of the editor window—as shown, the version used as of this writing is 1.0.0009.

Locating a Component in the UI

In this section, we'll take a closer look at how to locate components in the Grasshopper editor UI and place them on the canvas. Grasshopper components are categorized and sorted into tabs as shown in Figure 1-4, for example, Params, Maths, Sets, etc. Each tab is further organized into panels representing sub-categories. A panel will by default only show a subset of the most commonly used components. To see the whole set, click the black bar at the bottom of the panel containing that panel's name. For example, as shown in Figure 1-4, the (1) Vector tab is in focus, which is divided into the following panels: Field, Grid, (2) Plane, Point, and Vector. To see all the components in the (2) Plane panel, click the black bar containing its name to expand it.

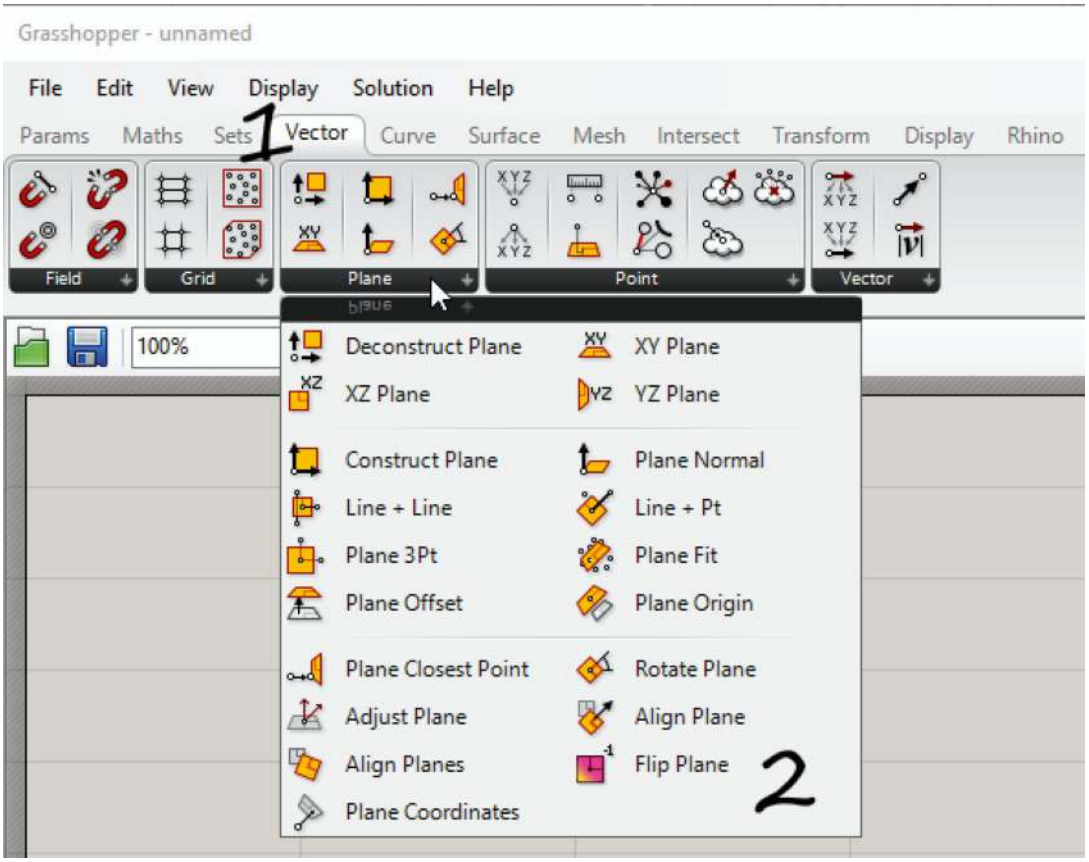


Figure 1-4. *Grasshopper components are categorized and sorted into tabs. Each tab is further divided into panels representing sub-categories. As shown, the (1) Vector tab is in focus, with its (2) Plane panel expanded*

If you come across an unfamiliar component while going through someone else's script or simply can't remember where a component is located, you can hold down Ctrl-Alt LMB over a component already *on the canvas* to pan to its icon in the GH editor UI. The icon will be circled in an overlay, with its tab pointed to by a red arrow, as shown in Figure 1-5.

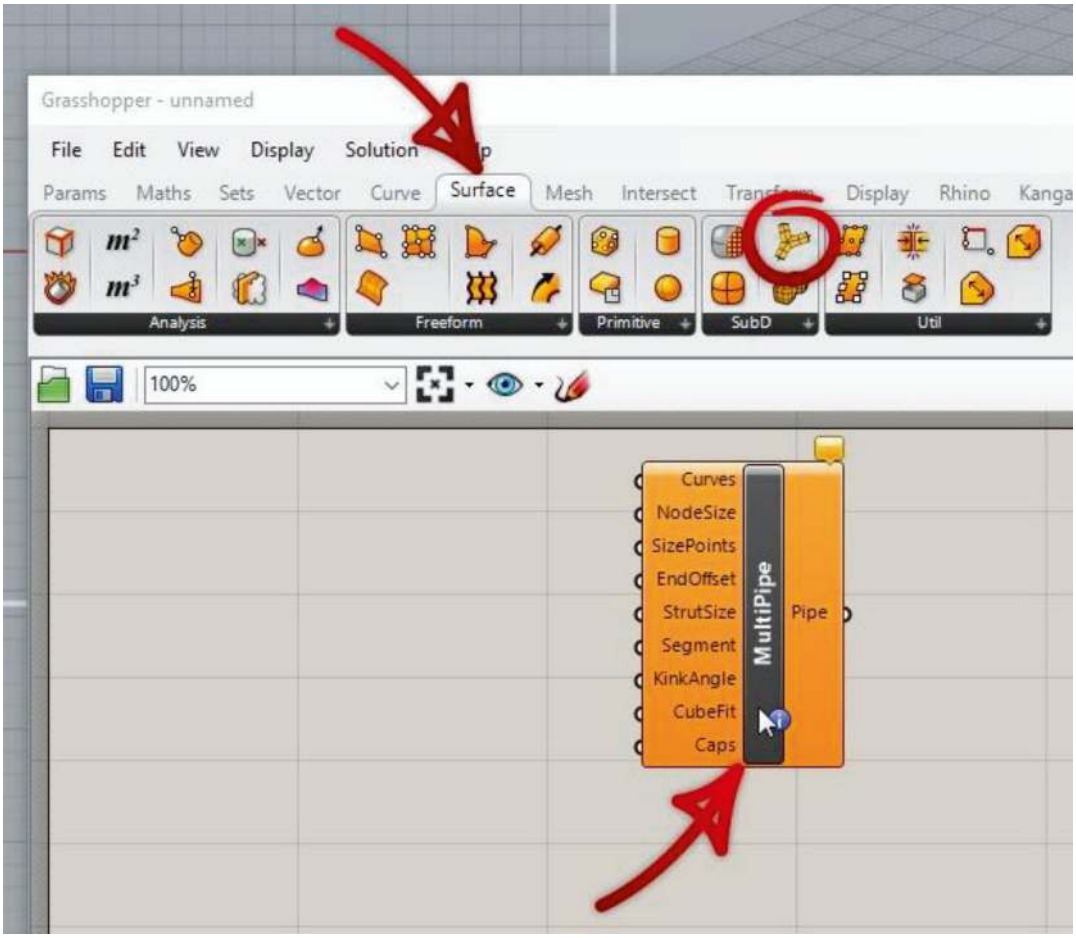


Figure 1-5. *You can look up the icon of a component already on the canvas by holding down Ctrl-Alt LMB over it. The editor window will automatically pan to the appropriate tab and point to it with a red arrow in an overlay, with the icon circled. The overlay disappears when you let go*

Placing a Component on the Canvas

There are several ways to place a component on the canvas. The first is to locate the component's icon in the editor UI and click and drag it to the canvas (notice a "+" sign will appear next to the cursor). Once the cursor is over the spot where you intend to place the component, let go LMB to confirm. An example of placing a Number Slider by clicking and dragging from the Params tab ➤ Input panel is shown in Figure 1-6.

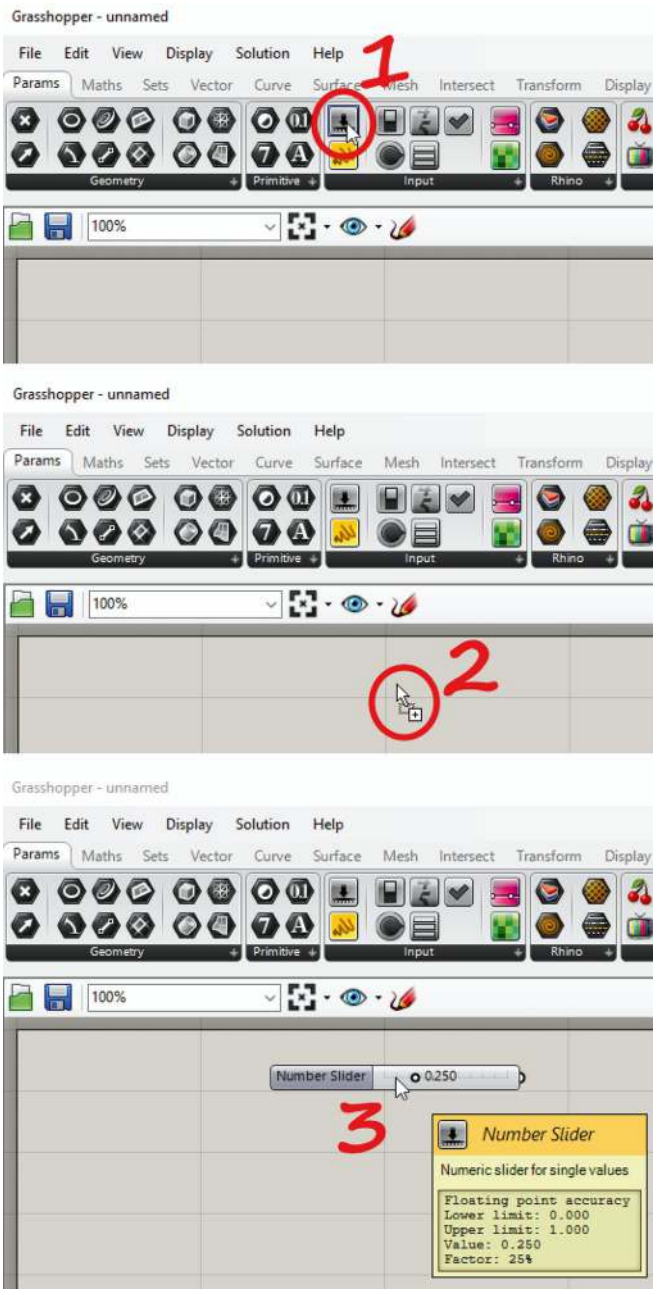


Figure 1-6. *(1) Hold down LMB over a component's icon, drag it to the canvas (2) where you intend to place it (notice a plus sign will appear next to the cursor), and (3) let go to confirm*

The second way to place a component is to double-click to bring up the search box, type in a keyword, and click a component in the search results to place it where the search box is currently positioned, as shown in Figure 1-7. The search results are sorted such that the most relevant appear at the bottom. If you are unsure which component is the right match, you can hover over the items in the results to see their tooltips. Alternatively, you can initiate a search via the spacebar. The difference is that the search box and ultimately the component to be placed will appear where you double-click, whereas the spacebar will always call up the search box at the center of the canvas.



Figure 1-7. *Looking up and placing a component via the search box. (1) Double-click anywhere on the canvas to summon the search box at that spot. (2) Type in a search term (e.g., partial name for a component). (3) Click an entry in the search results to place that component (4) where the search box is*

The Anatomy of Components

Components are the fundamental building blocks in Grasshopper. In this section, we'll go over various properties of components, including their appearances, the behaviors of their inputs and outputs, how to create common types of inputs, and how to inspect and preview outputs.

General Structure

A Grasshopper component is typically a rectangular box with inputs on the left, name or icon in the middle, and outputs on the right, as shown in Figure 1-8. You hold down LMB and drag from an output of one component to link it to an input of another and Ctrl-LMB drag to disconnect. Data flows through wires from left to right. It is possible to connect output(s) to input(s) one to one, many to one (via Shift-LMB drag), or one to many. The outputs and inputs must have compatible data types to be linked together.

Components with no input or output often have a jagged edge on that side, for example, Params ➤ Input ➤ Colour Picker (1c) has no input, while Display ➤ Dimensions ➤ Tag (2a) and Vector Display Ex (2b) have no output. Sometimes the missing input or output just has missing wire ports with no jaggedness, like Param ➤ Input ➤ Number Slider (1a) and Colour Swatch (1b), which have no input, and Display ➤ Graphic ➤ Image Gallery (3a), which has neither input nor output.

Params ➤ Util ➤ Relay (4a) is a special component that does nothing on its own and whose only purpose is to serve as a wire hub. You can double-click any wire to insert a Relay and then right-click to type in a name to label that wire. You can also Shift-LMB drag many wires into a Relay with a single wire coming out as a way to make the graph look neater. If you have one output that is linked to many other inputs, you could connect that output to one Relay, have multiple wires coming out of the Relay, and then move the Relay around until objects stop overlapping. To make a Relay go away, simply double-click it.

Though most components are rectangular, some are circular, like Params ➤ Input ➤ Clock (1d) and Params ➤ Input ➤ Color Wheel (4d). We'll go over how to display a component's name as full text, abbreviated text, or icon in the section "Name as Text or Icon."

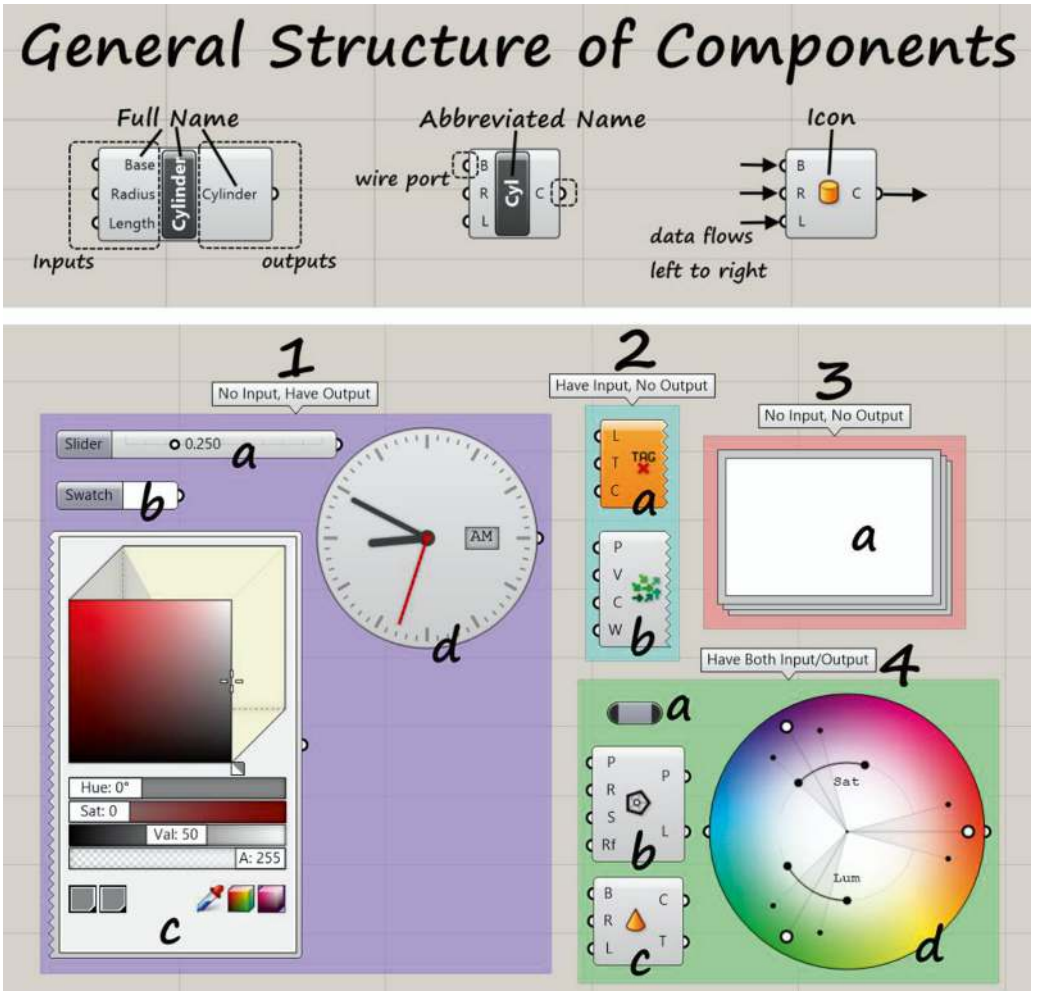


Figure 1-8. *GH components are typically rectangular, with inputs on the left, name or icon in the middle, and outputs on the right (though there are some exceptions, like 1d and 4d). Components are linked together via wires, with data flowing from left to right. A component may have (1) only output(s), (2) only input(s), (3) neither input or output, or (4) both. The individual components within each group are explained in the section text that follows*

Color

You can tell the state of a component by its color as described in this section, with corresponding examples shown as 1–6 in Figure 1-9.

A **light gray** color with black center indicates **normal** status, which means the component is functioning without warning or error and all its inputs are present and valid.

An **orange** color means the component has **warning(s)**. One or more of the component's inputs may be missing, or the component may have encountered a minor issue, such as producing an empty curve among its list of output curves, which does not prevent the script from producing an overall result. Hovering over the balloon at the component's upper right will display a tooltip containing some of the warning messages. To display all warnings, click on the balloon.

A **red** color indicates **error(s)**. When this happens, the component has encountered a critical error, like division by zero or the wrong input type. You can hover over or click the balloon at the upper right to see the error messages, which will help you diagnose the problem. Note that when a component is red, the downstream nodes (in the portion of the graph to its right) will likely stop working.

A **dark gray** color with **black center** tells us the component produces **no viewport preview**, or the preview has been toggled off. A component could have no preview because its outputs have no inherent visual representations (e.g., numbers) or could only be drawn with special components (e.g., vectors, with Display ➤ Vector ➤ Vector Display Ex). A component's preview can be toggled off via right-clicking the component's name ➤ Preview, or Ctrl-Q while the component is selected. Repeating either action once more will toggle the preview back on.

A **dark gray** color with **gray center and faded text** tells us the component has been **disabled**. The component will not run, therefore causing components downstream to stop functioning. You can disable a component via right-clicking its name ➤ Enabled.

You'll find examples of all component colors in Figure 1-9. (1) is a Circle component in light gray (normal) since all its inputs have default values. (2a) shows Divide Curve in orange due to missing Curve input. To confirm, hover over the balloon at its upper right to bring up the tooltip, which shows the warning "Input parameter Curve failed to collect data." (2b) shows Series in orange due to a bad Count input of –10, which should have been positive—note that Series is able to cope with this and output an empty list rather than erroring out, although this will likely cause invalid results downstream. (3a) illustrates a Division component in error due to dividing by zero, while (3b) shows Divide Curve with an input of the wrong type (a surface being linked to its Curve input)—to verify this, click the balloon at its upper right, which says "Data conversion failed from Surface to Curve." (4a) shows Unit X in dark gray with a black center due to its output (a vector) being a type that is not automatically previewed in the viewport. The copy of Circle in (4b) is also dark gray with a black center, since its preview has been toggled off. The Circle in (5), however, is dark gray with a *gray* center since it is disabled (not run).

Lastly, regardless of the status of the component, if it is selected, it will carry an additional green tint on top of its current color. (6) in Figure 1-9 shows one example where a Circle component in normal status is selected and has a **green** tint. You can click to select one component or select multiple components using box select, by holding down LMB and drawing a box around them.

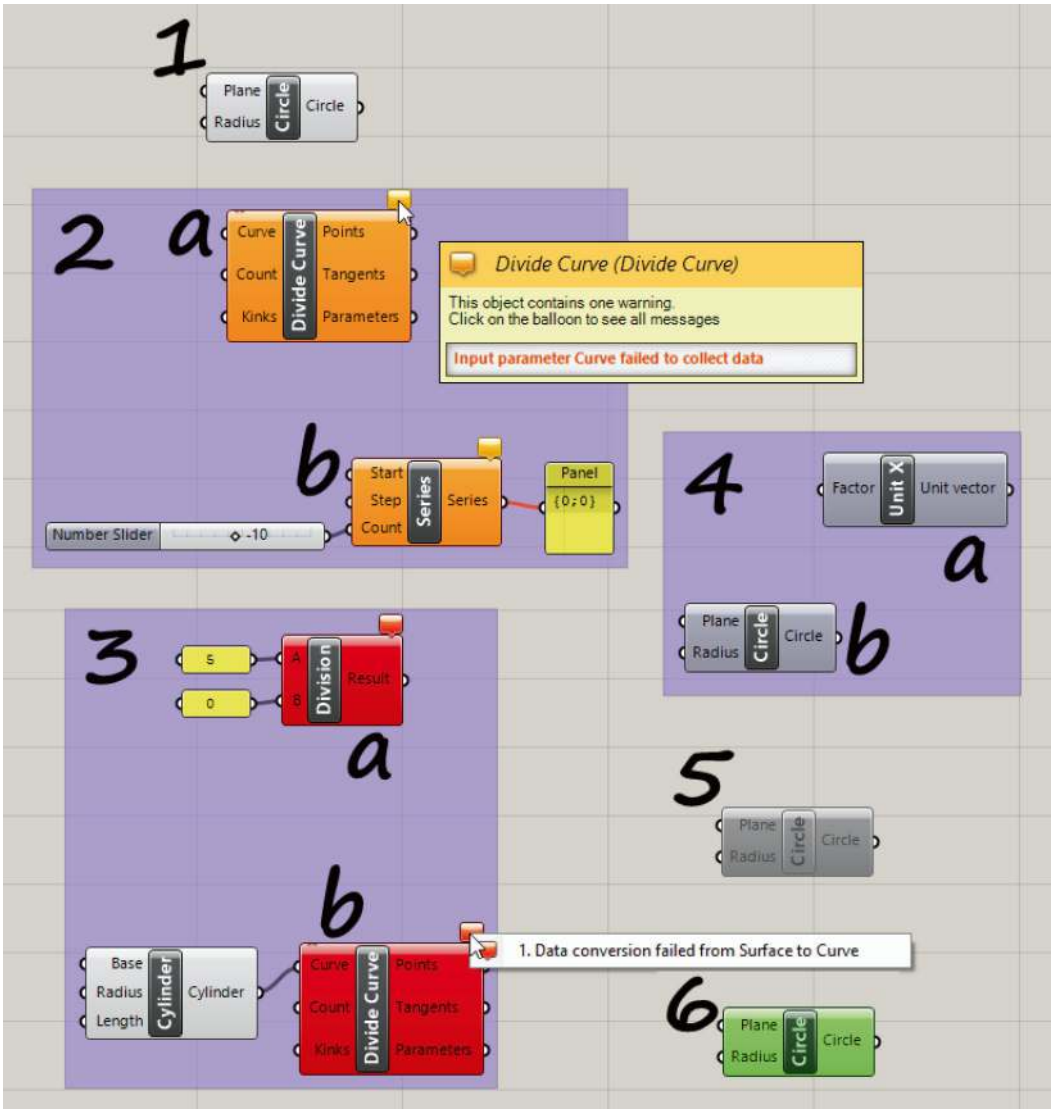


Figure 1-9. *Possible component colors in Grasshopper. (1) Light gray: normal; (2) orange: warning; (3) red: error; (4) dark gray with black center: no preview or preview is toggled off; (5) dark gray with gray center: disabled. (6) Selecting a component gives it a green tint*

Name as Text or Icon

You can choose whether to display the names of components as text or icons via Display ➤ Draw Icons, which when off (default) shows the names as text, for example, "Cylinder" (1 in Figure 1-10) or "Cyl" (7). When "Draw Icons" is toggled on, the names are instead shown as icons (2 and 3).

You can customize whether text is shown as full text or abbreviation via a separate option: Display ➤ Draw Full Names, which will impact both the names of components and their inputs and outputs. When "Draw Full Names" is on (default), names are shown as full text, for example, "Cylinder" (1 in Figure 1-10) and "Base" (1, 2, 4). When "Draw Full Names" is off, component names are shown as abbreviations, e.g., "Cyl" instead of "Cylinder" (7), whereas inputs and outputs are represented by their first letters, for example, "B" instead of "Base" (3, 6, 7).

You can right-click a component's center and enter a custom name at the top of the context menu that appears (4 and 5). Note that the custom name will only affect this particular copy of the Cylinder component, but not any future copies of Cylinder created via the search box or dragged down from the component tabs (those will follow script-wise settings). If you click the copy of the component with the custom name and Copy/Paste it, it will retain the custom name, however.

To override the script-wise setting and make a component's name draw as text even when Display ➤ Draw Icons is on, you can right-click the component's center and toggle the button to the right of the name field (5 in Figure 1-10).

Preferring text or icon is a matter of personal taste—some like icons for their compactness, while others prefer the self-explanatory nature of text. In this book I will tend to favor text since it is easier to look up components depicted in figures, but prefer icons when space is limited. I encourage you to experiment with different settings and find what works for you.

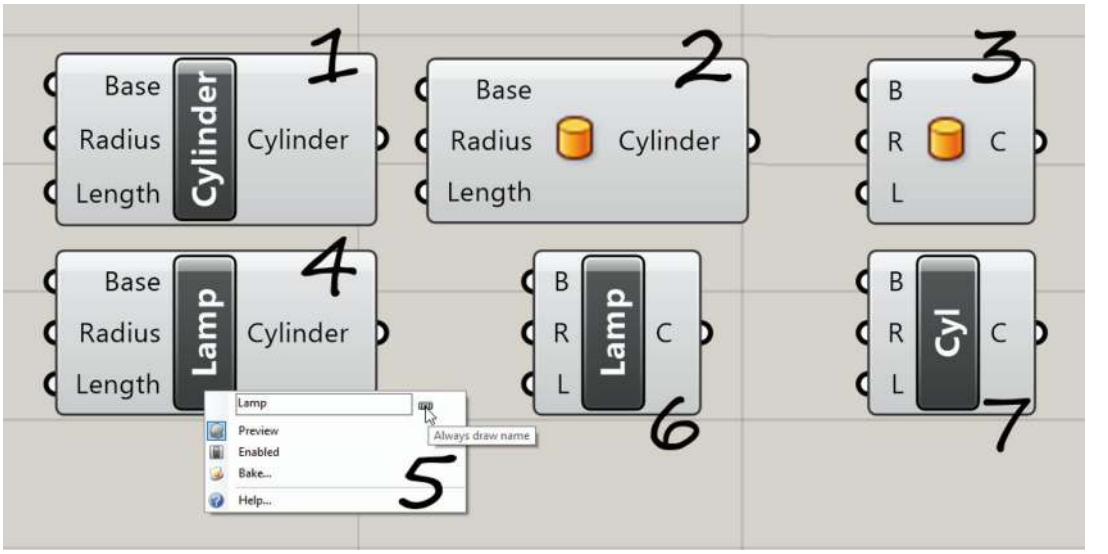


Figure 1-10. *Displaying names of components and their inputs as full text, abbreviations, or icons. (1) Draw Icons—off, Draw Full Names—on (default). (2) Draw Icons—on, Draw Full Names—on. (3) Draw Icons—on, Draw Full Names—off. (4) Right-click a component to enter a custom name. (5) Override the script-wise "Draw Icons" setting by toggling the "Always draw name" button. (6) is (3) with a custom name and "Always draw name" on. (7) (3) Draw Icons—off, Draw Full Names—off*

Component Outputs

In this section, we'll take a closer look at the data flowing out of components. We'll start by reviewing several methods for inspecting and previewing component outputs, followed by a discussion on how to add or delete outputs.

Previewing a Component's Output in the Rhino Viewport

You can customize which components' outputs are previewed as well as their appearances in Grasshopper. By default, the outputs of *all* components currently on the canvas are previewed in the viewport, which is like visualizing every step of building a parametric model all at once! To change the script-wise settings of which components are drawn and how, we can use the row of buttons at the upper right of the canvas, as shown in Figure 1-11. From left to right, the buttons are (1) no preview, (2) wireframe, (3) shaded, (4) drawing a boundary on canvas to indicate which components to preview, (5) preview only selected components, (6) preview colors, and (7) preview mesh quality.



Figure 1-11. *Grasshopper preview options. (1) No preview. (2) Wireframe. (3) Shaded. (4) Draw a lasso outline on the canvas to select the component(s) to preview. (5) Preview only selected component(s). (6) Document preview settings (colors). (7) Preview mesh quality*

To toggle the preview of one component, right-click its name (or icon) ➤ Preview. To toggle the preview of multiple components, first select them by Shift-clicking each component, right-clicking within a group's outline ➤ Select All, or clicking and dragging a box around the components. Once the components are selected, use Ctrl-Q to toggle their preview.

Figure 1-12 shows the results of applying different preview options to the same script. The script creates two concentric arcs and a ruled surface between them. By default, the outputs of all components are previewed in the Rhino viewport (top). By clicking button #5 in Figure 1-11, only the selected component's output is previewed (middle), which as shown is the inner arc. By clicking button #4 in Figure 1-11, Grasshopper will prompt you to lasso-select the component(s) you want to preview and show only those components' output in the viewport (bottom), which as shown are the two arcs.

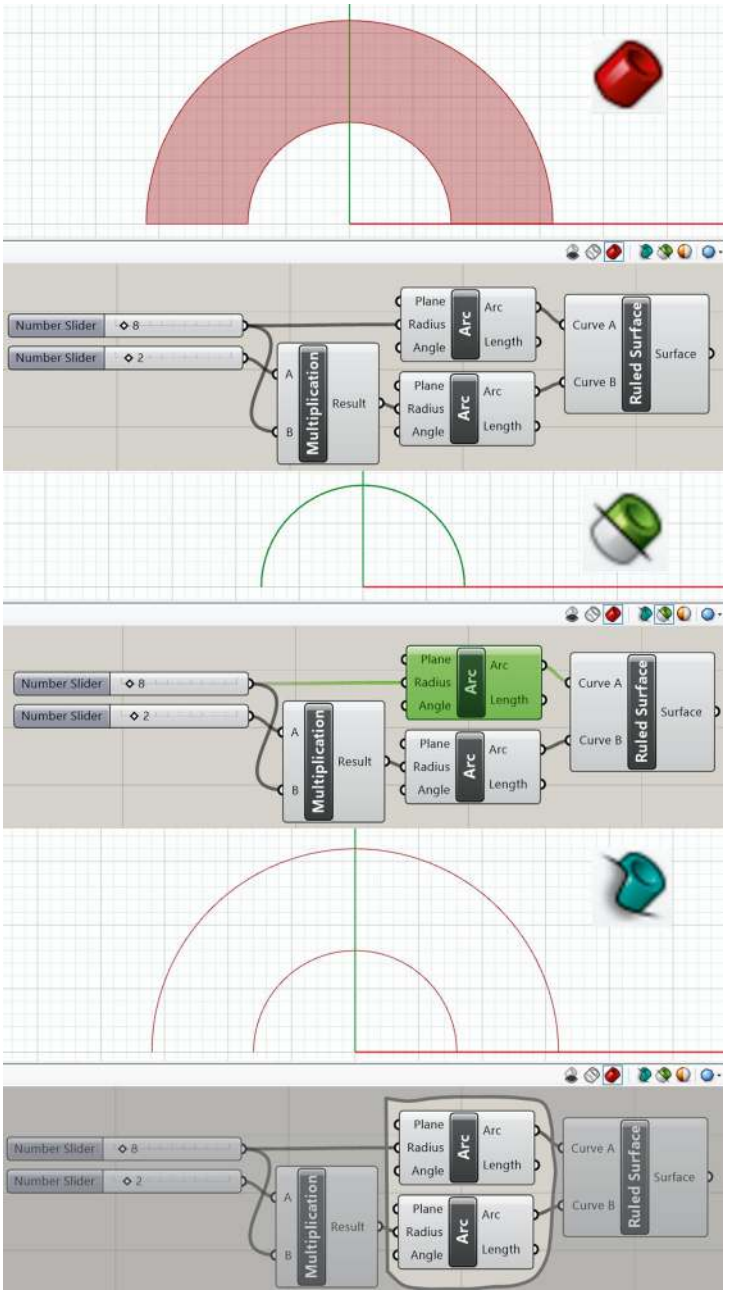


Figure 1-12. *(Top to bottom) Preview shaded, selected only, and lasso-selected on canvas only*

Inspecting a Component's Output with a Panel

For components that output geometry, previewing in the viewport is the obvious go-to. However, many other components produce textual outputs, such as the results of a math expression. The quickest way to preview text-based output is through a Panel. To do so, create a Panel and then connect any output to it (4 in Figure 1-13). Panels are great for checking the structure of the output data when you expect a list or a data tree as well, as we'll see soon.

You can create a Panel by typing either a '"' (double quote) or the word "panel" in the search box (1 and 2 in Figure 1-13) or dragging one down to the canvas from Params ➤ Input ➤ Panel (3).

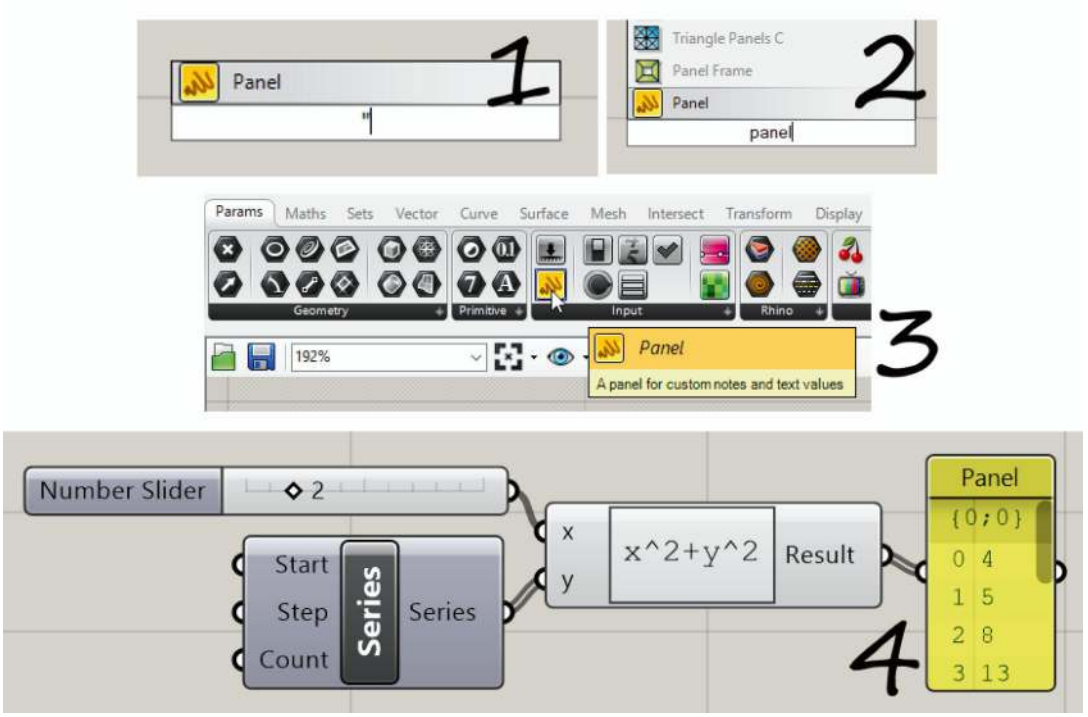


Figure 1-13. *Using a Panel to inspect a component's output. You can create a Panel by (1) typing a '"' (double quote) or (2) the word "panel" in the search box or (3) dragging one down from Params ➤ Input. After that you can simply (4) connect an output from any component to the Panel to see its content (as shown, the Result of a Maths ➤ Script ➤ Expression)*

Adding or Deleting Outputs

Perhaps not immediately obvious, you could add outputs to a component and later remove them if need be. Let's work through an example in Figure 1-14. We'll create a square curve, explode its edges, and use a modified List Item component with four outputs to access each individual edge. Let's start with a (2) Curve ➤ Primitive ➤ Polygon and use a (1) Panel set to 4 to control its Segments input to create a curve with four sides. Next, we feed this curve to (3) Curve ➤ Util ➤ Explode to get a list of its edges via the Segments output. We can use (4) Sets ➤ List ➤ List Item to retrieve an individual edge from Segments; however, it would be nice to grab all four edges at once. To achieve this, you could connect Segments to four List Items and set the Index of each to 0, 1, 2, and 3, respectively. There is a better way, however—if you keep scrolling the mouse wheel to zoom in on List Item, you'll see a "+" sign appear (as shown at the top of Figure 1-14). Clicking the "+" will create a new output named +1—we'll click "+" two more times, to create two more outputs, +2 and +3, for a total of four outputs (5 in Figure 1-14).

Since we did not hook up a value to List Item's Index input, it has a default value of 0; therefore, the first output i gives us the first item of the input list (at index 0), corresponding to the first edge of the square. The three additional outputs +1, +2, and +3 that we've created will output the subsequent three items at indices 1, 2, and 3, which are the second, third, and fourth edges of the square.

For convenience, we'll connect four Params ➤ Geometry ➤ Curve components to List Item's four outputs (6–9 in Figure 1-14). This way we can set Grasshopper to "Only draw preview geometry for selected objects" and then click each of the four Curve components to preview them one by one (shown as the four highlighted square edges at the bottom of Figure 1-14). We can see that the list of edges goes counterclockwise starting at the upper right. Note that here I've renamed the four Curve components to C0–C3 so it is easier to tell them apart.

Just as you can add outputs, you can also remove them—if you scroll the mouse wheel again to zoom in on List Item, you'll see a "–" sign appear to the left of +3. Clicking the "–" will remove +3, after which a "–" will appear next to +2, the output that is now the last in the list. Note that you can only remove the outputs you've created here, but not the original output i. We'll see later in the chapter that it is also possible to add/remove inputs to/from certain components, such as Maths ➤ Script ➤ Expression.

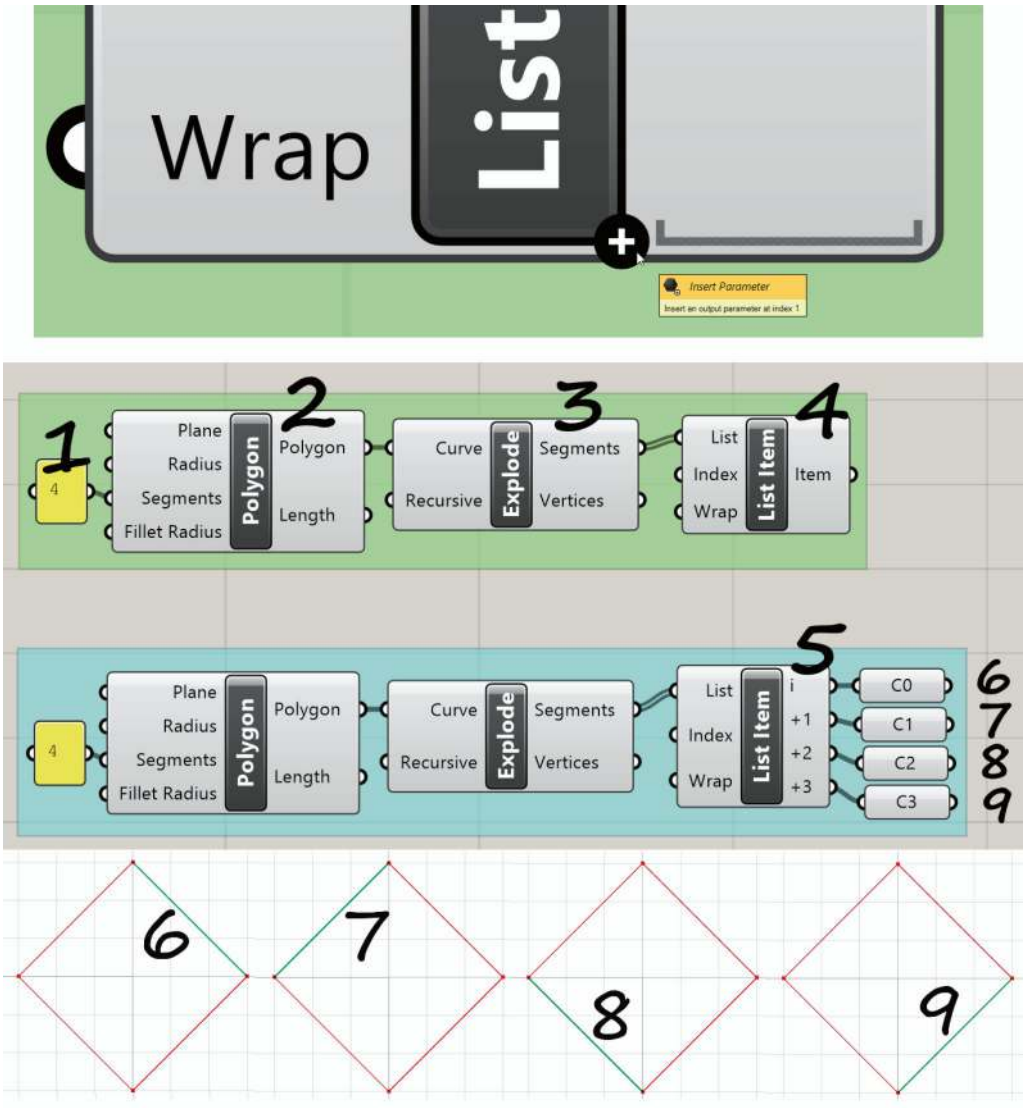


Figure 1-14. *Increasing the number of List Item outputs to retrieve several consecutive list entries at once. The top node graph shows using (4) List Item's default setup to grab the (6) first of the edges of a (2) square curve. The bottom node graph shows (5) modifying List Item to add three more outputs to obtain items at indices 0, 1, 2, and 3 (6, 7, 8, and 9), which correspond to the visual previews at the bottom tagged with those numbers*

Strategies for Creating Input Values

In this section, we'll look at some common strategies for creating input values for components.

Default Values

Many components have default values for some or all of their inputs. For example, the Curve ➤ Primitive ➤ Circle component we've seen in Figure 1-9 has the XY plane as the default value for its Plane input and 1 for its Radius input. As you start to work with GH on a regular basis, you'll get a feel for the convention of default values GH uses for each category of inputs, for example, any input requiring a plane has the XY plane as its default. When in doubt, simply hover over an input to see what its default value is.

Inheriting Data from Upstream Component(s)

Since data flows through wires from left to right, most components in the middle of a node graph get some of their inputs from the component(s) linked from their left.

Referencing Data from Rhino

Inputs for GH components can also be referenced from Rhino. Let's work through an example. We'll create some points in the Rhino viewport, like those shown in (1) of Figure 1-15, and then pick two of them to create a line in GH with the (2) Curve ➤ Primitive ➤ Line component. Line has two inputs, Start Point (A) and End Point (B). Right-clicking each input ➤ (3) Set one Point will minimize GH and take you to the Rhino viewport to select a point. Once a selection is made, the GH editor window will automatically restore itself. Once you've set both inputs, you'll see the (4) line preview appear between them. Since the points were created in Rhino, you can still (5) move them around in the viewport, and the GH line will automatically update.

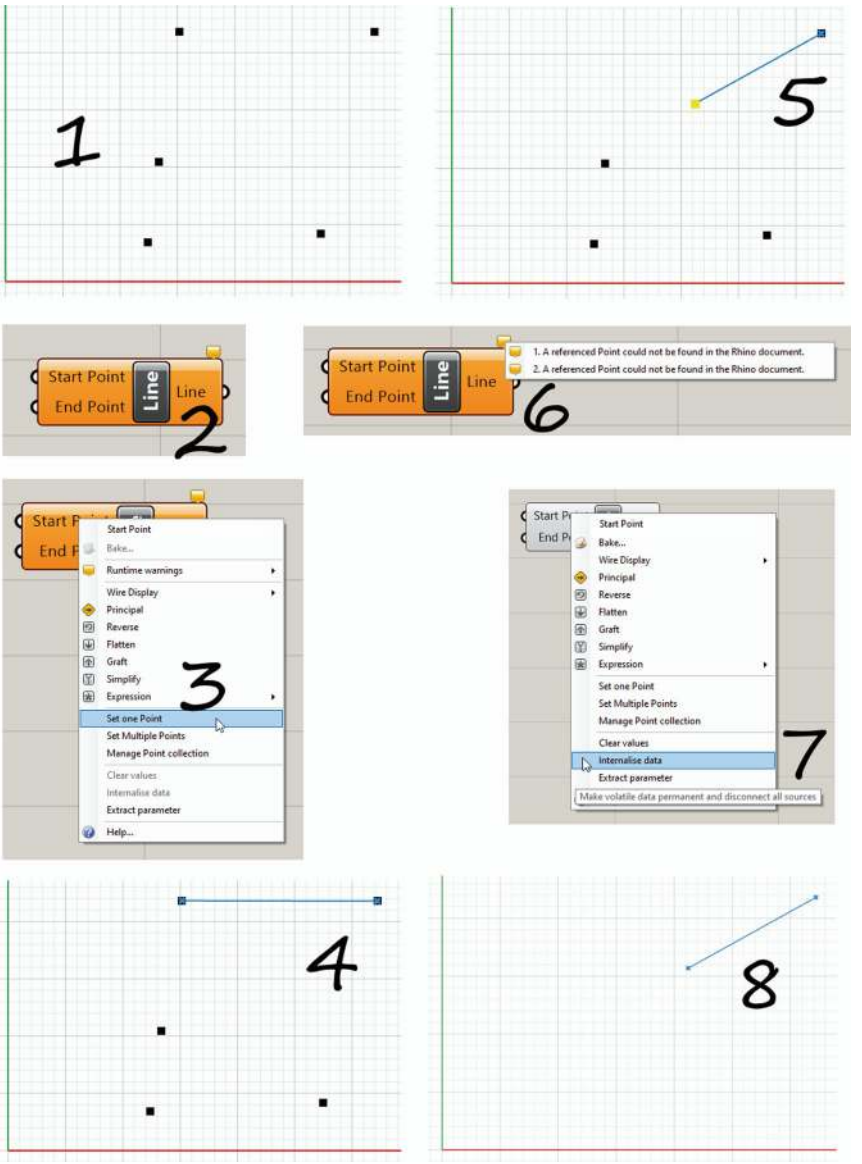


Figure 1-15. *Referencing data in Rhino from GH. We can use the (2) Line component in GH to create a line with (1) Rhino points, by right-clicking each of its inputs ➤ (3) Set one Point. Selecting the top two points will create the line in (4). We can subsequently (5) move one of the two points, and the GH line will auto-update. The same Rhino file must be opened every time the GH file is open, however; (6) otherwise, GH won't find the data. We can cut the tie to the Rhino file by right-clicking Line's inputs ➤ (7) Internalise data. The next time the GH file is opened on its own, only the points referenced by the GH line will show up*

One caveat with using data from Rhino is that the same Rhino file must be open alongside the GH file that references data from it; otherwise, GH won't be able to find the data, and you'll get warnings like those shown in (6) of Figure 1-15: "A referenced Point could not be found in the Rhino document." You could decouple the GH file from the Rhino file by right-clicking each input ➤ Internalise data (7 in Figure 1-15), which copies the Rhino data referenced to the GH file. In our example, once you've set the Internalise data option on both Start Point (A) and End Point (B) for the Line component, you can open the GH file on its own, which will produce the line preview in (8) of Figure 1-15. Notice that the rest of the points in the original Rhino file are gone; only the line's end points remain. You can no longer select and move the end points like you could before, unless you "extract" them from the Line component, as we'll see next.

Extracting an Input of a Component and Using It for Another

If a component's inputs are set via Set or Set Multiple, you can "extract" them into their own components to edit them. Continuing from our example above in Figure 1-15 of the Line component with two internalized inputs, you can right-click each input ➤ Extract parameter (1 in Figure 1-16) to trigger GH to create a (2) separate component for each internalized input that can be selected and modified on its own. If you ensure that Display ➤ Gumballs is toggled on (3 in Figure 1-16) and then select either the (2) Start Point or End Point component, you'll see a (4) widget appear at the selected point in the viewport, which can then be used to move the point by dragging one of the gumball's handles. In addition, you can reuse the extracted Start Point and End Point components by linking them to other parts of the node graph.

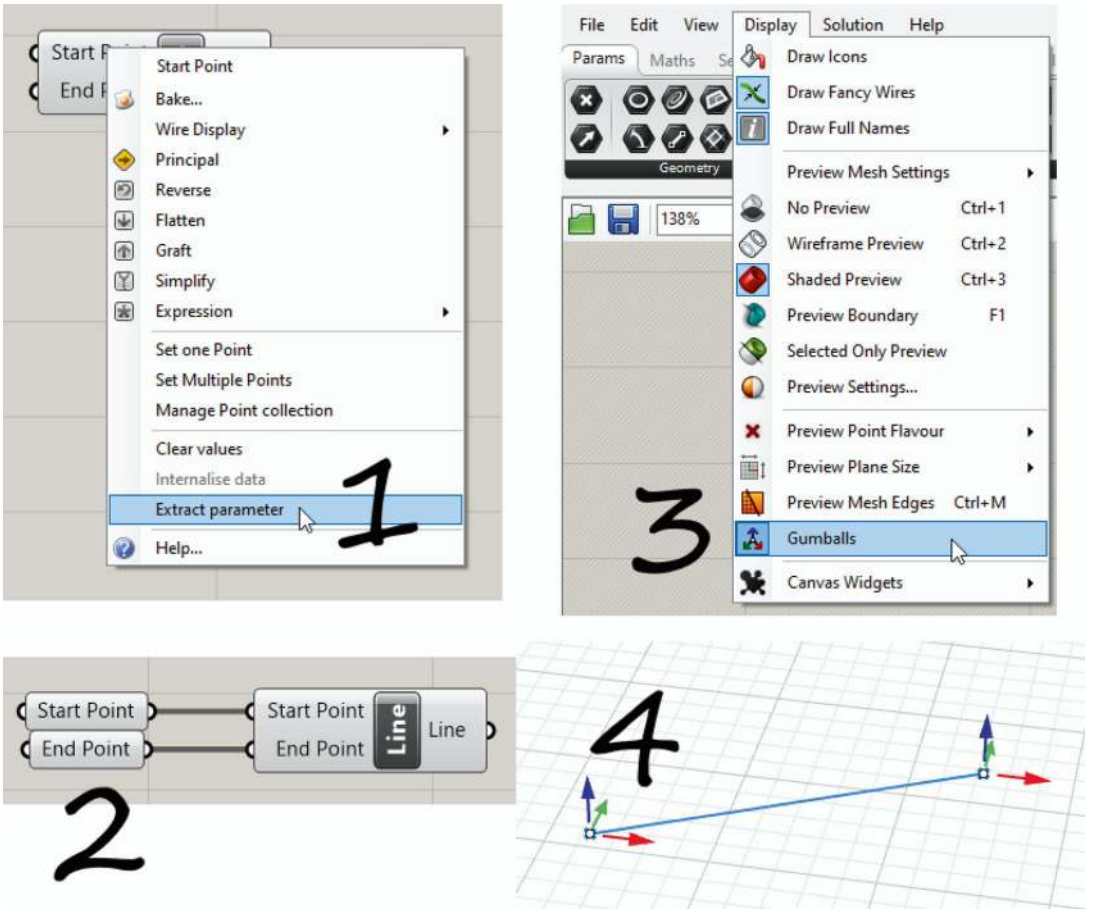


Figure 1-16. *Extracting an input value previously set via Set (or Set Multiple). (1) Right-clicking an input ➤ Extract parameter will trigger GH to create a (2) separate component for it. For the Line example, once a point input is extracted, if you ensure that (3) Display ➤ Gumballs is enabled, you'll be able to select the point in the Rhino viewport and drag the gumball handles to move it*

Setting One or Multiple Values Directly

When you right-click an input ➤ Set <Value> (or Set Multiple <Values>), instead of selecting data that already exist in Rhino, you can create new data. For example, with the Line component from Figure 1-15, when you right-click an input ➤ Set one Point, instead of selecting a point from the Rhino file, you can click somewhere where no point exists—GH will then create a point there and internalize it to that particular input.

You can tell whether a point is created in Rhino or GH by how it's drawn in the Rhino viewport—a Rhino point will be a circle or square, whereas a GH point will be an "X", as shown in Figure 1-17.

Values set directly on an input, whether created on the fly in GH as we've done here or selected from existing Rhino data, can be extracted just the same into their own components as we've seen in the section "Extracting an Input of a Component and Using It for Another." Non-geometric data such as numbers and Booleans can be set directly on an input as well, in which case you'd just type in the value(s).

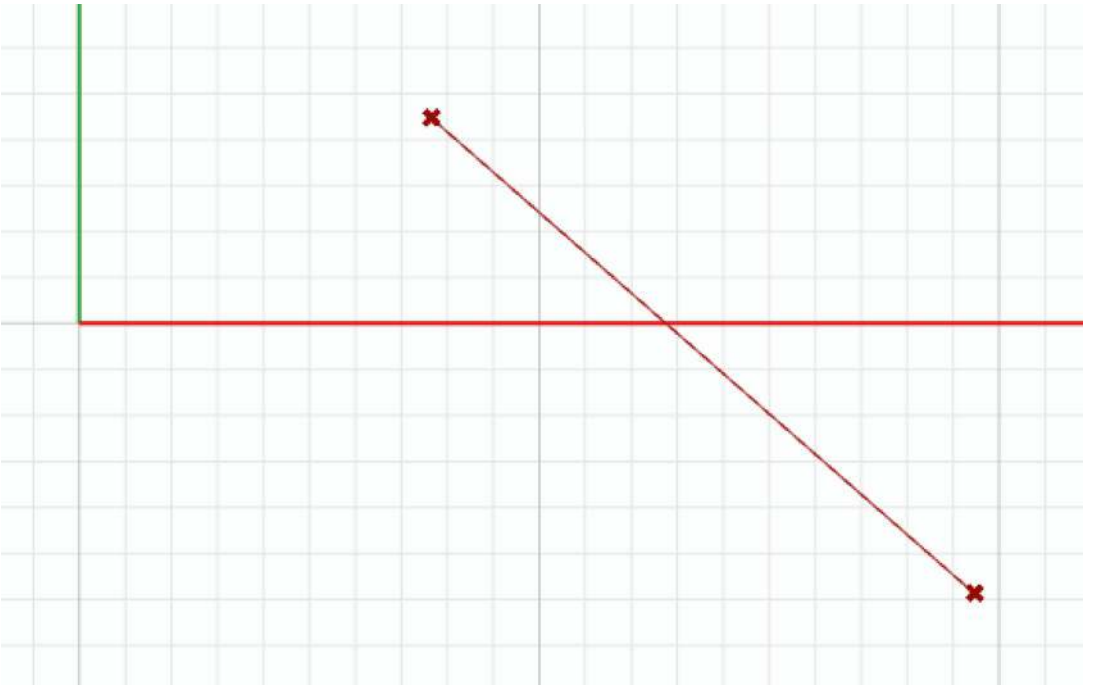


Figure 1-17. *Points created by GH are drawn as Xs in the Rhino viewport*

Creating One or More Numbers

Besides geometry, numbers are arguably the most common type of inputs in GH—you need numbers to specify dimensions, counts for repetitions, number of sides for polygons, how many times to divide a curve or surface, etc. In this section we'll go over some common ways to create one or more numbers.

Using Panels to Create Number(s)

Earlier in this chapter, we used Panels to inspect the outputs of components. It turns out Panels can be used to create inputs as well. To create a single numerical input, invoke the search box by double-clicking the canvas where you intend the Panel to be, and then type in a double quote (‘”’) followed by the desired number (1 in Figure 1-18). A Panel pre-filled with that number will then appear, which you can link to wherever a number is expected as input. (2) in Figure 1-18 shows an example of using a Panel to input 7.5 as the X and Y sizes for a Rectangle node. You can right-click and enter a more descriptive name for the Panel or delete the name altogether for a cleaner look (2, 3). To resize a Panel, click and drag any of its corners and edges.

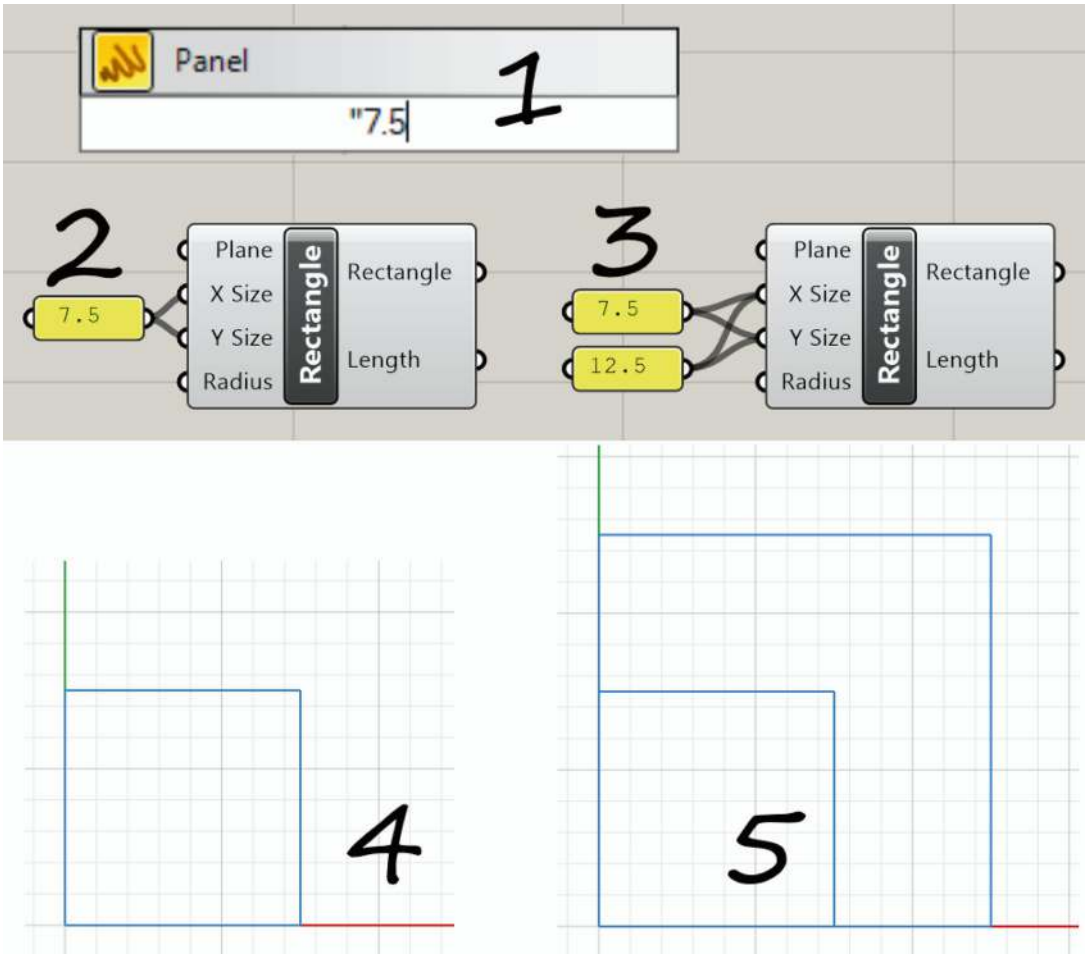


Figure 1-18. *(1) Enter ‘”’ (double quote) followed by a number in the search box to create a Panel pre-filled with that number*

In general, any input that expects a single number can also accept a list of numbers; the effect is that multiple outputs are produced, each corresponding to one value in the list, in the same order they are given. Figure 1-19 shows an example where we use one Panel with multiple lines to create a list of numbers. We create a Panel, double-click to open its (1) editor, and enter the list of numbers with (2) one number on each line without commas (as shown, 1, 2, 3, 4, and 5). Click OK, then right-click the Panel, and toggle off (3) Multiline Data. As shown, the Panel is linked to both the X Size and Y Size inputs of (2) Rectangle, hence creating (4) five squares with (X Size, Y Size) = (1, 1), (2, 2), (3, 3), (4, 4), and (5, 5).

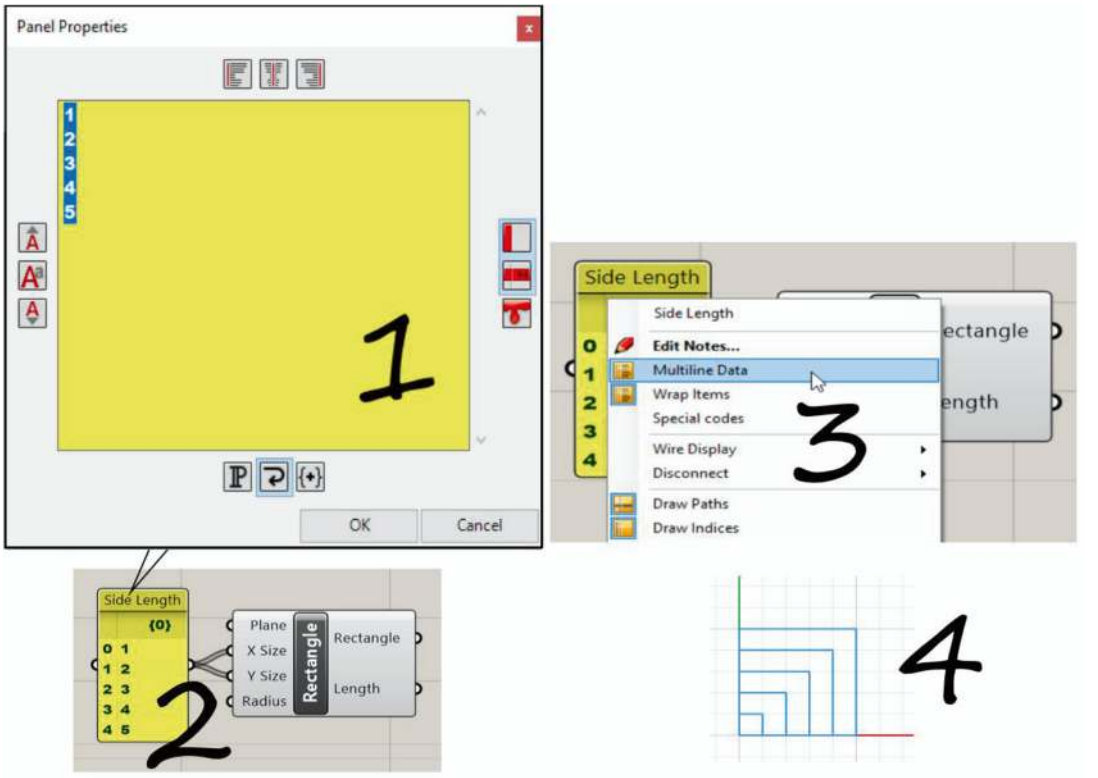


Figure 1-19. *Create a Panel, double-click to open its (1) editor, and enter (2) one number per line without any commas, followed by (3) right-clicking ➤ Multiline Data (toggle off). Connecting the Panel to both the X Size and Y Size inputs of Rectangle will produce (4) five squares*

Using Params Containers to Create Number(s)

Another way to create one or more numbers is using either Params ➤ Primitive ➤ Integer for whole numbers or Params ➤ Primitive ➤ Number for decimals and then right-clicking ➤ Set Integer (Number) or Set Multiple Integers (Numbers). When setting multiples, you'd enter one number per line without commas, just like you would with Panels.

Setting Numbers Directly on an Input via Set/Set Multiple

Alternatively, you could set the number(s) directly by right-clicking a component's input ➤ Set <Value> or Set Multiple <Values>, although I don't recommend setting a number this way since you cannot tell what value is set on an input unless you hover over it with the mouse.

Creating Domains with Panels or Construct Domain

You may have noticed when you hover over either the X Size or Y Size input of Rectangle, it says " … to…"—this is because Rectangle actually takes a domain instead of a single number for either of these inputs, for example, an X Size of 2.5 to 11.5 means Rectangle's side in the X direction spans from X = 2.5 to X = 11.5. When you connect a single number N to X Size, GH implicitly converts it to "0 to N" for you in a process called "data conversion," which we'll discuss more shortly. Using a domain to specify X Size and Y Size for Rectangle gives you more control, as you don't have to align the rectangle's lower-left corner to (0, 0, 0).

There are two ways to quickly create a domain. The first is to create a Panel with a string in the format "<start> to <end>", where <start> and <end> are the minimum and maximum of the domain, for example, "2.5 to 11.5". We can again use the search box shortcut, by double-clicking where you intend the Panel to be and then typing in a double quote followed by the domain string, for example, "2.5 to 11.5, which is shown as (1) in Figure 1-20.

The second way to create a domain is to use Maths ➤ Domain ➤ Construct Domain (2 in Figure 1-20). You'd specify a number for Domain start and a second (larger) number for Domain end. In Figure 1-20 we use an X domain starting at 2.5 and Y domain at 4.5, to move the rectangle away from the origin (3).

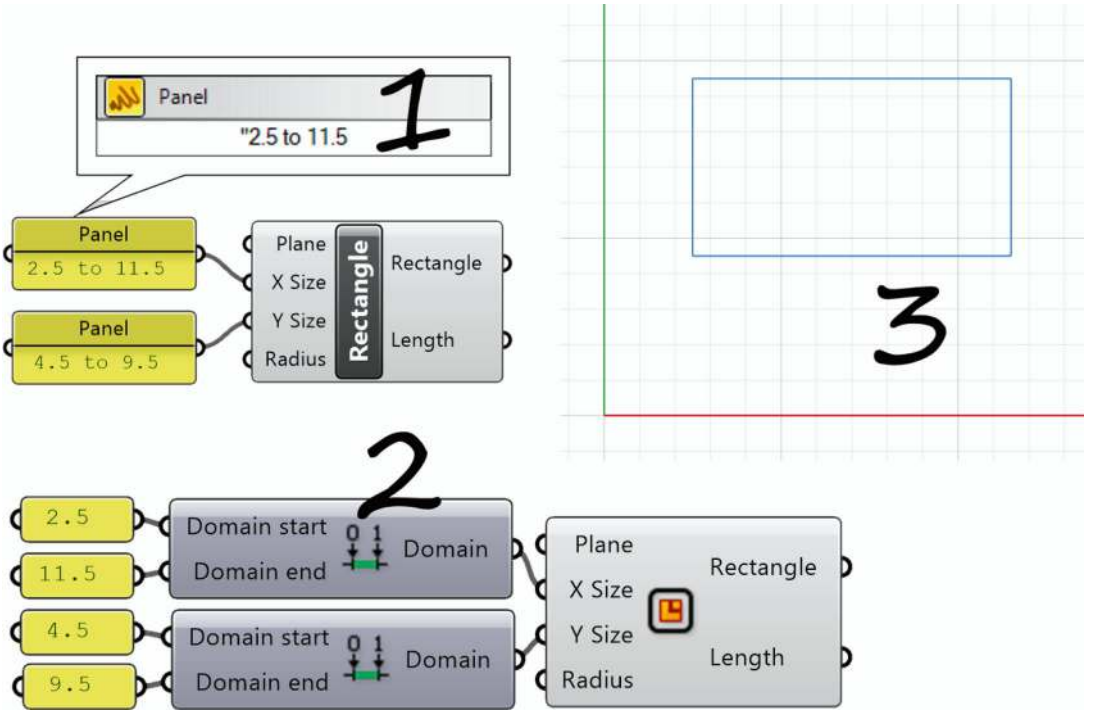


Figure 1-20. *You can create a 2D domain by entering a string in the format (1) "<start> to <end>" in a Panel or using the (2) Construct Domain component. As shown, both produce the same (3) Rectangle with X Size from 2.5 to 11.5 and Y Size from 4.5 to 9.5*

Number Slider

If you ask me what the most common input component is in GH, I'd say the Number Slider—you need numbers for everything! To adjust the length of curves, the magnitude of vectors, the number of divisions on a surface, and so on. The Number Slider gives you an interactive widget with custom precision and range. In this section, I'll show you my favorite methods for creating Number Sliders and configuring their options.

You can drag a Number Slider down to the canvas from Params ➤ Input; however, in practice I never do. Instead, I use the following four search box shortcuts to create Number Sliders while specifying their options at the same time.

1. Integer Slider with Default Range

When you type an integer into the search box, GH will see it as a shorthand for creating an integer slider dialed to that value and automatically derive a default range from it. For example, if you type in a 2, GH will create a Number Slider set to 2 with the range 0 to 10 (1 in Figure 1-21). When you click and drag the slider, the value will go up or down by 1s. When a Number Slider is created this way, the default range goes between 0 and powers of 10—0 to 10, 0 to 100, and so on.

2. Decimal Slider with Default Range

Typing a floating-point number in the search box will tell GH how many decimal places you want for the slider. (2) in Figure 1-21 shows an example where 0.5 is typed into the search box to create a slider set to 0.5 with the range 0.0 to 1.0, which increases and decreases by multiples of 0.1 when dragged. When a Number Slider is created this way, the default range goes between 0 and powers of 10 with the specified number of decimal places, for example, if two decimal places are given, 0.00 to 10.00, 0.00 to 100.00, and so on.

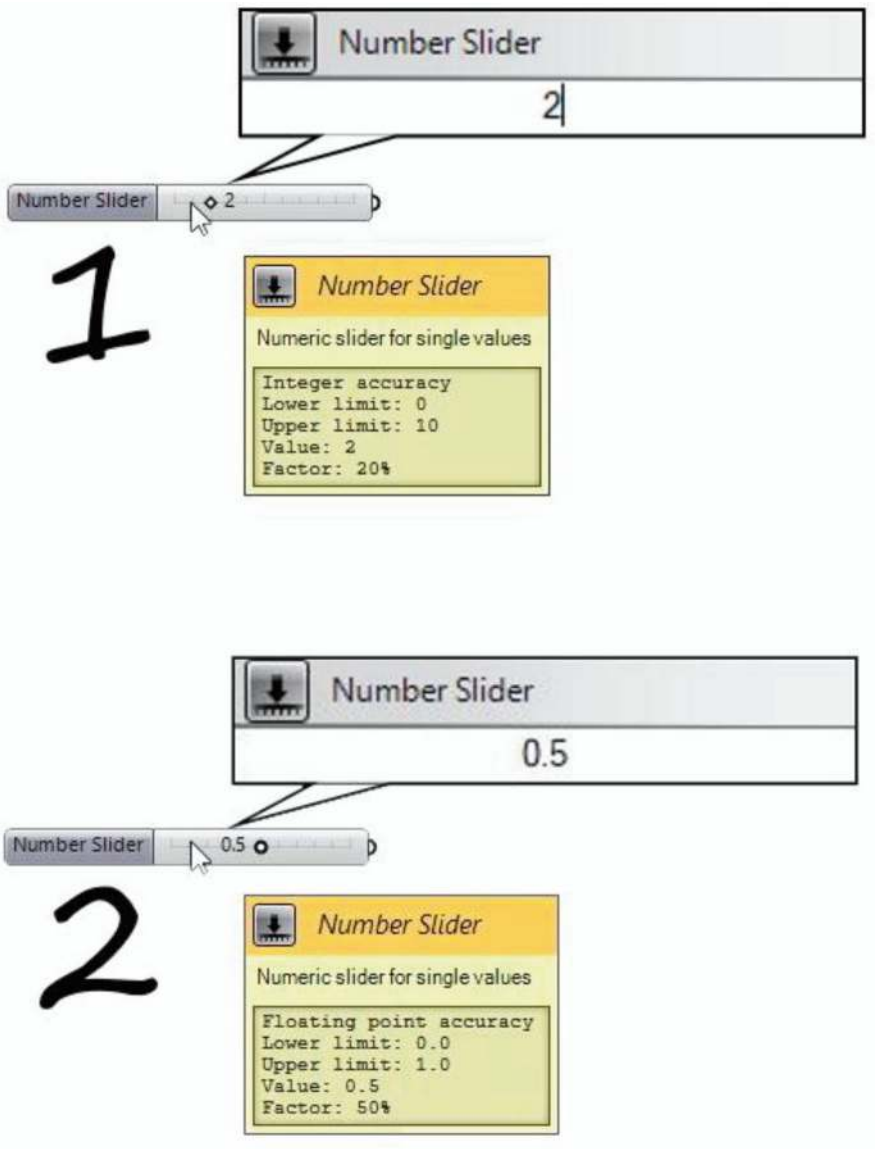


Figure 1-21. *Search box shortcuts for creating Number Sliders with default ranges. (1) Typing in a 2 will create an integer slider set to 2 with the range 0 to 10. (2) Typing in a 0.5 will create a decimal slider set to 0.5 with the range 0.0 to 1.0*

3. Integer Slider with Custom Range

You can specify a custom range along with the slider value in the same string typed into the search box, using the format

$$min < slider value < \max$$

(1) in Figure 1-22 shows an example where 1<15<50 is typed into the search box to create a whole number slider with the range 1 to 50 and a current value of 15. The slider goes up and down by increments of 1.

4. Decimal Slider with Custom Range

Finally, you can combine a decimal number with a range in the search box, which will customize both the precision and the min/max of the slider in one go. (2) in Figure 1-22 shows such an example, where 2.25<3.75<12.00 is typed into the search box to create a slider with two decimal places, a minimum of 2.25, and a maximum of 12.00, which is set to 3.75. The slider changes by increments of 0.01 when dragged.

If you give only two numbers instead of three, in the format

$$min < \max$$

the slider created will be set to the min value and have the range of min to max.

Property Editor for the Number Slider

With several strategies for creating Number Sliders under your belt, let's look at how to fine-tune a Number Slider's properties. If you double-click the name of the slider (which by default is "Number Slider"), the property editor opens as shown in Figure 1-23. You can type in a (1) new name for the slider to make it more descriptive and (2) change the slider number type between decimal (R), integer (N), even integers (E), or odd integers (O). For decimals (3) the number of places after the point can be set to 0–6. You can also adjust the (4) min, (5) max, and (6) range of the slider and of course the (7) current value it's set to.

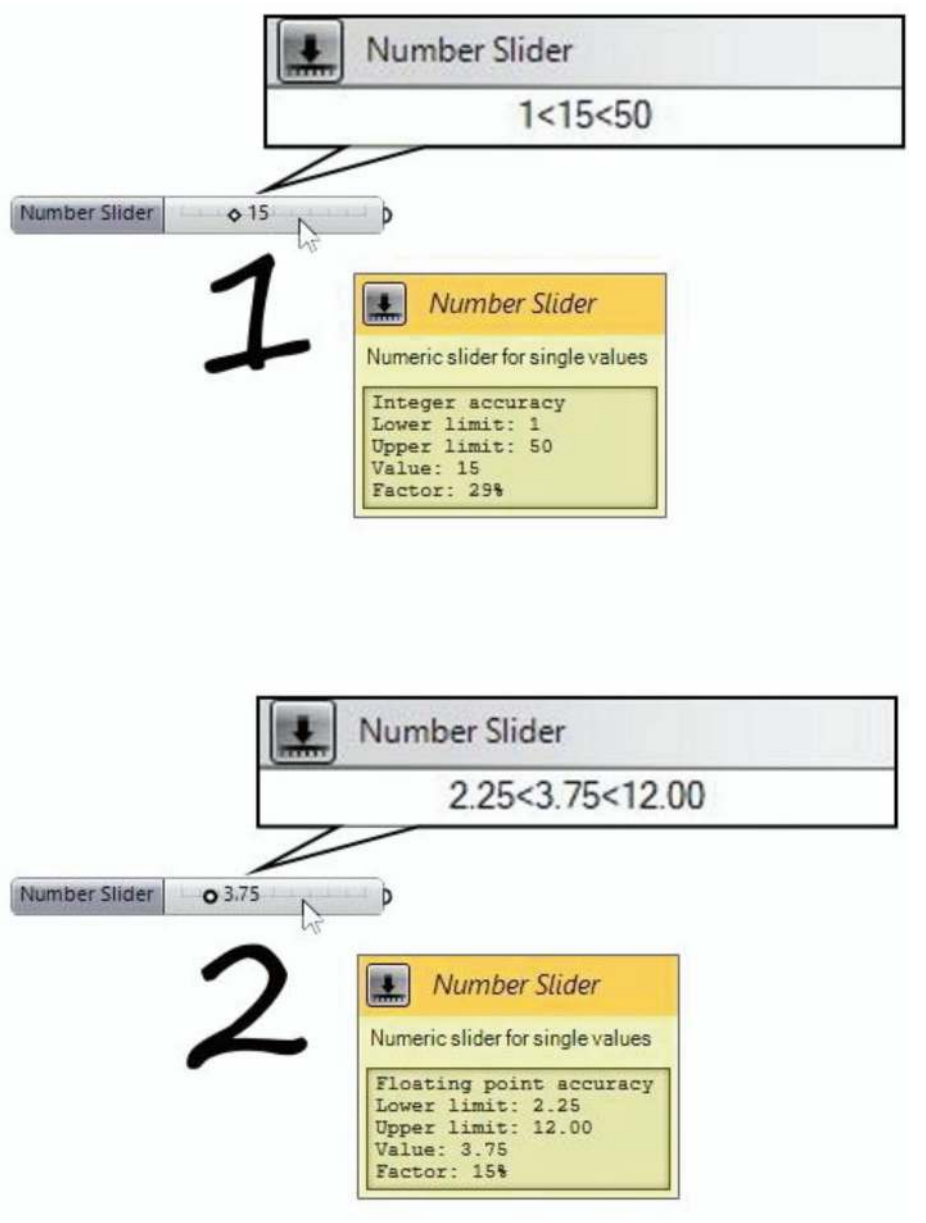


Figure 1-22. *Search box shortcuts for creating Number Sliders with custom ranges. (1) Typing in 1<15<50 will create an integer slider set to 15 with the range 1 to 50. (2) Typing in 2.25<3.75<12.00 will create a decimal slider set to 3.75 with the range 2.25 to 12.00 and two decimal places*

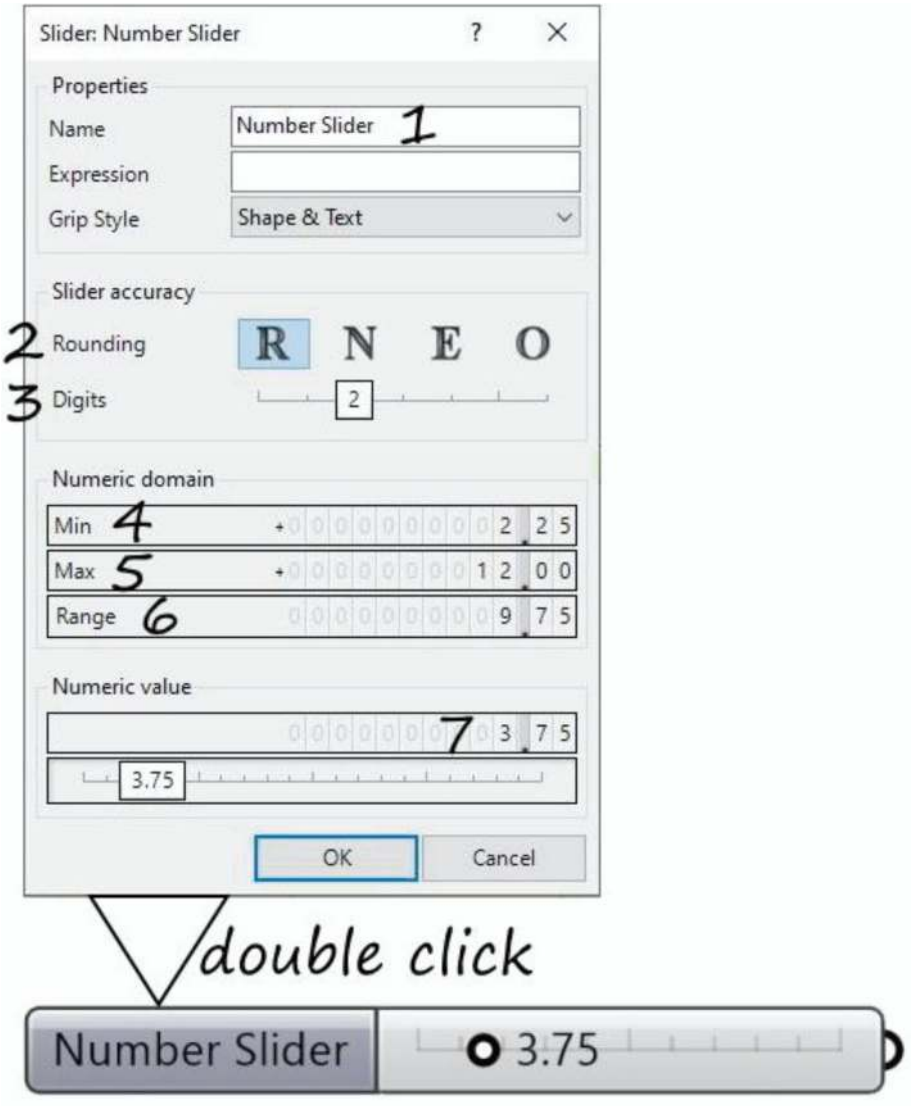


Figure 1-23. *Double-click a Number Slider's name to open its property editor, where you can edit its (1) name, (2) rounding scheme, (3) number of decimal places, (4) min, (5) max, (6) range, and (7) current value*

Creating a Boolean Value (True or False)

Boolean values (True or False) are useful for toggling options on and off, for example, the Wrap input for Sets ➤ List ➤ Shift List, which specifies whether to wrap a list around whose items have been scooted up or down. Figure 1-24 shows a few ways you could create a single Boolean value: (1) is a Boolean Toggle whose value can be toggled via double-clicking. (2) is a Panel with the word True or False, which is automatically converted to the corresponding Boolean. (3)–(5) are negative, zero, and positive numbers connected to Params ➤ Input ➤ Boolean components and converted to True, False, and True, respectively. In general, zero is converted to False, while all other numbers True. (6) is a Params ➤ Input ➤ Button, which is False unless you hold down LMB over the black area.

In addition to a single True or False for toggling options, Boolean values are also used for specifying patterns for weaving, splitting, and culling streams of data (Sets ➤ List ➤ Weave, Sets ➤ List ➤ Dispatch, and Sets ➤ Sequence ➤ Cull Pattern), which we'll go over in Chapter 2. To create a pattern, you'd fill a Panel with one word (number) per line and right-click to toggle off Multiline Data, with the word False (or number 0) for false and the word True (or number 1) for true.

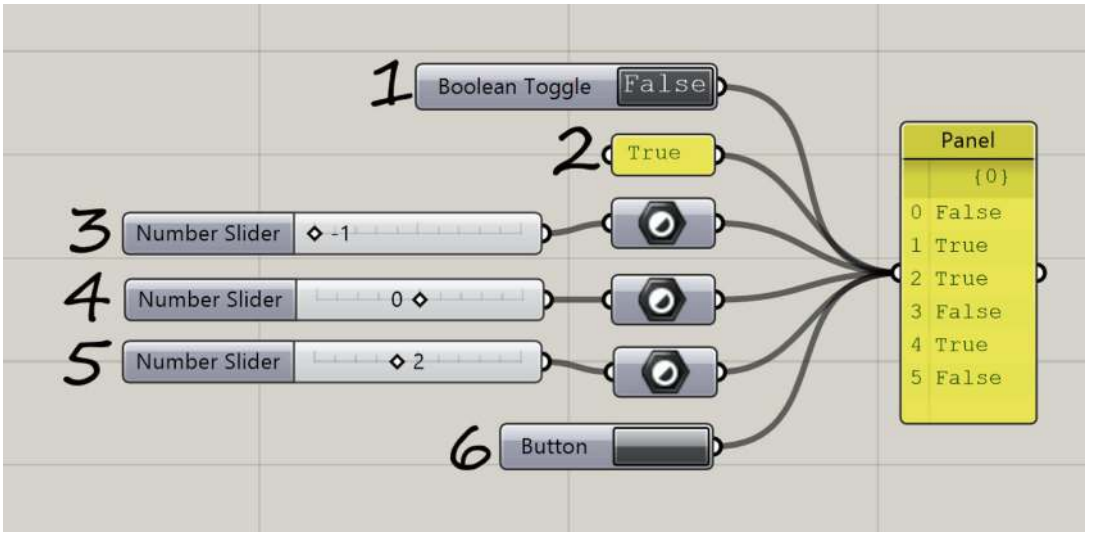


Figure 1-24. *A few ways to create single Boolean values. (1) Boolean Toggle. (2) Panel with the word True. (3)–(5) are Number Sliders with negative, zero, and positive values linked to Boolean components, which are converted to True, False, and True. (6) is a Button, which is False unless you hold down LMB over the black area*

Creating a Point

Points are the building blocks for many types of geometry. You can interpolate points into curves, network points into surfaces, triangulate points into meshes, etc. We'll be creating many points of our own throughout the book to kick off complex projects.

A quick way to create a point is by typing a string of the format "{X,Y,Z} into the search box (note the double quote at the beginning) to create a Panel, where X, Y, and Z are the coordinates for the point you intend to create. (1) in Figure 1-25 shows an example where the string "{1,2,3} is typed into the search box to create a Panel representing the point with coordinates (1,2,3), which can then be linked to any input expecting a point, such as the Start input of Curve ➤ Primitive ➤ Line SDL (2 in Figure 1-25), which creates a line segment originating at that point with a default direction of positive Z and length of 1.

(3) in Figure 1-25 shows another way to create a point, which is by using a Vector ➤ Point ➤ Construct Point component. As shown, three Panels are used to create the numbers for the X, Y, and Z coordinate inputs.

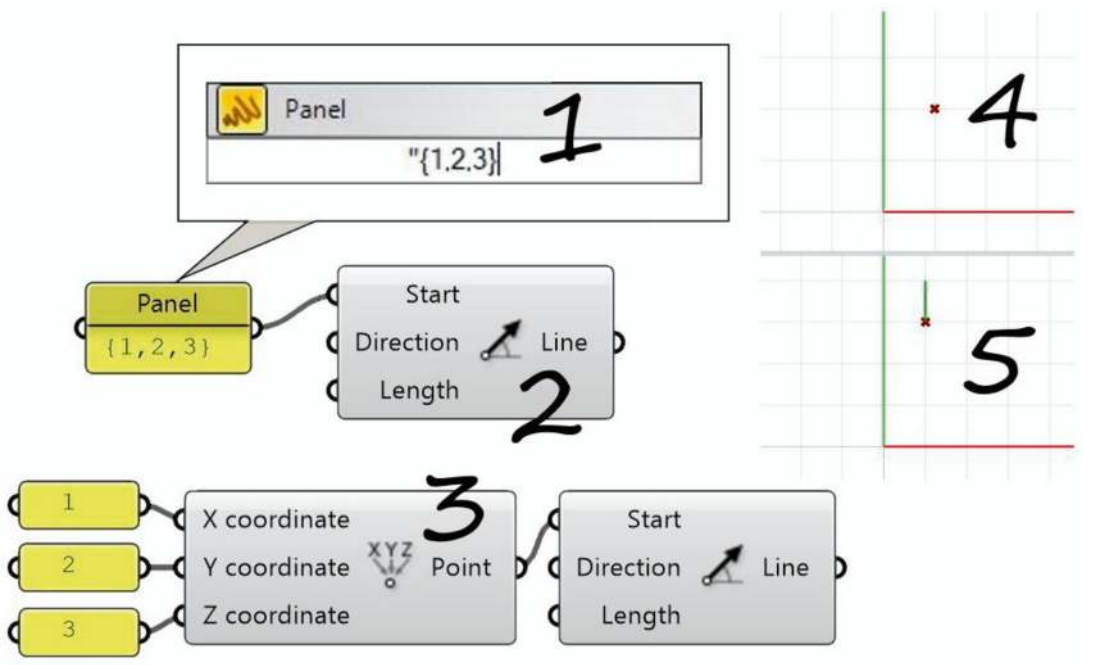


Figure 1-25. *Two ways to create a point: (1) You can make a point with coordinates (1,2,3) via a Panel by typing "{1,2,3} into the search box, which can then be linked to any input that expects a point, such as the Start input for (2) Line SDL. You can also make a point with the (3) Construct Point component. (4) and (5) show the top and front views of the line segment produced by (2)*

Types of Wires

Wires appear differently depending on the structure of content flowing through them. There are three types of wires: a ***single solid line*** carries a single item, a ***double line*** carries a flat list, and a ***double dashed line*** transmits a data tree (which we'll look at in Chapter 2). Figure 1-26 shows examples for all three wire types: The Number Slider sends 3 (a single item) to the two Series components via solid wires. Series takes a default Start = 0 and Step = 1, plus the given Count (3 in this case) to generate *a list of numbers* (0, 1, and 2), which is in turn linked to the X and Y inputs of Construct Point via double lines.

Based on the combination of inputs, Construct Point produces a list of points [(0,0,0), (1,1,0), (2,2,0)], which forms a diagonal line. To create a grid of 3 by 3 points instead, we can "Graft" X (double-click X ➤ Graft) to force GH to pair up each value of X with *every* value of Y, thus cross-referencing the two and creating a nested list of points, which we can inspect via a Panel. Notice that the wire between Construct Point and Panel is a double dashed line. We can see that the first list of points is [(0,0,0), (0,1,0), (0,2,0)], the second list is [(1,0,0), (1,1,0), (1,2,0)], and the third list is [(2,0,0), (2,1,0), (2,2,0)]. This nested list structure is called a *data tree* in Grasshopper, which we'll go over in detail in Chapter 2.

To change wire display across the definition, you can toggle the option Display (top menu bar) ➤ Draw Fancy Wires, which when "on" draws the three wire types as described above (solid, double, double dashed) and when "off" draws all wires as solid lines. Draw Fancy Wires is "on" by default, which is an option I highly recommend that you keep, since the different wire types will help you confirm the structure of data flowing in and out of each component is as you expect.

To change the wire display on a per-component basis, you can right-click each *input* of a component ➤ Wire Display, which gives you three options: Default, Faint, or Hidden. "Default" will show wires based on the definition-wise setting, "Faint" will draw a faded version, whereas "Hidden" will hide the wires from view unless you select that component. When "Hidden" is selected for an input, there will be a "wireless" icon displayed at that input; however, the output(s) from which its connected wires originate will not show any indication—in other words, you can't tell where the input in question is connected to until you select the component, which could make your definition confusing, since some components could appear to be dead ends, but in fact connected to some other components with hidden wires. It is for this reason that I caution the use of hidden wires. When used sparingly, however, hidden wires could help eliminate visual clutter in a complex script, for example, when wires cross over one another.

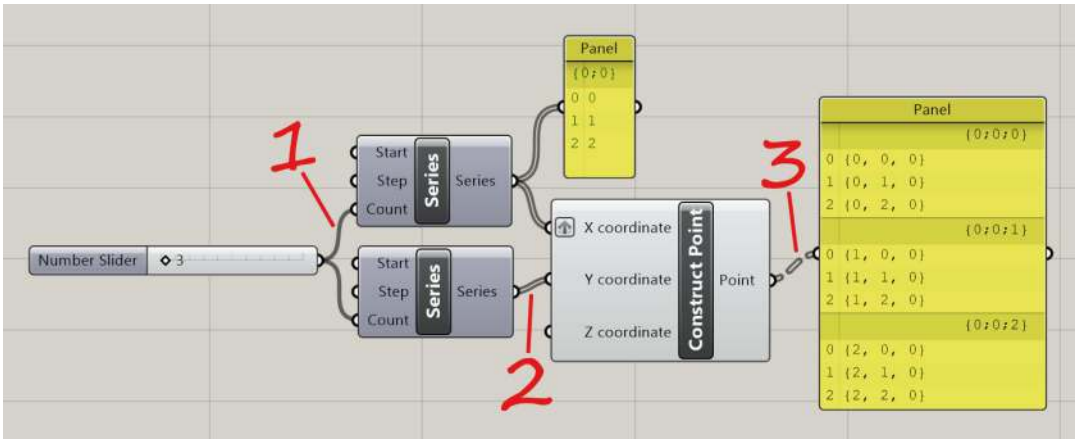


Figure 1-26. *The three types of wire in Grasshopper. (1) Solid single line, which carries a single item. (2) Double line, which carries a flat list of items. (3) Double dashed line, which carries nested lists (data trees)*

Basic Transforms

In this section, we'll go over how to perform basic transforms on geometry and use combinations of them to quickly build up interesting silhouettes.

Move

In Figure 1-27, we'll start with a (1) Polygon node and use Number Sliders to control its Radius (R) and number of Segments (S)—as shown, we use R = 10 and S = 6 to make a hexagon. We create a second hexagon at an offset of Z = 20 by linking (1) Polygon to the Geometry (G) input of a (3) Move node, with a (2) Unit Z as its Motion (T) with a Factor of 20.0. The original and moved hexagons (1 and 3) together form the stack of curves shown in (5). We then Shift-LMB drag both (1) and (3) to the Curves (C) input of (4) Loft to create a simple surface, which is shown as (6).

You can find Polygon under Curve ➤ Primitive, Move under Transform ➤ Euclidean, Unit Z under Vector ➤ Vector, and Loft under Surface ➤ Freeform.

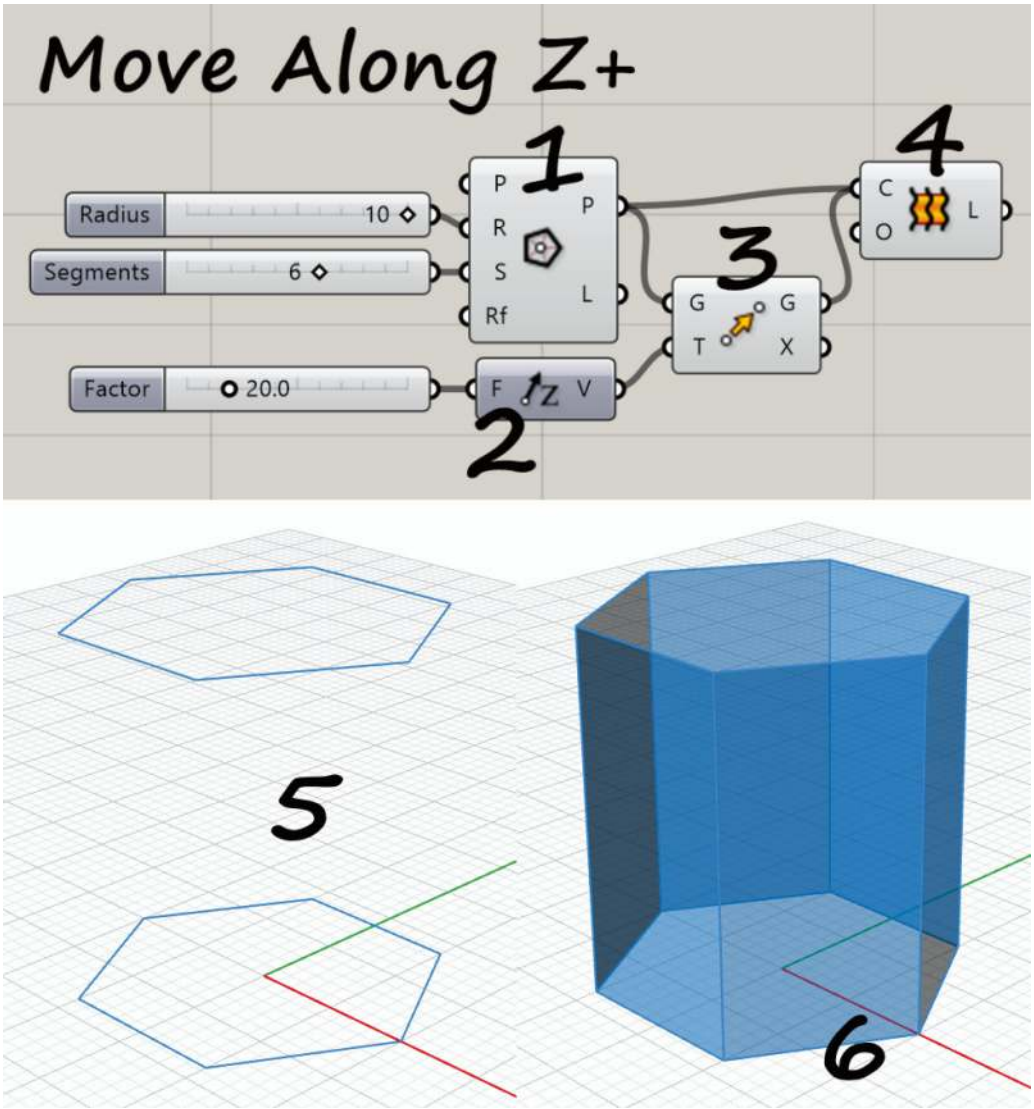


Figure 1-27. *Using a (3) Move node with a (2) Unit Z vector scaled to 20.0X to offset a hexagonal curve created with a (1) Polygon node. Then (4) Loft the (5) two hex curves to form a (6) simple surface*

Rotate

In Figure 1-28, we modify the node graph from Figure 1-27, by inserting a (4) Transform ➤ Euclidean ➤ Rotate node between (3) Move and (5) Loft, thus rotating the second hexagonal curve before lofting the two hexagons. By default, (4) Rotate has an Angle (A) value of 0.5*pi = 90 degrees, therefore giving us the hexagon stack shown as (6) and, when lofted, producing the twisted column shown in (7).

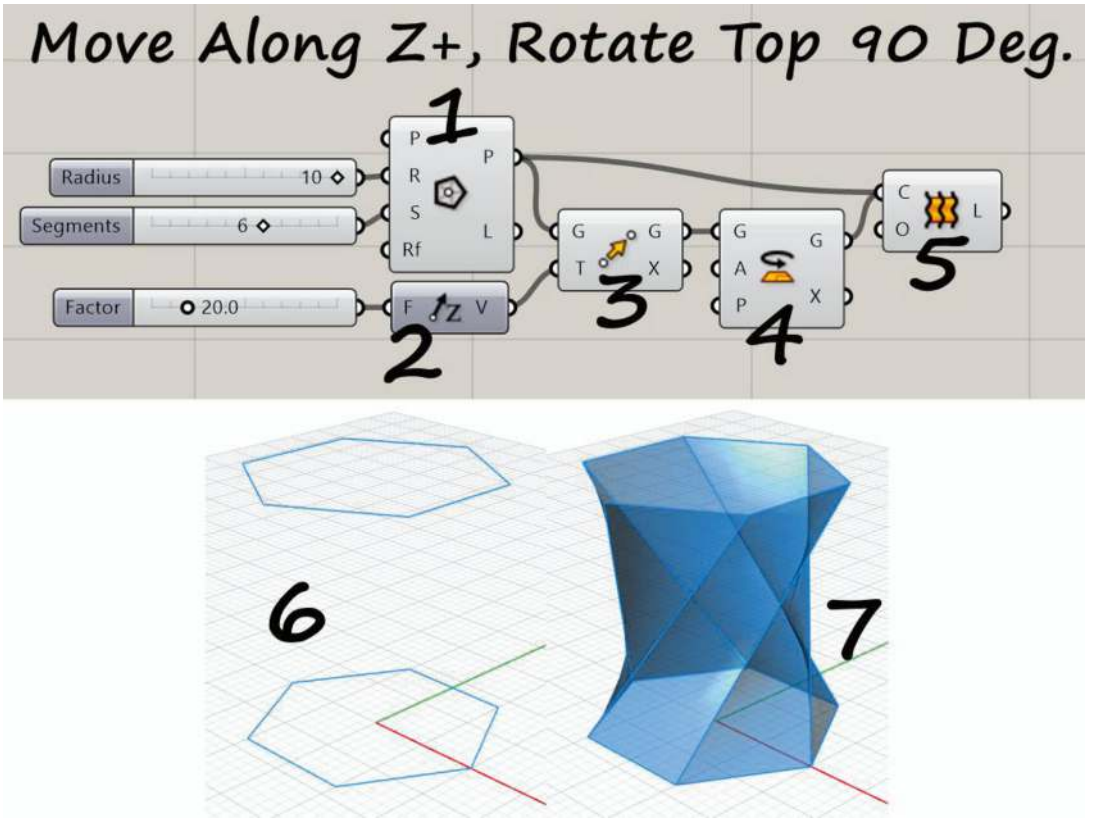


Figure 1-28. *We can modify the node graph from Figure 1-27 by inserting a (4) Rotate node between (3) Move and (5) Loft, which by default rotates the top hexagonal curve by 90 degrees, thus producing the twisted column in (7) when the two hexagons are lofted*

Scale

We can refine the twisted column from Figure 1-28 in Figure 1-29, by adding a (5) Transform ➤ Affine ➤ Scale NU node between (4) Rotate and (6) Loft. We use a Number Slider to dial the X and Y inputs of Scale NU to 0.7 and leave Z = 1 at default, which has the effect of scaling the top hexagon horizontally to 70%, as shown in 7a (top view) and 7b. When lofted, the original and the offset, rotated, and scaled hexagon produce the surface shown in 8a (top view) and 8b.

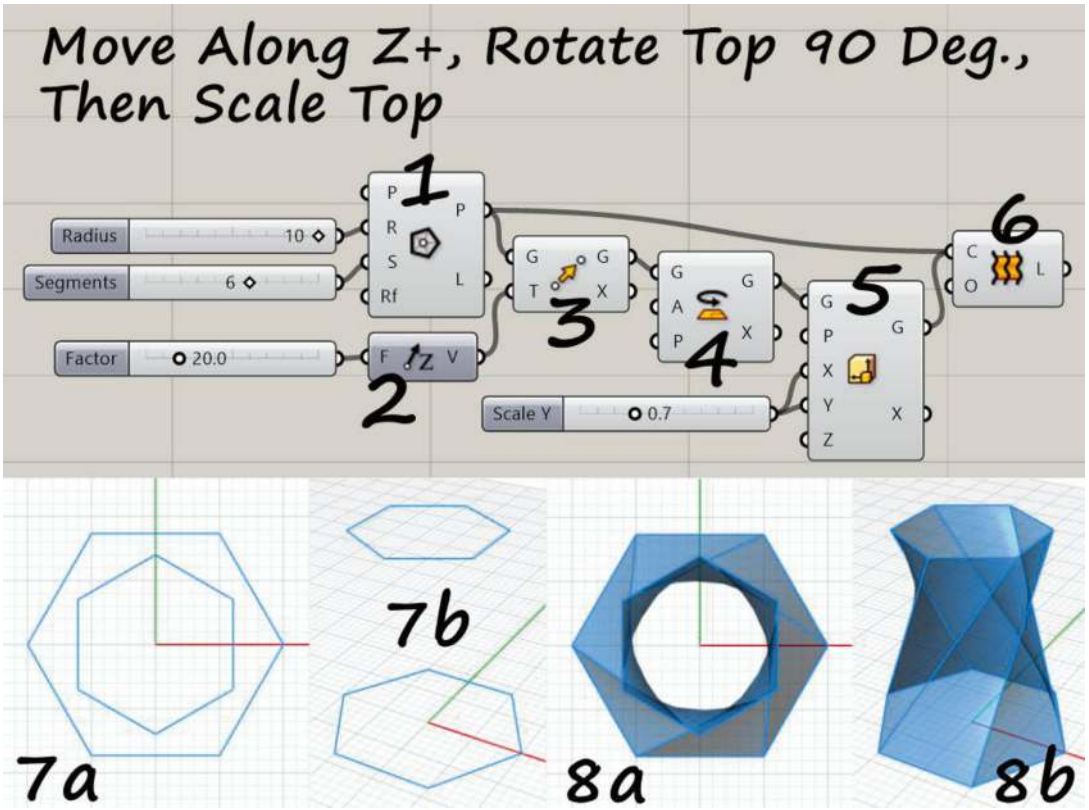


Figure 1-29. *We can modify the node graph from Figure 1-28 by inserting a (5) Scale NU node with X = Y = 0.7 between (4) Rotate and (6) Loft, which scales the top hexagonal curve horizontally to 70% (7a and 7b); thus, when lofted, the original and the offset, rotated, and scaled hexagon together form the twisted column in (8a, top view) and (8b)*

Creating Variations

You can create many interesting variations of the twisted column from Figure 1-29, by changing the number of Segments (S) of (1) Polygon, the Factor (F) to (2) Unit Z (which changes the height of the column), the Angle (A) of (4) Rotate (which affects the amount of twist), and the X and Y inputs of (5) Scale NU (which vary the relative proportions of the column's top and bottom).

Summary

In this chapter, you learned the basics of building parametric models in Grasshopper (GH). Starting with the overall workflow for creating a script ("definition"), which is to place "components" (code blocks) on the "canvas" (editor backdrop) and connect them with "wires," you progressed to learning about components—their appearances and structure, how to use color to tell their current statuses, and how to customize their display (text vs. icons). After that we discussed how to configure a component's outputs, preview output geometry in the Rhino viewport, and inspect textual versions of outputs via Panels.

In the next part of the chapter, we delved deeper into strategies for creating input values for components, which included using default values, inheriting data from nodes upstream, setting values directly, and referencing data from Rhino. We went over various methods for creating one or more numbers, including using Number Sliders, Params containers, and Domains created using Panels and search box shortcuts. We also discussed the creation of other commonly used types such as Booleans (True or False) and points.

After immersing ourselves with components, we got up to speed with wires—the connections between components, how data flows through them, and how their various appearances indicate the structure of data they carry.

Putting it all together, in the final part of the chapter you learned how to create curves and edit them with basic transforms like move, rotate, and scale and loft them into a surface resembling a twisted column. You also contemplated how to change the parameters to the curves themselves as well as their transforms to create variations.

CHAPTER 2

Grasshopper and Rhino Python Quick Start: Part 2—Advanced Topics

In Chapter 1, you learned the basics of working with Grasshopper components and various strategies for creating common input values like numbers and Booleans and geometry like curves and surfaces. You also practiced referencing data from Rhino to edit in GH and used combinations of transforms like movement, rotation, and scaling to create parametric models.

In this chapter, we'll delve into more advanced scripting topics, starting with *data conversion*, the procedure with which Grasshopper converts data when compatible but non-identical types are connected, such as when a point is linked to an input expecting a plane. The second part of the chapter will cover how to work with lists (linear sequences of data) through building graphs with Range, Series, Cull Pattern, Partition List, Sub List, Dispatch, and Weave nodes. In the third part of the chapter, we'll get into the nitty gritty of *data matching*—the process with which Grasshopper deals with inputs of mismatched sizes, using strategies like Longest List, Shortest List, and Cross Reference. After that, we'll discuss *data trees*, which are nested lists in GH—how they work, how to access and edit tree branches, and how to utilize the tree structure to get components to behave the way we want.

So far, we've focused on building GH definitions with predefined nodes. In the fifth part of the chapter, we'll discuss how to edit a Python 3 Script node to generate parametric seashell spiral curves and save it to disk as a *User* Object, which behaves like a custom component installed to the local copy of Grasshopper. After that, we'll discuss methods for tidying up GH definitions, including aligning, distributing, and grouping

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_2

components, as well as creating comments with Scribble and Panel nodes. Lastly, we'll go over how to turn GH preview into geometry that can be edited in Rhino using a process called *baking*.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper. Download the files and navigate to the /Ch2 folder, where you'll find *.gh files named after the figures in which they appear along with a short description, for example, Figure2-1_data_conversion.gh. The sample files have been tested in Rhino 9 on Windows. As you read the chapter, you can build your own node graphs by following the steps described in each section or start with the downloaded *.gh files and experiment by modifying their parameters.

Data Conversion

Grasshopper provides a convenience feature such that if the data flowing from the output of one component is compatible but not identical to the type expected by the input of the next component, the data is automatically converted. We've seen a few examples of this already, where Panels containing appropriately formatted strings are automatically converted to numbers, domains, and points. In this section, we'll look at a few more examples of *data conversion*.

Numbers to Booleans and Rounding Numbers

As we've seen in the section "Creating a Boolean Value (True or False)" in Chapter 1, when a nonzero number is connected to an input expecting a Boolean, it is converted to True (1 in Figure 2-1). Another example is when you connect a floating-point number to an input expecting an integer, it is automatically rounded to the nearest whole number (2 in Figure 2-1).

Implicit Conversion to Planes

Auto-conversions between geometric types can be more subtle. In GH convention, any curve or surface component with a yellowish icon expects an input plane with which to position the output geometry, for example, Curve ➤ Primitive ➤ Rectangle (Figure 2-1), and Surface ➤ Primitive ➤ Cone (Figure 2-1). To construct the plane input, you can use one of the components under Vector ➤ Plane, such as XY Plane or Construct Plane. However, data conversion gives us a few shortcuts, which we'll look at next.

Point to Plane

The first shortcut is that connecting a point to a plane input will automatically create a plane parallel to the XY plane centered at that point. For example, (3) in Figure 2-1 creates a point with the string {1,2,3} in a Panel, which when linked to the Plane input of Rectangle (4) is automatically converted to the plane parallel to the XY plane centered at (1,2,3), with unit Z as its normal vector.

Planar Geometry to Plane

The second shortcut data conversion provides us for constructing planes is that when a piece of planar geometry is linked to another component expecting a plane, the plane the geometry sits on is extracted and used as the input value. Continuing from our previous example, when the Rectangle (4 in Figure 2-1) is linked to the Base input of Surface ➤ Primitive ➤ Cone (7), the plane the Rectangle sits on is extracted and used as the input.

Conversion of Planar and Closed Curves to Breps and Surfaces

Connecting a planar, closed curve to an input expecting a Brep or surface will automatically create a Brep or surface with that curve as its boundary. (5) in Figure 2-1 shows an example where a Rectangle is linked to the Surface input of Surface ➤ Analysis ➤ Surface Points, in which case, a surface with the rectangular curve as its boundary is created and used as input. (6) in Figure 2-1 shows another example where the same Rectangle is linked to a Surface ➤ Analysis ➤ Brep Edges node, in which case, an untrimmed Brep enclosed by the Rectangle is constructed and used as input.

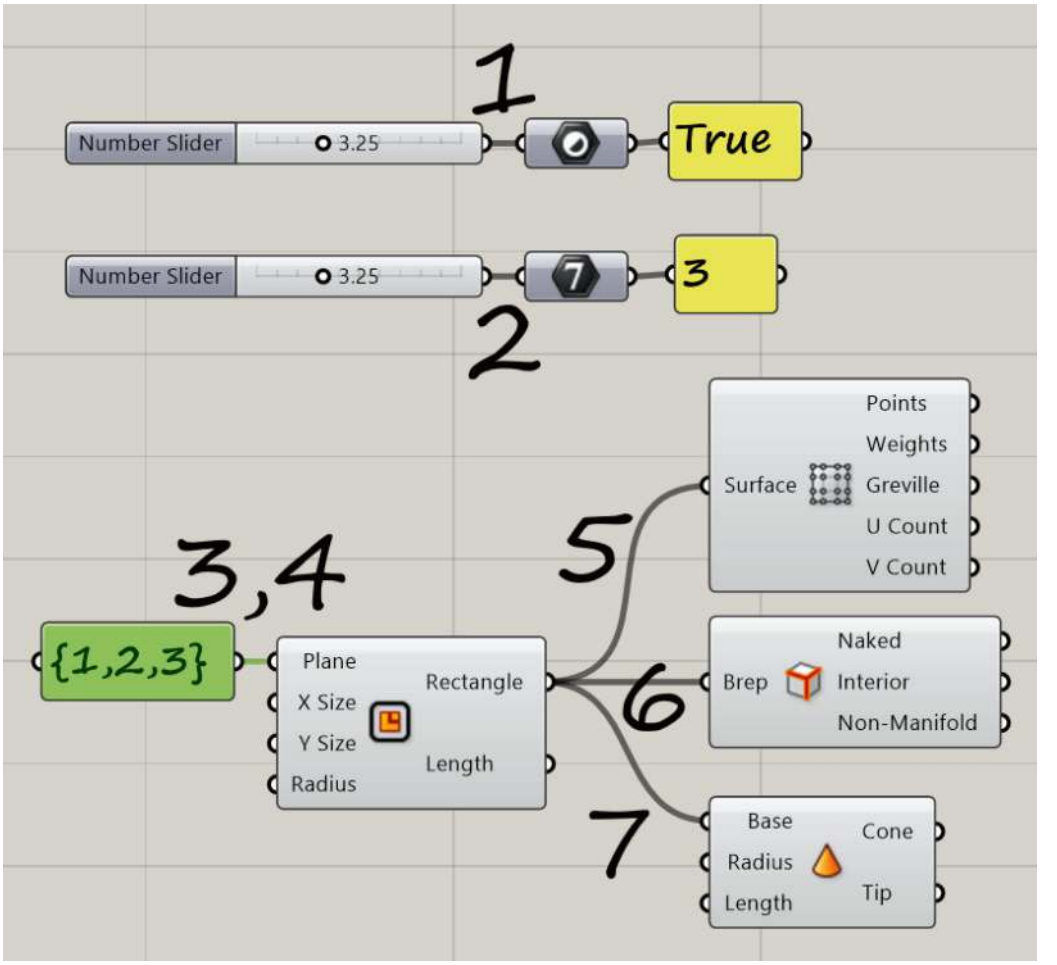


Figure 2-1. *Implicit data conversion in Grasshopper. (1) Number to Boolean. (2) Floating-point number rounded to the nearest integer. (3) Panel with the text {1,2,3} converted to a point with those coordinates. (4) When a point is linked to a plane input, a plane parallel to the XY plane centered at that point is used as input. (5) Planar curve converted to a surface with that curve as its boundary. (6) Planer curve converted to a Brep. (7) When a planar curve is linked to a plane input, the plane the curve sits on is used as input*

Working with Lists

Many Grasshopper components have inputs that can accept either a single value or a list of values—in the latter case, a list of outputs are produced, with each value corresponding to one value in the input list, in the same order. For example, if you hook

a single number to the Radius input of Curve ➤ Primitive ➤ Circle, it produces a single circle; if you hook a list of numbers to that same input, you get a list of circles, one for each radius in the input. In this section, we'll take a closer look at different ways to create and manipulate lists and the effects you can achieve through using lists as inputs.

Range: Creating a Number Sequence by Dividing a Range (with Fixed End)

You can create a sequence of numbers at fixed increments using Set ➤ Sequence ➤ Range. Range is useful when you know the start and end value for the sequence ahead of time and want a certain number of evenly spaced "tick marks" between the two. Let's work through an example. Say we want to create a series of nine cylindrical columns to span 10 meters widthwise along a curve, as shown in Figure 2-2. We can use (1) Range with Domain = 0.0–10.0 and Steps = 8, which will output a list of nine numbers: 0, 1.25, 2.5, ..., 10, with an increment of 10.0/8 = 1.25. Note that even though we specified Steps = 8, Range gives us nine numbers, counting the start and end of the domain.

We can use the nine numbers output by (1) Range as X coordinates for the columns. To find the Y coordinates, we use a Params ➤ Input ➤ Graph Mapper (3) set to Bezier (right-click ➤ Graph types ➤ Bezier), with a (2) X domain of 0.0–10.0 to match the Range domain, and (3) Y domain of 1.0–5.0 (you'd double-click Graph Mapper to open the editor to set those two domains). You can manually drag the control points in the Graph Mapper until it produces a nice shape, as I've done here. The Graph Mapper will output a list of nine Y values scaled to between 1.0 and 5.0 based on the configured Bezier formula, corresponding to the nine X input values.

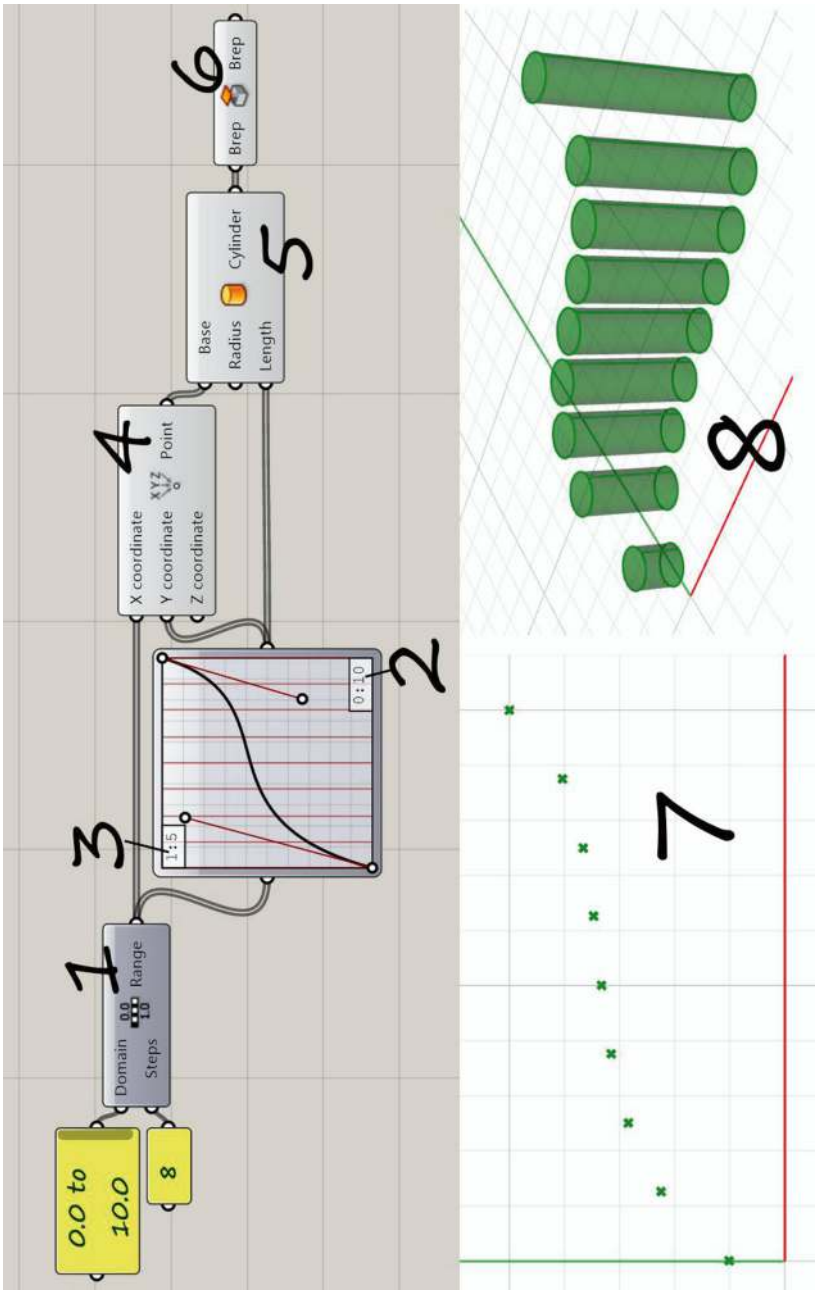


Figure 2-2. *Using Range and Graph Mapper to produce evenly spaced coordinates along a Bezier curve for positioning cylindrical columns. The numbered items are explained in the chapter text that follows*

We can now link the X and Y values to the X coordinate and Y coordinate inputs of (4) Construct Point and leave Z coordinate blank (which defaults to 0)—this will create the list of points shown in (7), which we can use as the base centers for our cylindrical columns. We link the list of points to the (4) Base input of Surface ➤ Primitive ➤ Cylinder, leave the Radius blank to use the default value of 1, and link the list of Y values from the Graph Mapper to the Length input, which controls the height of the cylinders—this will make the columns taller the further they are along X. Finally, we close the top and bottom of the cylinders by linking the (5) Cylinder component to (6) Surface ➤ Util ➤ Cap Holes, which produce the columns shown in (8) of Figure 2-2.

Series: Creating a Sequence by Skip Counting a Given Number of Times (with the End to be Determined)

If you know the start value, count, and step size, you can use Set ➤ Sequence ➤ Series to create the sequence. We'll work through an example in Figure 2-3 where we use Series to generate the radii, angle, and height for a set of arc walls.

Since we are using the output of Series to define the dimensions of arc walls, we use a (1) start value of 1.0, in combination with the default Step value of 1 and Count of 10, to get the list of ten numbers from 1 to 10. Note that Series will output the exact number of values specified by Count, unlike Range, which gives you Count plus one.

Our plan is to use (4) Curve ➤ Primitive ➤ Arc to create ten arcs, as increasing portions of ten concentric circles from the inside out. The list of ten numbers output by Series can be used directly as the Radius input for Arc. As for the Angle input, we'll create a list of values in degrees since it is more human-friendly, then convert them to radians, which is the unit required by Arc. To do so, we'll (2) Multiply the list of ten numbers from Series with 25.0 to get the list 25.0, 50.0, ..., 250.0 degrees, then link it to (3) Maths ➤ Trig ➤ Radians to convert it to a list of radians to connect to the Angle input of (4) Arc. Since both the Arc's Radius and Angle inputs received a list of ten values, GH will automatically pair the two lists up one-to-one, as (r1, a1), (r2, a2), ..., (r10, a10). As a result, (7) ten arcs are produced, with the first arc at the innermost, with radius = 1 and angle = 25.0 degrees; the second arc with radius = 2 and angle = 50.0; and so on.

With the arcs created, we can now extrude them upward along +Z into surfaces with (6) Surface ➤ Freeform ➤ Extrude. We connect the output of (4) Arc to the (6) Base input as geometry to be extruded. To define the extrusion direction and amount, we link the ten numbers output by (1) Series to the Factor input of (5) Vector ➤ Vector ➤ Unit Z to produce ten scaled copies of unit Z vectors (with length 1, 2, ..., 10), then link them to the Direction input of (6) Extrude, which get paired up with the ten input arcs, producing the ten surfaces shown in (8).

Finally, let's compare Range with Series to see when you'd opt for which. You'd use Range when you know the domain and want to divide it into N segments, to get N+1 numbers, in order to have evenly spaced tick marks within an interval, for example, to divide up a curve, or to arrange objects in a confined region.

You'd use Series when you know the start, count, and step size for the sequence you want to generate. Series is useful for creating a steadily increasing sequence of numbers (or decreasing, if a negative step size is used). The sequence can be used to transform objects (e.g., scale or rotate) or to populate geometry at fixed intervals.

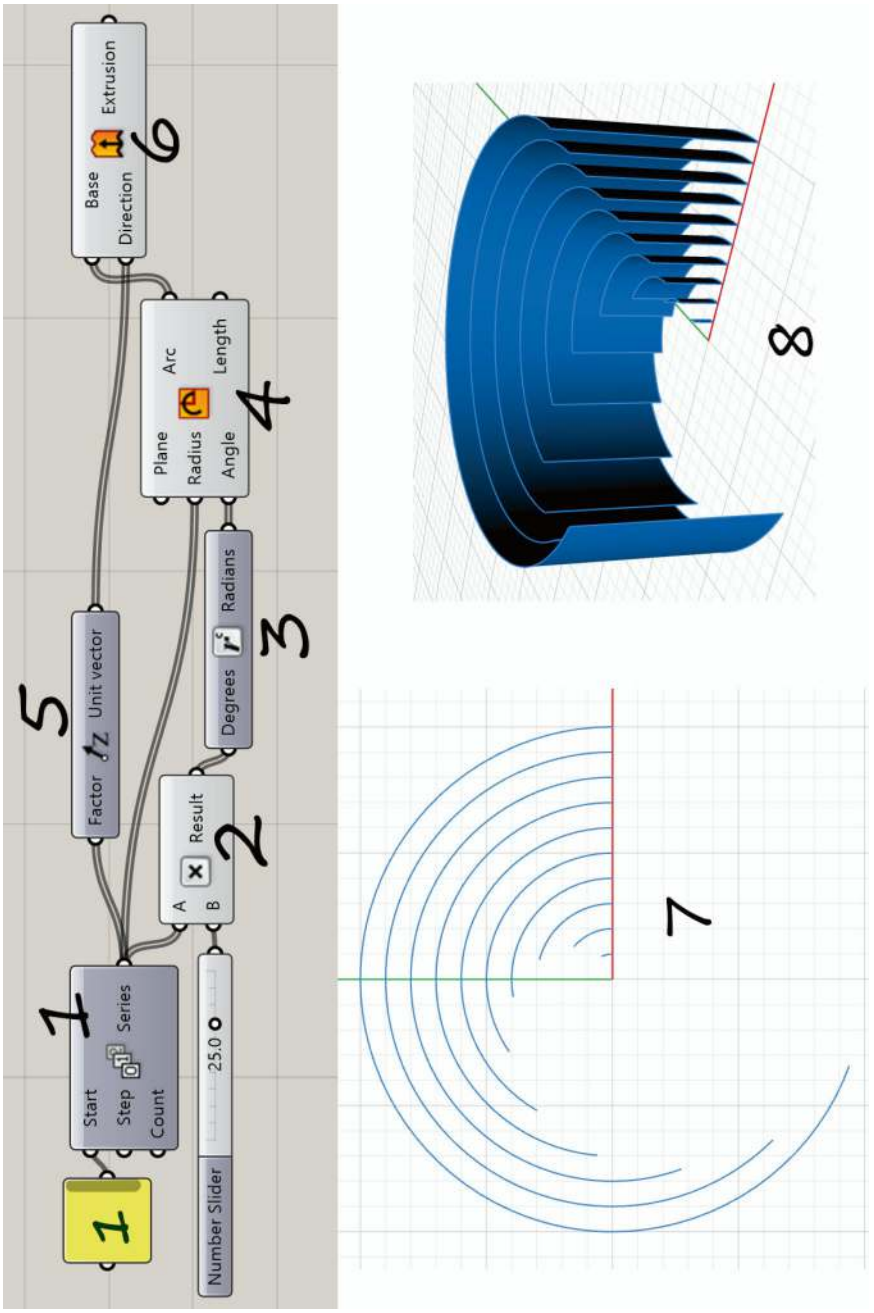


Figure 2-3. *Using Series to generate an increasing sequence of ten numbers to define the radius, angle, and height of a set of arc walls*

Removing Items with a Recurring Pattern (Cull)

In this section, we'll look at two components under Sets ➤ Sequence that will help you filter a list of data with a given pattern. The first is Cull Nth, which removes every Nth item from the given list (N is by default 2, which removes every other item). Figure 2-4 shows an example where we use (2) Curve ➤ Division ➤ Divide Curve to extract an evenly spaced list of ten points around a (1) Circle with radius = 5, which is then filtered with (3) Cull Nth with N = 2 (default), resulting in the list of five points shown in (8).

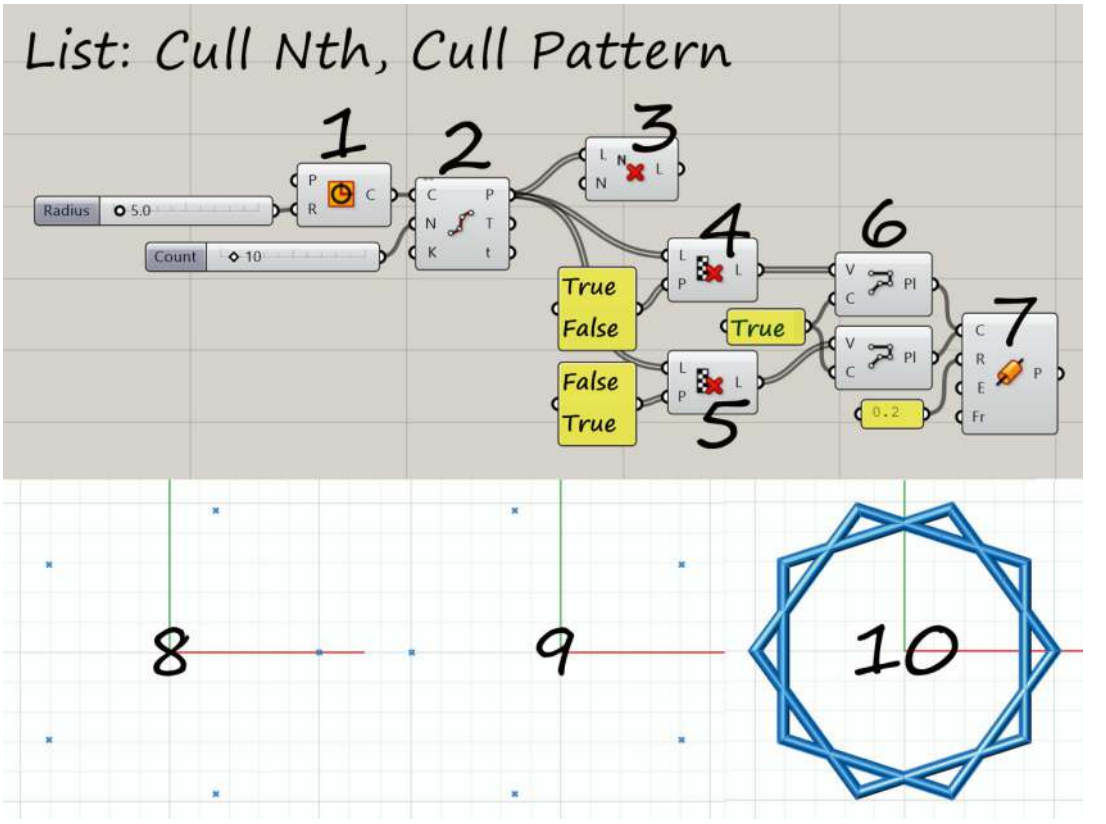


Figure 2-4. *To make a parametric star pattern, we use (2) Divide Curve to get a list of evenly spaced points around a (1) Circle, then use two Cull Pattern nodes (4 and 5) with the patterns True, False and False, True to extract points with even and odd indices in the list (8 and 9), respectively. After that, we link each list of points into a (6) PolyLine and (7) Pipe them both with radius=0.2 to create the geometry shown in (10)*

The second component we'll look at for filtering a list is Cull Pattern, which uses a repeating list of Boolean (True or False) values for removing items—the item whose position coincides with a False in the list is removed. For example, in Figure 2-4, the list of ten points around the circle have indices 0, 1, 2, …., and 9; therefore, when linked to a (4) Cull Pattern node with the pattern True, False, the items with even indices are kept (#0, #2, …) while the items with odd indices (#1, #3, …) are removed, producing the points shown in (8), which are the same as (3) Cull Nth with N = 2. Similarly, if we use a second (5) Cull Pattern node with the pattern False, True, we get the points with odd indices instead, as shown in (9).

We can use (6) two Curve ➤ Spline ➤ PolyLine nodes to create one polyline linking the (8) points with even indices and a second polyline linking the (9) points with odd indices. After that, we (7) Pipe both curves with a radius of 0.2 to create the (10) interweaving star pattern.

Splitting a List (Partition and Sub List)

Sets ➤ List ➤ Partition List lets you split a list into two chunks of given sizes—if the two sizes combined do not use up the list, the pattern continues until the list is exhausted. For example, in Figure 2-5, we use Divide Curve to get 15 evenly spaced points around a Circle with radius = 5 and link them to (1) Partition List, which has 3, 2 for its "Size as list" input by default. Since the list is 15 points long, it is split repeatedly into sublists using the 3, 2 pattern until the list is exhausted (3, 2, 3, 2, 3, 2). If we then interpolate the sublists of points into curves and extrude them ten units along +Z, we get the six walls shown as (3) in Figure 2-5. You can see that the walls are disjointed since the sublists of points do not overlap.

We can link the same 15 points around the circle to a (2) Sub List node to extract a contiguous stretch of the points (as shown, indices 3–8). We then interpolate the points into a curve and extrude ten units along +Z to produce the wall shown in (4).

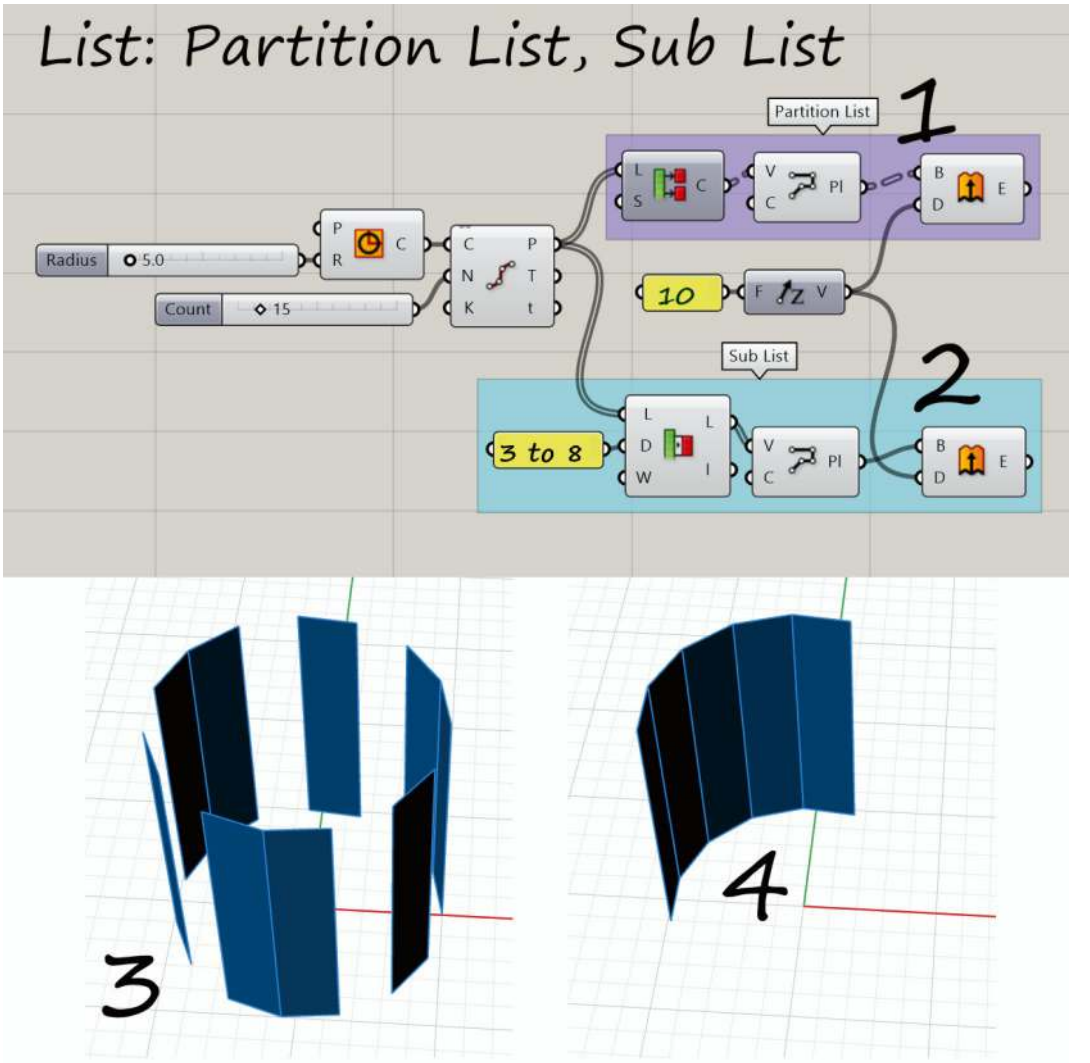


Figure 2-5. *We use Divide Curve to get a list of evenly spaced points around a Circle, then interpolate and extrude the points filtered with (1) Partition List (with default sizes 3, 2) and (2) Sub List (with indices 3 to 8) for comparison, which are shown in (3) and (4) , respectively*

Dispatch and Weave

One of the tricks I love to use in Grasshopper is splitting a list of geometry into two (or more) streams, editing each stream in some way, then weaving them back together to their original positions in the list. In Figure 2-6, we modify the node graph from Figure 2-2, where we position an array of cylinders at evenly spaced points along a (1) Parabolic curve instead of Bezier. The curve domains are adjusted to X=0 to 20 and Y=1 to 15, with the Range Steps (N) changed to 15 to make 16 cylinders. Recall that we take the list of 16 numbers generated by Range between 0 to 20 as X and use the Graph Mapper with Graph type set to Parabolic to find the Y corresponding to these X values. We use the resulting (X, Y) pairs as inputs to Construct Point, leaving the Z's at 0, and use the output points as the bottom centers (Base input) of Surface ➤ Primitive ➤ Cylinder, which creates the cylinders shown in (6) of Figure 2-6.

Our goal is to scale the height of the cylinders in an alternating pattern, two short, two tall, two short, two tall..., and so on. To do so, we split the cylinders into two lists using Sets ➤ Sequence ➤ Dispatch, with the pattern True, True, False, False (2 in Figure 2-6), which produces A = [cylinder #0, #1, #4, #5, #8, #9, #12, #13] and B = [#2, #3, #6, #7, #10, #11, #14, #15]. We then link A to (3) Scale NU with a Z factor of 0.6, leaving X and Y at 1, therefore scaling that half of the cylinders height-wise to 60%. Next, we combine the scaled A list with the original B list with (4) Sets ➤ Sequence ➤ Weave, by linking A to 0, B to 1, using the same pattern as (2) Dispatch—note that to weave dispatched streams of data back to their original indices, you must link the streams in the same order they were split, using the same pattern. Lastly, we make the cylinders solid by linking them to (5) Surface ➤ Util ➤ Cap Holes, which produces the solids shown in (7) in Figure 2-6.

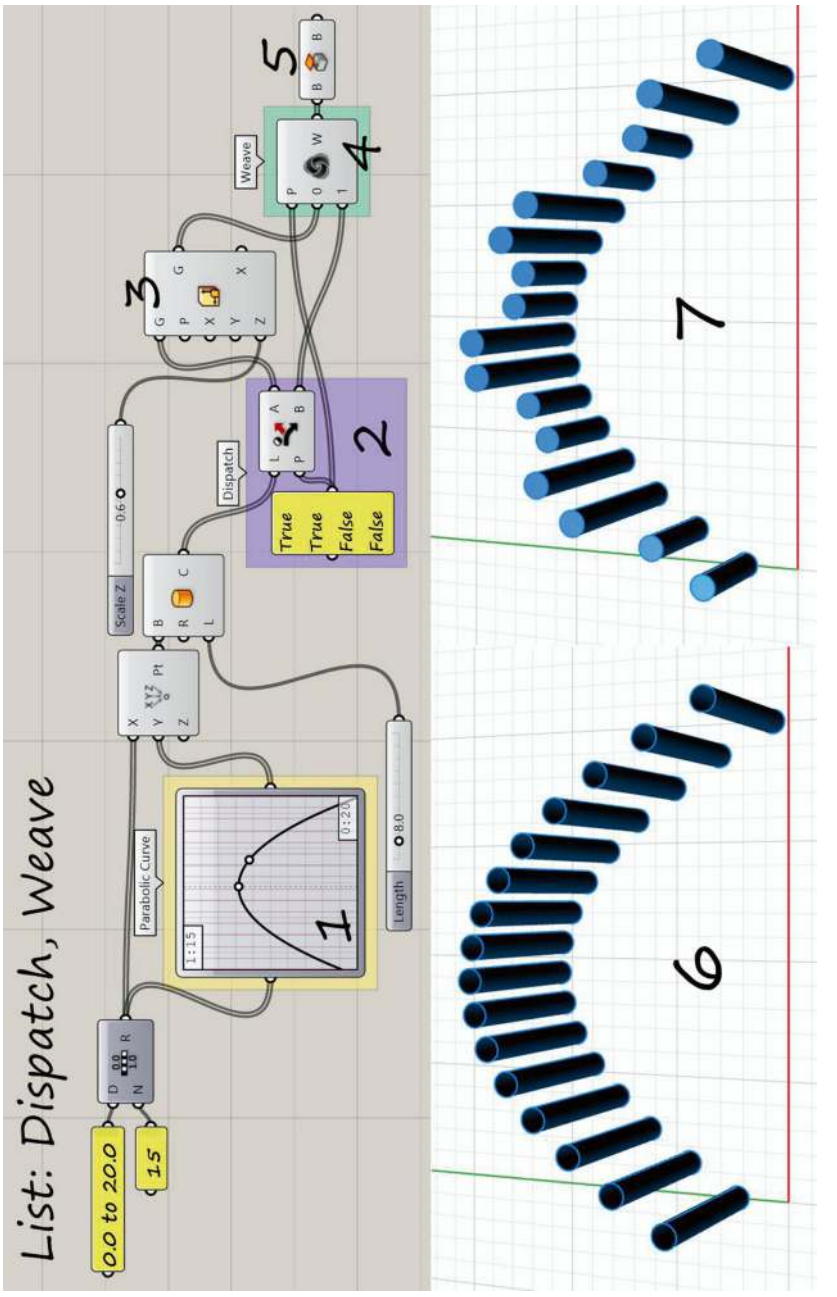


Figure 2-6. *We modify the example from Figure 2-2 by using a (1) different curve type (Parabolic instead of Bezier) and a (2) Cull Pattern node with True, True, False, False to (3) scale the height of every two cylinders, then (4) weave the two batches of cylinders back together and use (5) Cap Holes to make them solid. (6) shows the cylinders before the scaling and (7) the final output of the node graph*

Data Matching

Data matching refers to the way Grasshopper pairs up items between two lists. There are three ways to data match: Longest List (default), Shortest List, and Cross Reference. To compare the difference between the three, we'll set up two lists of points next, then data match them into pairs of end points for creating line segments.

To create two lists of points, we start by making two lines, then divide each line evenly into segments as shown in Figure 2-7. First, we create a line with Line SDL, using (0,0,0) as the Start point (from Construct Point with default params), Unit X as Direction, and a Length of 10 set via a Number Slider—this creates a line from (0,0,0) to (10,0,0), which is shown as (1) in Figure 2-7. We create a second line the same way except with a Length of 15, then displace it using the Move component, which has (0, 0, 10) as its default Motion, therefore moving it ten units above the first line in the Z direction (2 in Figure 2-7). To create points from the two lines, we connect each line to a Divide Curve node with Count set to the line's length (N) to divide it into that many single-unit segments, therefore N+1 points, shown as 3 and 4 in Figure 2-7.

You can find Line SDL under Curve ➤ Primitive, Construct Point under Vector ➤ Point, Unit X under Vector ➤ Vector, Move under Transform ➤ Euclidean, and Divide Curve under Curve ➤ Division.

Longest List (Default)

Longest List (Sets ➤ List ➤ Longest List) matches two lists one-to-one, then pairs up the remainder of the longer list with part of the shorter list. There are five options for specifying which portion of the shorter list to use for matching the leftovers; however, we'll start by looking at the default option—Repeat Last, which pairs the two lists one-to-one then matches the last item of the shorter list with each of the leftover of the longer list.

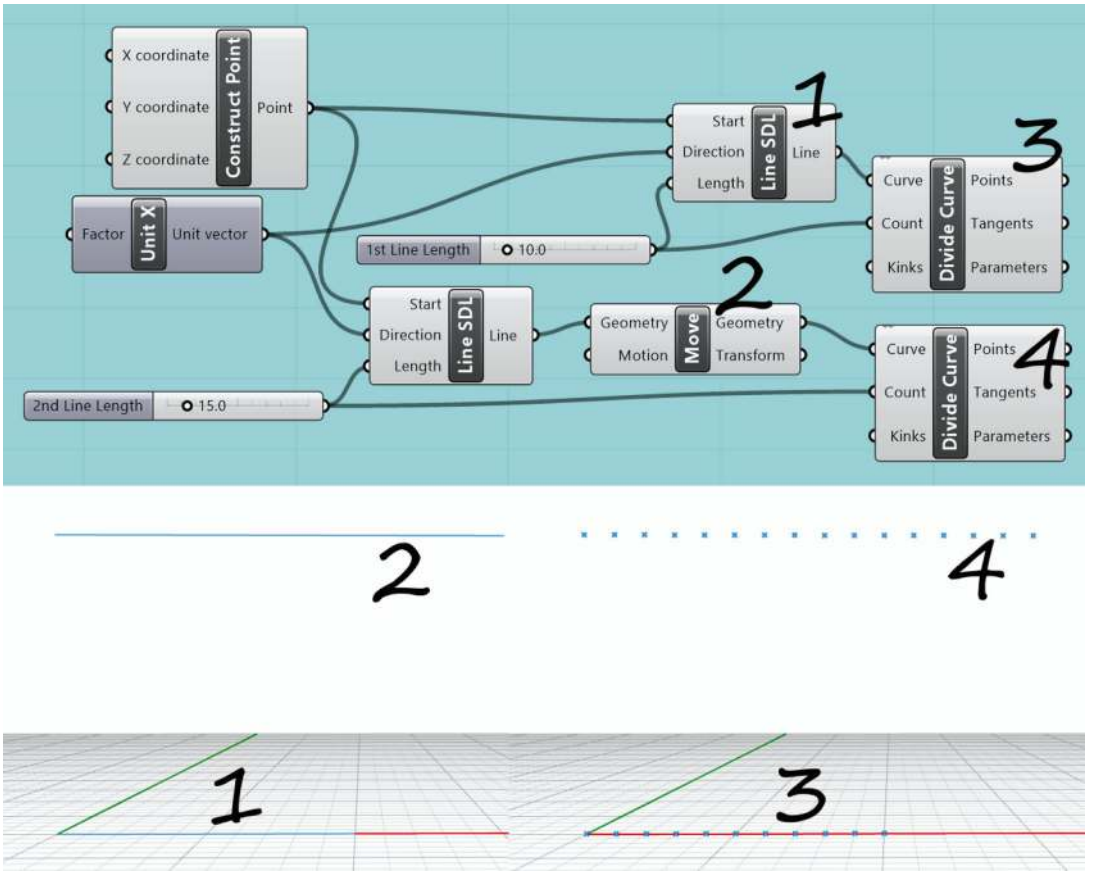


Figure 2-7. *Dividing two lines evenly into segments to create two lists of points. (1) Creating a line with (0,0,0) as Start, Unit X as Direction, and 10.0 as Length. (2) Creating a line from (0,0,0) along X that is 15 units long, then moving it to +10 in the Z direction. (3) and (4) divide the lines from (1) and (2) into one-unit segments, respectively*

In Figure 2-8, we'll extend the node graph from Figure 2-7 to match the two lists of points into lines. We start by linking the Points output of the two Divide Curve components (3 and 4 in Figure 2-8) to List (A) and (B) of Longest List (5) and leave the options for Longest List at default (Repeat Last). We then link the two output lists of Longest List to the Start Point and End Point inputs of Line (6), which produces the lines shown as (1) in Figure 2-9. You can see that after the start of the two lists are lined up one-to-one, the five remaining items in list #2 (top) are matched with the last item of list #1 (bottom). You can find Longest List under Sets ➤ List and Line under Curve ➤ Primitive.

If you don't use a component like Longest List to specify which method to use for data matching, Grasshopper will by default use Longest List with Repeat Last under the hood. The reason you might use the Longest List component explicitly is when you want to specify an option other than Repeat Last for matching, which we'll go over next.

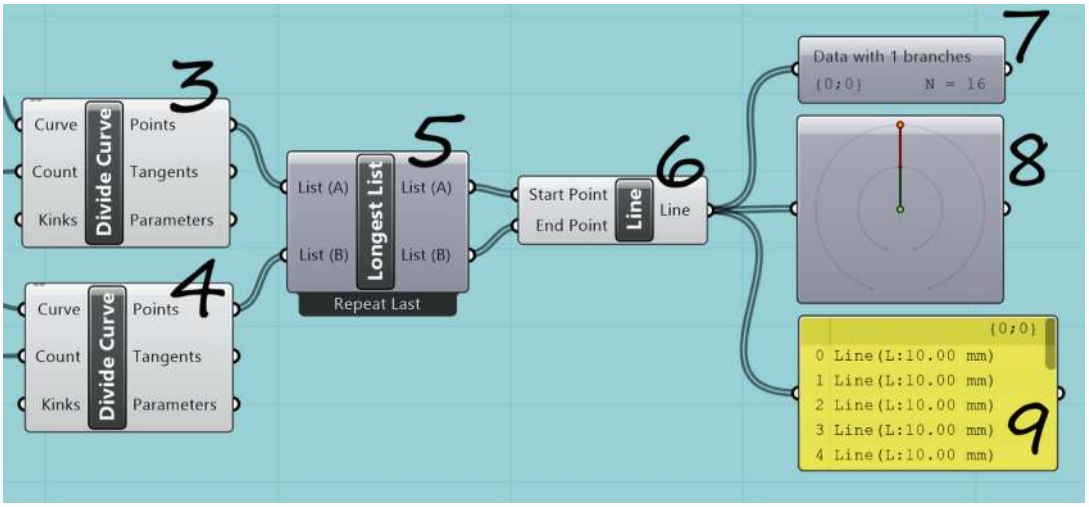


Figure 2-8. *Creating lines by data matching two lists of points using Longest List with Repeat Last. Continuing from Figure 2-7, we connect the Points output of the two Divide Curve components (3 and 4) to List (A) and (B) of Longest List (5), followed by Start Point and End Point of Line (6) to create a data tree of lines. We can inspect the content of the data tree using Param Viewer (7), Param Viewer set to Tree View (8), or a Panel (9)*

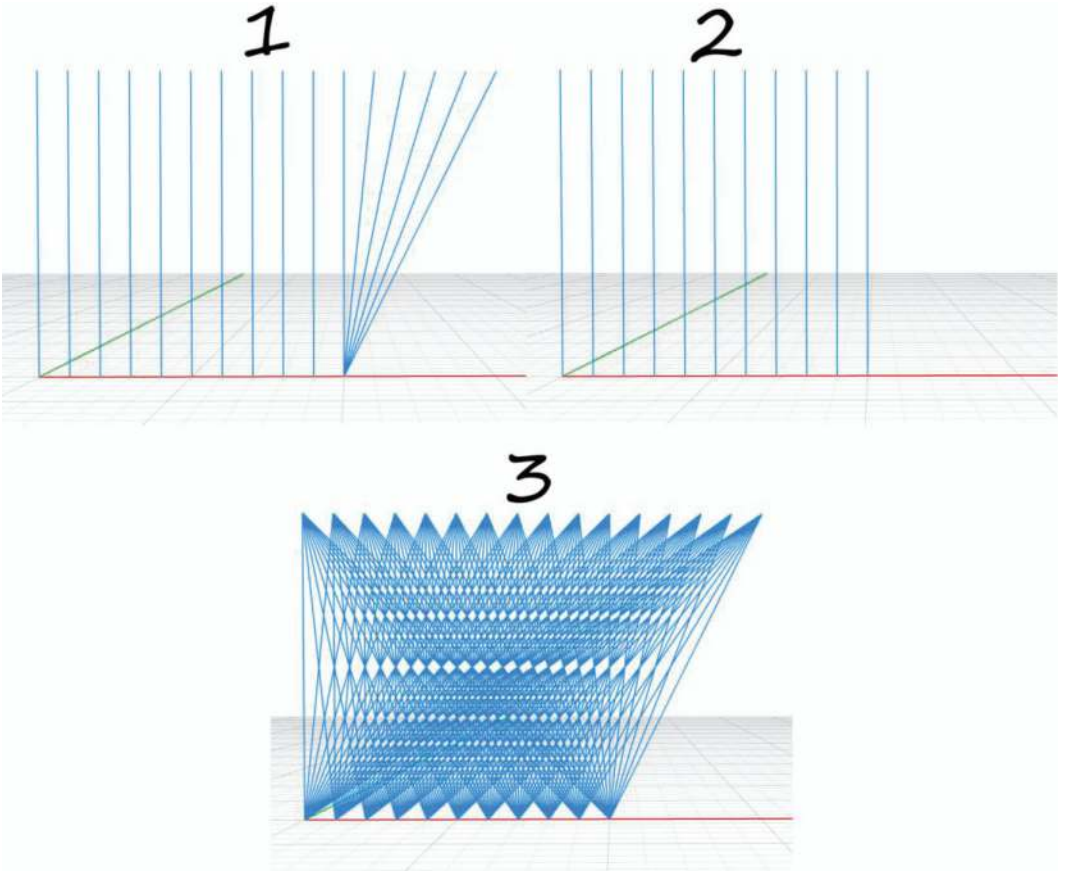


Figure 2-9. *Matching two lists using (1) Longest List with Repeat Last, the default way of data matching in GH, (2) Shortest List with Trim End, and (3) Cross Reference with Holistic*

Longest List Options

Longest List has five options for specifying which part of the shorter list to use and how it is aligned with the "leftover" portion of the longer list. The options are Repeat First, Repeat Last, Interpolate, Wrap, and Flip, as shown in Figure 2-10.

With "Repeat First," you line up the front of the longer list with the first item of the short list as many times as it takes until what's left of the longer list is the same length as the shorter list. We've already seen "Repeat Last" in the previous section, which is the default option and matches the two lists one-to-one, then each item in the leftover of the longer list with the last item of the shorter list. "Interpolate" attempts to evenly distribute items of the longer list to the shorter list (many to one). "Wrap" starts by matching the

two lists one-to-one from the beginning until the shorter list runs out, then matches the remainder of the longer list one-to-one with the shorter list again from the *beginning*. "Flip" acts like Wrap but matches the leftover portion of the longer list one-to-one with the shorter list in reverse order (starting from the end).

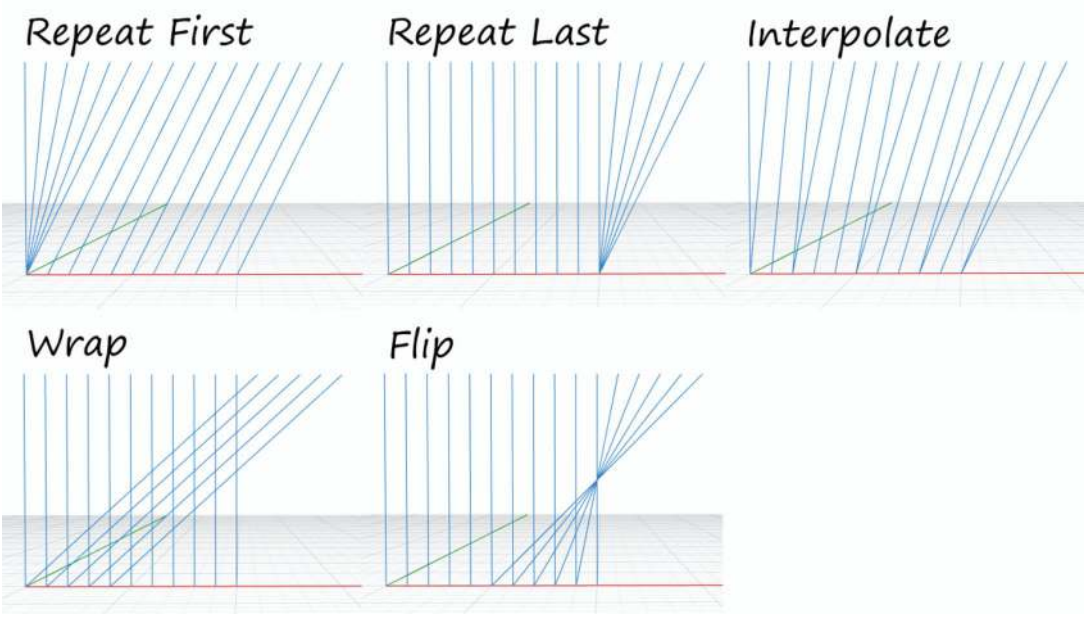


Figure 2-10. *Five options for Longest List Matching: Repeat First, Repeat Last (default), Interpolate, Wrap, and Flip*

Shortest List

Shortest List (Sets ➤ List ➤ Shortest List) matches two lists one-to-one by default from the beginning (Trim End), then chops off the remainder of the longer list. To modify the node graph in Figure 2-8 to use Shortest List, simply replace (5) with Shortest List, which produces the second set of lines shown in Figure 2-9.

In addition to "Trim End," there are two other options available: "Trim Start," which chops up the beginning of the longer list, leaving the same length as the shorter list, and "Interpolate," which intersperses the removal of items from the longer list until the two lists are the same length.

Cross Reference

Cross Reference (Sets ➤ List ➤ Cross Reference) by default matches two lists so that each item of the first list is paired with every item of the other list (Holistic). To modify the node graph in Figure 2-8 to use Cross Reference, simply replace (5) with Sets ➤ List ➤ Cross Reference, which produces the third set of lines shown in Figure 2-9.

In addition to "Holistic," there are numerous other options available for specifying which portion of one list is (or is not) matched with which portion of the other list, such as "Diagonal," which skips pairs of items with the same indices from the two lists.

Data Matching: More Examples

Let's see some examples of data matching in action. In Figure 2-11, we create two points with Vector ➤ Point ➤ Construct Point, the first with default inputs at (0, 0, 0) and the second at (0, 5, 0), then use them as end points for Curve ➤ Primitive ➤ Line to create a line segment 5.00 mm long (1 in Figure 2-11). Linking the Line node to a Transform ➤ Affine ➤ Scale NU node with Y = 2 will produce a line segment twice as long, at 10.00 mm (2). If we link the same Line node to a Scale NU node with a list of three scale factors (2, 4, and 6), we'll get three lines, with lengths 10.00 mm, 20.00 mm, and 30.00 mm, corresponding to the three scale factors (3 in Figure 2-11):

$$5mm \times 2 = 10mm$$

$$5mm \times 4 = 20mm$$

$$5mm \times 6 = 30mm$$

Let's try pairing a list of lines to various numbers of scale factors and see what happens. In the first example (top) in Figure 2-12, we link a list of three numbers [5, 10, 15] to the X input of the first Construct Point node, thus creating a list of three points: (5, 0, 0), (10, 0, 0), and (15, 0, 0). We link the same list of numbers to both the X and Y inputs of the second Construct Point node, thus creating another list of three points: (5, 5, 0), (10, 10, 0), and (15, 15, 0). Since the two lists of points are the same length, when they are used as the A and B inputs for the Line node, the points are paired up at each index, creating three lines:

From (5, 0, 0) to (5, 5, 0), with length = 5.00 mm
From (10, 0, 0) to (10, 10, 0), with length = 10.00 mm
From (15, 0, 0) to (15, 15, 0), with length = 15.00 mm

If we link the above list of lines to a Scale NU node with Factor = 2, we get another list of three lines, each scaled to 2X the length (second example in Figure 2-12). If we use the list [2, 4, 6] as the Factor input for Scale NU (third example), we get a list of three lines, scaled to 2X, 4X, and 6X the length since the length of the Factor list is the same as the number of input lines. If we use the list [2, 4] as Factor for Scale NU, however, since there are more lines than scale factors, using the "Longest List–Repeat Last" principle, the last item of the shorter list [2, 4] is repeated, so the three lines are scaled to 2X, 4X, then 4X again (fourth example in Figure 2-12).

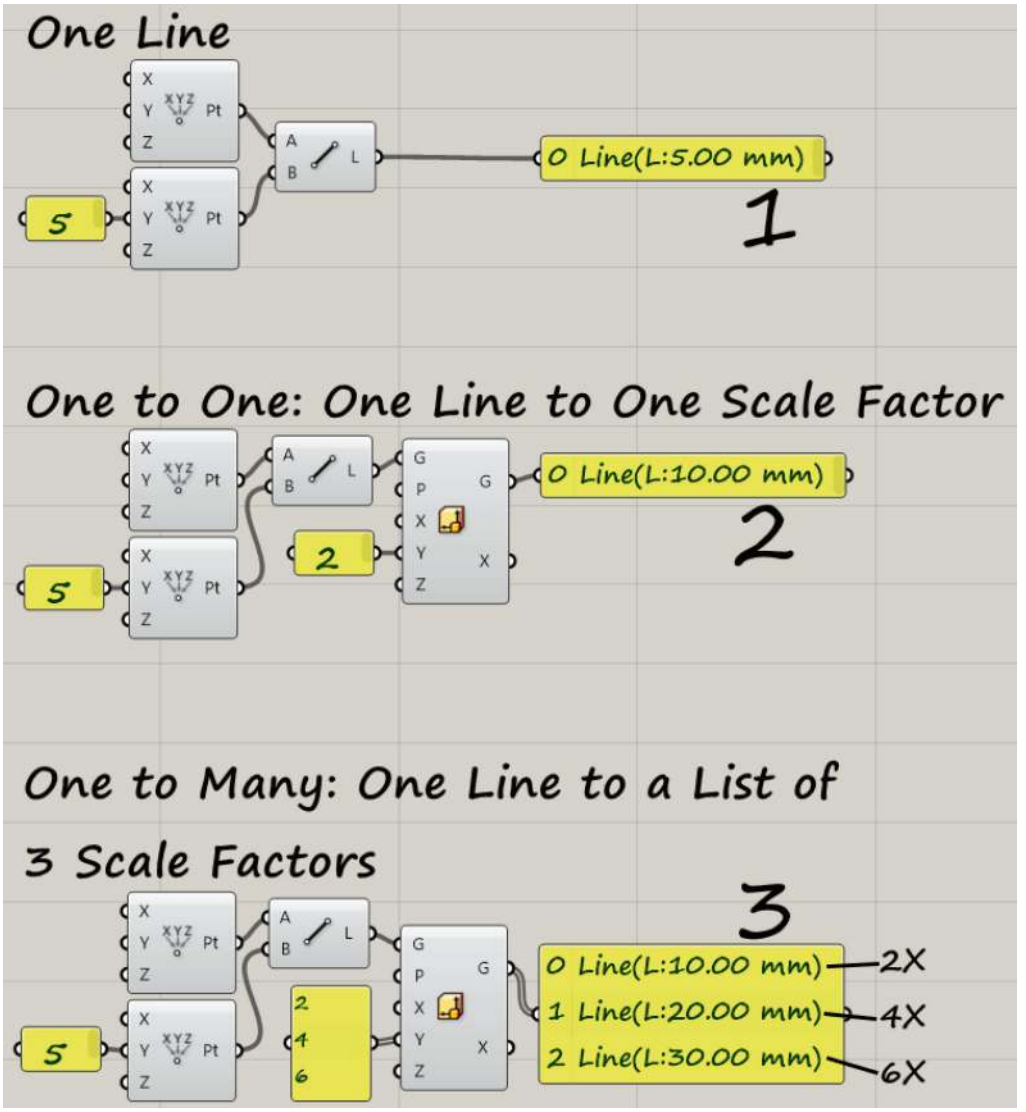


Figure 2-11. *(Top) One line from two end points. (Middle) The same line scaled with Scale NU with Y=2, producing one line with twice the length. (Bottom) The same line scaled with Scale NU with Y = 2, 4, 6, producing a list of three lines, at 2X, 4X, and 6X the length, respectively*

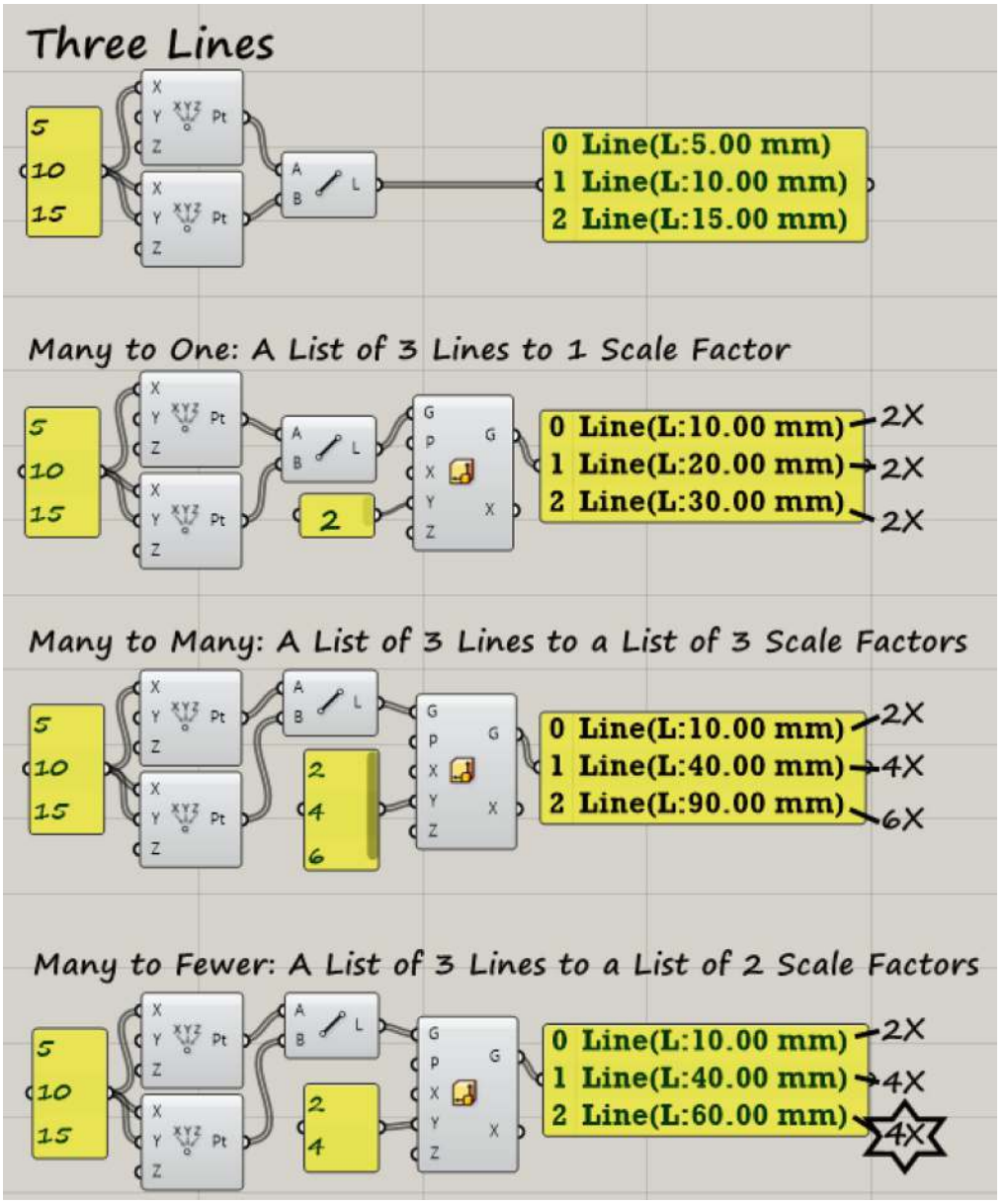


Figure 2-12. *(Top) Three lines of lengths 5mm, 10mm, and 15mm. (2nd) Scaling the three lines using Scale NU with a single Factor = 2, producing three lines, each twice the length. (3rd) Scaling the three lines using Scale NU with three scale Factors, producing three lines at 2X, 4X, and 6X the length. (4th) Scaling the three lines using Scale NU with only two scale factors, thus applying Longest List–Repeat Last for data matching, we end up with three lines at 2X, 4X, and 4X (again) the length, respectively*

Data Trees (Nested Lists)

Data is organized in Grasshopper in three ways: as a single item, as a list (a linear sequence of items), or as a data tree (a nested list). In this section, we'll take a closer look at data trees and the operations available to modify them.

Let's create a data tree. In Figure 2-13, we take a (1) Surface ➤ Primitive ➤ Cylinder node with Radius = 5 and Length (height) = 15.0 and derive an evenly spaced grid of points along its surface with (2) Surface ➤ Util ➤ Divide Surface. By default, Divide Surface has U = 10 (number of horizontal divisions) and V = 10 (number of vertical divisions); therefore, we get

$$(10+1)\times(10+1)=121\ points$$

The cylindrical surface is shown in (5), and the divided points are shown in (6).

We can see a textual summary of these points' data by linking them to a Params ➤ Util ➤ Param Viewer node (4 in Figure 2-13)—in this case, a data tree composed of 11 branches, with each branch containing 11 items. The strings {0;0;0}, {0;0;1}, ...,{0,0,10;} are called the *data paths* (or paths for short) of the branches. A *path* is like an address that tells you how to navigate the tree to get to a specific branch. Take the path {0;0;1} as an example—the three numbers separated by semicolons tell you the tree has three levels. You take the first (0^{th}) branch of the first level, go down the first (0^{th}) branch of the second level, then the second (1^{st}) branch of the third level.

You may have noticed in this data tree the first two numbers of *every* path are zeros—which means each of the first two levels contains only one branch, therefore making them redundant. We can "Simplify" the tree by removing these two levels, which we'll discuss in more detail in the next section.

You can make Param Viewer display an image of the data tree instead of text with right-clicking ➤ Draw Tree, which shows the tree (3 in Figure 2-13) as three concentric arcs, with each arc representing one level. Following the arcs from inside out, we see one root (first level), that leads into one trunk (second level), which further splits into 11 branches (third level).

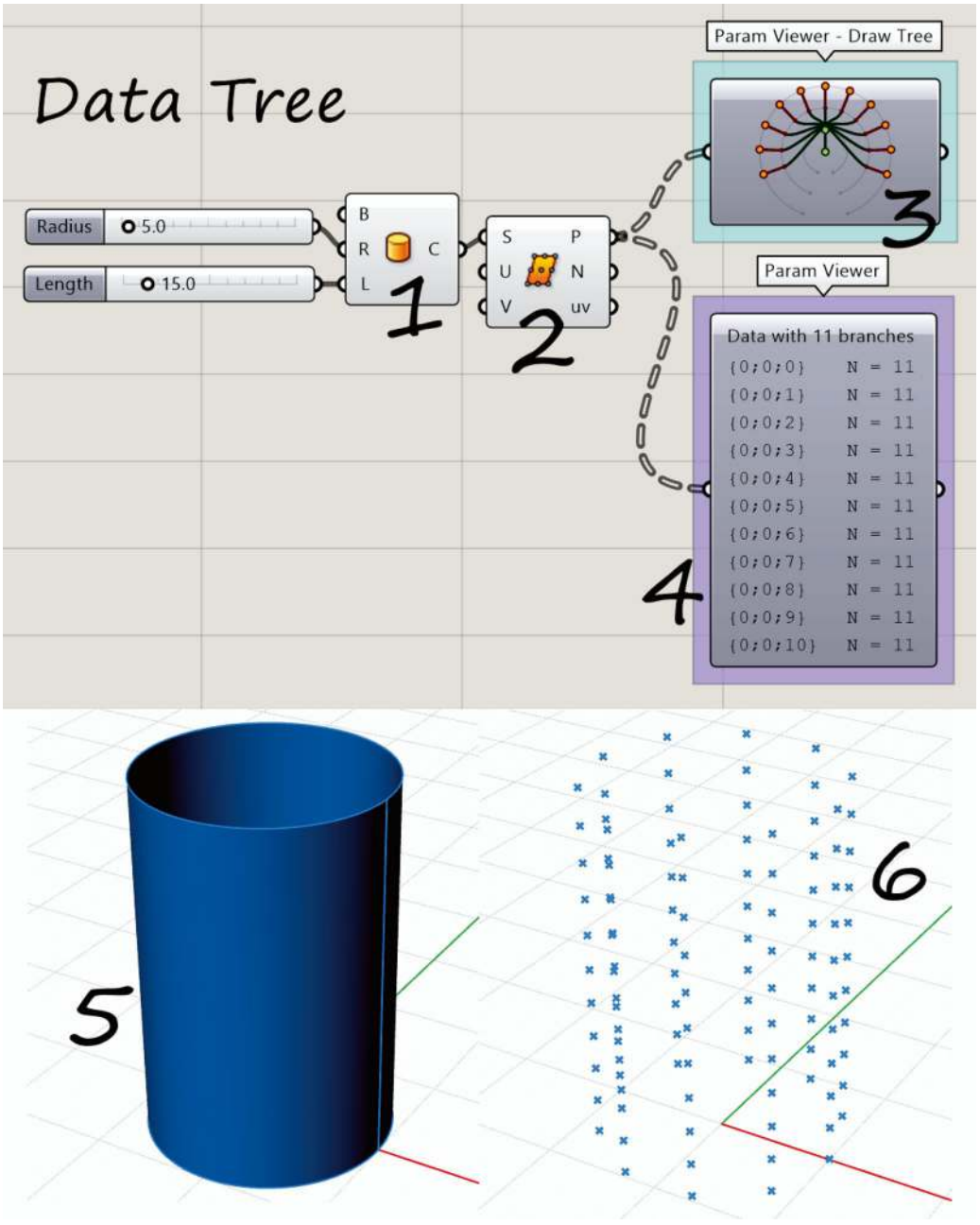


Figure 2-13. *Linking a (1) Cylinder to a (2) Divide Surface node produces a data tree, of which the (3) branching structure we can view using a (3) Param Viewer node with right-clicking ➤ Draw Tree. We can also use Param Viewer with default settings to see the branches (4) summarized as text (11 branches, each with 11 items). (5) is the cylindrical surface produced by (1), and (6) is the points output by (2)*

Simplify (Remove Unnecessary Branching) and Flip Matrix (Rows Become Columns)

We'll continue with the example from the previous section where we use Divide Surface with U=10 and V=10 to extract an 11x11=121 grid of points from a cylindrical surface. Recall that Param Viewer ➤ Draw Tree (3 in Figure 2-13) showed the points as a data tree three levels deep—this structure is clearly redundant, as the grid is 2D, therefore 11 rows by 11 columns.

We can remove the unnecessary nesting in the data tree by right-clicking the Points output of Divide Surface ➤ Simplify, which will reduce the tree to a single level composed of 11 branches, with each branch containing 11 items (2 and 3 in Figure 2-14). Notice that the data paths displayed by the Param Viewer have gone from {0;0;0}, {0;0;1}, ... in (4) of Figure 2-13 to {0}, {1}, ... in (3) of Figure 2-14—the shortening of paths following Simplify make accessing branches and items in the tree easier, as we'll see next.

So far, we know the data tree has 11 branches, with each branch containing 11 items. However, we don't know where each branch of points are on the cylinder—is it one vertical column of points? Or one horizontal ring of points? An easy way to find out is to use a Sets ➤ Tree ➤ Tree Branch node to retrieve the first branch (4 in Figure 2-14) and preview it in the Rhino viewport—doing so confirms that each branch is one vertical column of points. Note that since we've simplified the data tree, the paths to the branches are {0}, {1}, {2}, ..., and so on.

What if we want to access the horizontal rings of points instead? The data tree's current structure does not allow us to do this easily, as we'll need to collect the Nth item of every branch to get the Nth ring. Luckily, there is a much simpler solution. If we link the data tree to a Sets ➤ Tree ➤ Flip Matrix node, it will turn each "column" into a "row" and vice versa (i.e., transpose the 2D matrix)—since a "column" in our data tree is one vertical line of points (4 in Figure 2-14), a "row" is one horizontal ring of points (5 in Figure 2-14). Another way to think about this is to imagine the cylinder as a rolled sheet of paper—if you unroll it into a flat rectangle, draw a grid of points on it, then roll it back up; each row of points becomes a ring.

You can simplify any input or output with right-clicking ➤ Simplify. Alternatively, you can connect an output to a Sets ➤ Tree ➤ Simplify Tree node, which produces the same effects, except with an additional option to limit the path collapse to the indices at the start of the paths.

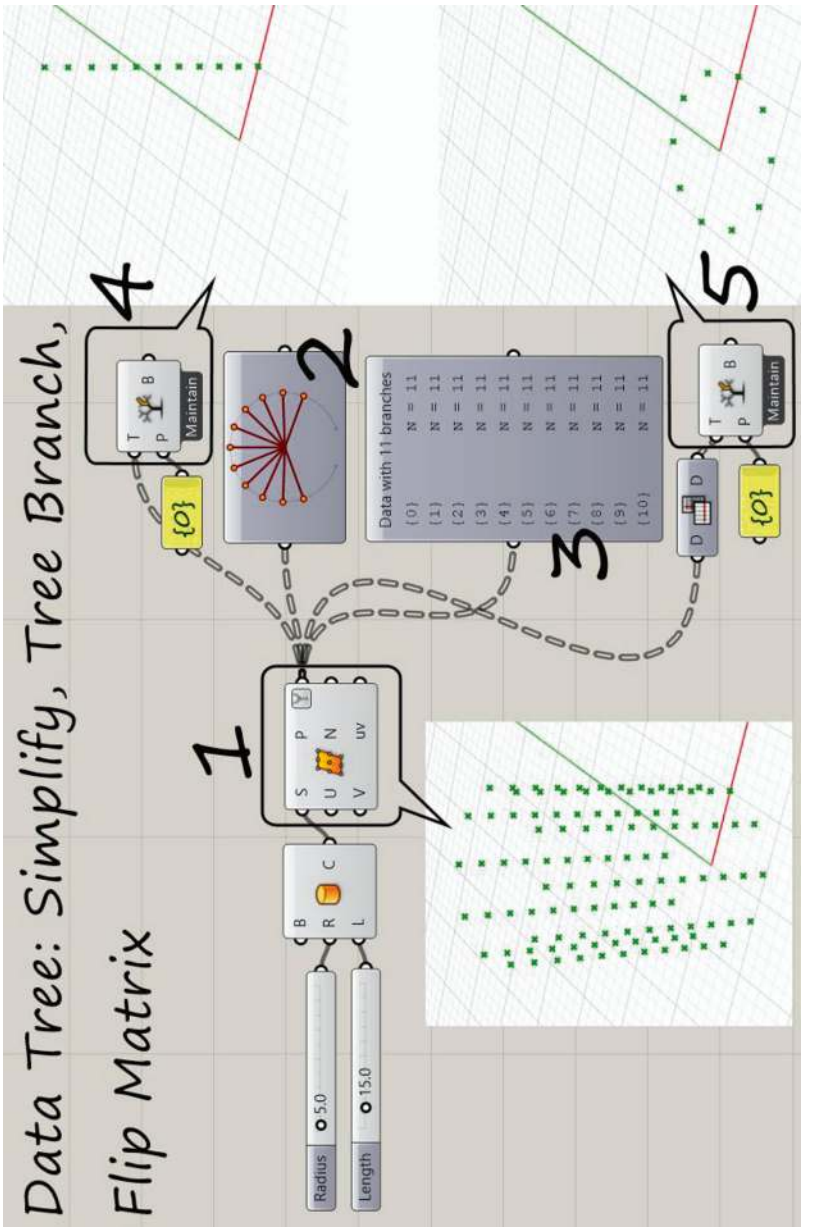


Figure 2-14. *The data tree produced by (1) Divide Surface can be simplified via right-clicking on its Points output ➤ Simplify, which reduces it to a nested list, as shown by the two Param Viewer nodes (2—right-click ➤ Draw Tree, and 3—default settings). To better understand the tree's structure, we can use a (4) Tree Branch node to access its 0th branch and preview it in the viewport, which shows a single vertical column of points. If we use (5) Flip Matrix to turn the columns of the tree into rows, the 0th branch becomes a single horizontal ring of points*

Flatten: Convert a Data Tree into a List

You can convert a data tree into a list by right-clicking any input or output ➤ Flatten or connect an output to a Sets ➤ Tree ➤ Flatten Tree node. Using the same example from Figure 2-13, we right-click ➤ Flatten the Points output of Divide Surface (1 in Figure 2-15) to convert it from a data tree of 11 branches with 11 points per branch to a flat list with 121 points, as confirmed by Param Viewer (2). We can view the coordinates of the points in the list by linking it to a Panel (3).

Grasshopper flattens the data tree by appending the second branch to the end of the first, the third to the end of the second, and so on. Recall we found in the previous section that each branch is one vertical column of points around the cylinder. Therefore, when the tree is flattened, the points go from column to column. We can visualize the ordering of the points by linking the list to an Interpolate Curve node (4 in Figure 2-15), followed by (5) Pipe with radius = 0.2, which produces the geometry shown in (6). By cross-referencing the preview of (6) and the point coordinates in the (3) Panel, we can see that in the flattened list, the points go up one column, then hop over to the bottom of the next column before going up again, and so on.

Flattening a data tree is useful when you want to perform an edit on *every* item of a tree separately. Grasshopper uses the structure of the data coming into an input to infer how to operate on it—when it is a list, each item in the list is edited in turn, either with the same operation if the other inputs have single values, or each with a different operation if the other inputs are lists of the same length. If the inputs receive lists of different lengths, by default, the "Longest List–Repeat Last" rule of data matching applies, where the start of the two lists are matched one-to-one, then each leftover of the longer list is matched with the last item of the shorter list (see the earlier section "Data Matching" for how this works).

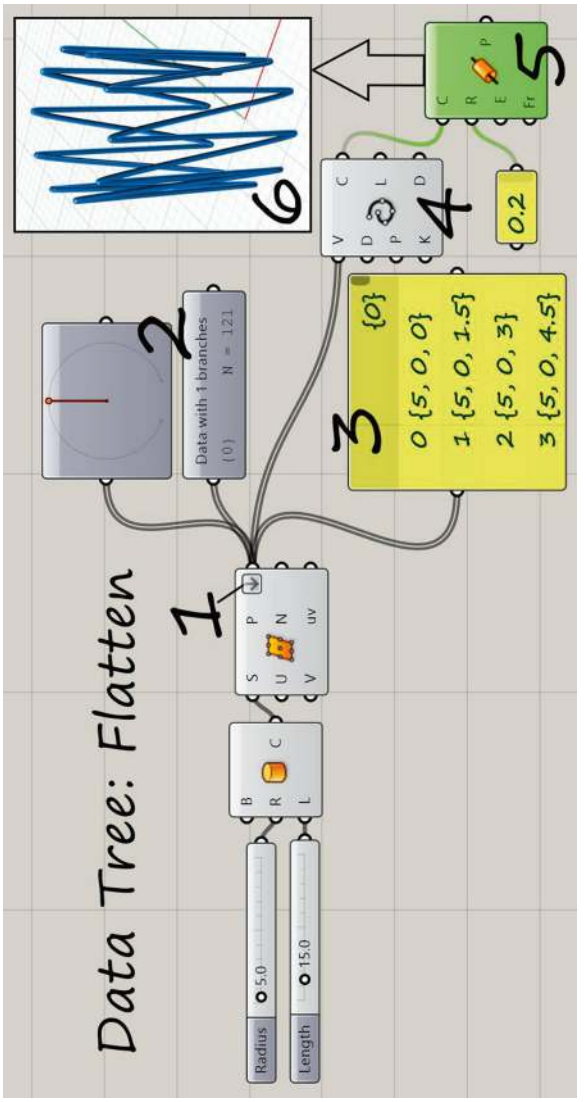


Figure 2-15. *We can convert the output of (1) Divide Surface into a list by right-clicking its Points output ➤ Flatten, which when linked to a (2) Param Viewer (right-click ➤ Draw Tree) shows a single branch with 121 items and when linked to a (3) Panel shows a list of 121 coordinates. If we connect the points list to (4) Interpolate Curve followed by (5) Pipe with a radius of 0.2, we get the zigzag structure in (6)*

Graft (Turn Every Item into a Tree Branch)

In the last section, we experimented with *flattening* a data tree, which removes all but one branch and converts the tree into a flat list. In this section, we'll do the opposite—by *grafting* a list; we turn it into a data tree by making *every* list item a tree branch.

In Figure 2-16, we use a (1) Series node with S = 0, N = 2, C = 10 to create a list of 10 numbers [0, 2, 4, ..., 18] and link it to both the X and Y inputs of (2) Construct Point to create a list of ten diagonal points, as visualized in (1) in Figure 2-17. If we right-click ➤ Graft the Y input of Construct Point (5 in Figure 2-16), we convert Y from a list of ten numbers to a data tree with ten branches, each with one number [[0], [2], [4], ..., [18]]. As a result, we force Grasshopper to match the following:

$$X = [0,2,4,\ldots,18] \; with \; Y = [0]$$

$$X = [0,2,4,\ldots,18] \; with \; Y = [2]$$

$$X = [0,2,4,\ldots,18] \; with \; Y = [4]$$

$$\cdots, and$$

$$X = [0,2,4,\ldots 18] with Y = [18]$$

Since each of the above pairing tries to match a longer list to a shorter one, using the principle of "Longest List–Repeat Last," X is matched to Y one-to-one (the first item of X to the first item of Y), then Y runs out, so the last item in Y is repeated nine times to match the rest of X. As a result, each pairing produces ten points:

$$X = [0,2,4,\ldots,18] \; with \; Y = [0,0,0,\ldots,0] \rightarrow (0,0),(2,0),(4,0),\ldots,(18,0)$$

$$X = [0,2,4,\ldots,18] \; with \; Y = [2,2,2,\ldots,2] \rightarrow (0,2),(2,2),(4,2),\ldots,(18,2)$$

$$X = [0,2,4,\ldots,18] \; with \; Y = [4,4,4,\ldots,4] \rightarrow (0,4),(2,4),(4,4),\ldots,(18,4)$$

$$\ldots, and$$

$$X = [0,2,4,\ldots,18] \; with \; Y = [18,18,18,\ldots,18] \rightarrow (0,18),(2,18),(4,18),\ldots,(18,18)$$

Hence, we get a total of

$$(10 \; pairings) \times (10 \; points \; per \; pairing) = 100 \; points$$

which is structured as a data tree of ten branches, with ten points per branch (6 and 7 in Figure 2-16), as visualized in (2) of Figure 2-17.

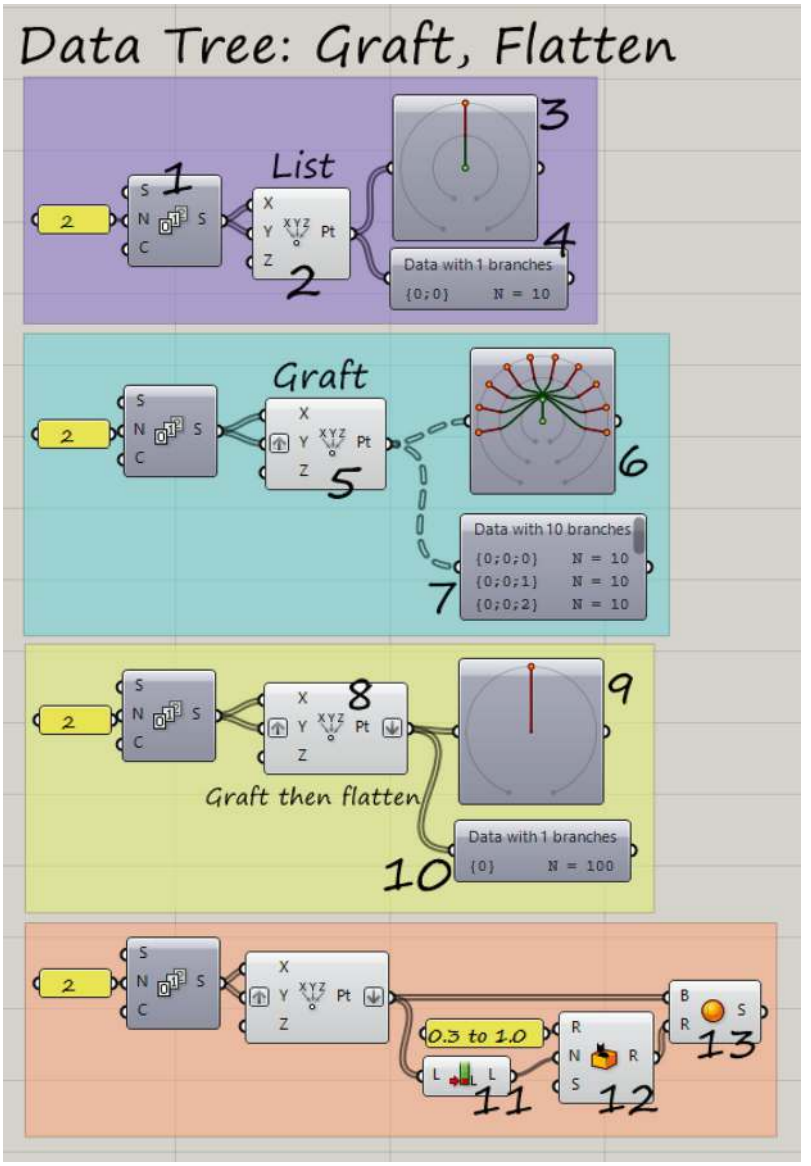


Figure 2-16. *We edit the output of a (1) Series node (with starting value 0, step size=2, and count=10) and observe the effects. When linked to both the (2) X and Y inputs of Construct Point, we get a list of ten diagonal points (3 and 4), which are shown as (1) in Figure 2-17. If we right-click ➤ Graft the Y input of Construct Point, we get a data tree with ten branches (6 and 7), each with ten items (2 in Figure 2-17). If we (8) flatten the tree, we get a list of 100 points (8 to 10, as 2 in Figure 2-17), which can then be used as centers for (13) Sphere with (12) Random radii between 0.3 and 1.0 (3 in Figure 2-17)*

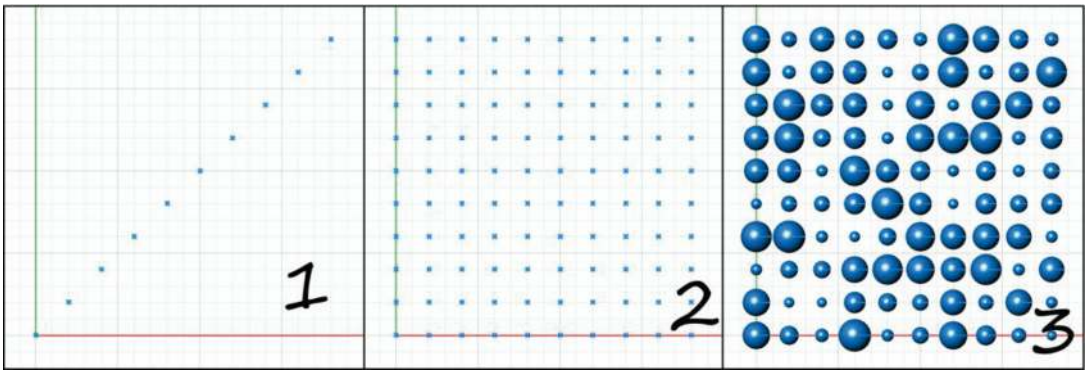


Figure 2-17. *(1) Ten diagonal points produced by linking the output of Series (0, 2, 4, ..., 18) to both the X and Y inputs of Construct Point (2 in Figure 2-16). (2) 100 points produced by grafting the Y input of Construct Point (5 and 8 in Figure 2-16). (3) Spheres with radii from 0.3 to 1.0 produced by using the points from (2) as centers and random radii (13 in Figure 2-16)*

If we subsequently flatten the output of Construct Point (8 in Figure 2-16), we get a flat list of 100 points (9 and 10), which can then be used as the Base input for Sphere (13) to populate a 2D grid of spheres centered at those points. We can make the grid more interesting by using a (12) Sets ➤ Sequence ➤ Random node to generate values for the Sphere node's Radius input. We create a list of 100 random radii, one for each sphere, by using the (11) List Length of the points list as the Number (N) input and Range (R) = 0.3 to 1.0 for the (12) Random node. The spheres produced by (13) are visualized in (3) of Figure 2-17.

You can graft any input or output of a component with right-clicking ➤ Graft. There is also a Sets ➤ Tree ➤ Graft Tree node available for grafting a data tree.

Reverse

In the last section, we grafted the Y input of Construct Point to turn a diagonal line of points into a 2D grid, then flattened the grid's data tree into a list of points to use as Sphere centers so we can vary their radii on an individual basis. In this section, we'll try something different—instead of flattening, we'll simplify the tree before linking it to Sphere's Base input so we can vary the radii on a per-tree-branch basis, as shown in Figure 2-18.

After simplifying the output of Construct Point, we link it to a Sets ➤ Tree ➤ Tree Statistics node (1 in Figure 2-18), which gives us the number of paths via its Count (C) output. We want to vary the radii for (4) Sphere such that the starting radius is half the distance between neighboring spheres (therefore making them tangent), and the radius *decreases* as the path index of the tree branch *increases*. The Step (N) of the Series node we used to populate the point grid as sphere centers is equal to the distance between sphere centers; therefore, by making the radius half of Step, the spheres will touch (as shown on the far-left column of 7 in Figure 2-18)

$$Starting\ radius = Step\ of\ Series \div 2 \rightarrow 1st\ branch, path\{0\}$$

We want the radius to be inversely proportional to the branch index, as in

$$Starting\ radius \div 2 \rightarrow 2nd\ branch, path\{1\}$$

$$Starting\ radius \div 3 \rightarrow 3rd\ branch, path\{2\}$$

$$Starting\ radius \div 4 \rightarrow 4th\ branch, path\{3\}$$

$$..., and$$

$$Starting\ radius \div 10 \rightarrow 10th\ branch, path\{9\}$$

By using a second Series node with Start (S) = 1, the default Step (N) = 1, and Count (C) = number of paths in the points data tree (2 in Figure 2-18), we generate the list of ten numbers, [1, 2, 3, ..., 10], for dividing the starting radius to get the radius for each branch.

As shown in Figure 2-18, we use 3.0 from a Number Slider as Step (N) for the first Series node for generating the point grid; therefore:

$$Starting\ radius = 3.0 \div 2 = 1.5$$

Dividing 1.5 by the second Series output, which is the list of numbers counting from 1 up to the number of branches (paths) in the points tree, will then give us the list of radii for Sphere:

$$\left[1.5 \div 1, 1.5 \div 2, 1.5 \div 3, \ldots 1.5 \div 10\right] = \left[1.5, 0.75, 0.5, \ldots, 0.15\right]$$

Finally, by linking the simplified points data tree to the Base (B) input of Sphere, and the above list of radii to the Radius (R) input, we match the ten tree branches to

the list of ten radii, therefore assigning one radius per branch. As a result, the array of spheres in (7) are generated. You can tell since the radii decrease from left to right, with each vertical column of spheres sharing the same radius; the tree branches are vertical columns of points going from left to right.

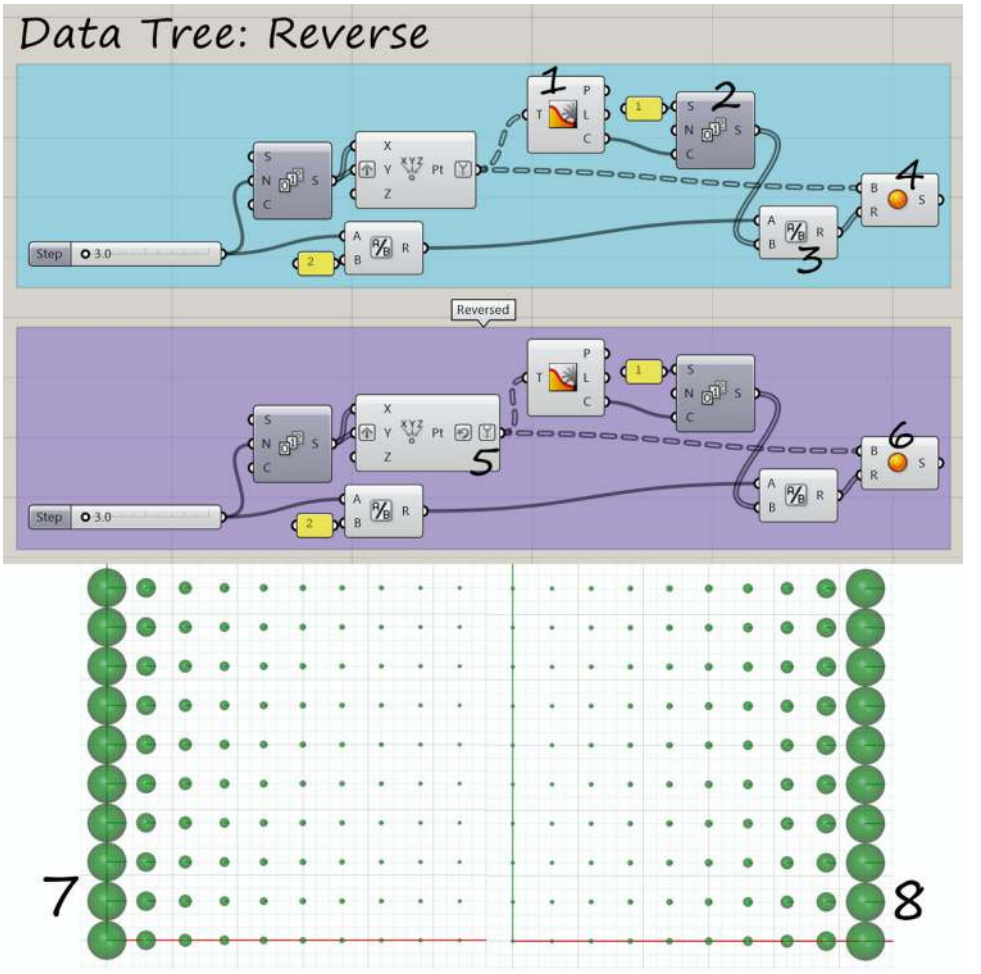


Figure 2-18. *We'll use a simplified version of the data tree from Figure 2-16 (second example) to see the effect of reversing a data tree. First, we'll use (1) Tree Statistics to get the number of branches to use as the count for a (2) Series node starting at 1. We then use the (2) Series output to (3) inversely scale the radii for (4) Spheres centered at those 100 points (7). If we (5) right-click ➤ Reverse the Pt output of the data tree, the (8) Sphere radii will increase from left to right instead of decrease*

Let's do something fun—if we reverse the order of the branches in the points data tree, by right-clicking Construct Point's output ➤ Reverse (5 in Figure 2-18), we get the Sphere radii to *increase* from left to right (8). Reverse simply makes the order of the tree branches backward, so the first branch becomes the last, the second becomes the second to last, and so on. You can reverse any input or output with right-clicking ➤ Reverse, which works on both lists and data trees.

Using the Python 3 Script Node to Define a Custom Component

Grasshopper provides several "programming" nodes that allow you to create custom components written in C#, Iron Python 2, Python 3, and VB.NET. In this section, we'll implement a simple Python 3 Script node to learn the basics of creating inputs and providing type hints, accessing Rhino APIs, and saving the script node as a User Object to be reused.

Writing Python vs. Using Existing GH Components

First, let's think about the context in which to use Python 3 over existing GH nodes. After all, if you are so inclined, you could use nothing but a single Python 3 Script node to define an entire Grasshopper definition! That would obviously be missing the point.

There are two scenarios in which I opt for using Python 3. The first is when no existing GH component (or combination of components) would do what I want, so I have no choice but to write my own—you'll see an example of this in Chapter 11: L-System, where we implement a turtle graphics style interpreter in Python 3. The second situation is when it would take an inordinate number of off-the-shelf nodes or such complex wiring to accomplish a task that I'm better off writing a few lines of Python code—you'll see examples of this throughout the book, often in the case where math formulas are used in the midst of building geometry.

Writing Your First Python 3 Script

Let's work through a simple example. We'll deploy a Python 3 Script node onto the canvas, then write a few lines of Python based on the following formula for a seashell curve (von Seggern, 2007; Weisstein, 2025):

$$x = \left(\left(1 - \frac{v}{2\pi}\right)\left(1 + \cos(u)\right) + c\right) \cdot \cos(n \cdot v)$$

$$y = \left(\left(1 - \frac{v}{2\pi}\right)\left(1 + \cos(u)\right) + c\right) \cdot \sin(n \cdot v)$$

$$z = b \cdot \frac{v}{2\pi} + a \cdot \sin(u) \cdot \left(1 - \frac{v}{2\pi}\right)$$

I won't get into the details of the math here as it's not important for our purpose.

We'll convert the parameters n, c, u, and v from the above formula into inputs for the Python 3 Script node and use a fixed value of b = 10. In addition, we'll create an input named *scale* to set the value of a, as well as to scale x and y, as follows:

$$x = scale \cdot \left(\left(1 - \frac{v}{2\pi}\right)\left(1 + \cos(u)\right) + c\right) \cdot \cos(n \cdot v)$$

$$y = scale \cdot \left(\left(1 - \frac{v}{2\pi}\right)\left(1 + \cos(u)\right) + c\right) \cdot \sin(n \cdot v)$$

In other words, we'll multiply x and y from the original formula by *scale*. I found through experimentation that this produces a nicer looking curve.

Figure 2-19 shows the finished (1) Python 3 Script node with three Number Sliders controlling its scale, n, and c inputs, plus two Range components controlling its u and v inputs. For u, we divide the domain from 0 to Pi into 100 segments (therefore 101 points), and for v, we divide –2Pi to 2Pi into the same number of segments. The Python 3 Script node will output 101 points based on the input values applied to the above formula. If you then link the output points to (2) Interpolate Curve, you'll get the spiral curve shown in Figure 2-20.

If you double-click the center of the (1) Python 3 Script node, the code editor window will open, as shown on the bottom of Figure 2-19. You type the Python code in the (3) text area and click the (4) play button to run it. Any error messages or outputs from Python print statements will appear in the (5) Terminal area below.

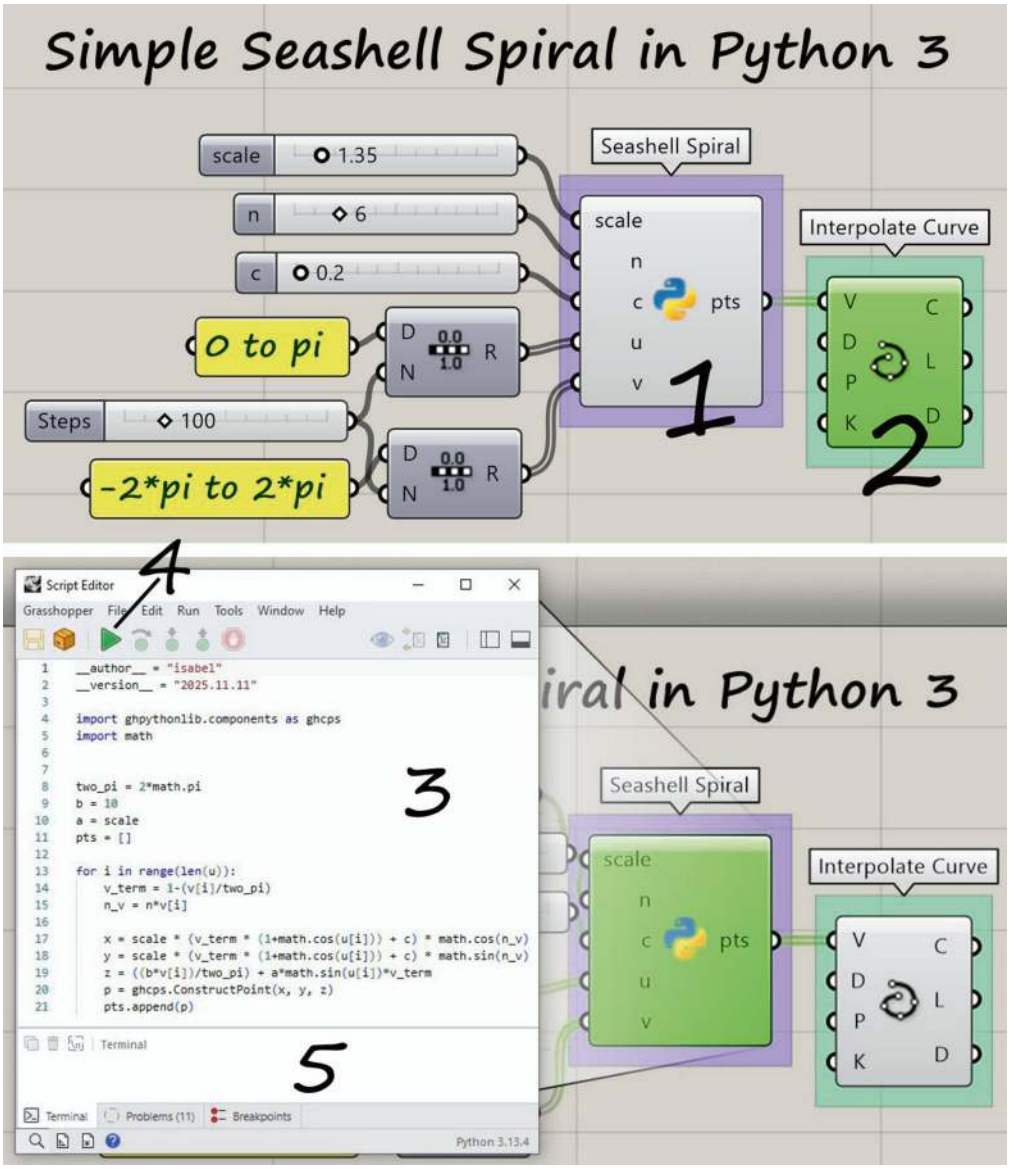


Figure 2-19. *Generating a parametric seashell curve using a (1) Python 3 Script component, which outputs a list of points based on the von Seggern 2007 formula, that we subsequently (2) interpolate into a curve. Double-clicking the center of the (1) Python 3 Script node opens its editor—you type in code in the (3) text editing area, then press the (4) play button to run. The results of print statements and any error messages from the script run will appear in the (5) terminal below*

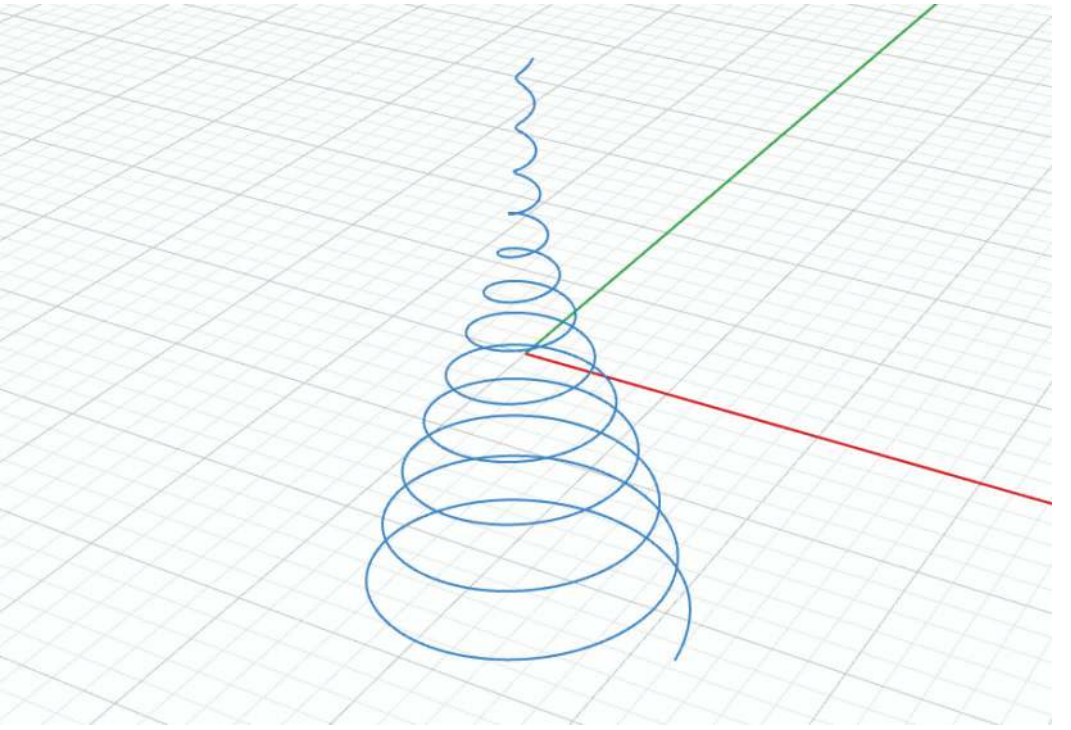


Figure 2-20. *The seashell curve produced by the node graph in Figure 2-19*

The code inside the Python 3 Script node is reproduced in Listing 2-1. We start by specifying the name of the author (`__author__`) and version number (`__version__`), which is typically a string denoting the date the script is written. On the next line, we import the `ghpythonlib.components` module as the alias `ghcps`, which lets us call virtually every Grasshopper component as a Python function. After that, we import Python's built-in `math` module, which gives us access to `Pi` and the trig functions (`sin`, `cos`) needed to implement the spiral formula.

In the next block of code, we compute 2*Pi and save it in a variable (`two_pi`) so it can be reused everywhere it appears in the formula. We set `b` to 10, `a` to the input value `scale`, and create a list `pts` for accumulating the output points.

In the final block of code, we iterate through the 101 pairs of (u, v) values in a `for` loop. In each iteration, we call `ghcps.ConstructPoint(x, y, z)`, which is equivalent to the Construct Point component, to create a point `p` from the `x`, `y`, and `z` values computed for the current pair of (u, v) based on the seashell formula. We then add the point to the end of the `pts` list via `pts.append(p)`.

Listing 2-1. Content of the Python 3 Script node to generate 3D points based on the seashell curve formula of von Seggern 2007 (see `/Ch2/Figure2-19_2-20_python3_spiral.gh`)

```
__author__ = "isabel"
__version__ = "2025.11.11"

import ghpythonlib.components as ghcps
import math

two_pi = 2*math.pi
b = 10
a = scale
pts = []

for i in range(len(u)):
    v_term = 1-(v[i]/two_pi)
    n_v = n*v[i]
    x = scale*(v_term*(1 + math.cos(u[i])) + c)*math.cos(n_v)
    y = scale*(v_term*(1 + math.cos(u[i])) + c)*math.sin(n_v)
    z = ((b*v[i])/two_pi) + a*math.sin(u[i])*v_term
    p = ghcps.ConstructPoint(x, y, z)
    pts.append(p)
```

Adding Inputs and Outputs

With the code implemented, next we'll look at how to customize the inputs and outputs for the Python 3 Script node. Keep zooming in on the Python 3 Script node by scrolling the mouse wheel, until + and – buttons appear, as shown in Figure 2-21. You can press the – button to the right of an input (or left of an output) to delete it or the + button between two inputs (or outputs) to insert a new port there. You can also right-click an input (or output)'s name to change it. The names and types of the inputs and outputs must match their counterparts in the Python code in both spelling and capitalization. We'll look at how to specify the data types for inputs and outputs in the next section.

When a Python 3 Script node is first deployed to the canvas, it has two default outputs named "out" and "a". Also known as "standard output," "out" is the content of the Terminal at the bottom of the Script node's code editor, which includes the results of any print statements in the Python code and error messages from the script execution.

Note Recall that you can open a Python 3 Script node's code editor by double-clicking its center.

The output "a" is of type "ghdoc" object, which is a generic object type in Grasshopper. By default, "a" is designated as the output for geometry produced by the Python code. You will usually change "a" to a more descriptive name and edit its access and type hints (discussed next) to match the content of the script.

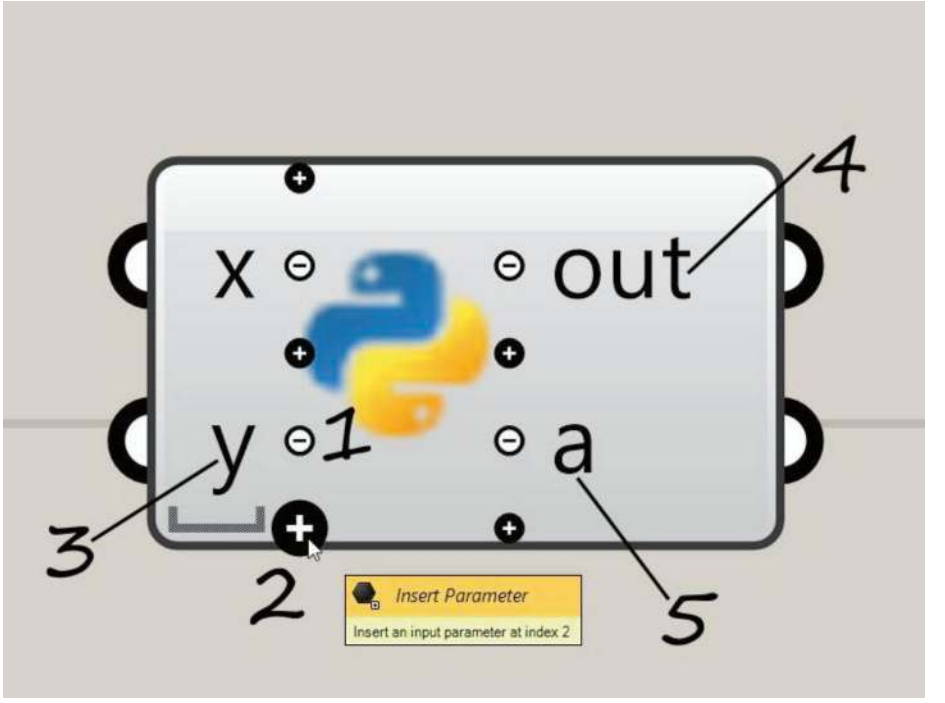


Figure 2-21. *When you first place a Python 3 Script node on the canvas, it has the default inputs and outputs as shown. You'll (1) delete or (2) add inputs and outputs as needed, (3) give them more descriptive names, and edit their access and type hits (discussed in the next section). The two default outputs are (4) "out," which is where the standard output is directed, and (5) "a," where the script's output geometry (as a ghdoc object) goes*

Access and Type Hints

Once you've created and renamed the inputs and outputs for the Python 3 Script node, the next step is to specify the appropriate *Access type*, which tells Grasshopper about the *structure* of the data to expect through that input or output port. To do so, right-click an input or output and select either *Item Access*, *List Access*, or *Data Tree Access*, which correspond to a single value, a linear sequence of values, or a nested list.

In addition to the structure, you'll also inform Grasshopper of the type of the data expected at an input or output, by selecting one of the types under right-clicking ➤ Type Hints.

For example, the u input for our spiral curve script node in Figure 2-19 expects a list of decimal numbers; therefore, we choose "List Access" and Type Hints ➤ float (3, 4, and 5 in Figure 2-22).

Saving a Custom Script As a User Object

You can save the Python 3 Script node from Figure 2-19 as a "User Object," which will behave like a user-defined component installed locally to your copy of Grasshopper. Click to select the Python 3 Script node in Figure 2-19, then go to File ➤ Create User Object... (1 in Figure 2-23), which will open the User Object Properties window, where you can enter a (2) name for the node and a (3) description (which will appear in the tooltip when you hover over the node with the mouse). You'll also select a (4) category, subcategory, and an (5) icon for the node to customize how it will appear in the GH editor menus. For example, I entered "Seashell Spiral" for the node's name, Curve ➤ Spline as its category and subcategory, and the Python logo as its icon; therefore, it will appear in the Curve tab under the Spline panel (6 in Figure 2-23). From here on, you can use the node like any other GH component. It will also appear in the search box.

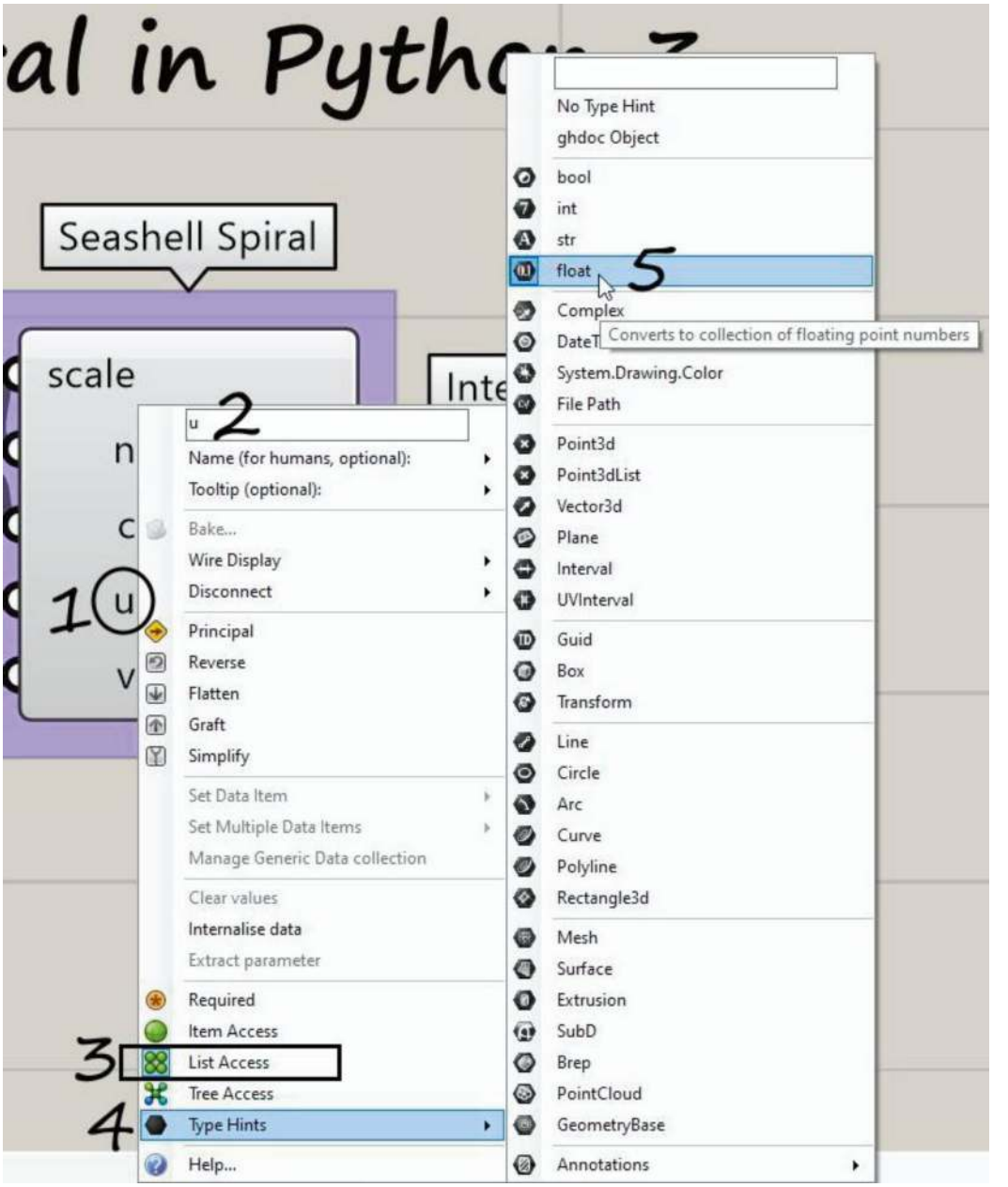


Figure 2-22. *You can (1) right-click an input and (2) customize its name, specify (3) its type of "access" (i.e., single item, list, or data tree), and denote its (4) data type. As shown, the u input for our spiral curve script node expects a list of decimal numbers; therefore, we set it to (3) List Access and (4) Type Hints ➤ (5) float*

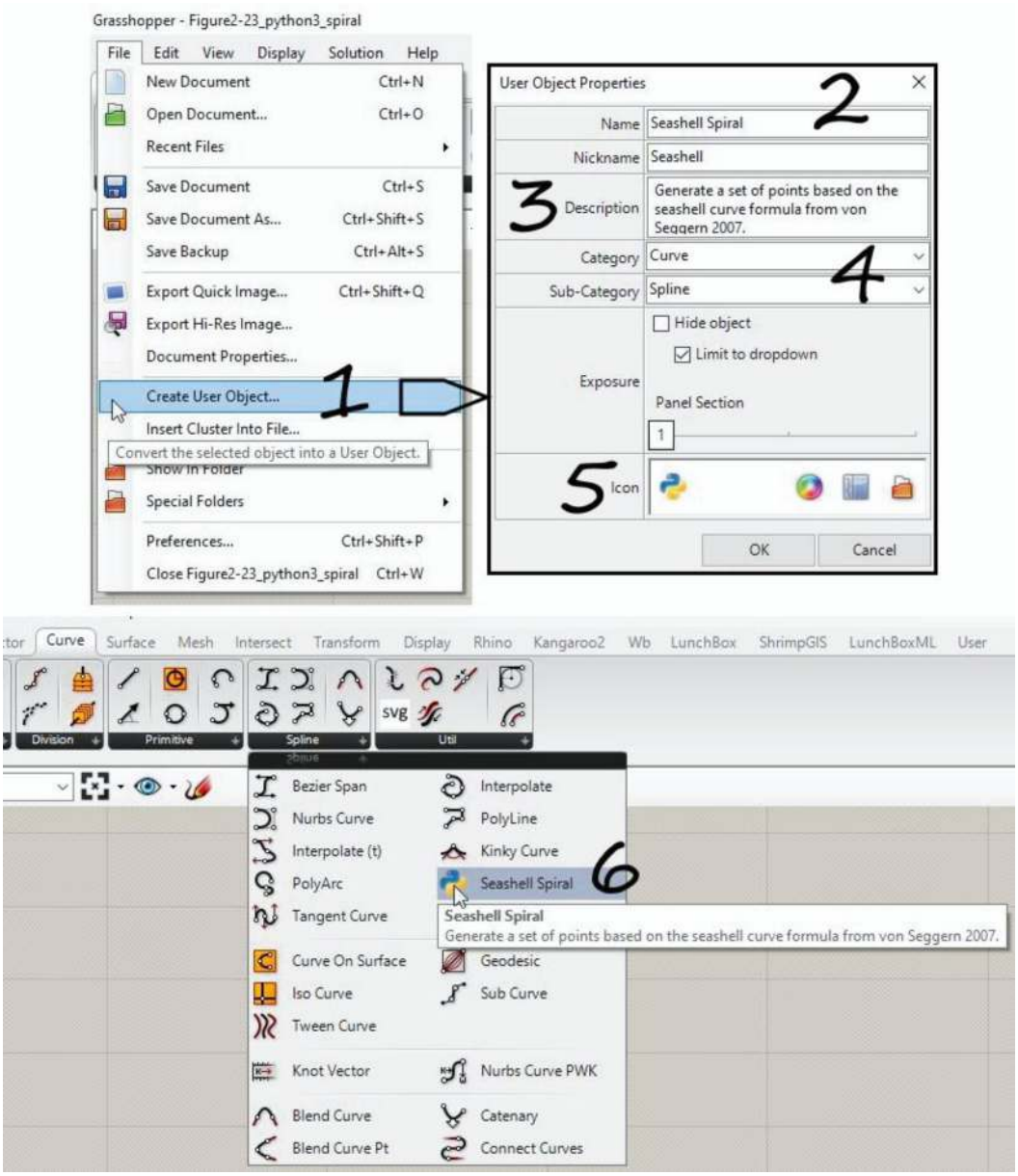


Figure 2-23. *With the Python 3 Script node selected, go to (1) File ➤ Create User Object..., then in the window that opens, enter a (2) name and (3) description, select the (4) category and subcategory (as shown, Curve ➤ Spline), and choose an (5) icon. Click OK. You'll see that the Python 3 Script node has been saved as a (6) reusable component in the designated category/subcategory*

Organizing Grasshopper Definitions

In this section, we'll look at a few techniques for organizing and annotating Grasshopper definitions so they are easier to understand. We'll start by aligning and distributing components to declutter the canvas, followed by arranging components into groups, and finally using Scribble and Panel to add comments to the graph.

Aligning and Distributing Components

In Grasshopper, the relative positioning and spacing between components do not alter the behavior of the script. However, if a graph is messy, it's hard to tell where the inputs are and which part(s) of the graph they influence, making it difficult for both the developer and the user to understand. Hence, it's always a good idea to keep a definition clean, with minimal overlap of components and wires and proper spacing between them.

Grasshopper provides built-in widgets for aligning components and distributing them (evening out their spacing). Once you've selected more than one component, a box enclosing the selection will appear, with a group of widgets on each edge of the box, as shown in Figure 2-24. Each group consists of four buttons—one button with horizontal bars will *distribute* objects in the orthogonal direction (the button at the top/bottom edge of the box will distribute vertically, whereas the button on the left/right edge will distribute horizontally). The other three buttons with arrows will *align* objects in the directions the arrows point. On the top and bottom edges of the box, the three buttons will align objects on their left edges, vertically, and on their right edges, as shown in Figure 2-24. On the left and right edges of the box, the three buttons will align objects on their top edges, horizontally, and on their bottom edges.

Figure 2-25 shows the effect of distributing a group of three components vertically. In this example, the component in the middle is moved downward so the pairwise spacing of the three components is evened out.

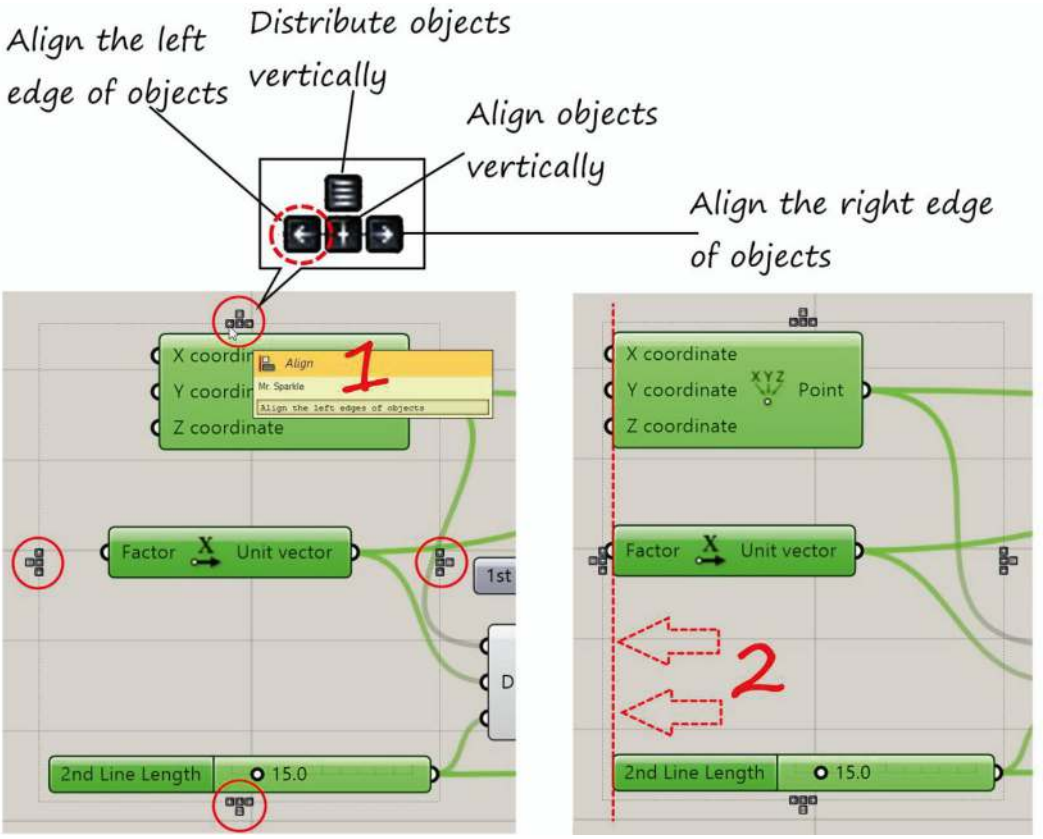


Figure 2-24. *As soon as more than one component are selected, groups of widgets (circled) for aligning and distributing components will appear around the box enclosing those components. You can hover over each button to see what it does. As shown, we click the (1) button to (2) align the selection to its left edge*

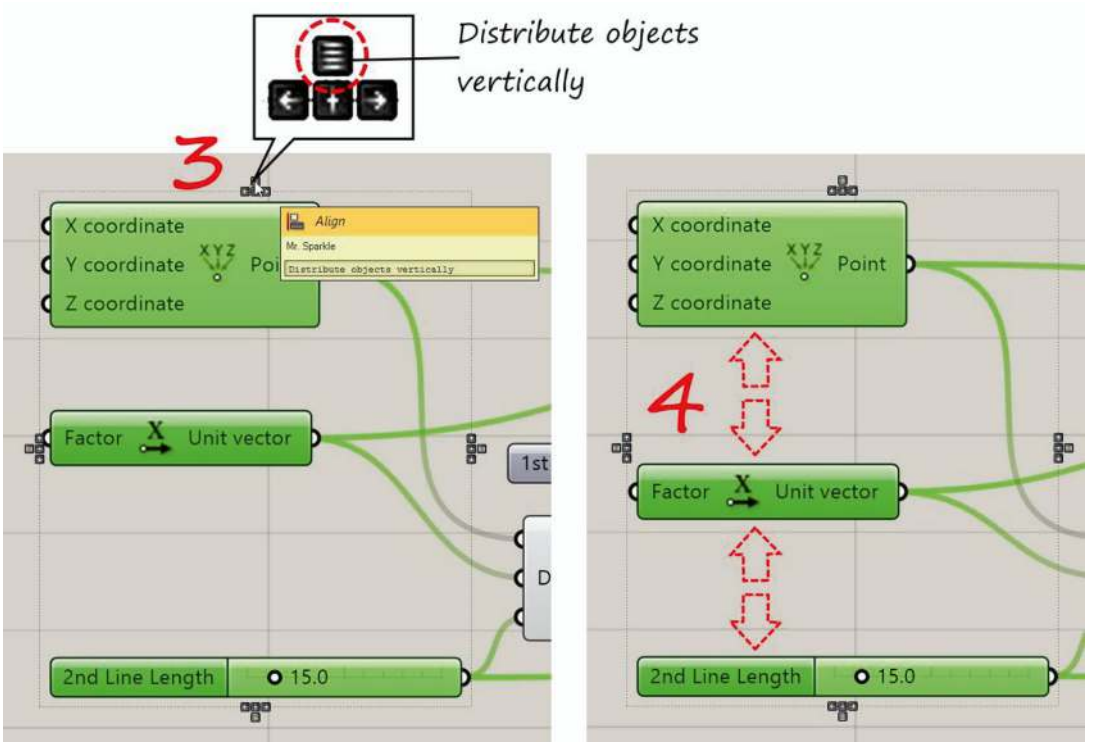


Figure 2-25. *Distributing selected components so they are evenly spaced. As shown, we click the (3) button to "Distribute objects vertically," which (4) moves them up and down to even out their pairwise spacing*

Grouping Components

As your definition becomes more complex, it makes sense to divide the node graph into groups, with each group representing one stage of model generation. You can select components to go in a group by holding down shift and clicking components one by one or holding down LMB while dragging to draw a box around components you want to select and let go to confirm. The selected components will turn green (1 in Figure 2-26). To group the current selection, right-click anywhere on the canvas ➤ Group or press Ctrl-G. A Box outline will appear by default around the components with a color background (2 in Figure 2-26).

You can click anywhere within the group background to select the group or right-click to invoke the group context menu, where you can change the shape of the outline or the color of the background (3 in Figure 2-26). You can also enter a name for the group, which acts like a description for what the group does (3). You can move the group as a whole by positioning the cursor inside the group background, then holding down LMB and drag.

To add (or remove) component(s) to an existing group, select the component(s) to add (remove), select the group by Shift-clicking anywhere within the group background, then right-click ➤ Add to (Remove from) group. You can quickly select all components within a group via right-clicking ➤ Select all. To remove a group, select the group then press the Delete key or right-click ➤ Ungroup.

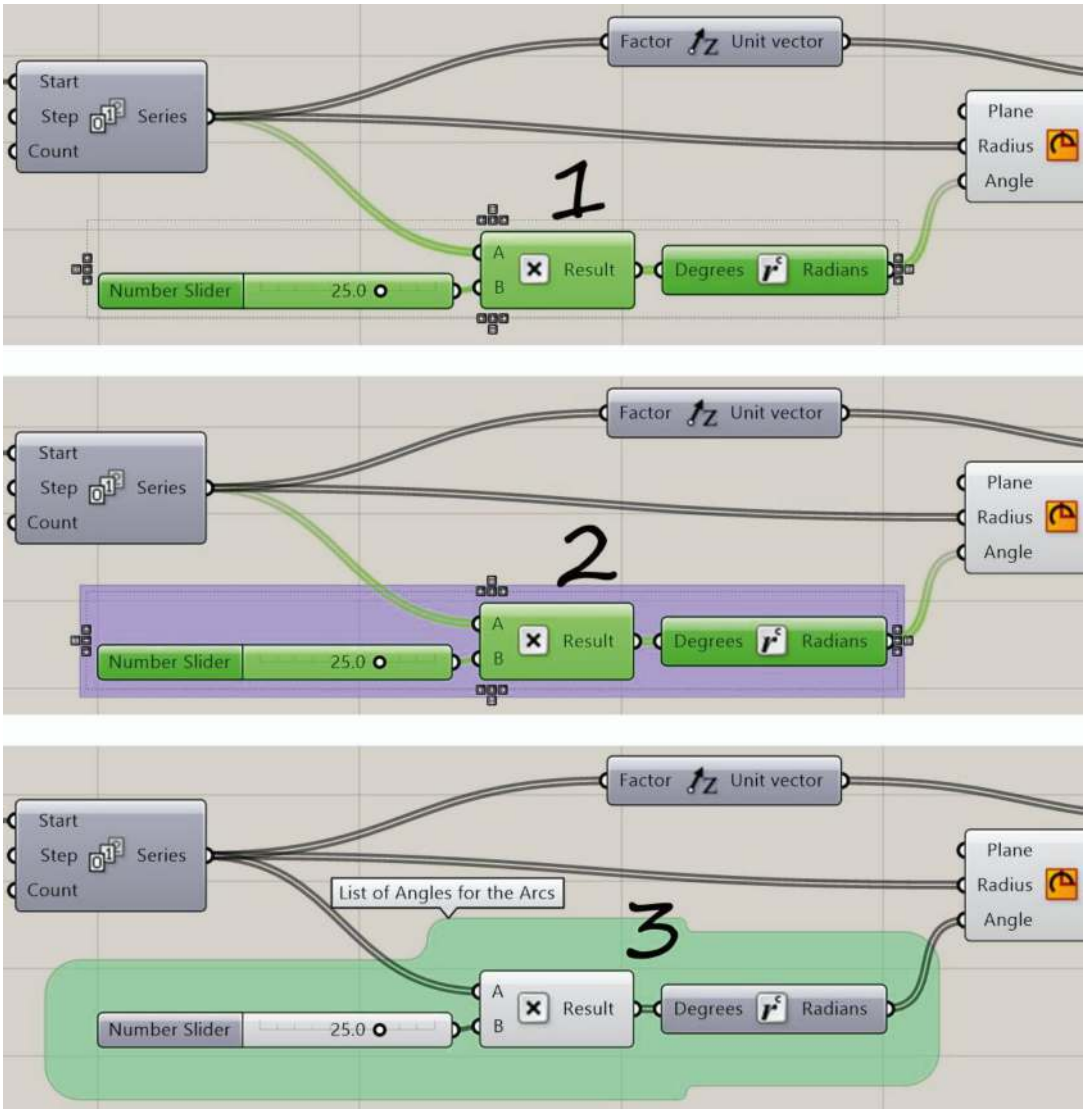


Figure 2-26. *Grouping components. (1) Shift-click or click-drag a box to select the components you intend to group, then (2) Ctrl-G or right-click ➤ Group to group them. (3) Right-click anywhere inside the group background to invoke the context menu, where you can change the group outline, color, and enter a group name*

Commenting with Scribbles and Panels

As you work through the sample files for the book, you'll notice that I like to use the Scribble node to label GH definitions and leave standalone Panel nodes as sticky notes. Params ➤ Util ➤ Scribble (1 in Figure 2-27) is a 2D text object that you can place anywhere on the canvas, which can be added to a group like any other component. You can double-click a Scribble to change its font, style, and size. Figure 2-27 shows an edited version of the node graph from the Range example (Figure 2-2, with Display ➤ "Draw Icons" on and "Draw Full Names" off). Here, a (1) Scribble node is used as the title for the node graph and a (2) Panel as a sticky note containing a description. As shown, the Panel's font and color have been changed via right-clicking ➤ Font Settings ➤ Custom Font... and right-clicking ➤ Colour, respectively.

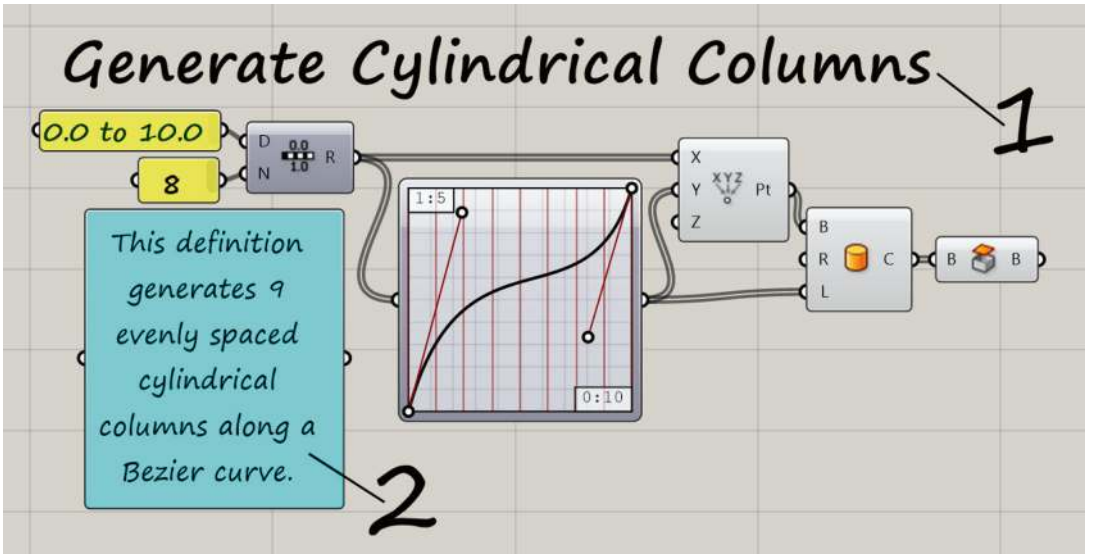


Figure 2-27. *You can use a (1) Scribble node to label a GH definition and a standalone (2) Panel as a sticky note*

Baking: Turning Grasshopper Preview into Rhino Geometry

Geometry generated by Grasshopper stays in preview form until it is "baked" and converted into real geometry. To bake a component's output, right-click the component's name ➤ Bake to open the Attributes window, where you'll select a layer as

bake destination, choose the Display mode (e.g., wireframe) for the baked geometry, and decide whether to Group the bake outcome, then click OK, as shown in Figure 2-28. After baking, the geometry becomes Rhino data and is no longer under GH control.

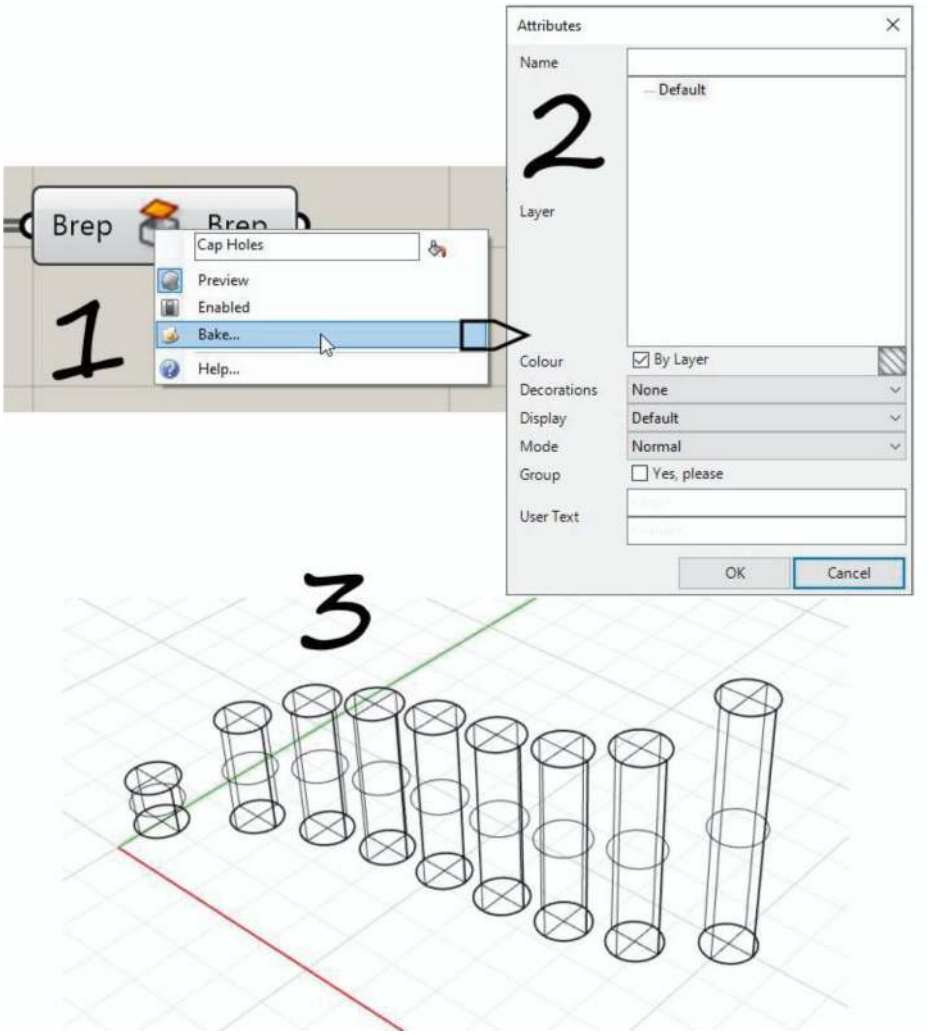


Figure 2-28. *To convert GH preview into real geometry, (1) right-click the component's name ➤ Bake to open the (2) Attributes window, where you'll configure baking options, and click OK. As shown, the Range example from Figure 2-2 is baked and displayed as a (3) wireframe in the Rhino viewport. The baked geometry is detached from GH control*

Summary

In this chapter, we delved into more advanced scripting topics, starting with data conversion, the process with which Grasshopper converts data between outputs and inputs of compatible but non-identical types, for example, when a point is linked to an input expecting a plane, a plane centering at that point parallel to the XY plane is automatically created and used as input.

In the second part of the chapter, we discussed various strategies for working with *lists*—linear sequences of data in Grasshopper. We created lists with the Range node and Series node, filtered lists with Cull Nth and Cull Pattern, extracted partial lists with Partition List and Sub List, split a list into streams with Dispatch using an input pattern, and finally, recombined streams into a list with Weave.

The third topic we went over was Data Matching, the process with which Grasshopper handles input lists of different lengths. By default, with a rule called "Longest List–Repeat Last," two lists are matched one-to-one until the shorter one runs out, then the leftover of the longer list is matched repeatedly with the last item of the shorter list. GH provides options for Longest List to align different portions of the shorter list to the longer one. There are also completely different matching methods called Shortest List and Cross Reference, which we also discussed.

In the fourth part of the chapter, we learned about another essential data structure—the data tree, which are nested lists in Grasshopper. You learned how to create, access, and edit tree branches, as well as utilize the tree structure to get components to behave in different ways.

So far, we've focused on building node graphs with existing GH components. In the fifth part of the chapter, we looked at how to edit a Python 3 Script node to generate parametric seashell curves and save the node into a User Object, which behaves like a reusable custom component installed to your local copy of Grasshopper.

The sixth topic we went over was how to organize objects in a GH definition, such as aligning, distributing, and grouping components, as well as creating comments using Scribble and Panel nodes. Finally, we discussed how to turn GH preview data into real Rhino geometry, which then becomes detached from GH and can be edited in Rhino.

References

von Seggern, D. *CRC Standard Curves and Surfaces with Mathematica, 2nd ed.* Boca Raton, FL: CRC Press, 2007.

Weisstein, Eric W. "Seashell." From *MathWorld*—A Wolfram Resource. https://mathworld.wolfram.com/Seashell.html

CHAPTER 3

Attractors

In this chapter, we'll work through a series of exercises centered around the theme of *attractors*—arrays of objects transformed based on their proximities to one or more objects called *emitters*. We'll start by building a 1D attractor with one row of geometry and a single emitter point and create variations by scaling array items in different ways and adding levitation. We'll then generalize the concepts to 2D rectangular grids and extrude to add a third dimension. We'll also experiment with squeezing and expanding the grids with vector force fields. In addition to rectangular grids, we'll discuss how to create polar grid-based attractors with the help of Python 3. The chapter will conclude with exercises on creating emitter curves parametrically and using them to transform various types of arrays.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch3` folder, where you'll find `*.gh` files named after the figures in which they appear, along with a short description. For example, `Figure3-2_1D_array.gh`. The sample files have been tested in Rhino 9 on Windows. As you read the chapter, you can build your own node graphs by following the steps described in each section or start with the downloaded `*.gh` files and experiment by modifying their parameters.

What Is an Attractor?

An *attractor* is a grid of objects that transform proportionally based on their distances to the *emitter(s)*, which are usually points but could be lines or curves.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_3

For example, Figure 3-1 shows a grid of hex cells on the left, which we turn into the attractor shown in the center by scaling the cells such that the closer a cell is to the emitter (marked by X) the bigger it is, in other words, the cells are scaled inversely proportionally based on their distances to the emitter, which creates the illusion of the cells being pulled toward the emitter—hence the name "attractor."

We can create a *repeller* just as easily by scaling the cells *proportionally* based on their proximity to the emitter, as shown on the right of Figure 3-1, where the *farther* a cell is from the emitter the bigger, creating the impression that cells are being *pushed away* from the emitter, which is the opposite of an attractor.

Note I like to use the terms "attractor" and "repeller" to differentiate between whether objects appear to be pulled toward or pushed away from the emitter(s). However, in practice, you'll often hear the word "attractor" used to describe both.

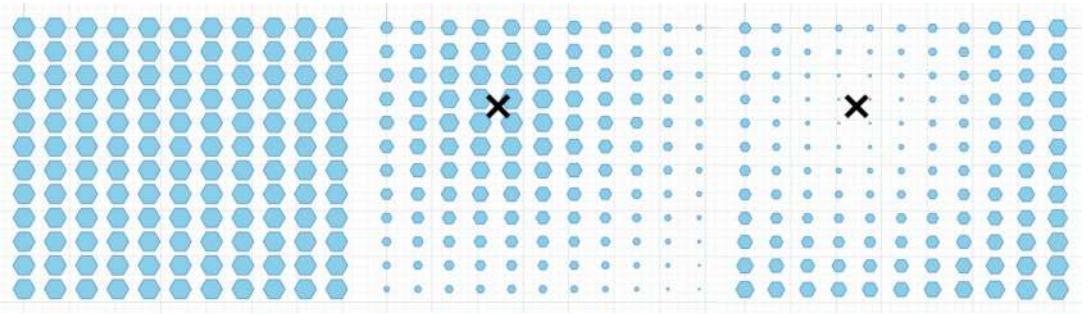


Figure 3-1. *Left: The original hex grid. Center: "Attractor" in which the cells are scaled inversely proportionally by their distances to the "emitter" (marked by X), such that the closer a cell is to the emitter the bigger. Right: "Repeller" in which the farther a cell is to the emitter, the larger it is*

1D Attractors

We'll start by creating a basic 1D attractor, which is a sequence of objects flowing along a line or curve with an emitter point, by arranging hexagonal curves along the X axis and scaling them, so the closer a hexagon is to the emitter the larger it is. After that, we'll create a repeller by inversing the scaling and use the Stream Filter component to create a switch for toggling between attraction and repulsion. Finally, we'll add levitation along the Y axis that is also influenced by the emitter force.

Creating a Linear Array with Custom Spacing

We'll create a 1D array of polygonal curves along the X axis, by linking a (1) Curve ➤ Primitive ➤ Polygon node to the Geometry (G) input of a (2) Transform ➤ Array ➤ Linear Array node as shown in Figure 3-2. You control the direction and spacing of how Linear Array repeats the given geometry with the Direction (D) input—in this case, a (3) Unit X vector with a Factor of 3, to repeat the input curve 3-units center-to-center along X. (4). You can vary the number of repetitions by linking a Number Slider to the Count (N) input, which is dialed to 5 here to create five hexagons.

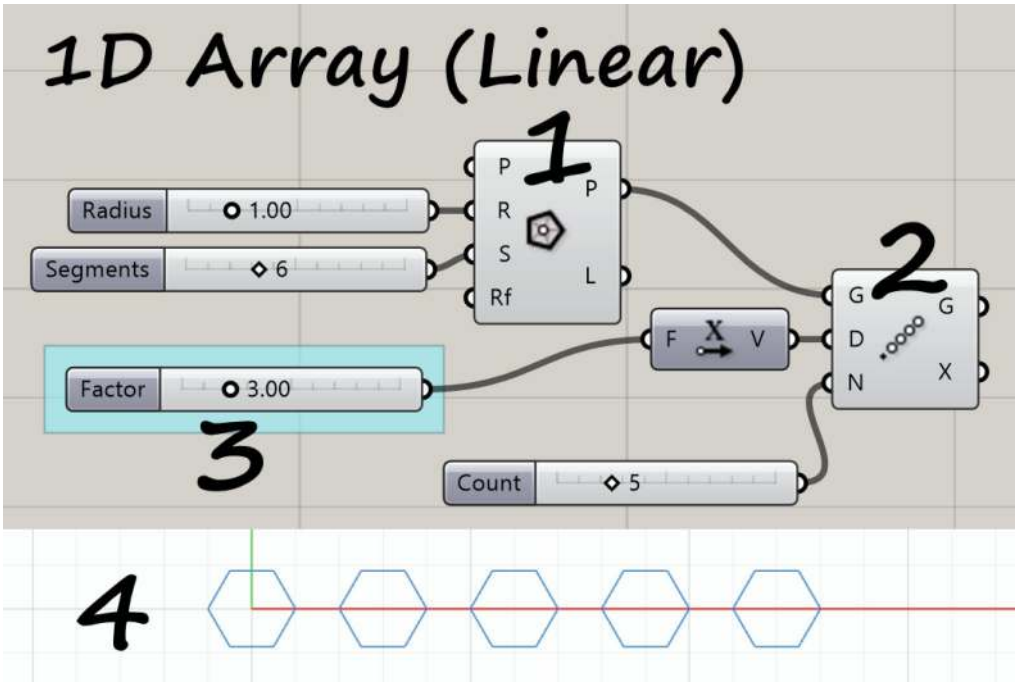


Figure 3-2. *Creating a 1D array (or linear sequence) of objects, by linking a (1) Curve ➤ Primitive ➤ Polygon node to a (2) Transform ➤ Array ➤ Linear Array node with a Count (N) of 5 and a (3) vector as direction and spacing (as shown, 3*Unit X), which produces the (4) sequence of hex curves below*

On the surface, it may seem our node graph is parametric; however, there is one problem—since the (3) Factor to Unit X, which dials the distance between adjacent curve centers, is controlled *independently* from the Radius (R) of the (1) Polygon node, every time you increase the curve radius, you have to calculate how much to adjust the array spacing to prevent adjacent curves from overlapping. In Figure 3-3, we modify the node graph so the spacing automatically adapts to the curve radius. First, we replace the Factor slider (3 in Figure 3-2) with the nodes marked (1) to (4) in Figure 3-3, which use the curve's bounding box to derive the exact distance for positioning neighboring curves so they are tangent (or with touching vertices).

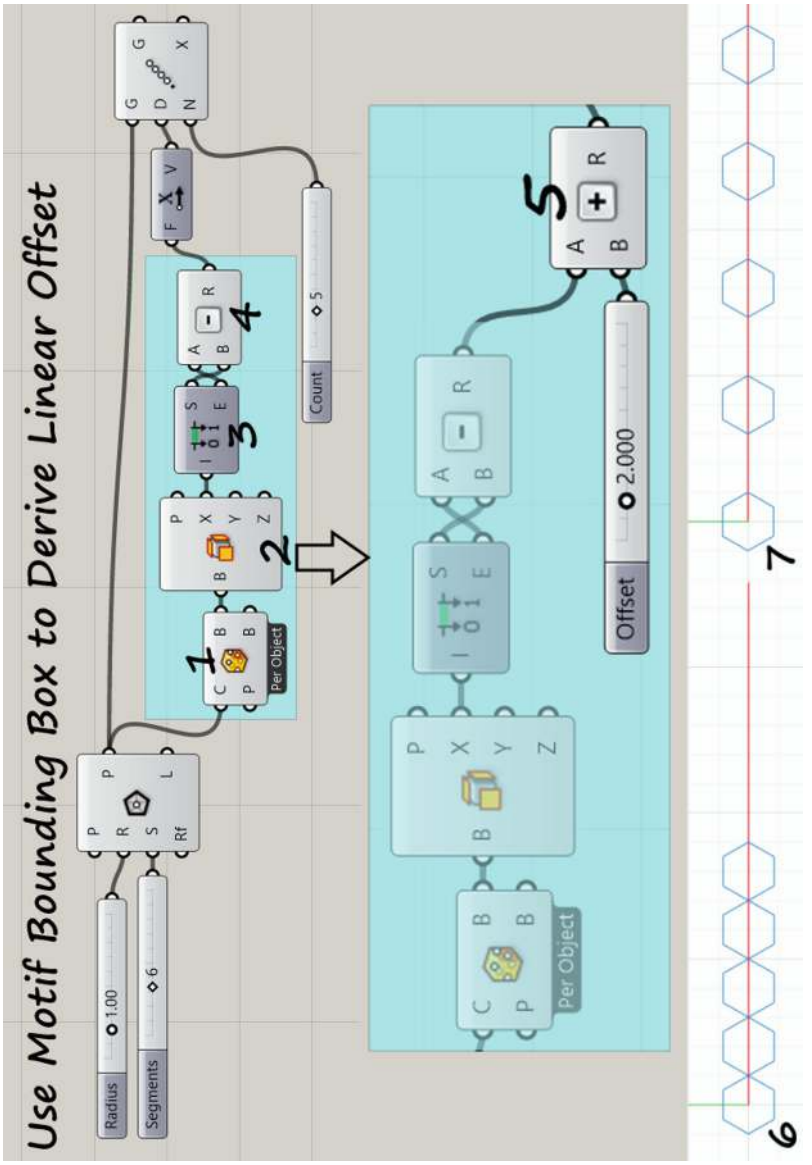


Figure 3-3. *Continuing from Figure 3-2, we can find the X span of each motif and use it to scale the Direction input of Linear Array to arrange the motifs (6) side-by-side without overlap. To do so, we find the (1) Bounding Box of each polygon, then use (2) Deconstruct Box to find the box's X dimension, (3) Deconstruct Domain to find the dimension's Start and End, and (4) Subtract to get the difference between the two, which is the X span. We can (5) add an optional offset pairwise between motifs—as shown, we add 2 to the (4) X span, which gives us (7), where the distance = 2 from motif border to border*

Let's go over Figure 3-3 step by step: We link Polygon to a (1) Surface ➤ Primitive ➤ Bounding Box node, which gives us the minimal rectangular box that will enclose the entirety of the curve, then a (2) Surface ➤ Analysis ➤ Deconstruct Box node to extract the dimension of the box per axis (X, Y, and Z) as a domain. However, we need the width of the domain as opposed to the domain itself to use as the scaling factor for the array spacing vector. For example, if the domain is 2 to 5, 5-2=3 gives us the diameter of the curve along X, which when used as the distance between adjacent curve centers (i.e., array spacing) makes neighboring curves tangent. To calculate the width of the box's X domain, we link it to a (3) Math ➤ Domain ➤ Deconstruct Domain node, then (4) subtract the start of the domain from its end using a (4) Maths ➤ Operators ➤ Subtract node. We then link the result of the subtraction to the Factor (F) input of Unit X, which scales the Direction (D) vector for Linear Array to exactly the width of the curve along X, thus producing the array shown in (6), with hexagons touching.

Now that the array spacing auto adjusts to the curve radius, we can modify the node graph in Figure 3-3 to add the option for specifying gaps between curves. We connect (4) and a Number Slider to the two inputs of a (5) Maths ➤ Operators ➤ Add node, then by dragging the slider, we add an offset *on top of* the diameter of the curve as array spacing. For example, in Figure 3-3, the slider is set to 2, which makes the hexagons 2 units apart from border to border (7).

1D Attractor

So far, our node graph creates a linear array of polygonal curves with custom spacing that automatically accounts for the polygons' changing radii, which is reproduced as (1) in Figure 3-4, with the only modification being that the Count (N) for Linear Array is increased from 5 to 12. In this section, we'll extend the graph to create an attractor (shown as 2 in Figure 3-4), by creating an emitter point (2b) on the invisible line through the polygon centers, then scale the polygons' radii based on their distances from the emitter.

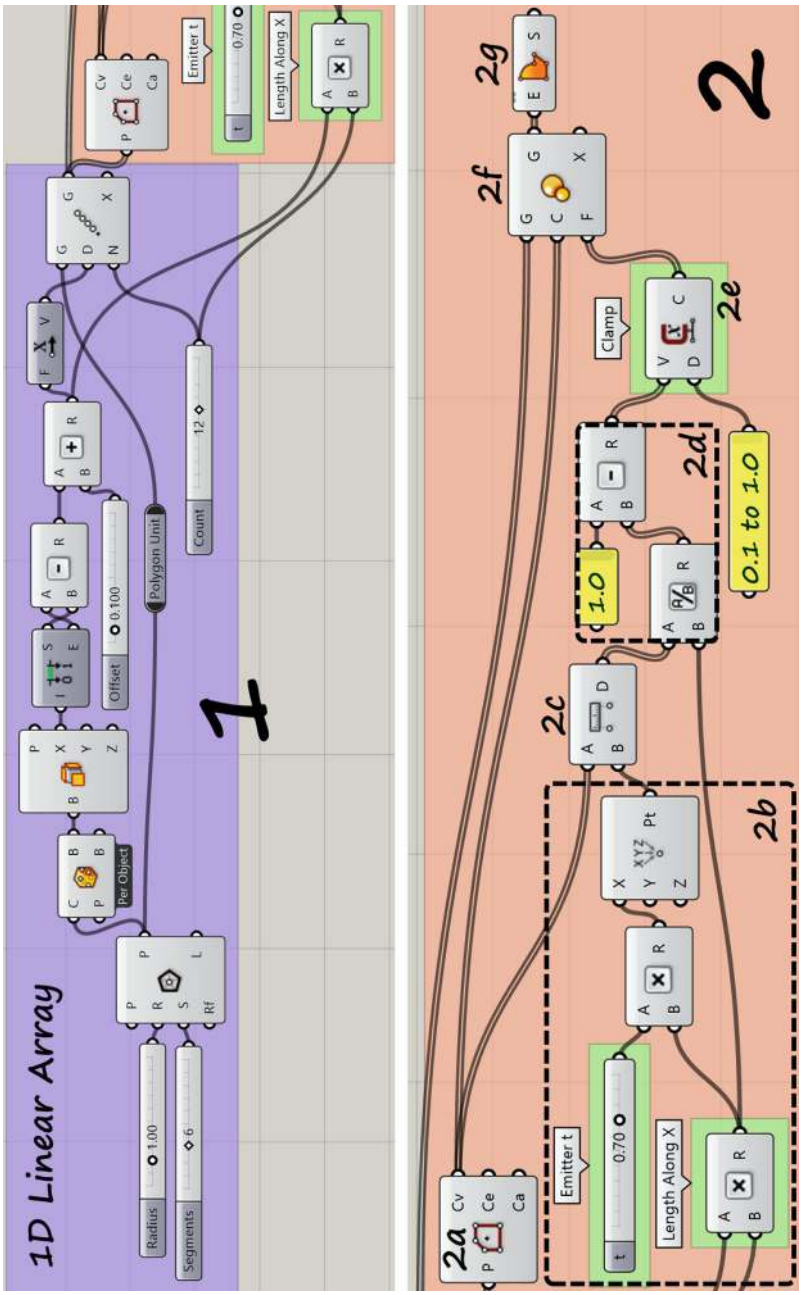


Figure 3-4. *One-dimensional attractor node graph. (1) is a reproduction of Figure 3-3 that creates a linear array of polygon curves along X with custom spacing. (2) creates a parametric emitter point along the line through the array and scales the polygons inverse proportionally based on their distances from the emitter*

Stage 1: Creating an Emitter Point Along the Line Through Polygon Centers

To create an emitter point that slides along the line through polygon centers, we'll first calculate the length of the line by multiplying the Number Slider controlling the Count (N) of Linear Array with the distance between neighboring polygon centers ((5) in Figure 3-3). We then add another slider to control the percentage along the line where we want the emitter to be (i.e., 0.70 means 70% from the beginning of the line), which when multiplied with the line length gives us the X coordinate for the emitter. We create the emitter point using a Construct Point node, as shown in (2b) of Figure 3-4.

Stage 2: Scaling the Polygons Based on Their Distances to the Emitter

Next, we'll compute the distances between the polygons and the emitter point. We find the polygon centers by linking Linear Array's Geometry (G) output to a Curve ➤ Analysis ➤ Polygon Center node (2a). We then find the distances between the curve centers and the emitter by linking Polygon Center (2a) and Construct Point (2b) to Vector ➤ Point ➤ Distance (2c).

To create an attractor, we want to compute scale factors that are inversely proportional to the polygons' distances from the emitter. We divide the distances by their maximum (the line length of the whole array) to convert them into percentages, so they are at most 100%.

Since the polygons should be at their original size closest to the emitter then progressively get smaller as they get further away, we subtract the distance percentage from 1 (e.g., 0.25 becomes 0.75) to use as scale factors (2d). To prevent the scale factors from going to 0, which could happen if a polygon center coincides with the emitter point, we pass the scale factors through a Maths ➤ Util ➤ Clamp node (2e) with a Domain (D) of 0.1 to 1.0 to make them at least 0.1. To scale the polygons, we link Linear Array to a Transform ➤ Affine ➤ Scale node (2f), link Polygon Center (2a) to its Center (C) input so each polygon is scaled relative to its center, and link the inverse scale factors (2e) we've computed to its Factor (F) input. Note that since all three of Scale's (2f) inputs are lists of the same length, GH matches them one-to-one; therefore, each polygon is scaled about its center based on its corresponding scale factor. To finish the attractor, we link Scale (2f) to a Surface ➤ Freeform ➤ Boundary Surface node (2g) to create surfaces with the polygons as their borders.

The top of Figure 3-5 shows the output at the end of (1) in Figure 3-4, which is a linear array of hexagonal curves with radius 1 along the X axis. The bottom shows the output at the end of (2) in Figure 3-4, which are the surfaces scaled by the attractor, with the emitter point marked by an X. You can see that the hexagons' sizes peak at the emitter then decrease as they become further away.

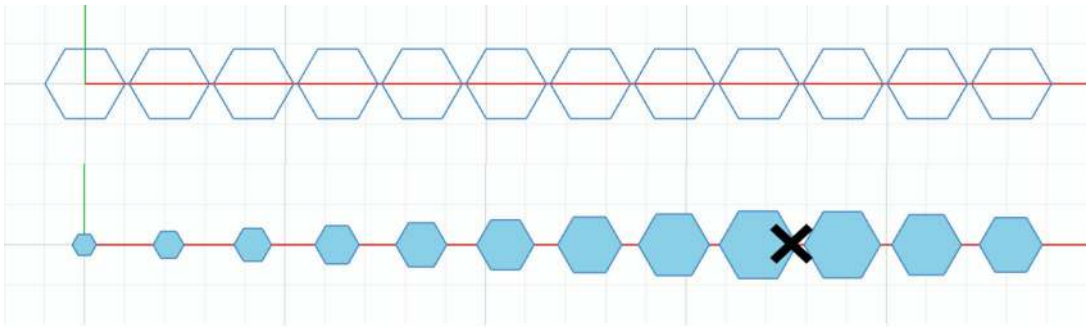


Figure 3-5. *One-dimensional attractor sample results. The top shows the output at the end of (1) in Figure 3-4 and the bottom the output at the end of (2), with the emitter marked as X. Notice that the hex surfaces are scaled such that the further they are from the emitter the smaller they get*

1D Repeller

We can modify Figure 3-4 to make it act like either an attractor or a repeller at the flip of a switch. To do so, we insert the portion marked (3) in Figure 3-6 between (2e) and (2f) in Figure 3-4. Recall that (2d) are the scale factors based on one minus the distance percentages of polygon centers to the emitter point. Here, we modify (2e) slightly so it clamps the scale factors to the range of 0.1 to 0.9 instead of 0.1 to 1.0—we'll see why in a moment.

To make the emitter repel instead of attract, all we need to do is use the *complement* of the scale factors, i.e., one minus (3c), so 0.75 becomes 1 - 0.75 = 0.25, 0.1 becomes 1 - 0.1 = 0.9, and so on. Since 1.0 - 1.0 = 0.0 will create a zero scale factor and cause an error with the Scale node, we clamp the scale factors with (2e) so they are at most 0.9.

To create a switch for toggling between the original scale factors (attractor) and the complement (repeller), we use a Sets ➤ Tree ➤ Stream Filter node (3a), which takes two or more input data streams and selects one to output based on the value of its Gate (G) input.

Here, we connect (2e) to 0, its complement (3c) to 1, and a Number Slider (3b) to G, so when you slide (3b) to 0, the attractor scale factors (2e) pass through, creating the output shown as (3d) in Figure 3-6. When you slide (3b) to 1, the repeller scale factors (3c) pass through, creating the output shown as (3e).

Tip You can zoom in on Stream Filter and use the "+" buttons that appear to add more inputs (they will appear as 2, 3, ..., and so on), to get more than two input streams to select from.

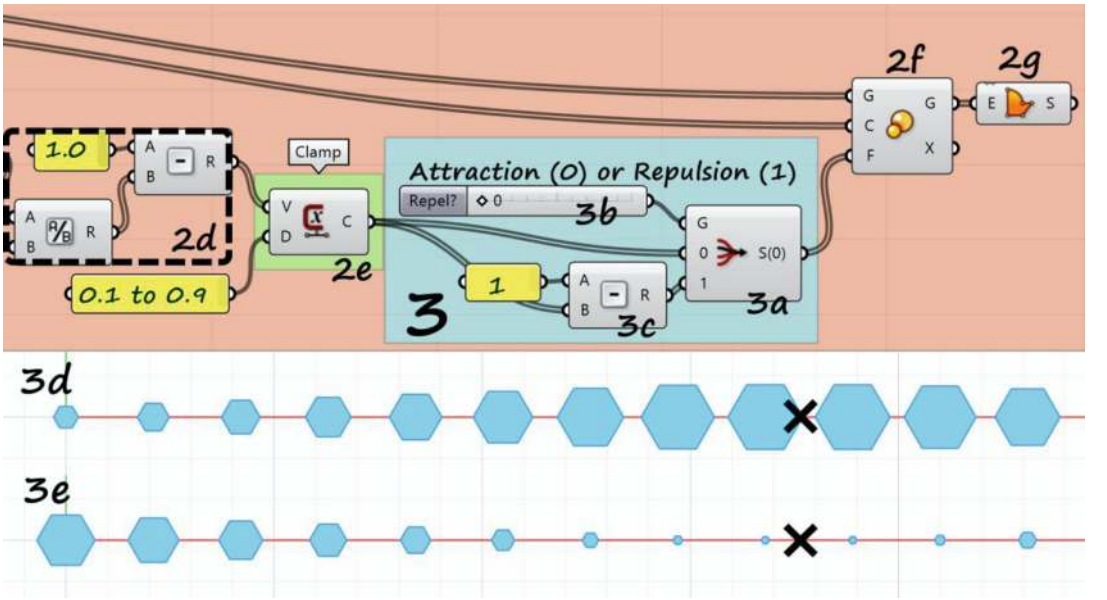


Figure 3-6. *Inserting a (3a) Stream Filter to toggle between the scale factors from Figure 3-4 (2e) to create an attractor (3d) or the complement of the scale factors (3c) to create a repeller (3e). The emitter point is marked by an X in (3d) and (3e)*

Adding Levitation

To make our attractor/repeller more interesting, we'll make the polygons levitate along Y, by calculating Y offsets for the polygons based on the scale factors, as shown by (4) in Figure 3-7. Since the scale factors (3a) are between 0.1 and 0.9, to make the Y offset more prominent, we multiply them by 10 (4c) before using them as Factor (F) for Unit Y (4b) and subsequently as Motion (T) for Transform ➤ Euclidean ➤ Move (4a) to move the polygons. (4d) in Figure 3-7 shows the output of the node graph in "attractor" mode

with the Stream Filter (3a) in Figure 3-6 dialed to 0. Since we're using ten times the scale factors to move the polygons, the closer they are to the emitter, the higher they are. (4e) shows the output in "repeller" mode, which is the opposite of (4d), where polygons are smaller and lower when they are closer to the emitter.

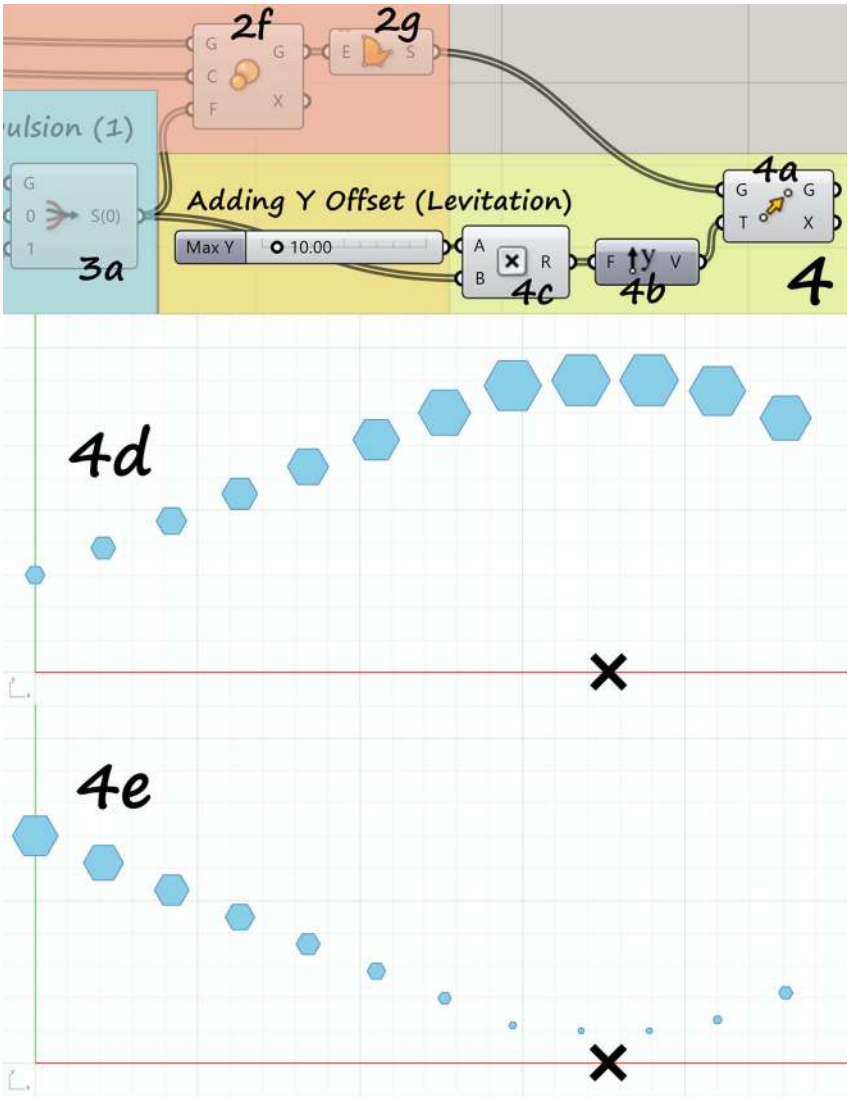


Figure 3-7. *Adding Y levitation to the 1D attractor/repeller. We multiply the scale factors by 10 (4c) before using them as Factor (F) for Unit Y (4b) and subsequently Motion (T) for Move (4a) to offset the polygons along Y. (4d) shows the output of "attractor" mode where polygons are larger and higher nearer the emitter. (4e) shows the output of "repeller" mode, which are the opposite of the attractor*

2D Attractors

With a 1D attractor/repeller under our belt, we're now ready to generalize it to 2D. Figure 3-8 shows a bird's-eye view of the node graph for the 2D attractor/repeller outlining four stages. In stage 1, we set up the array of polygonal curves; in stage 2, we create the emitter point; in stage 3, we compute the scale factors based on the distances between polygon centers and the emitter; and in stage 4, we create surfaces from the polygon curves and scale them.

Tip To get a bird's-eye view of a node graph like the one shown in Figure 3-8, where only the silhouettes of the nodes remain, keep scrolling the mouse wheel to zoom out until the texts within the nodes start to disappear. You can save this view by clicking the "eye" icon in the GH editor just above the canvas ➤ Add named view..., which will save a shortcut to the view in a dropdown list.

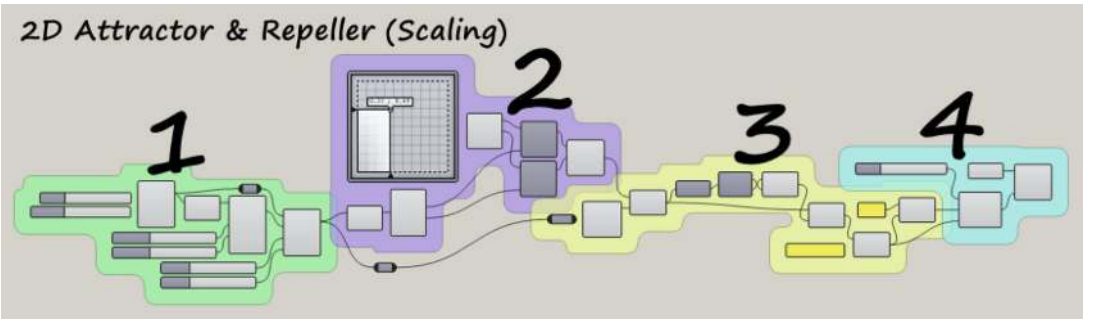


Figure 3-8. *Outline of the 2D attractor/repeller node graph, in four stages. (1) creates a rectangular array of polygon curves. (2) creates an emitter point from user input. (3) computes the scale factors based on distances between polygon centers and the emitter. (4) creates surfaces with the polygon curves as boundaries and scales them*

Stage 1: Setting Up a 2D Array of Polygon Curves

Figure 3-9 shows each stage of Figure 3-8 zoomed in. In stage 1, we create a 2D grid of curves with Transform ➤ Array ➤ Rectangular Array (1b), which requires a Rectangle linked to its Cell (C) input to define the extent of a single grid cell, and the number of rows and columns specified through the X and Y inputs.

A "Rectangle" is a closed, rectangular polyline in GH that can be used anywhere a curve or polyline is expected. We can create the Rectangle here in a parametric way, by first linking Polygon to Surface ➤ Primitive ➤ Bounding Box to get the minimum box that will enclose one curve, followed by Transform ➤ Affine ➤ Scale NU (1a) to scale the box along X and Y to create additional slack, in other words, making the grid cell larger than the polygon, so the polygons don't touch.

Stage 2: Creating an Emitter Point

In stage 2, we create an emitter point within the confine of the grid based on user input. We use Params ➤ Input ➤ MD Slider (2a) to collect a user specified point with X and Y ranging from 0 to 1. For example, the lower-left corner of the slider is (0.00, 0.00), the center of the slider is (0.50, 0.50), and so on. To construct an emitter point, we need to translate the MD Slider output into coordinates in our 2D array of curves. For example, if the 2D array spans X = 0 to 8, and Y = 0 to 12, then (0.50, 0.50) from the MD Slider should correspond to (4, 6).

We link Rectangular Array (1b) to Bounding Box (2d) (set to right-click ➤ Union Box) to derive the minimal box that will enclose the entire array, followed by Surface ➤ Freeform ➤ Deconstruct Box (2e) to extract its spans along X and Y.

Next, we separate MD Slider's output into separate X and Y values by linking it to a Vector ➤ Point ➤ Deconstruct node (2b), so we can map each value to the outputs of Deconstruct Box (2e) using Maths ➤ Domain ➤ Remap Numbers (2c), with Source (S) left at the default value of 0 to 1 (to match MD Slider's domains), and the output of Deconstruct Box (2e) as the Target (T) domains. With X and Y remapped, we can combine them into the emitter point via Vector ➤ Point ➤ Construct Point (2f).

Stage 3: Deriving Scale Factors for Polygons Based on Their Distances to the Emitter

In stage 3, we compute the scale factors based on the distances between polygon centers and the emitter. We link Rectangular Array (1b) to Curve ➤ Analysis ➤ Polygon Center (3a) to find the curve centers, then Vector ➤ Point ➤ Distance (3b) to compute the distances between the centers and the emitter point (2f).

As we've done with the 1D attractor, we divide the distances by their span to convert them into *percentages*. To find the span, we link Distance (3b) to Maths ➤ Domain ➤ Bounds (3c) to get the min to max of the distances as a domain, followed by Maths ➤

Domain ➤ Deconstruct Domain (3d) to extract its Start (S) and End (E) values. We then use the Subtract node (3e) to get the difference between the two and Divide (3f) to find the ratio of the distances to the span and Clamp (3g) to limit it to 0.1–0.9, which become the scale factors for the "repeller" mode. The complement (one minus) of the scale factors (3h) then become the factors for the "attractor" mode.

Note You can double-click an MD Slider to change its X domain and Y domain. However, since we need to pass MD Slider's output through Remap Domain anyway, it is irrelevant whether the point comes from 0 to 1 or another domain.

Furthermore, if we set a non-default X or Y domain in MD Slider, we would need to set the Source (S) input of Remap Numbers to match. Moreover, since MD Slider does *not* have any input, there is no way to set its domains parametrically from the grid size.

Tip To navigate a large node graph like the one in Figure 3-9, you can hold down RMB and drag to pan around the canvas and scroll the mouse wheel to zoom in or out.

Stage 4: Creating Surfaces from Curves and Scaling Them

In stage 4, we create surfaces from the polygon curves and scale them based on the scale factors computed in stage 3. We follow the same setup as the 1D attractor, where we link the two lists of scale factors, (3h) for attractor and (3g) for repeller, to a Stream Filter (4a) node, and use a Number Slider to toggle between the two.

We then create surfaces with the polygon curves as their borders using Surface ➤ Freeform ➤ Boundary Surface (4c) and scale them with Transform ➤ Affine ➤ Scale (4b), using the filtered stream from (4a) as Factor (F).

Figure 3-10 shows sample outputs of the node graph from Figure 3-9 in "attractor" mode on the left and "repeller" mode on the right, with the emitter point marked by X.

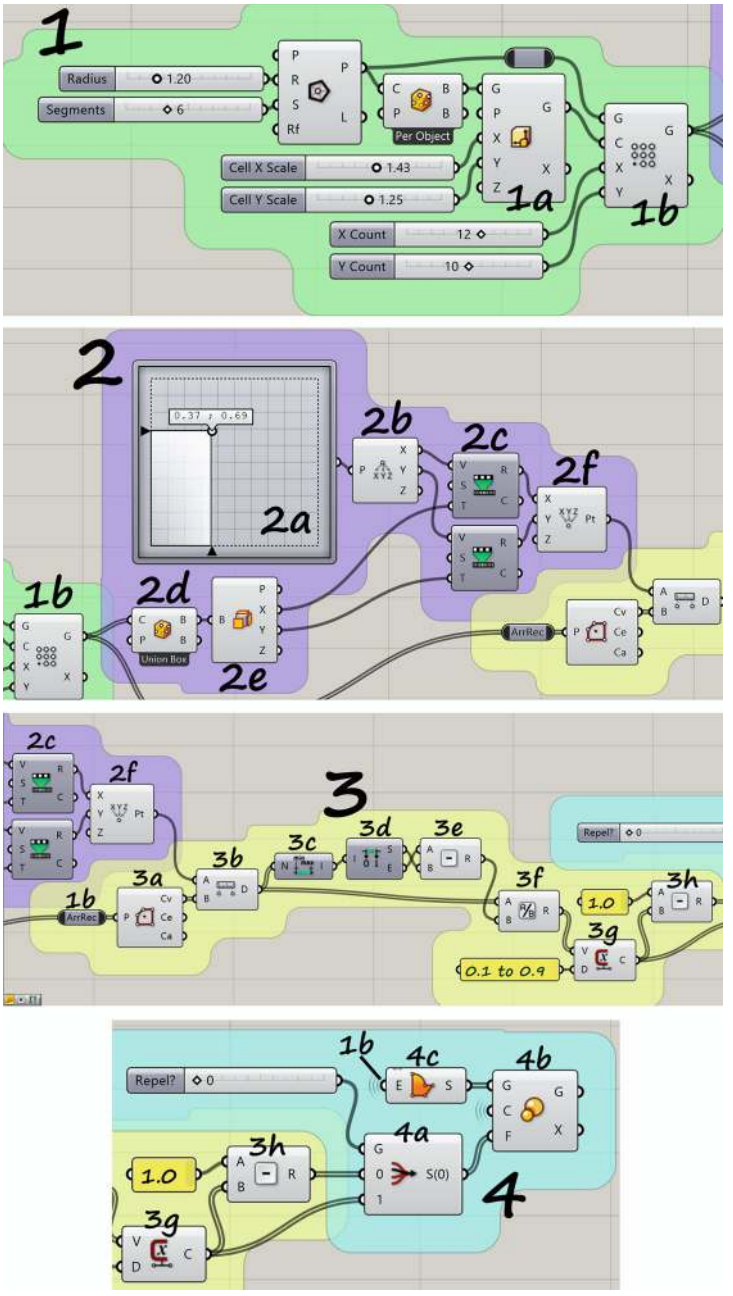


Figure 3-9. *The four stages of the node graph for the 2D attractor/repeller shown in detail. (1) creates a rectangular array of polygon curves. (2) creates an emitter point from user input. (3) computes the scale factors based on distances between polygon centers and the emitter. (4) creates surfaces with the polygon curves as boundaries and scales them*

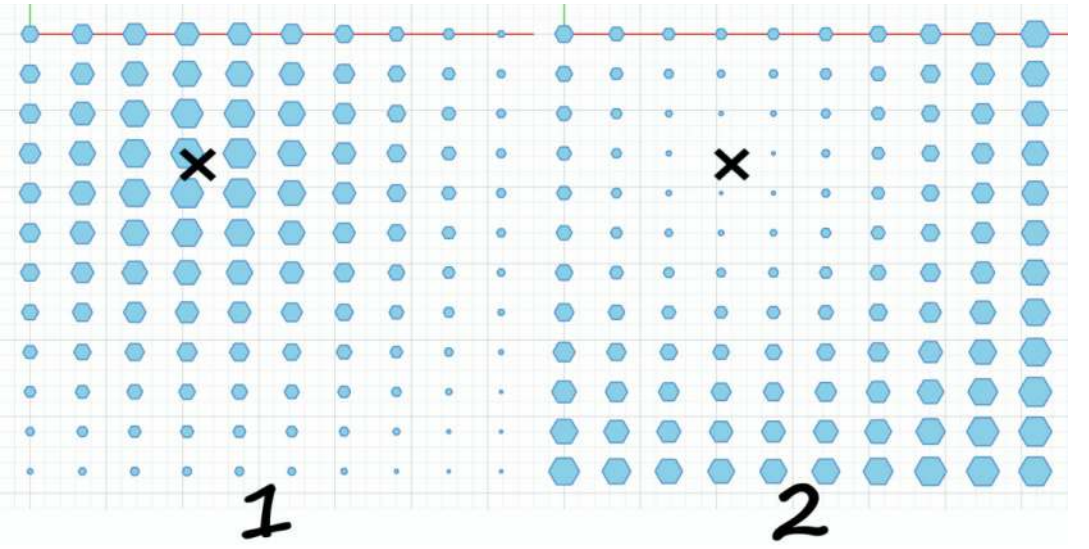


Figure 3-10. *Sample outputs of the 2D (1) attractor and (2) repeller from Figure 3-9. The emitter point is marked by X*

Tip: Organizing Node Graphs with Relay and Hidden Wires

Before we move on, I'd like to point out several tricks for organizing node graphs in Figure 3-9. Notice in stage 3, we used a Relay node labeled ArrRec (1b) on the wire from Rectangular Array (1b) to Polygon Center (3a)—this helps us in two ways, the first is a Relay acts like an extra "joint" in the wire, which allows you to bend and pose the wire with more control to avoid going under other components. The second is that you can label the wire by right-clicking the Relay and entering a custom name (e.g., ArrRec for Rectangular Array), which helps you keep track of the origin of a wire over long distances. To insert a Relay, double-click the spot on a wire where you'd like the Relay to appear. You can remove a Relay by double-clicking it again.

Two inputs in stage 4 have hidden wires, which are the Center (C) input for the Scale node (4b) and the Edges (E) input for the Boundary Surfaces node (4c)—you can tell these two wires are hidden by the "wireless" icons shown at the input ports. To set an input wire as hidden, such that the wire is only revealed when the node at either end of the wire is selected, right-click that input ➤ Wire Display ➤ Hidden. Note that you can only set an incoming wire to an input as hidden, not an outgoing wire from an output. The benefit of hiding a wire is the reduction of visual clutter due to long wires going

under other nodes. However, once a wire is hidden, there is no visual indication which output the wire emanates from—therefore, it would appear at a glance some outputs have missing connections. For this reason, it is best to use hidden wires sparingly.

Adding Levitation

Just as we did with the 1D attractor/repeller, we can add levitation to the 2D version by reusing the scale factors to offset the polygons along the third dimension (in our case, the Z axis). Figure 3-11 shows this modification as an added fifth stage, where we use Transform ➤ Euclidean ➤ Move to offset the polygon surfaces with Unit Z scaled by the output of the Stream Filter (4a) multiplied by a Number Slider, set to 12.5 as shown. The sample output for the attractor mode is shown in (5a), with perspective view on the top and front view on the bottom, where polygons closer to the emitter are higher. (5b) shows sample output for the repeller mode, where polygons closer to the emitter are lower.

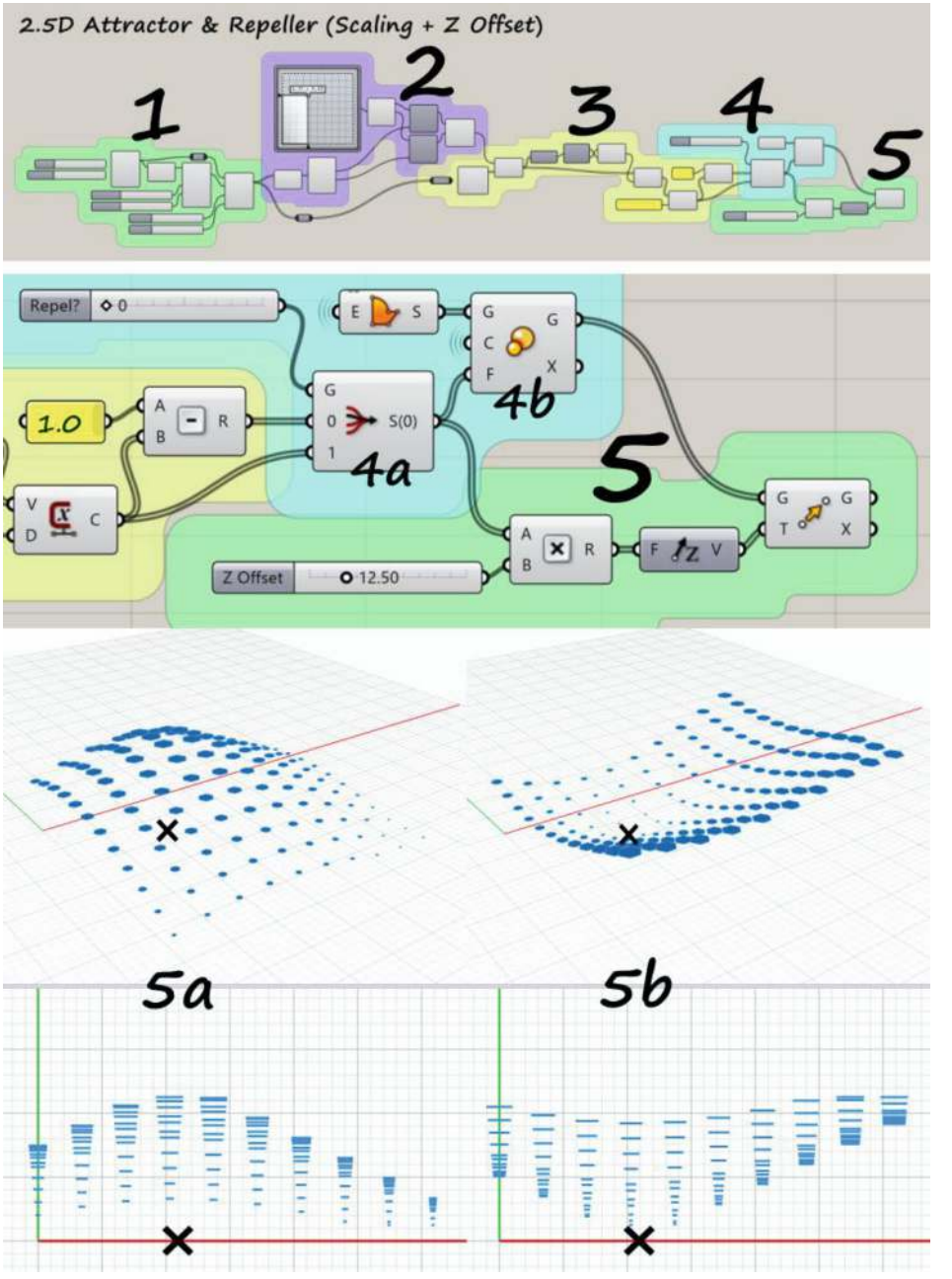


Figure 3-11. *Adding Z levitation to the 2D attractor/repeller. (5) adds a Move node to offset the polygon surfaces along Z based on the filtered scale factors (4a). (5a) shows sample output of the attractor mode, with perspective view on the top and front view on the bottom. (5b) shows sample output of the repeller mode. The emitter is marked by X*

3D Attractors

Instead of levitating the polygons in the 2D grid, we can extrude them along Z to create a 3D attractor/repeller. Figure 3-12 shows the node graph for the base 2D attractor/repeller from Figure 3-9 with a fifth stage added to extrude the polygons after scaling and a sixth stage to pull/push the columns using a vector field.

Creating a Basic 3D Attractor via Extrusion

In stage 5 in Figure 3-12, we link the scaled polygon surfaces (4b) to a Surface ➤ Freeform ➤ Extrude node (5a) to extrude them into columns along Z. To control the amount of extrusion, we multiply the output of the Stream Filter (4a) by 18.65 (from a Number Slider) to use as Factor (F) for Unit Z (5b). The output of stage 5 is shown in Figure 3-13, with attractor mode on the top and repeller mode on the bottom.

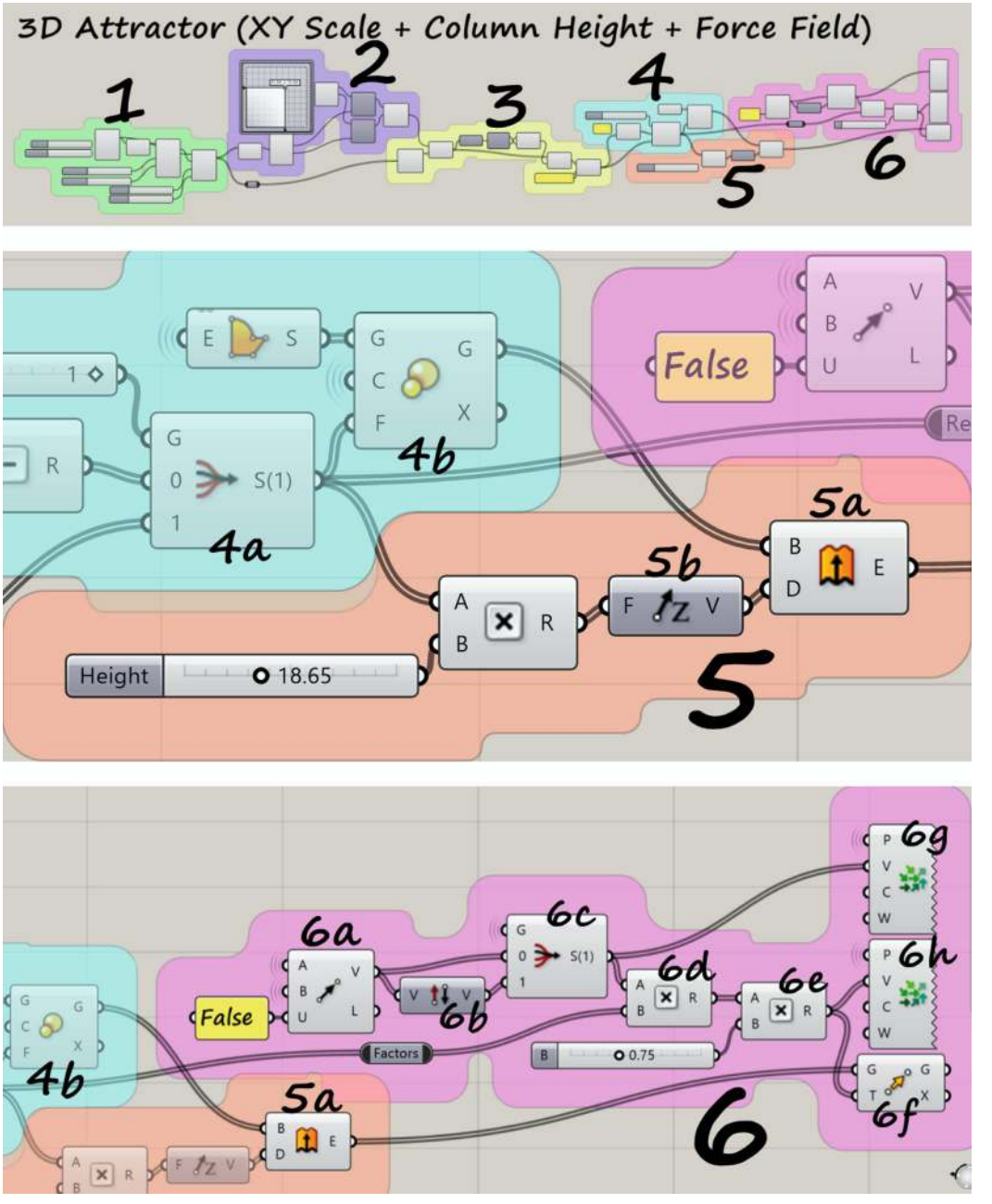


Figure 3-12. *Modifying the base 2D attractor/repeller from Figure 3-9 to add a 5th stage to extrude the polygon surfaces along Z by reusing the scale factors from the 4th stage and a 6th stage to deform the columns further using a force field (discussed in the next section)*

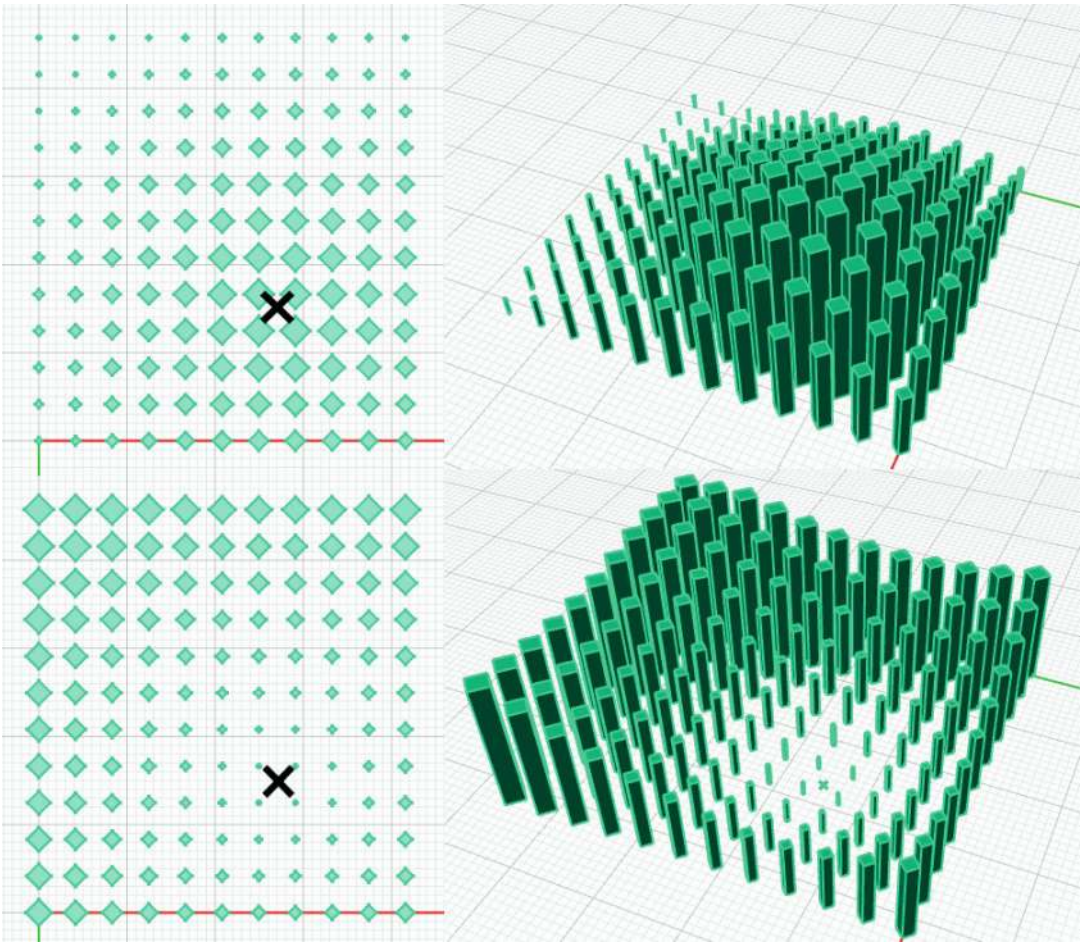


Figure 3-13. *Sample outputs of stage 5 from Figure 3-12, where polygon surfaces are extruded along Z in proportion to the scale factors from the 4th stage. The top shows the top (left) and perspective (right) views of the attractor mode, with the emitter point marked by X. The bottoms shows the top (left) and perspective (right) views of the repeller mode*

Displacement Using a Vector Force Field

To accentuate the effect of the emitter, we'll use a vector force field to contract the grid of columns by pulling them horizontally toward the emitter in attractor mode and pushing them away from the emitter in repeller mode.

A *vector* tells you the direction and amount to move from one point to another. Say we have a point p1 = (x1, y1, z1) and a second point p2 = (x2, y2, z2). To move from p1 to p2, you would displace p1 by the vector:

$$v_{p1\,to\,p2} = p2 - p1 = (x2, y2, z2) - (x1, y1, z1) = (x2 - x1, y2 - y1, z2 - z1)$$

For example, if p1 = (1, 2, 3) and p2 = (4, 5, 6), the vector v to move from p1 to p2 is

$$v_{p1\,to\,p2} = (4,5,6) - (1,2,3) = (4-1, 5-2, 6-3) = (3,3,3)$$

In other words,

$$p1 + v_{p1\,to\,p2} = p2$$

Note that vectors are directional. If now you are moving from p2 to p1, which is the opposite direction, the vector becomes

$$v_{p2\,to\,p1} = (1,2,3) - (4,5,6) = (1-4, 2-5, 3-6) = (-3, -3, -3)$$

You may have noticed that

$$v_{p1\,to\,p2} \textit{ and } v_{p2\,to\,p1} \textit{ are opposite in direction and equal in length.}$$

You can derive one from the other by multiplying –1, like this:

$$v_{p1\,to\,p2} = v_{p2\,to\,p1} \times (-1)$$

$$v_{p2\,to\,p1} = v_{p1\,to\,p2} \times (-1)$$

This brings us back to the attractor/repeller. In attractor mode, you move each column toward the emitter and scale that movement based on the distance between them, so columns closer to the emitter move more. Since we extruded the polygon surfaces upward to form the columns, the polygon centers are at the bottom of the columns, sitting on the XY plane along with the emitter. Therefore, the vectors to move the columns *toward the emitter* are equal to the emitter point minus the polygon center points—we implement this in stage 6 of Figure 3-12, by using a Vector ➤ Vector ➤ Vector 2Pt node (6a) with A hooked up to Polygon Center (3a in Figure 3-9) and B to the emitter

point (Construct Point, 2f in Figure 3-9). We set U to False to not unitize the output vectors, meaning to not make the vectors' lengths 1, therefore retaining the vectors' lengths.

To get the vectors in the opposite direction from the emitter to the polygon centers, we link the Vector 2Pt (6a) output to Vector ➤ Vector ➤ Reverse (6b). We then link the attract vectors (6a) and repel vectors (6b) to 0 and 1 of a Sets ➤ Tree ➤ Stream Filter (6c) node, respectively, and link the same Number Slider that controls the previous Stream Filter used for scaling (4a in Figure 3-9) to its Gate (G) input, so when you drag that slider, you toggle between attract and repel for both scaling and vector field in a unified way.

Our job is not done, however, since so far, the vectors are the differences between the polygon centers and the emitter. Therefore, if you use the vectors as they are, you'll be moving all the columns to where the emitter is in attractor mode! To create the effect we want, such that columns closer to the emitter move more, we'll scale the vectors using the same factors used to scale and extrude the columns (6d). We then make the vector field user adjustable, by multiplying them with a Number Slider (set to 0.75) in 6e. Lastly, to move the columns, we use a Transform ➤ Euclidean ➤ Move node (6f) and link Extrude (5a) to its Geometry (G) input and the user-adjusted vectors (6e) to its Motion (T) input.

Figure 3-14 shows sample outputs at the end of stage 6 in Figure 3-12, with the attractor mode at the top, where columns are moved toward the emitter, and the closer a column is, the more it is pulled. The bottom shows output of the repeller mode, where columns are moved away from the emitter, and the further a column is, the more it is pushed.

The vector fields are visualized on the left side of Figure 3-14. Vectors are not previewed by default in Grasshopper. To draw vectors in the viewport, you must use either Display ➤ Vector ➤ Vector Display or Vector Display Ex—with the difference being the latter lets you customize line color and width of the vectors. The vector fields in Figure 3-14 are produced using a Vector Display Ex node (6g, 6h), with Polygon Center (3a in Figure 3-9) as its Point (P) input, which are the start of the arrows, and the user-adjusted vectors (6e) as the Vector (V) input. You can see that the vector arrows for the attractor mode (top) get progressively longer as they approach the emitter.

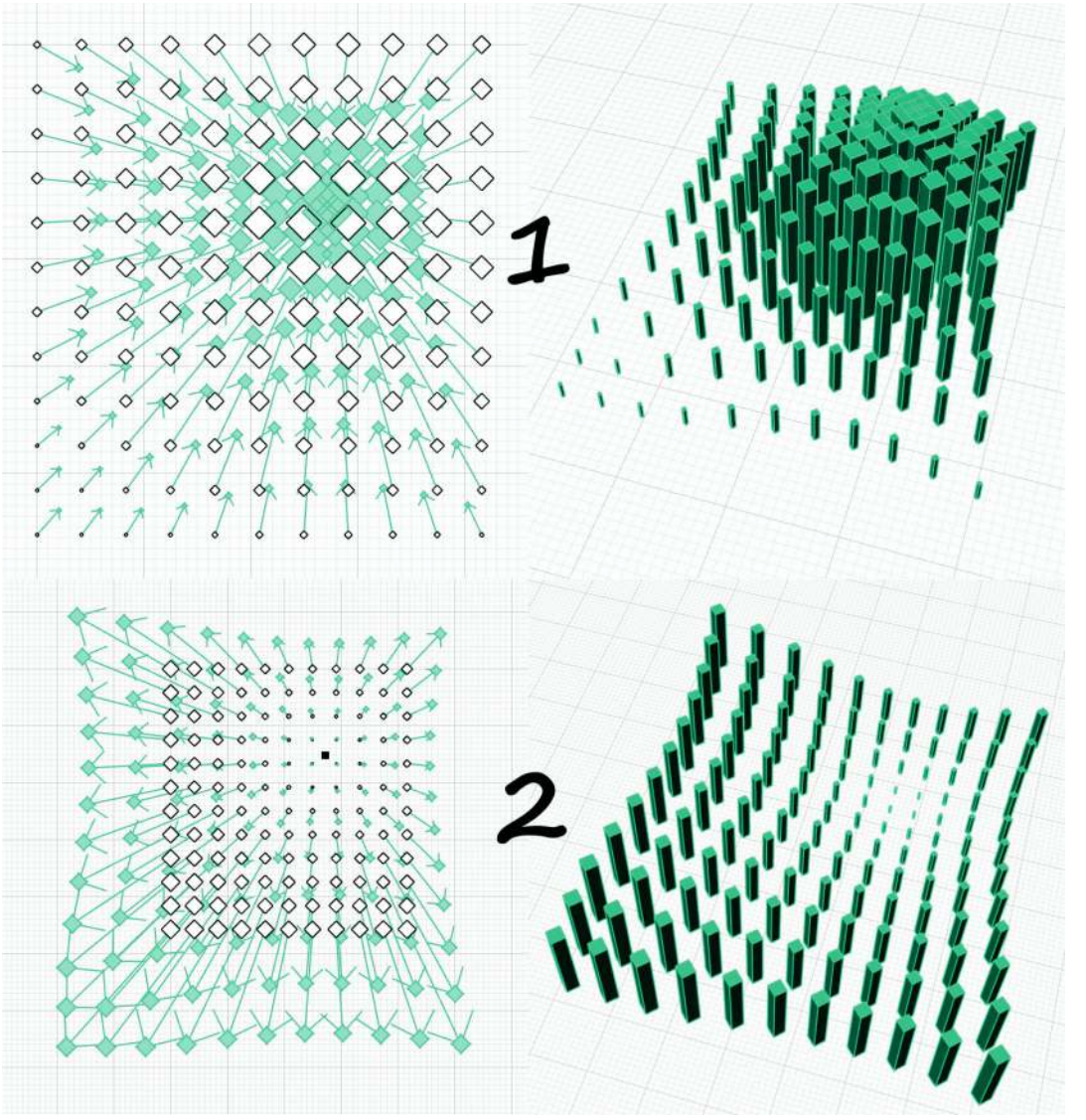


Figure 3-14. *Using vector force fields to attract (pull) columns toward the emitter (1) or repel (push) columns away from the emitter (2)*

Polar Grid Attractor

We've focused our attention so far on rectangular grids. It turns out there are other kinds of grids we can use to create attractors/repellers. We will explore the polar grid in this section, which is a circular grid with objects arranged radially outward from the center in concentric rings.

Stage 1: Creating a Polar Grid with Python 3 Script and Polar Array

Figures 3-16 and 3-18 show the node graph for a 2D polar attractor/repeller. The overall structure of the graph is identical to the 2D rectangular attractor/repeller in Figure 3-9, where the graph is broken into four stages. In stage 1, we create an array of curve objects, in this case, a polar array of circular curves using Rhino Python 3. In stage 2, we create an emitter point based on user input mapped to the polar grid, again with the help of Python 3. In stage 3, we compute the scale factors based on the distances between curve centers and the emitter point. In stage 4, we turn the curves into surfaces and scale them.

Let's take a closer look at stage 1. To construct a polar grid, we'll use the Transform ➤ Array ➤ Polar Array node (1c), which takes four inputs—Geometry (G) is the geometry to repeat, Plane (P) the plane the array will sit on, Count (N) the number of times to repeat G, and Angle (A) the angle in radians to sweep counterclockwise. We'll see how this works in a moment. Grasshopper draws an imaginary line from the outer edge of G to the origin of P and sweeps A radians counterclockwise starting at this line, then evenly distributes N copies of G along the pie arc. Figure 3-15 shows two simple polar arrays each spanning ¼ Pi created using a Polar Array node.

In Figure 3-15, we start by linking two Construct Point nodes (1 and 2) to the P input of a Circle node (3) to create two circular curves centered at (8, 0, 0) and (0, 8, 0), respectively. We then link the Circle node (3) to the G input of Transform ➤ Array ➤ Polar Array (4), leaving P at default (XY plane), setting N to 4 via a Number Slider, and A to 0.25*pi using a Panel (with the string 0.25*pi as shown, using a lowercase pi). Since two circles are fed to Polar Array, two arrays are created, as shown in the middle of Figure 3-15. Each array spans 0.25*pi (or 45 degrees) counterclockwise about P's origin, with N=4 circles distributed along the length of the arc, as shown at the bottom of Figure 3-15. Note that each array starts and ends where the first and last circles are *tangent* to the pie slice.

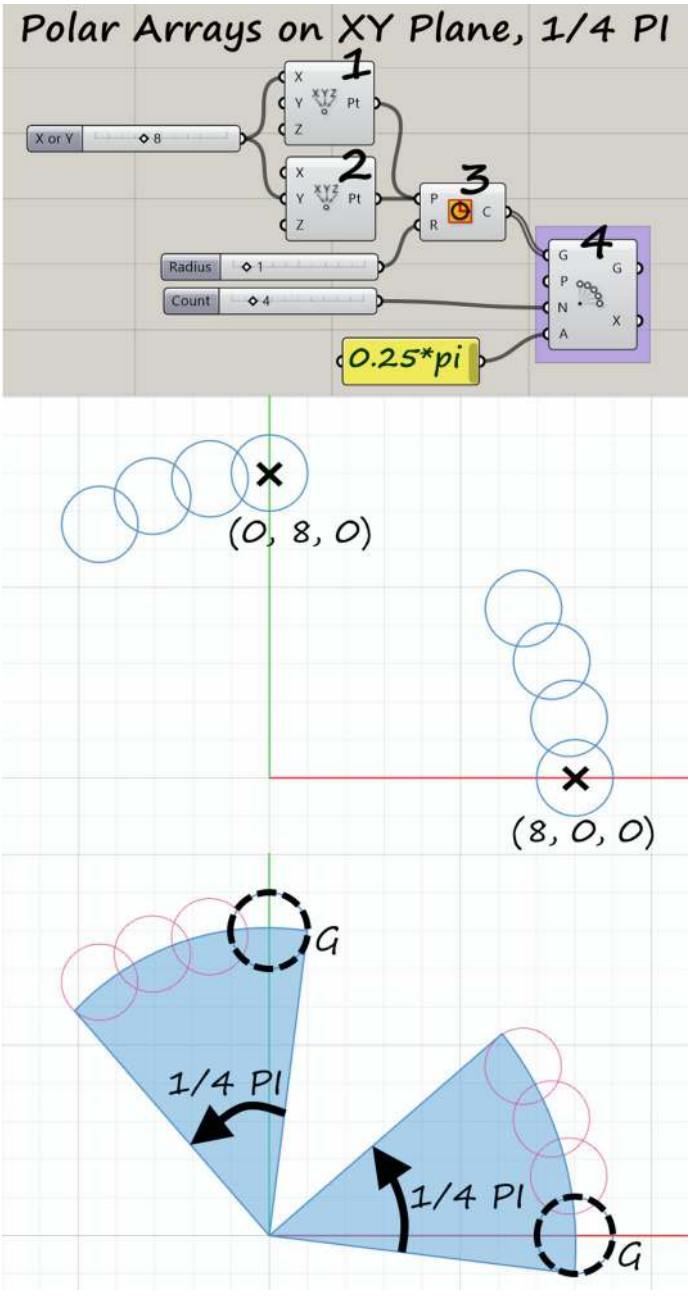


Figure 3-15. *Two sample arrays created using the Polar Array node, one starting at the circle centered at (8, 0, 0) with radius 1, the other starting at the circle centered at (0, 8, 0). Both arrays sweep ¼ Pi counterclockwise on the XY plane*

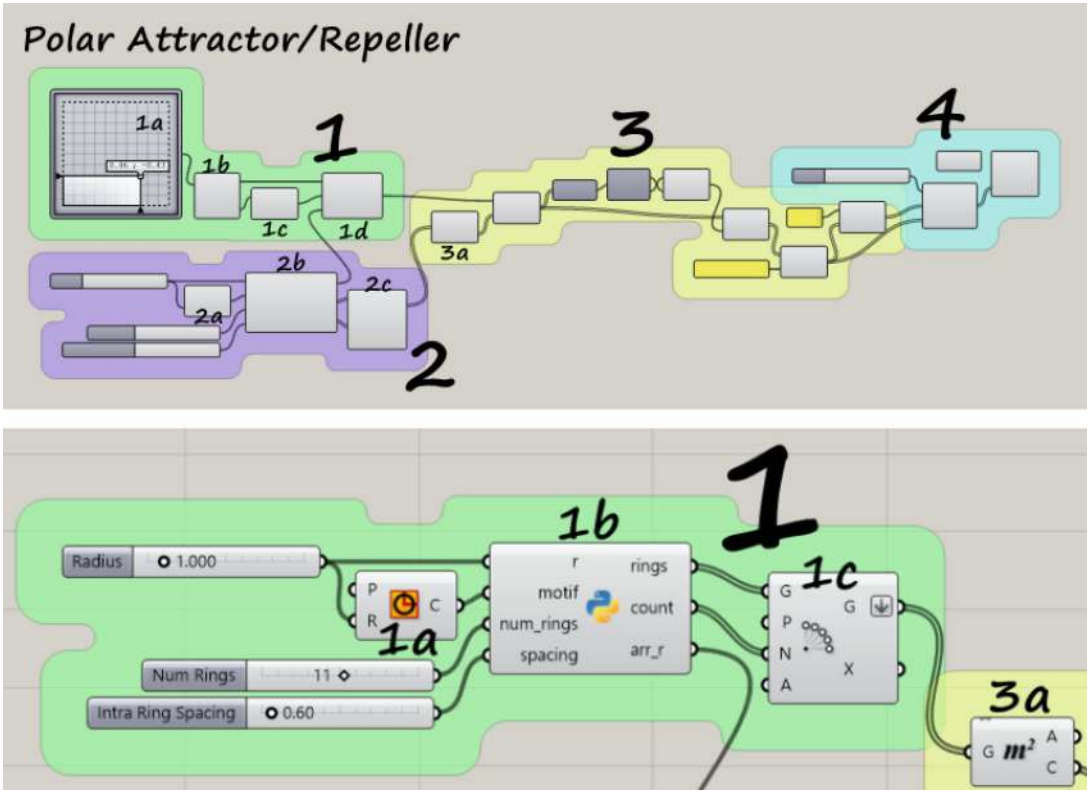


Figure 3-16. *Bird's-eye view (top) and the first stage (bottom) of the polar attractor/repeller graph. In stage 1, we create a polar array of circular curves with the help of Python 3*

You might have noticed that for each given instance of geometry, Polar Array creates a new arc of repeats starting at that instance. To make the arc come all the way around and form a circle, we make A=2*pi. Our purpose is to produce an array as the basis for an attractor/repeller; therefore, we want the array to consist of concentric rings of circles evenly spaced out. To achieve this, we want to feed the Polar Array node a list of circles, each centered on the edge of one concentric ring, with the rings' radii increasing outward such that the array elements do not overlap.

We will create a Maths ➤ Script ➤ Python 3 Script node (1b in Figure 3-16) to create this list of circles (the "rings" output) and compute how many circles will take per ring to fill it (the "count" output), along with the radius of the entire polar array as a whole (the "arr_r" output). The Python 3 Script node (1b) will take four inputs as shown in Table 3-1, which are

1. "r": The radius of the geometry to be repeated
2. "motif": The geometry to be repeated (assumed to be a curve in this case)
3. "num_rings": The number of concentric rings in the array
4. "spacing": Optional spacing between adjacent rings

Recall from Chapter 2 that you scroll the mouse wheel to zoom in on the Python 3 Script node and press the "+" ("-") button to add (remove) an input or output. You then right-click an input or output to change its name, Access, and Type Hints, which are summarized in Table 3-1 for (1b).

Table 3-1 *Summary of Access and Type Hints for the Python 3 Script node (1b) in stage 1 of the polar attractor/repeller graph in Figure 3-16, which derives parameters to be used with the Polar Array node (1c)*

Name	Input or Output	Access	Type Hints	Python Type
r	input	Item Access	float	float
motif	input	Item Access	Curve	Rhino.Geometry.ArcCurve
num_rings	input	Item Access	int	int
spacing	input	Item Access	float	float
rings	output	N/A	Curve	list
count	output	N/A	int	list
arr_r	output	N/A	float	Float

The Python code for (1b) in Figure 3-16 is reproduced in Listing 3-1 (recall from Chapter 2 that you double-click a Python 3 Script node to open its code editor). We start by importing the `ghpythonlib.components` module as the alias `ghcps`, via which to call Grasshopper components as Python functions. We then create variables for the three

outputs and assign initial values to them: `rings` and `counts` as empty lists and `arr_r` as zero. We also compute the distance (`motif_diam`) to set adjacent rings apart so they don't overlap, by summing the input curve radius (`r`) times two and the input `spacing`.

In the next block of code, we iterate `num_rings` times in a `for` loop to derive stats on the rings from the inside out. In each iteration, we calculate the radius of the current ring (`cur_r`) as follows. "`ring`" is the index of the current ring (counting from 0). The first ring has radius equal to a single curve's radius (`r`) with no previous ring. Each subsequent ring's radius increases by `motif_diam`. Therefore, the Nth ring has `cur_r = r + (N-1)*motif_diam`, which is equal to `r + ring*motif_diam`.

Next, we calculate the perimeter (`cur_circum`) of the current ring as 2Pi times the current ring's radius (`cur_r`). On the next line, we find the vector(`trans_vec`) to move a copy of the input curve to the current ring by calling `ghc.UnitX(cur_r)`, which is equivalent to linking `cur_r` to a Unit X node. On the line after, we move a copy of the input curve (`motif`) to the current ring by calling `ghc.Move(motif, trans_vec)`, which is equivalent to linking `motif` to the Geometry (G) input and `trans_vec` to the Motion (T) input of a Transform ➤ Euclidean ➤ Move node.

After that, we calculate the number of curve copies it will take to fill the ring all the way around, by dividing the ring's perimeter (`cur_circum`) by one curve copy's diameter plus spacing (`motif_diam`) and add it to the `count` list. We also add the curve copy moved to the current ring to the end of the `rings` list. Finally, if we're at the iteration for the outermost ring (`ring == num_rings-1`), we find the radius of the entire polar array by summing the current ring's radius (`cur_r`) and one curve copy's radius (`r`).

The variables used in Listing 3-1 are visualized at the top in Figure 3-17. The output of the Transform ➤ Array ➤ **Polar Array** node (1c) in Figure 3-16 are visualized on the bottom of Figure 3-17. As shown, the input `motif` is a circular curve (output of Circle (1a)) with `r` = 1.000, `num_rings` = 11, and `spacing` = 0.60. We make use of (1b) by linking its `rings` output to the G input of Transform ➤ Array ➤ **Polar Array (1c)** and the `count` output to the N input, while leaving P and A default (XY plane and 2Pi, respectively), to produce the polar array shown on the bottom of Figure 3-17 with 11 concentric rings of curves.

Listing 3-1. Code for the Python 3 Script node (1b in Figure 3-16) that creates the list of single curve motifs with one at each concentric ring for a polar array and computes the number of items for each ring and the overall array radius (see /Ch3/Figure3-16_3-18_3-19_polar_attractor_repeller.gh in the book's downloaded source files)

```
import ghpythonlib.components as ghc

rings = []
count = []
arr_r = 0
motif_diam = (r*2+spacing)

for ring in range(0, num_rings, 1):
    cur_r = r + motif_diam*ring
    cur_circum = 2*3.14*cur_r
    trans_vec = ghc.UnitX(cur_r)
    moved_shp, _ = ghc.Move(motif, trans_vec)
    count.append(cur_circum/motif_diam)
    rings.append(moved_shp)

    if ring == num_rings-1:
        arr_r = cur_r + r
```

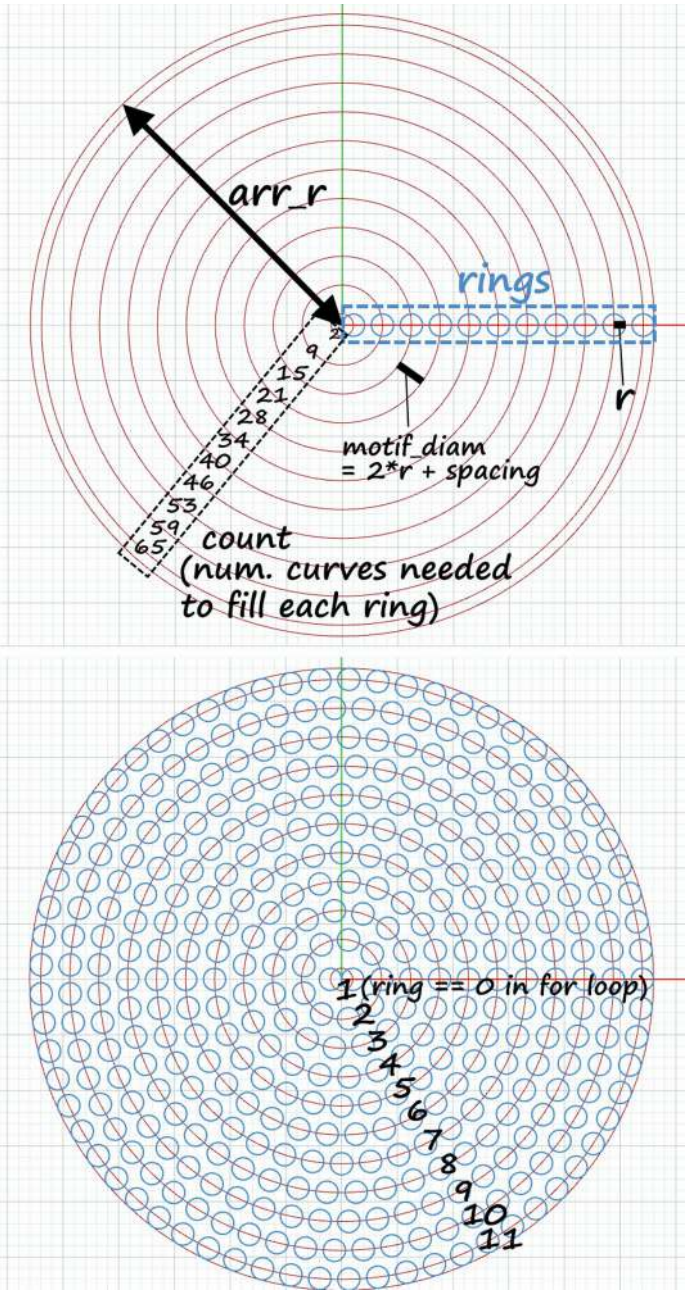


Figure 3-17. *(Top) Visualization of inputs and outputs of the Python 3 Script node (1b) in Figure 3-16 and Listing 3-1. (Bottom) Visualization of the polar array output by the Polar Array node (1c) in Figure 3-16*

Stage 2: Creating an Emitter Point for a Polar Grid

In this section, we'll zoom in on stage 2 of Figure 3-18, which creates an emitter point for the polar grid. We use a **MD Slider** (2a) with X and Y domains set to –1 to 1 to allow a user selection in all four quadrants of the polar grid, as shown in Figure 3-19. If the XY domains were 0 to 1, only the first quadrant (upper right) would be selectable. We'll implement the following steps to map the **MD Slider** (2a) output to coordinates within our custom polar array:

1. Link **MD Slider** (2a) to Vector ➤Point ➤**To Polar** (2b), which gives us the point's equivalent angle **Phi (P)** and **Radius (R)** in a polar array whose perimeter goes through the four vertices of the **MD Slider**'s domain square.
2. Link **To Polar** (2b)'s **Radius (R)** to Maths ➤Util ➤**Clamp** (2c) to limit it between 0 and 1, which has the effect of limiting the boundary of the polar array to at most the circle tangent *inside* **MD Slider**'s domain square.
3. Map the outputs of **To Polar** (2b) to our custom polar array (1c in Figure 3-16) using a **Python 3 Script** node (2d).

The Python 3 Script node (2d) takes three inputs:

1. **x** is the angle of the point in the source polar array (**To Polar** (2b)'s **Phi (P)** output).
2. **md_r** is the radius of the source polar array (**To Polar** (2b)'s **Radius (R)** output).
3. **arr_r** is the radius of the target polar array (**arr_r** output of the previous Python 3 Script node (1b in Figure 3-16).

Based on which, the Python 3 Script node (2d) then computes the equivalent polar coordinates in the target array and outputs a point (Pt) created from those coordinates. The Access and Type Hints for 2d are summarized in Table 3-2.

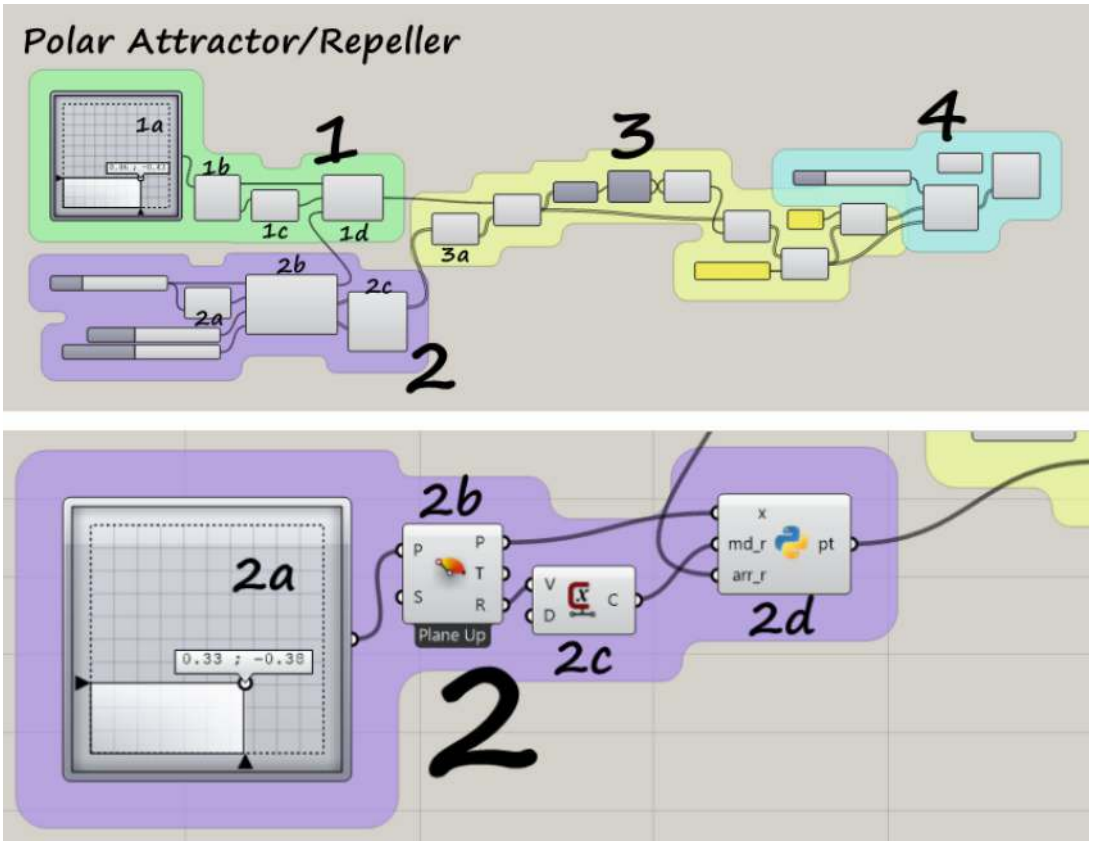


Figure 3-18. *Bird's-eye view (top) and the second stage (bottom) of the polar attractor/repeller graph. In stage 2, we create the emitter point based on user input, again with the help of Python 3*

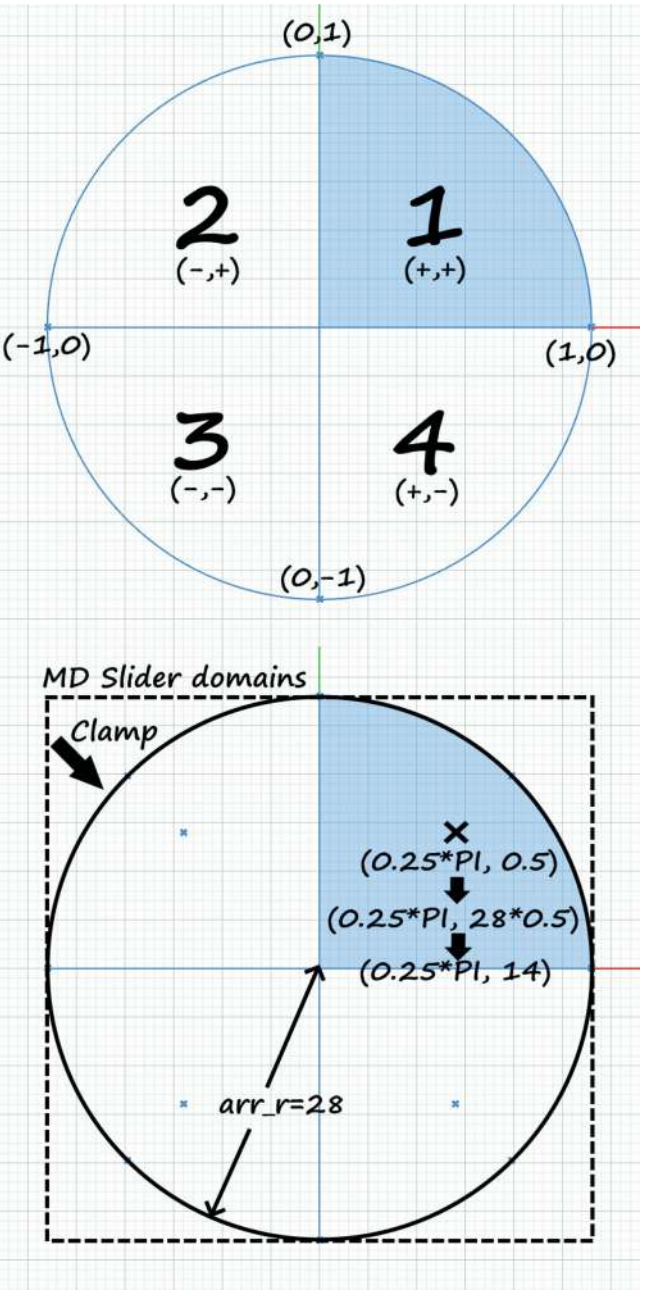


Figure 3-19. *Mapping MD Slider input to coordinates in our polar grid (produced by 1b and 1c in Figure 3-16). Top: To allow a point selection in all four quadrants, we must set both the X and Y domains for MD Slider to (-1,1). Bottom: The raw outputs of the To Polar node (2b in Figure 3-18) map to a square area that exceeds the bounds of our polar grid. Therefore, we must use a Clamp node (2c) to limit it*

Table 3-2 *Summary of Access and Type Hints for the Python 3 Script node (2d) in stage 2 of the polar attractor/repeller graph in Figure 3-18, which computes the coordinates for and creates the emitter point*

Name	Input or Output	Access	Type Hints	Python Type
x	input	Item Access	float	float
md_r	input	Item Access	float	float
arr_r	input	Item Access	float	float
pt	output	N/A	Point3d	Rhino.Geometry.Point3d

Listing 3-2 shows the code for the Python 3 Script node (2d) in Figure 3-18 that computes the coordinates for and creates the emitter point in our custom polar grid. We start by importing the `ghpythonlib.components` module as the alias `ghcps` so we can call Grasshopper components as Python functions. After that, we import the sine and cosine (`sin`, `cos`) functions from Python's built-in `math` module in preparation for the polar coordinate conversions.

In the following block of code, we compute the scaled radius (`scaled_r`) as the product of the source and target max polar radii. Let's think about why that is. The output of **To Polar** (2b) is the polar coordinates (P, R) of a point from the MD Slider with X and Y spanning –1 to 1, where P is the angle in radians measuring from the X axis counterclockwise and R the point's distance to the origin. The max R of this polar array is the half diagonal of the MD Slider's square domain, or 0.707. We use **Clamp** (2c) to limit the R output of **To Polar** (2b) so that it's between 0 and 1, thereby making the source polar array a *unit circle*.

As a result, R can be thought of as the percentage of a point along the line from the circle's origin to its perimeter—for example, a point (P=0.25*Pi, R=0.5) is 50% between the array origin and perimeter as shown in Figure 3-19—this percentage does *not* change when you enlarge the array. Since the source array has R at most 1 and the target array has R at most **arr_r** = 28, the point (P=0.25*Pi, R=0.5) can be mapped to the target array by multiplying its R component by 28, which becomes (P=0.25*Pi, R=28*0.5=14).

The P component stays the same since enlarging the circle does not change the relative proportion of the pie slice. The input parameters **x** and **md_r** for the Python 3 Script node in Listing 3-2 correspond to P and R, respectively, while **scaled_r** is the result of **md_r** scaled based on the max R of the target array (**arr_r**).

Now that we have the input point's polar coordinates mapped to the target polar array, the next step is to translate it to Cartesian coordinates in the form of (X, Y, Z) so we can use it in distance calculations for the attractor. As shown in Figure 3-20, you can draw a line straight down from the point in question to create a right triangle with x as the angle between *scaled_r* and the X axis. The point's Cartesian coordinates can then be calculated using the following trig formulas:

$$px = scaled_r \times \cos(x)$$

$$py = scaled_r \times \sin(x)$$

We then call `ghcps.ConstructPoint` to create a point from X=px, Y=py, and Z=0, which is equivalent to using a Vector ➤ Point ➤ Construct Point node.

Listing 3-2. Code for the Python 3 Script node (2d in Figure 3-18) that maps the output of To Polar (2b) to our custom polar grid and creates an emitter point (see `/Ch3/Figure3-16_3-18_3-21_3-22_polar_attractor_repeller.gh` in the book's downloaded source files)

```
import ghpythonlib.components as ghcps
from math import sin, cos

scaled_r = md_r*arr_r
px, py = scaled_r*cos(x), scaled_r*sin(x)
pt = ghcps.ConstructPoint(px, py, 0)
```

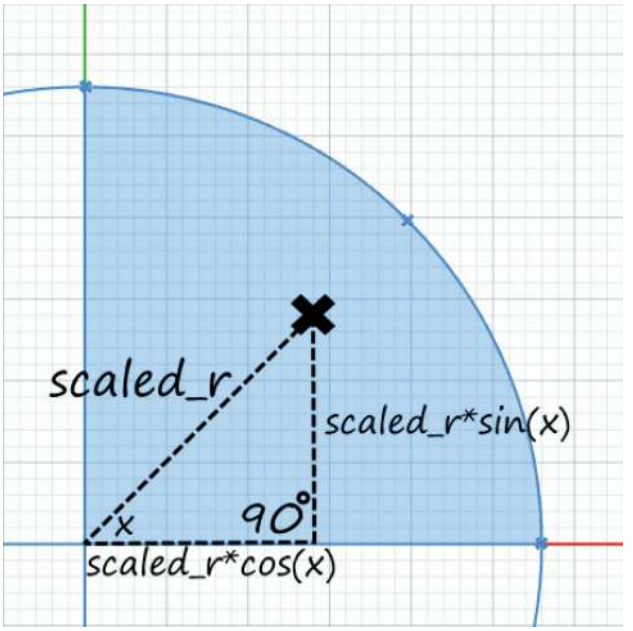


Figure 3-20. *Translating a point's coordinates from polar to Cartesian. As shown, the point has polar coordinates (x, scaled_r), which is equivalent to (scaled_r*cos(x), scaled_r*sin(x)) in Cartesian*

Stage 3: Computing Scale Factors from Distances Between the Emitter Point and Circle Centers

Now that we have a polar grid filled with circular curves and an emitter point created from user input, the next step is to compute scale factors for the attractor/repeller based on distances between the emitter and the circle centers. This step is identical to the rectangular grid implementation earlier in this chapter, except for we use a Surface ➤ Analysis ➤ **Area** node (3a in Figure 3-21) to get the curve centers instead of Curve ➤ Analysis ➤ Polygon Center (3a in Figure 3-9), since circular curves cannot be used as inputs to Polygon Center.

Stage 4: Creating Surfaces from Circles and Scale Them

In the final stage of the polar attractor/repeller graph, we use a Sets ➤ Tree ➤ Stream Filter node as a switch for toggling between attractor (0) and repeller (1) mode. We then use Surface ➤ Freeform ➤ Boundary Surfaces to create surfaces with the circular curves as their borders and use Transform ➤ Affine ➤ Scale to scale them based on the filtered list of scale factors. This stage is identical between the polar (Figure 3-21) and the rectangular grid (Figure 3-9) implementations. Sample results for the 2D polar attractor/repeller are shown in Figure 3-22.

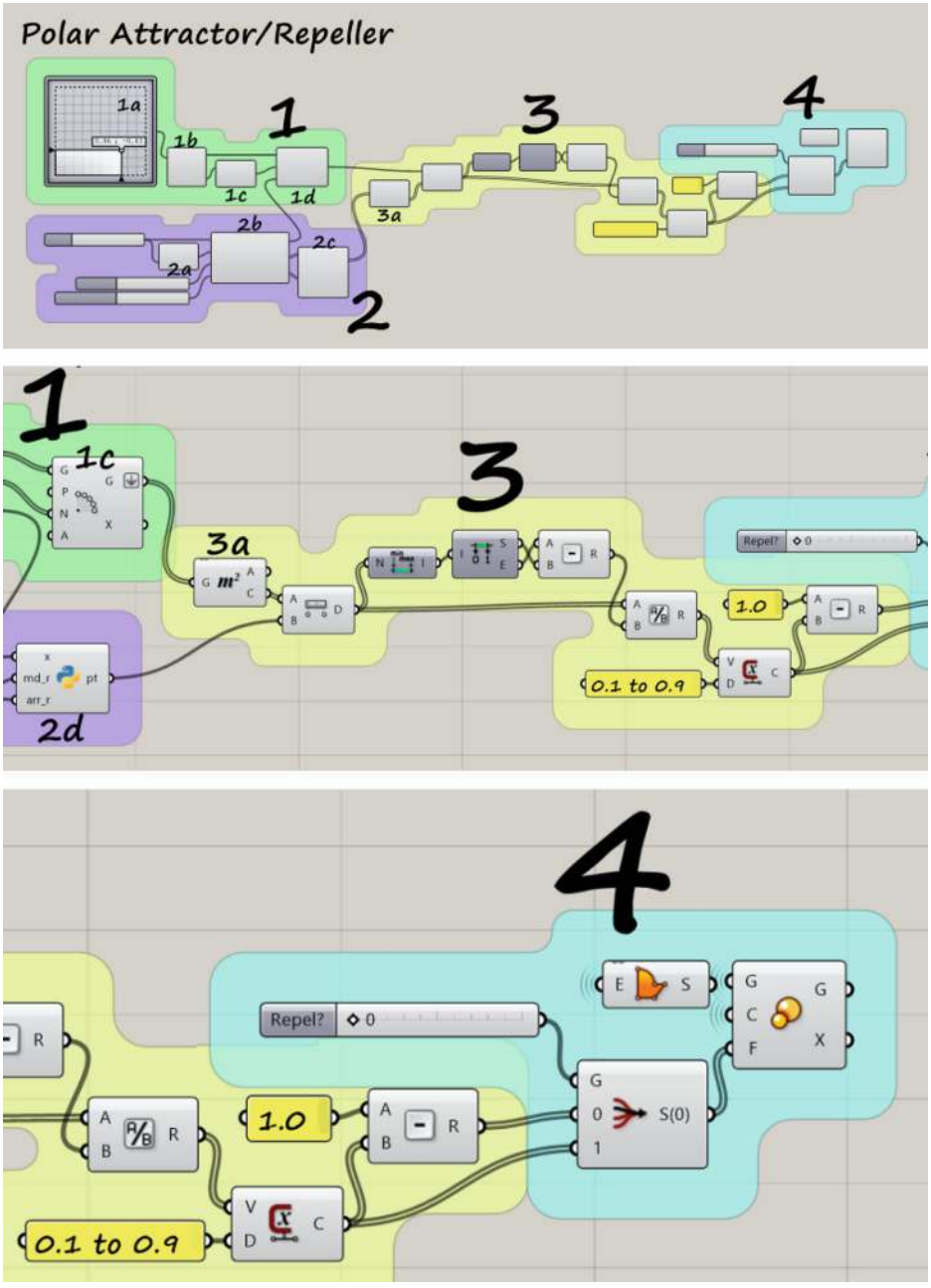


Figure 3-21. *Bird's-eye view (top) and stages 3 and 4 (bottom) of the polar attractor/repeller graph. In stage 3, we compute scale factors based on distances between curve centers and the emitter. In stage 4, we turn the curves into surfaces and scale them*

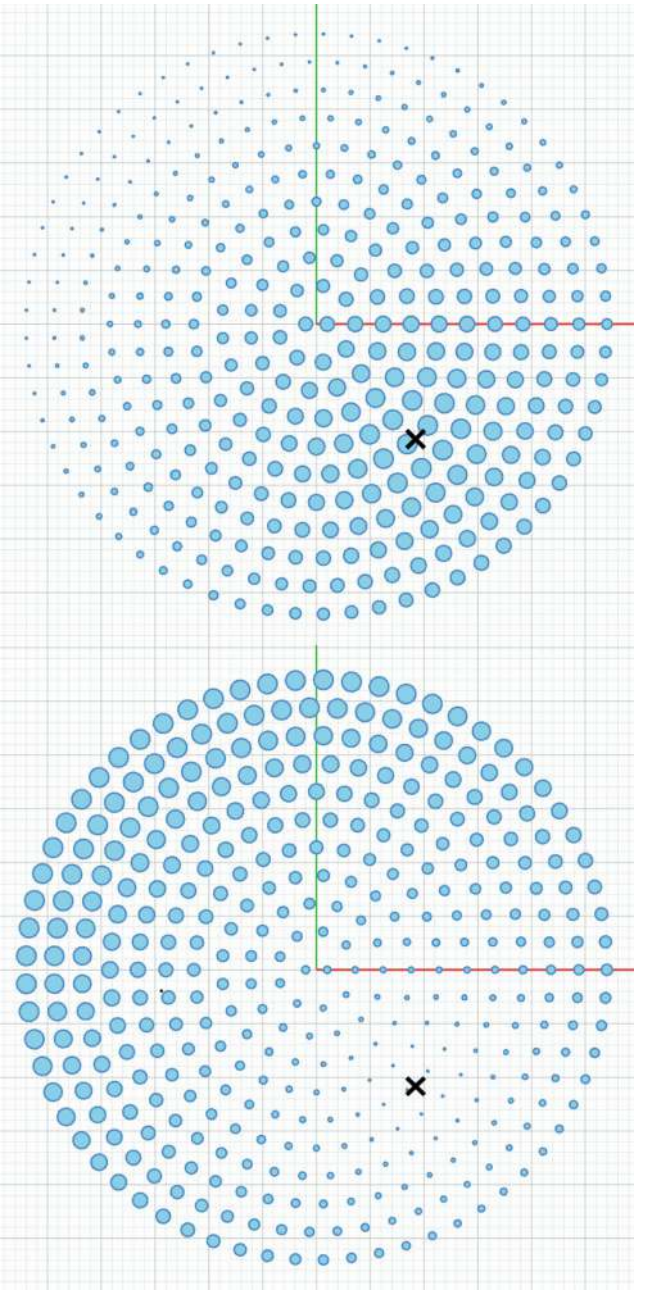


Figure 3-22. *Sample outputs of the 2D polar attractor (top) and repeller (bottom), implemented by the node graph in Figure 3-21. The emitter point is marked by X*

Tip You can copy the currently selected nodes with Ctrl-C and paste them with Ctrl-V, either within the same GH file or to a different file.

Curve Emitters

In this section, we'll explore using a curve emitter instead of a point. It turns out we can make this modification fairly easily by replacing *stage 2* of any of the 2D~3D attractor/repeller designs in this chapter with the node graph shown in Figure 3-23, which creates a parametric emitter curve. Recall that for all 2D~3D attractor/repeller designs:

1. Stage 1 creates the grid.
2. Stage 2 creates the emitter*.
3. Stage 3 computes the scaling factors based on the distances between each grid item's center point and the emitter.
4. Stage 4 transforms the grid items based on scale factors from stage 3.

Therefore, we are swapping out the old stage 2 that creates an emitter *point* with a new stage 2 that creates an emitter *curve*. We will only need to make one change to stage 3—instead of computing the distances between each grid item's center and the emitter point, we calculate the distance between each grid item's center and the item's closest point on the emitter curve.

We want to pick two points on opposite sides of the grid's bounding box and connect them to form an emitter curve, as shown in Figure 3-24. Since we've positioned all grids on the XY plane,[1] with X increasing widthwise from left to right and Y lengthwise from bottom to top, we define two parameters, `y_t1` and `y_t2`, such that they increase from 0.00 to 1.00 from bottom to top, for the left and right edges of the grid's bounding box, respectively. By sliding `y_t1` and `y_t2` up and down via two Number Sliders (2c and 2d in Figure 3-23), we move the start and end points of the emitter curve up and down.

Let's take a closer look at the node tree in Figure 3-23, which is a copy of the 2D attractor/repeller from Figure 3-9 with stage 2 replaced by the subtree shown, and the addition of the Curve Closest Point node (3a, more on this in a moment). We link the

[1] Except for the 2.5D attractor/repeller in Figure 3-11.

Rectangular Array node to a Surface ➤ Primitive ➤ Bounding Box node and set it to right-click ➤ Union Box to get the box enclosing all objects in the array, then link it to Surface ➤ Analysis ➤ Deconstruct Box to get the box's X and Y extents as domains. We extract the domains' Start (S) and End (E) values with two Maths ➤ Domain ➤ Deconstruct Domain nodes (2a and 2b) to find from *where to where* the box spans along X and Y. We then combine the box's extents with `y_t1` (2c) and `y_t2` (2d) to find where the end points of the emitter curve should be.

For example, if the bounding box spans X from 10 to 30 (2a: S=10, E=30), Y from 5 to 25 (2b: S=5, E=25), and the user has set the slider for y_t1 to 0.30 (2c) and y_t2 to 0.70 (2d), then

$$\textit{The X coordinate of the emitter curve's start point} = 10$$

$$\textit{The Y coordinate of the emitter curve's start point} = 5 + 0.30 \times (25 - 5) = 11$$

$$\textit{The X coordinate of the emitter curve's end point} = 30$$

$$\textit{The Y coordinate of the emitter curve's end point} = 5 + 0.70 \times (25 - 5) = 19$$

In other words, the Y coordinate of the emitter curve's start (end) point is interpolated between the bounds of the bounding box's Y based on y_t1 (y_t2). Luckily, we can perform this interpolation by simply linking both outputs of 2b to the D input of Maths ➤ Util ➤ Interpolate Data and the two Number Sliders 2c and 2d to the t input (in that order)—this tells Interpolate Data to use each value of t to interpolate the data in D or, in our case, use y_t1 (2c) and then y_t2 (2d) to interpolate the Y bounds, which give us the Y coordinates of the start and end points of the curve emitter.

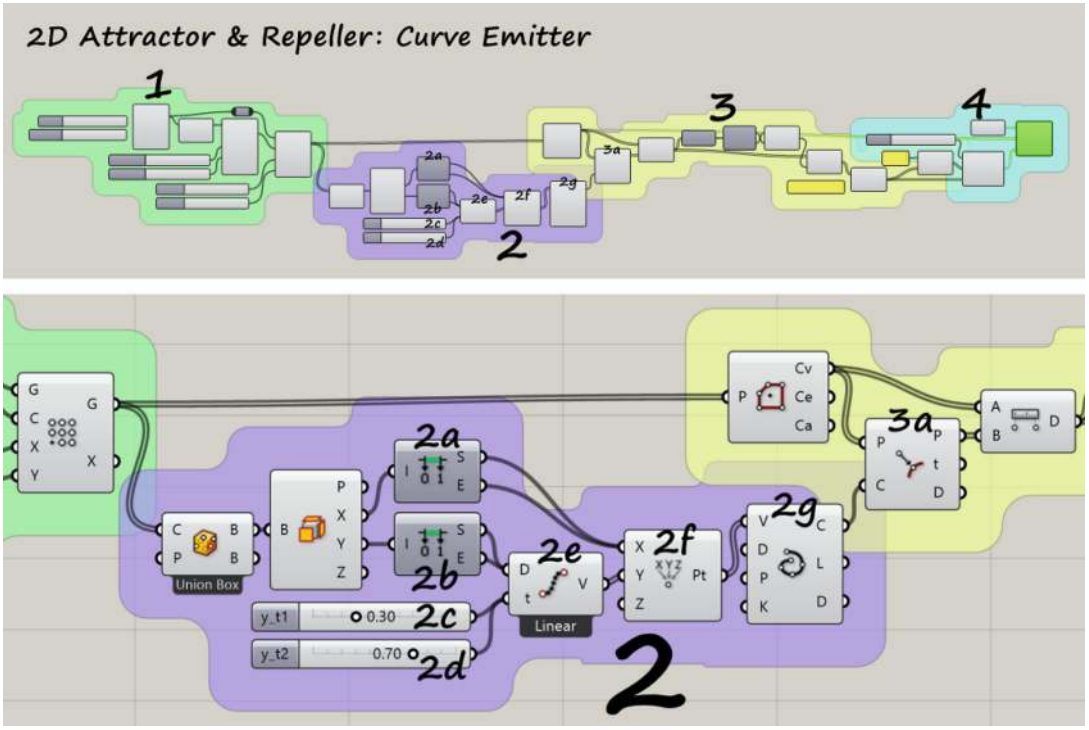


Figure 3-23. *Replacing stage 2 of the 2D attractor/repeller to create an emitter curve, by picking two points along the left and right edges of the grid's bounding box and interpolating them to form a curve. The scale factors in stage 3 are then calculated based on the distances between each grid item's center and its closest point on the emitter curve*

We already know the X coordinates for the start and end points of the curve emitter, which are the same as the min and max X values of the bounding box along X (2a). At this point, we know both the X and Y coordinates for the start and end points of the emitter curve. We create the points using Vector ➤ Point ➤ Construct Point (2f) by linking their X coordinates (2a) to X and Y coordinates (2e) to Y then create the curve using Curve ➤ Spline ➤ Interpolate (2g) by linking the two points (2f) to V in Figure 3-23.

Since the emitter is now a curve instead of a point, we compute the distance between each grid item's center with its closest point on the curve using the Curve ➤ Analysis ➤ Curve Closest Point node (3a in Figure 3-23), by linking Polygon Center to its P input and the emitter curve (2g) to its C input. Sample outputs of the modified 2D attractor/repeller with emitter curve are shown in Figure 3-25.

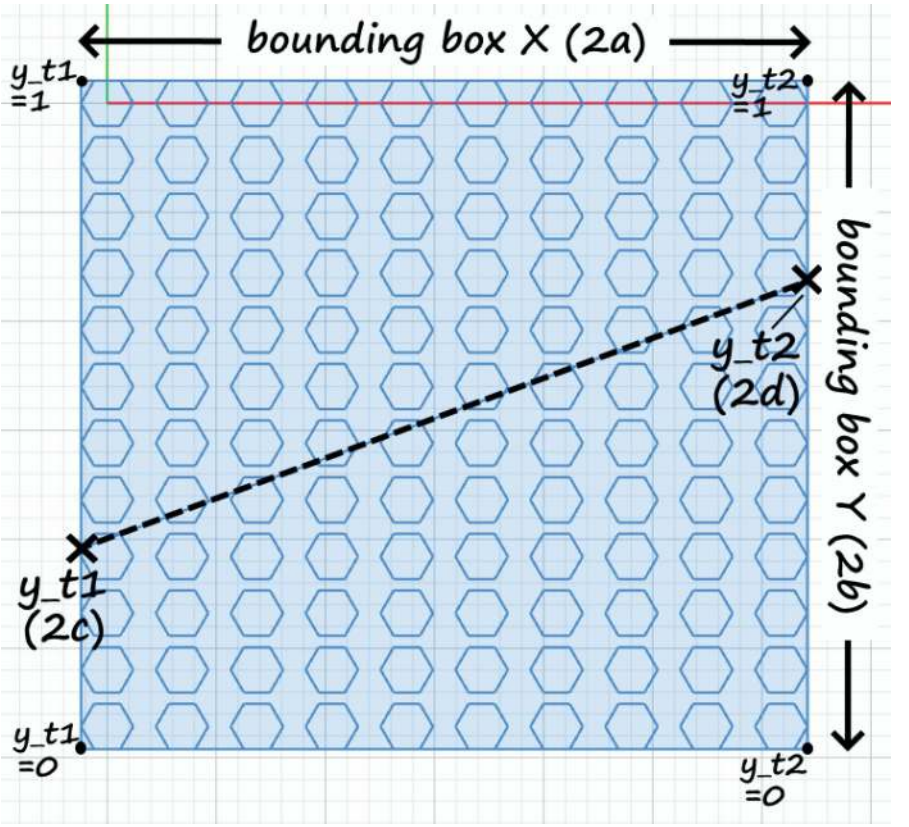


Figure 3-24. *Nodes from the node graph in Figure 3-23 with their relationships to the emitter curve and the grid's bounding box visualized. The two Number Sliders y_t1 and y_t2 control how far the emitter curve's two end points slide up and down the left and right edges of the grid's bounding box*

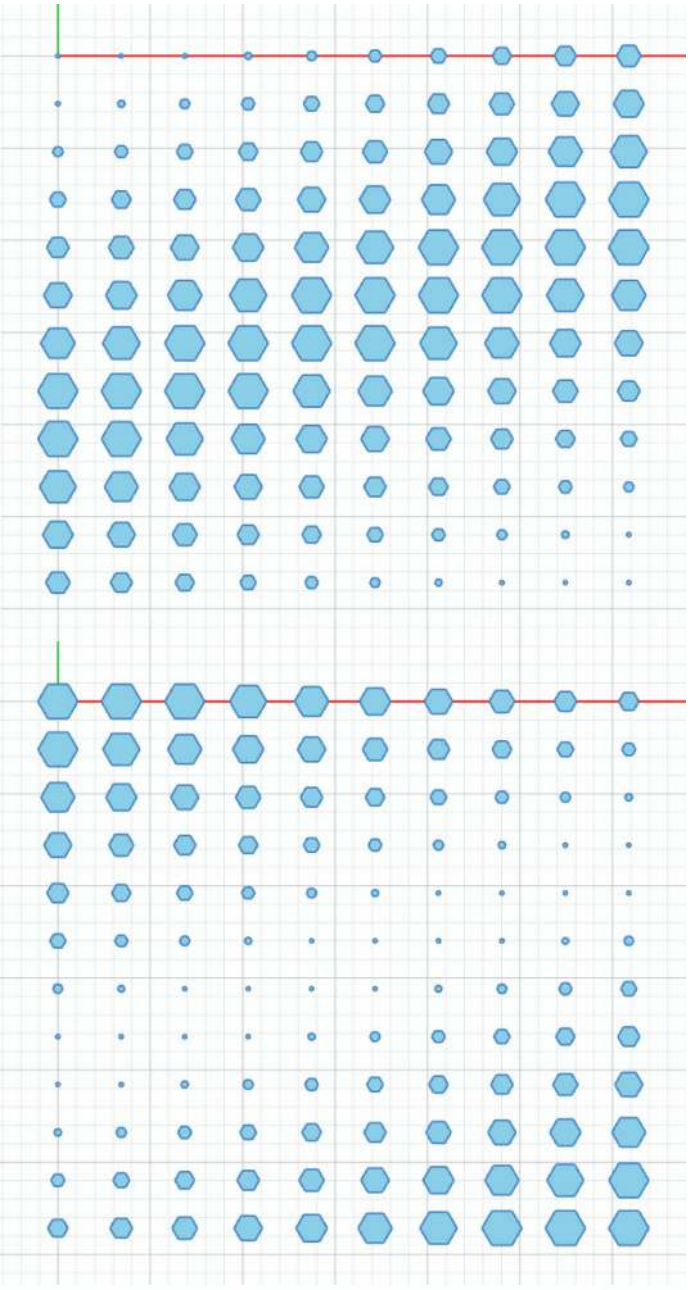


Figure 3-25. *Sample outputs of the modified 2D attractor/repeller with emitter curve in Figure 3-23. Top shows attractor mode, and bottom shows repeller mode*

Let's modify the 2D polar attractor/repeller we've implemented in Figures 3-16 to 3-21 to use an emitter curve instead of a point. Even though the polar grid is a circle, we can employ the same trick of picking points parametrically along the grid's bounding box edges, exactly as we have with the rectangular grid, to use as the emitter curve's end points. Figure 3-26 shows this modification applied to the polar attractor/repeller graph from Figure 3-18. Note that the only differences between the polar version (Figure 3-26) and the rectangular version (Figure 3-23) are that the grid is generated by a Polar Array node (1c in Figure 3-26) instead of Rectangular Array, and the grid items' center points are extracted using a Surface ➤ Analysis ➤ Area node (3b) instead of Polygon Center.

Sample results of the modified 2D polar attractor/repeller with an emitter curve are shown in Figure 3-27, which are annotated with the grid's bounding box and the emitter curve. Notice that even though the curve exceeds the boundary of the polar grid, each grid item center's closest point on the emitter curve is still within the grid.

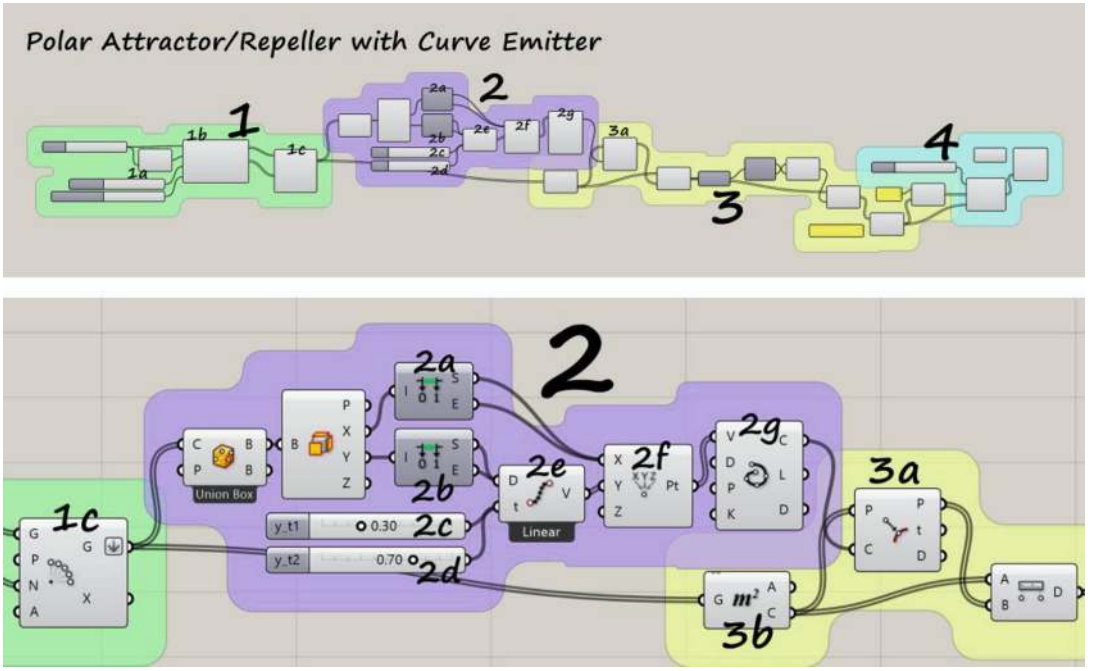


Figure 3-26. *Replacing stage 2 of the 2D polar attractor/repeller to create an emitter curve, by picking two points along the left and right edges of the grid's bounding box and interpolating them to form a curve. The scale factors in stage 3 are then calculated based on the distances between each grid item's center and its closest point on the emitter curve*

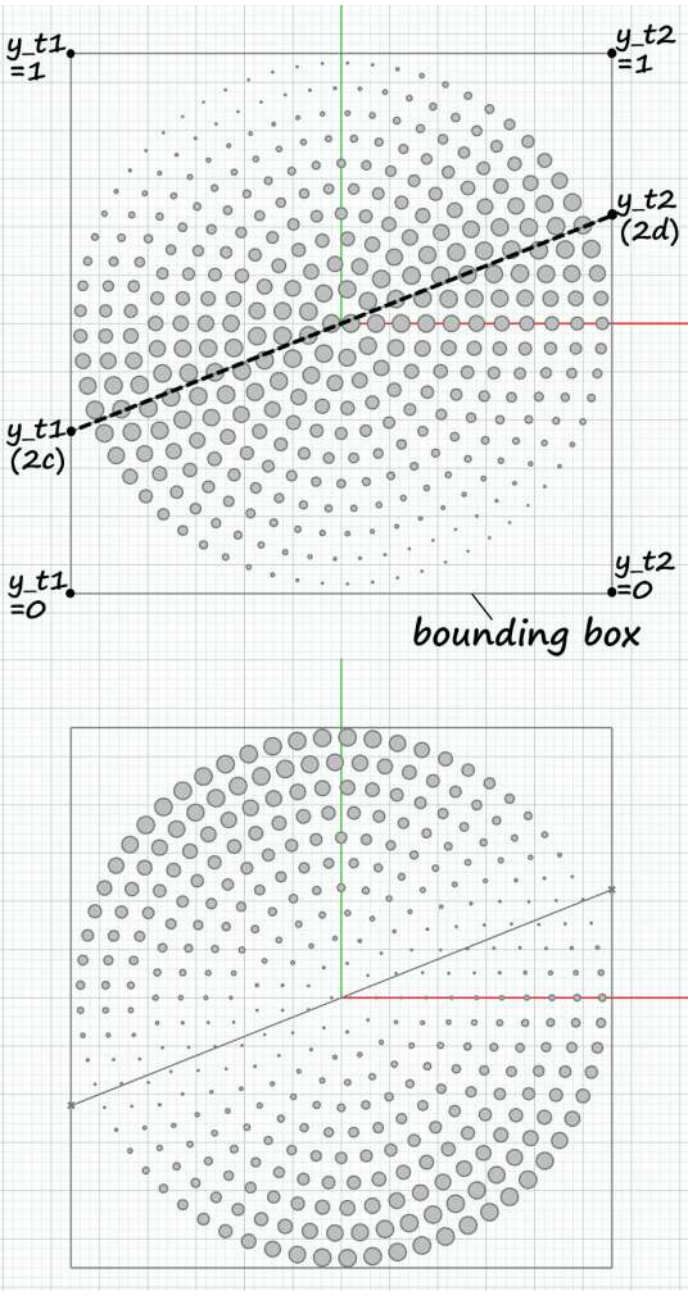


Figure 3-27. *Sample results of the modified 2D polar attractor/repeller with an emitter curve from Figure 3-26. The two Number Sliders y_t1 and y_t2 control how far the emitter curve's two end points slide up and down the left and right edges of the grid's bounding box*

We can just as easily modify the 3D attractor/repeller graph from Figure 3-12 to use an emitter curve instead of a point by applying the same set of changes, which produces the results shown in Figure 3-28. You can find the modified GH file in the book's downloaded sources (`/Ch3/Figure3-28_3D_attr_curve_emitter.gh`).

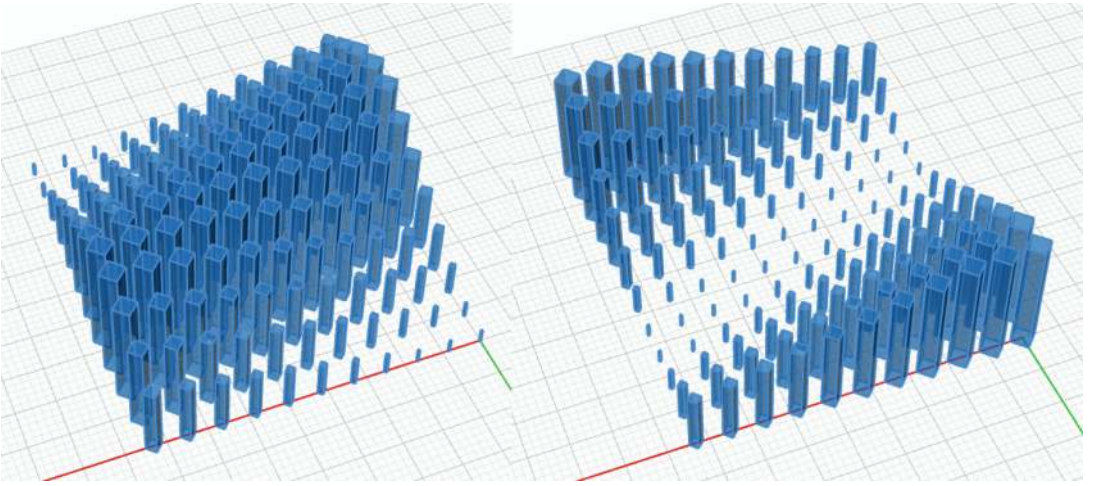


Figure 3-28. *Sample results of the 3D attractor/repeller from Figure 3-12 modified to use an emitter curve instead of a point. You can find the corresponding GH file* `/Ch3/Figure3-28_3D_attr_curve_emitter.gh` *in the book's downloaded source*

Summary

In this chapter, you learned all about *attractors*, which are grids of objects transformed based on their distances to one or more *emitter objects*. We started by building a 1D attractor from a single row of hexagonal curves that are scaled based on how far their center points are from an emitter point, so the closer they are to the emitter, the larger, creating the illusion that they are "pulled in." We then created a variation where the curves are scaled the opposite way, such that they are smaller when closer to the emitter, which we call "repeller" to differentiate from their "attractor" counterpart (although in practice, both variations are commonly referred to as "attractors").

To structure the script to support both "attractor" and "repeller" modes in the same graph, we made use of the Sets ➤ Tree ➤ Stream Filter node, which lets the user toggle between the two modes by allowing one stream of scale factors through at a time. After that, we experimented with one more variation, by adding Y offsets to the row of hex curves to make them appear to levitate based on their proximities to the emitter.

In the second part of the chapter, you continued to build on the attractor/repeller design to take it to 2D. You learned how to use a MD Slider to collect user input for an emitter point and map its coordinates from the XY domains of 0 to 1 to the parametric grid for the attractor. Just like the 1D version, you experimented with creating a repeller variation and adding Z offsets to make grid objects float proportionally to their distances from the emitter.

To create an attractor/repeller that is truly 3D, in the next part of the chapter, you extruded the grid of polygons along Z to form columns based on their proximities to the emitter. You created an additional variation by calculating a "force field" based on the vectors between the polygons' centers and the emitter point then used the field to "contract" and "expand" the columns by offsetting them horizontally.

All the attractor designs up to this point have been based on rectangular grids. In the fourth part of the chapter, you discovered how to write custom Python 3 Script nodes to create polar grids with concentric rings of circular curves. You also learned to translate MD Slider inputs to coordinates within these grids.

To conclude the chapter, you experimented with creating emitter curves based on points selected parametrically along opposite edges of a grid's bounding box and used this technique to modify the 2D, polar, and 3D attractors.

CHAPTER 4

Tessellations

A *tessellation* is created by repeating a single shape or a set of shapes across a surface to cover it completely without any gaps or overlap, like tiles covering the floor of a room. Figure 4-1 shows several examples of tessellations. Mathematicians often refer to surfaces as being infinite in size. However, here, we mean a "surface" like a Brep or Surface in Rhino that has a visual boundary and is not necessarily flat. In this chapter, we'll focus on tiling evenly across a plane.

Tessellations are used as design elements across many cultures. Instead of classifying tessellations based on their appearances (e.g., Islamic, geometrical, etc.) or mathematical properties, in this chapter, we'll take a constructionist approach and learn about tessellations by creating them using four different methods, plus their variations, along with ideas for further experimentation.

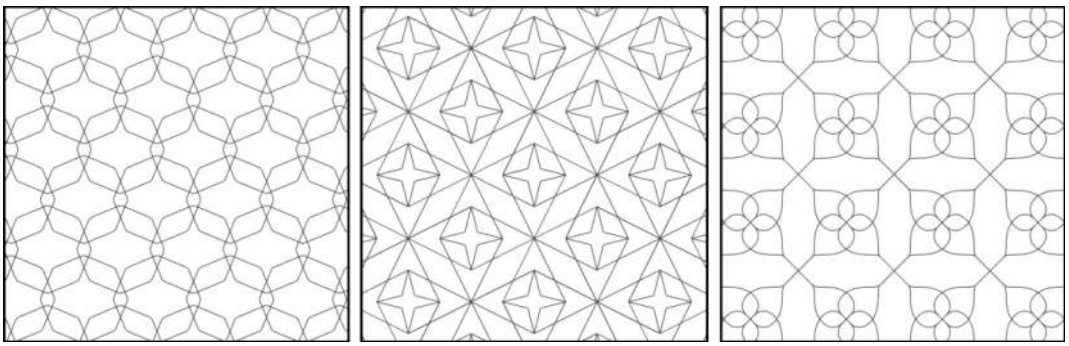


Figure 4-1. *Three examples of tessellations*

Note The tessellations shown in Figure 4-1 are constructed using methods #1, #3, and #4 that will be introduced later in this chapter. You can find their complete node graphs in the file `/Ch4/Figure4-1_tess_examples.gh` in the downloaded source.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_4

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper. Download the files and navigate to the /Ch4 folder where you'll find *.gh files named after the figures in which they appear, along with a short description. For example, Figure4-1_tess_examples.gh. The sample files have been tested in Rhino 9 on Windows. As you read the chapter, you can build your own node graphs by following the steps described in each section or start with the downloaded *.gh files and experiment by modifying their parameters.

Construction Method #1: Overlapping 2D Arrays of Primitives

We'll begin with the simplest construction method—create 2D arrays of primitive shapes (triangles, rectangles, and so on), one array per layer, then stack the layers on top of one another. Many distinctive tessellations can be achieved by varying the distance between shapes within each layer, changing the horizontal offsets used to arrange layers, and rotating shapes within a layer.

Let's start with the simplest case, where we create one 2D array; offset it horizontally to create a second array then combine the two arrays to create a tessellation, as shown in Figure 4-2. In stage 1, we create a rectangular array of triangular curves; in stage 2, we offset the array from stage 1 with a Move node; then in stage 3, we combine the two arrays and use MultiPipe to turn the network of curves into a closed SubD.

Note *SubD* objects in Rhino are Catmull-Clark subdivision surfaces that allow fast and accurate editing of complex organic shapes.

Stage 1: Creating a 2D Array of Polygon Curves

Stage 1 should look familiar, as it's identical to the first stage of the 2D attractor graph from Figure 3-9, with only the slider values changed. To recap, we use a Rectangular Array node (1d) to create a 2D array, with a Polygon node (1a) as the base geometry (G), and the Bounding Box (1b) of the Polygon scaled with Scale NU (1c) as the Cell (C).

We use Segments=3 for Polygon to create triangular curves and a Filet Radius > 0 to make their corners rounded. The spacing between triangles can be increased or decreased by dialing the X and Y inputs for Scale NU up and down to elongate or squash the array cells. Using the slider values as shown in stage 1 of Figure 4-2 produces the array of curves shown in (1) of Figure 4-3.

Note Here is a summary of where you can find the nodes in stage 1 of Figure 4-2: Curve ➤Primitive ➤Polygon (1a), Surface ➤ Primitive ➤ Bounding Box (1b, right-click ➤ Per Object), Transform ➤ Affine ➤ Scale NU (1c), and Transform ➤ Array ➤ Rectangular Array (1d).

Stage 2: Creating a Second 2D Array by Offsetting the First Array Horizontally

In stage 2, our goal is to move the array from stage 1 horizontally by a multiple of the cell size. Since we want most shapes of the two arrays to overlap, we use a multiple between 0 and 1 (0.5 as shown). To measure the size of an array cell, we link the output geometry of Scale NU (1c), which is the cell surface, to a Surface ➤ Analysis ➤ Dimensions node, which gives us the width (U) and height (V) of the cell. To find the amount to offset the array along X and Y, we simply multiply width (U) and height (V) with two Multiplication nodes using the same slider (as the second value) to scale it the same way width- and height-wise. We combine the X and Y offsets using Vector XYZ (2b) to use as the Motion (T) input for the Move node (2c), then link the Rectangular Array (1d) to the Geometry (G) input for Move to produce the shifted array shown in green (2) in Figure 4-3.

Note Here is a summary of where you can find the nodes used in stage 2 of Figure 4-2: Surface ➤ Analysis ➤ Dimensions (2a), Maths ➤Operators ➤Multiplication, Vector ➤ Vector ➤ Vector XYZ (2b), and Transform ➤ Euclidean ➤ Move (2c).

Stage 3: Overlaying the Two Arrays to Produce the Tessellation

In stage 3, we combine the two arrays of curves from stages 1 and 2 by linking Move (2c) and Rectangular Array (1d) to a Curve node (3a). We then turn the network of curves into a single closed SubD by linking Curve (3a) to a MultiPipe node (3b) with a pipe radius of 0.2 (controlled by the NodeSize input), and all other parameters left at default, to produce the tessellation shown as (3) in Figure 4-3.

Combining the two arrays with a Curve node (3a) is optional, as we could have just Shift-LMB dragged both Move (2c) and Rectangular Array (1d) directly to the Curves input of MultiPipe (3b)—the nice thing with using a Curve node (3a) is you can click on Curve to preview the combined arrays in the viewport, as shown in (2) of Figure 4-3.

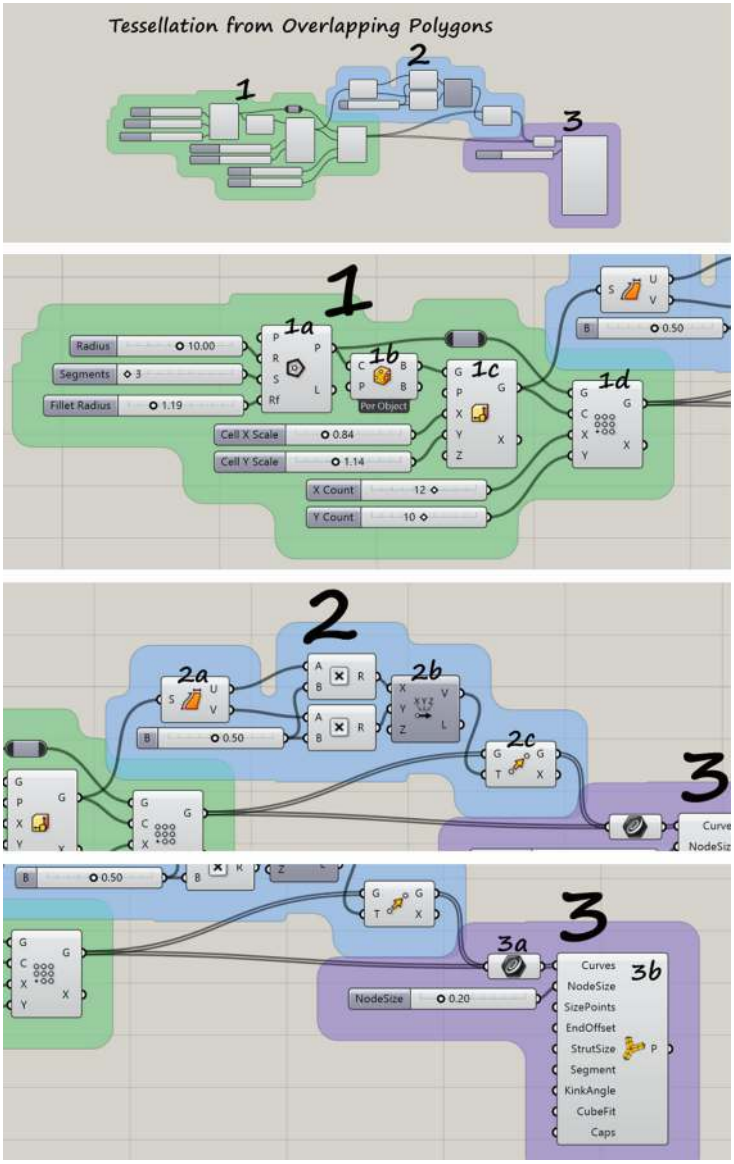


Figure 4-2. *Node graph for creating tessellations by overlaying 2D arrays of primitive shapes. We create a (1) Rectangular Array of curves, (2) offset this array horizontally using a Move node, and (3) combine the two arrays and turn them into a closed SubD using the MultiPipe node. The outputs at the end of these three stages are shown in Figure 4-3*

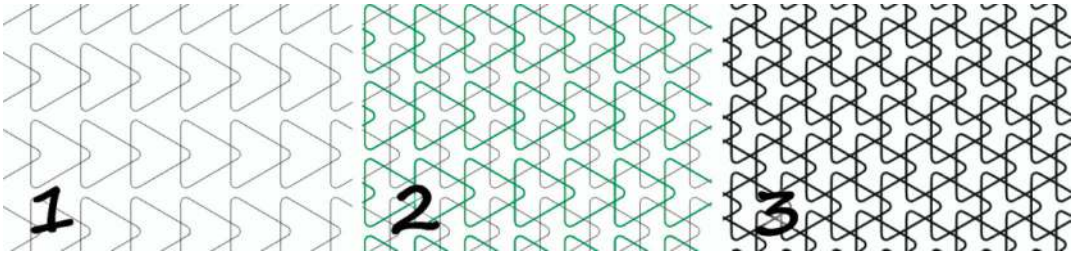


Figure 4-3. *Output of each of the three stages in Figure 4-2. (1) is a Rectangular Array of triangular curves, (2) is (1) moved horizontally by half a cell's size (in green, overlaid over (1)), and (3) is (1) and (2) combined and MultiPiped*

Variation #1: Squares with No Filet, No Rotation

Figure 4-4 shows a variation output by the node graph in Figure 4-2, produced by dialing the Segments (S) input of the Polygon node (1a) to 4 and Filet Radius (Rf) to 0 to create squares with sharp corners. The X input for Scale NU (1c) is reduced from 0.84 to 0.58 to make the array cells narrower.

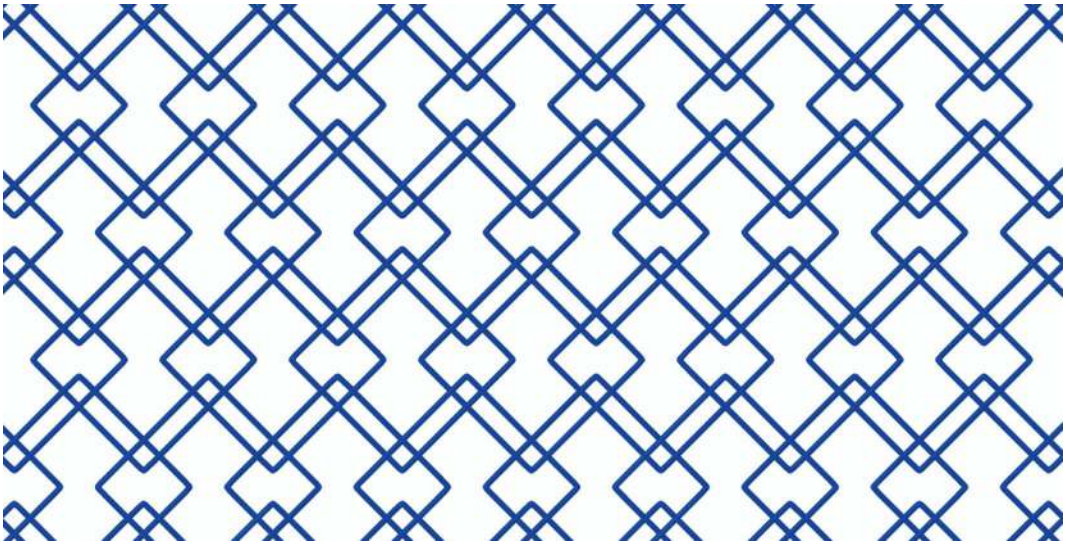


Figure 4-4. *Variation #1 of the graph in Figure 4-2, with the Polygon node (1a)'s Segments (S) set to 4 and Filet Radius (Rf) set to 0 to make squares with crisp corners. The X input for Scale NU (1c) is decreased from 0.84 to 0.58*

Variation #2: Squares with No Filet, ¼ Pi Radian Rotation

We can achieve a drastically different look by rotating the squares in variation #1 on an individual basis, as shown in Figure 4-5. We insert a Transform ➤ Euclidean ➤ Rotate node (r1) between Polygon and Bounding Box to rotate the curve before using it as the array element. The Angle (A) input is set up as multiples of Pi radians with a Number Slider (r3) linked to a Maths ➤ Util ➤ Pi node (r2)—in this case, 0.25 for ¼ Pi radian. Keeping all other parameters the same as variations #1, we create the tessellation shown at the bottom of Figure 4-5, reminiscent of a Japanese shoji.

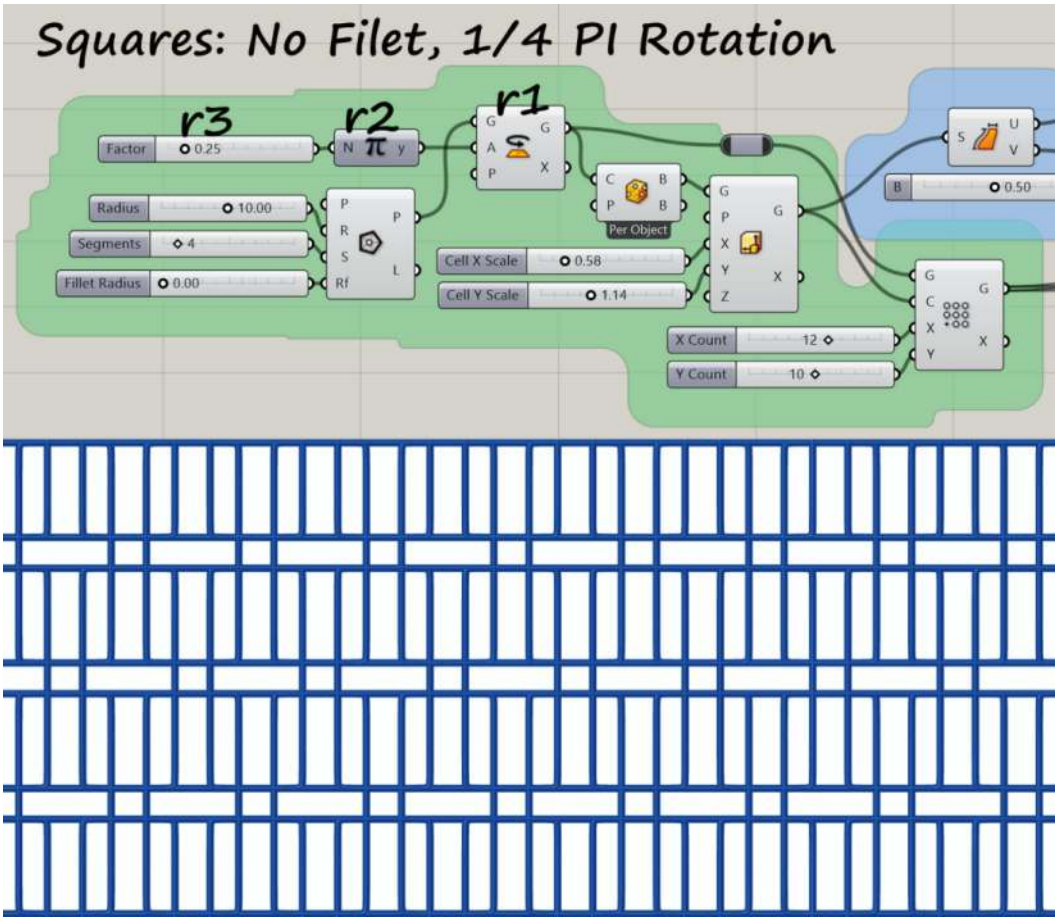


Figure 4-5. *By inserting a Rotate node (r1) with an Angle (A) in a multiple of Pi radians (r2 and r3) between Polygon and Bounding Box, we rotate the square curve before using it as an array element. Rotating each square by ¼ Pi radians while retaining all other parameters from variation #1 gives us the tessellation shown at the bottom, reminiscent of a shoji*

Note A *shoji* is a Japanese style sliding door, window, or room divider made of interlocking lattices of wood or bamboo with a paper covering.

Variation #3: Hexagons with No Filet, No Rotation

We can make an Islamic style tessellation as shown in Figure 4-6, by making one simple change to variation #1—by dialing the Segments (S) input of Polygon (1a) to 6.

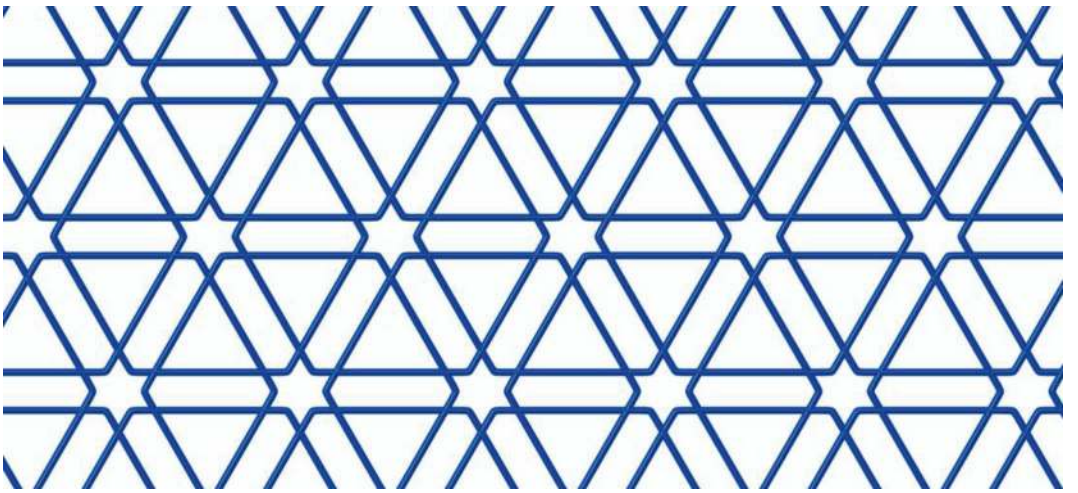


Figure 4-6. *Islamic style tessellation created by making one change to variation #1—by setting the Segments (S) input for Polygon (1a) to 6*

Variation #4: Hexagons with No Filet, ½ Pi Rotation

We'll modify variation #3 to rotate each hexagon by ½ Pi radian, by copying the three-node sequence (r1 to r3) from Figure 4-5 to between the Polygon and Bounding Box nodes and dragging the slider for Pi to 0.5. With this simple change, we produce the geometric tessellation shown in Figure 4-7.

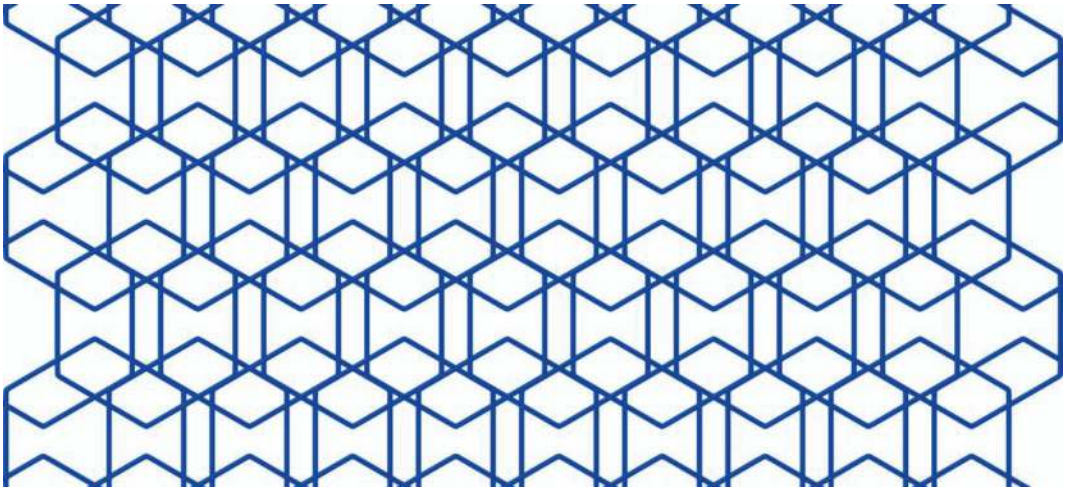


Figure 4-7. *Geometric style tessellation created by rotating each hexagon in variation #3 by ½ Pi radian. The rotation is produced by inserting the three-node sequence from Figure 4-5 (r1 to r3) between Polygon and Bounding Box and dragging the slider for Pi to 0.5*

Variation #5: Hexagons with No Filet, ¼ Pi Radian Rotation

We can keep experimenting with variation #4, by holding all parameters the same, except for dialing the rotation per hexagon up and down, by sliding the Number Slider (r3) for Pi (r2) that controls the Angle (A) input for Rotate (r1) in Figure 4-5. For example, by decreasing the slider from 0.5 to 0.25, therefore rotating each hexagon by ¼ Pi radian instead of ½ Pi radian, we create the tessellation shown in Figure 4-8.

Note You can find the complete node graphs for all five variations (Figures 4-4 to 4-8) in the file `/Ch4/Figure4-4_to_4-8_overlap_polygons_variations.gh` in the downloaded source for this chapter.

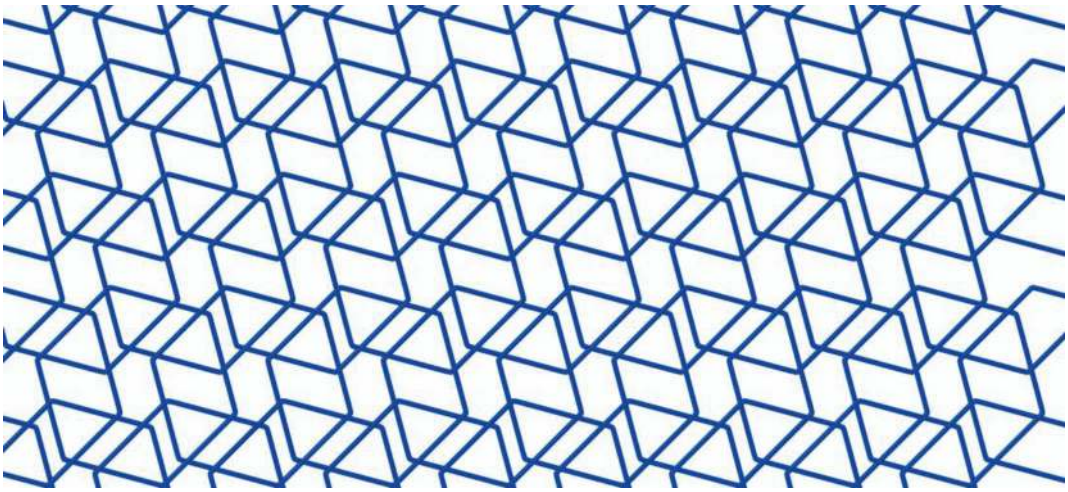


Figure 4-8. *Geometric-style tessellation created by rotating each hexagon in variation #3 by ¼ Pi radian instead of ½ Pi radian*

Ideas for Further Experimentation

So far, we've made variations by

1. Changing the primitive used as the array element (e.g., triangle, square, and hexagon, by setting the Polygon node's Segments (S) input to 3, 4, and 6)
2. Rotating the primitives on a per-shape basis, by inserting the three-node sequence (r1 to r3) from Figure 4-5 between Polygon and Bounding Box

You can continue to experiment by using a different shape per array, for example, triangle for the first array and square for the second array—to do so, you'll duplicate stage 1 in Figure 4-2 and connect the copy to stage 2, so you are able to independently set the Segments (S) input and other parameters for the Polygon node.

Another idea is to rotate an entire array at a time instead of on a per-shape basis. To do so, you'll move the three-node sequence r1 to r3 in Figure 4-5 to *after* the Rectangular Array node (1d), which will rotate the first array in its entirety. You can optionally rotate the second array (produced by stage 2) independently from the first array, by duplicating r1 to r3 and linking them to the output of the Move node (2c) at the end of the second stage.

Construction Method #2: Sliding Points Along a Cell's Edges and Linking Them thru the Cell's Center to Create a Motif

In this section, we'll introduce a second construction method, which is to slide points along an array cell's edges and link the points through the cell's center to create a motif (the "radial lines motif"). We then arrange copies of this motif in a grid to form a tessellation and create variations by sliding the edge points by different amounts. Later, we'll experiment with composing a larger motif by arranging mirrored copies of the single motif.

Figure 4-9 shows the node graph for creating the radial lines motif, with some of the intermediate steps visualized in Figure 4-10. The graph is broken into three stages. In stage 1, we start with a square, then create half a motif (1i in Figure 4-9 and 4-10) by linking its two diagonals (1d), plus sliding one point on each of its edges counterclockwise and linking them diagonally (1h). In stage 2, we mirror the half motif vertically (2c), then overlay it on top of the original to create a motif with *4-fold symmetry* (2e), meaning that each time you rotate the motif 90 degrees, you get the same motif back—this symmetry is needed so we are able to tile the motif both horizontally and vertically across the plane in a seamless way. In stage 3, we perform the tiling by arranging copies of the motif (2e) in a grid such that they are tangent (touching) and use MultiPipe (3d) to turn the curves into a set of SubDs.

Stage 1: Creating a Half Motif by Sliding Points Along the Array Cell's Edges and Linking Them thru Cell Center

Let's take a closer look at each stage. In stage 1, we start by creating a square curve using a Rectangle node (1a) with the same X and Y inputs. We link the Rectangle (1a) to Brep Edges (1b) followed by End Points (1c) to extract its vertices and Polygon Center (1f) to get the Rectangle's center point. We then form the diagonals (1d) by using the center (1f) and the vertices (S output of 1c) as the A and B inputs for the Line node, which creates four lines with the center as the start point and the four vertices as end points.

To tell which diagonal is which, we can link 1d to a Set ➤ List ➤ List Item node (1e), zoom in and click the + button on the lower right three times to create three additional outputs (which will appear as +1, +2, and +3), and link each of the four outputs to a

Params ➤ Geometry ➤ Line node—after that, you can click each Line node to preview the corresponding diagonal in the viewport, which will look like 1e-1 to 1e-4 in Figure 4-10.

Note Here is a summary of where you can find the nodes in stage 1 of Figure 4-9: Curve ➤ Primitive ➤ Rectangle (1a), Surface ➤Analysis ➤Brep Edges (1b), Curve ➤ Analysis ➤ End Points (1c), Curve ➤ Primitive ➤ Line (1d and 1h), Set ➤ List ➤ List Item (1e), Curve ➤ Analysis ➤ Polygon Center (1f), Curve ➤ Analysis ➤ Evaluate Curve (1g), and Params ➤ Geometry ➤ Line (1i).

Continuing with stage 1, we create the sliding points along the Rectangle's four edges, by linking Brep Edges (1b) to Curve ➤ Analysis ➤ Evaluate Curve (1g), which let us input an amount via Parameter (t) to indicate how much to slide a point along each edge. We then right-click the Curve (C) input of Evaluate Curve ➤ Reparametrize, so we can express Parameter (t) as a decimal number between 0 to 1—you can think of this as the percentage to slide the point from the start of the curve in terms of the curve's length. For example, t=0.1 means 10% from the start.

In Figure 4-9, t is set to 0.7—since the Rectangle sits on the XY plane, this translates to each point sliding from *right to left* when viewed from the top. Therefore, the four points sliding in unison with the same t appear as if a square with those four corners is rotating *counterclockwise* (1g in Figure 4-10). We use a Line node (1h) to create line segments from the Rectangle's center (1f) to the four sliding points (1g).

Lastly, we combine the Rectangle's diagonals (1d) with the lines between its center and the sliding points (1h) to create a half motif, by Shift-dragging both to a Params ➤ Geometry ➤ Line node (1i).

Stage 2: Overlaying a Mirrored Copy of the Half Motif on Top of Itself to Make the Final Motif

In stage 2, we mirror the half motif (1i) across the XZ plane (2b) with the Transform ➤ Euclidean ➤ Mirror node (2a), which creates the half motif shown as (2a) in Figure 4-10. We then move the mirrored copy toward +Y using a Move node (2c), so it has the same position as the original half motif (1i). We combine the original and the copy to form the whole motif, by Shift-dragging (1i) and (2c) to a Curve ➤ Util ➤ Join Curves node (2e).

Note Here is a summary of where you can find the nodes in stage 2 of Figure 4-9: Transform ➤ Euclidean ➤ Mirror (2a), Vector ➤Plane ➤XZ Plane (2b), Transform ➤ Euclidean ➤ Move (2c), Vector ➤ Vector ➤ Unit Y (2d), and Curve ➤ Util ➤ Join Curves (2e).

Stage 3: Tiling the Final Motif Using Rectangular Array

In stage 3, we form the tessellation by tiling the motif (2e) with Rectangular Array (3a). To make the motifs tile with no gap, we reuse the Rectangle node (1a) from stage 1 that the motif is based on as the Cell (C) input for Rectangular Array (3a)—note that this wire is hidden in Figure 4-9.

Next, we create a border for the tessellation by linking Rectangular Array (3a) to a Bounding Box node set to right-click ➤ Union Box, to get a box just large enough to enclose the array in its entirety, then a Brep Edges node (3c) to extract the box's edges.

Finally, we Shift-drag both Rectangular Array (3a) and the edges of its bounding box (3c) to a MultiPipe node (3d) to convert the curves in the tessellation to SubD objects.

Note Here is a summary of where you can find the nodes used in stage 3 of Figure 4-9: Transform ➤ Array ➤ Rectangular Array (3a), Surface ➤Primitive ➤Bounding Box (3b, right-click ➤ Union Box), Surface ➤ Analysis ➤ Brep Edges (3c), and Surface ➤ SubD ➤ MultiPipe (3d).

Tip Recall from Chapter 1 that you can hide a wire from view by right-clicking an input ➤ Wire Display ➤ Hidden. The wire will then be shown only when *at least one* of its connected nodes is selected.

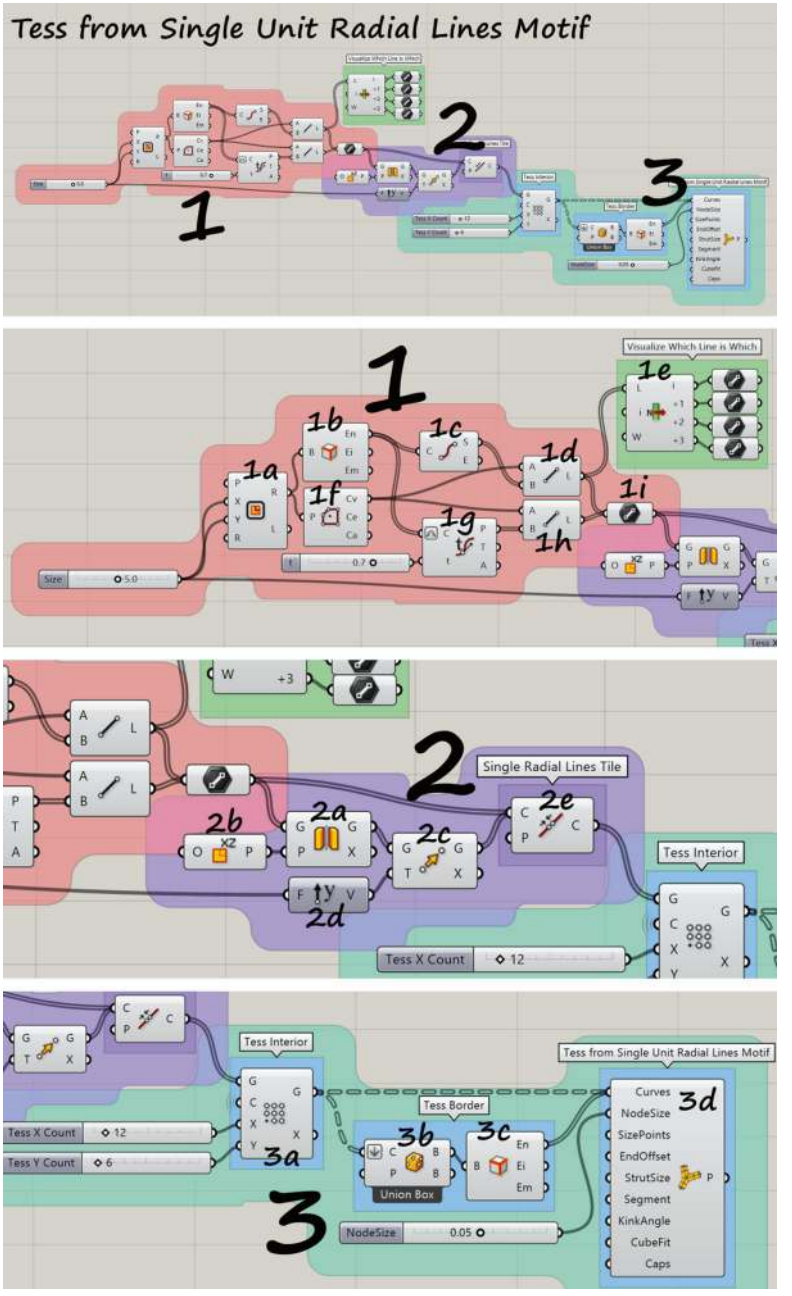


Figure 4-9. *Node graph that produces the tessellation shown in Figure 4-11 from radial lines motifs (2e in Figure 4-10). In stage 1, a half motif is composed from the Rectangle's diagonals and lines between sliding points on its edges. In stage 2, a whole motif is created by overlaying a mirrored copy of the half motif on itself. In stage 3, Rectangular Array (3a) is used to arrange the motif (2e) into a grid*

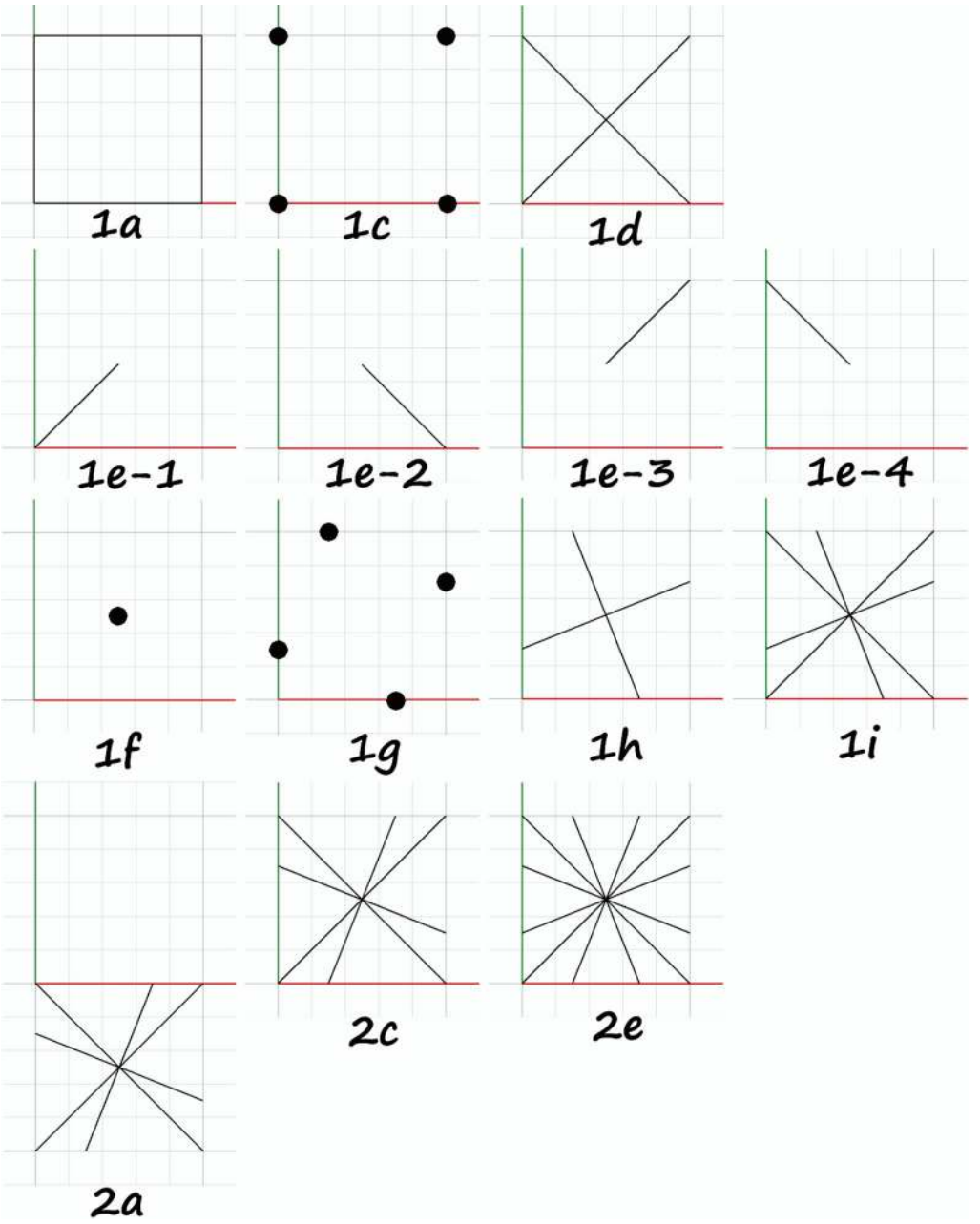


Figure 4-10. *Intermediate steps of the node graph in Figure 4-9 visualized. Note that the points in 1c, 1f, and 1g have been edited so they are easier to see*

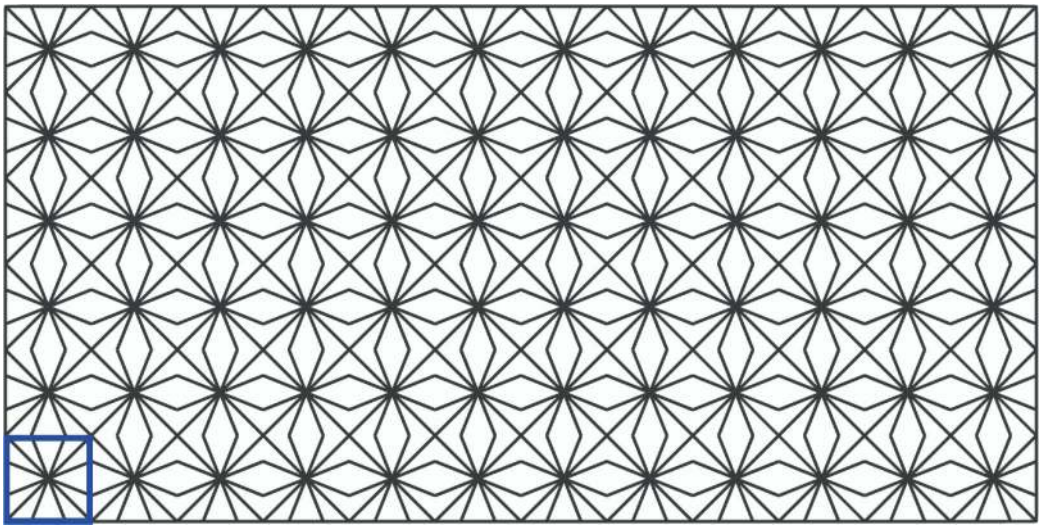


Figure 4-11. *The tessellation produced by the node graph in Figure 4-9. The portion highlighted by the blue square in the lower left corner is the radial lines motif*

Creating Variations by Sliding Edge Points by Different Amounts

Figure 4-12 shows several variations of the tessellation from Figure 4-11 produced by setting the Parameter (t) input for Evaluate Curve (1g) in Figure 4-9 to different values (clockwise, t = 0, t = 0.1, t = 0.4, and t=0.5), hence sliding the points along an array cell's edges by different amounts.

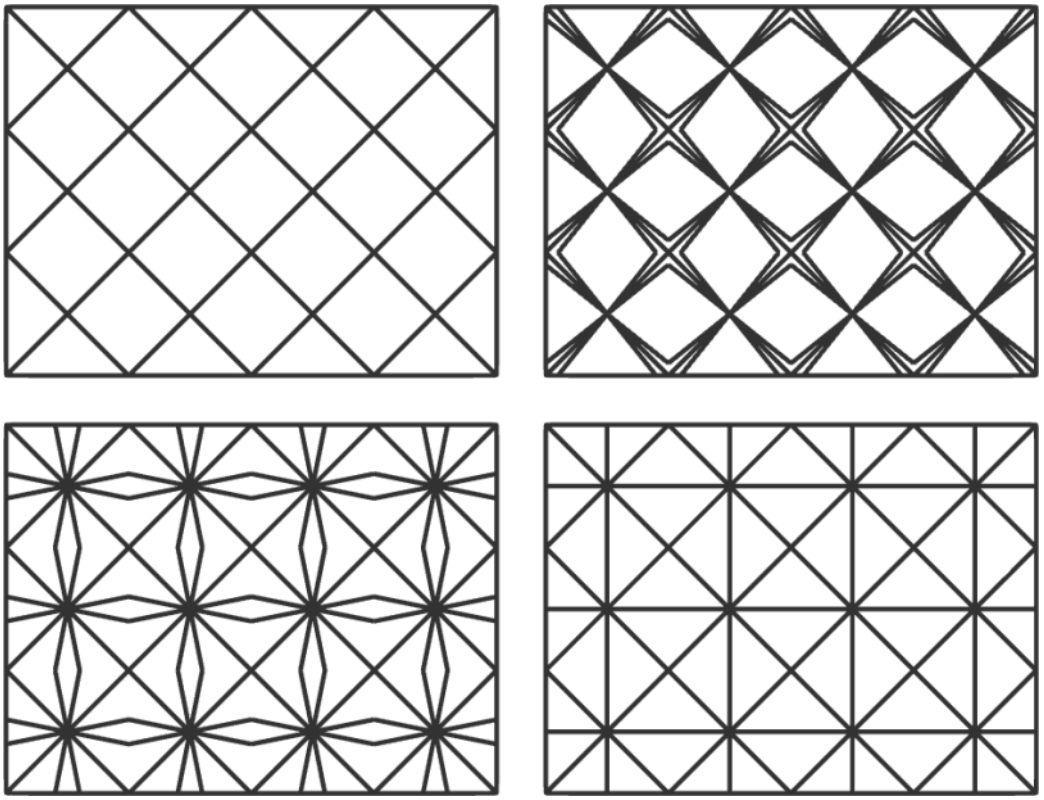


Figure 4-12. *Four variations of the tessellation in Figure 4-11 created by setting Parameter (t) of the Evaluate Curve node (1g) in Figure 4-9 to different values. Clockwise: t=0, t=0.1, t=0.4, and t=0.5*

Ideas for Further Experimentation

Instead of using one sliding point per edge, try using two per edge, or one point on some edges and two points on others. You can also experiment with sliding the points on each pair of opposing edges by a different amount. So far, we've only used square grid cells—try tiling with triangular or hexagonal cells and see what happens.

Composing a Larger Motif from Mirrored Copies of a Single Motif

In the last section, we created a motif with 4-fold symmetry by overlaying a mirrored copy of a half motif on top of itself so it can be tiled seamlessly across both directions on a 2D plane. In this section, we'll create 4-fold symmetry in another way—by composing

a motif from copies of the half-motif mirrored horizontally *and* vertically. To do so, we replace stage 2 in Figure 4-9 with the partial graph shown in Figure 4-13.

In the new stage 2, we start by linking the original half motif (1i) to a Mirror node (2a) with the Plane (P) input left at default (YZ) to create (2a) in Figure 4-14. We then combine (1i) with (2a) using a Join Curves node (2b) to create the top half of the new motif and reflect this half using a second Mirror node (2c) across Plane (P) = XZ to make the bottom half (2c in Figure 4-14). Combining the top (2b) and bottom half (2c) with a second Join Curves node (2d) makes the new motif (2d).

Since the new motif is now twice as tall and wide as the previous motif, we must adjust the Cell (C) input for Rectangular Array (3a) accordingly. Instead of using the Rectangle node (from which 1i is created) verbatim, we use a Scale NU node (2e) to make it twice the size along X and Y before linking it to the Cell (C) input of Rectangular Array (3a). Similarly, we decrease the X and Y counts for Rectangular Array (3a) to half as before, since it now takes half the number of motifs in each direction to make a tessellation the same size. The completed tessellation is shown in Figure 4-15, with the new motif highlighted in the lower left corner.

Note "n-fold symmetry" means that a shape will coincide with itself after being rotated 360/n degrees. For example, 4-fold symmetry means a shape will coincide with itself after a 360/4 = 90 degree rotation.

Here is a summary of where you can find the nodes used in stage 2 of Figure 4-13: Transform ➤ Euclidean ➤ Mirror (2a and 2c), Curve ➤ Util ➤ Join Curves (2b and 2d), and Transform ➤ Affine ➤ Scale NU (2e).

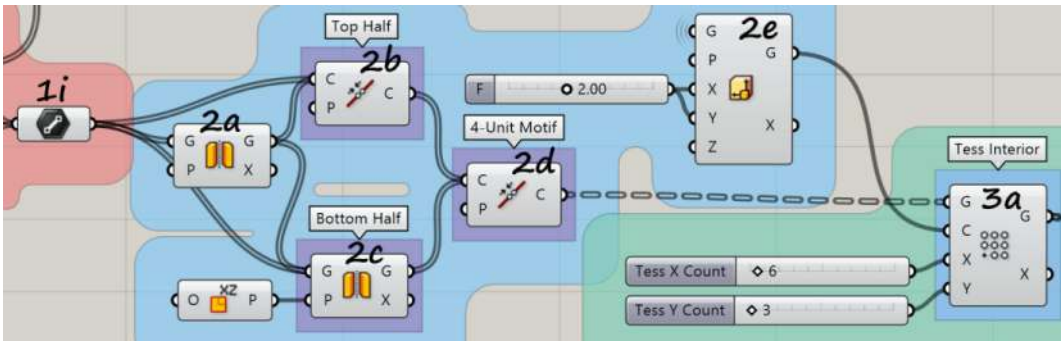


Figure 4-13. *The replacement stage 2 for the node graph in Figure 4-9 to compose a larger motif from copies of half-motifs (1i) mirrored vertically and horizontally. The larger motif produced is shown on the left side of Figure 4-14*

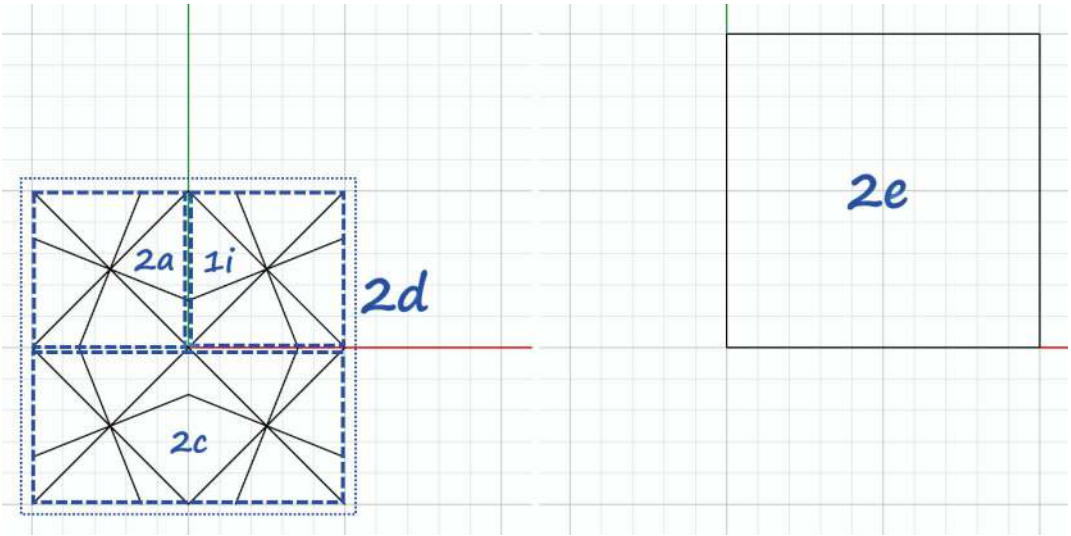


Figure 4-14. *Visualization of the intermediate steps in the replacement stage 2 depicted in Figure 4-13. The left shows the construction of the new motif composed from copies of the original half motif (1i) mirrored both horizontally (2a) and vertically (2c). The right shows the Rectangle node used to create 1i scaled to twice the size along X and Y to be used as the new Cell (C) size for Rectangular Array (3a)*

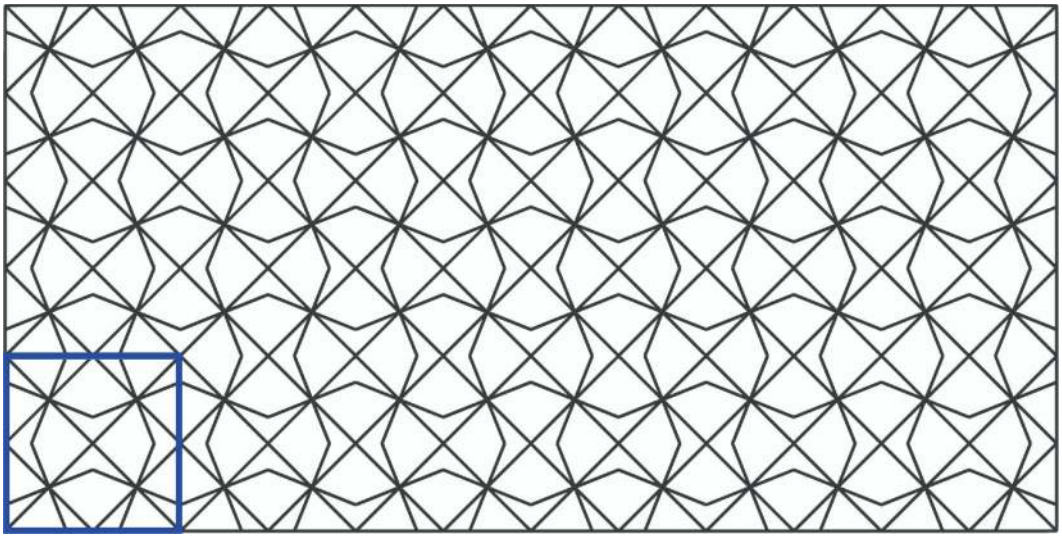


Figure 4-15. *Tessellation produced by the node graph from Figure 4-9 augmented with the new stage 2 in Figure 4-13, which uses a new, larger motif composed from copies of the half-motif mirrored both vertically and horizontally. The new motif is highlighted by the blue square in the lower left corner*

Creating Variations by Sliding Edge Points by Different Amounts

We can create variations with the larger motif by changing the Parameter (t) input for Evaluate Curve (1g) in Figure 4-9, just like what we did with the single-motif tessellation in Figure 4-12. Figure 4-16 shows two examples, with t = 0.1 on the left and t = 0.4 on the right.

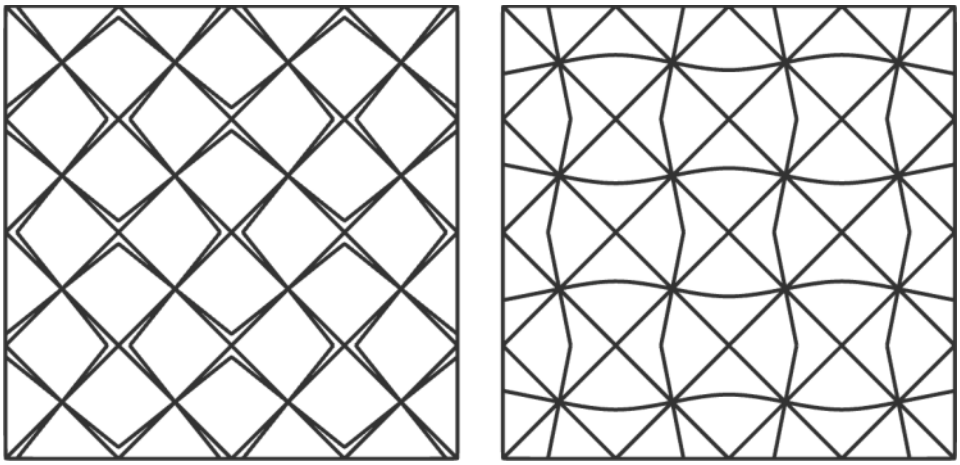


Figure 4-16. *Two variations of Figure 4-15 created by setting Parameter (t) of the Evaluate Curve node (1g) in Figure 4-9 to different values. Left: t=0.1, right: t=0.4*

Construction Method #3: Sliding Points Along Both an Array Cell's Edges and Diagonals

With construction method #2, we form half motifs by combining diagonals of an array cell with lines linking points sliding along its edges. We then overlaid a mirrored copy of the half motif on top of itself to create the final motif ("radial lines"). In this section, we'll also slide points along the cell's edges; however, we won't link these points to the cell center—instead, we'll slide a second set of points along the cell's diagonals, then link the edge sliding points to the diagonal sliding points, to create star-like motifs (the diagonal lines are *not* used as part of the motif).

Figures 4-17 and 4-18 show the node graph for this implementation, with the bird's-eye view and stages 1 and 3 shown in Figure 4-17 and stage 2 shown in Figure 4-18. Outputs of the intermediate steps from stages 2 and 3 that construct the motif are visualized in Figure 4-19.

In stage 1, we create a square curve as the array cell, extract its corner vertices, and slide points along the cell's edges. In stage 2, we slide a second set of points along the cell's two diagonals and form the single star motif by linking the two sets of points (the edge sliders and the diagonal sliders) in an alternating fashion. In stage 3, we arrange mirrored copies of the single star motifs into a larger motif with 4-fold symmetry, so it can be tiled in both directions seamlessly across the XY plane.

Stage 1: Sliding Points Along the Array Cell's Edges

Let's zoom in on stage 1. We use a Rectangle node (1a) as an array cell and Brep Edges (1b) to extract its edges. We then slide points along these edges using Evaluate Curve (1c). The edge sliders will become "peaks" of the single star motif in stage 2. The "valleys" of the star (as in points neighboring each peak) will be points that glide along the cell's diagonals.

To prepare for finding the diagonals in stage 2, we link Brep Edges (1b) to End Points (1d) at the end of stage 1 to find the pairs of end points of *every edge*. End Points (1d) has two outputs, Start (S) and End (E), that provide the list of start and list of end points, respectively; either output gives you the Rectangle's four vertices, although in slightly different order. For example, if the four edges have the pair of end points (#1, #2), (#2, #3), (#3, #4), and (#4, #1), then Start (S) is (#1, #2, #3, #4) and End (E) is (#2, #3, #4, #1). We opt for Start (S) since its vertex indices follow the natural order.

Note Here is a summary of where you can find the nodes in stage 1 of Figure 4-17: Curve ➤ Primitive ➤ Rectangle (1a), Surface ➤Analysis ➤Brep Edges (1b), Curve ➤ Analysis ➤ Evaluate Curve (1c), and Curve ➤ Analysis ➤ End Points (1d),

---

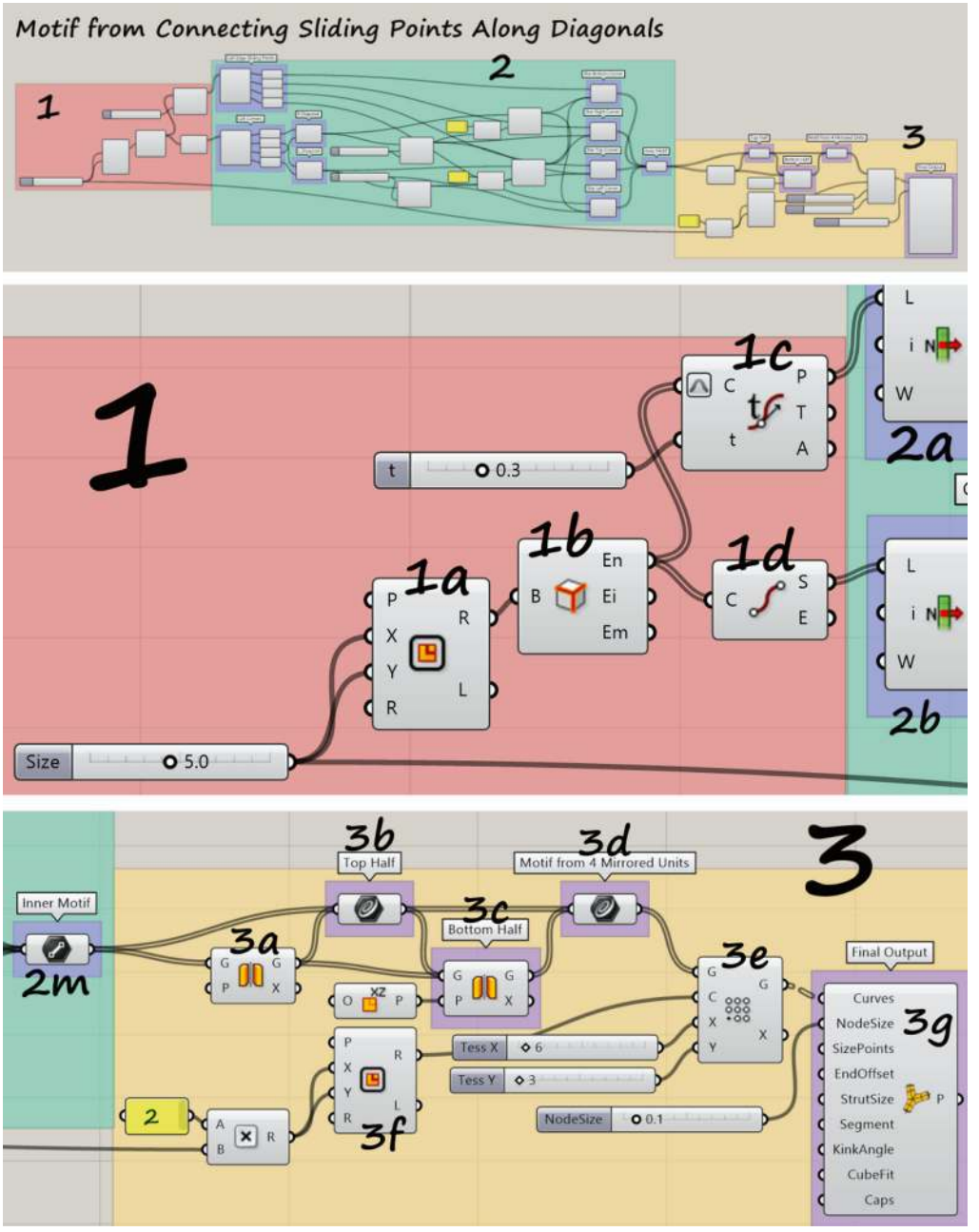


Figure 4-17. *Node graph that tessellates motifs based on linking points that slide along an array cell's edges as well as diagonals. The top shows a bird's-eye view of the entire node graph, and the middle and bottom show stages 1 and 3 of the graph zoomed in*

Stage 2: Sliding Points Along Cell Diagonals and Linking Edge Sliders and Diagonal Sliders in Alternating Pattern to Form Single Star Motif

In stage 2, we slide two points along each cell diagonal and compose the single star motif by linking edge sliding points to the diagonal sliding points in an alternating pattern—edge point (peak), diagonal point (valley), next edge point (next peak), and so on.

To find the right leaning diagonal, we link the cell vertices (1d) to a List Item node (2b) with three additional outputs and each output to a separate Point node to access the vertices one by one. After that, we link the first and third Point nodes to a Line node (2c) to get the right-leaning diagonal and the second and fourth Point nodes to a separate Line (2f) node to get the left-leaning diagonal.

Next, we glide two points along each diagonal from its end points in a symmetrical way. Take the right-leaning diagonal as an example; we link (2c) to a Evaluate Curve node (2d) to create the first sliding point, offsetting from the diagonal's start point at the upper-right corner. To create the second sliding point, we link (2c) to a second Evaluate Curve node (2e) and use "one minus" the first Evaluate Curve node (2d)'s t value.

For example, if 2d has t = 0.3, then 2e has t = 1 – 0.3 = 0.7. The first point offsets 30% inward from one end of the diagonal, while the second point ends up 30% inward from the opposite end. Since Evaluate Curve (2e) will in reality offset the second point from the same end of the diagonal as the first point, we use the "one minus" t value to land the point at the correct offset. We follow similar steps in deriving two sliding points (2g and 2h) along the left-leaning diagonal (2f). The creation process for the sliding diagonal points is shown in Figure 4-19.

Before leaving stage 2, we use four Line nodes (2i, 2j, 2k, and 2l) to construct the four peaks of the single star and Shift-drag them to a Curve node (2m) to combine. Each line node has one point sliding on a Rectangle edge as its A input and one sliding point from each diagonal as its B input (A is the peak, and B are the two valleys surrounding it). You can visualize which Line node corresponds to which peak of the star in Figure 4-19.

Note Here is a summary of where you can find the nodes in stage 2 of Figure 4-18: Sets ➤ List ➤ List Item (2a and 2b), Params ➤ Geometry ➤ Point, Curve ➤ Primitive ➤ Line (2c, 2f, 2i, 2j, 2k, and 2l), Curve ➤ Analysis ➤ Evaluate Curve (2d, 2e, 2g, and 2h), and Params ➤ Geometry ➤ Line (2m).

Recall that you can add an output to a List Item node by zooming in and pressing the "+" button that appears in the lower right corner. Pressing the "+" button repeatedly will create multiple additional outputs (+1, +2, ..., and so on). The first (original) output gives you the item at Index = i, the second output (+1) the item at i+1, third output (+2) the item at i+2, and so on.

Stage 3: Arranging Mirrored Copies of the Single Star into the Final Motif and Tiling the XY Plane

In stage 3 (Figure 4-17), we compose the final motif by arranging four mirrored copies of the single star motif from stage 2 and tile the final motif across the XY plane to create the tessellation.

We reflect the single star (2m) horizontally using a Mirror node (3a) with the default Plane (P) = YZ, then combine the original with the mirrored copy using a Curve node (3b) to form the top half of the final motif.

We then create the bottom half by reflecting the top half vertically using a second Mirror node (3c) with Plane (P) = XZ. Shift-dragging the top half (3b) and bottom half (3c) to a Curve node (3d) combines them into the final motif. The process for assembling the final motif is shown in Figure 4-19.

Finally, we tessellate by tiling the final motif (3d) across the XY plane. To do so, we link (3d) to the Geometry (G) input of a Rectangular Array node (3e), and for its Cell (C) input, we create a new Rectangle (3f) with twice the X and Y of the initial Rectangle (1a), since the final motif is four times the size of the single star. To finish up, we convert the tessellated curves to SubD using the MultiPipe node (3g), which produces the tessellation shown in Figure 4-20.

Note Here is a summary of where you can find the nodes used in stage 3 of Figure 4-17: Transform ➤ Euclidean ➤ Mirror (3a and 3c), Params ➤ Geometry ➤ Curve (3b and 3d), Transform ➤ Array ➤ Rectangular Array (3e), Curve ➤ Primitive ➤ Rectangle (3f), and Surface ➤ SubD ➤ MultiPipe (3g).

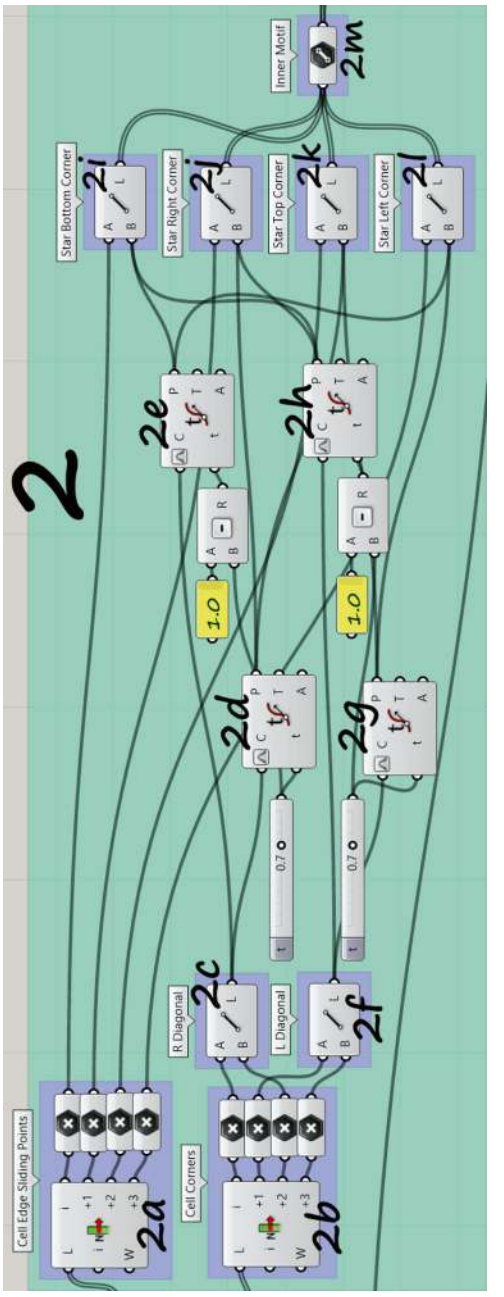


Figure 4-18. *Stage 2 of the graph from Figure 4-17, where we form a single motif by sliding points along an array cell's diagonals and linking them to points sliding on the cell's edges*

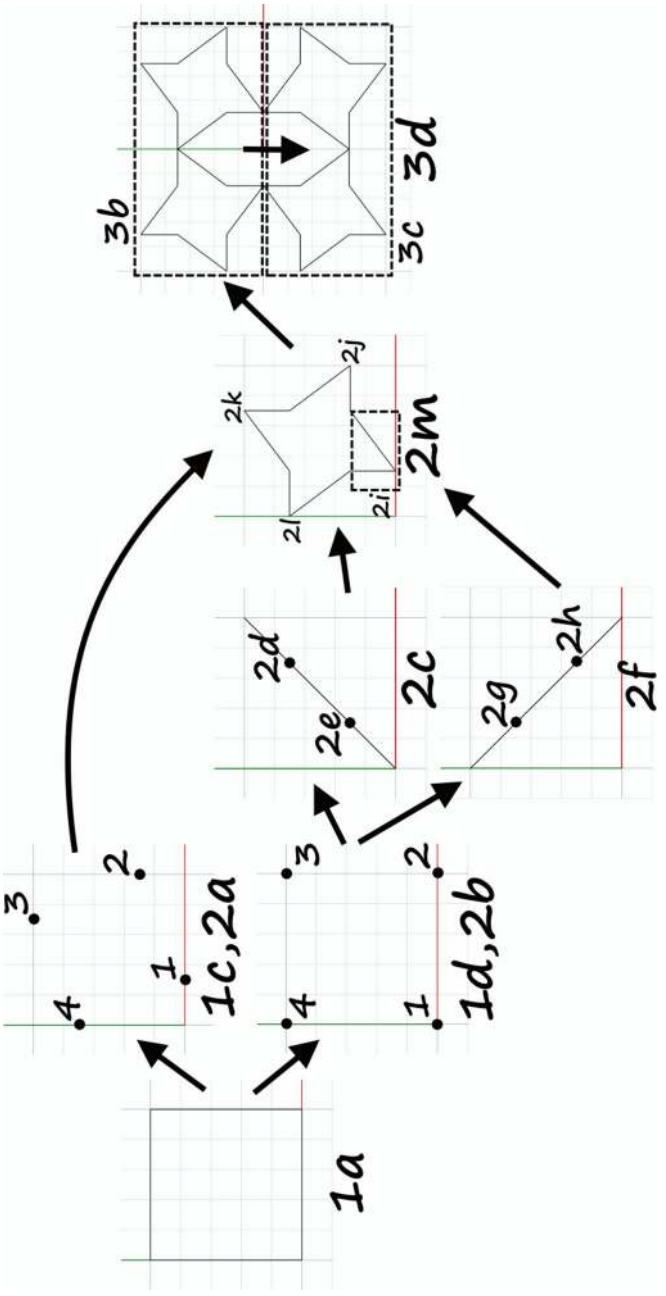


Figure 4-19. *Intermediate steps of the node graph in Figures 4-17 and 4-18 visualized*

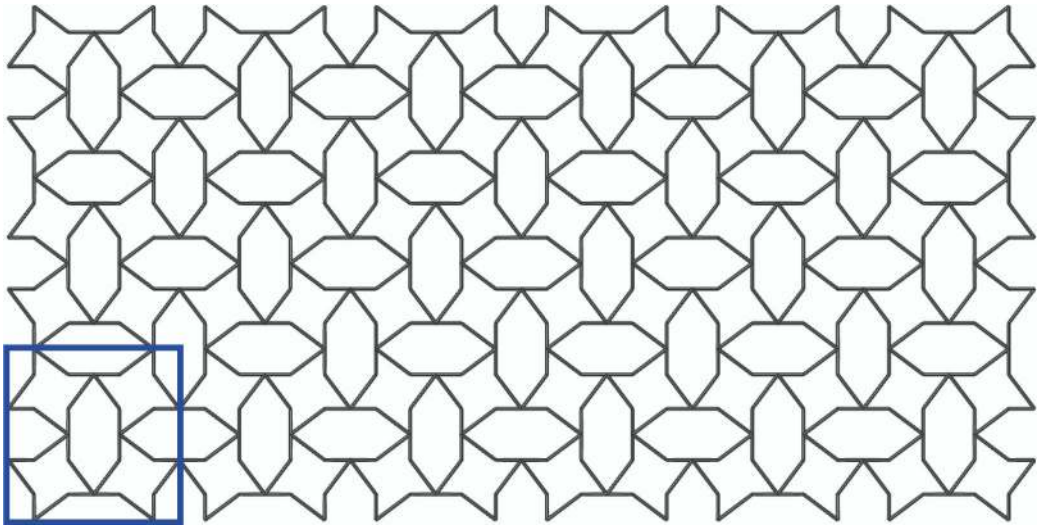


Figure 4-20. *Tessellation produced by the node graph in Figures 4-17 and 4-18. The repeating motif is highlighted by the blue square at the lower left corner*

Creating Variations by Sliding Edge Points and Points Along Diagonals by Different Amounts

Let's create some variations. Figure 4-21 shows six possibilities produced by changing the following three parameters:

1. The amount the edge points slide (Parameter (t) input for 1c in Figure 4-17)
2. The amount the points on the right-leaning diagonal slide (t for 2d)
3. The amount the points on the cell's left-leaning diagonals slide (t for 2g)

These changes will distort the shape of the star in different ways, and in some cases, causing the sides of the star to cave in and overlap one another, creating some interesting effects. The six sets of t values corresponding to the six variations in Figure 4-21 are summarized in Table 4-1.

Note that even though *two* Evaluate Curve nodes are used *per diagonal* to control two sliding points (2d and 2e for right-leaning and 2g and 2h for left-learning), we only set the Parameter (t) value for the first node (the node offsetting the point relative to the

corner vertex with the smaller index)—this is because we want symmetry in how the two points are displaced from the diagonal's end points. Therefore, the t value for the second Evaluate Curve node is automatically calculated as "one minus" the t value for the first node. For example, if t = 0.3 for the first point, then t = 1.0 - 0.3 = 0.7 for the second point.

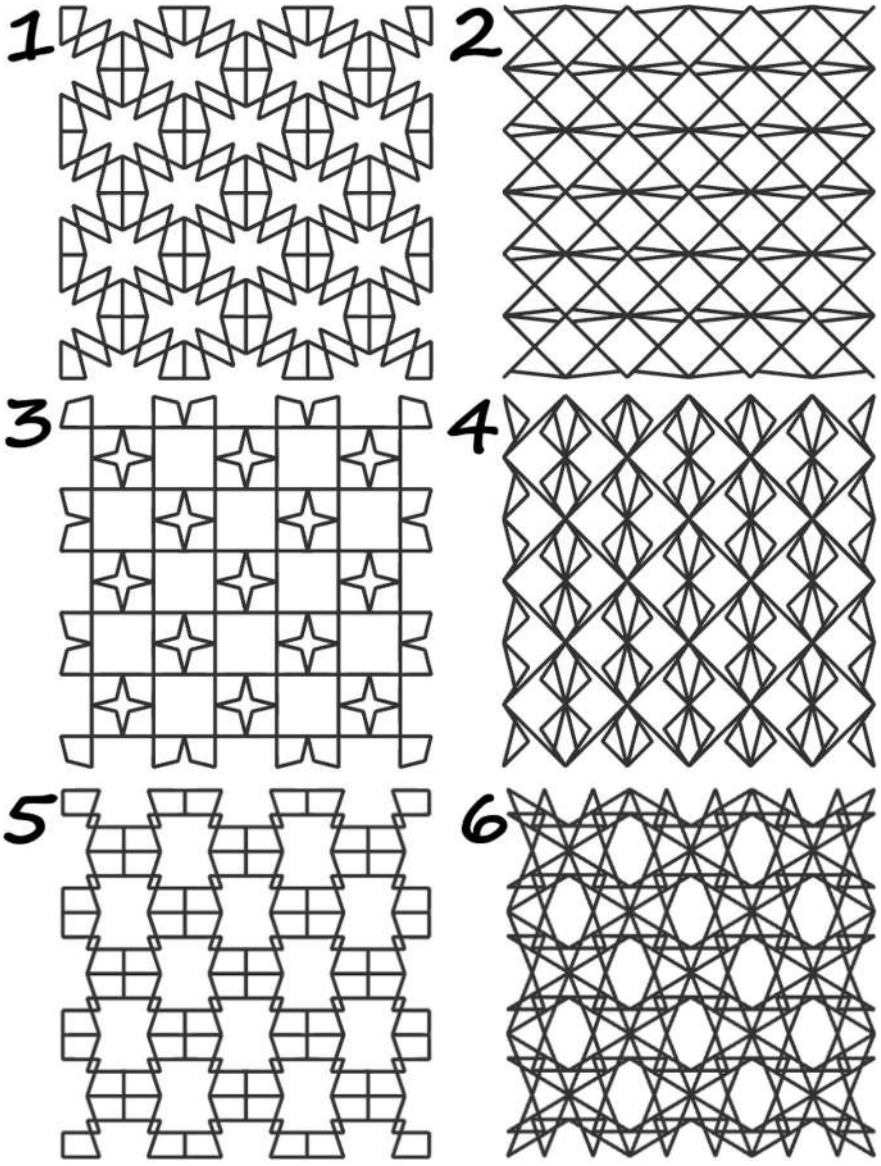


Figure 4-21. *Six different tessellations produced by the node graph in Figures 4-17 and 4-18, by setting different t values for sliding points along cell edges and diagonals, which are summarized in Table 4-1*

Table 4-1 *Summary of t values for sliding points along cell edges and diagonals that produce the six variations shown in Figure 4-21*

Variation #	t for array cell edges	t for array cell right-leaning diagonal	t for array cell left-leaning diagonal
1	0.4	1.0	0.2
2	1.0	0.9	0.5
3	0.5	0.9	0.5
4	0.0	0.6	0.2
5	0.6	1.0	0.4
6	0.6	0.0	0.4

Note If you are generating tessellations for the purpose of 3D printing, you may need to resolve overlaps between motifs (e.g., by adding vertices at intersections). The type of editing required will depend on your slicing software. Just something to keep in mind.

Ideas for Further Experimentation

Instead of rectangular array cells, try using different shapes like triangles, hexagons, etc. You can create closed curves of different shapes using the Curve ➤ Primitive ➤ Polygon node, by setting the Segments (S) input to the desired number of sides, for example, three for a triangle.

We created a motif with 4-fold symmetry by arranging four mirrored copies of the single star motif in a square, so that it can be tiled seamlessly across both the X and Y directions in 2D. Can you think of another way to create a motif with 4-fold symmetry? Can you use a similar method as our approach with the first "radial lines motif," by overlaying a mirrored copy of the star motif on top of itself?

Another idea to try is linking some of the opposing pairs of points in the motif. For example, linking the opposing peaks or opposing valleys of the star through its center.

So far, we've omitted the array cell borders in our design. It would create a different look if you added them back in (i.e., so the star is enclosed by a square). Following this trend of thought, you could also try adding the array cell's diagonals, or only one of the diagonals (all the left-leaning diagonals or all the right-leaning diagonals).

Construction Method #4: Using Maelstrom to Swirl the Radial Lines Motif from Method #2

In this section, we'll introduce a construction method that is a literal "twist" of method #2—we use the Transform ➤ Morph ➤ Maelstrom node to swirl the radial lines half motif, before overlaying a mirrored copy on top of itself, to create the organic looking motif shown in Figure 4-25 that is reminiscent of a flower. The node graph for this method is shown in Figures 4-22 and 4-23, with 4-22 showing the bird's-eye view and first stage and 4-23 showing stages 2 and 3.

In stage 1, we create a half motif by combining array cell diagonals (1e) with lines linking cell edge sliding points (1g). We then swirl the lines in the half motif with the Maelstrom node (1h). In stage 2, we compose the final motif (2c) by overlaying a vertically mirrored copy of the half motif from stage 1 on top of itself. In stage 3, we use Rectangular Array (3a) to tile the final motif (2c) in 2D and add a border to create the tessellation in Figure 4-25.

Stage 1: Creating a Half Motif by Combining Diagonals with Lines Linking Sliding Points on Cell Edges and "Swirling" the Lines

Let's take a closer look at stage 1 (Figure 4-22). We start with a Rectangle node (1a) as the array cell, use Brep Edges (1b) to extract its edges, and End Points (1c) to get the edges' start and end points, which are also the cell's corner vertices.

We then use Polygon Center (1d) to find the Rectangle's center and link it to the Rectangle's corners with the Line node (1e) to create the diagonal lines. Like what we've done with methods #2 and #3, we use Evaluate Curve (1f) to slide points along the cell's edges (1b) then link the cell center (1d) to these points with a second Line (1g) node.

The diagonals (1e) in conjunction with the edge sliding lines (1g) become input geometry for Maelstrom (1h) to create a swirling pattern. We use the XY Plane centering at the same position as the initial Rectangle as the Plane (P) input for Maelstrom (1h) and two decimal numbers as the R0 and R1 values—these two numbers define the portion of the input geometry that will get "swirled" as an inner and outer radii.

The closer R0 and R1 are in value, the narrower their range, therefore the smaller the area in the input geometry that gets swirled. You might think that R0 must be smaller than or equal to R1; however, this is not the case—later in this section, we'll experiment with various R0 and R1 values that will produce some surprising variations, sometimes with R0 larger than R1. The intermediate steps from stage 1 are visualized in Figure 4-24.

Note Here is a summary of where you can find the nodes used in stage 1 of Figure 4-22: Curve ➤ Primitive ➤ Rectangle (1a), Surface ➤Analysis ➤Brep Edges (1b), Curve ➤ Analysis ➤ End Points (1c), Curve ➤ Analysis ➤ Polygon Center (1d), Curve ➤ Primitive ➤ Line (1e and 1g), Curve ➤ Analysis ➤ Evaluate Curve (1f), and Transform ➤ Morph ➤ Maelstrom (1h).

Stage 2: Overlaying Mirrored Copy of the Half Motif on Top of Itself to Create the Final Motif

We'll now zoom in on stage 2 (Figure 4-23). We use a Mirror node (2a) with Plane (P) = XZ to reflect the half motif (1h) vertically, then move it along +Y with the Move node (2b) by one cell's height, so it lands on the same position as the original half motif (1h)—essentially overlaying it. We then combine the original with the mirrored copy by Shift-dragging both to Join Curves (2c) to create the final motif. The outputs for the intermediate steps in this stage are visualized in Figure 4-24.

Note Here is a summary of where you can find the nodes used in stage 2 of Figure 4-23: Transform ➤ Euclidean ➤ Mirror (2a), Transform ➤ Euclidean ➤ Move (2b), and Curve ➤ Util ➤ Join Curves (2c).

Stage 3: Tiling the Final Motif on the XY Plane with Rectangular Array

In stage 3, we create the tessellation by repeating the motif (2c) in a 2D grid using Rectangular Array (3a), with the original Rectangle node (1a) defining the array cell as the Cell (C) input, so the motifs tile without any gap. We find the smallest Box enclosing the array using Bounding Box ➤ right-click Union Box (3b), then extract the Box's edges using Brep Edges (3c) to use as a simple border for the tessellation. The border curves (3c) and the tessellation (3a) are Shift-dragged to a Curve node (3d) to combine into the final output for the graph, which is shown in Figure 4-25.

Note Here is a summary of where you can find the nodes used in stage 3 of Figure 4-23: Transform ➤ Array ➤ Rectangular Array (3a), Surface ➤Primitive ➤Bounding Box (3b, right-click ➤ Union Box), Surface ➤ Analysis ➤ Brep Edges (3c), and Params ➤ Geometry ➤ Curve (3d).

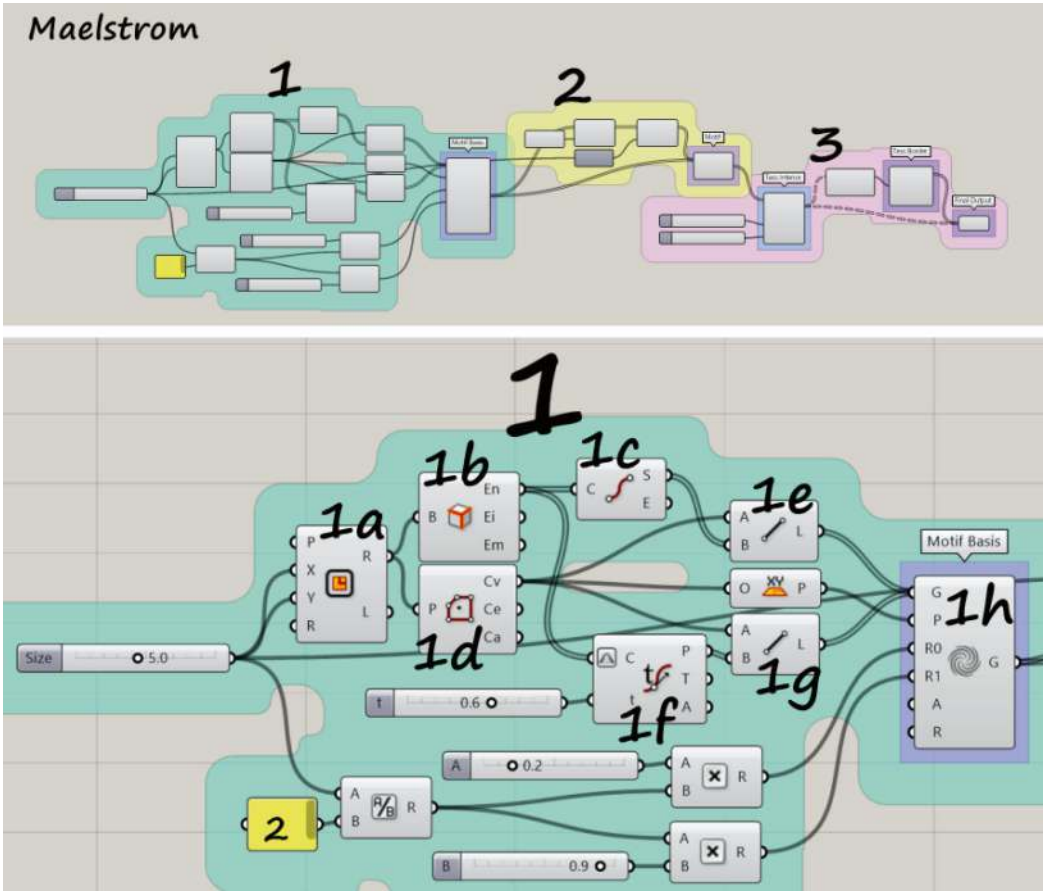


Figure 4-22. *Node graph that uses Maelstrom (1h) to swirl the radial lines motif from method #2. Top: Bird's-eye view of the entire graph. Bottom: Stage 1 zoomed in, where we form the array cell's diagonals (1e) and combine them with lines linking points sliding on the cell's edges (1g), before swirling them (1h). These steps are visualized in Figure 4-24*

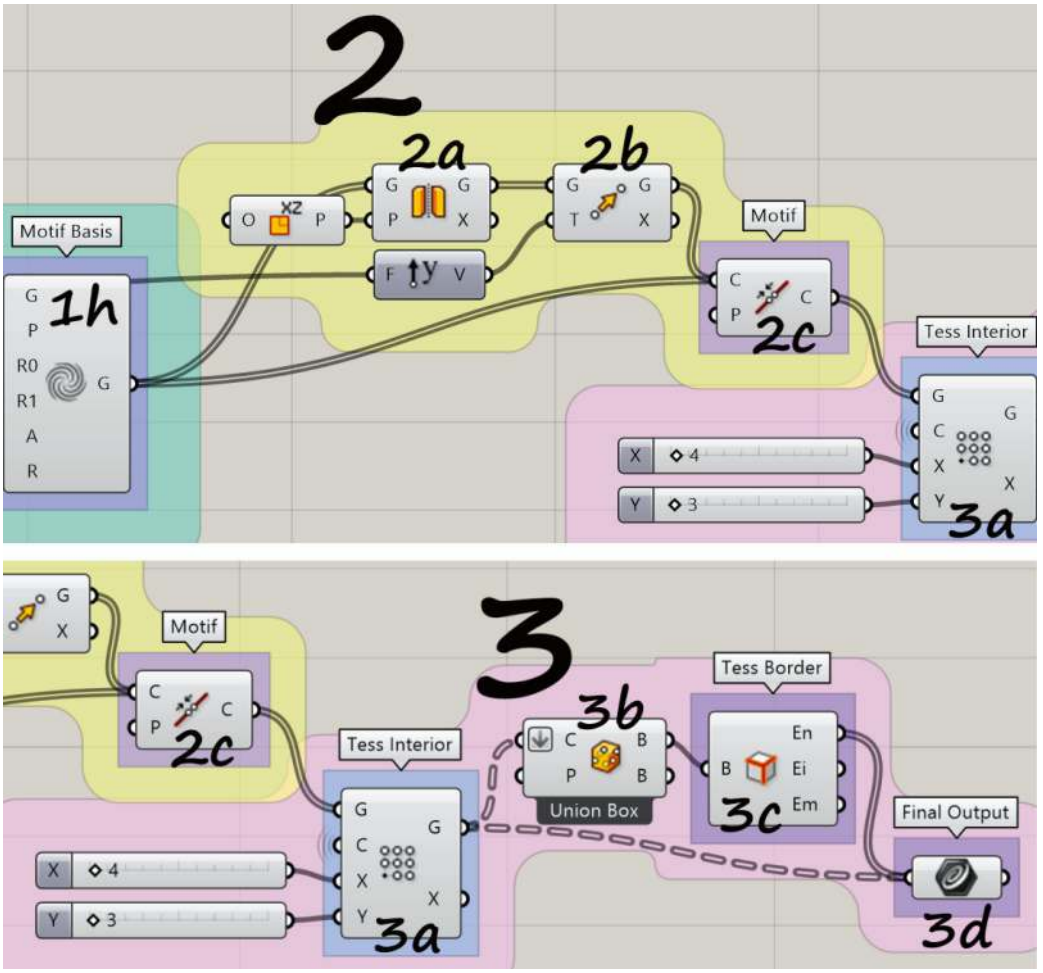


Figure 4-23. *Stages 2 (top) and 3 (bottom) of the node graph from Figure 4-22, where we swirl the radial lines motif from method #2 and tessellating it across the XY plane with Rectangular Array (3a)*

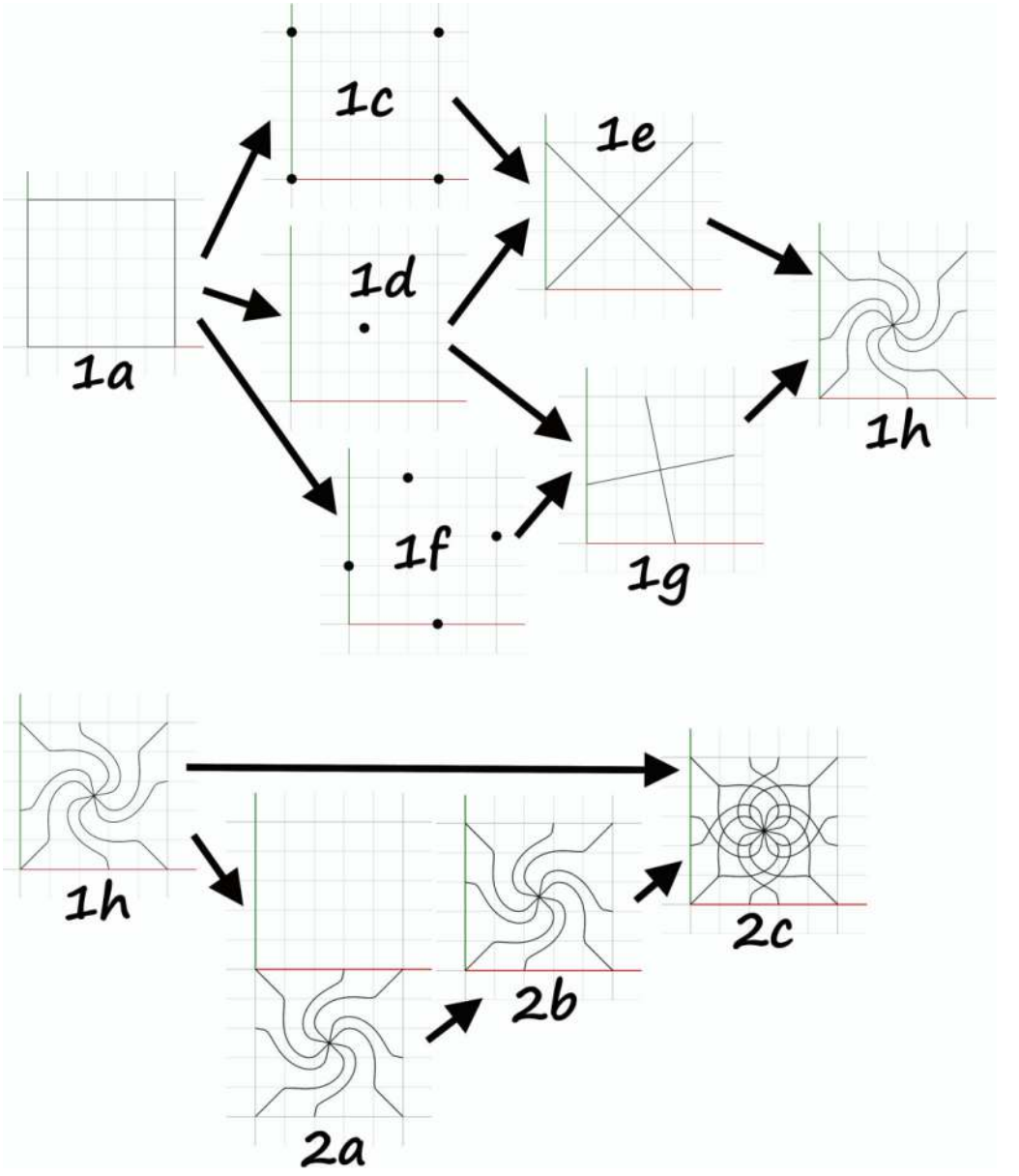


Figure 4-24. *Visualizing intermediate steps of the node graph from Figures 4-22 and 4-23. Top: In stage 1, we form array cell diagonals (1e), combine them with lines linking points sliding on the cell's edges (1g), and swirl them with Maelstrom (1h) to create a half motif. Bottom: In stage 2, we overlay a vertically mirrored copy (2a) of the half motif on top of the original (1h) to create the final motif (2c)*

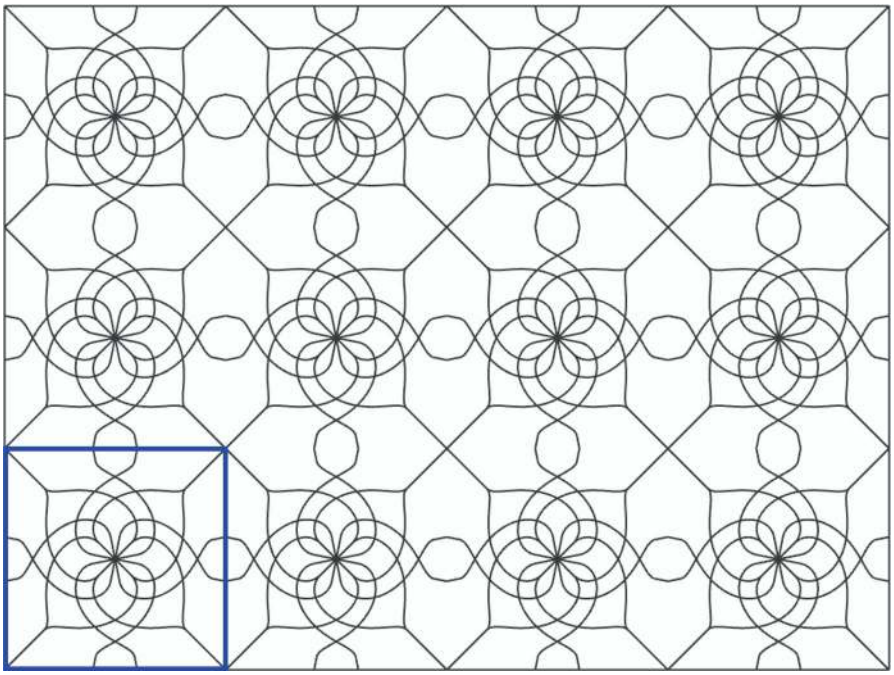


Figure 4-25. *Tessellation produced by the node graph from Figures 4-22 and 4-23. The repeating motif is highlighted by the blue square in the lower left corner*

Creating Variations by Sliding Edge Points by Different Amounts and Using Different R0 and R1 Values for Maelstrom

We can create some interesting variations of Figure 4-25 by changing

1. Parameter (t) value for Evaluate Curve (1f in Figure 4-22), which controls how much points slide along the array cell's edges
2. The R0 and R1 values for Maelstrom (1h in Figure 4-22), which control how much the motif gets "swirled"

Recall that R0 and R1 are two decimal numbers that define the inner and outer radii framing the area of the input geometry that Maelstrom will whirl; therefore, the further the two values are apart, the larger the portion of the motif that gets whirled. It may seem that R1 should be equal to or larger than R0. However, Grasshopper will let you set a R1 value smaller than R0, which sort of "inverts" the pattern. Figure 4-26 shows six very different variations produced this way, with their corresponding t, R0, and R1 values summarized in Table 4-2.

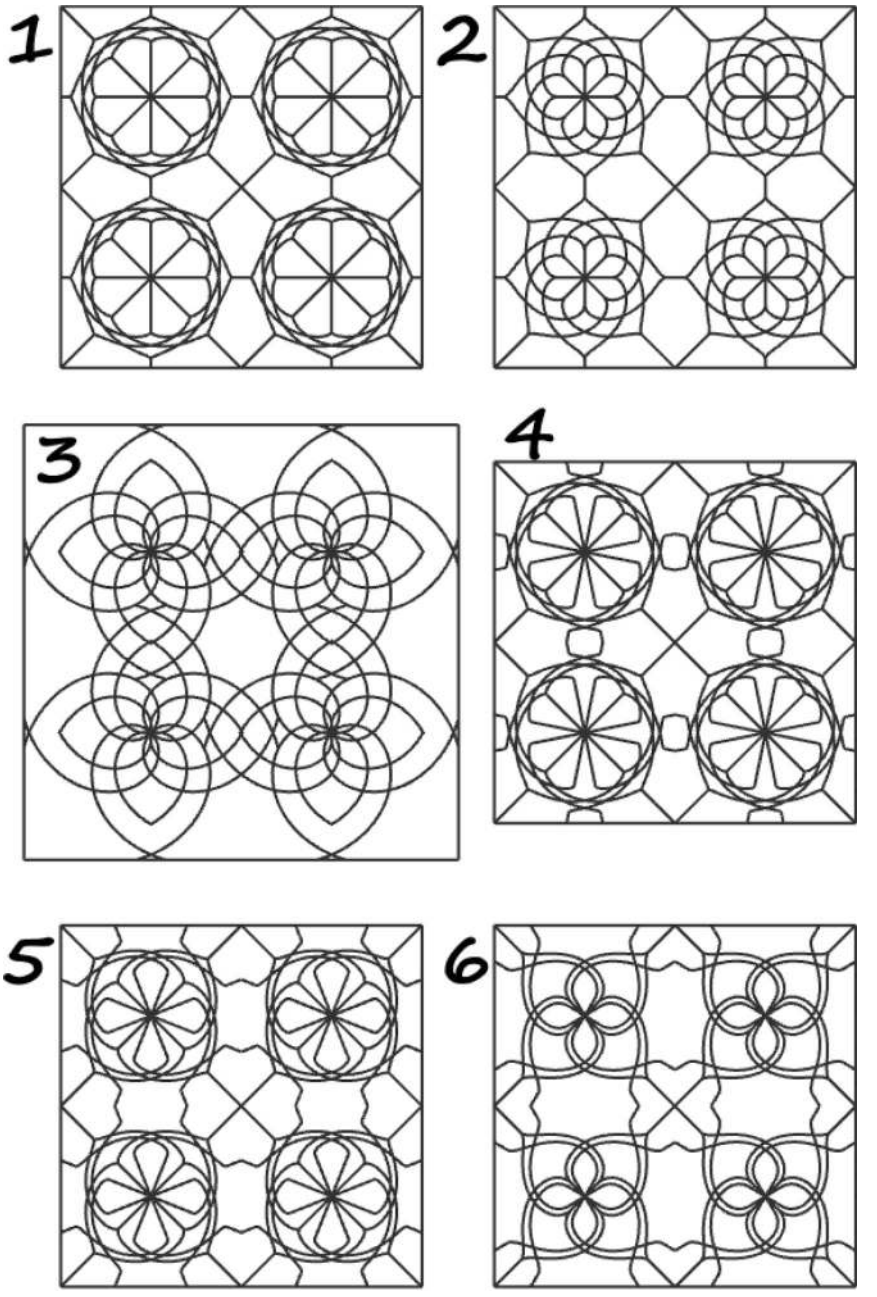


Figure 4-26. *Six variations of Figure 4-25, produced by changing t for sliding cell edges points (1f in Figure 4-22) and the R0, R1 values for the Maelstrom node (1h in Figure 4-22). The t, R0, and R1 values are summarized in Table 4-2*

Table 4-2 *Summary of t values for sliding points on cell edges and R0, R1 values for Maelstrom that produce the six variations in Figure 4-26*

Variation #	t for sliding points on array cell edges (1f in Figure 4-22)	R0 for Maelstrom (1h in Figure 4-22)	R1 for Maelstrom (1h in Figure 4-22)
1	0.5	0.6	0.9
2	0.5	0.3	0.9
3	0.4	0.9	0.9
4	0.4	0.9	0.6
5	0.7	0.9	0.5
6	0.8	1.0	0.2

Ideas for Further Experimentation

So far, we've only created motifs from rectangular array cells—what if you used other shapes, like triangles or hexagons? You can use Curve ➤ Primitive ➤ Polygon to create curves in different shapes, by setting the Segments (S) input to the desired number of sides.

We combined diagonals with lines linking points sliding on the cell edges, before whirling them with the Maelstrom node—what if you used Maelstrom only on the diagonals or only on the lines linking the edge sliding points?

Can you try composing a larger motif by arranging copies of the single swirling motif?

Summary

In this chapter, you learned four different methods for constructing tessellations. The first method creates several 2D arrays of primitive shapes (triangles, rectangles, etc.) then stacks them on top of one another with offsets. Interesting patterns can be created by varying the distances between shapes from layer to layer, rotating shapes within a layer, or changing the relative offsets between layers.

The second method slides points along an array cell's edges, links them through the cell's center, and combines those lines with the cell's diagonals to create a half motif. A mirrored copy of the half motif is then overlaid on top of the original to form a final motif

with 4-fold symmetry, for the purpose of seamlessly tiling in both directions across a 2D plane. We discussed a second way for creating a motif with 4-fold symmetry, which is to arrange four mirrored copies of the half motif in a square pattern to compose a larger motif. Lastly, we experimented with sliding the edge points in different amounts to create variations.

The third method slides points not only along a cell's edges, but also along its diagonals. A single star motif is formed by connecting the edge sliders with the diagonal sliders in an alternating pattern. To obtain a final motif with 4-fold symmetry, we compose a larger motif from four mirrored copies of the single star motif. We created many variations of this tessellation by varying both the amount the points slide along the cell edges and the amount the points slide along the cell diagonals.

The fourth and final method makes a literal twist on the second method, by creating radial lines from edge sliding points and diagonals then swirling them with the Maelstrom node to create a half motif. An organic looking final motif resembling a flower with 4-fold symmetry is then created by overlaying a mirrored copy of the half motif on top of itself. We experimented with creating variations by using different values for sliding edge points, as well as changing R0 and R1 values for Maelstrom, which control the inner and outer radii defining the portion of the array cell that gets whirled.

CHAPTER 5

Loops and Spirals

In this chapter, we'll work through three workshops centered on the theme of loops and spirals. The first workshop introduces you to Möbius strips, an interesting surface with a half-twist that always takes you back to where you start. We'll create Möbius surfaces in Grasshopper in three different ways—using the Mobius Surface node from LunchBox, using Grasshopper's built-in Twist and Flow nodes, and based on math formulas implemented with Expression or Python 3 Script nodes. The second workshop shows you how to build a shell-like surface based on a logarithmic spiral curve. The third workshop teaches you how to build a parametric spiral staircase generator from scratch.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch5` folder, where you'll find `*.gh` files named after the figures in which they appear, along with a short description. For example, `Figure5-2_to_5-3_mobius_with_LunchBox.gh`. The sample files have been tested in Rhino 9 on Windows. As you work through the chapter, build your own node graphs by following the steps described in each section, or start with the downloaded `*.gh` files and experiment by modifying their parameters.

Installing the LunchBox Plugin

Some examples in this chapter use the free GH plugin LunchBox. To install LunchBox, go to `https://www.food4rhino.com/en/app/lunchbox,` locate the latest version for your Rhino version and operating system, click "Install," and follow the prompts. Once installed, you'll see an additional component tab named "LunchBox" appear in the Grasshopper editor. The version of LunchBox used by the author at the time of writing is LunchBox 2025.5.5.0.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_5

Making Loops: Möbius Strips

A *Möbius strip* is a surface that resembles a flat ring with a half-twist in the middle. You can make a Möbius strip with paper as shown in Figure 5-1, using the following steps:

1. Cut out a rectangular strip of paper.
2. Grab one end of the strip and twist it halfway, so the bottom of the strip is showing after the point of the twist.
3. Tape the two ends of the strip together to form a ring.

Möbius strip is what mathematicians call a "non-orientable" surface—for example, if you take a marker, draw a line down the length of a paper Möbius strip, and keep going; eventually, you'll come around to where you started. In this chapter, we'll explore several ways to construct Möbius strip-like surfaces in Grasshopper.

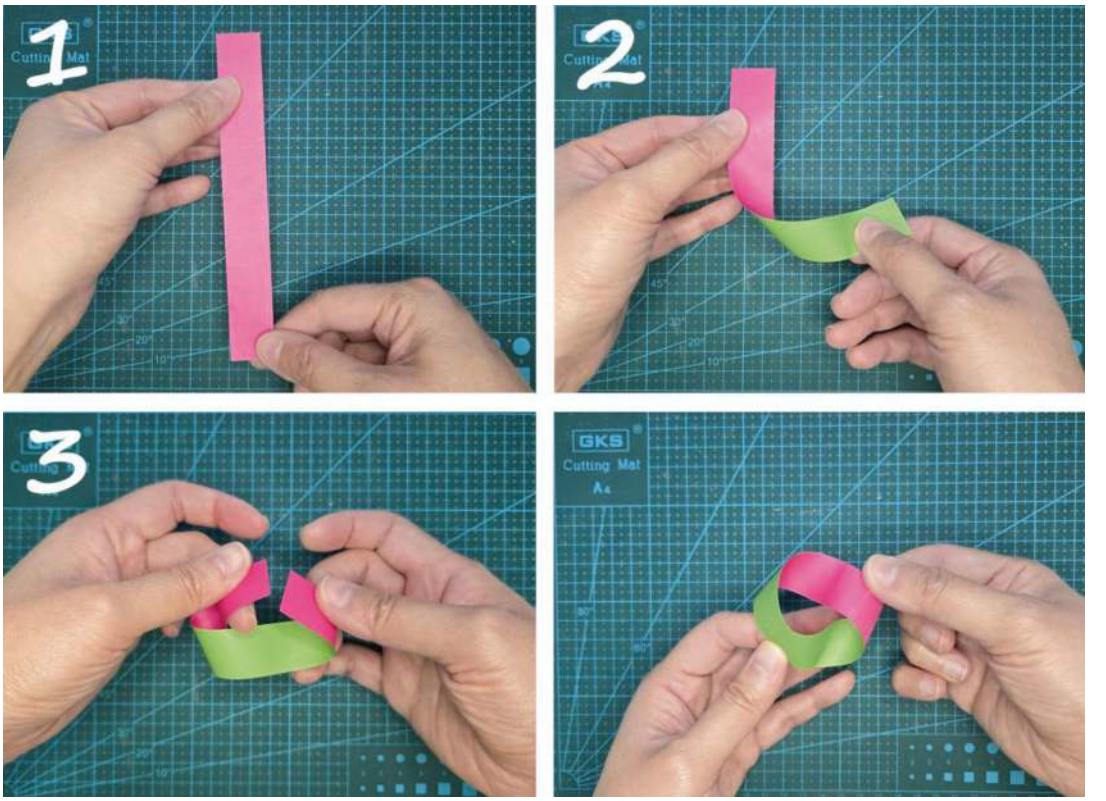


Figure 5-1. *You can create a Möbius strip by (1) cutting a rectangular strip of paper, (2) making a half twist in the middle, and (3) taping the ends together*

Construction Method #1: Using the Mobius Surface Node from LunchBox

Let's make some Möbius strips in Grasshopper. The first method we'll try is the simplest—using the LunchBox ➤ Form ➤ Mobius Surface node (1 in Figure 5-2). This node comes with default parameter values and will produce a Möbius surface as soon as you place it on the canvas.

The Mobius Surface node (1) has various parameters for adjusting the appearance of the strip. ***U Divisions*** and ***V Divisions*** are analogous to the U and V inputs for Divide Surface in Grasshopper—they control the number of evenly spaced "grid lines" dividing the surface, with U specifying the number of horizontal divisions (along X), or "columns," and V the number of vertical divisions (along Y), or rows. You won't see the impact of different *U* and *V Divisions* values by previewing the output of Mobius Surface directly as a brep—instead, link Mobius Surface to a Mesh ➤ Util ➤ Mesh Brep node to convert the brep to a mesh, then toggle on Display ➤ Preview Mesh Edges (Ctrl-M) to preview the *U* and *V Divisions* as mesh edges.

The ***R Parameter*** input for Mobius Surface controls the length of the strip or the radius of the "ring." ***T Parameter*** specifies the number of times to twist the strip (the default is 1). ***Scale*** is a uniform scale factor for the strip.

Domain 1 controls the "length" extent of the strip. Any domain that spans 2pi will produce a complete Möbius strip, for example, -pi to pi (default), 0 to 2pi, and so on. The domain 0 to ½ pi will produce only a quarter of a strip (try this!).

Domain 2 controls the "width" extent of the strip. The wider this domain, the wider the Möbius strip will be.

Note Here is a summary of where you can find the nodes in Figure 5-2: LunchBox ➤Form ➤Mobius Surface (1), Params ➤ Input ➤ Number Slider (2), LunchBox ➤ Panels ➤ Triangular Panels A (3), LunchBox ➤ Panels ➤ Quad Panels (4), LunchBox ➤ Panels ➤ Staggered Quad Panels (5), LunchBox ➤ Panels ➤ Triangle Panels C (6), and Surface ➤ Analysis ➤ Brep Edges (7).

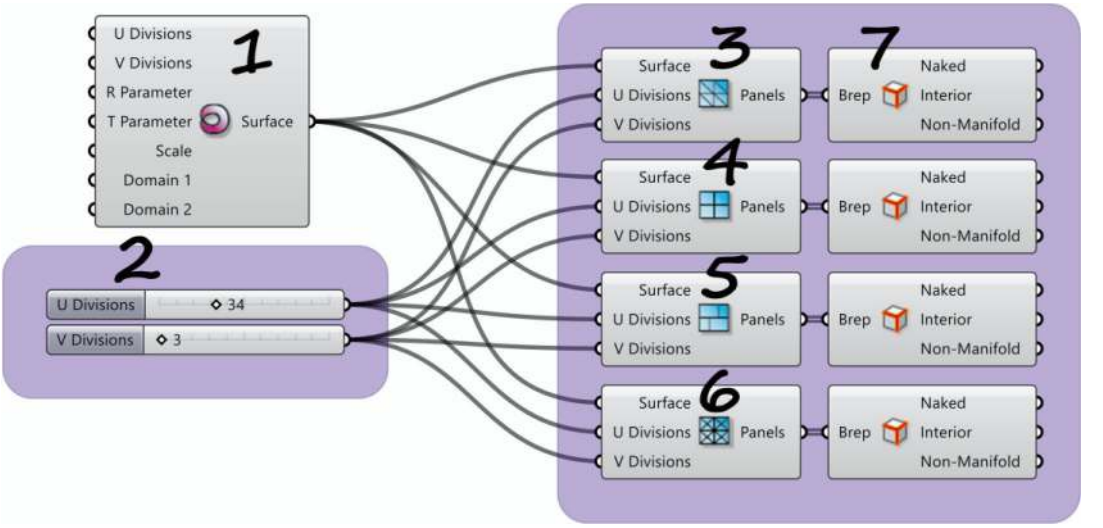


Figure 5-2. *We can quickly create Möbius strips using the Mobius Surface node (1) from the LunchBox plugin. By linking the created strip to nodes under LunchBox's Panels tab (3 to 6) followed by Brep Edges (7), we can create wireframe variations of the strip as shown in Figure 5-3*

Creating Variations Using LunchBox Panels Nodes

We can create some interesting wireframes from a Möbius surface using nodes under the LunchBox ➤ Panels tab. Figure 5-2 shows four examples, which are Triangular Panels A (3), Quad Panels (4), Staggered Quad Panels (5), and Triangle Panels C (6). You link the Surface output of Mobius Surface to the Surface input of each paneling node and use Number Slider to control the U Divisions and V Divisions. Dragging the output of each paneling node to Brep Edges will then turn the surface into a wireframe, as shown in Figure 5-3.

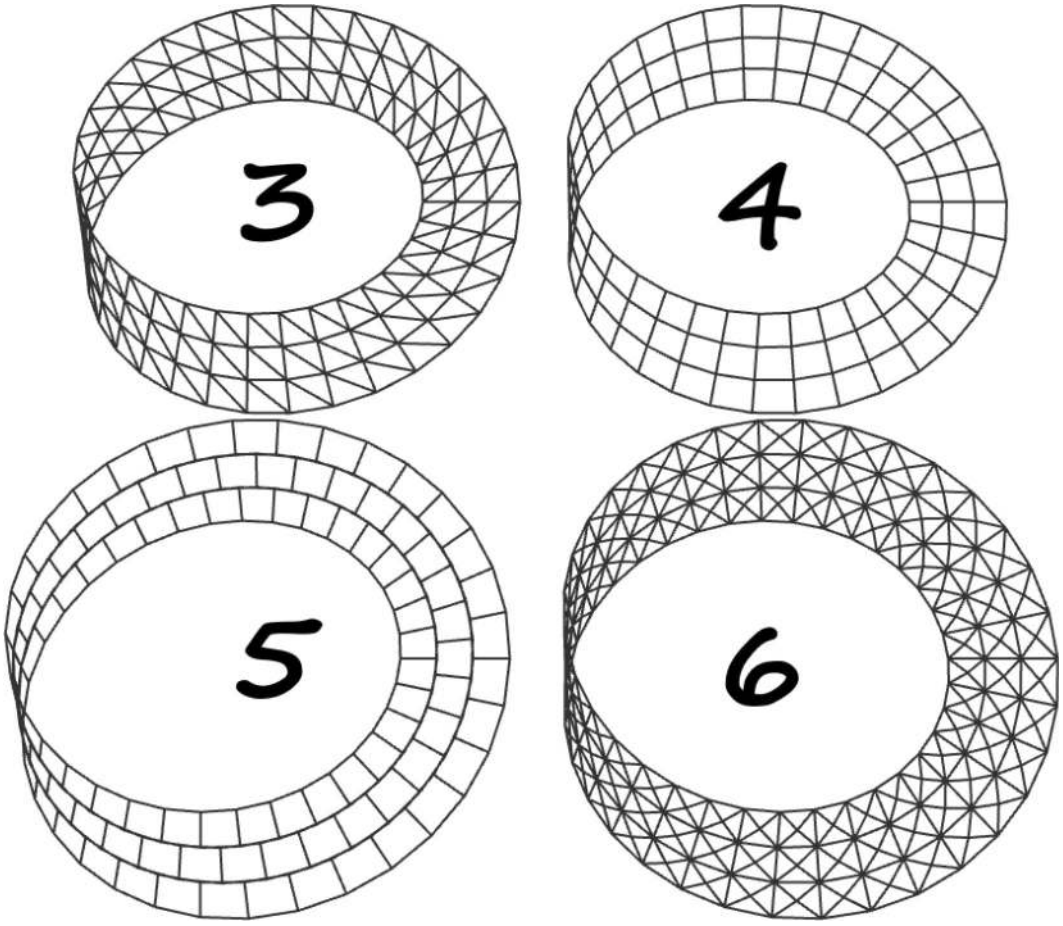


Figure 5-3. *Visual outputs of nodes 3 to 6 from Figure 5-2. (3) Triangular Panels A. (4) Quad Panels. (5) Staggered Quad Panels. (6) Triangle Panels C*

Construction Method #2: Using the Twist Node

Let's create a Möbius surface using only built-in Grasshopper components. It turns out we can mimic the steps in Figure 5-1 for making a paper Möbius strip—create a rectangular surface, twist it halfway lengthwise in the middle, and connect the long ends together. The node graph for this process is shown in Figure 5-4, with previews corresponding to intermediate steps shown in Figure 5-5.

We'll start by creating a Number Slider (1 in Figure 5-4) to control the radius of the Möbius—this radius is to compute the length of the rectangular strip as well as create the circular curve "template" for bending the strip into a Möbius, which we'll see shortly.

We use Rectangle (3) followed by Boundary Surfaces (4) to create a rectangular surface for the strip. Since we are rolling the strip into a circle later, we want the length of the strip to be the same as the circumference of the circle. To do so, we multiply the value of the radius slider (1) with 2*pi (2) and use it to construct a Y domain for Rectangle (3)—as shown, the radius slider is set to 5.0; therefore, the Y domain, or the length of the strip, is 0 to 10.0*pi. We leave the X domain for Rectangle (3) at default, which is –1 to 1, making the strip centering at the Y axis and two units widthwise along X.

Next, we use a Twist node (5) with Axis (X) set to a line segment centering on the strip going its whole length, and Angle (A) set to Pi, to twist the strip (4) halfway once in the middle. To create the line segment, we use Line SDL (6) with Start (S) at the origin, Unit Y as Direction (D), and Length (L) equal to the length of the strip (10.0*pi). You can visualize the result of the twist in Figure 5-5.

Finally, we'll use Flow (8) to bend the strip into a Möbius. Flow (8) derives a transform from a Base (C0) curve to a Target (C1) curve and applies this transform to the given Geometry (G). Therefore, to bend our strip into a Möbius, we want C0 to be the central line running lengthwise through the strip (the Line SDL (6) used for Twist (5)), and C1 to be a Circle (7) with the same radius (1) we used to compute the strip length, so that the strip will fit perfectly around the circle once.

Note Here is a summary of where you can find the nodes in Figure 5-4: Params ➤Input ➤Number Slider (1), Maths ➤ Operators ➤ Multiplication (2), Curve ➤ Primitive ➤ Rectangle (3), Surface ➤ Freeform ➤ Boundary Surfaces (4), Transform ➤ Morph ➤ Twist (5), Curve ➤ Primitive ➤ Line SDL (6), Curve ➤ Primitive ➤ Circle (7), and Transform ➤ Morph ➤ Flow (8).

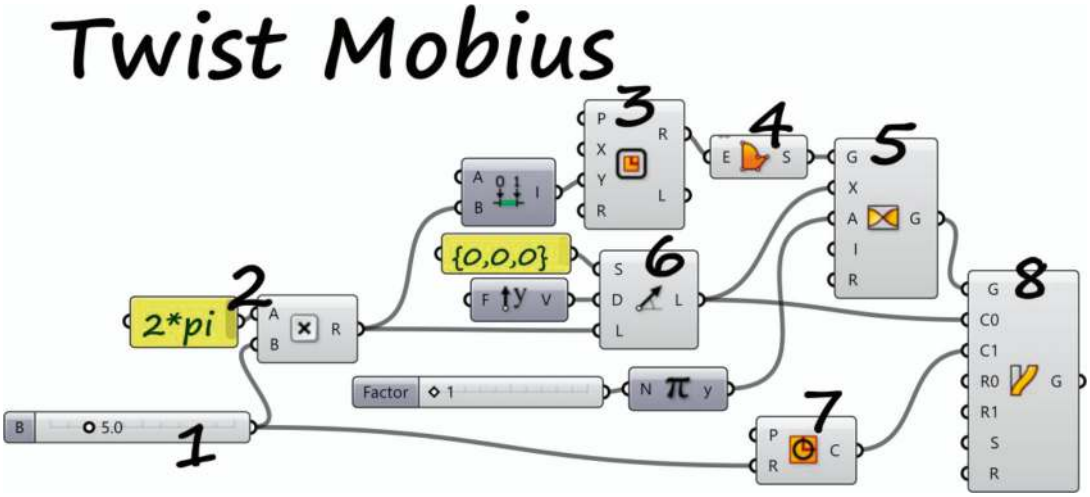


Figure 5-4. *Node graph for generating a Möbius surface using Twist (5) and Flow (8). We create a Boundary Surface (4) from a Rectangle (3), then Twist (5) the surface Pi radians (180 degrees) along a Line SDL (6) going through its central axis. We then use Flow (8) with the central axis as the Base curve (C0) and a Circle (7) as the Target curve (C1) to join the two ends of the surface*

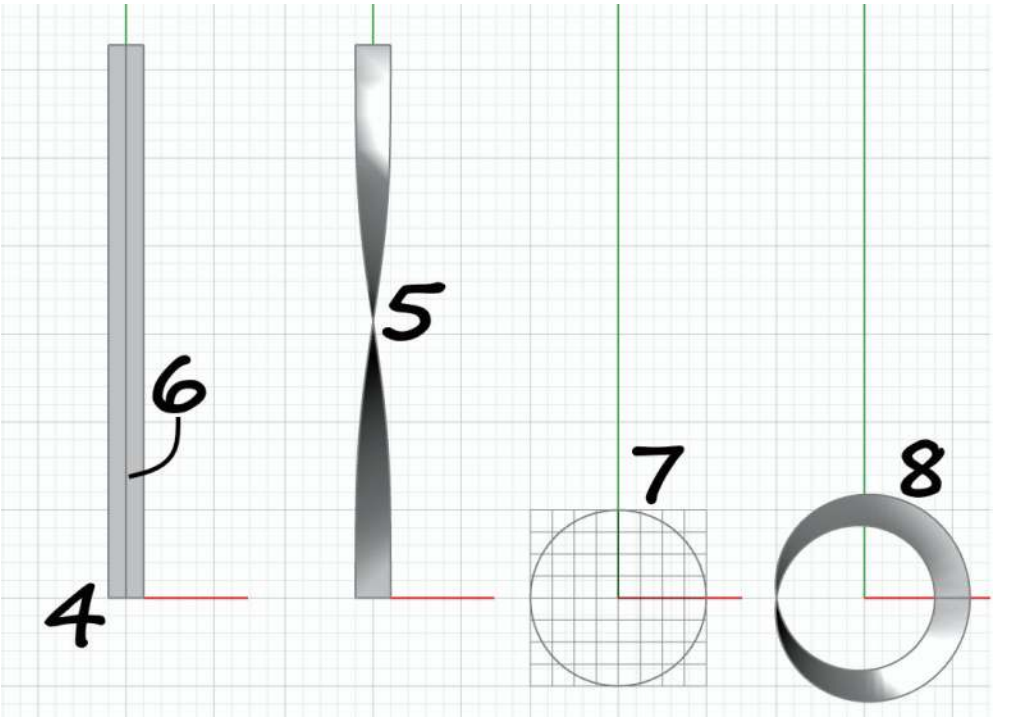


Figure 5-5. *Visual outputs of nodes 4 to 8 from Figure 5-4. We create a Boundary Surfaces (4) from a Rectangle as its border and Twist (5) the surface with a Line SDL (6) along its central axis. We then "join" the ends of the surface using the Flow (8) node, with the center line (6) as the Base (C0) curve and a Circle (7) as the Target (C1) curve*

Construction Method #3: Using Math Formulas

Another way to create a Möbius is to compute the point coordinates constituting a strip using the following formulas (Bourke, 2026). We then interpolate the points into parallel curves and loft the curves into a Möbius surface. This process is implemented in the node graph in Figure 5-6, which we'll look at next.

$$X = \cos(s) + t \times \cos\left(\frac{s}{2}\right) \times \cos(s)$$

$$Y = \sin(s) + t \times \cos\left(\frac{s}{2}\right) \times \sin(s)$$

$$Z = t \times \sin\left(\frac{s}{2}\right)$$

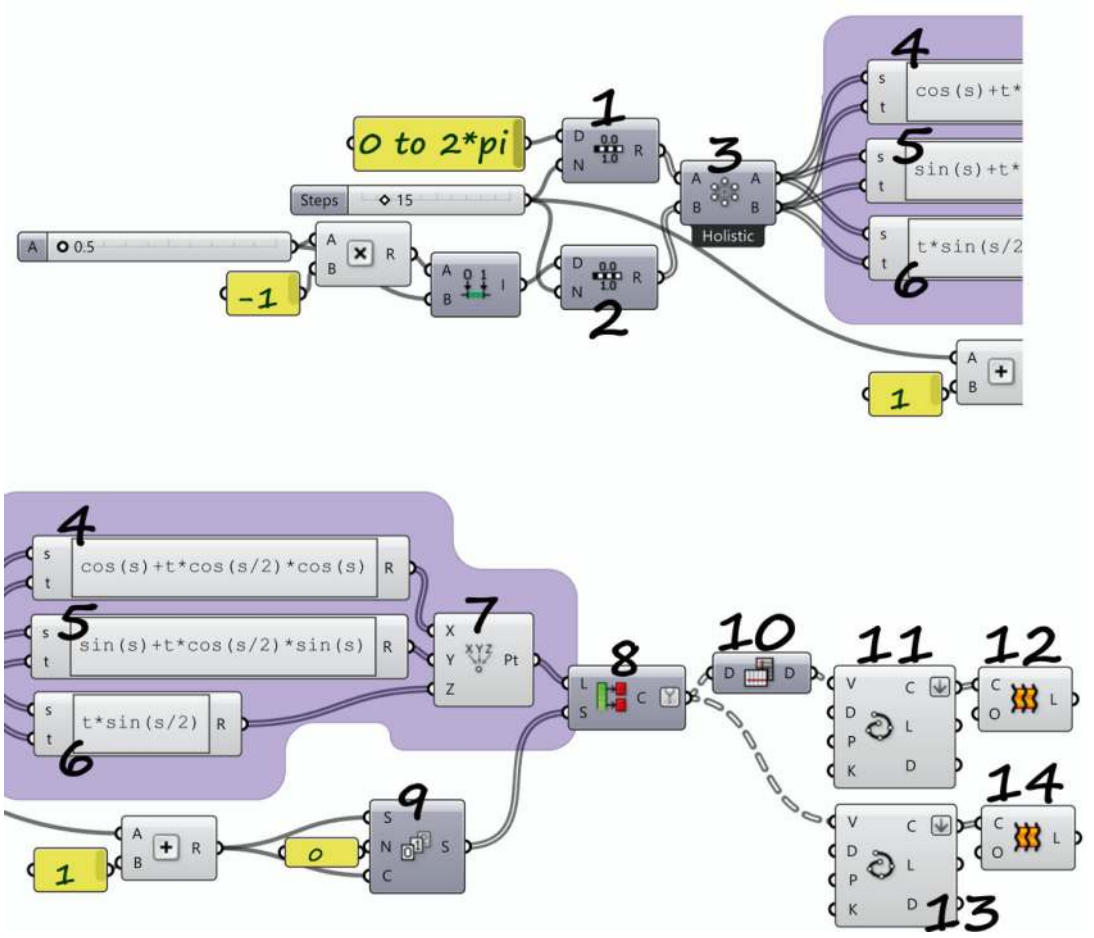


Figure 5-6. *Node graph for generating Möbius strips based on math formulas. We use two Range nodes (1 and 2) to create evenly spaced steps in the domains 0 to 2pi and –0.5 to 0.5, then use the Cross Reference node (Holistic) to pair the step values to create the s and t inputs for the Expression nodes (4 to 6) that compute XYZ coordinates for the Möbius points based on formulas. We then have the choice of interpolating the points width- (11) or lengthwise (13) along the strip, which when Lofted (12 and 14) create the same Möbius surface*

Note Here is a summary of where you can find the nodes in Figure 5-6: Sets ➤Sequence ➤Range (1, 2), Sets ➤ List ➤ Cross Reference (3), Maths ➤ Script ➤ Expression (4, 5, 6), Vector ➤ Point ➤ Construct Point (7), Sets ➤ List ➤ Partition List (8), Sets ➤ Sequence ➤ Series (9), Sets ➤ Tree ➤ Flip Matrix (10), Curve ➤ Spline ➤ Interpolate (11, 13), and Surface ➤ Freeform ➤ Loft (12, 14).

Step 1 of 3: Creating s and t Values for the Formulas

We use two Range nodes (1 and 2 in Figure 5-6) with N = 15 to create 16 stepped values for s and t, with s ranging from 0 to 2*pi and t from –0.5 to 0.5. We then link both Range nodes (1 and 2) to a Cross Reference node (3, right-click ➤ Holistic) to pair up the list of s and list of t values in every possible way, creating 16*16 = 256 pairings.

Note that Cross Reference outputs the pairings in two lists—A and B, with all the first values of the pairs in A and second values in B—in other words, A is the list of s values in the pairings and B the t values.

Step 2 of 3: Computing Point XYZs Based on the Formulas

Next, we create three Expression nodes (4, 5, and 6), each for entering one of the three formulas discussed above for computing the X, Y, and Z values for points on a Möbius surface, respectively.

For each Expression node, right-click the first input and rename it s and the second input and rename it t, and double-click the body of the node to open the "Expression Designer" window, where you'll type in the formula.

You'll then drag the Cross Reference node's A output to the s input of each Expression node and B to the t input. Since we compute the X, Y, and Z coordinates separately, we use a Construct Point node (7) to combine them and create the points.

Step 3 of 3: Interpolating the Points and Lofting

The next step is to Interpolate (11, 13) the points into parallel curves and Loft (12, 14) the curves into a Möbius surface. There are two ways we can interpolate the points—either with the curves running widthwise (11) or lengthwise (13) through the strip, as visualized in Figure 5-7.

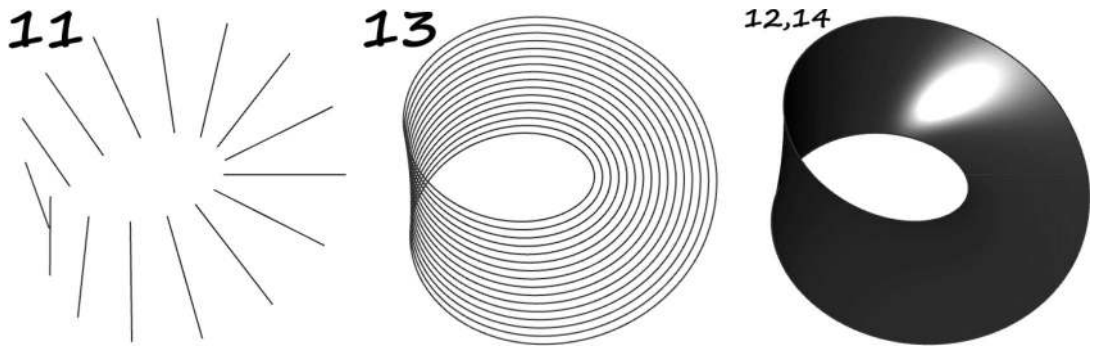


Figure 5-7. *Visual outputs of nodes 11 to 14 from Figure 5-6. (11) shows curves interpolated widthwise along the strip (after Flip Matrix swaps rows and columns of the points data tree). (13) shows curves interpolated lengthwise. A Möbius strip-like surface is produced when either 11 or 13 is passed through a Loft node (12, 14)*

Formatting Input Params into Lists of N+1 Pairs

With either method, we must first format the points into lists of 16 points each before we can interpolate. Recall that in step 1 we used two Range nodes (1 and 2) with N = 15 and a Cross Reference—Holistic (3) node to create a table of

$$(N+1)\times(N+1)=(15+1)\times(15+1)=256\,\textit{pairs of numbers}$$

This table (or grid, or matrix) has 16 columns and 16 rows. Therefore, we can either interpolate each column of points into one curve or each row of points into one curve—either way gives a set of parallel curves we can loft.

We use Partition List (8 in Figure 5-6) to divide the points into lists of 16 each. The only tricky thing is that you must specify the size of *each* sublist, even if all the sublists are the same size. In our case, we want to divvy up a list of 256 points into 16 lists of 16 points each, so we need to create the following list and link it to the Size (S) input of Partition List (8):

$$\left[16,16,16,16,16,16,16,16,16,16,16,16,16,16,16,16\right]$$

which is literally a list of sixteen 16s. To create this list of 16s, we use a Series node (9) with Start (S) = N + 1 = 15 + 1 = 16, where N is the same value for the Range nodes (1 and 2) used to create the s and t values for the math formulas. We make the Count (C) input for Series the same value so it will generate a list of 16 values. Making Step (N) 0 will then repeat the Start (S) value of 16 sixteen times.

Interpolating Points into Curves and Lofting (Per Row or Per Column)

Recall that we can interpolate either per row or per column of points. There is no obvious way to tell whether a row of points goes width- or lengthwise along the Möbius—I found out simply by experimenting. By linking Partition List (8) directly to Interpolate (13) and previewing in the viewport, you get the parallel curves going lengthwise along the Möbius in Figure 5-7. You can then Loft (14) the curves into a Möbius surface.

To interpolate the points widthwise instead, you can link Partition List (8) to Flip Matrix (10), which turns the rows into columns (or columns into rows), followed by Interpolate (11). Using Loft (12) on this set of curves creates the same Möbius surface.

Tip When you partition a list or deal with a data tree in Grasshopper, it is often not obvious what a "row" of data refers to. For example, if you use Divide Surface on a surface, does a branch in the output data tree refer to points going vertically or horizontally? The best way to tell is to just access one branch (or row, or column) and preview it in the viewport.

Replacing Expression Nodes with a Python 3 Script Node

We can combine the three Expression nodes (4, 5, and 6) and Construct Point (7) from Figure 5-6 into a single Python 3 Script node, as shown in Figure 5-8.

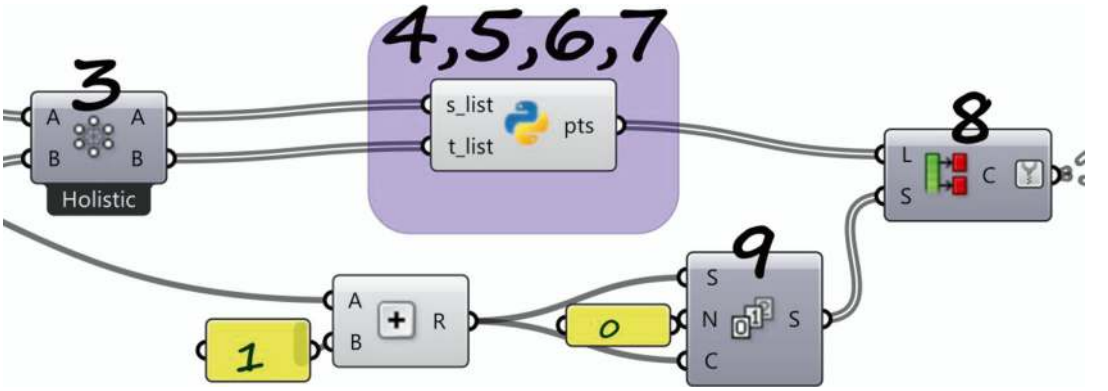


Figure 5-8. *The Expression nodes (4,5,6) and Construct Point (7) in Figure 5-6 are replaced with a Python 3 Script node, whose code is shown in Listing 5-1*

The code for the Python 3 Script node is shown in Listing 5-1. The Access and Type Hints for the node's inputs and output are summarized in Table 5-1.

Listing 5-1. Code inside the Python 3 Script node in Figure 5-8 (see `/Ch5/Figure5-8_math_mobius_python3.gh`).

```
import ghpythonlib.components as ghcps
from math import sin, cos

pts = []
for s, t in zip(s_list, t_list):
    x = cos(s) + t*cos(s/2)*cos(s)
    y = sin(s) + t*cos(s/2)*sin(s)
    z = t*sin(s/2)
    pts.append(ghcps.ConstructPoint(x, y, z))
```

In Listing 5-1, we start by importing `ghpythonlib.components` as the shorthand `ghcps` so we can call Grasshopper components as functions (we'll need it to access Construct Point). We also import the `sin` and `cos` functions from Python's built-in `math` module so we can use those for the math formulas.

Next, we initialize the `pts` output as an empty list, then create a point for each input pairing of s and t values in the `for` loop based on the math formulas. Since the s and t values are given as two separate lists, `s_list` and `t_list`, we pair up the values item-wise by calling Python's built-in `zip` function. For each index, `zip` takes one item from each list and forms a pair. For example, if `s_list = [1,2,3,4,5]` and `t_list = [6,7,8,9,10]`, then `zip(s_list, t_list)` will give you the following:

```
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

For each iteration of the `for` loop, we use one pair of s and t values to compute a set of x, y, and z values based on the math formulas used for the Expression nodes in Figure 5-6. We then call `ghcps.ConstructPoint()` with the x, y, and z values to create a point and add it to the end of the `pts` list. Calling `ghcps.ConstructPoint()` in the Python code is equivalent to using a Construct Point node on a GH canvas.

Table 5-1. *Access and Type Hints for the Python 3 Script node in Figure 5-8*

Name	Input or Output	Access	Type Hints
s_list	input	List Access	float
t_list	input	List Access	float
pts	output	N/A	Point3d

Note Recall that for a Python 3 Script node, you right-click an input to change its name and set its Access (Item, List, or Tree Access) and Type Hints.

You also right-click an output to set its name and Type Hints. There is no Access Hint for an output.

Tip If you use two lists of different lengths with the `zip` function, it will produce pairings up to the end of the shorter list. For example, if l1=[1,2,3], l2=[4,5,6,7,8], zip(l1, l2) will give you the pairings [(1, 4), (2, 5), (3, 6)].

Making Spirals: Logarithmic Shells

In this section, we'll create a GH definition (Figure 5-9) for generating a shell-like surface based on a logarithmic spiral curve. We use a Python 3 Script node (4) to derive point coordinates based on the spiral curve's formula plus corresponding radii at those points. We then connect the points with PolyLine (5), spawn evenly spaced circles along the polyline, and Loft (8) the circles into a shell-like surface.

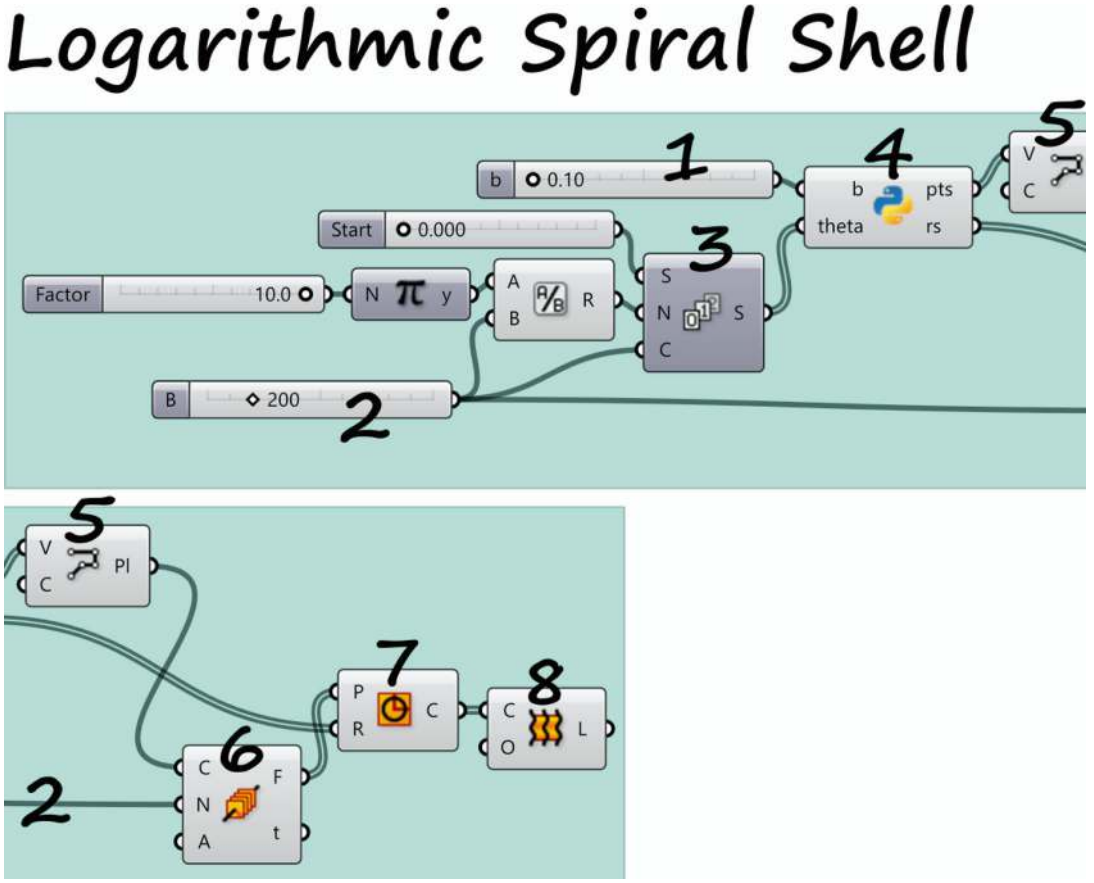


Figure 5-9. *Node graph for generating a shell-like surface based on a logarithmic curve. We use a Python 3 Script node (4) with the code in Listing 5-2 to create points for the curve, link the points with PolyLine (5), and create evenly spaced planes perpendicular to the curve with Perp Frames (6). We then make Circle's (7) centered on these planes and Loft (8) them into a shell. Visual outputs of nodes 4, 5, 6, 7, and 8 are shown in Figure 5-10*

Step 1 of 3: Creating Input Values for the Spiral Curve

We'll be using the Logarithmic Spiral formula from Wolfram MathWorld (Weisstein, 2026), which is as follows:

$$r = a \times e^{b\theta}$$

$$x = r \times \cos(\theta)$$

$$y = r \times \sin(\theta)$$

where θ (theta) is the angle from the X axis. Each multiple of 2Pi radians makes the spiral go around once, so theta = 10Pi makes the spiral go around five times. r is the distance between a point (x, y) on the spiral and the origin. We'll use r as the shell radius at (x, y), which gives a good approximation, although *not* perfect—since r is the distance to the origin, using it as shell radius will make that part of the shell overlap portions of the shell corresponding to smaller x and y's. "a" and "b" are arbitrary numbers, and e is Euler's number, a math constant equal to roughly 2.718.

We'll simplify the formula by making a = 1. A Number Slider (1) is used to control b, which is set to 0.1 in Figure 5-9. A second Number Slider (2) controls the total number of points we'll derive for the curve (200 as shown in Figure 5-9). We'll use a total of theta = 10Pi to make the spiral go around five times. Therefore, a Series node (3) with Count (C) = 200 and Step (N) = 10Pi / 200 is used to produce the 200 incremental theta values.

Step 2 of 3: Using Python to Derive Points for the Spiral Curve and Radii for the Shell

With the values for a, b, and θ (theta) established, we're ready to create the Python 3 Script node (4 in Figure 5-9) with the code shown in Listing 5-2 and the Access and Type Hints summarized in Table 5-2. The node has two inputs, a `b` value and a list of `theta` values required by the formula, along with two outputs: `pts`, the list of point coordinates calculated based on the inputs, and `rs`, the list of r values at those points (shell radii).

In Listing 5-2, we start by importing `ghpythonlib.components` as the shorthand `ghcps` so we can call GH components as Python functions. We also import `e` (Euler's number) and the functions `cos`, `sin`, and `pow` from Python's built-in `math` module as they are required by the curve formula.

Next, we initialize the two outputs `pts` and `rs` to empty lists. Then, in each iteration of a `for` loop, we calculate r, x, and y for one theta value based on the curve formula and call `ghcps.ConstructPoint()` to create a point at (x, y, 0) and append it to the output list `pts`. We also append r to the output list `rs`.

Listing 5-2. Code inside the Python 3 Script node from Figure 5-9, which produces the points and radii for a logarithmic spiral curve (see `/Ch5/Figure5-9_5-10_log_shell.gh`)

```
import ghpythonlib.components as ghcps
from math import e, cos, sin, pow

pts = []
rs = []
for i in range(len(theta)):
    ti = theta[i]
    r = pow(e, b*ti)
    x = r*cos(ti)
    y = r*sin(ti)

    p = ghcps.ConstructPoint(x, y, 0)
    pts.append(p)
    rs.append(r)
```

Table 5-2. *Access and Type Hints for the Python 3 Script node in Figure 5-9*

Name	Input or Output	Access	Type Hints
b	input	Item Access	float
theta	input	List Access	float
pts	output	N/A	Point3d
rs	output	N/A	float

Step 3 of 3: Generating the Shell Surface

In the final part of the node graph (Figure 5-9), we create the spiral curve and loft it into a shell surface.

Since the Python 3 Script (4) node in step 2 derives evenly spaced points based on the formula, we simply use PolyLine (5) to link the points into a polyline to create the spiral.

Next, we use Perp Frames (6) to spawn evenly spaced planes with the polyline going through their centers. If we ensure the number of planes matches the number of points output by the Python 3 Script node (4), the planes' origins will coincide with those points—this is required since the Python 3 Script node (4) also calculates the shell radii at those points.

Next, we use the planes as the P input for Circle (7) and the shell radii output by the Python 3 Script node as the R input, to create circles centered at the origins of those planes as the shell's cross-sectional curves.

Lastly, we use Loft (8) to create the shell surface through the circles. The outputs of nodes 4, 5, 6, 7, and 8 from Figure 5-9 are visualized in Figure 5-10.

Note Here is a summary of where you can find the nodes in Figure 5-9: Params ➤Input ➤Number Slider (1, 2), Sets ➤ Sequence ➤ Series (3), Maths ➤ Script ➤ Python 3 Script (4), Curve ➤ Spline ➤ PolyLine (5), Curve ➤ Division ➤ Perp Frames (6), Curve ➤ Primitive ➤ Circle (7), and Surface ➤ Freeform ➤ Loft (8).

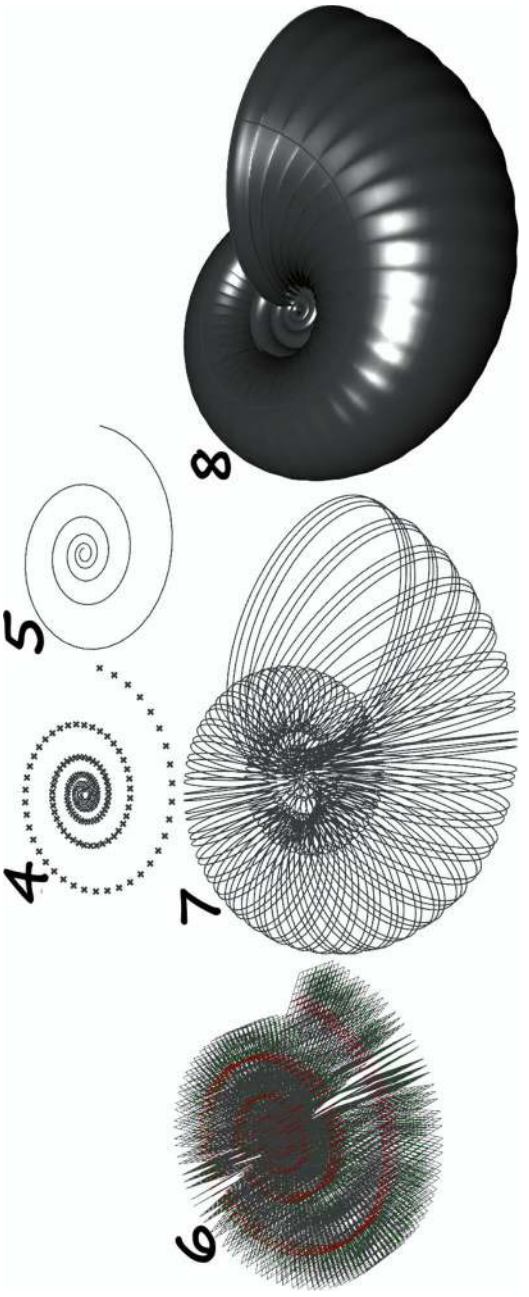


Figure 5-10. *Visual outputs of the nodes in Figure 5-9. (4) are the points output by the Python 3 Script node based on the log curve formula. (5) is the curve created by linking the points with PolyLine. (6) are the planes perpendicular to the curve created using Perp Frames. (7) are the Circle's centered on these planes. (8) is the surface Lofted from the circles*

Making Spirals: Staircases

So far, we've used loops and spirals to create ornaments and abstract structures. In this section, we'll use spirals to generate something practical—a parametric staircase.

Figure 5-11 shows a bird's-eye view of the node graph, which is broken into six stages. In stage 1, we form the center pole and a spiral curve around the pole with evenly spaced planes to be used in stage 2 to position the tread rectangles. In stage 3, we taper the treads toward the pole and extrude them in stage 4 to add thickness. In stage 5, we find positions inset from the outer edges of the treads to extrude balusters from. Finally, in stage 6, we add the handrail by interpolating and piping the top center points of the balusters.

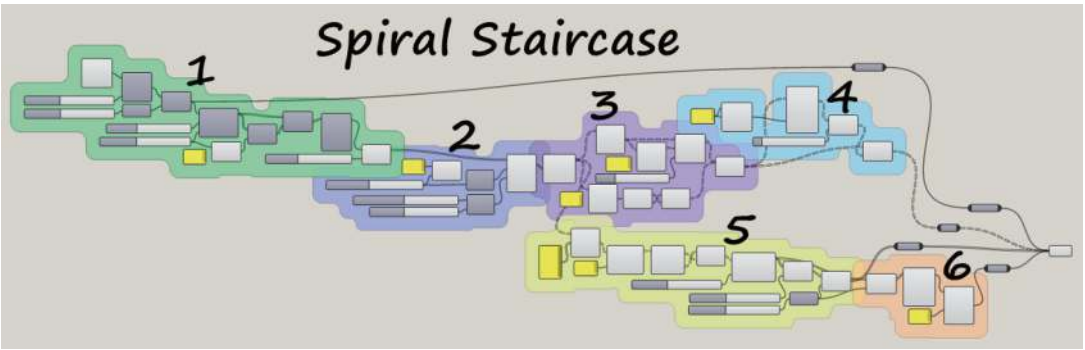


Figure 5-11. *Bird's-eye view of the spiral staircase node graph, which consists of six stages*

Stage 1 of 6: Creating the Center Pole and a Spiral Curve Around the Pole

In Figure 5-12, we zoom in on stage 1, where we form the center pole and a spiral curve around it with evenly spaced planes for spawning the tread Rectangle's later.

We use Circle CNR (1a) to create a base circular curve and Extrude (1b) along +Z to create a cylindrical surface as the center pole. We then use Divide Surface (1c) to create an evenly spaced grid of points on the pole, by specifying the number of row and column lines through the U and V inputs, respectively. Note that U and V are the number of "division lines" for the grid; therefore, the number of points created is (U+1)*(V+1).

To create a spiral curve around the pole, we want to extract every (V-1)th point and Interpolate (1f). Let's take this step-by-step—First, we use Cull Nth (1d) to remove every (V-1)th point starting at the bottom of the pole. Next, the original grid of points (1c) and

the grid missing every (V-1)th point are linked to a Set Difference node (1e), which gives you the difference between the two, hence every (V-1)th point. We then Interpolate (1f) these points into a curve and create evenly spaced planes along this curve with Horizontal Frames (1g) that we'll use in stage 2 to spawn the tread Rectangle's.

Note Here is a summary of where you can find the nodes in stage 1 of Figure 5-12: Curve ➤Primitive ➤Circle CNR (1a), Surface ➤ Freeform ➤ Extrude (1b), Surface ➤ Util ➤ Divide Surface (1c), Sets ➤ Sequence ➤ Cull Nth (1d), Sets ➤ Sets ➤ Set Difference (1e), Curve ➤ Spline ➤ Interpolate (1f), and Curve ➤ Division ➤ Horizontal Frames (1g).

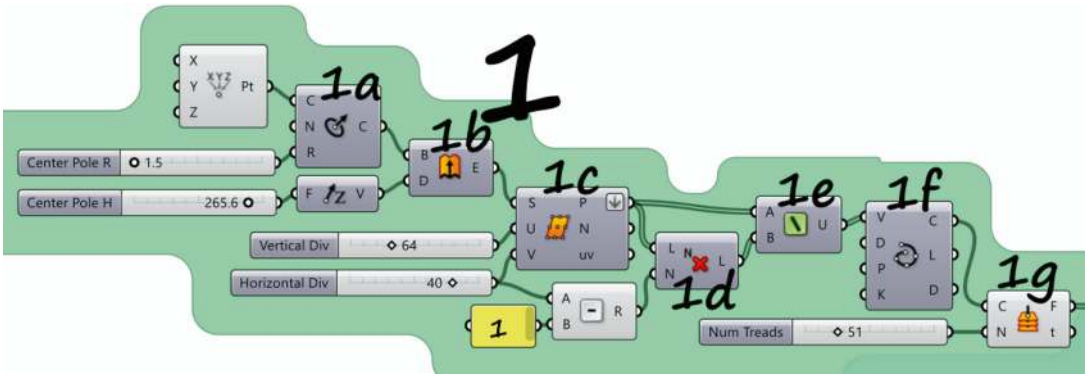


Figure 5-12. *Stage 1 of the spiral staircase node graph (Figure 5-11). We Extrude (1b) a Circle CNR (1a) along +Z to form the center pole and use Divide Surface (1c) to create a grid of points on the pole. We then extract every (V-1)th point in a two-step process—use Cull Nth (1d) to prune every (V-1)th point, then Set Difference (1e) between the original set of points (1c) and 1d. After that, we Interpolate (1f) every (V-1)th point to create a spiral curve around the pole. Visual outputs of nodes 1b to 1f are shown in Figure 5-13*

Visualizing the Steps for Creating a Spiral Around a Pole

Figure 5-13 shows the steps for creating a spiral curve around the pole with nodes 1b to 1f from Figure 5-12. The example is simplified with V reduced from 40 to 10; hence, every (V-1)th = (10-1)th = 9th point is isolated. Grasshopper uses 0-based indexing, for example; a list of three items has indices 0, 1, and 2. Therefore, removing every 9th point means removing the points with the indices:

$$9\times1-1=8$$

$$9\times2-1=17$$

$$9\times3-1=26$$

...and so on. Because our surface is a cylinder, Divide Surface (1c) creates rings of points with indices counting up from the bottom as shown in Figure 5-13.

Imagine drawing a grid of evenly spaced points on a piece of printer paper and taping up its ends to form a cylinder—the first and last columns of points coincide after the join, which is why the first and last point of each "ring" along the cylinder overlap. Because of this overlap, taking every (V-1)th point as we ascend the rings will land us at a position offset by 2 points each time, therefore creating a spiral.

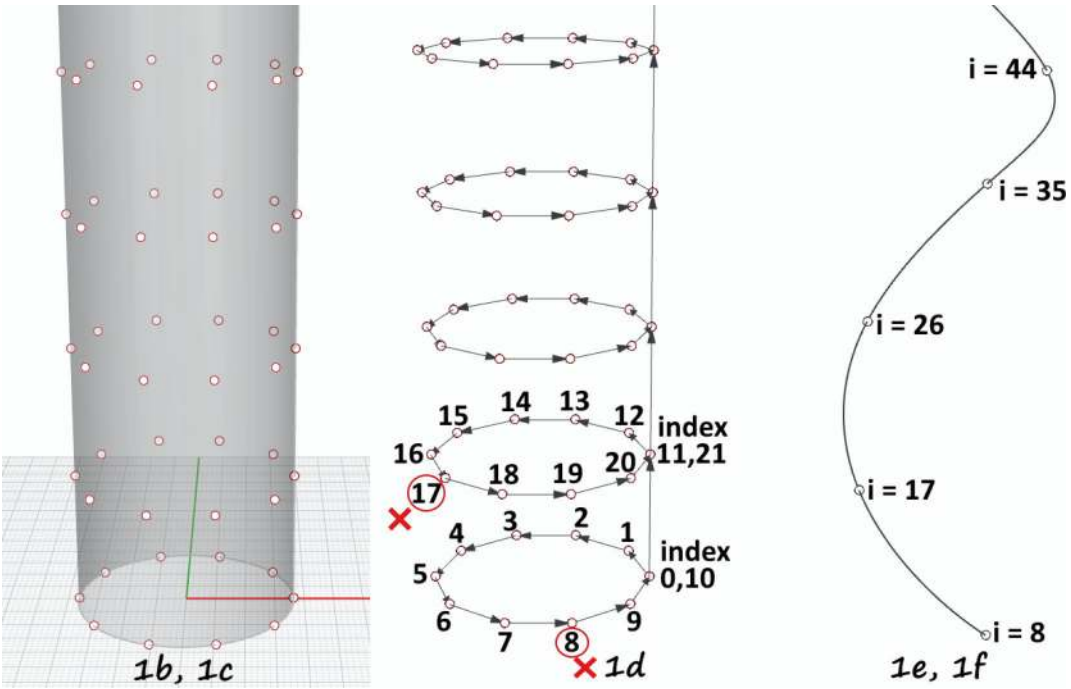


Figure 5-13. *Visual outputs of nodes 1b to 1f from Figure 5-12. We Extrude (1b) the Circle CNR (1a) to create the center pole and use Divide Surface (1c) to create a grid of points on the pole surface. We then Interpolate (1f) every V^{th} point (which has indices V-1, 2V-1, ...etc.) to create a spiral curve around the pole. To isolate the desired points, we use Cull Vth (1d) to prune every Vth point, then Set Difference (1e) between 1c and 1d. The figure shows a simple example where V is 9*

Stage 2 of 6: Forming Tread Rectangles Along the Spiral Curve

In Figure 5-14, we zoom in on stage 2 of the node graph from Figure 5-11, where we create the curve outlines for the treads with Rectangle (2c). Recall that at the end of stage 1 (Figure 5-12), we use Horizontal Frames (1g) to create planes evenly spaced along the spiral curve wrapped around the center pole. We use those planes there as the Plane (P) input for Rectangle (2c).

The tread depth and width are given via the X and Y inputs (the shorter and longer sides of the treads). As shown, we use Construct Domain (2a) with A = 0 - 5.4 = -5.4 and B = 5.4 for X and a second Construct Domain (2b) with A = -5.4 and B = 31.2 for Y.

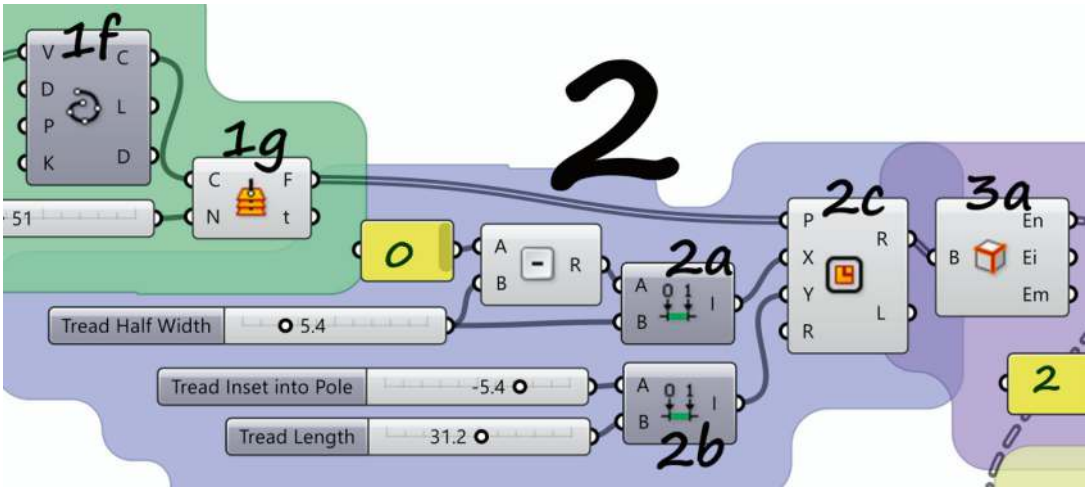


Figure 5-14. *Stage 2 of the spiral staircase node graph (Figure 5-11). We use Rectangle (2c) to create tread curves based on frames (1g) along the spiral curve (1f). Visual outputs of 1f, 1g, and 2c are shown in Figure 5-15*

You might have noticed that the X domain is symmetrical about 0, but Y is not—this has to do with how we want the tread to "mount" on to the pole. We want the pole to center on the tread depth-wise, but anchors on the inner end widthwise.

By making the X domain symmetrical about 0, we position the input plane for Rectangle (2c) so that the sides of the rectangle come out equal amounts from the plane's center along X. The negative part of the Y domain controls how much a tread sticks out past the pole, toward the center of the staircase, while the positive part adjusts how wide the tread is.

Note Here is a summary of where you can find the nodes in stage 2 of Figure 5-14: Maths ➤Domain ➤Construct Domain (2a, 2b) and Curve ➤ Primitive ➤ Rectangle (2c).

Figure 5-15 shows the outputs of nodes 1f, 1g, and 2c from stages 1 and 2 of the staircase node graphs (Figures 5-12 and 5-14). 1f is the Interpolate node that links every (V-1)th point of the UV grid on the cylindrical surface of the center pole to create the spiral curve. 1g is the Horizontal Frames node that creates evenly spaced planes along the spiral. 2c is the list of rectangular curves defining the outlines of the treads, with positions based on the planes defined by 1g.

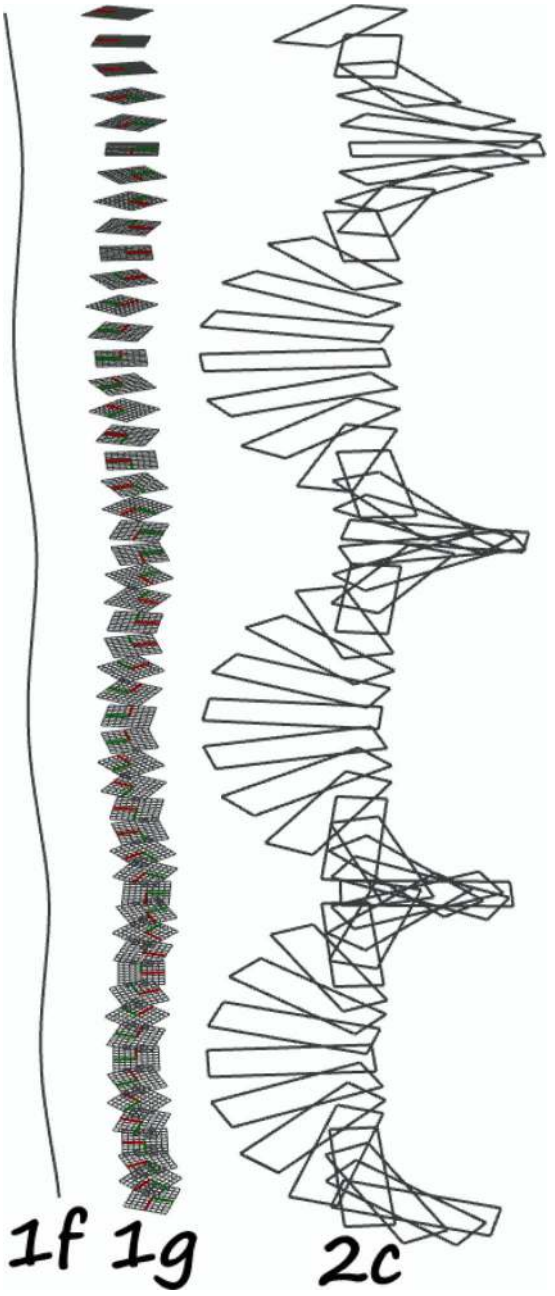


Figure 5-15. *Visual outputs of nodes from stages 1 and 2 (Figures 5-12 and 5-14). After creating a spiral curve (1f) around the center pole, Horizontal Frames (1g) is used to create evenly spaced planes along the curve which are in turn used to position the tread Rectangle's (2c)*

Stage 3 of 6: Tapering Treads Toward the Center Pole

Figure 5-16 shows stage 3 of the staircase node graph from Figure 5-11, where we taper the treads, so the end closer to the center pole is narrower—a common feature of spiral staircases. We'll structure the graph so that the amount of tapering is adjustable through a Number Slider.

We start by linking Rectangle (2c) to Brep Edges (3a) to extract the edges of the tread outlines through the En output. We then use List Item (3b) to access the inner edges at i = 0 (closer to the center pole) and another List Item (3e) to access the outer edges at i = 2. Our goal is to shrink the inner edges (3b), then Loft (3h) them with the outer edges to create tapered tread surfaces.

The first step is to find the inner edges' midpoints via Evaluate Curve (3c) with t = 0.5, which we then use as the Center (C) input for Scale (3d) to shrink the inner edges. We use a Number Slider to control the Factor (F) for Scale (3d) to make it parametric.

The next step is to reverse the direction of the outer edges so Loft will behave correctly. To do so, we link the outer edges (3e) to End Points (3f) then use its two outputs in the wrong order to create line segments with the Line node (3g).

The last step is to Shift-drag the shrunken inner edges (3d) and the reversed outer edges (3g) to a Loft node (3h) to create the tapered tread surfaces. Figure 5-17 visualizes the outputs of nodes 3b to 3h as described in this section.

Note Here is a summary of where you can find the nodes in stage 3 of Figure 5-16: Surface ➤Analysis ➤Brep Edges (3a), Sets ➤ List ➤ List Item (3b, 3e), Curve ➤ Analysis ➤ Evaluate Curve (3c), Transform ➤ Affine ➤ Scale (3d), Curve ➤ Analysis ➤ End Points (3f), Curve ➤ Primitive ➤ Line (3g), and Surface ➤ Freeform ➤ Loft (3h).

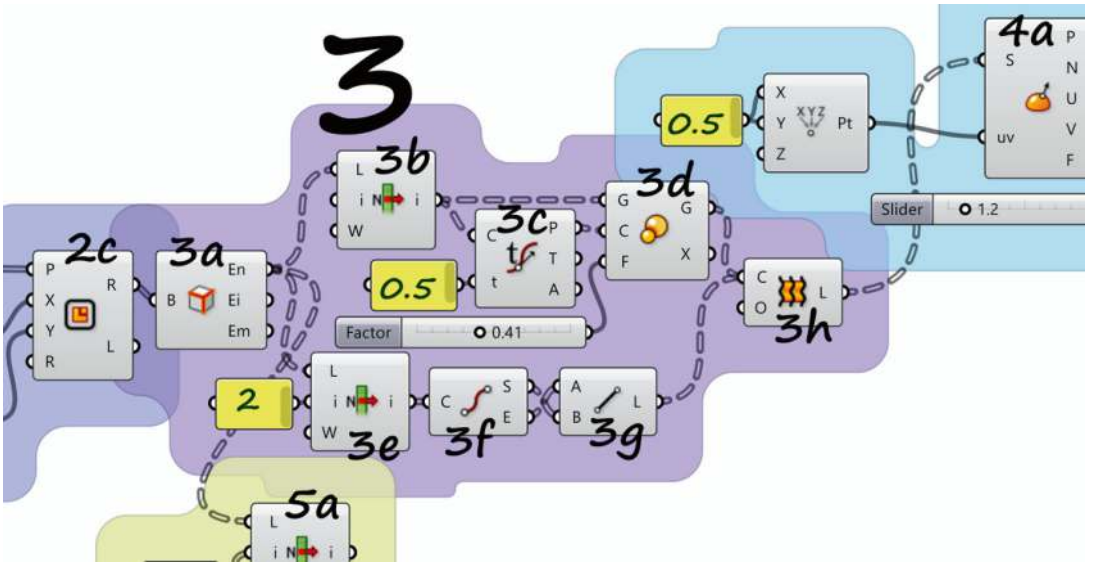


Figure 5-16. *Stage 3 of the spiral staircase node graph (Figure 5-11). We Scale (3d) the inner tread Rectangle edges (3b, at i = 0) about their centers (3c, t = 0.5), then Loft (3h) the inner and outer edges (3g) to form the new tread surfaces that taper toward the center pole. Note that we must flip the direction of the outer edges so Loft (3h) will work correctly. We flip the edges by extracting their End Points (3f) then using End (E) and Start (S) as the A and B inputs for Line (3g). The visual outputs of 3b to 3h are shown in Figure 5-17*

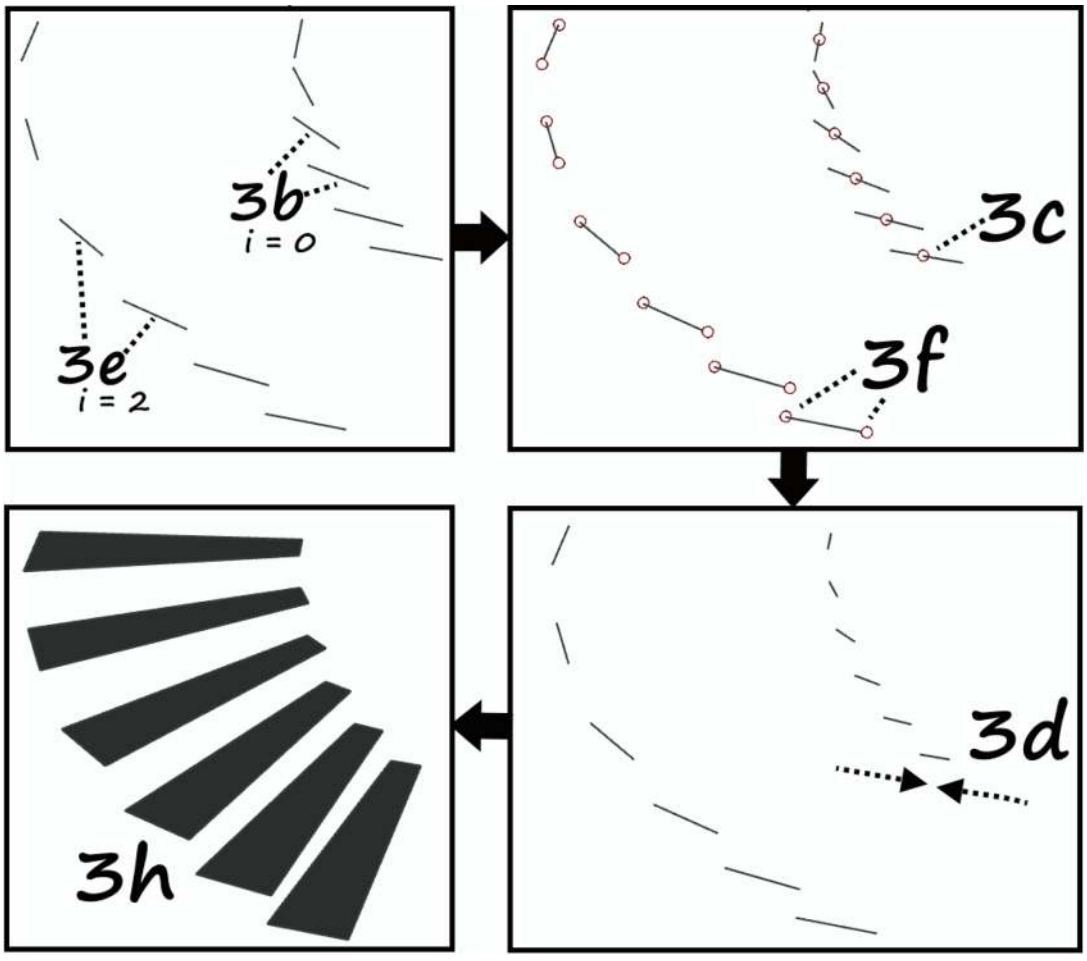


Figure 5-17. *Visual outputs of the nodes from stage 3 (Figure 5-16). We use two List Item nodes (3b and 3e) to access the tread edges at indices 0 and 2, respectively. We then Scale (3d) the edges at i = 0 about their centers (found via Evaluate Curve (3c) with t = 0.5). Next, we extract the End Points (3f) of the edges at i = 2 and use them to reverse the edges' directions, so Loft (3h) will form the tapered surfaces correctly*

Stage 4 of 6: Extruding Tread Surfaces to Create Thickness

Figure 5-18 shows stage 4 of the staircase node graph from Figure 5-11, where we extrude the tread surfaces to give them thickness.

We want to extrude each tread height-wise no matter how they are oriented; therefore, we derive the normal vector of the same UV coordinates per tread and use that as the extrude direction.

Recall that we can divide a surface into a grid of evenly spaced points, for example, using the Divide Surface node (1c) as we did in Figure 5-12—we can then reference each divided point using its UV coordinates, with U specifying the horizontal and V the vertical coordinate, just like X and Y on the Cartesian plane.

We can use Evaluate Surface (4a) to refer to a point on a surface using UV coordinates without having physically divided that surface. Evaluate Surface (4a) works in a similar way as Evaluate Curve, except we give Evaluate Surface UV coordinates (as a point) instead of a single t value to specify where we are referring to on the surface. Here, we use Construct Point to create a point with coordinates (0.5, 0.5, 0) to use as the uv input for Evaluate Surface (4a), to get the normal vectors (N) of length 1 at that point for every tread surface. We then scale the normal vectors with a Number Slider before using it as the Direction (D) input for Extrude (4b) so we can control the thickness of the treads.

The outputs of nodes 4a (Evaluate Surface) and 4b (Extrude) are visualized in Figure 5-19.

Note Here is a summary of where you can find the nodes in stage 4 of Figure 5-18: Surface ➤Analysis ➤Evaluate Surface (4a) and Surface ➤ Freeform ➤ Extrude (4b).

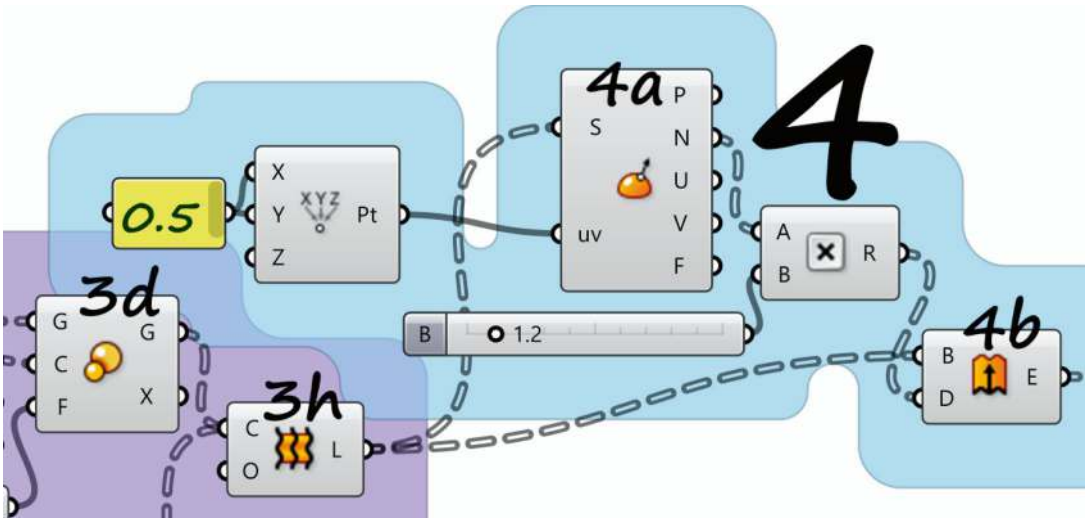


Figure 5-18. *Stage 4 of the spiral staircase node graph (Figure 5-11). We Extrude (4b) the tread surfaces (3h) along their normal vectors, found via the Evaluate Surface node (4a). The visual outputs of these nodes are shown in Figure 5-19*

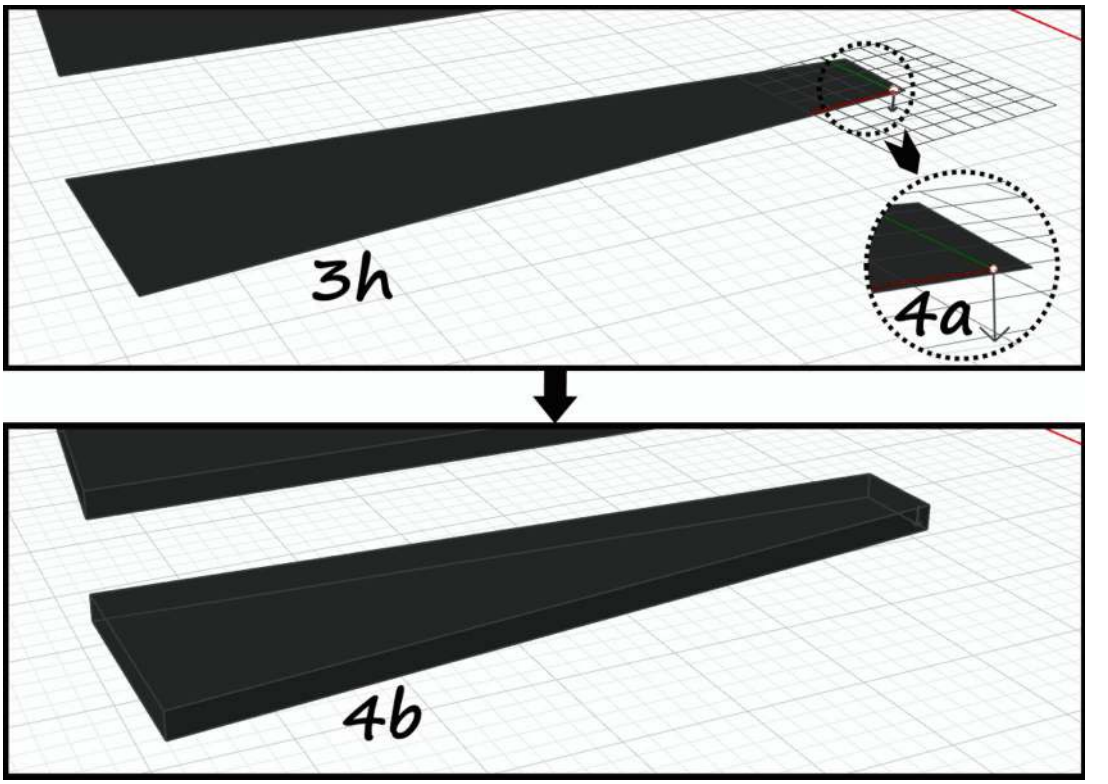


Figure 5-19. *We use Evaluate Surface (4a in Figure 5-18) to find the normal vector at the point with UV coordinates (0.5, 0.5, 0) on each tread surface (3h), then Extrude (4b) along these normals to give the treads thickness*

Stage 5 of 6: Creating Balusters

Figure 5-20 shows stage 5 of the staircase node graph from Figure 5-11, where we extrude balusters from points found parametrically along the central axes of the treads.

Figure 5-21 shows the visualization of nodes 5a to 5g. Recall that in stage 3 (Figure 5-16), we use Brep Edges (3a) to extract the tread outlines' edges. Here we link 3a to a List Item node (5a) with i = 0 and 2 to retrieve the inner and outer edges of the treads, which are the closest and farthest edges from the center pole. We then use Evaluate Curve (with right-clicking C ➤ Reparameterize and t = 0.5) to find the midpoints of the inner and outer edges.

We want the baluster to sit somewhere between the midpoints linking the inner and outer edges of the tread, but closer to the outer edge. Put another way, we want to decide how far the baluster should inset from the outer edge.

We can do this with a List Item (5c) node with an extra output and use the +1 output as the start (A) and the i output as the end (B) for Line (5d) to create a line that goes from the outer edge toward the center pole. We can then use Evaluate Curve (5e, with the C input set to Reparameterize) with t controlled by a Number Slider, to dial the amount inward from the outer edge the balusters should sit. All that's left to do is create Circle's (5f) centered at these inset points and Extrude (5g) toward +Z to create the baluster surfaces.

Note Here is a summary of where you can find the nodes in stage 5 of Figure 5-20: Sets ➤ List ➤ List Item (5a, 5c), Curve ➤ Analysis ➤ Evaluate Curve (5b & 5e, right-click the C input ➤Reparameterize), Curve ➤ Primitive ➤ Line (5d), Curve ➤ Primitive ➤ Circle (5f), and Surface ➤ Freeform ➤ Extrude (5g).

When we right-click the C input of Evaluate Curve and select Reparameterize, we're telling Grasshopper to map the length of the curve to a domain from 0 to 1, so 0.5 means 50% along the curve from the start.

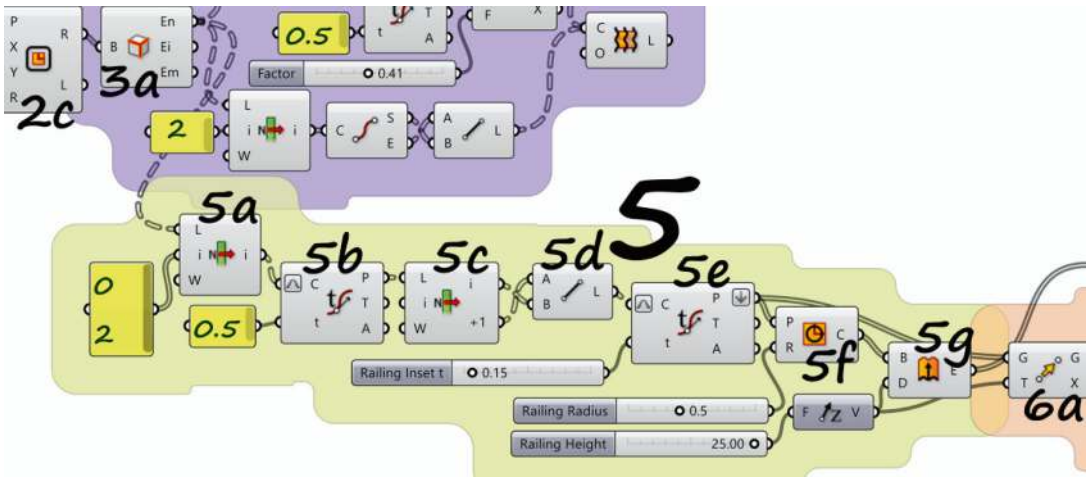


Figure 5-20. *Stage 5 of the spiral staircase node graph (Figure 5-11). We generate balusters extruded from circles (5f) positioned slightly inset (5e) from the outer edges of the treads relative to the center pole. The outputs of the intermediate nodes (5a to 5f) are visualized in Figure 5-21*

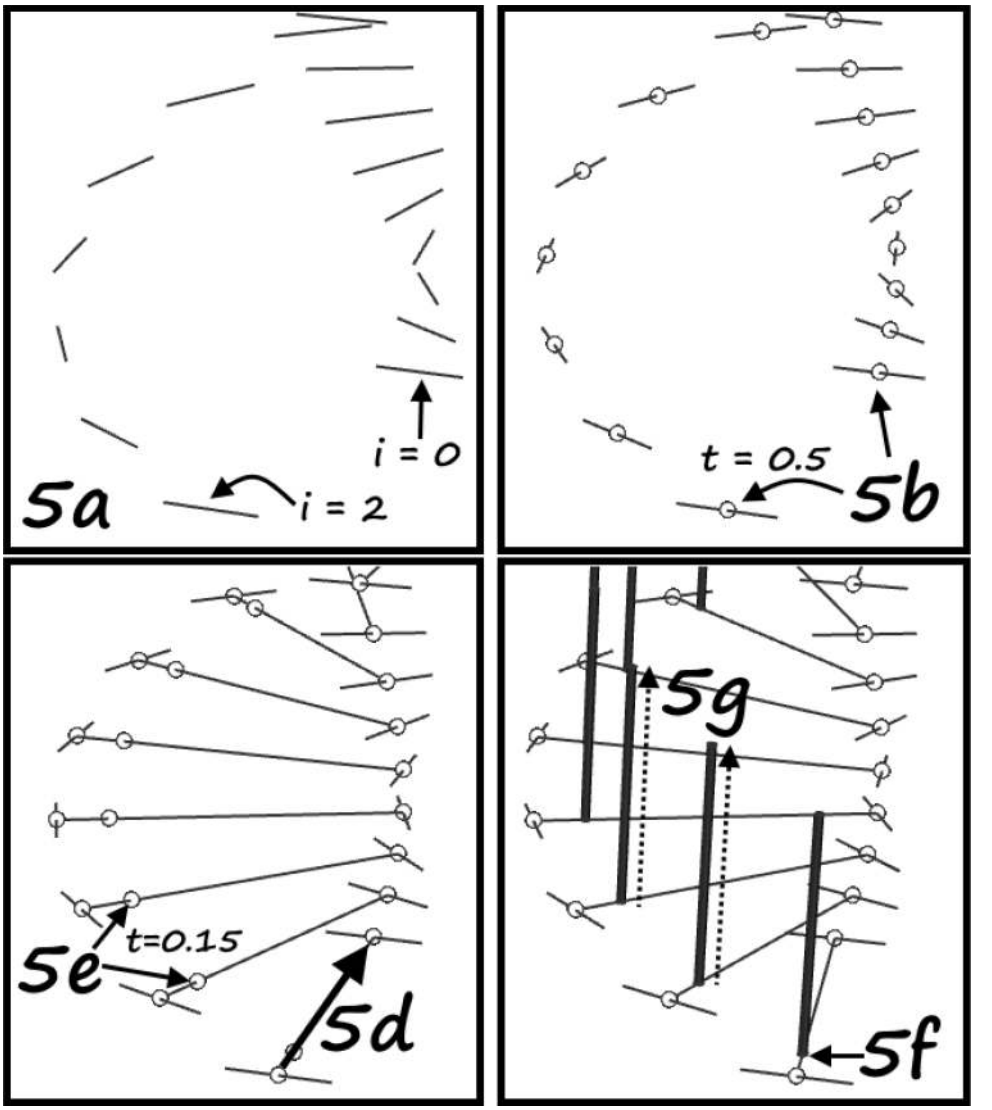


Figure 5-21. *Visual outputs of the nodes from stage 5 (Figure 5-20). We use List Item (5a) to access the tread edges (3a) at indices 0 and 2. We then use Evaluate Curve—Reparameterize (5b) with t=0.5 to extract the edges' midpoints, and Line (5d) to link them from outside in. Using another Evaluate Curve—Reparameterize (5e) with t=0.15, we find points slightly inset from the outer edges of the treads to use as the P input for Circle (5f), which we then Extrude (5g) into balusters*

Stage 6 of 6: Creating the Handrail

Figure 5-22 shows stage 6 of the staircase node graph from Figure 5-11, where we create the handrail by interpolating top center points of the balusters and piping the curve to create volume.

Remember that at the end of stage 5, we slid points inward from the outer edges of the treads to position the bottom circles of the balusters. We'll offset those points here with a Move node (6a), using the same vectors for extruding the balusters as the Motion (T) input, so the points end up at the top center of the balusters. We then Interpolate (6b) the top center points and feed the resulting curve through the Pipe node (6c) with Caps (E) = 2 to form the handrail surface with rounded caps.

You'll find the outputs of nodes 6a to 6c visualized in Figure 5-23.

Note Here is a summary of where you can find the nodes in stage 6 of Figure 5-22: Transform ➤ Euclidean ➤ Move (6a), Curve ➤ Spline ➤ Interpolate (6b), and Surface ➤ Freeform ➤ Pipe (6c).

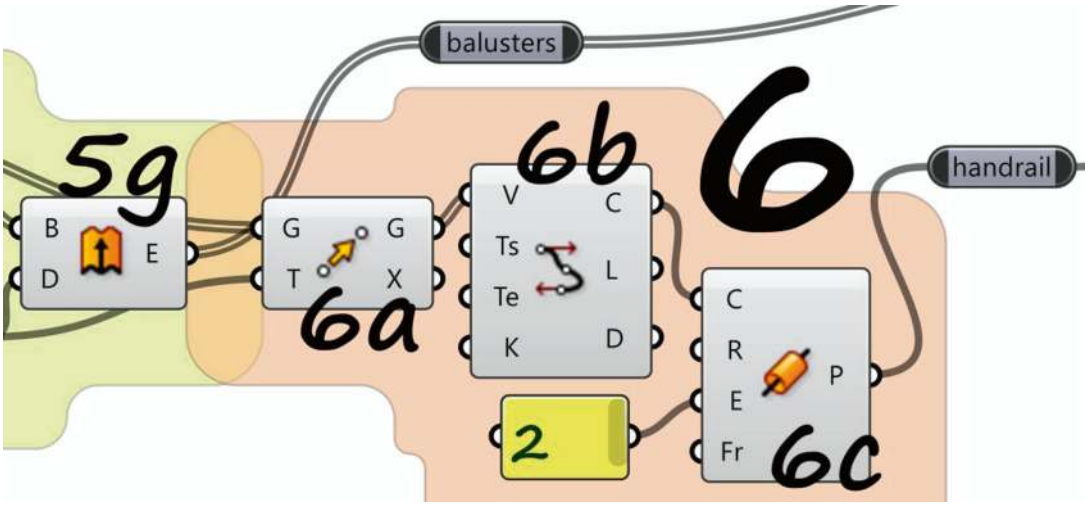


Figure 5-22. *Stage 6 of the spiral staircase node graph (Figure 5-11). We generate the handrail by using Move (6a) to offset the bottom center points of the balusters (5g) to the top, Interpolate (6b) these points, and Pipe (6c) along the curve to create the handrail surface*

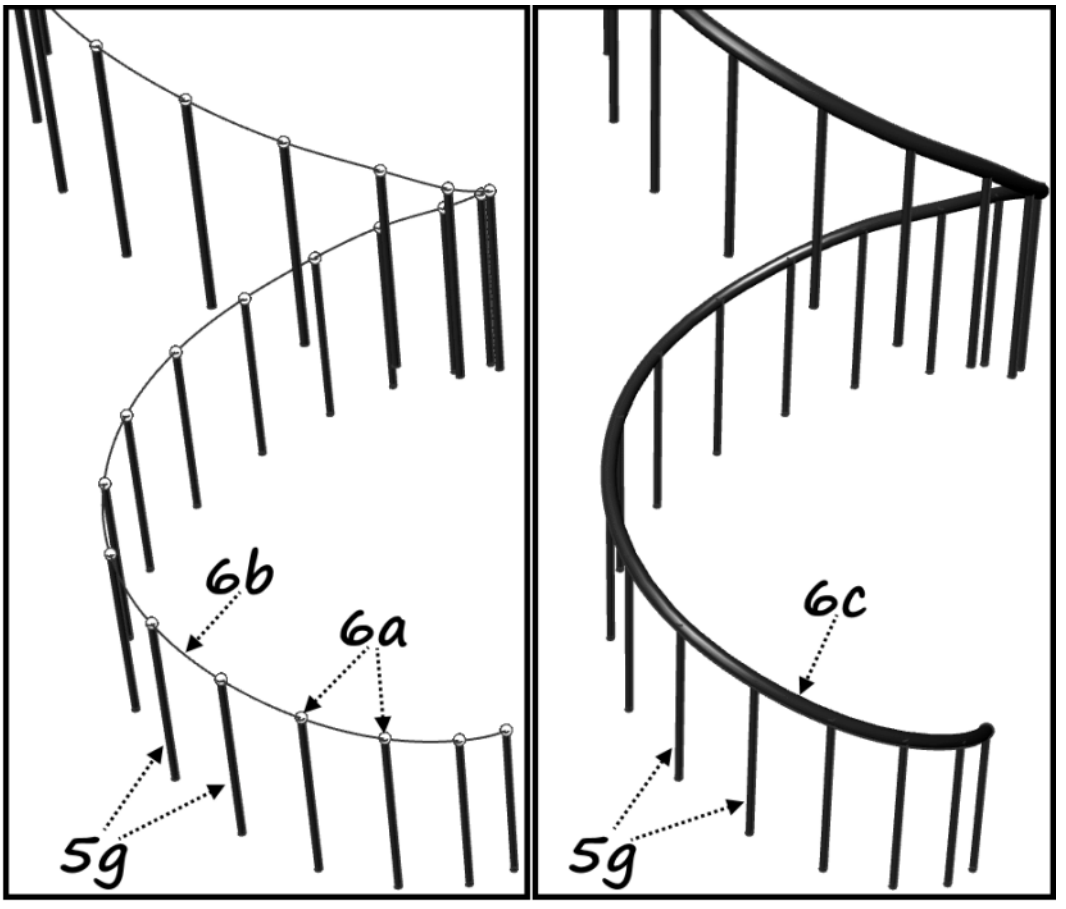


Figure 5-23. *Visual outputs of the nodes from Figure 5-22. We use Move (6a) to offset the bottom center points of the balusters (5g) to the top, Interpolate (6b) these points, then Pipe (6c) along the curve to create the handrail surface*

Tip: Combining Outputs of Sub-graphs for Easy Preview

We've completed the staircase node graph at this point; however, before we go, let's combine parts of the graph responsible for generating major components of the staircase by Shift-dragging them into a single Geometry node so we can preview the staircase more easily.

Figure 5-24 shows how we combine these nodes, with their corresponding previews shown in Figure 5-25. Node 1b (Figure 5-12) outputs the center pole, 4b (Figure 5-18) the tapered treads, 5g (Figure 5-20) the balusters, and 6c (Figure 5-22) the handrail. We Shift-drag these four nodes into a single Geometry node at the end of the entire graph, so we can click on only that Geometry node to preview the complete staircase. Note that we've also created relays in the wires leading into the Geometry node and right-clicked to label them based on the nodes they originate from, so it is easier to tell which wire is which.

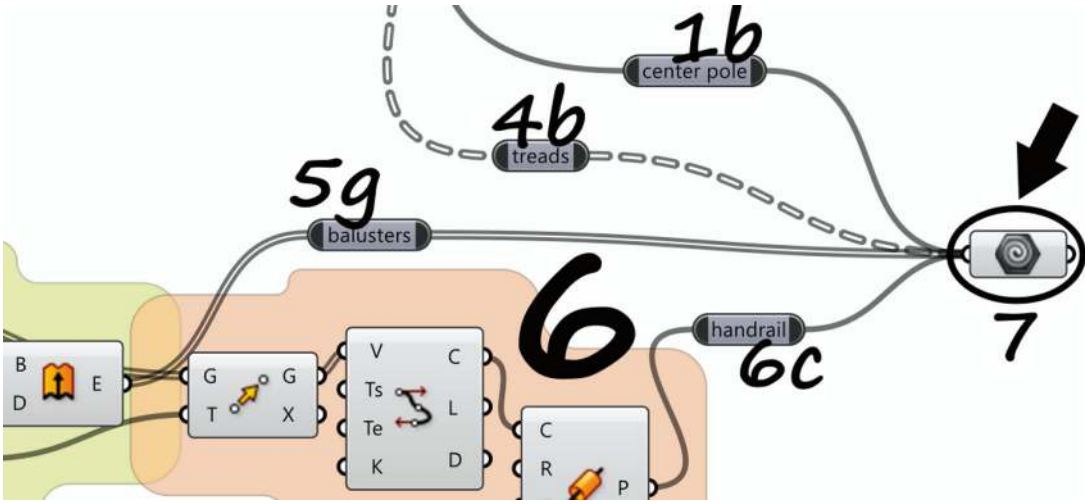


Figure 5-24. *We Shift-drag the following nodes that generate various parts of the spiral staircase into one Geometry node (7): Center pole (1b in Figure 5-12), treads (4b in Figure 5-18), balusters (5g in Figure 5-20), and handrail (6c in Figure 5-22). With this setup, you can click on the single Geometry node (7) to preview the complete staircase*

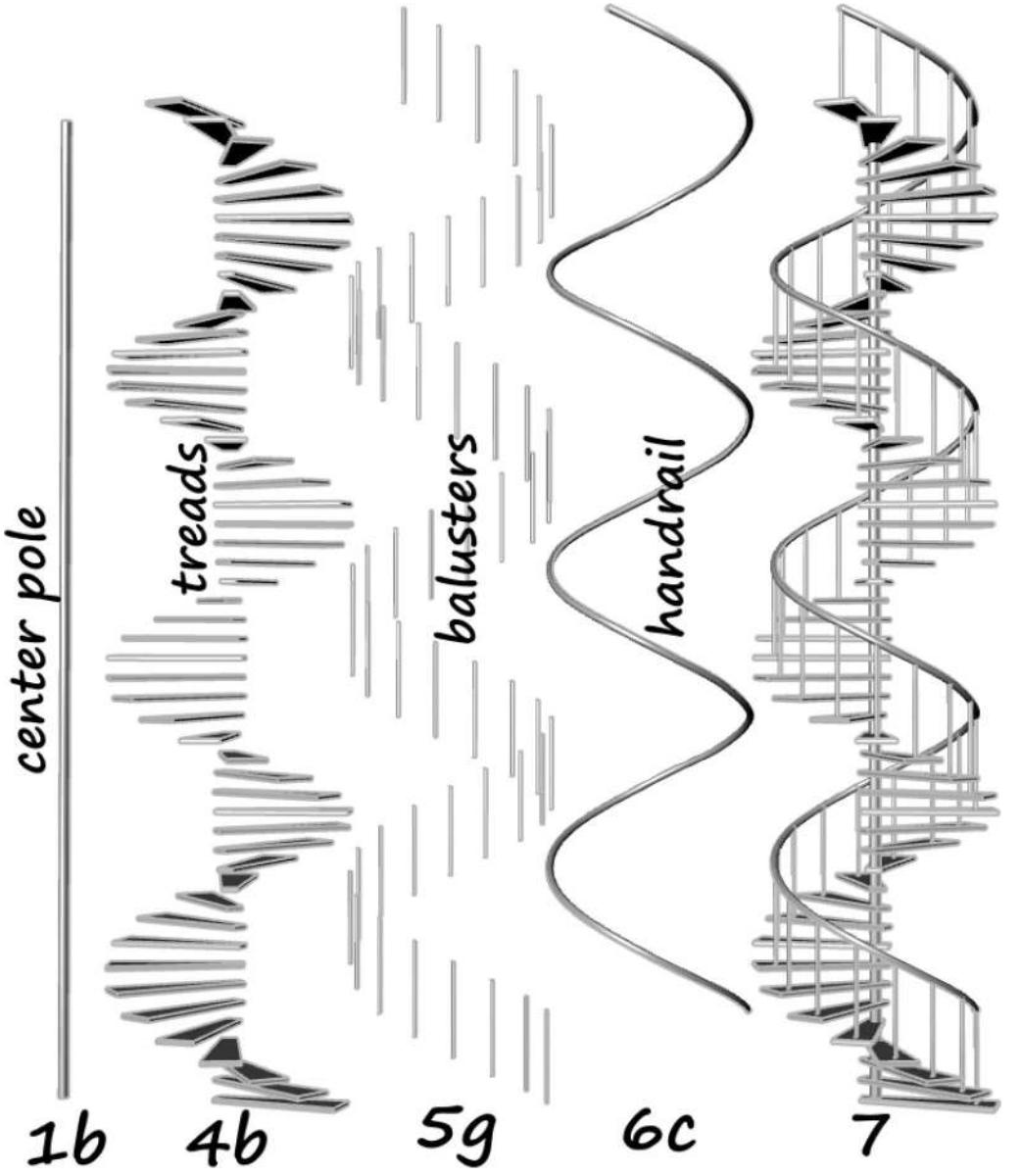


Figure 5-25. *Main components of the spiral staircase and the GH nodes that produce them. From left to right: Center pole (1b in Figure 5-12), treads (4b in Figure 5-18), balusters (5g in Figure 5-20), and handrail (6c in Figure 5-22). The far right shows the complete staircase (7 in Figure 5-24)*

Creating Variations

Since we use Number Sliders to control inputs for each of the major staircase components outlined in Figure 5-25, we can easily create variations like the ones shown in Figure 5-26: (1) is the original staircase generated by the unaltered node graph from Figure 5-11. (2) has fewer treads, with the treads insetting more toward the center pole. (3) has a lower total height but otherwise unchanged. (4) has shorter balusters.

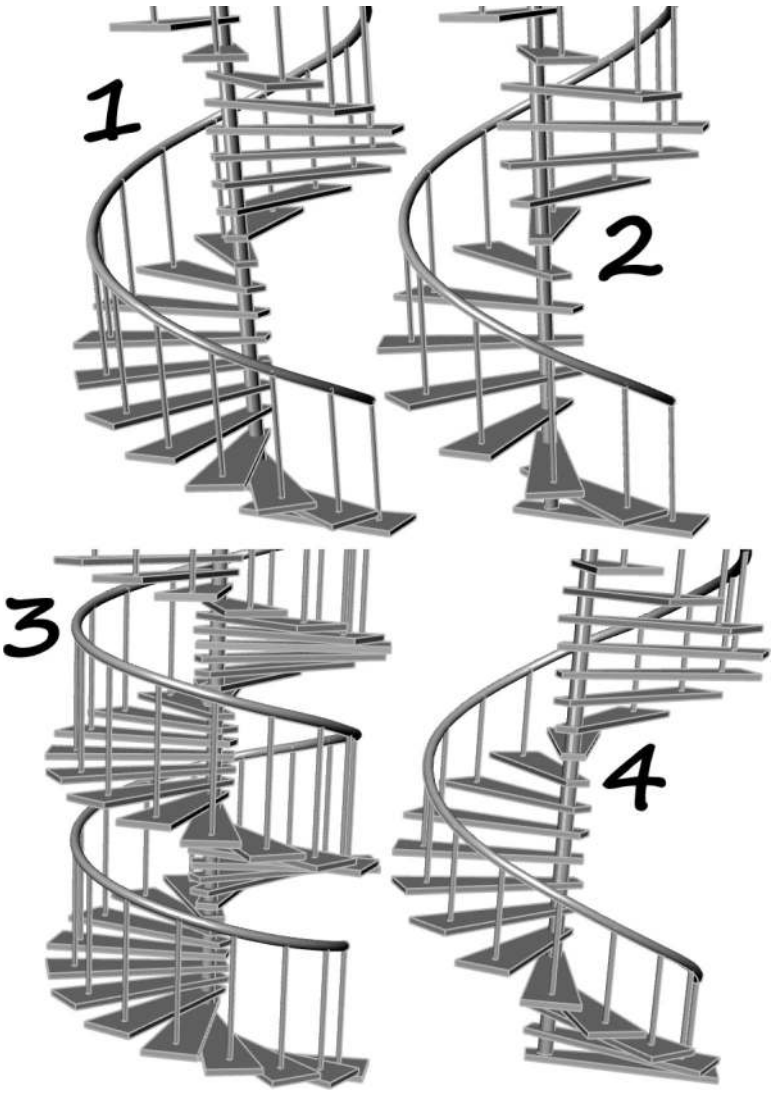


Figure 5-26. *Variations of the spiral staircase generated by the node graph discussed throughout this section. (1) The original; (2) fewer treads, with treads insetting more toward the center column; (3) lower total height; (4) shorter balusters*

Summary

In this chapter, you worked through three loops-and-spirals-themed workshops. The first workshop centered around Möbius strips—a type of non-orientable surface that always takes you back to the starting point when you go down the strip lengthwise. You first created a Möbius strip by making a half-twist in a rectangular paper strip and taping its ends together. You then experimented with creating Möbius surfaces in Grasshopper using three different methods. The first method uses the Mobius Surface node from the LunchBox plugin and various paneling nodes to create wireframe variations. The second method bends a rectangular Brep with the Twist node then conforms the Brep to a circle using the Flow node. The third method computes evenly spaced points using math formulas, interpolates these points into parallel curves length- or widthwise along the strip, and lofts the curves into a surface. You practiced using both Expression nodes and a Python 3 Script node to implement the math formulas.

The second workshop showed you how to create a logarithmic spiral curve by interpolating points computed from math formulas in Python. You spawned evenly spaced planes centered at points along the spiral to position circular curves representing the cross-sections of a shell structure. Distances from the plane centers to the spiral origin are used to approximate the circles' radii. You then lofted the circles to create the shell surface.

The third workshop showed you how to create a parametric spiral staircase generator. The node graph was broken into six stages. In stage 1, you created the center pole and a spiral curve around the pole. In stage 2, you positioned rectangular curves along the spiral to create outlines for the treads. In stage 3, you tapered the treads toward the center pole. In stage 4, you extruded the treads along their normals to give them thickness. Balusters were extruded based on positions inset from the treads' outer edges in stage 5, and the handrail was derived in stage 6 by interpolating points at the top centers of the balusters and piping the curve.

References

Bourke, Paul. "Application of Parametric Equations by Paul Bourke". *David Pleacher's Web Pages*, 23, January 2026, https://www.pleacher.com/handley/lessons/calc2004/param.html.

Weisstein, Eric W. "Logarithmic Spiral." From *MathWorld*--A Wolfram Resource. https://mathworld.wolfram.com/LogarithmicSpiral.html

CHAPTER 6

Stacking and Voxelizing

In this chapter, we'll tackle a series of workshops centered on the theme of stacking blocks and voxelizing 3D surfaces. We'll start simple by stacking blocks in straight columns and create variations by scaling and rotating the blocks. We'll then extend the "column" concept by randomizing the dimensions of the blocks and stacking along curves like sine waves. The final stacking example generalizes the concept to 3D and creates a building model with randomized blocks and poles.

The second part of the chapter discusses how to *voxelize* models to create a "pixelated," blocky appearance in the 3D sense. We'll work through two examples; the first of which creates voxels on a divided surface, leaving the inside of the model hollow. The second treats the model as a solid and fills its volume with either nonuniform or uniform blocks.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch6` folder, where you'll find `*.gh` files named after the figures in which they appear, along with a short description. For example, `Figure6-1_6-2_stacking_column_scale.gh`. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, you can build your own node graphs by following the steps described in the text or start with the downloaded `*.gh` files and modify their parameters to experiment.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_6

Making Stacks: Columns

Let's start with the simplest way to make a stack—stacking upward to form a column. We'll work through three examples: The first example uses blocks of decreasing sizes toward the top. The second example rotates blocks in steady increments. The third example creates blocks of random thicknesses, decreases block sizes smoothly with Taper, and shifts stack layers horizontally by random amounts.

Column with Scaling

Figure 6-1 shows the node graph where we create a vertical stack of blocks that decrease in size steadily toward the top. The graph is broken into three stages. Stage 1 creates a vertical polyline. Stage 2 divides the polyline evenly and computes the scaling factors for the blocks based on their positions along the column. Stage 3 extrudes the blocks from polygonal curves and scales them.

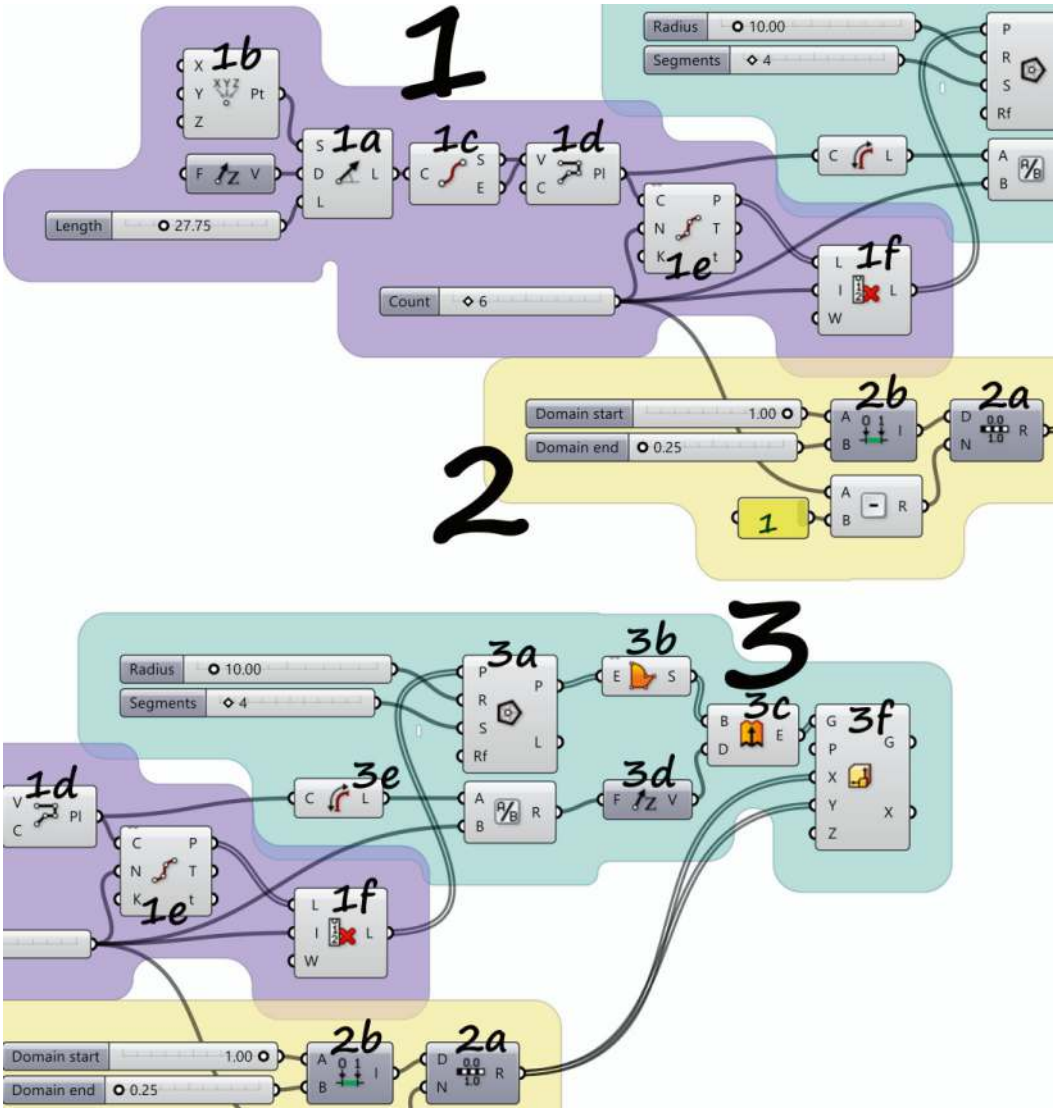


Figure 6-1. *Node graph for stacking blocks into a column, with blocks decreasing in size incrementally toward the top. Stage 1 evenly divides the vertical line representing the stack's central axis. Stage 2 computes the scale factors for the blocks. Stage 3 extrudes the blocks at line division points and scales them*

Stage 1: Creating the Central Axis and Dividing It into Points

Let's zoom in on stage 1 of Figure 6-1. Our goal is to create the central axis of the column and divide it into points for positioning the blocks later. We start by creating a line segment using Line SDL (1a). Since we want the line to go straight up, we set Start (S) to (0, 0, 0) using Construct Point (1b) with default parameters, Direction (D) to Unit Z, and Length (L) to a Number Slider to vary the height of the stack.

Next, we use Divide Curve (1e) to split the line into even chunks. Since Divide Curve only works on curves or polylines, *not* lines, we must link Line SDL (1a) to End Points (1c) to extract its vertices, followed by PolyLine (1d) to convert the line to a polyline. We use a Number Slider to control the Count (N) for Divide Curve (1e) to vary the number of blocks in the stack.

There is a slight problem—Divide Curve (1e) gives us the division points *plus* both end points of the polyline; however, we want to exclude the end point at the end (top) of the polyline, so we can position a polygon at each point and extrude *upward* into a block, as shown in Figure 6-2. Therefore, we link Divide Curve (1e) to Cull Index (1f) with i set to the number of blocks (the Count (N) slider), which will eliminate the last point.

Note Where to find the nodes in stage 1 of Figure 6-1: Curve ➤ Primitive ➤ Line SDL (1a), Vector ➤ Point ➤ Construct Point (1b), Curve ➤ Analysis ➤ End Points (1c), Curve ➤ Spline ➤ PolyLine (1d), Curve ➤ Division ➤ Divide Curve (1e), and Sets ➤ Sequence ➤ Cull Index (1f).

In Grasshopper, a PolyLine consists of a series of two or more points linked by straight line segments, whereas a Line or Line SDL is a straight line with two end points.

Stage 2: Computing the Scaling Factors for the Blocks

In stage 2 (Figure 6-1), we compute the XY scale factors for the blocks based on their positions along the stack. We use Construct Domain (2b) with two Number Sliders to define the start (A) and end (B) of the scale. As shown, we set A to 1.00 and B to 0.25, which means the bottom block is unaltered, while the top block is 25% of its size, as shown in Figure 6-2.

To compute the scale factors for the blocks, we link Construct Domain (2b) to Range (2a), with N = number of blocks minus one. As shown in Figure 6-1, the Count (N) slider for Divide Curve (1e) is set to six, so we get six blocks. By setting N for Range (2a) to 6-1 = 5, we get the following list of six numbers:

1, 0.85, 0.7, 0.55, 0.4, and 0.25

which starts and ends at the values we specified through Construct Domain (2a) and decreases in steps of 0.15, producing the stack shown in Figure 6-2.

Note Where to find the nodes in stage 2 of Figure 6-1: Sets ➤ Sequence ➤ Range (2a), and Maths ➤Domain ➤Construct Domain (2b).

Stage 3: Creating the Blocks and Scaling Them

In stage 3 (Figure 6-1), we create the blocks at the polyline's division points and scale them using factors derived in stage 2. As shown in Figure 6-2, the output of Cull Index (1f) gives us the start points of the polyline's segments from bottom to top. Linking Cull Index (1f) to the P input of Polygon (3a) implicitly triggers Grasshopper to create planes parallel to XY with their origins coinciding with those points. By using Number Sliders to control the R and S inputs for Polygon (3a), we can change the size and shape of the blocks at will. R = 10.00 and S = 4 make the square stack shown in Figure 6-2.

To turn the polygon curves into blocks, we link Polygon (3a) to Boundary Surfaces (3b) followed by Extrude (3c), as visualized in Figure 6-2. To control the direction and amount of extrusion, we use the Length node (3e) to measure the polyline as a whole and divide it by the number of segments to get the height per block. Since we've oriented the polyline to go up toward +Z, we simply use Unit Z for the direction (3d).

The last step is to scale the blocks using the factors derived in stage 2, so they decrease in size as we ascend the tower. To do so, we link Extrude (3c) to Scale NU (3f) and Shift-drag the output of the Range node (2a) to both the X and Y inputs while leaving Z at the default value of 1, to scale the blocks horizontally but leaving their heights unchanged. The output of Scale NU (3f) is shown in Figure 6-2.

Note Where to find the nodes in stage 3 of Figure 6-1: Curve ➤ Primitive ➤ Polygon (3a), Surface ➤ Freeform ➤ Boundary Surfaces (3b), Surface ➤ Freeform ➤ Extrude (3c), Vector ➤ Vector ➤ Unit Z (3d), Curve ➤ Analysis ➤ Length (3e), and Transform ➤ Affine ➤ Scale NU (3f).

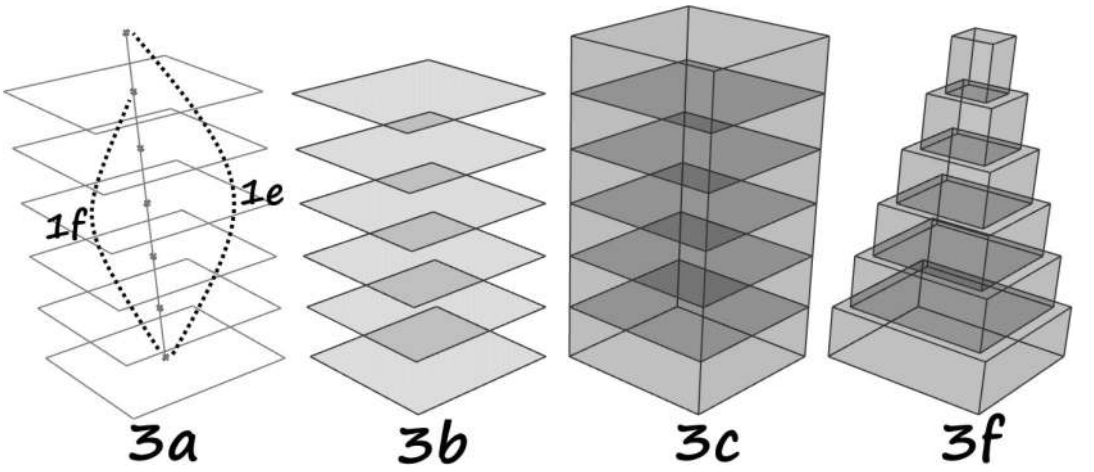


Figure 6-2. *Visual outputs of nodes from Figure 6-1. From left to right: Stack of polygonal curves (3a) centered at even division points along the polyline, surfaces created with the curves as boundaries (3b), surfaces extruded into blocks (3c), and blocks incrementally scaled toward the top (3f). The far left also shows the output of Divide Curve (1e), which includes the end points of the curve as a whole. We use End Points (1f) to separate the Start (S) and End (E) of the segments so we can exclude the last point*

Column with Rotation

Instead of descending sizes, we can create stacks with incremental rotation with the modification shown in Figure 6-3. We link the Range node (2a) and a Pi node (2c) to the two inputs of a Multiplication node (2d) to get the list of scale factors multiplied by Pi radians. As discussed in stage 2 of the previous section, by setting N for Range (2a) to 6-1 = 5, we get the following list of six numbers:

1, 0.85, 0.7, 0.55, 0.4, and 0.25

Therefore, by multiplying the output of Range (2a) by Pi (2c), we get

$$\neq,0.85\neq,0.7\neq,0.55\neq,0.4\neq,0.25\neq radians$$

which is a list of rotation angles that steadily decrease by 0.15Pi radian per step, which is about 27 degrees.

To apply the rotations, we replace the Scale NU node (3f) in Figure 6-1 with a Rotate node, as shown in Figure 6-3, and link Extrude to its G input and the angles (2d) to its A input. Some examples produced with this modification are shown in Figure 6-4, with the Segments (S) input of the Polygon node (3a) in Figure 6-1 dialed to 3, 4, 5, and 6 to make blocks with cross-sections that are triangles, squares, pentagons, and hexagons, respectively.

Note Where to find the nodes in Figure 6-3: Sets ➤ Sequence ➤ Range (2a), Maths ➤ Util ➤ Pi (2c), Maths ➤ Operators ➤ Multiplication (2d), and Transform ➤ Euclidean ➤ Rotate (3f)

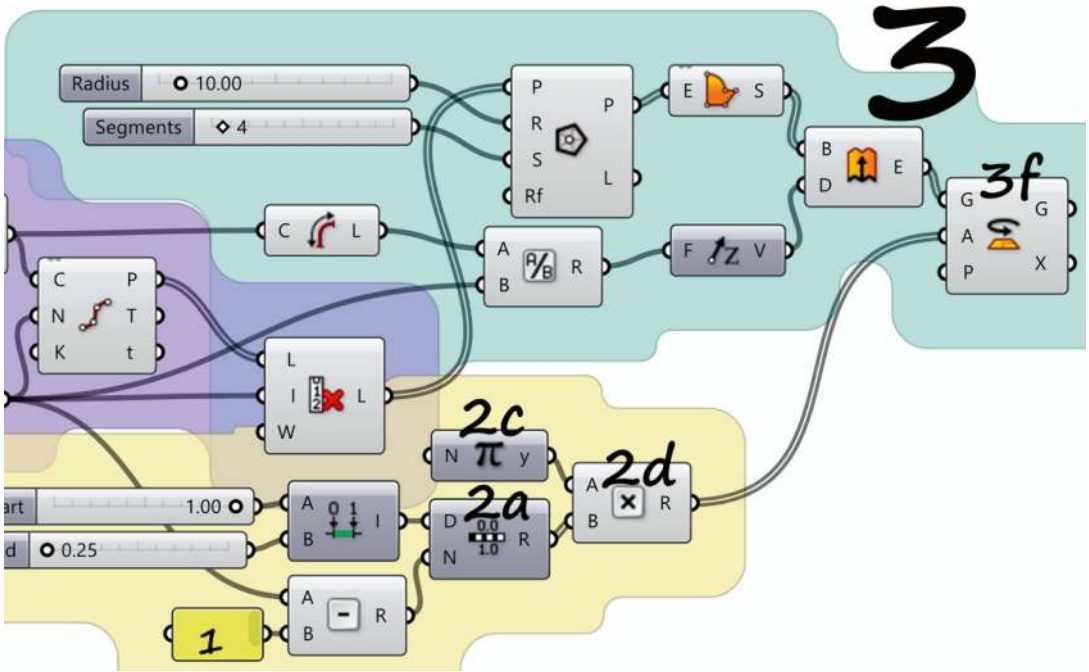


Figure 6-3. *Modifying the node graph from Figure 6-1 to create incremental rotations along the stack instead of scaling. The factors output by the Range node (2a) is multiplied (2d) by Pi (2c) to create a list of angles. The Scale NU node is then replaced by Rotate (3f) to rotate the blocks*

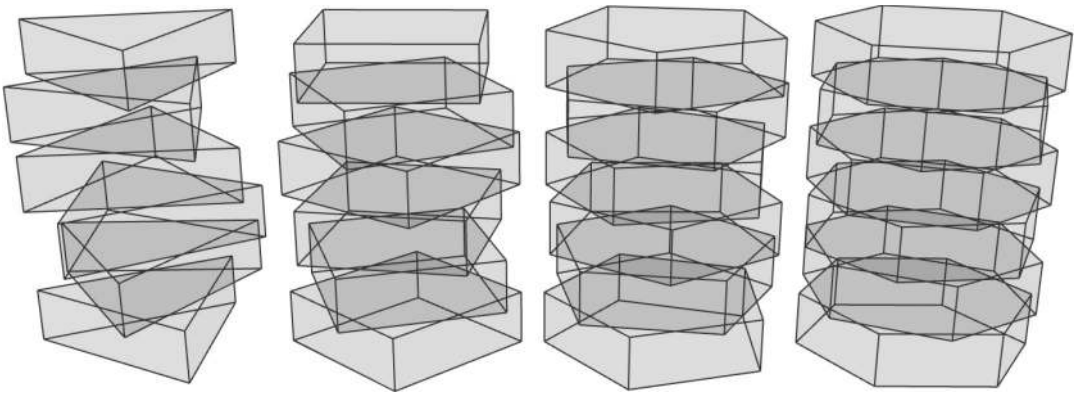


Figure 6-4. *A few stacks generated with the graph modification in Figure 6-3. Each block is rotated by an evenly divided share of a total rotation. The shape of the blocks is changed by setting the Segments (S) input for Polygon (3a) in Figure 6-1 to 3, 4, 5, and 6, respectively.*

Column with Variable Block Thickness, Tapering, and Random Horizontal Offsets

We'll now expand on the node graph for creating a straight stack from Figure 6-1. Instead of forming blocks with even thicknesses, we'll divide the column's central axis into random length segments to use as heights of the blocks. In Figure 6-1, we scaled the blocks incrementally at fixed steps—here we'll use a Taper node (4a in Figure 6-5) to scale the blocks smoothly in a stepless manner. At the end of the section, we'll wrap up by adding another (fifth) stage to the node graph to add random horizontal offsets to the blocks with the help of Python.

Figure 6-5 shows stages 1 to 4 of the modified node graph, and Figure 6-6 shows stage 5. In stage 1, we create the column's central axis and convert it to a polyline. In stage 2, we divide the polyline into random length segments with the help of Python (2a to 2e), followed by Evaluate Length (2f) to derive the Parameter (t) values based on the segment lengths, and Shatter (2g) to extract the division points. In stage 3, we extrude the blocks based on their thicknesses, and in stage 4, we use Taper (4a) to scale the blocks steplessly. In the next few sections, we'll zoom in on each stage and examine it in detail.

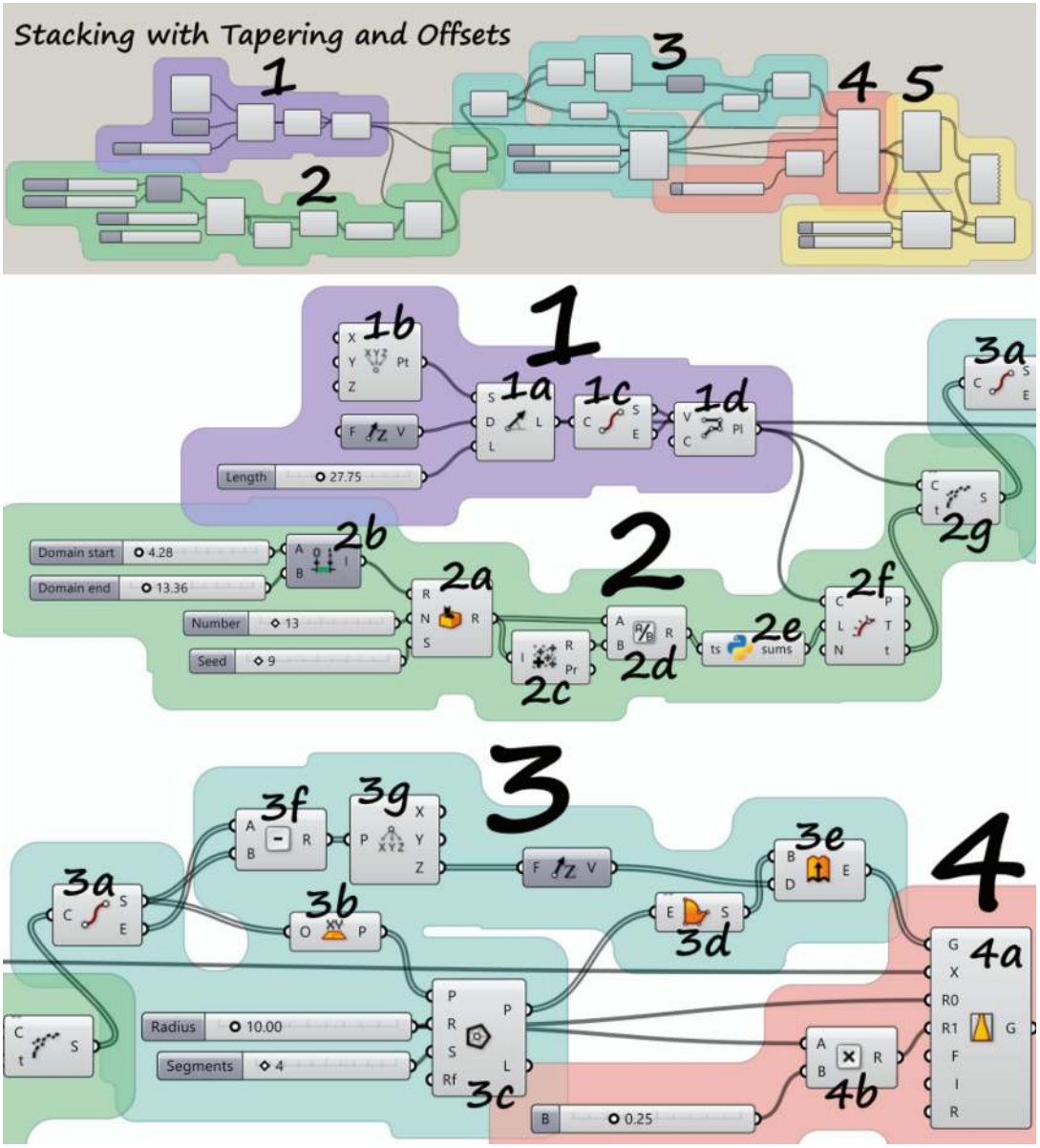


Figure 6-5. *Node graph for creating a stack of blocks with varying thickness. The stack is tapered toward the top, with each block offset by a random amount horizontally. Stage 1 creates a vertical polyline. Stage 2 randomly divides the line. Stage 3 extrudes a polygonal block at each of the line's division points. Stage 4 tapers the stack so it's smaller at the top. Stage 5 is shown in Figure 6-6 and offsets each block by a random amount horizontally*

Stage 1: Creating the Central Axis and Converting It into a Polyline

We start by creating a line segment using Line SDL (1a). Since we want the line to go straight up, we set S to (0, 0, 0) using Construct Point (1b) with default parameters, Direction (D) to Unit Z, and Length (L) to a Number Slider to vary the height of the stack.

We'll be using Evaluate Length (2f) and Shatter (2g) in stage 2, which only work on curves or polylines, *not lines*—therefore, we'll preemptively convert the line to a polyline, by linking Line SDL (1a) to End Points (1c) to extract its vertices, followed by PolyLine (1d) to convert it to a polyline.

Note Where to find the nodes in stage 1 of Figure 6-5: Curve ➤ Primitive ➤ Line SDL (1a), Vector ➤ Point ➤ Construct Point (1b), Curve ➤ Analysis ➤ End Points (1c), and Curve ➤ Spline ➤ PolyLine (1d).

Stage 2: Deriving Random Thicknesses for the Blocks

In stage 2, we'll break up the polyline into random length segments, which in turn will become random heights for the blocks. We'll use *normalized lengths*, which are ratios of segment lengths against the polyline (or curve) as a whole, which lets us evaluate the curve without knowing its makeup ahead of time. Let's see how this is done step by step.

First, we'll use the Random node (2a) to generate N = 13 numbers in the range given by Construct Domain (2b) to represent the lengths of 13 line segments we'll cut. We then use a Division node (2d) to divide the 13 lengths by their sum (tallied by Mass Addition (2c)) to get the *normalized lengths*, or their ratios against the curve as a whole.

The next step is to find the *cumulative normal lengths*, which are the sum of normalized lengths up to and include each segment or the ratios of running lengths up to and include each segment against the whole curve. We use a Python 3 Script (2e) node to do this work with the code shown in Listing 6-1 and Access and Type Hints in Table 6-1.

In Listing 6-1, we take the list of *normalized lengths*, `ts`, and iterate through it in a `for` loop. In each iteration of the loop, we add up the lengths up to and include the length at the current index `i` and append the result to the list `sums`, which is output by the Python 3 Script node (2e) once the loop concludes.

Listing 6-1. Code inside the Python 3 Script node (2e) from Figure 6-5 (see `/Ch6/Figure6-5_6-7_stacking_column_taper_and_shifts.gh`)

```
num_items = len(ts)
sums = []
for i in range(1, num_items+1):
    sums.append(sum(ts[:i]))
```

Table 6-1. *Access and Type Hints for the Python 3 Script node (2e) in Figure 6-5*

Name	Input or Output	Access	Type Hints
ts	input	List Access	float
sums	output	N/A	float

Our goal is to find the starting point of each segment to use as the reference location for creating blocks. To do so, we link the *cumulative normalized lengths* output by the Python 3 Script node (2e) to the L input of Evaluate Length (2f) with PolyLine (1d) as the C input to get the t parameter at the start of each segment. We then link the t values to Shatter (2g) to retrieve the segments, and End Points (3a) to isolate the start points of the segments.

Note Where to find the nodes in stage 2 of Figure 6-5: Sets ➤ Sequence ➤ Random (2a), Maths ➤ Domain ➤ Construct Domain (2b), Maths ➤ Operators ➤ Mass Addition (2c), Maths ➤ Operators ➤ Division (2d), Maths ➤ Script ➤ Python 3 Script (2e), Curve ➤ Analysis ➤ Evaluate Length (2f), and Curve ➤ Division ➤ Shatter (2g).

Recall that for a Python 3 Script node, you right-click an input to change its name and set its Access Hint (Item, List, or Tree Access) and Type Hint.

You also right-click an output to set its name and Type Hint. There is no Access Hint for an output.

Using "Pseudo Random" to Create Repeatable Randomness

Note that Random (2a in Figure 6-5) produces *pseudo random* numbers, which appear to be random but are actually generated by algorithms that are deterministic and repeatable. Given the same "Seed" input (an integer), Random will produce an identical sequence of numbers every time. This is not a bad thing, since you can hook a Number Slider to the Seed input, drag it around until the graph produces something you like, then share the graph with other Grasshopper users knowing that they will get the exact same output.

Stage 3: Extruding the Blocks Based on Their Thicknesses

In stage 3 (Figure 6-5), we'll create polygons centered at the random curve division points from stage 2 and extrude them into blocks such that no gap exists between neighbors on the stack.

We start by linking the list of curve segments (2g) to End Points (3a) to separate their Start (S) and End (E) points, which are at the bottom and top of each segment, respectively. By linking the start points to the P input of Polygon (3c), we implicitly request Grasshopper to convert them to planes parallel to XY with origins at those points. Therefore, the polygon curves produced will center at those points.

To find the *heights* of the blocks, we subtract the start points from the end points (3f) then separate the X, Y, and Z components of the difference with Deconstruct (3g)—the Z value, which is the vertical offset between the start and end points of each segment, is then the height of each block.

We finish by using Boundary Surfaces (3d) to create surfaces bordered by the polygon curves and Extrude (3e) them into blocks using the heights we've just derived (3g). The process of this stage is visualized at the top of Figure 6-7.

Note Where to find the nodes in stage 3 of Figure 6-5: Curve ➤ Analysis ➤ End Points (3a), Vector ➤ Plane ➤ XY Plane (3b), Curve ➤ Primitive ➤ Polygon (3c), Surface ➤ Freeform ➤ Boundary Surfaces (3d), Surface ➤ Freeform ➤ Extrude (3e), Maths ➤ Operators ➤ Subtraction (3f), and Vector ➤ Point ➤ Deconstruct (3g).

Stage 4: Using the Taper Node to Scale the Stack Steplessly

In stage 4, we'll gradually scale the stack in a stepless manner using the Taper node (4a). We link the extruded blocks (3e) to Taper's G input and Line SDL (1a) to the X input, which defines the axis for tapering. Note that 1a is the original Line object *before* conversion to polyline. R0 and R1 control the radius at the start and end of the taper axis, respectively. We use the same radius slider for Polygon (3c) as we do for R0, so the bottom of the stack has the original radius, and R1 equals R0 multiplied with a slider (set to 0.25 as shown). The result is the stack ranging by scale from 1.00 to 0.25 smoothly in a stepless way, as shown in Figure 6-7.

Note Where to find the nodes in stage 4 of Figure 6-5: Transform ➤ Morph ➤ Taper (4a), and Maths ➤ Operators ➤ Multiplication (4b).

Stage 5: Shifting the Blocks Horizontally by Random Amounts

In stage 5, we'll use a Python 3 Script node (5a) to compute vectors for randomly offsetting blocks horizontally, as shown in Figures 6-6 and 6-7. The code for the node is shown in Listing 6-2, with Access and Type Hints summarized in Table 6-2.

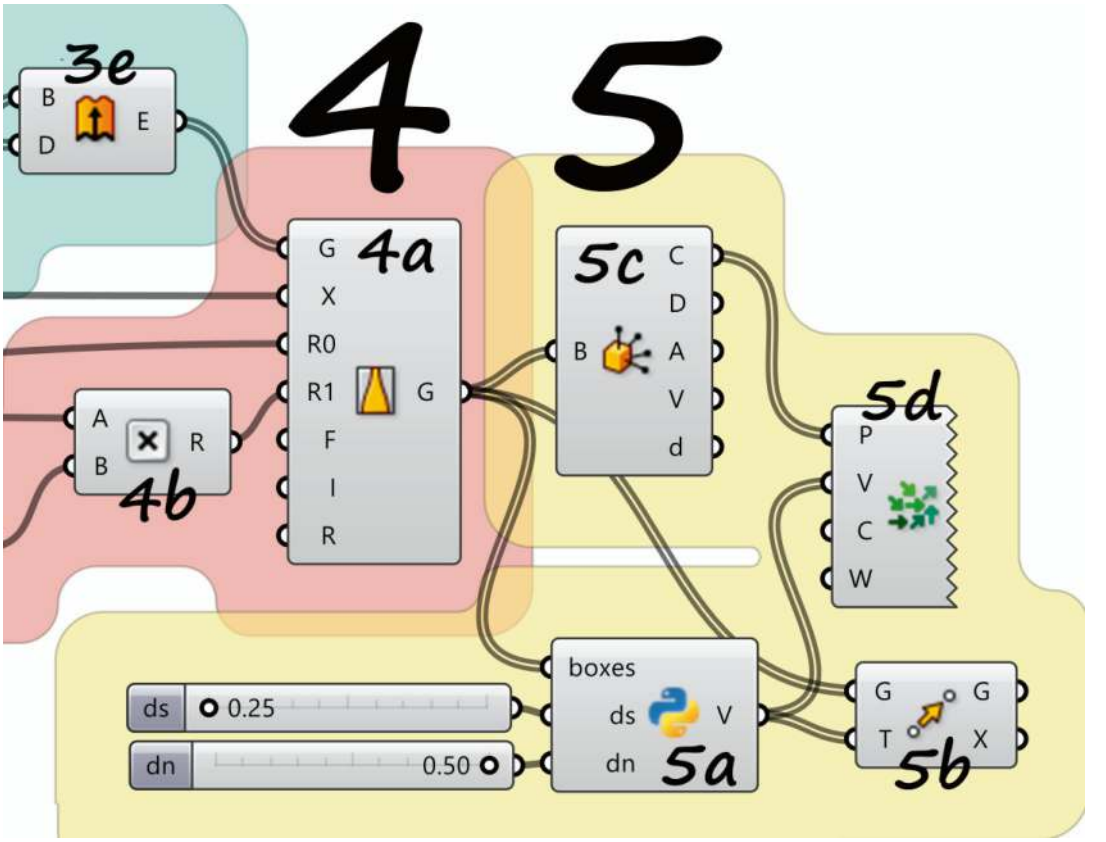


Figure 6-6. *Stage 5 of the node graph from Figure 6-5. For each block in the stack, the Python 3 Script node (5a) finds a vector to offset the block toward a random corner of the stack's bounding box. Once the vectors are found, the movements for all blocks are done in one shot with a Move node (5b)*

Let's look at the code for the Python 3 Script node (5a) more closely. The code takes three inputs—`boxes`, the stack of Breps, and `ds` and `dn`, which define the range to scale the offset vectors. The code outputs a single list, `V`, of offset vectors, one for each Brep.

In Listing 6-2, we start by importing the `ghpythonlib.components` module as the shorthand `ghcps` so we can call Grasshopper components as Python functions. We also import two functions from Python's built-in `random` module, `randint` and `uniform`, for generating pseudo random whole and decimal numbers within a given range.

After the imports, we initialize the output list `V` to an empty list and iterate through the list of input Breps in a `for` loop. In each iteration of the loop, we call `ghcps.BoxCorners` to retrieve the eight corner vertices of the Brep's bounding box. We only need the first four, which are at the bottom of the box, so we use *anonymous variables*

(underscores) to receive the second four returned vertices, which tell Python they are throw-away values. We then call `randint(0, 3)` to pick a random number from 0, 1, 2, and 3 to decide which of the four bottom vertices of the bounding box to use as the *target* of the offset vector.

Next, we call `ghcps.BoxProperties` to retrieve the Brep's center (`b_center`) and discard the rest of the return values. We then create a vector from the Brep's center to the random bounding box vertex we've previously selected by calling `ghcps.Vector2Pt`.

In the next few lines, we finalize the vector by zeroing out the z component to make it horizontal and scaling it based on the input `ds` and `dn` values. We do so by extracting the vector's x and y components using `ghcps.Deconstruct` and discarding the z value. We then call `uniform` to get a pseudo random decimal number in the range specified by `ds` and `dn` and reconstitute the vector by calling `ghcps.VectorXYZ(x*scale, y*scale, 0)` with the scaled x and y values and z = 0. Lastly, we append the finalized vector to the output list `V`.

A visualization of the stack with random offset vectors applied is shown in Figure 6-7.

Listing 6-2. Code inside the Python 3 Script node (5a) from Figure 6-6 (see `/Ch6/Figure6-5_6-7_stacking_column_taper_and_shifts.gh`)

```
import ghpythonlib.components as ghcps
from random import randint, uniform

V = []
for b in boxes:
    ca, cb, cc, cd, _, _, _, _ = ghcps.BoxCorners(b)
    c = [ca, cb, cc, cd][randint(0, 3)]
    b_center, _, _, _, _ = ghcps.BoxProperties(b)
    raw_v, _ = ghcps.Vector2Pt(b_center, c, False)
    x, y, _ = ghcps.Deconstruct(raw_v)
    scale = uniform(ds, dn)
    v_zero_z, _ = ghcps.VectorXYZ(x*scale, y*scale, 0)
    V.append(v_zero_z)
```

Table 6-2. *Access and Type Hints for the Python 3 Script node (5a) in Figure 6-6*

Name	Input or Output	Access	Type Hints
boxes	input	List Access	Box
ds	input	Item Access	float
dn	input	Item Access	float
V	output	N/A	Vector3d

Note Where to find the nodes in stage 5 of Figure 6-6: Maths ➤ Script ➤ Python 3 Script (5a), Transform ➤ Euclidean ➤ Move (5b), Surface ➤ Analysis ➤ Box Properties (5c), and Display ➤ Vector ➤ Vector Display Ex (5d).

Visualizing Vectors in the Rhino Viewport

You can preview Grasshopper vectors in the Rhino viewport with the Vector Display Ex node (5d in Figure 6-6), which takes the start points of the vector through its P input and the end points through its V input. You can optionally configure the vectors' Colour (C) and Width (W, which is the line weight).

To visualize the offset vectors output by the Python 3 Script node (5a), we link Taper (4a) to Box Properties (5c) to retrieve the blocks' center points (C) to use as start points (P) for Vector Display EX (5d). The output of the Python 3 Script node (5a) are used as the end points (V). The results are shown in Figure 6-7.

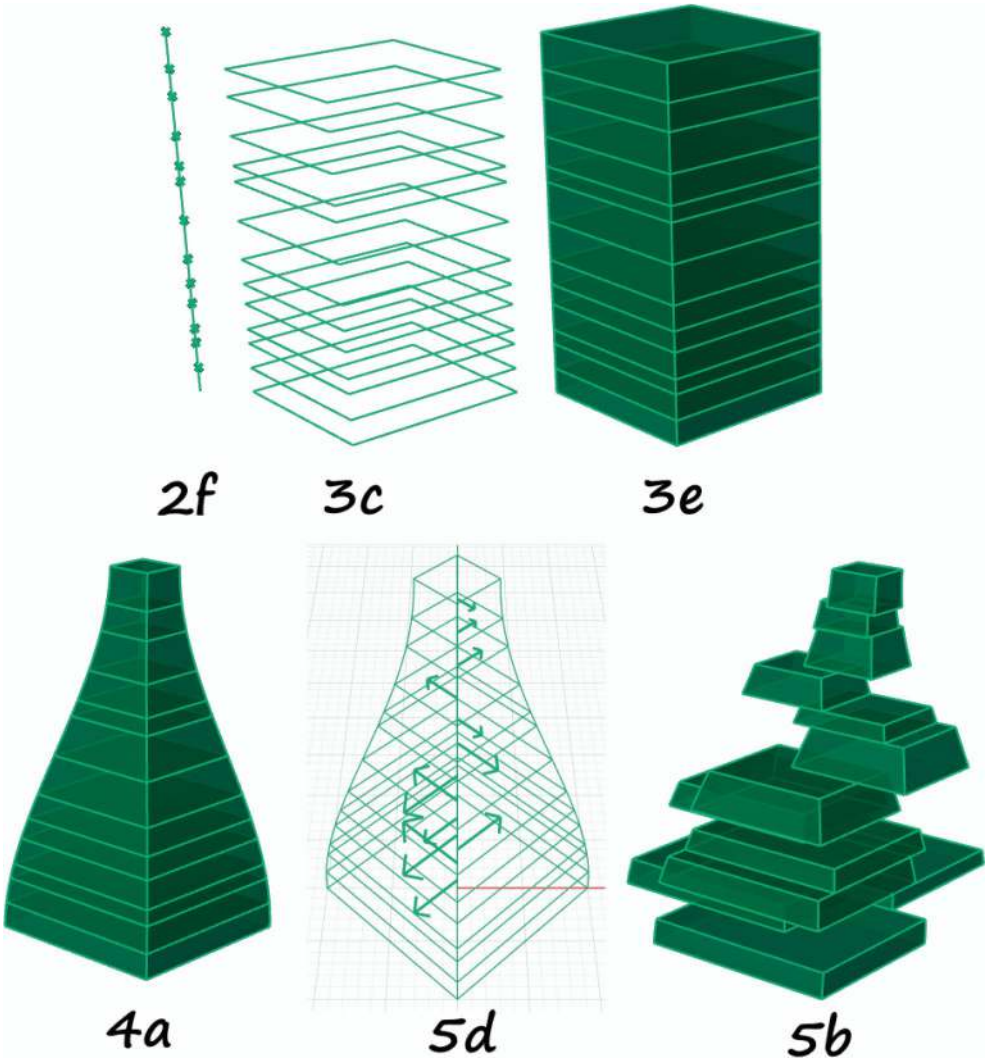


Figure 6-7. *Visual outputs of nodes from Figures 6-5 and 6-6. We divide the curve by evaluating it with random t values (2f), create square curves centered at the division points (3c), extrude the squares into blocks (3e), taper the blocks (4a), and move them by random amounts (5d) toward the stack's bounding box corners (5b)*

Making Stacks: Random Blocks Along an Arbitrary Curve

So far, we've constructed stacks as straight columns. In this section, we'll stack along arbitrary curves such as the sine wave. Figure 6-8 shows the node graph which implements this process in four stages. Stage 1 uses Graph Mapper (1b) to drive the creation of an interpolated sine curve. Stage 2 is an exact copy from Figure 6-5 and breaks the curve into random length segments. Stage 3 creates random sized cross-sectional curves for the blocks centered at the division points. Stage 4 extrudes the blocks.

Stage 1: Creating a Sine Curve

In stage 1, we use Graph Mapper to generate points at fixed steps in the domain of 0 to 1 based on the Sine graph type. We then remap the points to a domain of our choosing and interpolate them back into a curve.

We start by using Range (1a) with default parameters to create a list of 11 numbers in 0.1 increments (0, 0.1, 0.2, ..., 1.0), to use as input for Graph Mapper (1b), with Graph type set to Sine (right-click ➤ Graph types ➤ Sine). By default, Graph Mapper has domains 0 to 1 for both X and Y, which matches Range's default domain.

Next, we use two Remap Numbers nodes (1c, 1d) to map the X values from Range (1a) and Y values from Graph Mapper (1b) to a custom domain created using Construct Domain. We want the peaks of the sine curve to point toward +Z, so we link the mapped X and Y values to X and Z of Construct Point (1e), followed by Interpolate (1f) to reconstitute the curve from the points.

Note Where to find the nodes in stage 1 of Figure 6-8: Sets ➤ Sequence ➤ Range (1a), Params ➤ Input ➤ Graph Mapper (1b), Maths ➤ Domain ➤ Remap Numbers (1c, 1d), Vector ➤ Point ➤ Construct Point (1e), and Curve ➤ Spline ➤ Interpolate (1f).

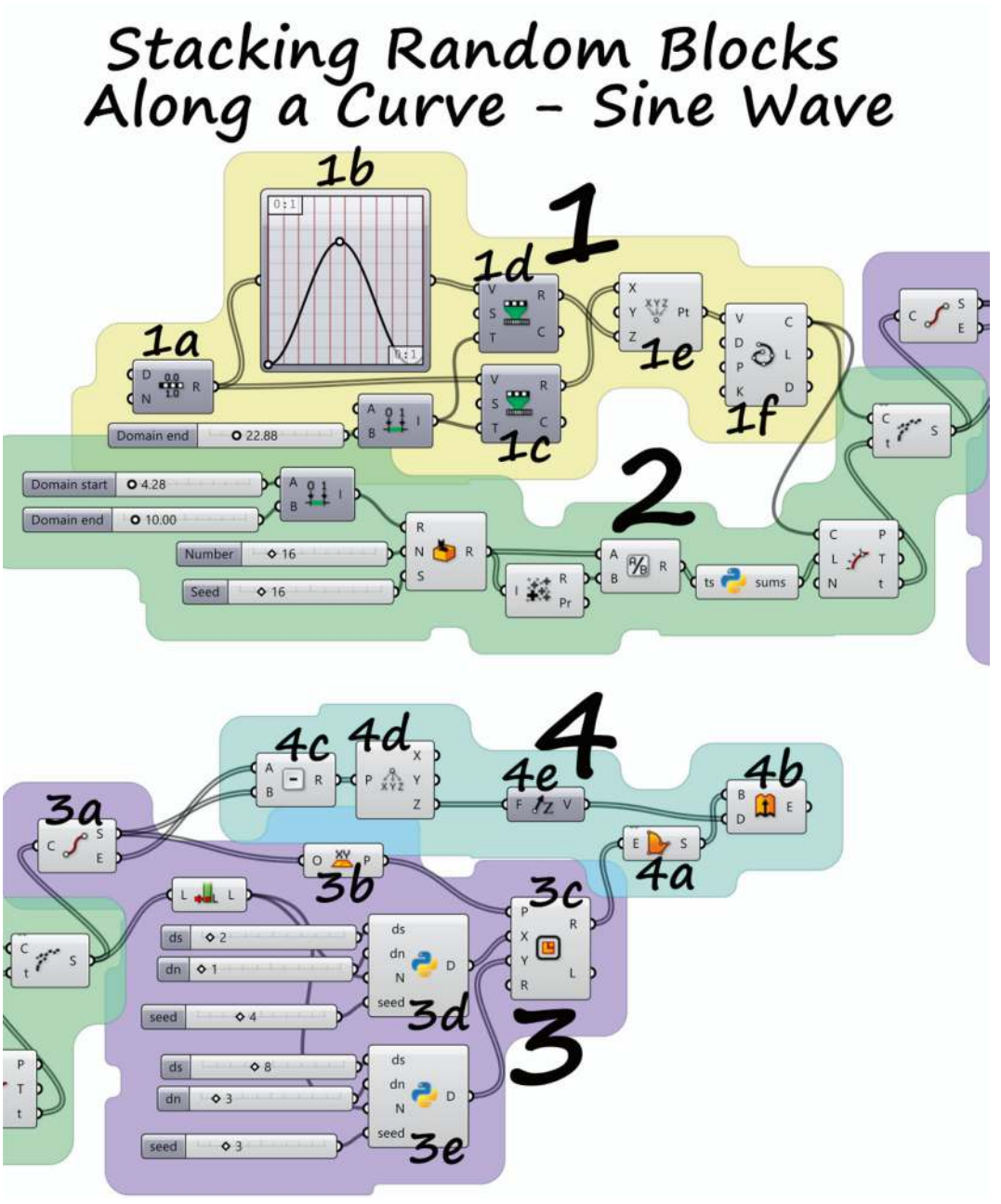


Figure 6-8. *Node graph for stacking randomly sized blocks along an arbitrary curve. Stage 1 creates a curve using Graph Mapper (as shown, Graph type is Sine). Stage 2 randomly divides the curve. Stage 3 creates randomly sized rectangular curves centered at those points. Stage 4 extrudes the rectangles based on the curve's direction relative to where they are and their distances to neighboring boxes*

Stage 2: Deriving Thicknesses for the Blocks by Randomly Dividing the Curve

Stage 2 in Figure 6-8 is identical to stage 2 in Figure 6-5, which breaks the sine curve into random segments. Please refer to the previous section for a detailed walkthrough.

Stage 3: Creating Random 2D Outlines for the Blocks

In stage 3 (Figure 6-8), we'll use two Python 3 Script nodes (3d, 3e) to generate X and Y inputs for Rectangle (3c) to create random 2D outlines for the blocks. The code for the Script nodes (3d, 3e) is shown in Listing 6-3 with its Access and Type Hints summarized in Table 6-3.

The node takes four inputs for the pseudo random number generation—the range the values should fall in (from `ds` to `dn`), number of values (`N`) to generate, and the `seed` for the generator. The code generates `N` numbers between `ds` and `dn` then uses the numbers to create domains centered at 0. For example, if N = 5, ds = 1, and dn = 4, the code *might* generate the numbers

2,4,3,2,1

which in turn becomes the domains –2 to 2, -4 to 4, -3 to 3, -2 to 2, and –1 to 1. These domains are then used as either the X or Y input for Rectangle (3c). Since we want the blocks to be parallel to the XY plane, we use End Points (3a) to isolate the start point of each curve segment, link them to XY Plane (3b) to create planes parallel to XY centered at those points, and use them as the P input for Rectangle (3c).

In Listing 6-3, we import `ghpythonlib.components` as the shorthand `ghcps` so we can call Grasshopper components as Python functions. We then call `ghcps.ConstructDomain` to create a domain from `ds` to `dn`. Next, we call `ghcps.Random` to create `N` numbers between `ds` and `dn` using the given `seed`. Finally, we create a list of domains (`D`) centered at 0 using the random numbers as bounds.

Note Where to find the nodes in stage 3 of Figure 6-8: Curve ➤ Analysis ➤ End Points (3a), Vector ➤ Plane ➤ XY Plane (3b), Curve ➤ Primitive ➤ Rectangle (3c), and Maths ➤ Script ➤ Python 3 Script (3d, 3e).

Listing 6-3. Code inside the Python 3 Script node (3d, 3e) from Figure 6-8 (see /Ch6/Figure6-8_6-9_stacking_along_curve_sine.gh)

```
import ghpythonlib.components as ghcps

ran_range = ghcps.ConstructDomain(ds, dn)
ran_numbers = ghcps.Random(ran_range, N, seed)
D = ghcps.ConstructDomain( ↩
    ghcps.Negative(ran_numbers), ran_numbers)
```

Table 6-3. *Access and Type Hints for the Python 3 Script node (3d, 3e) in Figure 6-8*

Name	Input or Output	Access	Type Hints
ds	input	Item Access	float
dn	input	Item Access	float
N	input	Item Access	int
seed	input	Item Access	int
D	output	N/A	Interval

Stage 4: Extruding the Blocks Along Curve Direction

Stage 4 (Figure 6-8) turns the rectangles (3c) into surfaces (4a) and extrudes them into blocks by following the general direction of the curve, as visualized in Figure 6-9.

We start by linking Rectangle (3c) to Boundary Surfaces (4a) to create the bottoms for the blocks. We then derive the height vectors for the blocks as follows—we subtract (4c) the curve segments' start points from the end points to get the segments' tangents. We then use Deconstruct (4d) to extract only the Z values with which to scale Unit Z vectors (4e), which are in turn used as the Direction (D) input for Extrude (4b).

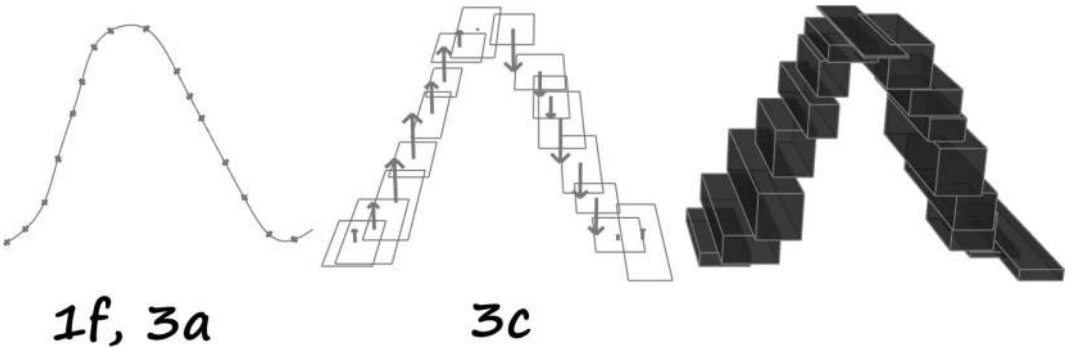


Figure 6-9. *Visual outputs of nodes from Figure 6-8. From left to right: Randomly dividing a sine curve (1f) and extracting the segments' start points (3a), creating randomly sized rectangular curves (3c) centered at those points, and turning the rectangles into boxes of random heights. The arrows show how the rectangles "grow" into boxes*

Stacking Along Different Curve Types: Bezier, Power, and Sine Summation

We can create variations of the node graph in Figure 6-8 by right-clicking the Graph Mapper node (1b) and setting different Graph types. Figure 6-10 shows three examples where the Graph type is set to Bezier, Power, and Sine Summation, respectively. Note that I've further modified the Graph Mapper nodes by dragging the handles of the control points.

A few other changes are made to Figure 6-8 to produce the outputs shown at the bottom of Figure 6-10, including slider values for the Random node in stage 2, which controls the divisions of the curve, and inputs for the Python 3 Script nodes in stage 3 (3d and 3e), which varies the width and length of the blocks

You can find the complete node graphs in the file `/Ch6/Figure6-10_stacking_along_curve_variations.gh` available at the book's Git repo.

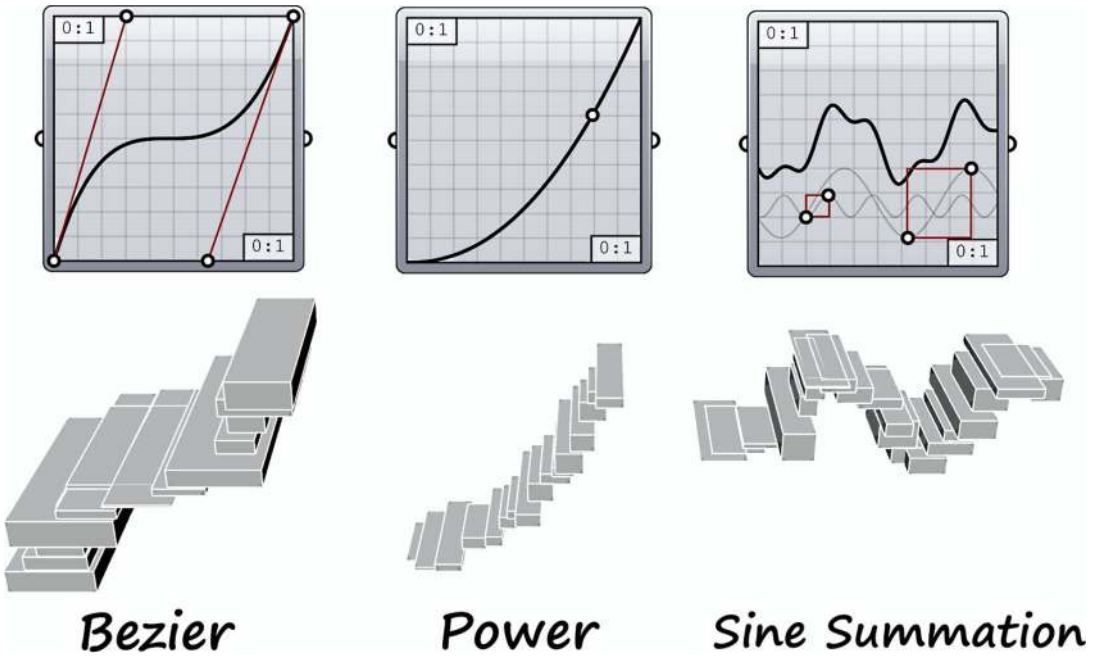


Figure 6-10. *Three variations of the node graph from Figure 6-8, made by changing the Graph type (as shown) along with slider values for the Random node in stage 2, which controls the divisions of the curve, and inputs for the Python 3 Script nodes in stage 3 (3d and 3e), which varies the width and length of the blocks (see* `/Ch6/Figure6-10_stacking_along_curve_variations.gh`*)*

Making Stacks: Stacking Within a Volume

With the node graphs we've built so far, it's relatively simple to extend the concept from 2D to 3D. In this section, we'll generate a point cloud within a 3D region and use that as the basis for stacking blocks in 3D. We'll add poles extending from the blocks to the ground plane to make the block structure resemble a building. The node graph in Figure 6-11 implements this process, which is broken into three stages. In stage 1, we create a box and use Populate 3D (1d) to randomly scatter points within the box. In stage 2, we create boxes of random dimensions with their bottoms centered at those points. In stage 3, we create poles that run from the XY plane to the boxes. Let's look at each stage in detail.

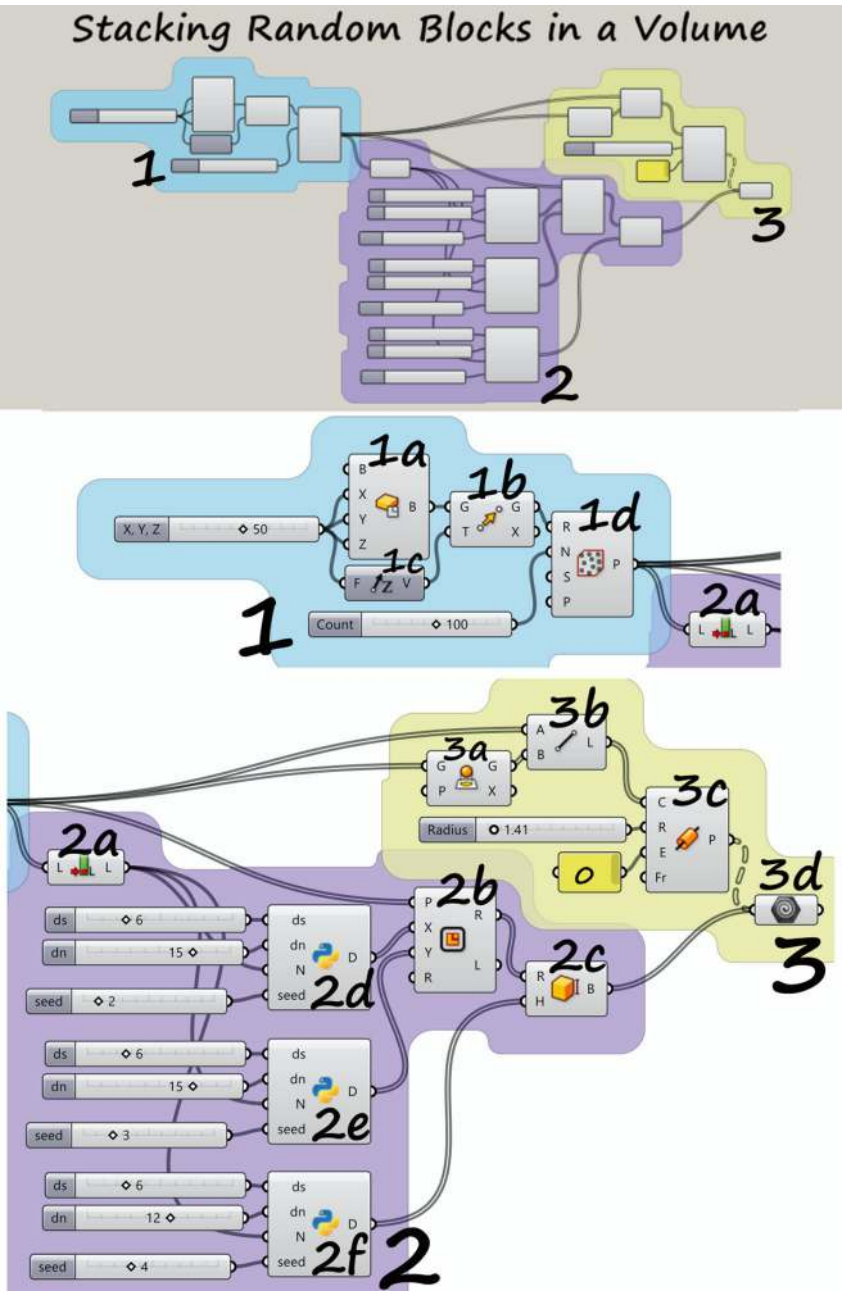


Figure 6-11. *Node graph for stacking randomly sized blocks to fill a volume, with each block supported by a pole. Stage 1 creates a point cloud in a box. Stage 2 creates randomly sized blocks whose bottoms center at those points. Stage 3 creates poles from the blocks to the "ground" (the XY plane)*

Stage 1: Populating Points in a 3D Volume

In stage 1, we create a box and populate a point cloud inside. We use the Center Box (1a) node, which creates a box centered at the origin of its input plane (B) with "radius" along each dimension specified via X, Y, and Z. As shown, we have a single Number Slider dialed to 50 Shift-dragged to X, Y, and Z and B left at default (XY), which creates a 100x100x100 cube centered at (0, 0, 0). We use the Move node (1b) to move the cube by 50 along +Z (1c), so it sits flush on the XY plane. We then use Populate 3D (1d) to generate N = 100 points within the cube, as visualized in Figure 6-12.

Note Where to find the nodes in stage 1 of Figure 6-11: Surface ➤ Primitive ➤ Center Box (1a), Transform ➤ Euclidean ➤ Move (1b), Vector ➤ Vector ➤ Unit Z (1c), and Vector ➤ Grid ➤ Populate 3D (1d).

Stage 2: Creating Blocks with Random Dimensions with Bottoms Centered at Those Points

In stage 2, we create Rectangles (2b) of random dimensions centered at the points from stage 1, turn them into surfaces, and extrude them to random heights to create blocks.

We reuse the Python 3 Script node from the previous section (3d and 3e in Figure 6-8) to generate the X and Y inputs (2d, 2e) for Rectangle (2b). By linking Populate 3D (1d) to the P input of Rectangle (2b), we tell Grasshopper to make rectangular curves parallel to XY that are centered at those points.

We finish by extruding the rectangles into blocks using Box Rectangle (2c), with random height domains (2f) generated by another copy of the Python 3 Script node from Figure 6-8, as visualized in Figure 6-12.

Note Where to find the nodes in stage 2 of Figure 6-11: Sets ➤ List ➤ List Length (2a), Curve ➤ Primitive ➤ Rectangle (2b), Surface ➤ Primitive ➤ Box Rectangle (2c), and Maths ➤ Script ➤ Python 3 Script (2d, 2e, 2f).

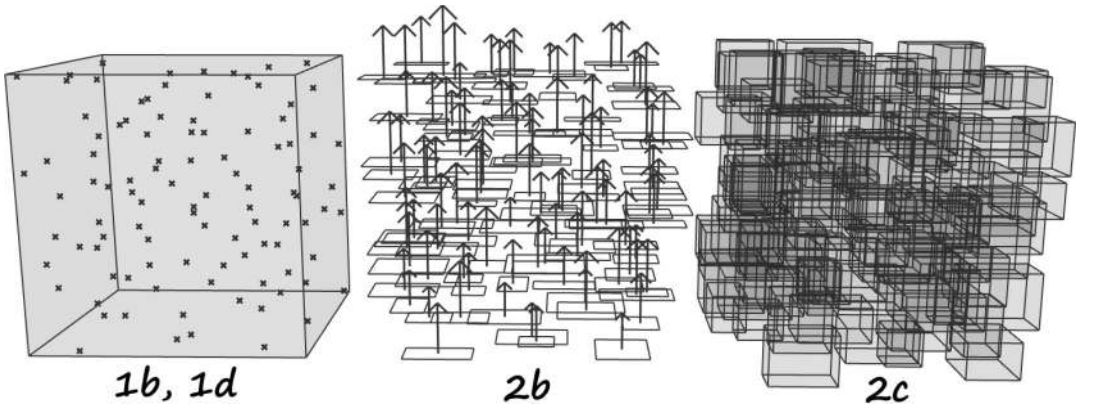


Figure 6-12. *Visual outputs of nodes from Figure 6-11. From left to right: Populating a point cloud (1d) in the box (1b) and creating rectangular curves of random dimensions (2b) centered at those points, which in turn become the bottoms of boxes with random heights (2c). The arrows show how the rectangles grow into boxes*

Stage 3: Creating Poles from the Blocks to the Ground

In stage 3, we wrap up by creating poles that extend from the blocks to the "ground" (XY plane) to make the model resemble a building.

Since we positioned the blocks such that their bottom centers coincide with the point cloud from stage 1, to create poles that extend from the blocks to the ground, all we have to do is Project (3a) the points to the XY plane (using the default value of P) to find the lines' footings. After that, we use Line (3b) to link the points and their projections and Pipe (3c) the lines to create poles. Note that we use E = 0 for Pipe to create flat end caps.

We finish by combining the blocks (2c) and poles (3c) by Shift-dragging them to a single Geometry node (3d). The process of projecting the points, creating the poles, and combining the blocks and poles is visualized in Figure 6-13.

Note Where to find the nodes in stage 3 of Figure 6-11: Transform ➤ Affine ➤ Project (3a), Curve ➤ Primitive ➤ Line (3b), Surface ➤ Freeform ➤ Pipe (3c), and Params ➤Geometry ➤Geometry (3d).

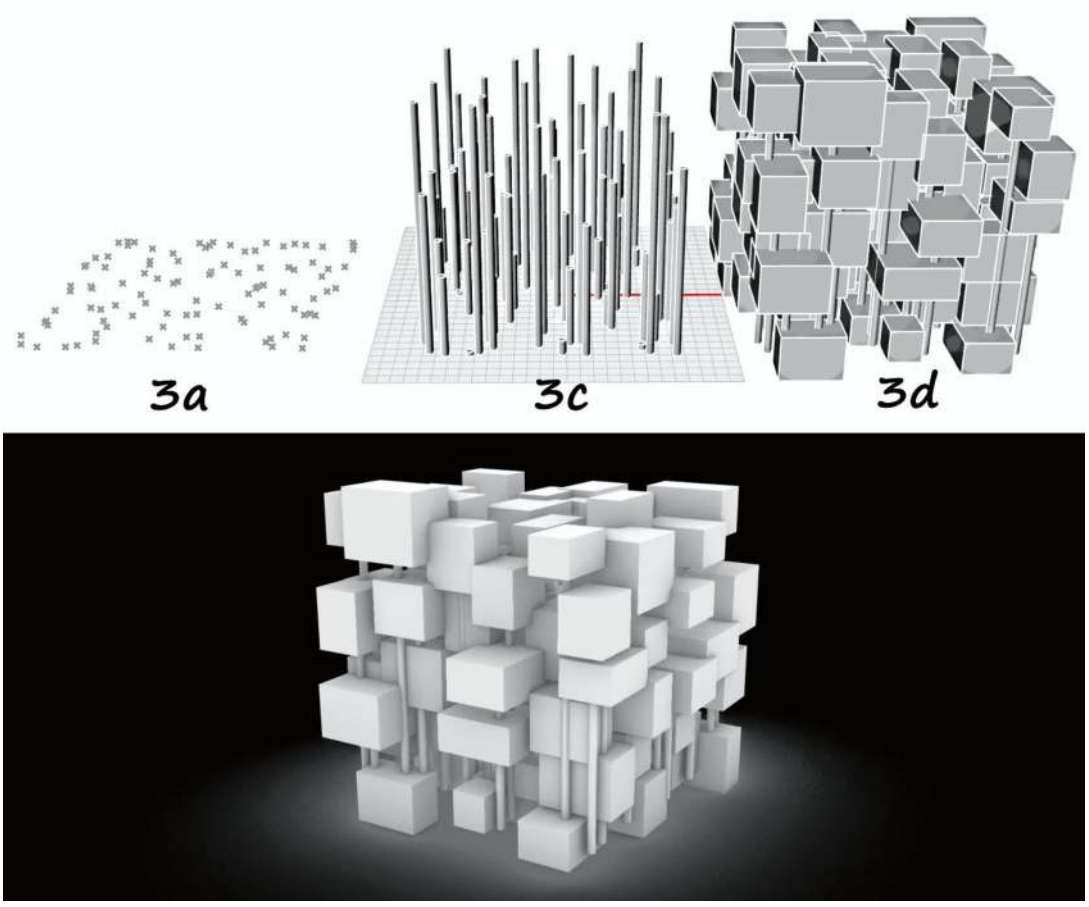


Figure 6-13. *Visual outputs of nodes from Figure 6-11. Top: Projecting the point cloud to the XY plane (3a), linking the point cloud to its projections (3b) and piping (3c), and combining the blocks with the pipes (3d). The bottom shows the completed model*

Voxelizing: Divided Surface (Hollow)

When you decrease the resolution of a 2D image, it starts to pixelate, creating a retro, "8-bit" games kind of look that is reminiscent of the 1980s. We can generalize this concept and *voxelize* a 3D model to make it look "coarse," as if made of blocks.

The node graph in 6-14 shows one possible way of voxelizing a model, by evenly dividing a surface into a grid of points then extruding boxes with bottoms centered at those points. The process is broken into two stages, which we'll look at next.

Stage 1: Evenly Dividing a Surface into a Grid of Points

In stage 1, we create a sphere and evenly divide its surface into a grid of UV points. We create an XY Plane (1b) centered at (0, 0, 50) using Construct Point (1a) with Z = 50 and, in turn, use the plane as the Base (B) input for Sphere (1c) with Radius (R) also set to 50. The result is a sphere with a radius = 50 that sits flush with the XY plane. We then use Divide Surface (1d) with U = 15 and V = 13 to create a grid of (15+1)*(13+1) = 224 points on the sphere—you can think of U and V as the number of horizontal (X) and vertical (Y) division lines, or the number of columns and rows, respectively.

Note Where to find the nodes in stage 1 of Figure 6-14: Vector ➤ Point ➤ Construct Point (1a), Vector ➤ Plane ➤ XY Plane (1b), Surface ➤ Primitive ➤ Sphere (1c), and Surface ➤ Util ➤ Divide Surface (1d).

Stage 2: Creating Blocks of Random Dimensions with Bottoms Centered at Those Points

We use an exact copy of stage 2 from Figure 6-11 here to create blocks of random dimensions, with their bottoms centered at the sphere's surface division points. This creates the voxelized sphere shown in Figure 6-15.

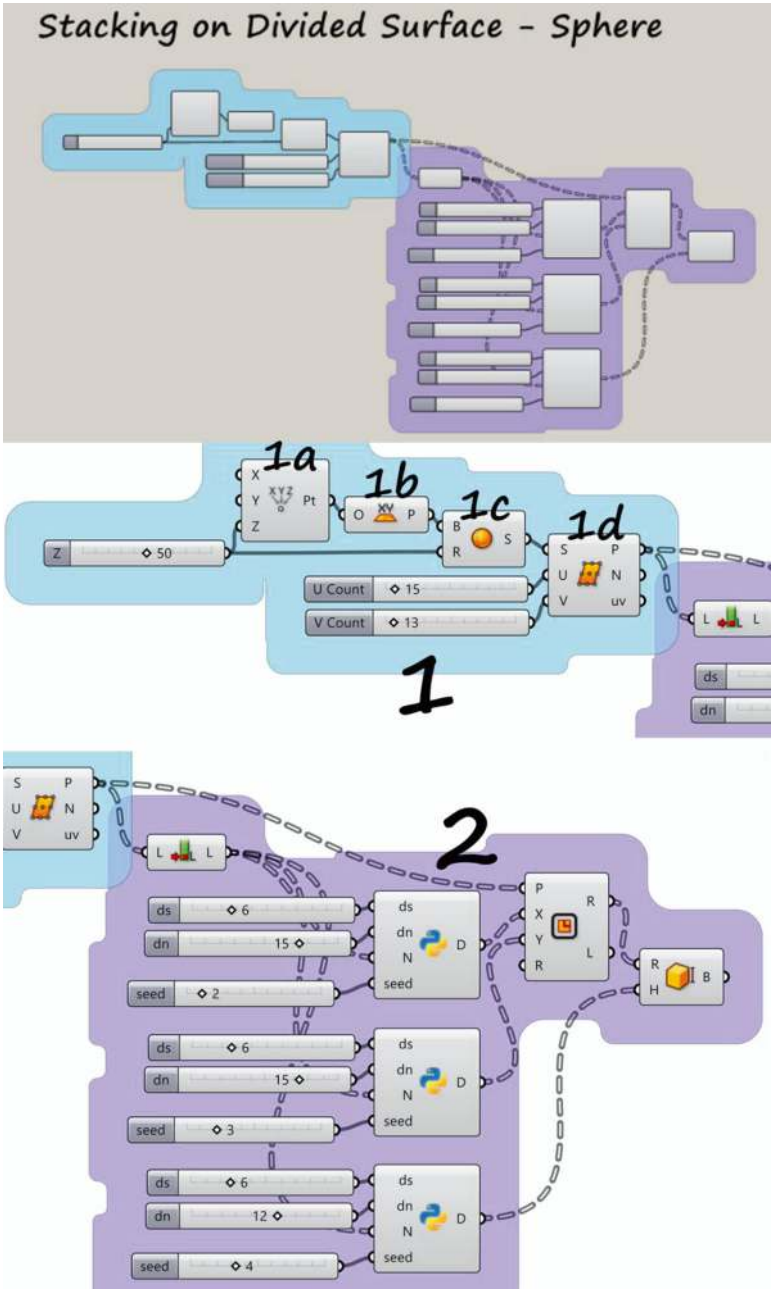


Figure 6-14. *Node graph for stacking blocks of random dimensions on a divided surface. Stage 1 creates a UV grid of points along the surface, and stage 2 extrudes boxes of random dimensions from those points*

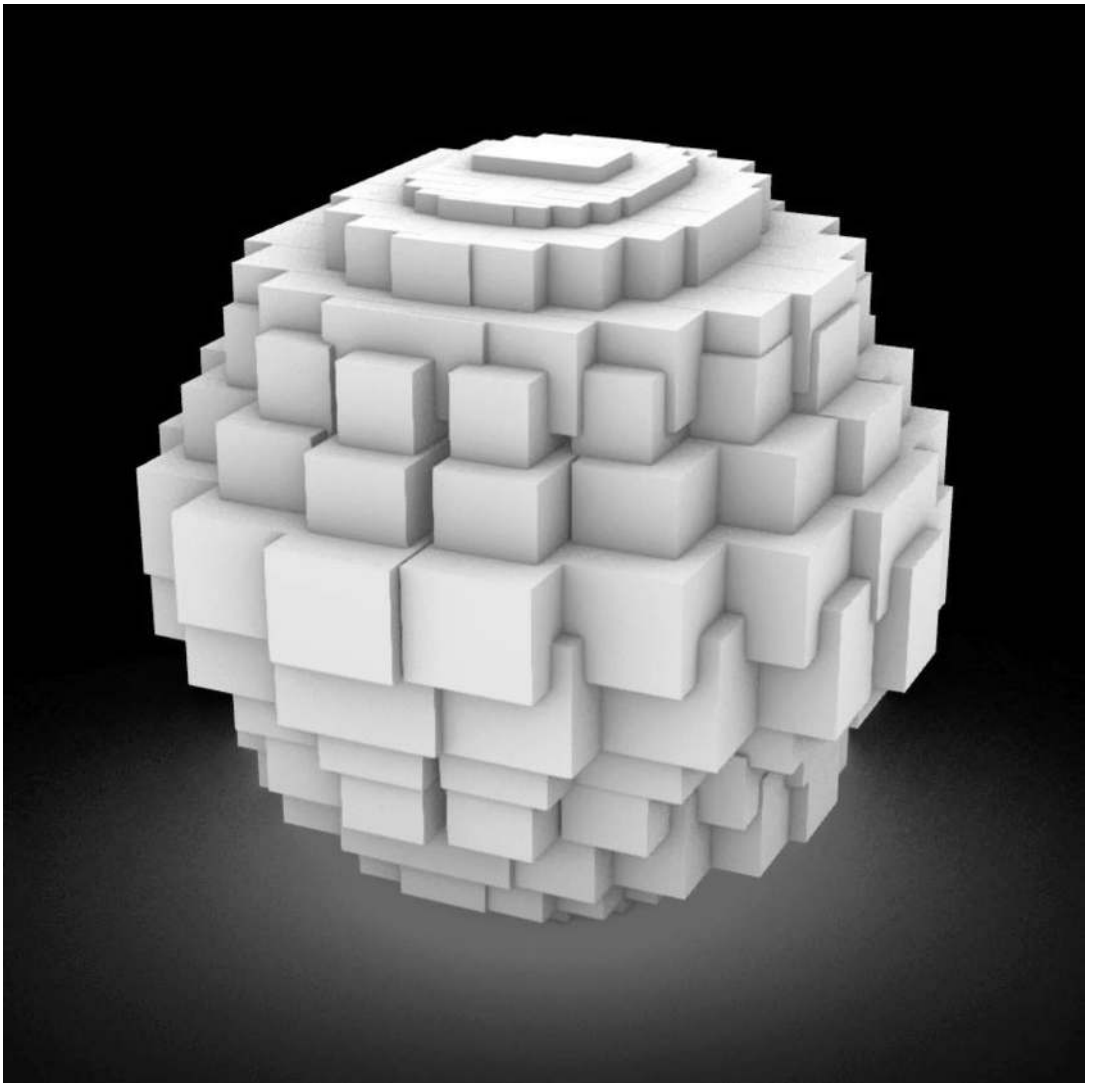


Figure 6-15. *Voxelized sphere, created by dividing the spherical surface into a grid of points, and spawning cubes with those points as their bottom centers*

Variation: Voxelizing a Cylindrical Surface

It is hard to tell with a sphere that the voxelization is only done to the surface, leaving the inside hollow. We can swap out the Sphere node in Figure 6-14 with Cylinder (1a in Figure 6-16) to visualize this effect better. Using Number Sliders to control the Radius (R) and Length (L) inputs for Cylinder (1a) makes the model parametric. Note that Length (L) controls the cylinder's height.

Note Where to find the node in stage 1 of Figure 6-14: Surface ➤ Primitive ➤ Cylinder (1a).

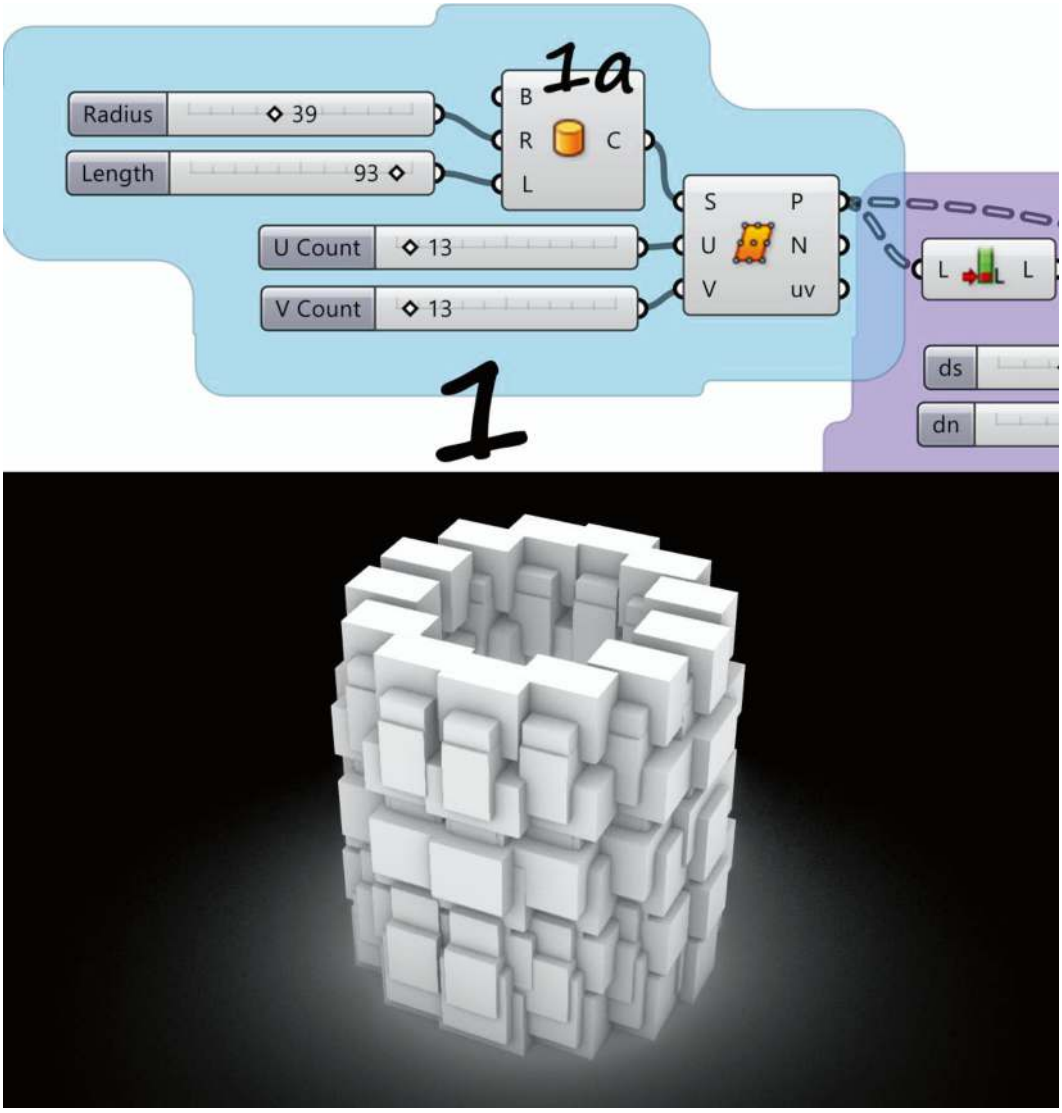


Figure 6-16. *We can voxelize a cylinder instead of a sphere by replacing node 1a in Figure 6-14 with Cylinder (1a) and add a Number Slider to dial its Length (L), which varies the height of the cylinder. The voxelized cylinder using the given parameters is shown on the bottom*

Voxelizing: Mesh or Solid

In the last section, we voxelized surfaces by dividing them into UV grids to create hollow blocky models. In this section, we'll voxelize a mesh or surface by treating it like a solid, with blocks filling its volume.

The node graph in Figure 6-17 implements such a solid "voxelizer" in a four-stage process:

1. Stage 1 simplifies the mesh by turning its faces into quads and reducing its poly count, before converting the mesh into a SubD object.
2. Stage 2 inflates the mesh's bounding box by 10%.
3. Stage 3 populates a point cloud inside the enlarged bounding box as the basis for creating a 3D Voronoi.
4. Stage 4 trims the 3D Voronoi with the mesh by performing a Solid Intersection (4a) and creates two flavors of voxelization:
    a. One using the bounding boxes of the trimmed Voronoi cells
    b. The other using cubes created at the same locations as the cell bounding boxes

Attribution to the Stanford Bunny

The Grasshopper definition `/Ch6/Figure6-17_to_6-19_voronoi_voxelizer.gh` accompanying this section is internalized with the Standford bunny mesh, which is a well-known computer graphics test model developed by Greg Turk and Marc Levoy in 1994 at Stanford University (Wikipedia, 2025). The version included with the GH file has been modified by the author to have a closed bottom.

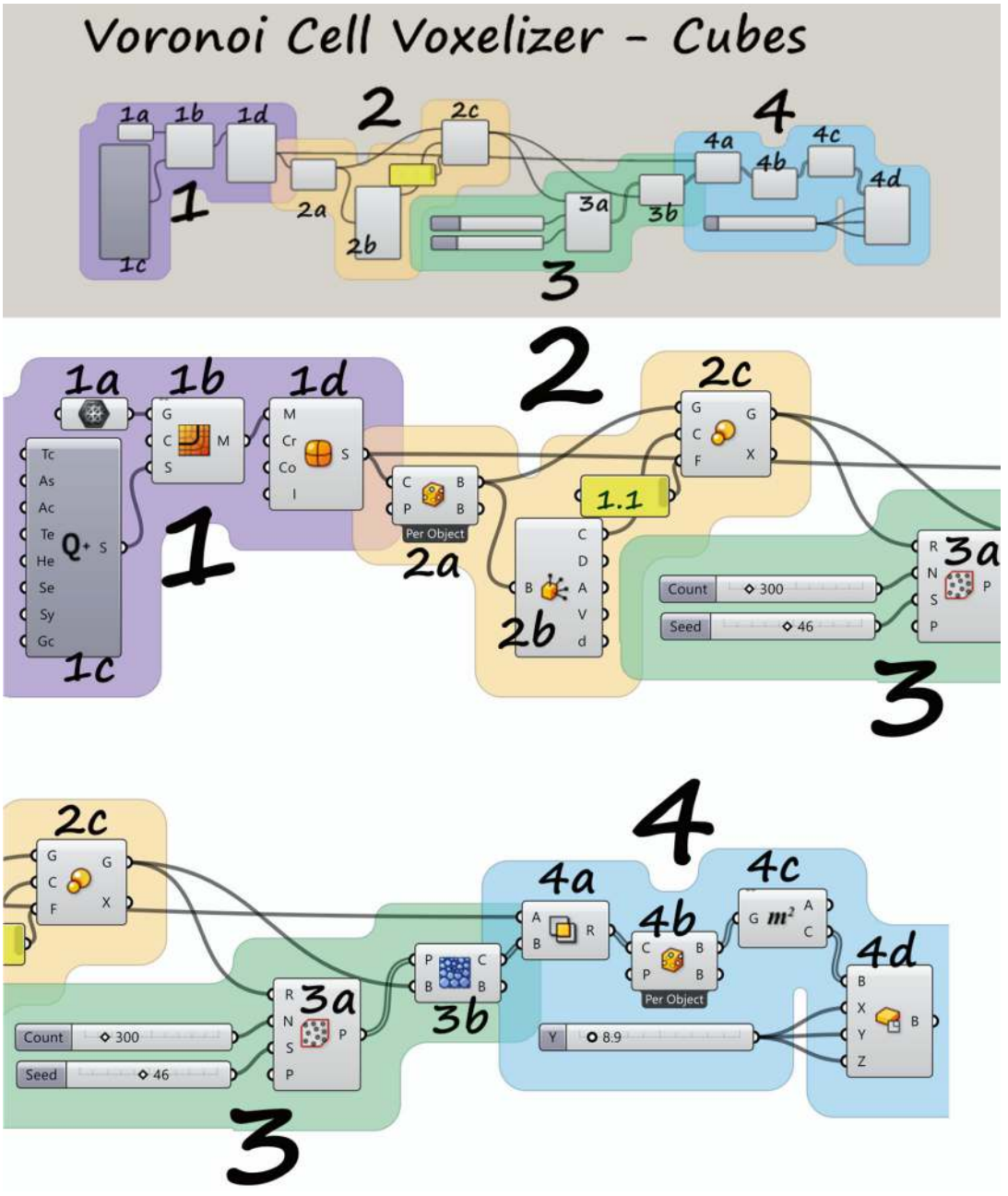


Figure 6-17. *Node graph for voxelizing a mesh two ways, in four stages. Stage 1 reduces the poly count and converts the mesh to a SubD. Stage 2 derives a slightly larger version of the mesh's bounding box, to be used by stage 3 for creating a 3D Voronoi. In Stage 4, either the Voronoi cell's bounding boxes are used as voxels or cubes spawned using the cell's centers*

Stage 1: Reducing Poly Count and Converting to SubD

In stage 1, we'll simplify the mesh and convert it to a SubD object, so we can use nodes that only accept Breps as inputs in subsequent stages of the graph.

We start by linking the Mesh node (1a) internalized with Standford bunny to Quad Remesh (1b) to turn its faces from triangles into quads and reduce its polygon count. The parameters for Quad Remesh (1b) are specified via a Quad Remesh Settings node (1c) with Target Count (Tc) = 1000 left at default, to simplify the bunny mesh to about 1000 faces. After simplifying, we use SubD from Mesh (1d) to convert the mesh into a SubD object.

Note Where to find the nodes in stage 1 of Figure 6-17: Params ➤ Geometry ➤ Mesh (1a), Mesh ➤ Triangulation ➤ Quad Remesh (1b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (1c), and Surface ➤ SubD ➤ SubD from Mesh (1d).

Stage 2: Deriving an Enlarged Bounding Box from the SubD

In stage 2, we inflate the bounding box of the SubD object in preparation for creating the point cloud. We want the bounding box to be a little larger than the bunny to ensure that the Solid Intersection in stage 4 will cut cleanly and be left with the entire bunny.

We link Sub D from Mesh (1d) to Bounding Box (2a, Per Object), followed by a Scale node (2c) to scale it to 110% uniformly. We use a Box Properties node (2b) to find the bounding box's center point to use as the reference point for Scale (2c), specified via its Center (C) input.

Note Where to find the nodes in stage 2 of Figure 6-17: Surface ➤ Primitive ➤ Bounding Box (2a, Per Object (default)), Surface ➤ Analysis ➤ Box Properties (2b), and Transform ➤ Affine ➤ Scale (2c).

Stage 3: Creating a 3D Voronoi Bordered by the Bounding Box

In stage 3, we use Populate 3D (3a) to create 300 random points inside the inflated bunny bounding box (2c) from stage 2. We then use the points to create a Voronoi 3D (3b) with the inflated bounding box (2c) as the diagram's boundary (B).

As defined by Wolfram MathWorld, a *Voronoi diagram* is produced by partitioning a plane into convex polygons based on a set of "generating points," such that each polygon contains only one generating point, and every point enclosed by a polygon is closest to its generating point than to any other (Weisstein, 2026). The Voronoi 3D node (3b) generalizes this concept from 2D to 3D and partitions a 3D volume into convex polyhedrons.

Note Where to find the nodes in stage 3 of Figure 6-17: Vector ➤ Grid ➤ Populate 3D (3a) and Mesh ➤ Triangulation ➤ Voronoi 3D (3b).

Stage 4: Creating Voxelized Model from the Voronoi Cells' Bounding Boxes

In stage 4, we trim the Voronoi 3D diagram (3b) with the bunny SubD (1d) using Solid Intersection (4a). We then create two types of voxelization—the first using the Bounding Boxes (4b, Per Object) of the trimmed Voronoi 3D cells. Since the cells are differently sized, so are the bounding boxes. Therefore, this method creates an abstract looking voxelization, as shown in Figure 6-18.

We can create a second voxelization by using uniformly sized voxels. To do so, we find the center points of the Voronoi 3D cells using the Area node (4c). We then create cubes centered at those points using Center Box (4d) with Base (B) set to the cell centers and the X, Y, and Z inputs controlled by a single slider.

The two types of voxelizations (nonuniform and uniform) are visualized in Figure 6-18.

Note Where to find the nodes in stage 4 of Figure 6-17: Intersect ➤ Shape ➤ Solid Intersection (4a), Surface ➤ Primitive ➤ Bounding Box (4b, Per Object (default)), Surface ➤ Analysis ➤ Area (4c), and Surface ➤ Primitive ➤ Center Box (4d).

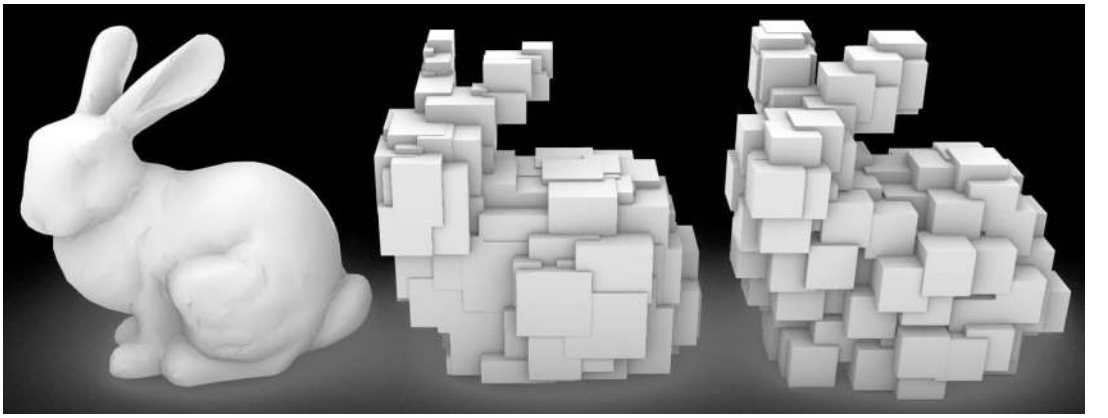


Figure 6-18. *Left: The original Stanford bunny mesh. Center: Voxelized version using bounding boxes of Voronoi cells. Right: Voxelized version using cubes centered at Voronoi cells*

Variation: Using Spheres As Voxels

We can change the shape of the voxels simply by swapping out the Center Box (4d) node in Figure 6-17. Figure 6-19 shows an example where we replace Center Box (4d) with a Sphere node, creating a "bubbles" look.

Note Where to find the nodes in the modified stage 4 in Figure 6-19: Intersect ➤ Shape ➤ Solid Intersection (4a), Surface ➤ Primitive ➤ Bounding Box (4b, Per Object (default)), Surface ➤ Analysis ➤ Area (4c), and Surface ➤ Primitive ➤ Sphere (4d).

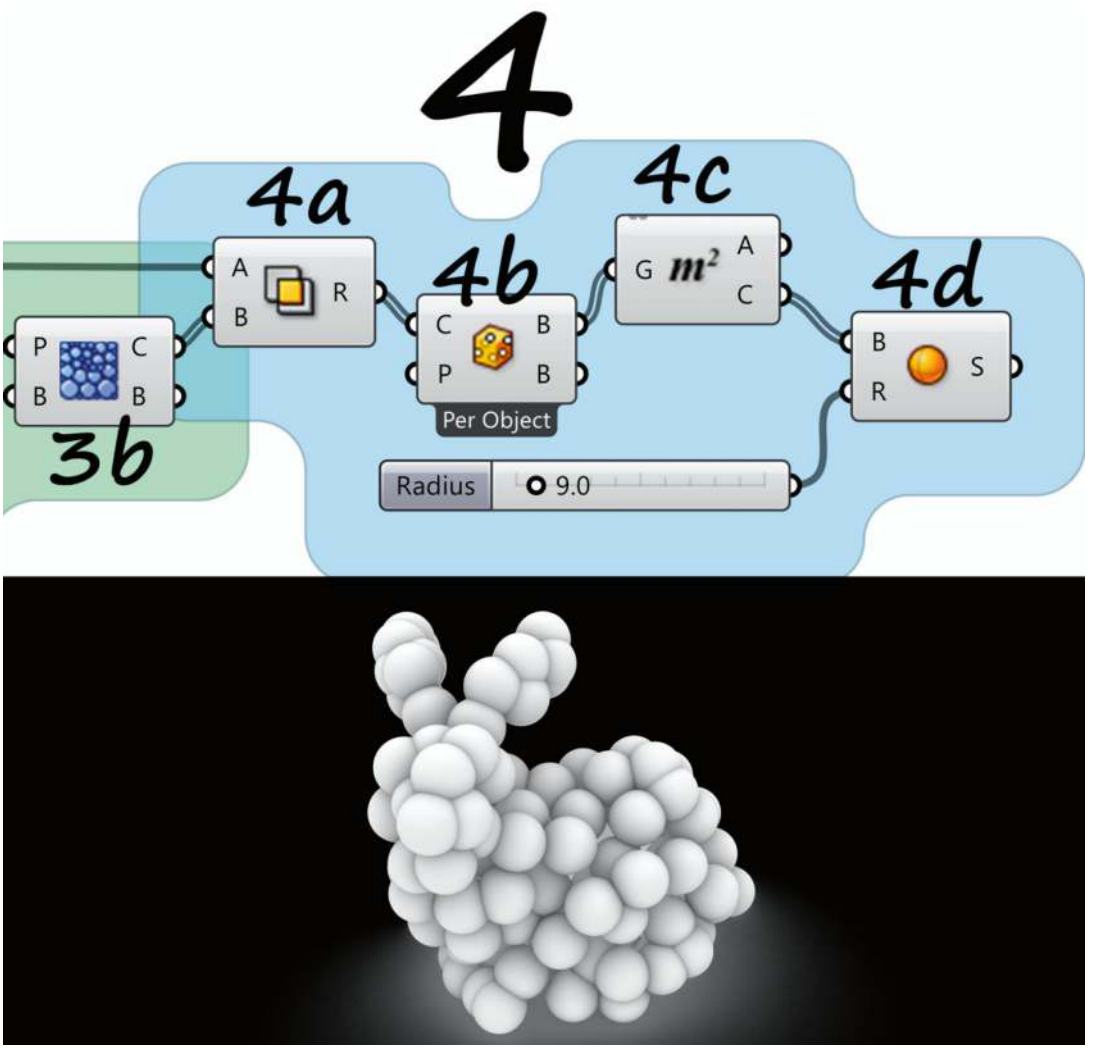


Figure 6-19. *We can create a "bubbles" variation of the mesh voxelizer by replacing 4d of the node graph in Figure 6-17 with a Sphere node and a Number Slider to control its Radius (R). The result is shown at the bottom*

Summary

In this chapter, you worked through four stacking and two voxelization workshops. The first stacking workshop showed you how to stack blocks into straight columns and create variations by steadily scaling or rotating blocks in fixed increments. The second workshop increased complexity by randomizing the thicknesses of the blocks, scaling blocks steplessly by using the Taper node, and offsetting blocks horizontally by

picking one of the blocks' bounding box corners at random to shift toward. The third workshop expanded on the concept of stacking along a trajectory—instead of stacking along a straight line, we modified the node graph to stack along curves like sine waves and power curves. The final workshop takes stacking from 2D to 3D, by creating blocks relative to randomly populated points inside a volume. The stack is made to resemble a building with the addition of poles extending from the blocks to their projections on the XY plane.

The second half of the chapter showed you two ways to implement *voxelizers*, node graphs that generate "pixelated" or blocky versions of a given surface or mesh. The first example voxelized models by positioning blocks at evenly divided points along their surfaces, leaving the inside of the models hollow. The second example treated the input mesh as a solid and filled its volume with blocks. The process entailed creating a 3D Voronoi diagram based on points scattered inside the mesh's bounding box, followed by trimming the Voronoi with the mesh using Solid Intersection. A voxelization can then be created either from the bounding boxes of the 3D Voronoi cells (for an abstract look) or cubes centered at the cells' center points (a uniform look).

References

Wikipedia. "Stanford bunny." *Wikipedia*, 29, August 2025, CC BY-SA 4.0, https://en.wikipedia.org/wiki/Stanford_bunny.

Weisstein, Eric W. "Voronoi Diagram." *From MathWorld—A Wolfram Resource,* 11, Feb 2026, https://mathworld.wolfram.com/VoronoiDiagram.html

CHAPTER 7

Cracks and Wireframes

In this chapter, we'll work through seven workshops—the first five playing on the theme of wireframes and the last two cracking and exploding Breps and meshes. The wireframes you'll create include a star dome, a beehive dome, and a floral tower, based on manipulating meshes and surfaces. The cracked and exploded models you'll create include cubes, spheres, and the Stanford bunny, using approaches that are generalizable to any closed Breps and meshes.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch7` folder, where you'll find `*.gh` files named after the figures in which they appear, along with a short description. For example, `/Ch7/Figure7-1_7-2_wireframe_LunchBox_star_dome.gh`. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, build your own node graphs by following the steps described in the text or start with the downloaded `*.gh` files and modify the parameters to experiment.

Installing the LunchBox Plugin

Some examples in this chapter make use of the free GH plugin LunchBox. To install LunchBox, go to `https://www.food4rhino.com/en/app/lunchbox`, locate the latest version for your Rhino version and operating system, click "Install", and follow the prompts. Once installed, you'll see an additional component tab named "LunchBox" appear in the Grasshopper editor.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_7

Attribution to the Stanford Bunny

The Grasshopper files listed below are internalized with the Stanford bunny mesh, a computer graphics test model developed by Greg Turk and Marc Levoy in 1994 at Stanford University (Wikipedia, 2025). The version included is modified by the author to have a closed bottom.

- /Ch7/Figure7-7_quad_wireframe_from_mesh.gh
- /Ch7/Figure7-8_tri_wireframe_from_mesh.gh
- /Ch7/Figure7-9_hex_wireframe_from_mesh.gh
- /Ch7/Figure7-19_to_7-21_cracks_voronoi_input_mesh.gh

Making Wireframes: LunchBox Star Dome

In the first workshop, we'll create a wireframe dome using the Skewed Quads node under LunchBox ➤ Panels. Skewed Quads tiles a given surface with panels shaped like parallelograms. The tiles degenerate into triangles at the north and south poles of a sphere due to its tapering shape, creating the star patterns shown in Figure 7-2.

Figure 7-1 shows the node graph that generates the star dome in three stages. Stage 1 creates a half cube with its top centered at (0, 0, 0). Stage 2 uses Skewed Quads to panel all over a sphere also centered at (0, 0, 0), with a diameter equal to the half cube's side length. Stage 3 performs a Solid Difference between the paneling and the half cube, hence slashing the bottom half of the paneling away and turning it into a dome shape. Finally, the edges of the dome are piped into the structure shown in Figure 7-2.

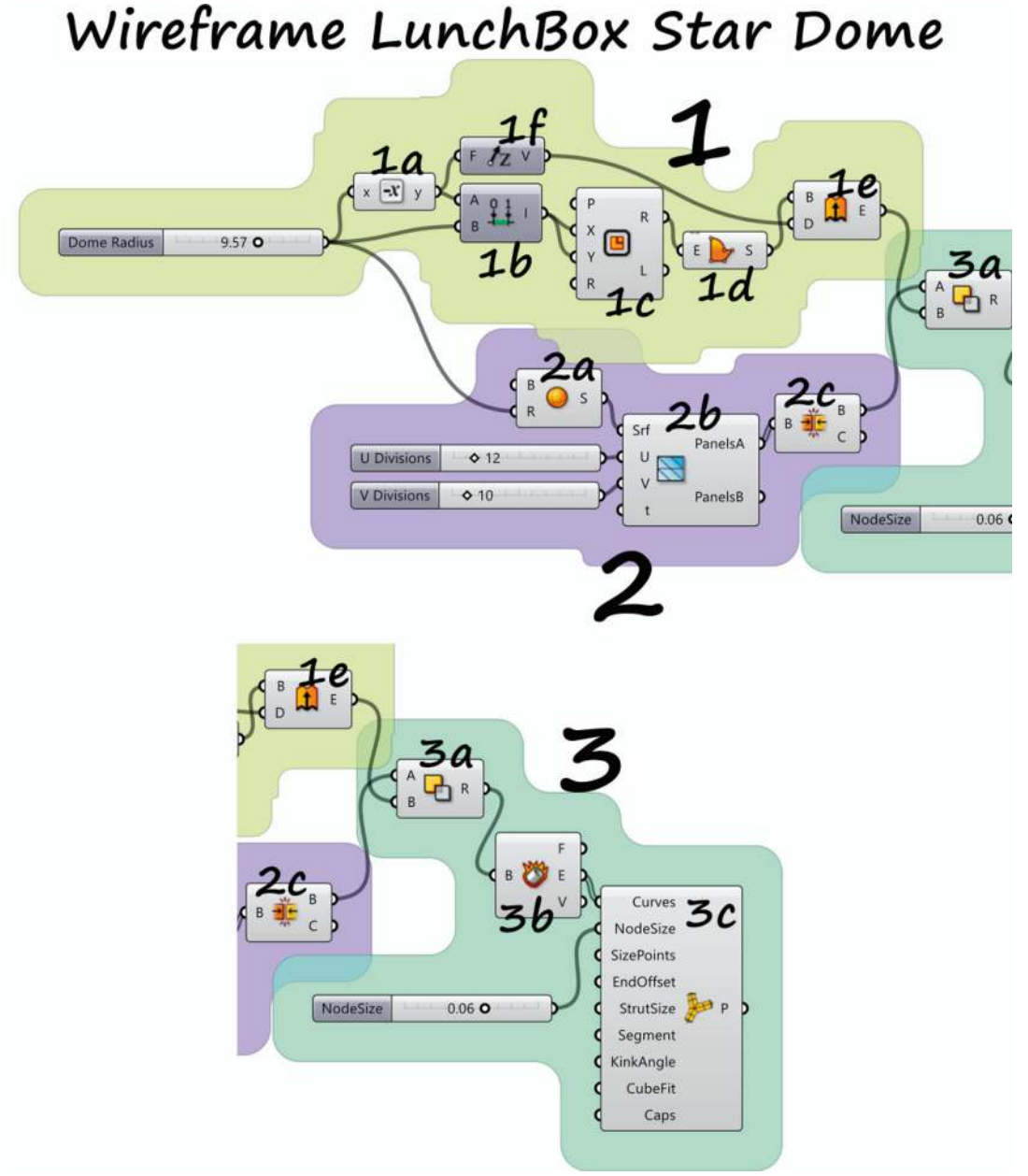


Figure 7-1. *Node graph that creates a wireframe dome with a star pattern on top using LunchBox's Skewed Quads. Stage 1 creates a half cube enclosing the bottom half of the sphere used in Stage 2 for paneling with Skewed Quads. Stage 3 uses Solid Difference to cut the bottom half of the paneling away and extracts and pipes the remaining portion into a wireframe dome*

Stage 1: Creating a Half Cube to Enclose the Bottom Half of the Sphere

Our goal in stage 1 (Figure 7-1) is to create a half cube that is just big enough to enclose the bottom half of the sphere we'll use to panel in stage 2, so that we can perform a Solid Difference on the two later to cut away the sphere's bottom half.

We start by creating a domain centered at (0, 0) with Construct Domain (1b) by using a Number Slider to set its larger end (B) and the same slider passed through Negative (1a) to set the smaller end (A). For example, if the slider is set to 9.57, this creates the domain -9.57 to 9.57.

We then use the domain as the X and Y inputs for Rectangle (1c) with P left at default (XY Plane) to create a square centered at (0, 0, 0). We turn the square into a surface using Boundary Surfaces (1d) and Extrude (1e) toward –Z to form the half cube, using the negated half domain to scale Unit Z (1f) as the Direction (D). Figure 7-2 shows the front view of the half cube (1e).

Note Where to find the nodes in stage 1 of Figure 7-1: Maths ➤ Operators ➤ Negative (1a), Maths ➤ Domain ➤ Construct Domain (1b), Curve ➤ Primitive ➤ Rectangle (1c), Surface ➤ Freeform ➤ Boundary Surfaces (1d), Surface ➤ Freeform ➤ Extrude (1e), and Vector ➤ Vector ➤ Unit Z (1f).

Stage 2: Paneling the Sphere with Skewed Quads

In stage 2 (Figure 7-1), we use Skew Quads (2b) to panel a sphere then join the panels into a single Brep.

We deploy a Sphere node (2a) and use the same slider from stage 1 to control both the half cube's domain and the sphere's radius, so the half cube will enclose the bottom half of the sphere. We then use Skewed Quads (2b) to panel the sphere, with two sliders controlling the number of U and V divisions. Recall that U and V represent the number of horizontal and vertical division lines or the number of columns and rows, respectively. Therefore, the larger the U (or V) the smaller the panels. The results of the paneling are visualized in Figure 7-2.

Skewed Quads (2b) outputs the panels as individual surfaces. We want to use Solid Difference (3a) in the next stage to trim off the bottom half of the sphere paneling. However, Solid Difference (3a) will only work correctly if the paneling is given as a single Brep. Therefore, we link Skewed Quads (2b) to Brep Join (2c) to join all the panels into one closed Brep.

Note Where to find the nodes in stage 2 of Figure 7-1: Surface ➤ Primitive ➤ Sphere (2a), LunchBox ➤ Panels ➤ Skewed Quads (2b), and Surface ➤ Util ➤ Brep Join (2c).

Stage 3: Creating the Wireframe Dome from the Top Half of the Paneling

In Stage 3 (Figure 7-1), we use Solid Difference (3a) between the joined paneling from stage 2 (2c) and the half cube from stage 1 (1e) to trim off the bottom half of the paneling. Note that the order of the two inputs for Solid Difference (3a) matters—the object to be trimmed goes first, and the shape to cut out goes second.

We're now left with the top half of the paneling in a hemisphere shape. We isolate its edges with Deconstruct Brep (3b), followed by MultiPipe (3c) to give the edges volume to create the star dome shown in Figure 7-2.

Note Where to find the nodes in stage 3 of Figure 7-1: Intersect ➤ Shape ➤ Solid Difference (3a), Surface ➤ Analysis ➤ Deconstruct Brep (3b), and Surface ➤ SubD ➤ MultiPipe (3c).

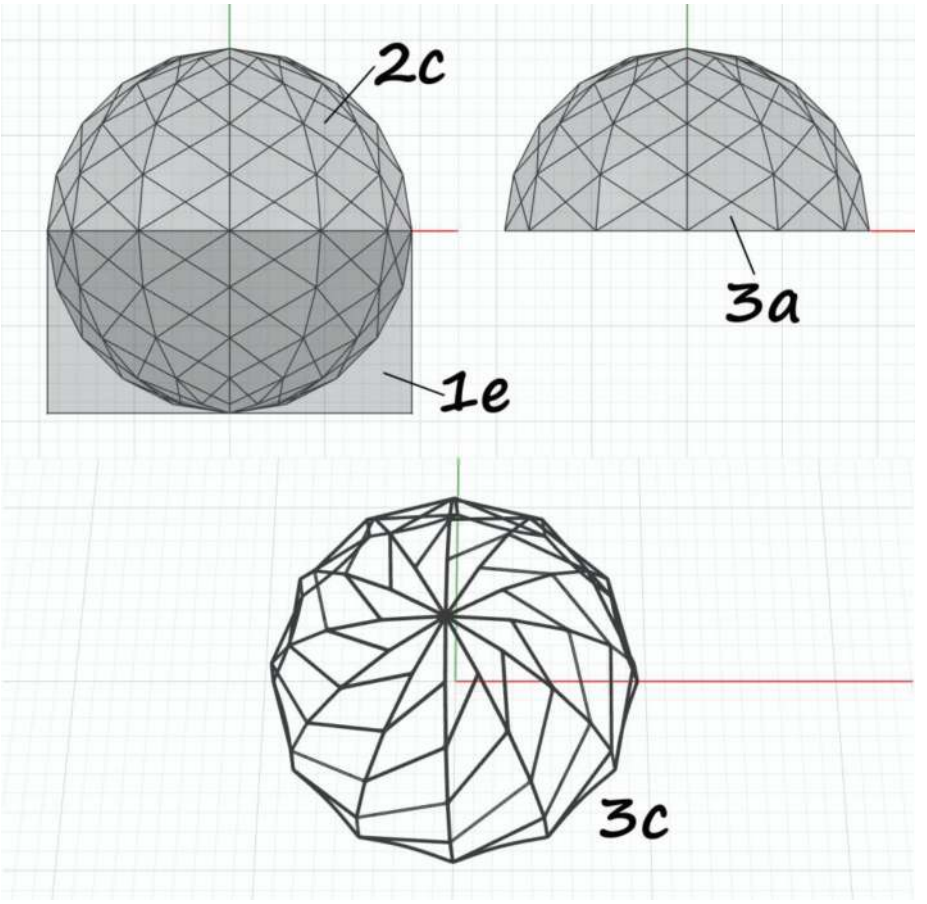


Figure 7-2. *Visualizing intermediate steps of the node graph from Figure 7-1. We use Solid Difference between the joined sphere paneling (2c) and the half cube (1e) to trim off the bottom half of the paneling, leaving the top (3a). We then isolate the top's edges followed by MultiPipe (3c) to create the star dome*

Making Wireframes: Beehive Dome

For the second workshop, we'll make a wireframe dome with hexagonal cells reminiscent of a beehive, using the node graph shown in Figure 7-3. We reuse the same stage 1 from Figure 7-1, where we create a half cube just large enough to enclose the bottom half of the sphere that we create in stage 2, this time with no paneling. We again use Solid Difference (2b) to cut off the bottom half of the sphere. In stage 3, we create a hexagonal grid, project it onto the hemisphere, and use MultiPipe (3g) to create the dome, as visualized in Figure 7-4.

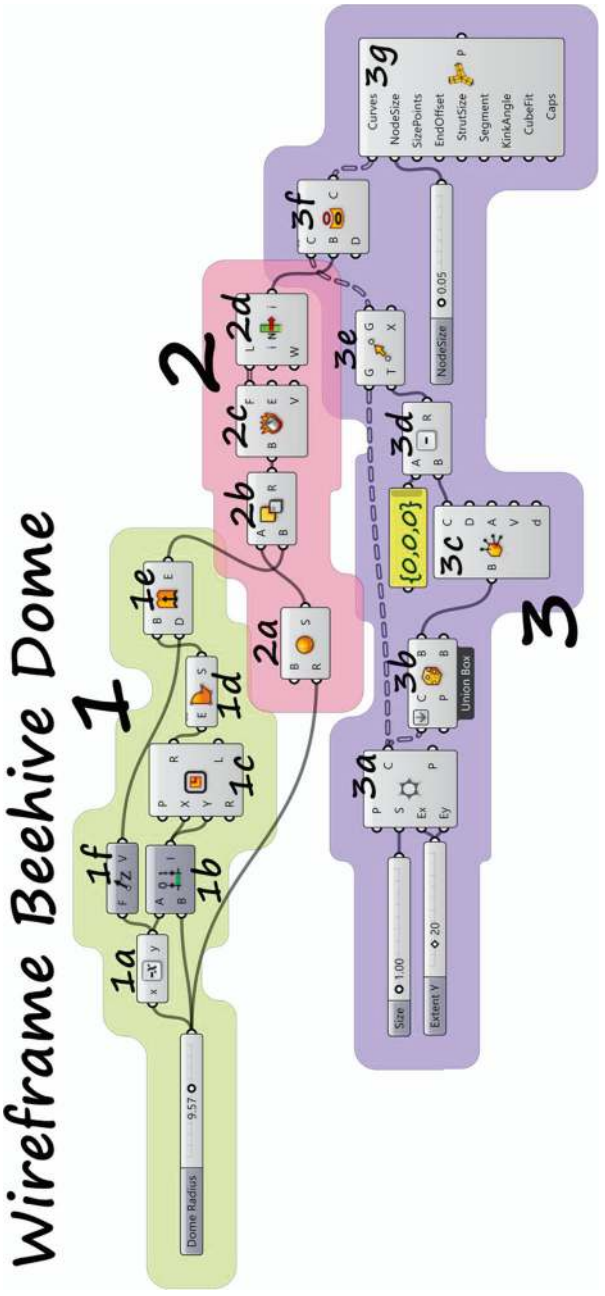


Figure 7-3. *Node graph that creates a beehive dome in three stages. Stage 1 creates a half cube the same way as Figure 7-1 to be used as a trimmer object in stage 2 to cut off the bottom half of a sphere. Stage 3 projects a hexagonal grid on the hemisphere followed by MultiPipe to produce the final dome*

Stage 1: Creating a Half Cube to Enclose the Bottom Half of the Sphere

We reuse stage 1 from the node graph in Figure 7-1 to produce a half cube that will enclose the bottom half of the sphere in stage 2. Please refer to stage 1 in the section titled "Making Wireframes: LunchBox Star Dome" for a detailed explanation.

Stage 2: Creating a Hemisphere with an Open Bottom by Trimming a Sphere with the Half Cube

In stage 2 (Figure 7-3), we perform a Solid Difference (2b) between a sphere and the half cube from stage 1 to create a hemisphere.

We start by creating a plain Sphere (2a) with its B input left at default (XY) to center it at (0, 0, 0). The same slider from stage 1 is used to control both the half cube's domain and the Radius (R) input of the Sphere (2a) so that the half cube will enclose the bottom half of the sphere perfectly.

Next, we perform a Solid Difference (2b) between the Sphere (2a) and the half cube (1e) to trim off the bottom half of the sphere. Recall that the ordering of the two inputs for Solid Difference (2b) matters—the object to be trimmed must go first, and the shape to cut out is second. In this case, Solid Difference (2b) produces a hemisphere with a bottom as a closed Brep. To create the dome, we only want the top portion—so we use Deconstruct Brep (2c) to extract the top and the bottom of the hemisphere through its F output as two surfaces, followed by List Item (2d) with i left at default (= 0) to retrieve the top surface.

The outputs of Sphere (2a), Solid Difference (2b), and List Item (2d) are visualized in Figure 7-4.

Note Where to find the nodes in stage 2 of Figure 7-3: Surface ➤ Primitive ➤ Sphere (2a), Intersect ➤ Shape ➤ Solid Difference (2b), Surface ➤ Analysis ➤ Deconstruct Brep (2c), and Sets ➤ List ➤ List Item (2d).

Stage 3: Projecting a Hexagonal Grid onto the Open Hemisphere and Thickening the Edges to Create the Dome

In stage 3 (Figure 7-3), we create a hexagonal grid, project it onto the open hemisphere from stage 2, and use MultiPipe (3g) on the grid edges to create the final dome.

To project a hex pattern onto the hemisphere, we want a hexagonal grid centered on and sitting under the hemisphere that is large enough to cover the hemisphere's entire footprint. To create the hex grid, we use Hexagonal (3a) with a cell Size (S) = 1. Since the hemisphere's radius is set to 9.57 in stage 2, we use Extent X (Ex) = Extent Y (Ey) = 20 for Hexagonal (3a) to ensure the grid is larger than the hemisphere's bottom.

With a default P, Hexagonal (3a) creates a grid on the XY plane with the bottom left corner at (0, 0, 0). However, the hemisphere is centered at (0, 0, 0). Therefore, we must move the grid to also center at (0, 0, 0). To do so, we find the grid's center or, equivalently, its bounding box center, using Bounding Box (3b, *right-click* ➤ *Union Box*) followed by Box Properties (3c). The amount of movement needed to center the grid at (0, 0, 0) is then the vector found by (0, 0, 0) minus the grid's center point (3d). We use the Move node (3e) to perform the move. A visualization of how the grid is moved is shown in Figure 7-4.

With both the hemisphere and the hex grid in place, we use Project Curve (3f) to project the grid onto the hemisphere, followed by MultiPipe (3g) to give the projected edges volume. The finished dome is shown in Figure 7-4.

Note Where to find the nodes in stage 3 of Figure 7-3: Vector ➤ Grid ➤ Hexagonal (3a), Surface ➤ Primitive ➤ Bounding Box (3b, *right-click* ➤ *Union Box*), Surface ➤ Analysis ➤ Box Properties (3c), Maths ➤ Operators ➤ Subtraction (3d), Transform ➤ Euclidean ➤ Move (3e), **Curve ➤ Util ➤ Project Curve (3f)**, and Surface ➤ SubD ➤ MultiPipe (3g).

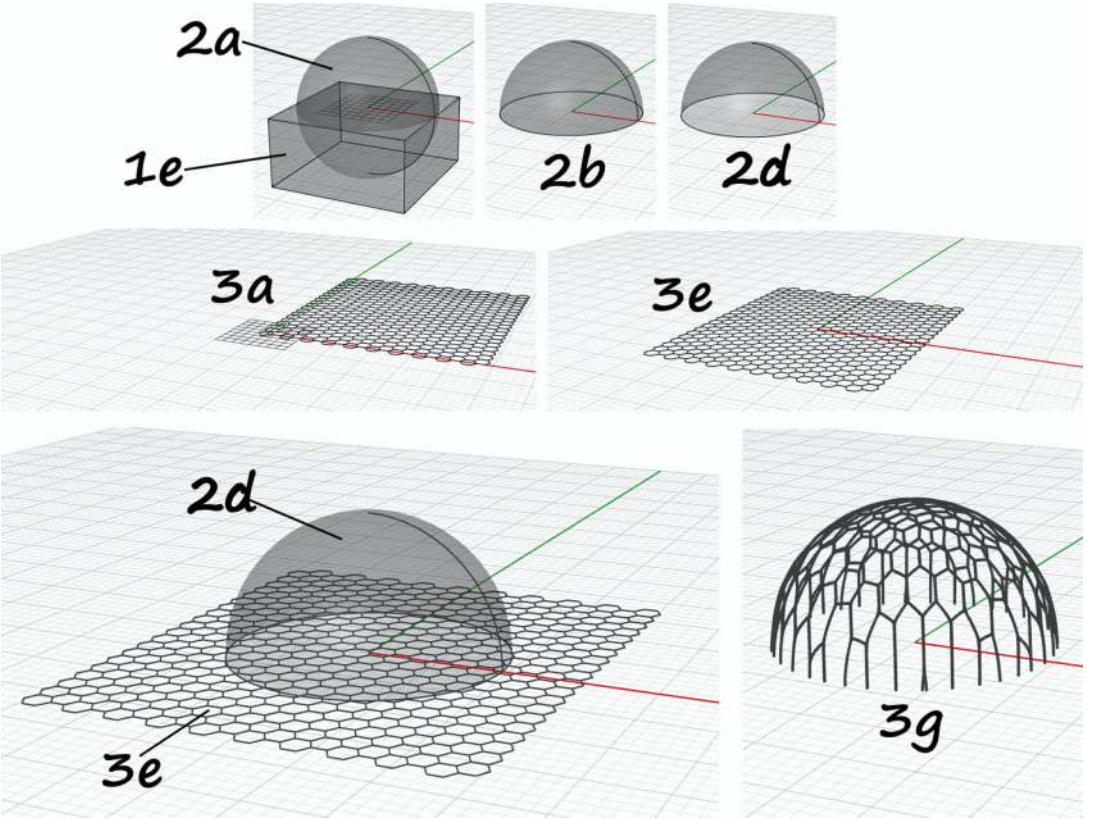


Figure 7-4. *Visualizing intermediate node outputs of Figure 7-3. Top: Performing a Solid Difference (2b) between the sphere (2a) and the half cube (1e) and extracting only the top of the hemisphere (2d). Center: Creating a hex grid (3a) and moving it (3e) so it centers at (0, 0, 0). Bottom: The dome (3g) is produced by projecting the hex grid (3e) on the hemisphere (2d) followed by MultiPipe*

Making Wireframes: Floral Tower

In the third workshop, we'll use Maelstrom to swirl two sets of radial lines and combine the results with a mirrored copy to create a floral pattern, a technique we've used in Chapter 4: Tessellations. The difference here is we'll use the floral pattern to produce a tower-like structure. This process is realized by the node graph in Figure 7-5 in three stages. Stage 1 creates a set of vertical division lines around a cone. Stage 2 slides the division lines' bottom points around the cone's base to create a second set of lines. Stage 3 uses Maelstrom to swirl the two sets of lines and combines the results with a mirrored copy to create a floral, tower-like structure. We'll look at each stage in detail next.

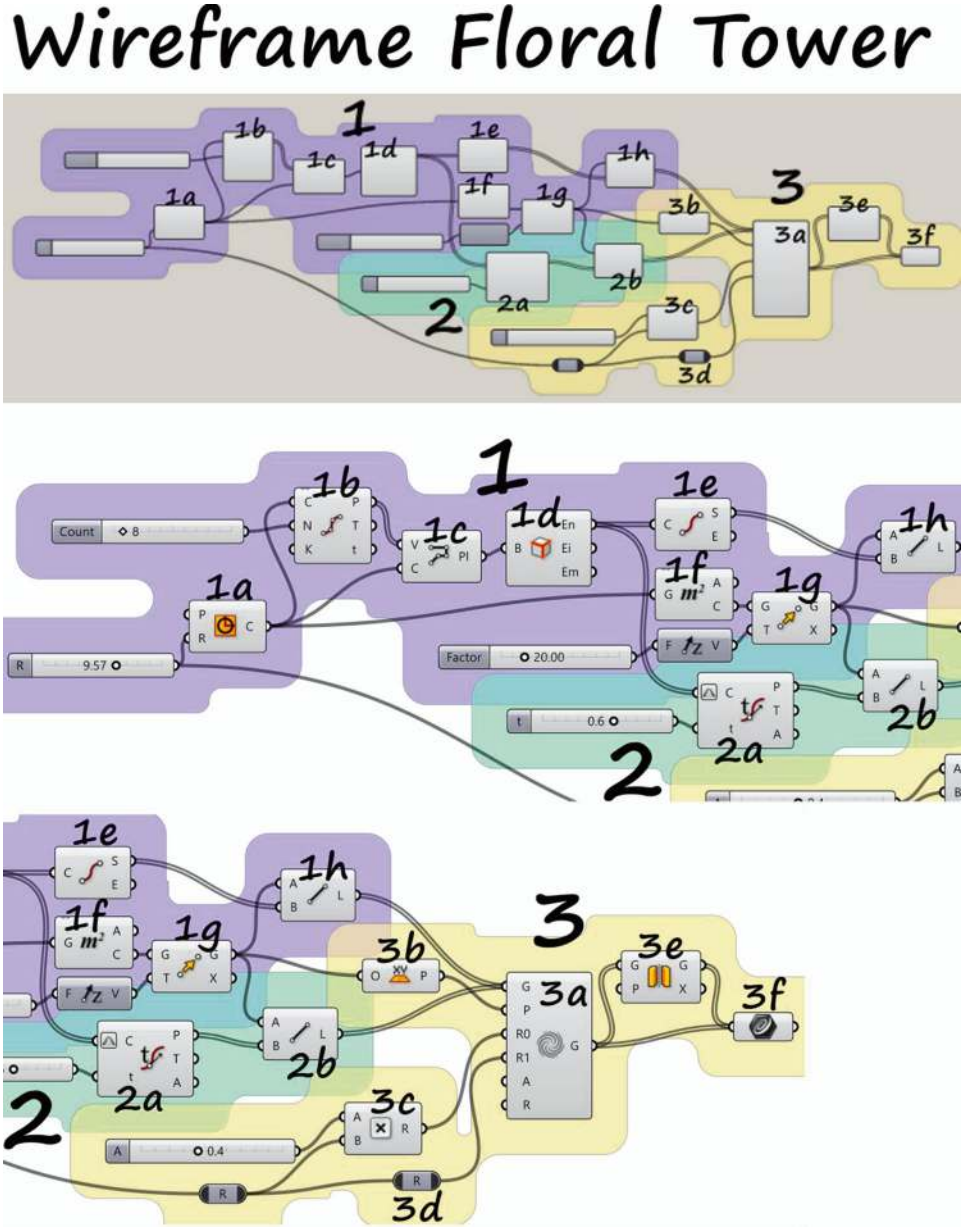


Figure 7-5. *Node graph that generates a floral tower composed of swirled cone division lines and their mirrored copy. Stage 1 creates the first set of lines, linking the cone's tip to division points around the cone's bottom. Stage 2 creates a second set of lines by linking the cone's tip to points slid around the cone's base. Stage 3 uses Maelstrom to swirl the two sets of lines and combines the results with a mirrored copy to produce the floral tower*

Stage 1: Creating Vertical Division Lines Around a Cone

In stage 1 (Figure 7-5), we'll create vertical lines that go from a cone's tip to evenly divided points around the cone's base, as shown in Figure 7-6.

We start with a Circle (1a), divide it evenly into eight points with Divide Curve (1b), and use the points to create a PolyLine (1c). We then use Brep Edges (1d) to extract the individual segments of the PolyLine (1c)—this is so we can slide a point along each segment using Evaluate Curve (2a) later in stage 2 to create a second set of points that offset in unison relative to the first set.

To create the first set of lines, we link Circle (1a) to Area (1f) to find its center and elevate the center to the cone's height using Move (1g) with a scaled Unit Z vector—the elevated center point becomes the cone's tip. We then use End Points (1e) to find the Start (S) point of every segment around the cone's base. Finally, we link the tip (1g) to the points around the cone's base (1e) with Line (1h), creating the set of slanted vertical lines shown in Figure 7-6.

Note Where to find the nodes in stage 1 of Figure 7-5: Curve ➤ Primitive ➤ Circle (1a), Curve ➤ Division ➤ Divide Curve (1b), Curve ➤ Spline ➤ PolyLine (1c), Surface ➤ Analysis ➤ Brep Edges (1d), Curve ➤ Analysis ➤ End Points (1e), Surface ➤ Analysis ➤ Area (1f), Transform ➤ Euclidean ➤ Move (1g), and Curve ➤ Primitive ➤ Line (1h).

Stage 2: Sliding Points Around the Cone's Base to Create a Second Set of Lines

In stage 2 (Figure 7-5), we'll create a second set of lines by linking the cone's tip to points slid along the cone's base.

In stage 1, we used Brep Edges (1d) to extract the line segments of the cone's base. We link Brep Edges (1d) here to Evaluate Curve (2a) to slide a point along each segment by the same amount (t = 0.6), as shown in Figure 7-6. Since we are using a t value between 0 and 1 to specify the amount to slide, we must right-click the C input of Evaluate Curve (2a) and choose "Reparameterize", so t is treated like a *percentage* along each segment's length.

We create the second set of lines by linking the cone's tip (1g) to the points slid around the cone's base (2a) with Line (2b). The second set of lines appear to "rotate" in unison about the cone's axis relative to the set from stage 1.

Note Where to find the nodes in stage 2 of Figure 7-5: Curve ➤ Analysis ➤ Evaluate Curve (2a, *right-click C input* ➤ *Reparameterize*), and Curve ➤ Primitive ➤ Line (2b).

Stage 3: Using Maelstrom to Swirl the Lines and Combining the Results with a Mirrored Copy

In stage 3 (Figure 7-5), we'll use Maelstrom (3a) to swirl the two sets of lines from stages 1 and 2. We then combine the results with a mirrored copy to create the floral tower.

We start by Shift-dragging both sets of cone division lines (1h and 2b) to the G input of Maelstrom (3a). Recall that R0 and R1 are two decimal numbers that define an inner and outer radii, such that the area between the two is the portion of input geometry Maelstrom will "swirl." The closer R0 and R1 are in value, the narrower their range; therefore, the smaller the portion of input geometry gets swirled. Here, we set R1 to the cone's radius and R0 40% of that (times 0.4). The swirled lines are visualized in Figure 7-6.

At this point, the swirled lines look like spirals from the top. To create a floral pattern, we create a mirrored copy (3e) of the swirled lines with P = YZ as the default mirror plane and combine the copy (3e) with the original (3a) by Shift-dragging both to a Curve node (3f). This process is visualized in Figure 7-6.

Note Where to find the nodes in stage 3 of Figure 7-5: **Transform ➤ Morph ➤ Maelstrom (3a)**, Vector ➤ Plane ➤ XY Plane (3b), Maths ➤ Operators ➤ Multiplication (3c), Params ➤ Util ➤ Relay (3d), Transform ➤ Euclidean ➤ Mirror (3e), and Params ➤ Geometry ➤ Curve (3f).

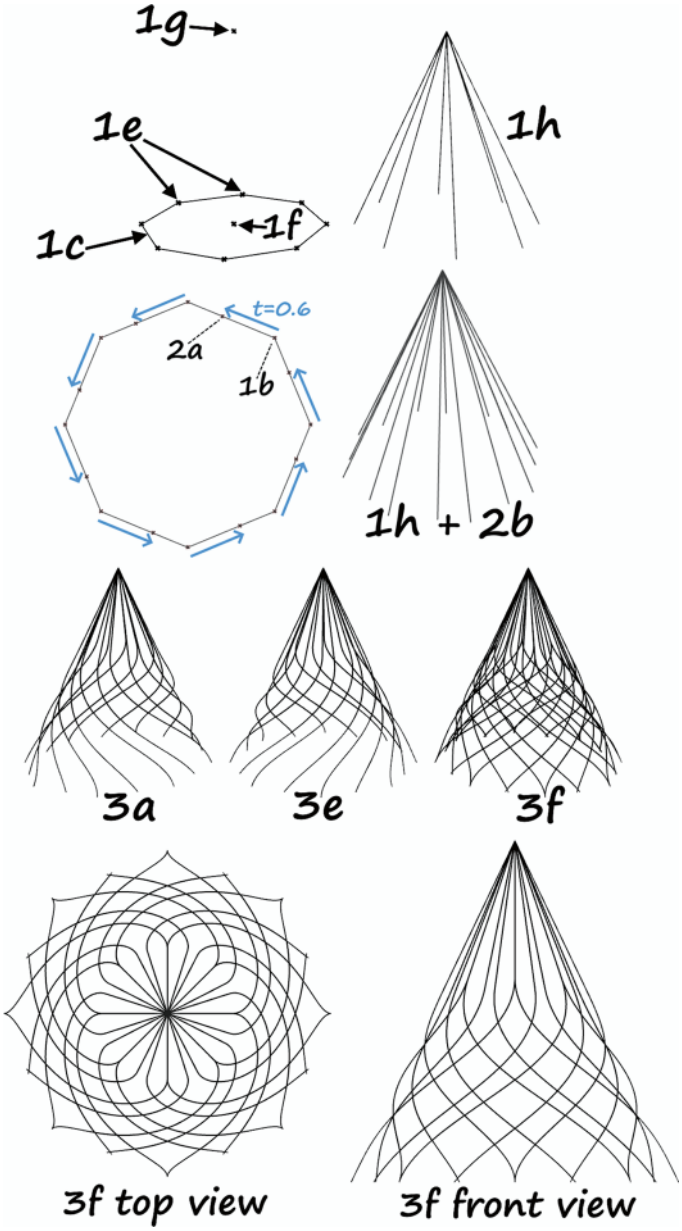


Figure 7-6. *Visualizing node outputs from Figure 7-5. The top shows the formation of the first set of lines (1h) by linking the cone's tip (1g) to evenly divided points around the cone's base (1e). Second row shows using Evaluate Curve (2a) to slide points around the base in unison to create a second set of lines (2b). Third row from the top shows using Maelstrom (3a) to swirl both sets of lines and combining the results with a mirrored copy (3e) to make a floral tower (3f). The fourth row shows the top and front views of the floral tower*

Making Wireframes: From a Mesh

In the fourth workshop, we'll create three types of wireframes—quadrilateral, triangular, and hexagonal, based on an arbitrary mesh.

Quad Wireframe

Figure 7-7 shows the node graph for producing a quad wireframe from the Stanford bunny mesh, internalized in the Mesh node (a) in the provided `*.gh` file. If you hover over Mesh (a) with the mouse cursor, the tool tip that pops up will read "Mesh (V:3555 F:4999)", telling you that the mesh contains 4999 faces (or polygons), which is very dense and won't work well as the basis for producing a wireframe.

We can use Quad Remesh (b) to convert the bunny's faces to quads and reduce its polygon count at the same time. You specify the parameters for Quad Remesh (b) through a Quad Remesh Settings node (c), which by default sets the target polygon count (Tc) to 1000 and produces reasonable face sizes on the bunny. After that, we link Quad Remesh (b) to MultiPipe (d) to produce the wireframe structure by giving thickness to the bunny's edges, as shown in Figure 7-7.

Note Where to find the nodes in Figure 7-7: Params ➤ Geometry ➤ Mesh (a), Mesh ➤ Triangulation ➤ Quad Remesh (b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (c), and Surface ➤ SubD ➤ MultiPipe (d).

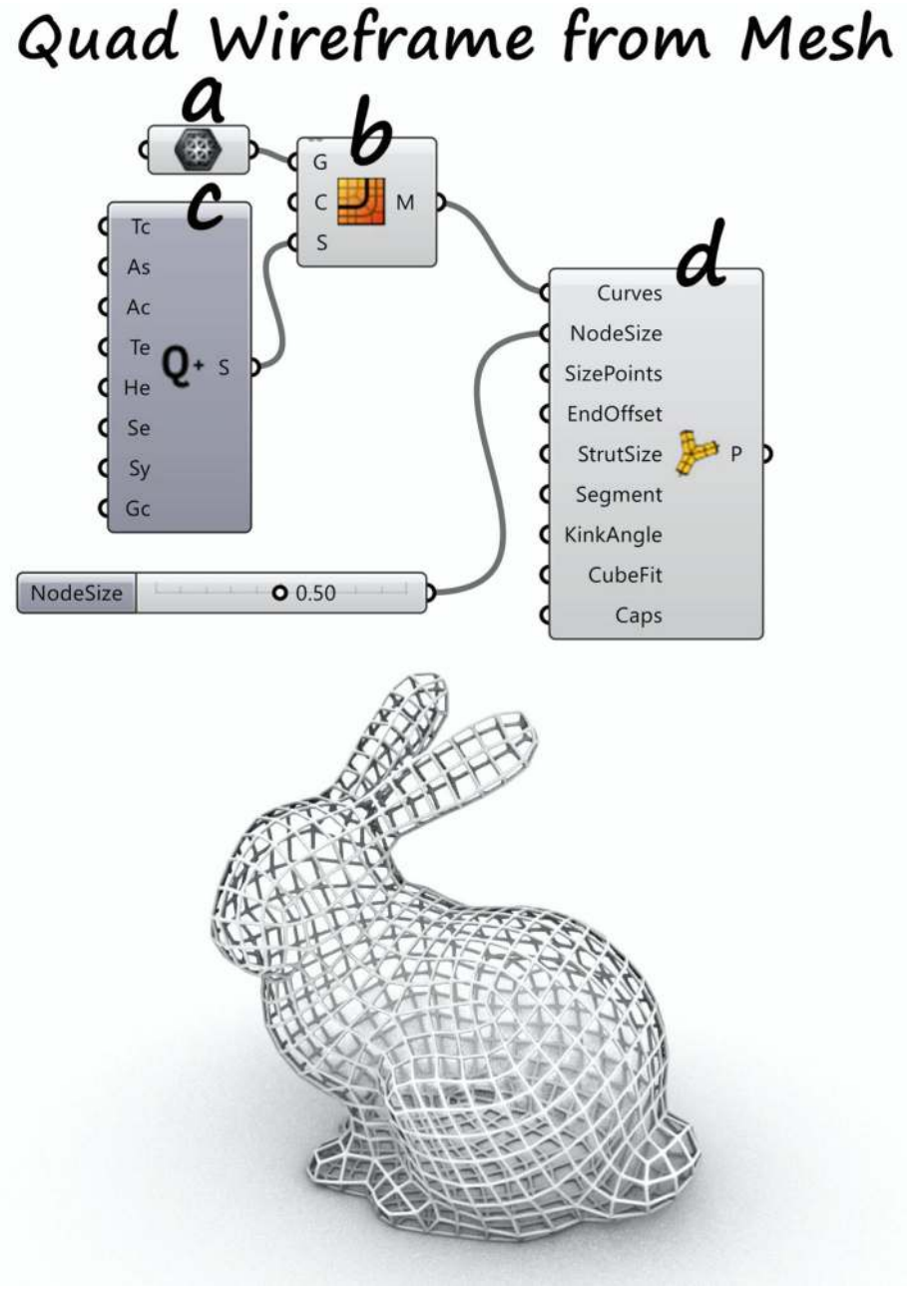


Figure 7-7. *Node graph for creating a quad wireframe based on an input mesh. The mesh (a) goes through a Quad Remesh (b) with the default value of Target Count (Tc) = 1000, followed by MultiPipe (d) to give the processed edges volume*

Triangular Wireframe

We can modify the node graph in Figure 7-7 to create a triangular wireframe, as shown in Figure 7-8, by inserting a TriRemesh node (e) between Quad Remesh (b) and MultiPipe (d). We still use Quad Remesh (b) first to reduce the bunny's polygon count to a manageable level, before using TriRemesh (e) to convert the bunny's faces to near equilateral triangles. Note that the triangulated mesh is output via Triangulation (T), which we link to the Curves input of MultiPipe (d).

We set the Length input to 8 for TriRemesh (e), which controls the side length for the triangular faces, and NodeSize to 0.50 for MultiPipe (d), which controls the pipe radius, to produce the triangular wireframe shown on the bottom of Figure 7-8.

Note Where to find the nodes in Figure 7-8: Params ➤ Geometry ➤ Mesh (a), Mesh ➤ Triangulation ➤ Quad Remesh (b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (c), Surface ➤ SubD ➤ MultiPipe (d), and **Mesh ➤ Triangulation ➤ TriRemesh (e)**.

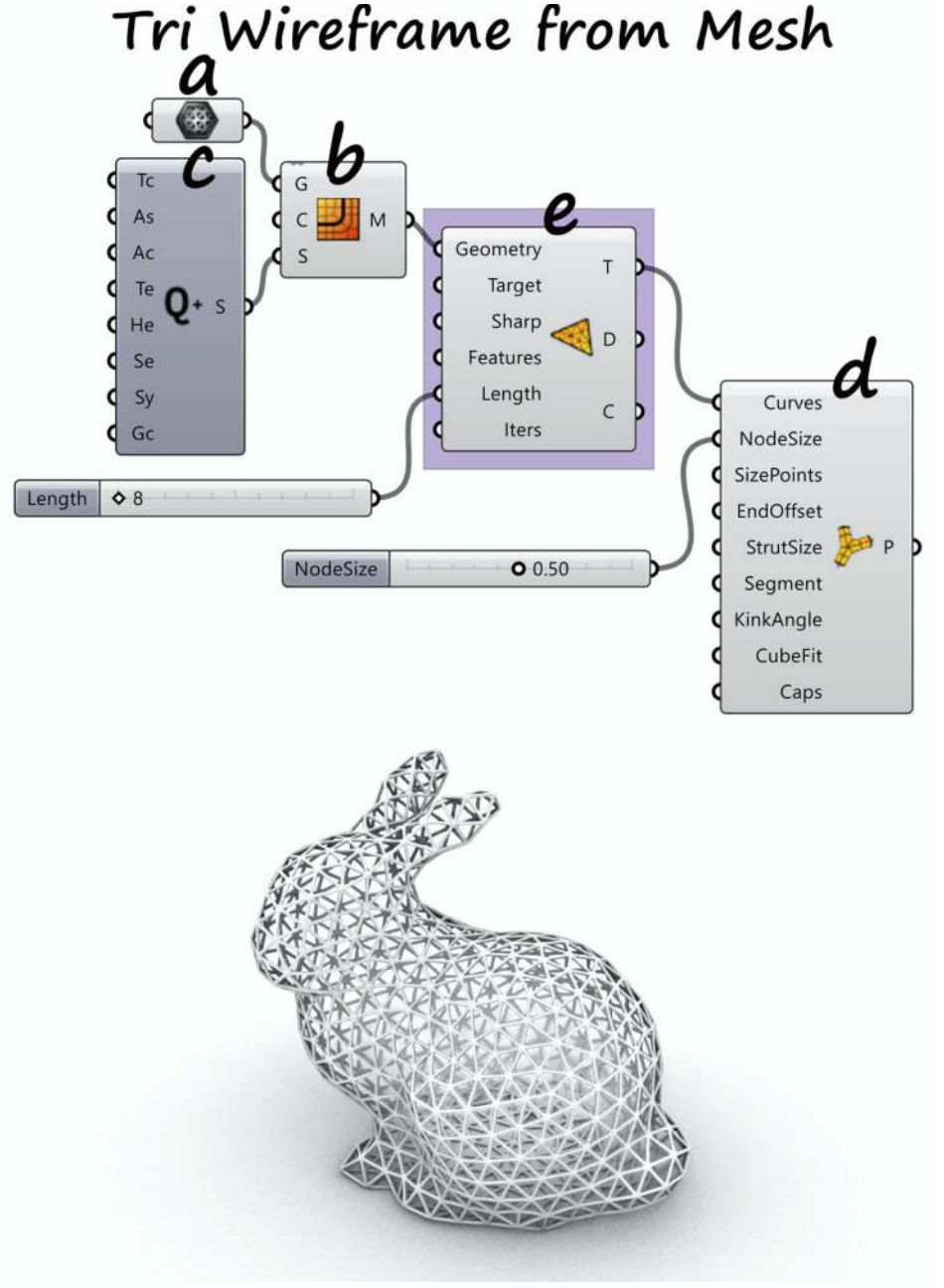


Figure 7-8. *Node graph for creating triangular wireframes of the input bunny mesh. The graph is identical to Figure 7-7 except for the addition of TriRemesh (e). We reduce the bunny's polygon count with Quad Remesh (b), convert its faces to near equilateral triangles with TriRemesh (e), and give its edges volume with MultiPipe (d)*

Hex Wireframe

We can continue modifying the node graph in Figure 7-8 to produce a (near) hexagonal wireframe, as shown in Figure 7-9. Instead of Triangulation (T), we link TriRemesh's Dual (D) output to MultiPipe (d).

The "dual" of a mesh consists of edges linking neighboring face centers in the original mesh. You can visualize this process by linking the T output of TriRemesh (e) to Face Boundaries (f) to find the edges framing each face as a curve, followed by Area (g) to find the curve's (therefore the face's) center, as shown in Figure 7-9. Linking neighboring face centers of the mesh to one another then produces either hexagons or shapes close to a hexagon.

Note Where to find the nodes in Figure 7-9: Params ➤ Geometry ➤ Mesh (a), Mesh ➤ Triangulation ➤ Quad Remesh (b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (c), Surface ➤ SubD ➤ MultiPipe (d), **Mesh ➤ Triangulation ➤ TriRemesh (e)**, **Mesh ➤ Analysis ➤ Face Boundaries (f)**, and **Surface ➤ Analysis ➤ Area (g)**.

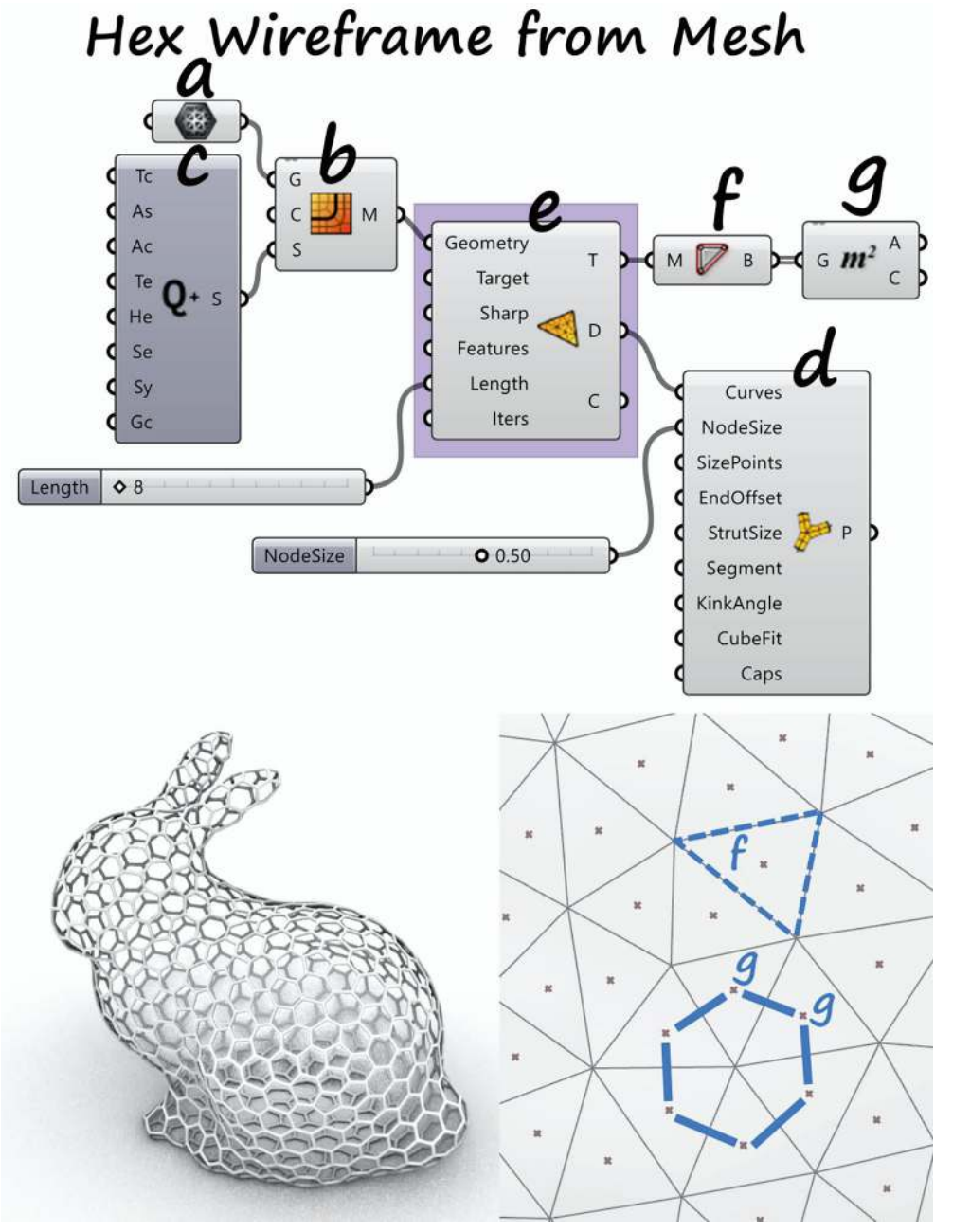


Figure 7-9. *Node graph for creating a near hex wireframe from the bunny's dual mesh (D output of TriRemesh (e)). The dual mesh is created by linking center points of neighboring faces on the original mesh, as shown in the lower-right corner. You can find and visualize the center points of mesh faces by linking the mesh to Face Boundaries (f) followed by Area (g)*

Making Wireframes: By Cracking a Surface

In the fifth workshop, we'll make another set of wireframes by cracking a cube and a sphere then MultiPipe along the cracked edges.

Cracked Cube Wireframe

The node graph in Figure 7-10 shows how to simulate cracks in a cube using a 3D Voronoi and then MultiPipe along the cracked edges to create a wireframe. The process is broken into two stages—stage 1 creates a point cloud in a cube and from which a 3D Voronoi diagram. Stage 2 isolates the edges of the 3D Voronoi and MultiPipe them to create the wireframe. We'll look at each stage in detail next.

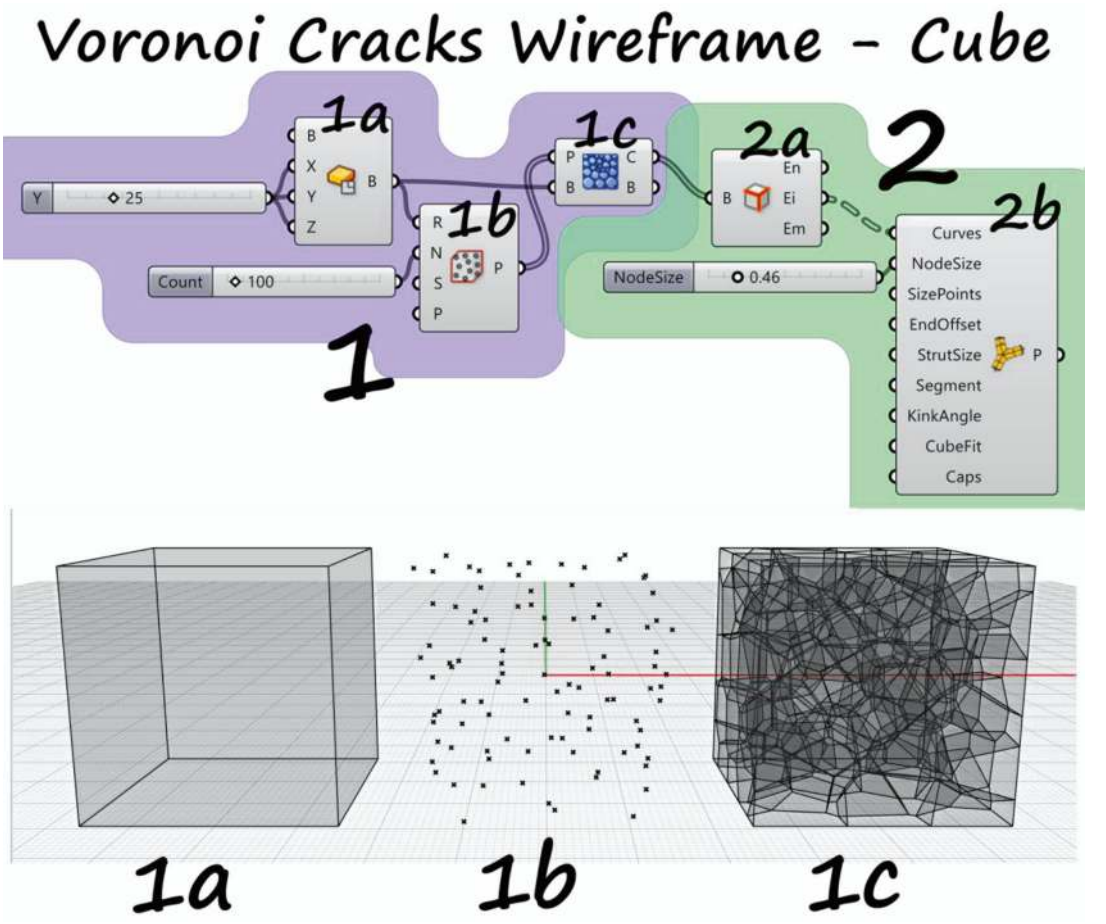


Figure 7-10. *Node graph for producing a wireframe from a cracked cube. In stage 1, we scatter points (1b) inside a cube (1a) and from which create a 3D Voronoi (1c) that resembles cracks. In stage 2, we isolate the cracks' edges (2a) and MultiPipe (2b) to give the edges volume. The bottom shows visualization of the intermediate node outputs*

Stage 1: Creating a 3D Voronoi Inside a Cube to Emulate Cracks

In stage 1 (Figure 7-10), we create a 3D Voronoi inside a cube to emulate cracks. We create a cubic surface by using Center Box (1a) with its X, Y, and Z inputs controlled by the same Number Slider. We then use Populate 3D (1b) to create a point cloud inside the cube, which is in turn used as input for Voronoi 3D (1c). The cells of Voronoi 3D resemble cracked chunks. The outputs of nodes 1a to 1c are visualized on the bottom of Figure 7-10.

Recall from Chapter 6 that a Voronoi diagram partitions a plane into convex polygons based on a set of "generating points." Each polygon contains only one generating point, and every point inside a polygon is closest to its generating point than to any other point (Weisstein, 2026). The Voronoi 3D node (1c) generalizes this concept from 2D to 3D and partitions a 3D volume into convex polyhedrons

---

Note Where to find the nodes in stage 1 of Figure 7-10: Surface ➤ Primitive ➤ Center Box (1a), **Vector ➤ Grid ➤ Populate 3D (1b)**, and **Mesh ➤ Triangulation ➤ Voronoi 3D (1c)**.

Stage 2: MultiPipe the Cracks' Edges to Create the Wireframe

In stage 2 (Figure 7-10), we create the cracked wireframe depicted in Figure 7-11 by isolating edges of the Voronoi 3D (1c) with Brep Edges (2a), followed by MultiPipe (2b) to give the edges volume.

Note Where to find the nodes in stage 2 of Figure 7-10: Surface ➤ Analysis ➤ Brep Edges (2a) and Surface ➤ SubD ➤ MultiPipe (2b).

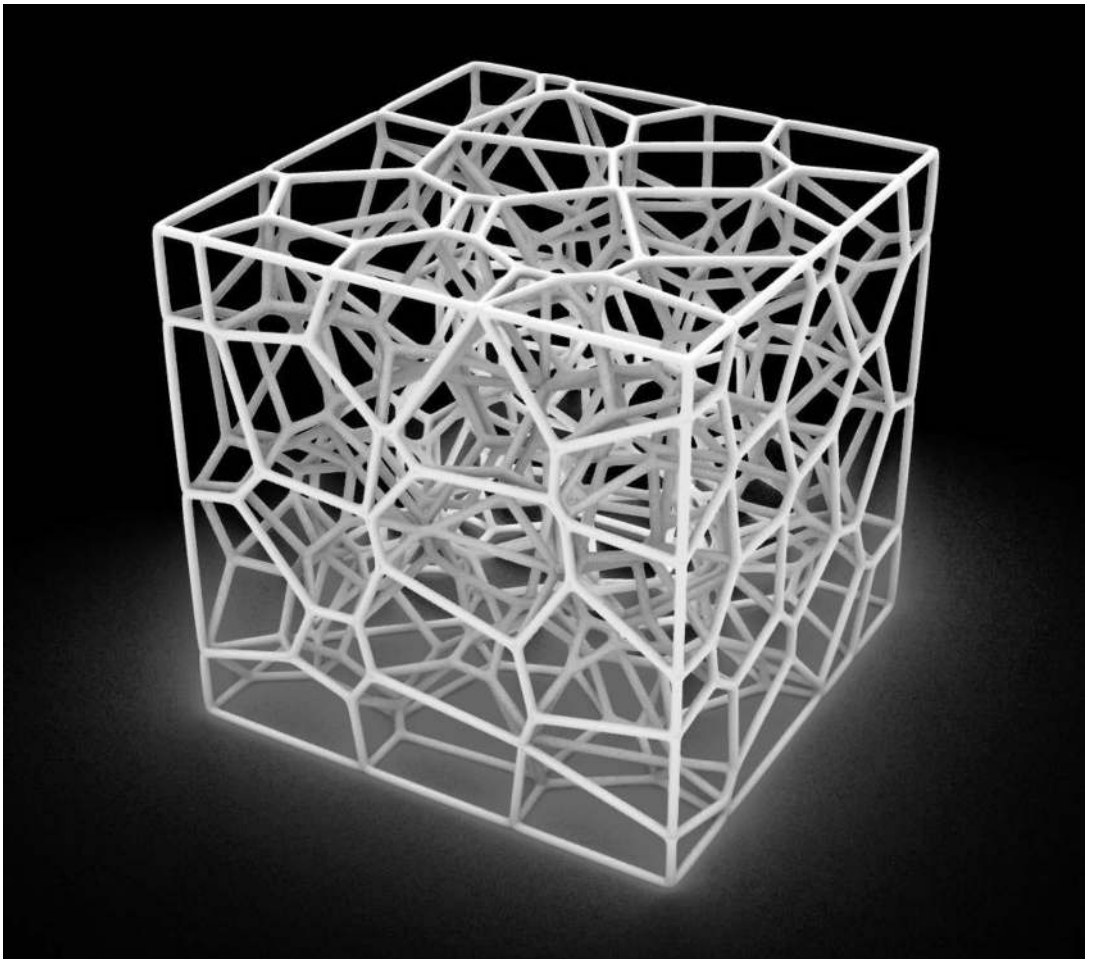


Figure 7-11. *The cracked wireframe structure produced by the node graph in Figure 7-10. We create a 3D Voronoi diagram from points scattered inside a cube and MultiPipe the Voronoi cells' edges*

Cracked Sphere Wireframe

Let's think about how we can modify the node graph from Figure 7-10 to create cracked wireframes of other shapes. Since Voronoi 3D (1c) only accepts a box for defining the diagram's boundary (via the input Box (B)), we'll start with a cubic Voronoi and use "Brep | Brep" to trim it with a shape of our choosing.

Figure 7-12 shows an example where we create a Sphere (2a) using the same radius as the cube (1a) enclosing the Voronoi, and perform a Brep | Brep (2b) intersection between Voronoi 3D (1c) and the Sphere (2a) to create a spherical wireframe. Since Brep | Brep (2b) outputs the intersection as curves, we can use them directly as inputs for MultiPipe (2c) without the help of Brep Edges.

The bottom of Figure 7-12 visualizes outputs of intermediate nodes described in this section. The final cracked spherical wireframe is shown in Figure 7-13.

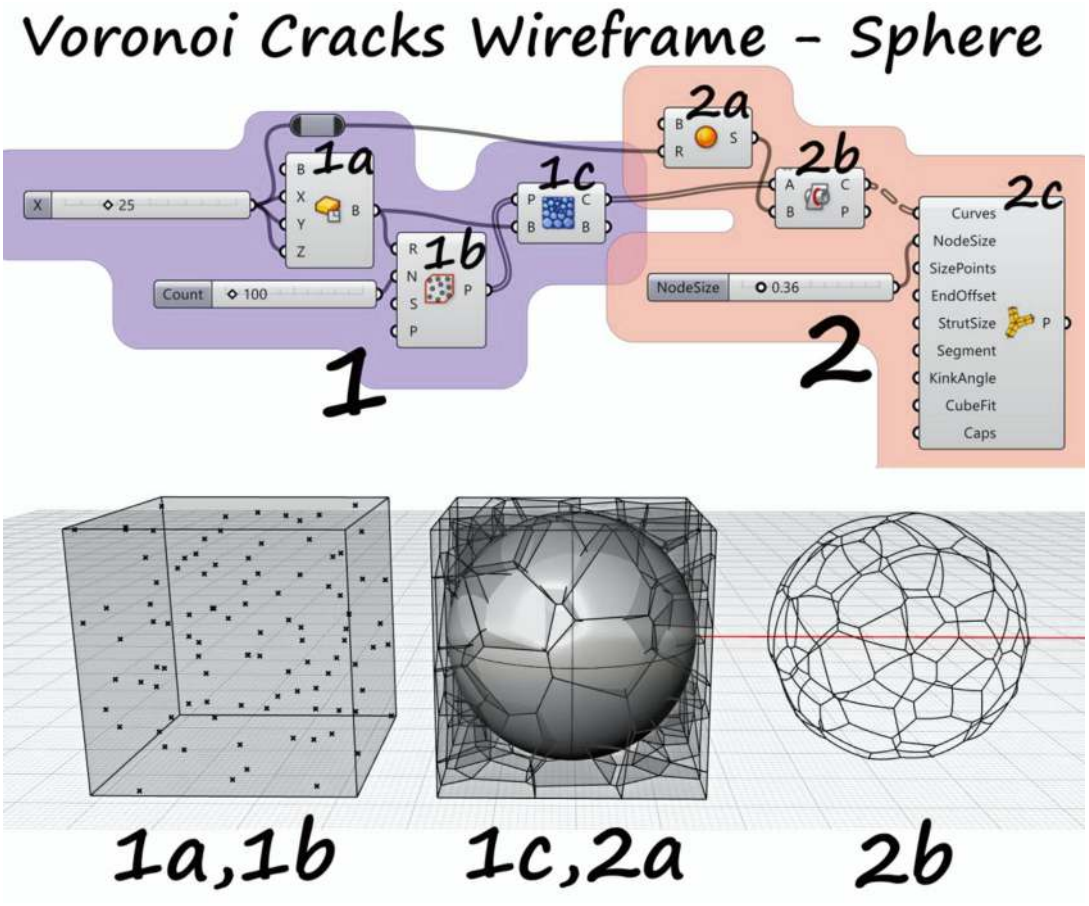


Figure 7-12. *Modified version of Figure 7-10 that trims a 3D cubic Voronoi with a sphere by performing a Brep | Brep (2b) intersection. The bottom shows outputs of nodes 1a to 2b visualized*

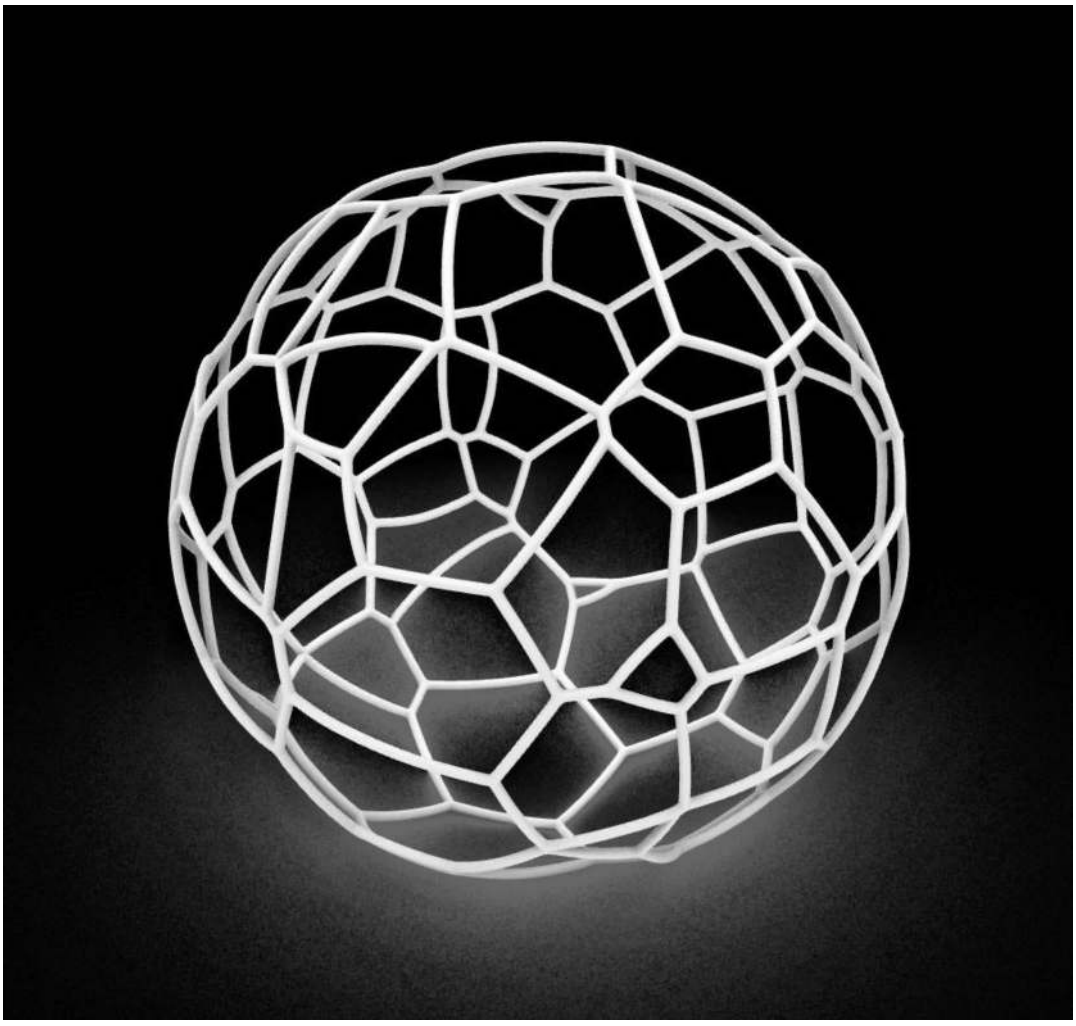


Figure 7-13. *The spherical cracked wireframe produced by the node graph in Figure 7-12. Since Voronoi 3D only accepts a box for defining a boundary for the diagram (via the Box (B) input), we start with a cubic Voronoi and perform a Brep | Brep intersection with a sphere to trim it*

Making Cracks: Exploded Cube

For the sixth workshop, we'll switch gears and create a node graph for generating a cracked and exploded cube over a two-stage process, as shown in Figure 7-14. In stage 1, we create a cube, scatter points inside it, and from which create a 3D Voronoi diagram,

whose cells resemble cracked chunks. In stage 2, we displace the chucks radially outward from the cube's center to mimic the look of an explosion.

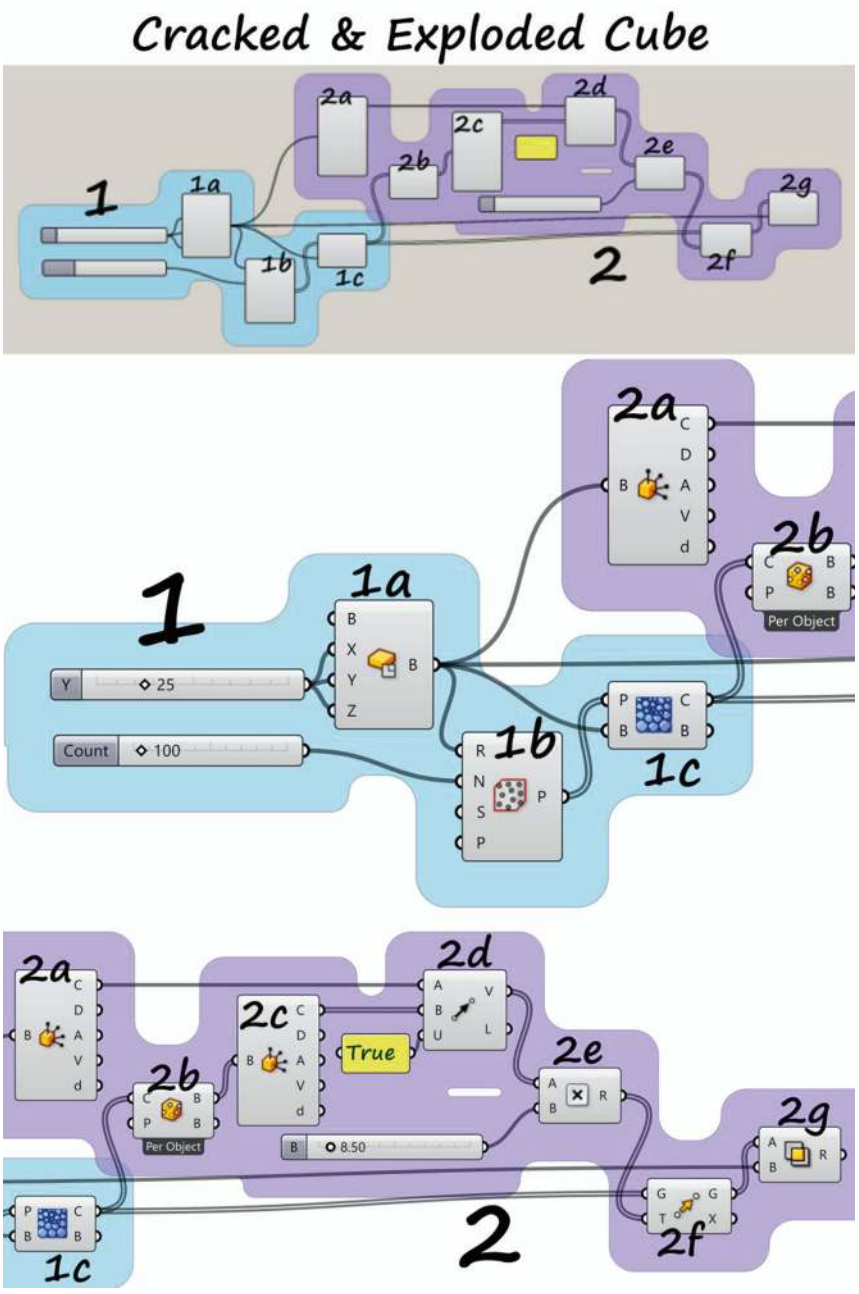


Figure 7-14. *Node graph for generating a cracked and exploded cube. Stage 1 creates a 3D Voronoi diagram inside a cube to emulate cracked chucks, and stage 2 "explodes" the chunks by displacing them radially outward from the center of the cube*

Stage 1: Creating a 3D Voronoi Diagram Inside a Cube to Emulate Cracked Chunks

Stage 1 in Figure 7-14 is identical to stage 1 used for creating cracked wireframes (Figures 7-10 and 7-12). We create a cube using Center Box (1a) with its X, Y, and Z inputs controlled by the same Number Slider, followed by Populate 3D (1b) to create a point cloud inside the cube, which is in turn used as input for Voronoi 3D (1c). The cells of Voronoi 3D (1c) resemble cracked chunks.

Note Where to find the nodes in stage 1 of Figure 7-10: Surface ➤ Primitive ➤ Center Box (1a), Vector ➤ Grid ➤ Populate 3D (1b), and Mesh ➤ Triangulation ➤ Voronoi 3D (1c).

Stage 2: Displacing Chunks Radially Outward from the Cube's Center to Create Explosion

To create the impression of an explosion, we want to offset the Voronoi cells radially outward from the cube's center. To make the final model look cleaner, we perform a Solid Intersection between the displaced cells and the cube to trim off any bits fallen outside the cube's boundary.

The 3D Voronoi in Figure 7-14 contains 100 cells, which are difficult to visualize. Therefore, I've made a simplified version in Figure 7-15 with only three cells to show you the directions of displacement vectors. Let's cross-reference the nodes from Figure 7-14 with the visuals in Figure 7-15. We start by finding the center of the cube with Box Properties (2a). We then derive the bounding box of *each* Voronoi cell by linking Voronoi 3D (1c) to Bounding Box (2b, *Per Object*).

Next, we derive the vectors for displacing the Voronoi cells. We find the cells' bounding box centers via Box Properties (2c). Since we want the cells to move *away* from the cube's center, the vectors should go *from* the cube's center *to* the individual cells' bounding box centers. Therefore, we hook the cube's center (C output of 2a) and the cell bounding box centers (C output of 2c) to the A and B inputs of Vector 2Pt (2d), respectively, which give us vectors *from A to B*. We also set Vector 2Pt's U input to True to *unitize* the vector, which means to make the vectors' lengths one but retaining their

directions—this is so we can scale the vectors easily, by multiplying (2e) them with a Number Slider. Therefore, the more we drag the slider to the right, the longer the vectors and the more the cells separate.

Finally, we displace the Voronoi cells using a Move (2f) node, with the *scaled* displacement vectors (2e) as the T input, to create the impression of an explosion. To make the explosion neater, we perform a Solid Intersection (2g) between the displaced cells (2f) and the original cube (1a), to trim off bits falling outside the cube's boundary.

Figure 7-16 shows a side-by-side comparison of the "cracked cube" (1c) and the final output of the node graph in Figure 7-14 (2g), where the cracked "chunks" (or 3D Voronoi cells) have been displaced and trimmed.

Note Where to find the nodes in stage 2 of Figure 7-14: Surface ➤ Analysis ➤ Box Properties (2a), Surface ➤ Primitive ➤ Bounding Box (2b, *Per Object (default)*), Surface ➤ Analysis ➤ Box Properties (2c), Vector ➤ Vector ➤ Vector 2Pt (2d), Maths ➤ Operators ➤ Multiplication (2e), Transform ➤ Euclidean ➤ Move (2f), and **Intersect ➤ Shape ➤ Solid Intersection (2g)**.

Tip "Unitize" a vector is the same as "normalize" a vector.

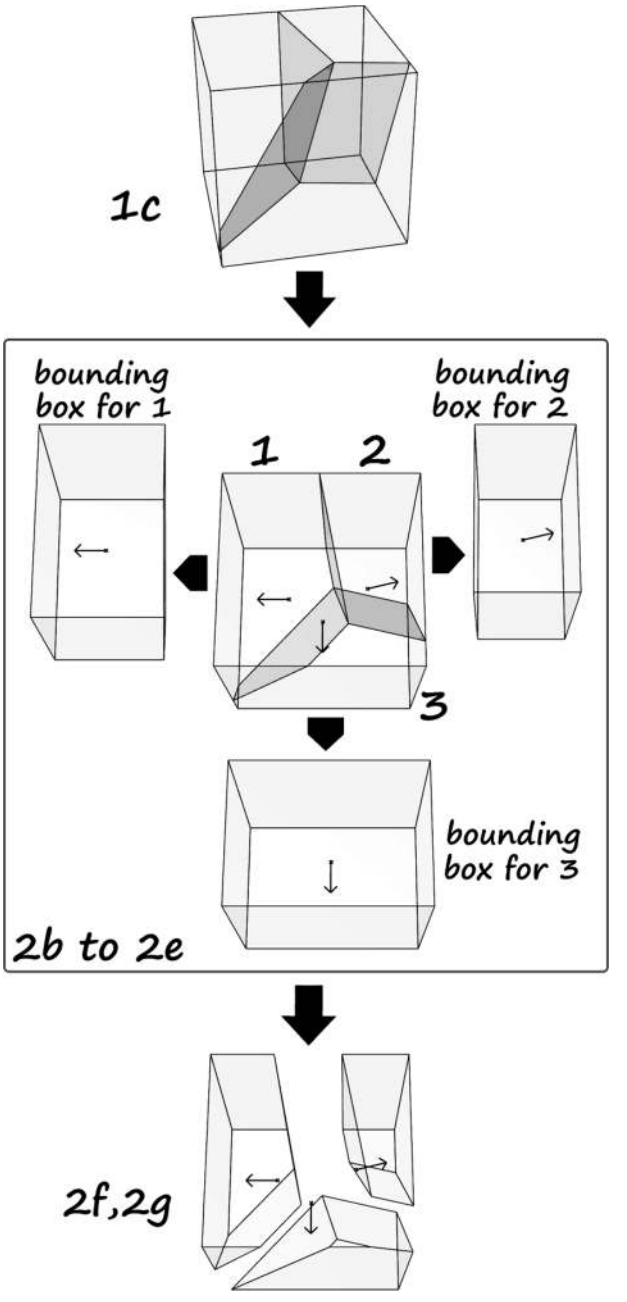


Figure 7-15. *Displacing the 3D Voronoi cells radially outward from the cube's center. The top shows a simplified depiction of 1c in Figure 7-14 with three cells. The middle shows displacement vectors being calculated by subtracting the cube's center from each cell's bounding box center. The bottom shows the cells after movements are applied*

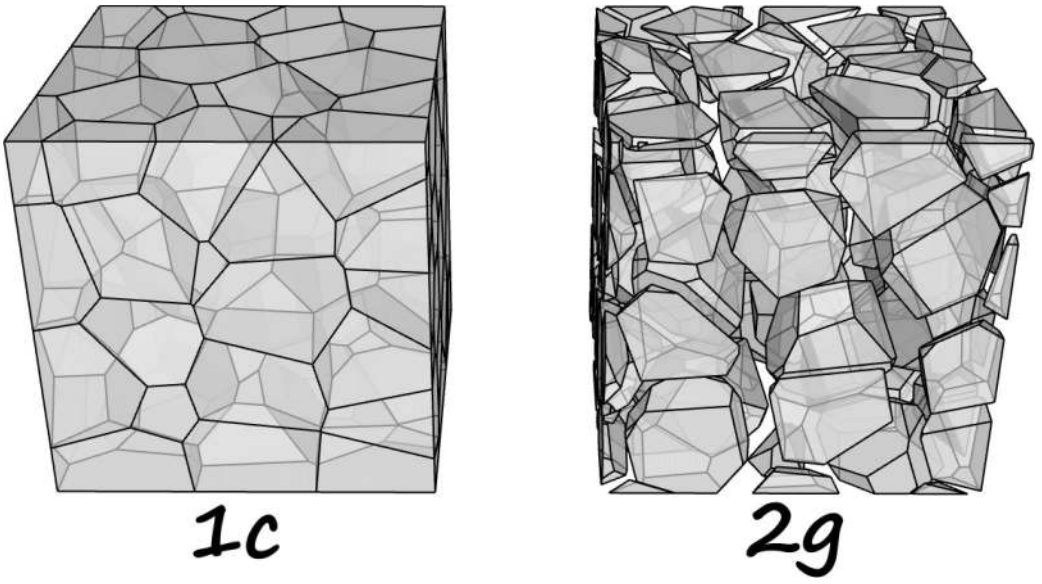


Figure 7-16. *Visualizing the "cracked cube" (left) and the "exploded cube" (right), which are outputs of nodes 1c and 2g from the graph in Figure 7-14*

Variation: Exploded Sphere

Let's try making exploded chunks out of other shapes. Since Voronoi 3D only accepts a box for defining the diagram's boundary (via the input Box (B)), we must start with a box-shaped Voronoi and use Solid Intersection to trim it later with the shape of our choosing. Figure 7-17 shows a modified version of the node graph from Figure 7-14, where we crack and explode a sphere instead of a cube. The only modification is the two-node addition marked "stage 3" (3a and 3b).

We create a Sphere (3a) with the same radius as half the X, Y, and Z extent of the Center Box (1a) that defines the cube, making the cube the same size as the sphere's bounding box. We then create the "cracked sphere" shown on the left of Figure 7-18 by performing a Solid Intersection (3b) between the Sphere (3a) and the cubic Voronoi 3D (1c).

Remember that we trimmed the exploded cube in Figure 7-14 by performing a Solid Intersection (2g) between the displaced Voronoi cells (A input) and the cube (B input)? In Figure 7-17, we simply swap the B input of 2g with the Sphere (3a), therefore trimming the Voronoi cells with the Sphere instead of the cube, to create the exploded sphere, which is shown on the right of Figure 7-18.

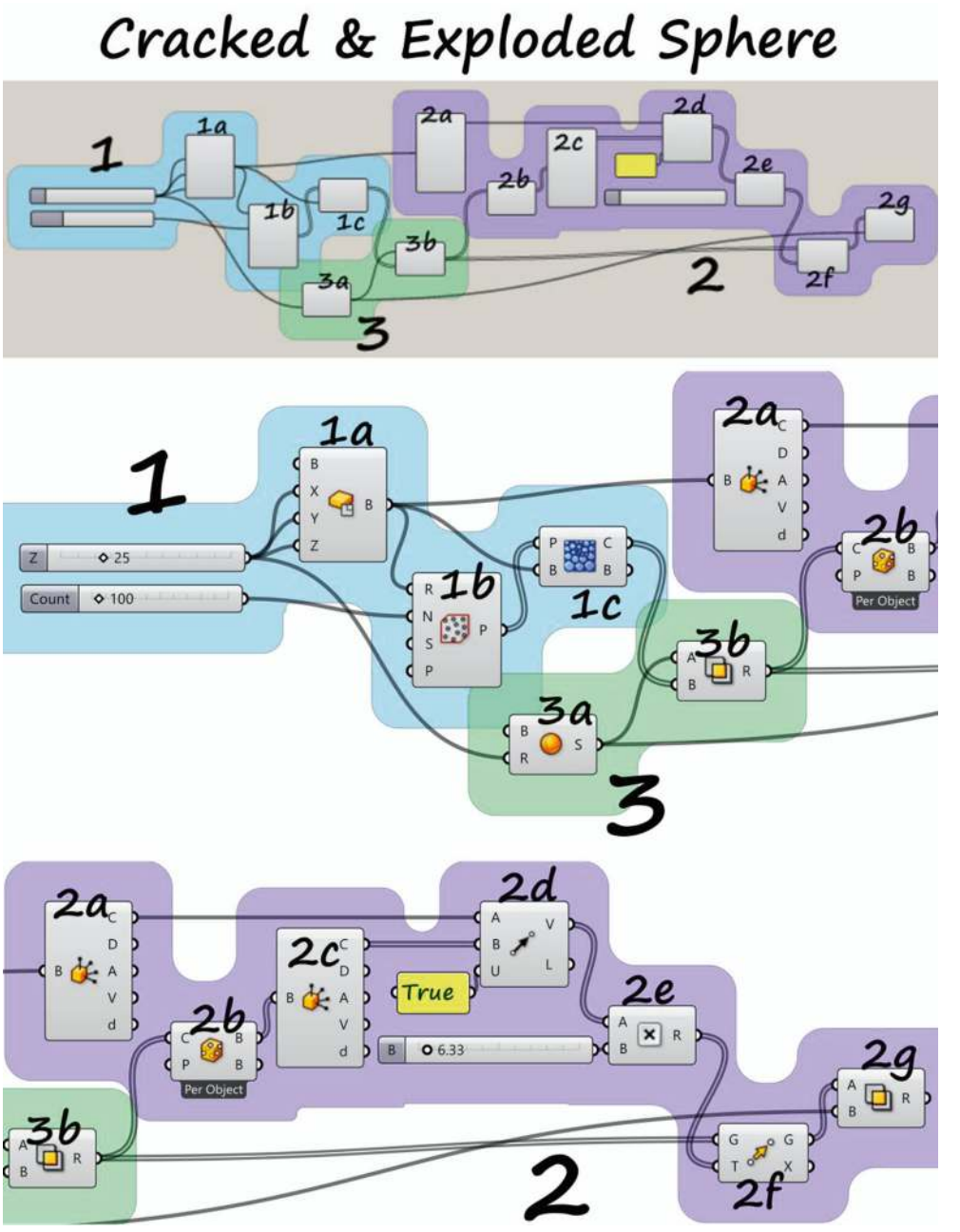


Figure 7-17. *Adding stage 3 with two nodes (3a and 3b) to the graph from Figure 7-14, to make the graph produce cracked and exploded spheres instead of cubes. The Sphere (3a) is used to perform a Solid Intersection with Voronoi 3D (1c) for the cracked sphere and with the displaced Voronoi cells (2f) to produce the exploded sphere*

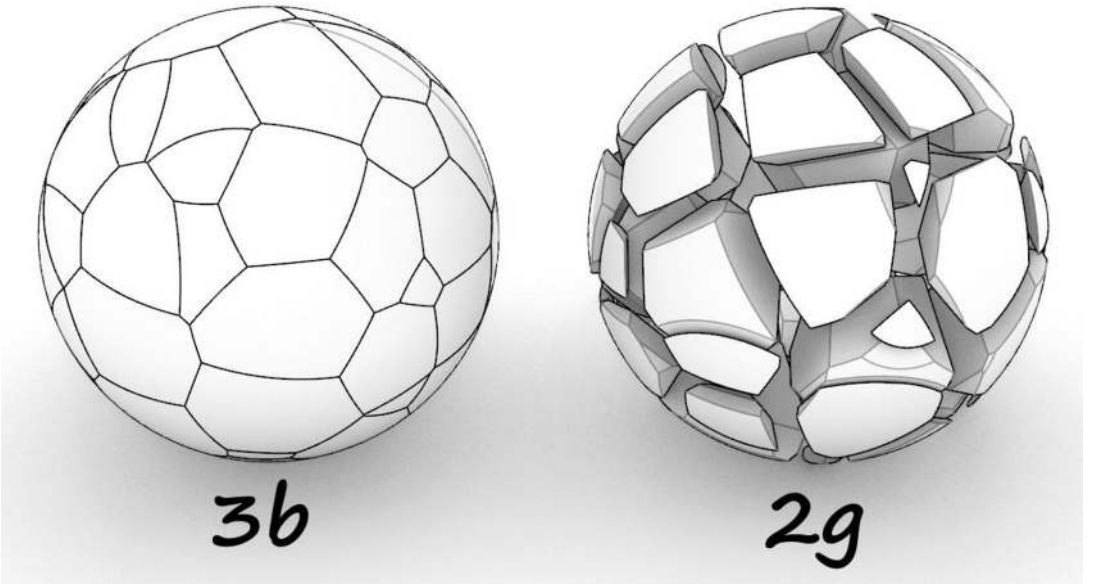


Figure 7-18. *Visualization of the cracked sphere (left) and exploded sphere (right) produced by the node graph from Figure 7-17*

Making Cracks: Exploded Mesh

We'll now generalize the concept of trimming cubic Voronoi with Solid Intersection to using arbitrary meshes as trimmer objects. Instead of a Sphere, we'll modify the node graph from Figure 7-17 to use the Stanford bunny mesh that's been simplified and converted to SubD as the trimmer object, as shown in Figure 7-19.

Most of the changes are in the added stage 0, where we reduce the polygon count of the bunny mesh and convert it to SubD. We then use a slightly enlarged version of the bunny's bounding box to create the 3D Voronoi to ensure that it will fully enclose the bunny. The bunny SubD is used in a Solid Intersection (1d) with Voronoi 3D (1c) to create the cracked bunny

The process for creating the explosion in stage 2 is unchanged, except for the bunny SubD is now used in the Solid Intersection (2g) with displaced Voronoi cells to create the exploded bunny.

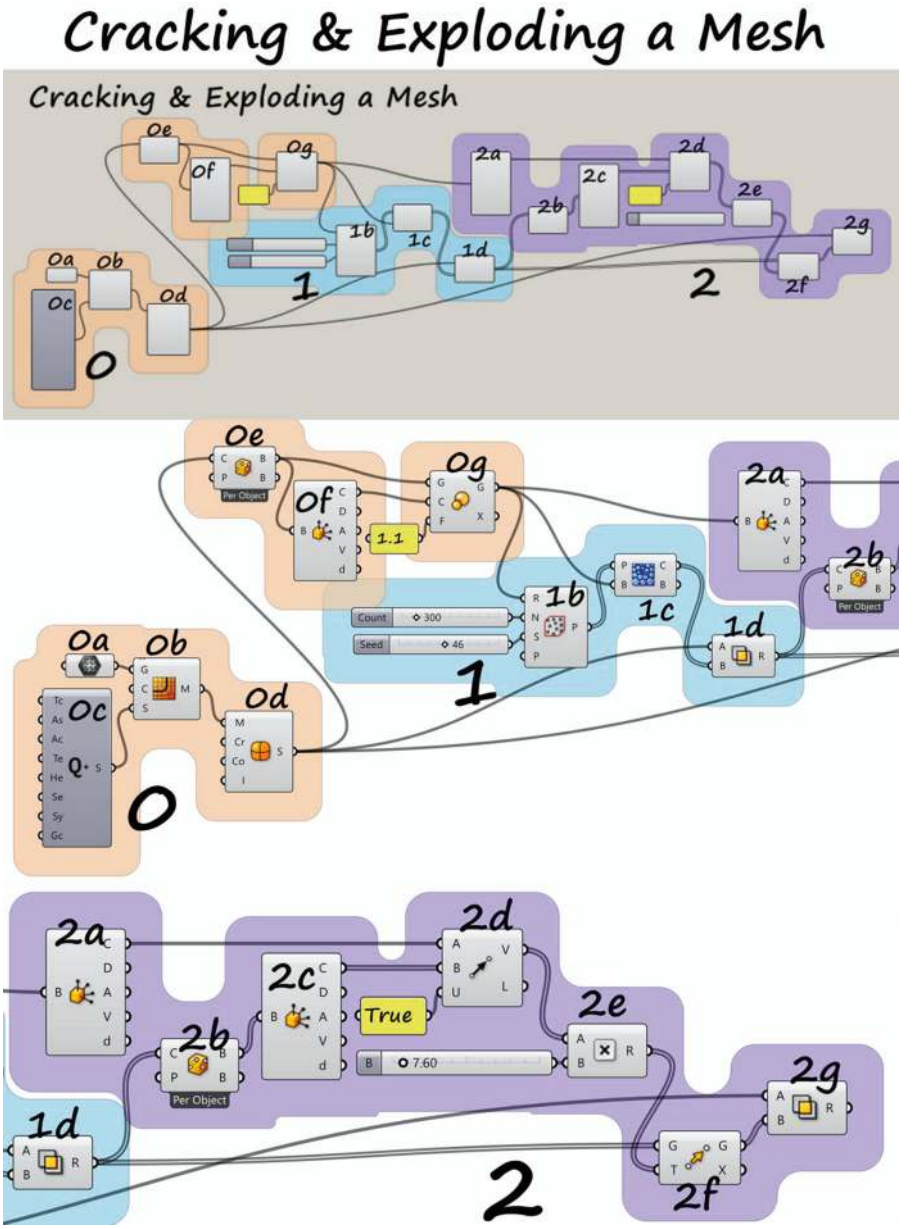


Figure 7-19. *Node graph from Figure 7-17 modified to generate cracked and exploded bunnies. A stage 0 is added to simplify and convert the bunny mesh to SubD, as well as enlarge the mesh's bounding box for creating the 3D Voronoi, to ensure that the bunny is fully contained. The bunny SubD is used in a Solid Intersection (1d) with the 3D Voronoi to create the cracked bunny. A second Solid Intersection (2g) between the bunny SubD and the displaced Voronoi cells creates the exploded bunny*

Stage 0: Simplifying + Converting the Mesh to a SubD and Finding the Mesh's Enlarged Bounding Box

Let's take a closer look at stage 0 in Figure 7-19, where we reduce the mesh's polygon count, convert it to a SubD, and enlarge its bounding box (110%) in preparation for creating the 3D Voronoi.

If you hover over the Mesh node (0a) internalized with the bunny data, a tool tip appears with the message "Mesh (V:3555 F:4999)", telling you that the mesh has 3555 vertices and 4999 polygons, which is very dense. We can use Quad Remesh (0b) to reduce the bunny's polygon count with parameters specified via a Quad Remesh Settings node (0c), which by default has Target Count (Tc) = 1000.

Next, we link the output of Quad Remesh (0b) to SubD from Mesh (0d) to convert the mesh to a SubD object—necessary since SubDs can interoperate with Breps, which we need in order to perform Solid Intersection between the bunny and the output of Voronoi 3D. The Voronoi 3D cells are closed Breps.

The last step is to enlarge the bunny's bounding box about its center to 110%—reason being we plan to use the enlarged bounding box as the boundary for the 3D Voronoi in stage 2. The Voronoi must be large enough to fully enclose the bunny, so when a Solid Intersection is carried out between the two, the result will capture the entire bunny without any part of it being cropped off. To do so, we link SubD from Mesh (0d) to Bounding Box (0e, *Per Object*) followed by Scale (0g). We find the bounding box's center via the C output of Box Properties (0f), which is in turn linked to the C input of Scale (0g). We set the F input of Scale (0g) to 1.1 via a Panel to make the scaling 110%.

Figure 7-20 visualizes the bunny in its original state (0a), simplified with Quad Remesh (0b), and converted to SubD (0d).

Note Mesh objects *cannot* interoperate with Breps and must be converted to SubDs first.

Tip Recall that you can link a Panel containing number(s) to inputs expecting those values. A shortcut for doing so is to double-click (or spacebar) over the canvas to bring up the search box and type in " (double quote) followed by the intended value, for example, "1.1.

Note Where to find the nodes in stage 0 of Figure 7-19: Params ➤ Geometry ➤ Mesh (0a), Mesh ➤ Triangulation ➤ Quad Remesh (0b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (0c), **Surface ➤ SubD ➤ SubD from Mesh (0d)**, Surface ➤ Primitive ➤ Bounding Box (0e, *Per Object (default)*), Surface ➤ Analysis ➤ Box Properties (0f), and Transform ➤ Affine ➤ Scale (0g).

Stage 1: Creating a 3D Voronoi Inside the Mesh's Enlarged Bounding Box and Intersecting It with the Bunny SubD

In stage 1 (Figure 7-19), we use the bunny's enlarged bounding box from stage 0 to create a box-shaped 3D Voronoi. We then perform a Solid Intersection between the SubD bunny and the Voronoi cells.

The first step is to create the set of "generating points" for Voronoi 3D (1c), which is carried out by linking the bunny's enlarged bounding box (Scale (0g)) to Populate 3D (1b), as illustrated in Figure 7-20. We adjust the number of points (hence the number of Voronoi cells) via Populate 3D's Count (C) input. The slider linked to the S input controls the seed for Populate 3D's pseudo randomness—recall that the same seed value always generates the same sequence. Therefore, by changing the seed, we can change the distribution and look of the Voronoi cells.

Next, we create the 3D Voronoi diagram with the bunny's enlarged bounding box as its boundary, by linking Scale (0g) to the B input of Voronoi 3D (1c), and Populate 3D (1b) to the P input.

Finally, we perform a Solid Intersection (1d) between the bunny SubD (0d) and Voronoi 3D (1c) to trim it into the bunny shape, as visualized in Figure 7-20.

Note Where to find the nodes in stage 1 of Figure 7-19: Vector ➤ Grid ➤ Populate 3D (1b), Mesh ➤ Triangulation ➤ Voronoi 3D (1c), and Intersect ➤ Shape ➤ Solid Intersection (1d).

Stage 2: Displacing the Voronoi Cells Away from the Box's Center and Trimming the Cells with the Bunny SubD

In stage 2 (Figure 7-19), we displace the Voronoi cells radially outward from the center of their enclosing box, using the same process as the exploded cube in the previous section. The only change here is we perform a Solid Intersection (2g) at the end between the bunny SubD (0d) and the displaced Voronoi cells (2f), trimming the collective of cells into a bunny shape, as shown in Figure 7-21.

You can create cracked and exploded versions of potentially any closed surface or mesh using the same approach. Note that the surface or mesh must be closed (without holes), since we rely on Solid Intersection with the surface (or mesh converted to SubD) to trim the 3D Voronoi into the desired shape.

Note Where to find the nodes in stage 2 of Figure 7-19: Surface ➤ Analysis ➤ Box Properties (2a), Surface ➤ Primitive ➤ Bounding Box (2b, *Per Object (default)*), Surface ➤ Analysis ➤ Box Properties (2c), Vector ➤ Vector ➤ Vector 2Pt (2d), Maths ➤ Operators ➤ Multiplication (2e), Transform ➤ Euclidean ➤ Move (2f), and Intersect ➤ Shape ➤ Solid Intersection (2g).

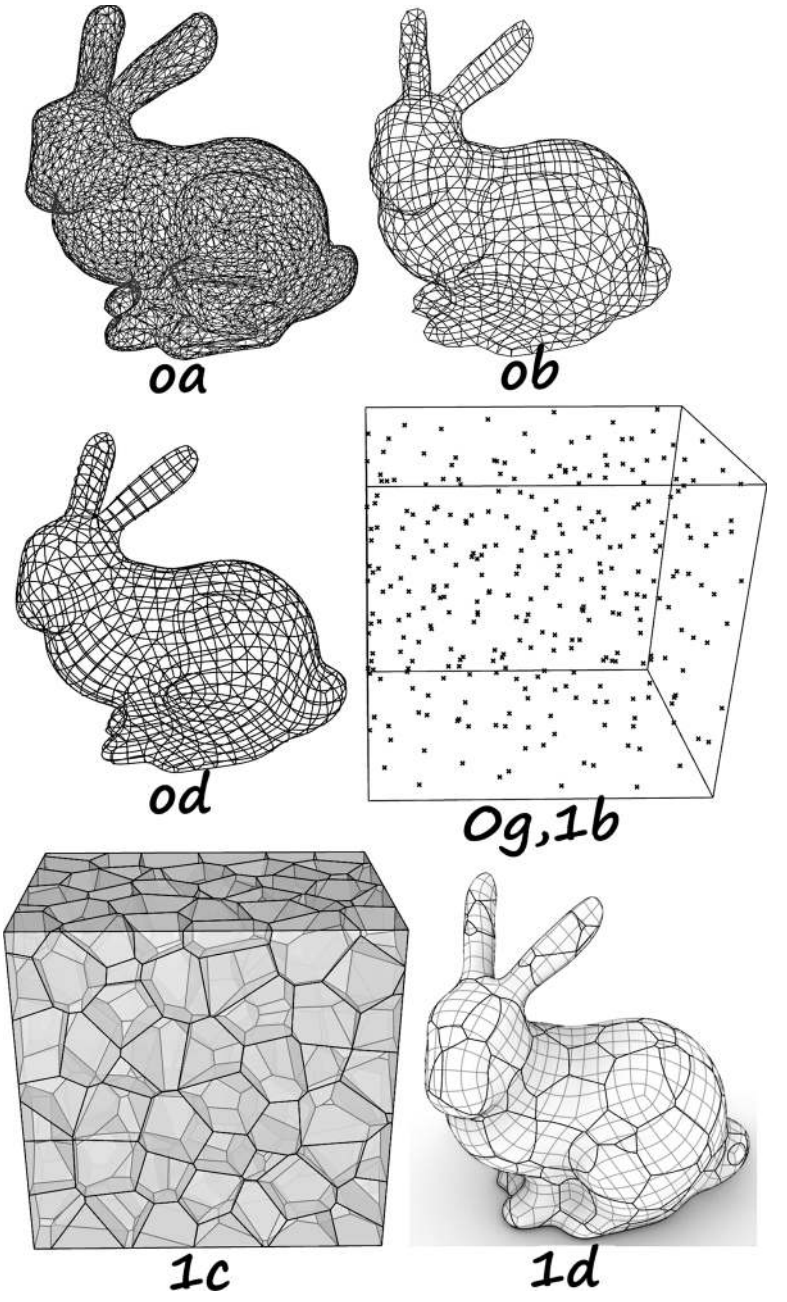


Figure 7-20. *Visual outputs of nodes from Figure 7-19. "0a, 0b, and 0d" show the bunny mesh in its original state, simplified by Quad Remesh, and converted to SubD. "0g,1b" shows points scattered by Populate 3D inside the bunny's enlarged bounding box. "1c" is the 3D Voronoi created inside the enlarged bounding box. "1d" is the cracked bunny created by performing Solid Intersection between the bunny SubD and 1c*

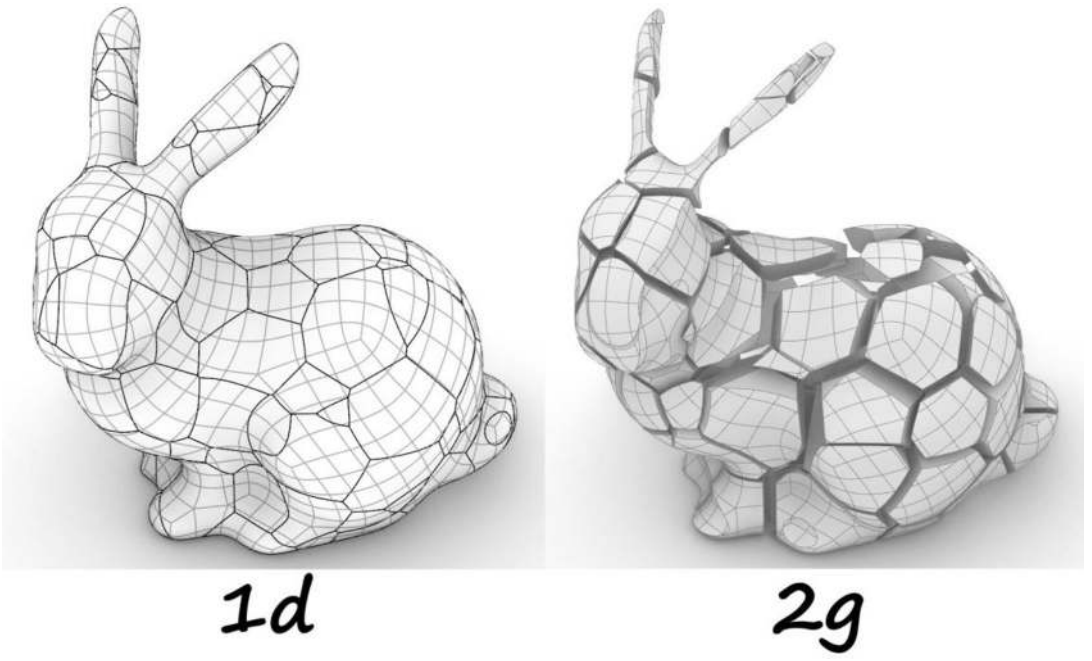


Figure 7-21. *The cracked bunny (left) and exploded bunny (right) output by the node graph in Figure 7-19. The cracked bunny is the Solid Intersection between the bunny SubD and the 3D Voronoi inside the bunny's enlarged bounding box. The exploded bunny is the Solid Intersection between the bunny SubD and the displaced 3D Voronoi cells*

Summary

In this chapter, you worked through seven themed workshops—the first five on wireframes and the last two on cracked and exploded models.

The first workshop showed you how to utilize the Skewed Quads paneling node under the LunchBox plugin to create a star dome. Skewed Quads produces an all-over tiling of parallelograms on a given surface. Due to the way a hemisphere tapers toward the top, the parallelograms degenerate into slanted triangles, which come together as a star. The tiles are joined into a single Brep, whose edges are isolated and MultiPiped into a wireframe.

The second workshop showed you how to create a beehive dome by projecting a hexagonal grid onto a hemisphere and MultiPipe the grid edges to create a wireframe.

The third workshop built on the floral tessellation example from Chapter 4 and took it to 3D, by swirling lines around a cone using Maelstrom. The set of swirly lines is overlaid with a mirrored copy to produce a tower that reveals a floral pattern when viewed from the top.

The fourth workshop showed you how to create three types of wireframe sculptures from the same input mesh—quadrilateral, triangular, and hexagonal. The Quad Remesh node is used to reduce polygon count and convert the mesh's faces to quads, before the mesh edges are MultiPiped into the quad wireframe structure. TriRemesh converts the mesh's faces to near equilateral triangles to enable the creation of the triangular wireframe. The hexagonal wireframe is based on the *dual* of the mesh, which is composed of edges linking centers of the mesh's neighboring faces.

The fifth workshop showed you how to create cracked wireframe structures occupying the volumes of closed Breps like cubes and spheres. We start with a box, scatter points inside the box with Populate 3D, and from which create a 3D Voronoi. Different shapes are achieved by performing Brep | Brep intersections between a closed Brep and the 3D Voronoi. The intersection curves are then MultiPiped to form the wireframe.

The sixth workshop took you through the creation of exploded cubes. Starting with the same process as the cracked cube wireframe, we displace 3D Voronoi cells radially outward from their enclosing box's center to create an explosion effect. An exploded sphere can be made by performing a Solid Intersection between a sphere Brep and the displaced cells.

The seventh workshop showed you how to generalize the explosion effect to arbitrary input meshes, using the Stanford bunny as an example. You first simplify the mesh with Quad Remesh and convert it to a SubD so it is interoperable with the 3D Voronoi cells (which are Breps). A copy of the mesh's inflated bounding box is used as the boundary for the 3D Voronoi, to ensure that the subsequent Solid Intersection between the bunny SubD and the displaced Voronoi cells will retain the complete bunny shape.

References

Wikipedia. "Stanford bunny." *Wikipedia*, 29, August 2025, CC BY-SA 4.0, https://en.wikipedia.org/wiki/Stanford_bunny.

Weisstein, Eric W. "Voronoi Diagram." *From MathWorld—A Wolfram Resource,* 11, Feb 2026, https://mathworld.wolfram.com/VoronoiDiagram.html

PART II

Applications

CHAPTER 8

Weaving

In this chapter, we'll explore two categories of weaving—basket weaves and triaxial weaves. A *basket weave* is formed by arranging threads along two axes repeatedly over and under each other in an alternating pattern, as shown in Figure 8-1. We'll start by implementing a node graph to generate a basic basket weave based on a rectangular grid. We'll then extend the concept to produce a basket weave based on an input mesh, using the Stanford bunny as an example. After that, we'll look at how to create three variations of basket weaves—double, triple, and twisted double weaves. In the last part of the chapter, we'll discuss *triaxial weaves*, which use threads along three axes at 60 degrees in an over-under alternating pattern (Figure 8-1). Similar to our approach with basket weaves, we'll start simple by weaving based on a triangular grid and take the concept further by weaving an input mesh.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_8

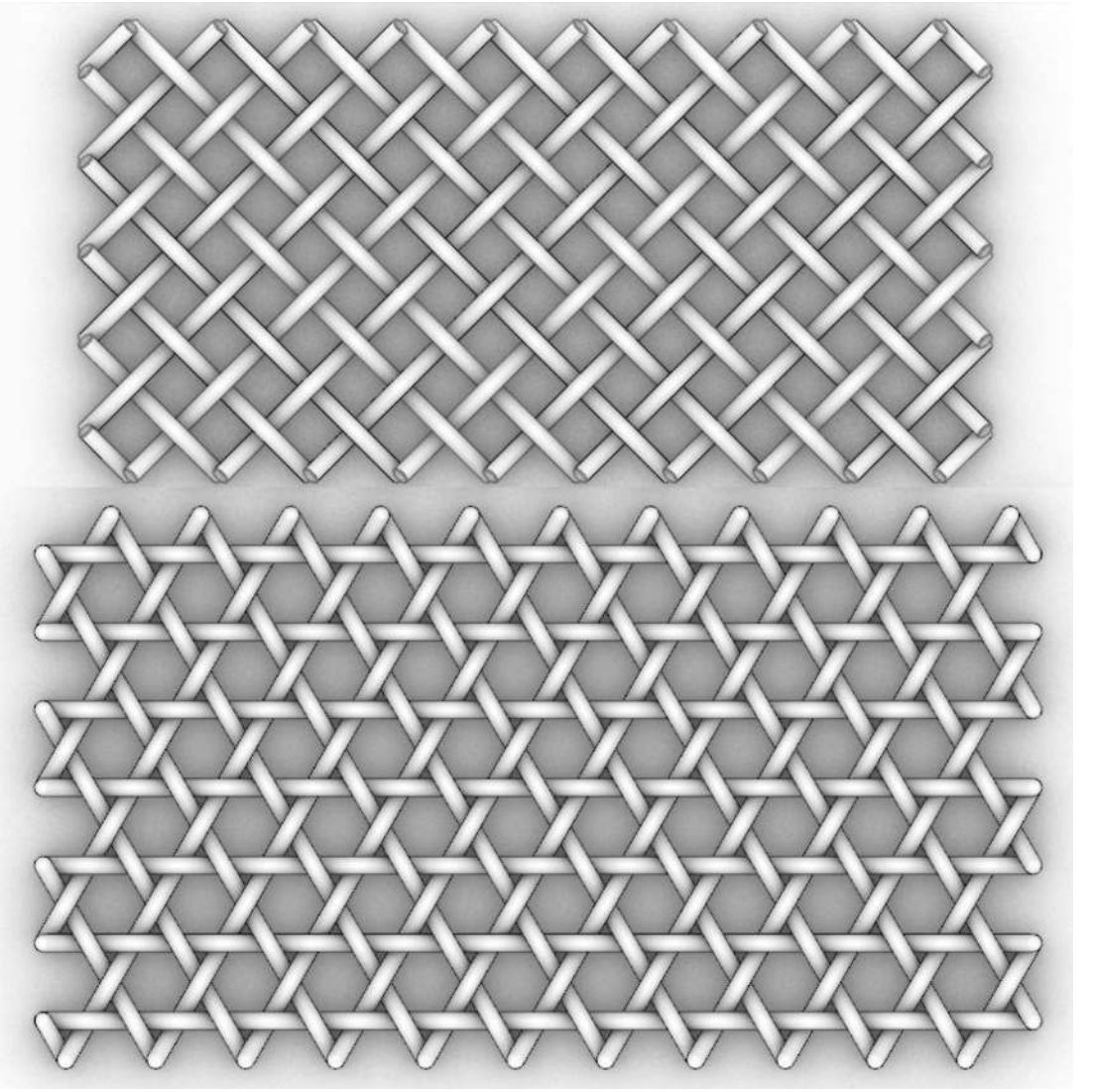


Figure 8-1. *Examples of a basket weave (top), which uses threads along two axes in an over-under pattern, and triaxial weave (bottom), which uses three axes*

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper. Download the files and navigate to the

/Ch8 folder, where you'll find *.gh files named after figures in which they appear along with a short description. For example, /Ch8/Figure8-1_to_8-8_basket_weave_from_rec_grids.gh. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, build your own node graphs by following the steps described in the text, or start with the downloaded *.gh files and modify their parameters to experiment.

Installing the Weaverbird Plugin

Some examples in this chapter require the free GH plugin Weaverbird developed by Giulio Piacentino. These examples will only work if you install the plugin. To install Weaverbird, go to https://www.giuliopiacentino.com/weaverbird/ and locate the instruction for your Rhino version. For Rhino 8 and 9, the process is simple and only requires you to download and open a *.gh file in Grasshopper—the Package Manager will handle the rest.

Attribution to the Stanford Bunny

The Grasshopper files listed below are internalized with the Stanford bunny mesh, a computer graphics test model developed by Greg Turk and Marc Levoy in 1994 at Stanford University (Wikipedia, 2025). The version included is modified by the author to have a closed bottom.

- /Ch8/Figure8-10_basket_weave_bunny.gh
- /Ch8/Figure8-23_to_8-25_triaxial_bunny_weave.gh

Basket Weave from a Rectangular Grid

We'll start with the most basic weave structure—the basket weave, where the warp (vertical) and weft (horizontal) threads go over and under one another in an alternating pattern (over, under, over, under, ...). As one of the most fundamental weaving patterns, basket weaves are commonly found in architecture and textile designs.

Figure 8-2 shows an overview of the node graph for generating a basket weave based on a rectangular grid. The graph encompasses four stages. Stage 1 creates a rectangular grid whose dimensions are parametrically controlled. Stage 2 derives the midpoints

of the grid cell edges. Stage 3 creates a second set of points based on the cell edge midpoints but offset them along +Z to float above the first set. Stage 4 interconnects the two sets of edge midpoints into a basket weave structure and uses MultiPipe to give it volume.

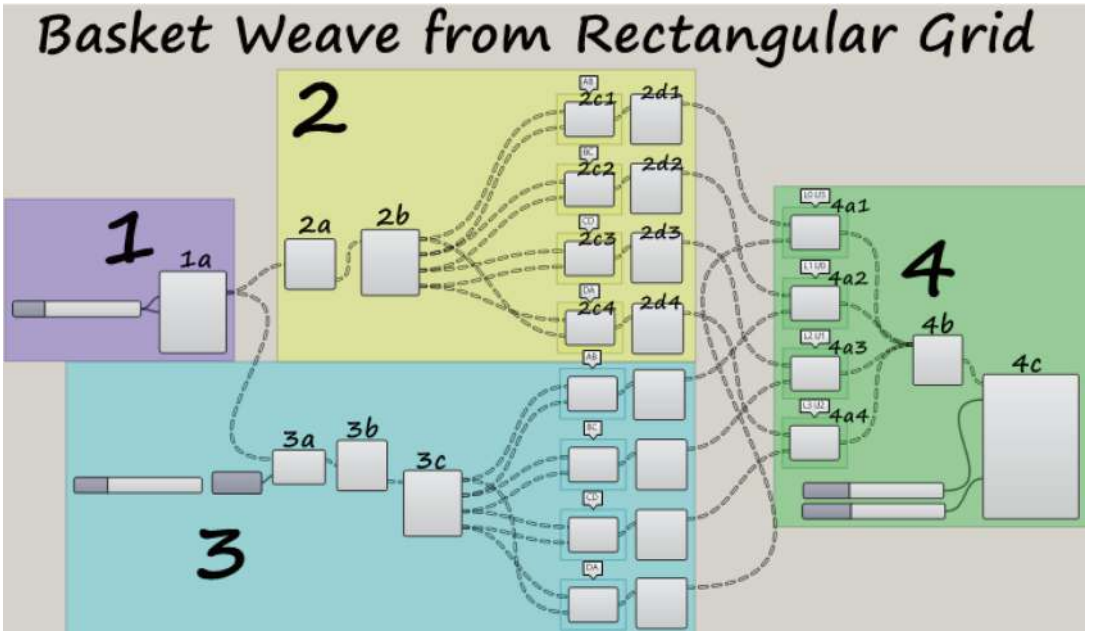


Figure 8-2. *Node graph to create a basket weave structure from a rectangular grid. Stage 1 creates a rectangular grid. Stage 2 finds the midpoints of grid cell edges. Stage 3 creates a copy of the cell edge midpoints that float above the first set. Stage 4 interconnects the two sets of points to create a basket weave and MultiPipes the structure to give it volume*

Stage 1: Create a 2D Rectangular Grid As a Template for the Weave

Our goal in stage 1 (Figure 8-3) is to create a rectangular grid as a template for the weave. We can create a grid with square cells using Rectangular (RecGrid) (1a), with a single Number Slider Shift-dragged to both Sx and Sy to make the width and height of the grid cells the same. We leave the other parameters of RecGrid (1a) at default so the grid will be ten cells wide (Ex) and five cells tall (Ey) and sit on the world XY plane (P).

Note Where to find the nodes in stage 1 of Figure 8-3: Vector ➤ Grid ➤ Rectangular (RecGrid) (1a).

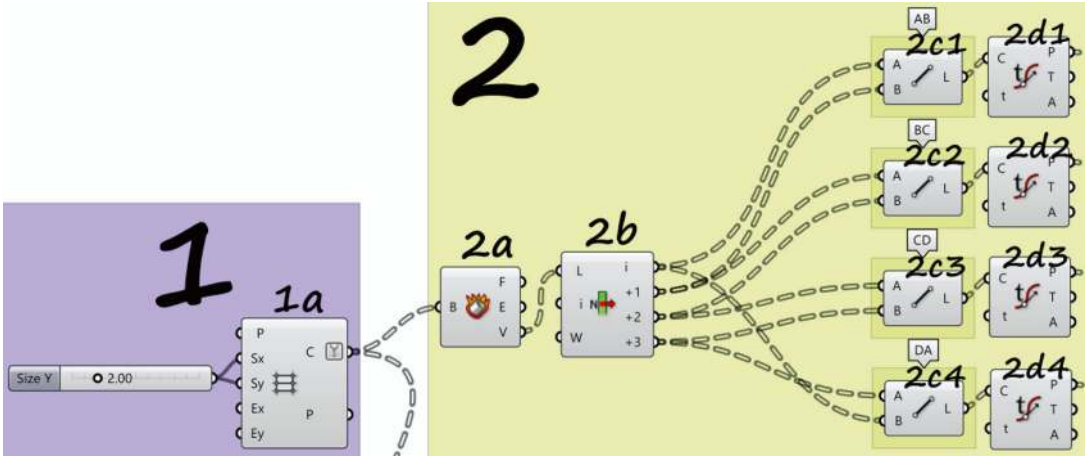


Figure 8-3. *Stages 1 and 2 of the node graph that creates a basket weave structure from a rectangular grid (Figure 8-2). Stage 1 creates a rectangular grid, while stage 2 finds the midpoints of grid cell edges*

Stage 2: Finding Midpoints of the Grid Cell Edges

In stage 2 (Figure 8-3), we'll find the midpoints of the cell edges in the grid from stage 1.

We start by right-clicking RecGrid's C output ➤ Simplify to remove unnecessary branching in the data tree encoding the grid cells. We then drag RecGrid's C output to Deconstruct Brep (2a) to extract the cell vertices in a data tree, with one branch per cell. Since the cells are square, each branch is a list of four vertices. Therefore, we can link the data tree to List Item (2b) with three added outputs (for a total of four outputs) to access each of the four vertices individually. Recall that you can zoom in on List Item until the "+" sign begins to appear, and click "+" three times to add three additional outputs, which are named +1, +2, and +3 by default. The four outputs (i, +1, +2, and +3) give you the cell vertices in counterclockwise order starting at the lower-left corner when viewed from the top.

Next, we'll use four Line nodes (2c1 to 2c4) to create line segments corresponding to the four cell edges in counterclockwise order (bottom, right, top, and left), as shown in Figure 8-4. Since the cell vertices are already ordered the same way, we can simply pair them up in sequence to use as end points for the line segments: (i, +1), (+1, +2), (+2, +3), and (+3, i).

To find the midpoints of the four edges, we link the four Line nodes (2c1 to 2c4) to four Evaluate Curve nodes (2d1 to 2d4), each with t left at the default value of 0.5. Recall that t is a float between 0.0 and 1.0 that controls the amount of interpolation between the two end points of a curve, with 0.0 corresponding to the start and 1.0 to the end point. Therefore, t = 0.5 retrieves the midpoint.

Note Where to find the nodes in stage 2 of Figure 8-3: Surface ➤ Analysis ➤ Deconstruct Brep (2a), Sets ➤ List ➤ List Item (2b), Curve ➤ Primitive ➤ Line (2c1, 2c2, 2c3, and 2c4), and Curve ➤ Analysis ➤ Evaluate Curve (2d1, 2d2, 2d3, and 2d4).

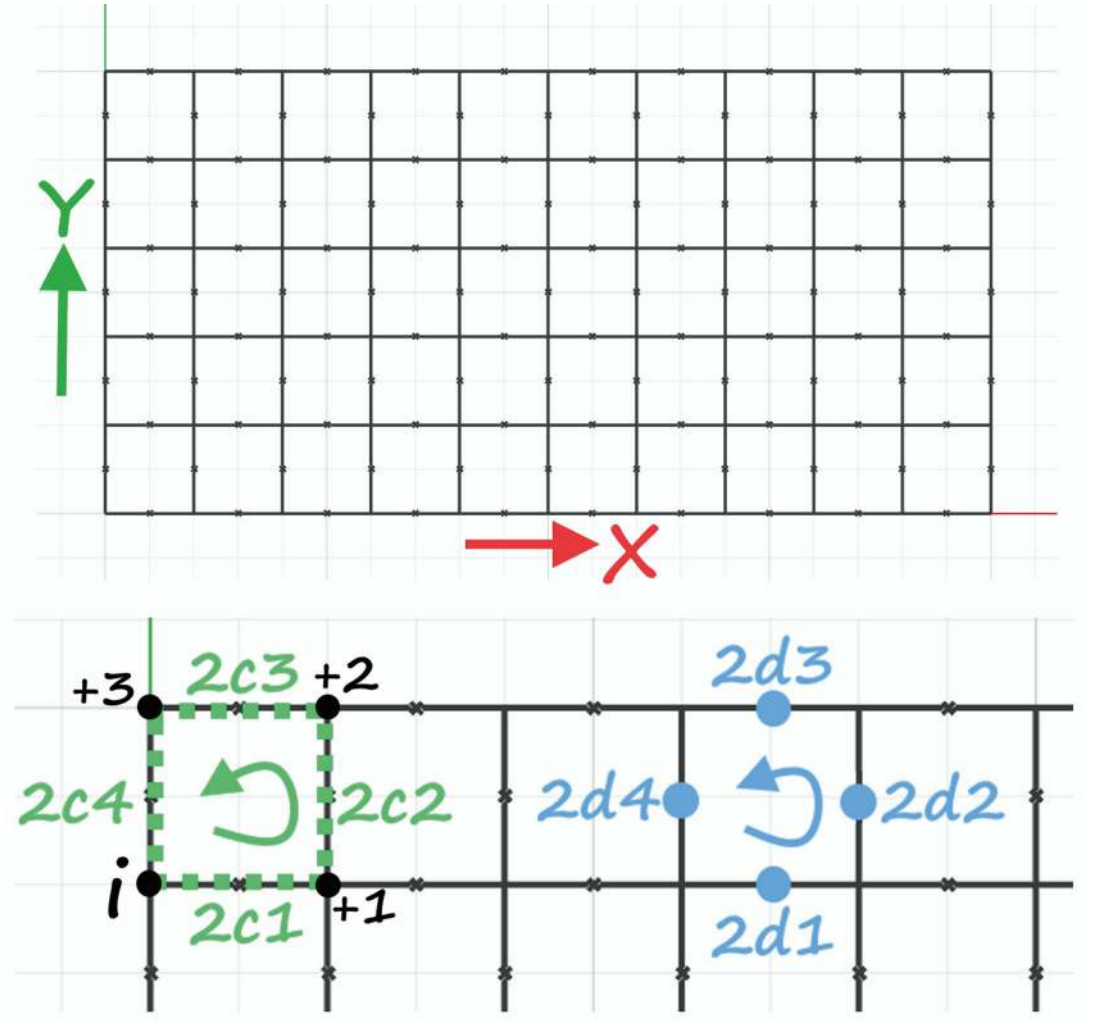


Figure 8-4. *Outputs of Figure 8-3 visualized. Top: rectangular grid (stage 1) and the midpoints of grid cell edges (stage 2). Bottom: grid cell edges (2c1 to 2c4) are ordered counterclockwise when viewed from the top (left). Therefore edge midpoints are also ordered counterclockwise (2d1 to 2d4, right)*

Stage 3: Deriving a Second Set of Cell Edge Midpoints, Elevated Along +Z

In Stage 3 (Figure 8-5), we'll create a second set of grid edge midpoints that are moved along +Z to float above the first set. To do so, we link RecGrid (1a) to Move (3a) with Unit Z as Motion (T) and link a Number Slider to Unit Z's F input to adjust its length, therefore the amount of Z offset. We then reproduce the same node sequence from stage 2 (2a to 2d) to derive the midpoints of the cell edges of the moved grid.

Note Where to find the nodes in stage 3 of Figure 8-5: Transform ➤ Euclidean ➤ Move (3a), Surface ➤ Analysis ➤ Deconstruct Brep (3b), and Sets ➤ List ➤ List Item (3c).

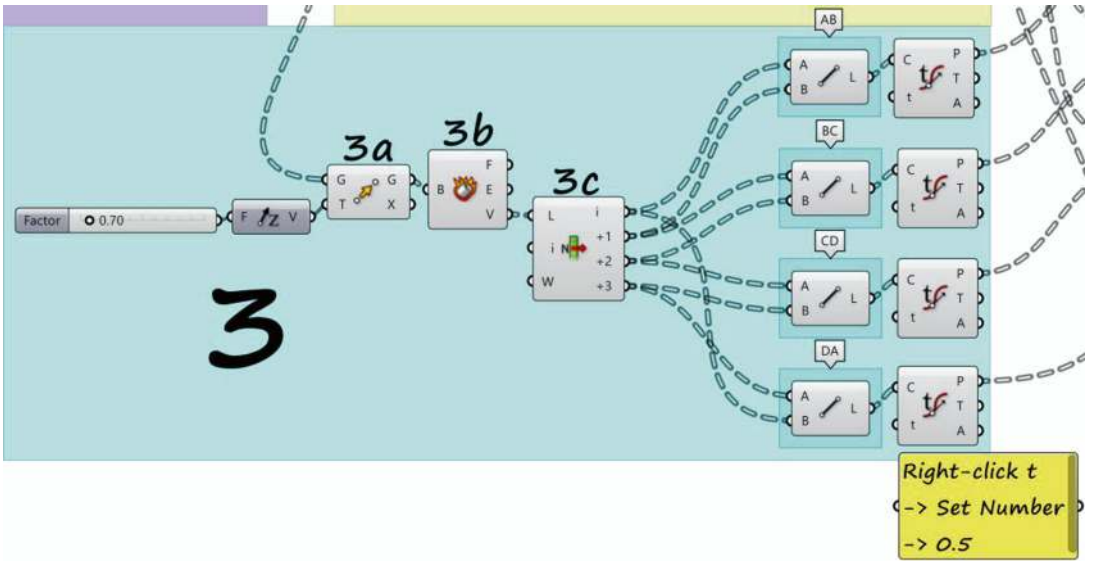


Figure 8-5. *Stage 3 of the node graph that creates a basket weave structure from a rectangular grid (Figure 8-2), where we create a second set of cell edge midpoints that are displaced along +Z to float above the first set*

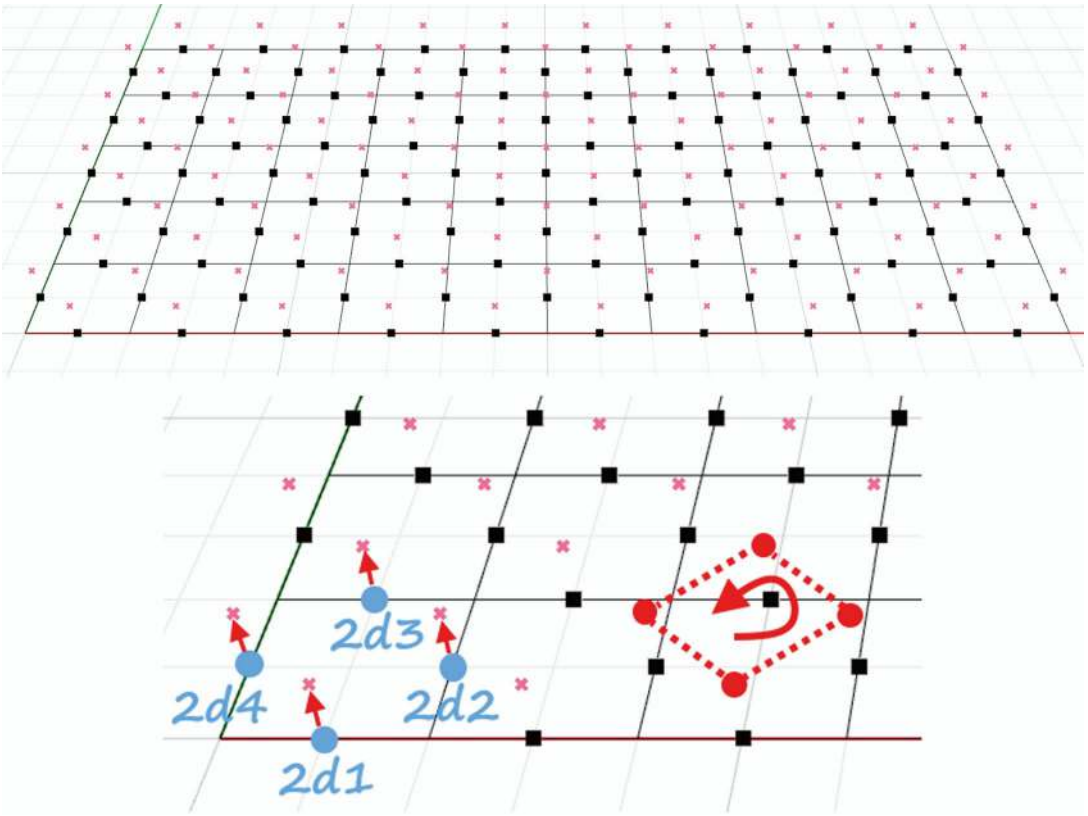


Figure 8-6. *Outputs of stage 3 (Figure 8-5) visualized. Top shows the grid and cell edge midpoints from stage 2, with the second set of points created by stage 3 floating above it. Bottom shows the lower-left corner of the grid zoomed-in, with red arrows showing the offset between the two sets of points (stage 2 on the bottom and stage 3 on top)*

Stage 4: Linking the Two Sets of Cell Edge Midpoints into a Basket Weave and MultiPipe

In Stage 4 (Figure 8-7), we'll interconnect the two sets of cell edge midpoints into a basket weave pattern and use MultiPipe to give it volume.

To create the over-under pattern in a basket weave, we follow the edge midpoints in order and link each point in the first set (at Z = 0) to its predecessor *in the second set* above. Recall from stages 2 and 3, the midpoints 1, 2, 3, and 4 correspond to the bottom, right, top, and left edges when viewed from the top in Figure 8-6. Therefore, we link

1-below (2d1) to 4-above, 2-below (2d2) to 1-above, 3-below (2d3) to 2-above, and 4-below (2d4) to 3-above, using four Line nodes (4a1 to 4a4), as summarized in Table 8-1 and shown in Figure 8-8.

Lastly, we'll smooth the weaved lines by Shift-dragging the four Line nodes (4a1 to 4a4) to Smooth Polyline (4b), followed by MultiPipe (4c) to give the weaves volume. Figure 8-8 shows what the output of MultiPipe (4c) looks like with NodeSize = 0.2 and KinkAngle = 0.10. NodeSize controls the radius of the output pipes, whereas KinkAngle controls the amount of smoothing around pipe joints (the higher the KinkAngle, the smoother the joints, but the pipe crossings also overlap more in the weave structure).

Table 8-1. *Summary of how the two sets of edge midpoints are linked in Figures 8-7 and 8-8*

Point Below (Stage 2)	Point Above (Stage 3, Figure 8-5)	Which Cell Edge?	Line Node (Stage 4)
2d1	4	Bottom	4a1
2d2	1	Right	4a2
2d3	2	Top	4a3
2d4	3	Left	4a4

Note Where to find the nodes in stage 4 of Figure 8-7: Curve ➤ Primitive ➤ Line (4a1, 4a2, 4a3, and 4a4), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

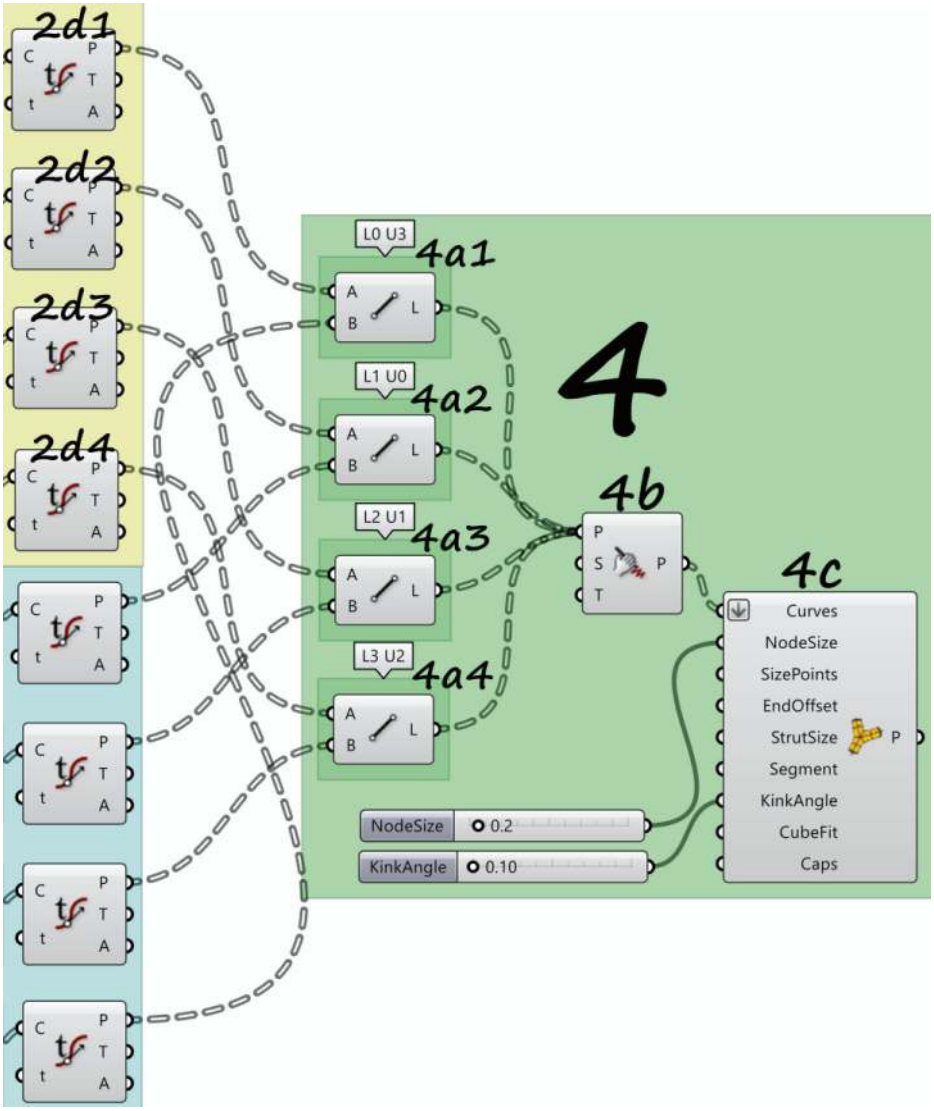


Figure 8-7. *Stage 4 of the node graph that creates a basket weave structure from a rectangular grid (Figure 8-2), where we link the two sets of cell edge midpoints to form a basket weave structure and pipe it to give it volume*

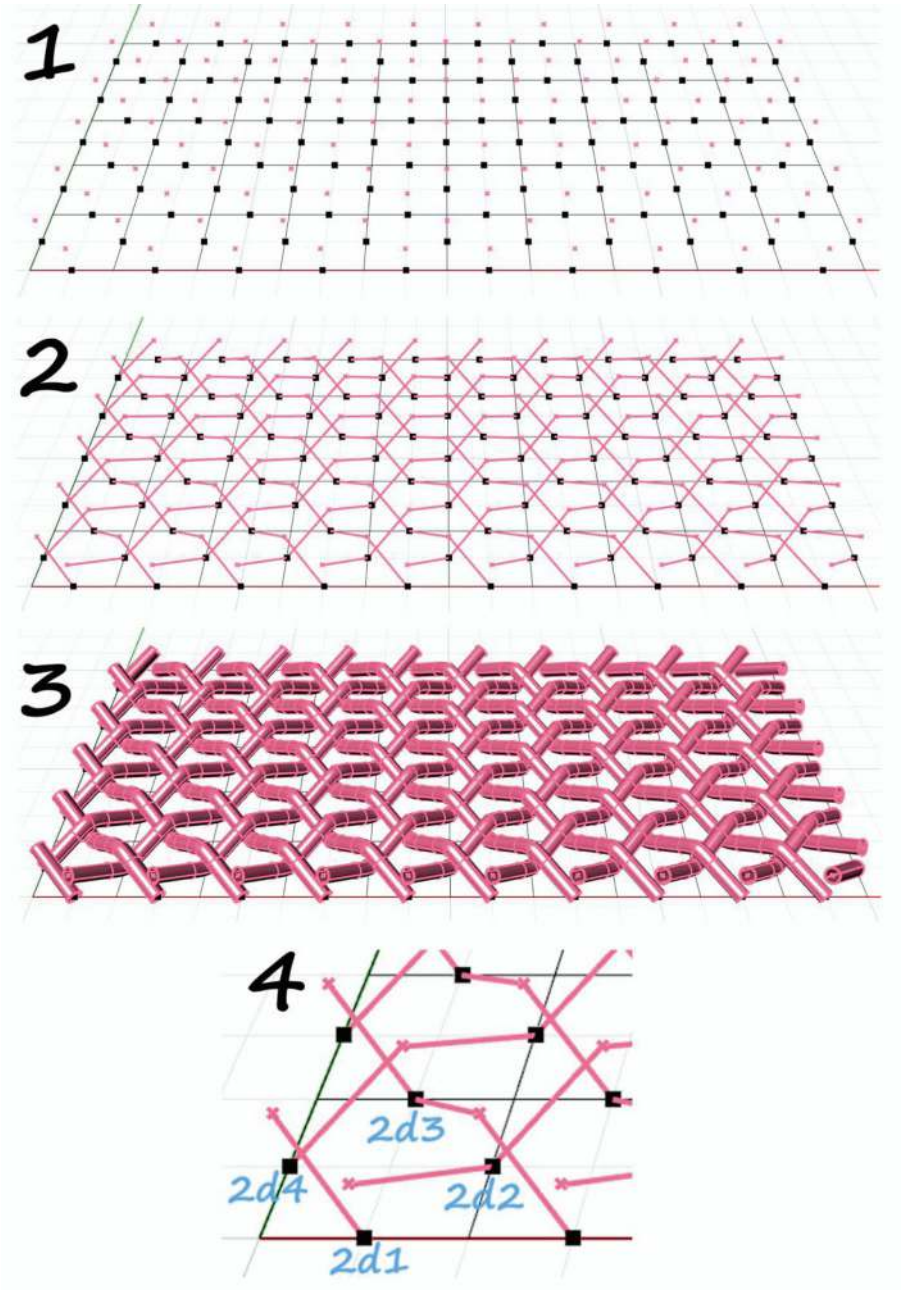


Figure 8-8. *Forming the basket weave structure. (1) The two sets of cell edge midpoints from stages 2 (below) and 3 (above). (2) The points linked to form the weave structure in stage 4. (3) Weave MultiPiped to create volume. (4) The weave structure zoomed in to show node ordering*

Using Python to Form the Basket Weave Structure

In this section, we'll explore an alternative approach for forming the weave structure. As shown in Figure 8-9, we'll use a Python 3 Script node to create the weave's underlying line segments and linking them, instead of multiple sets of Line and Evaluate Curve nodes as we did in Figures 8-3, 8-5, and 8-7.

In Figure 8-9, stage 1 is unchanged and identical to Figure 8-3. Stage 2 retains one node—Deconstruct Brep (2a); stage 3 retains the portion up to Deconstruct Brep (3b); and stage 4 retains the portion after the Line nodes. All the Line and Evaluate Curve nodes in stages 2, 3, and 4 are replaced by the Python 3 Script node (4a). The Access and Type Hints for the Python 3 Script node (4a) are summarized in Table 8-2, while its code is shown in Listing 8-1.

Note Recall that for a Python 3 Script node, you right-click an input to change its name and set its Access and Type Hints. You also right-click an output to set its name and Type Hints. There is no Access Hint for output.

Where to find the nodes in Figure 8-9: Vector ➤ Grid ➤ Rectangular (RecGrid) (1a), Surface ➤ Analysis ➤ Deconstruct Brep (2a and 3b), Transform ➤ Euclidean ➤ Move (3a), Maths ➤ Script ➤ Python 3 Script (4a), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

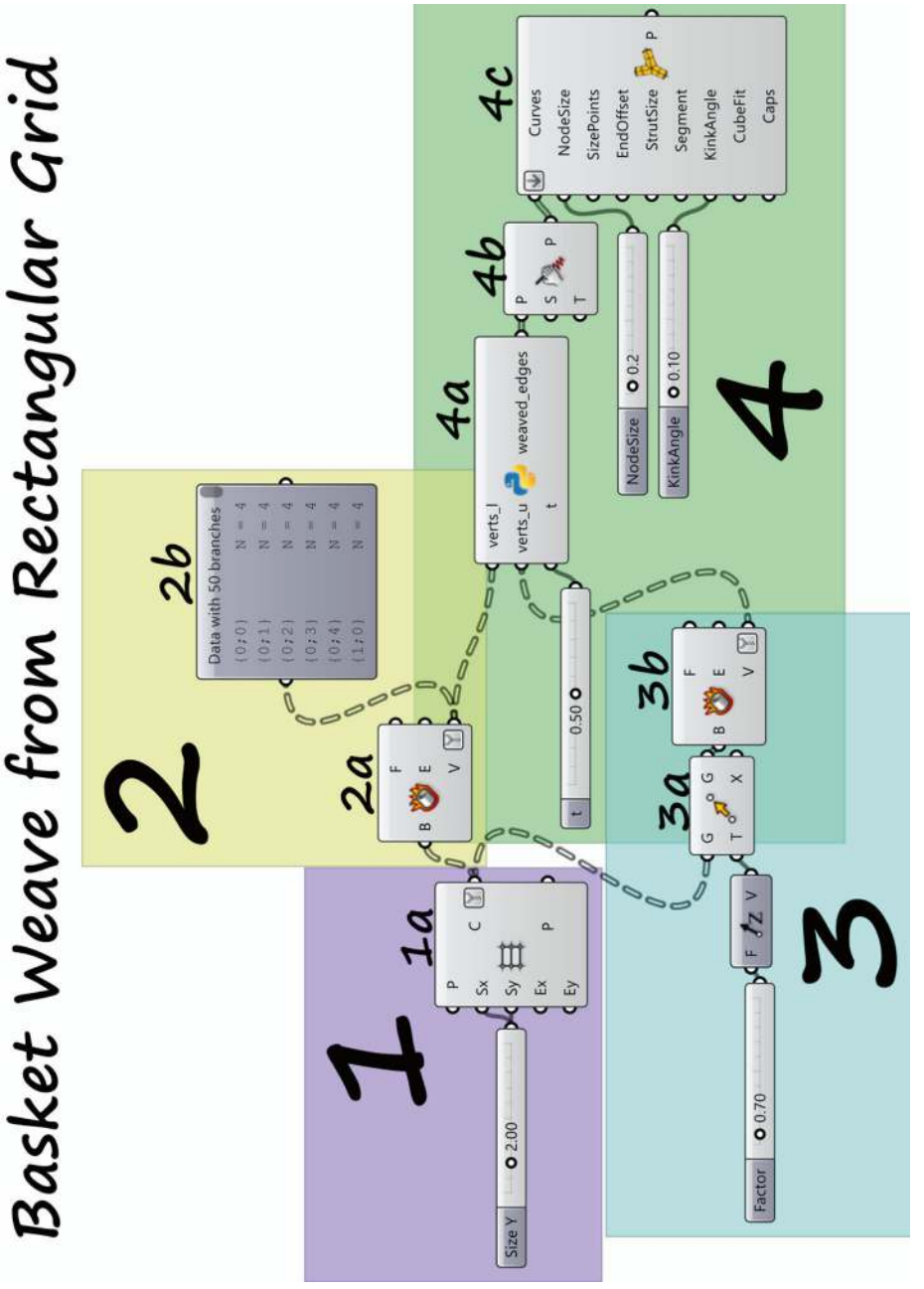


Figure 8-9. *Using a Python 3 Script node (4a) to replace the nodes for linking cell edge midpoints for forming the basket weave structure*

Table 8-2. *Access and Type Hints for the Python 3 Script node in Figure 8-9*

Name	Input or Output	Access	Type Hints	Purpose
verts_l	input	Tree Access	Point3d	Grid cell verts
verts_u	input	Tree Access	Point3d	Grid cell verts moved along +Z to float above the original
t	input	Item Access	float	Amount to interpolate between end points of a grid cell edge (by default 0.5, to retrieve the midpoint)
weaved_edges	output	N/A	Line	List of interconnected line segments between the two sets of cell edge midpoints, a.k.a. the basket weave structure

Listing 8-1. Code inside the Python 3 Script node (4a) in Figure 8-9 (see `/Ch8/Figure8-9_basket_weave_from_rec_grids_python3.gh`)

```
import ghpythonlib.components as ghc

paths_l, length_l, count_l = ghc.TreeStatistics(verts_l)
paths_u, length_u, count_u = ghc.TreeStatistics(verts_u)

def eval_edges(path, tree, t):
    b = ghc.TreeBranch(tree, path)
    shifted_b = ghc.ShiftList(list=b, shift=1, wrap=True)
    edges = ghc.Line(b, shifted_b)
    eval_pts, _, _ = ghc.EvaluateCurve(edges, t)
    return eval_pts

weaved_edges = []
for pl, pu in zip(paths_l, paths_u):
    pts_l = eval_edges(pl, verts_l, t)
    pts_u = eval_edges(pu, verts_u, t)
    shifted_pts_u = ghc.ShiftList(list=pts_u, shift=-1, wrap=True)
    weaved_edges.extend(ghc.Line(pts_l, shifted_pts_u))
```

Let's take a closer look at the Python code (Listing 8-1) inside the Python 3 Script node in Figure 8-9. Our overall strategy is to create a Python function `eval_edges` to retrieve the edge points based on t for the given set of edges, which replace the Evaluate Curve nodes in both stage 2 (Figure 8-3) and stage 3 (Figure 8-5) since we can call this function with both sets of grid edges, the original and the set moved along +Z.

We start by importing `ghpythonlib.components` as the shorthand `ghc` with which to call GH nodes as Python functions. We then make two calls to `ghc.TreeStatistics` with `verts_l` (original grid verts, 2a) and `verts_u` (grid verts moved along +Z, 3b) to get their data tree paths, lengths of each branch, and the number of paths in the tree—this will help us iterate through the tree later to link the two sets of grid edge midpoints into the weave structure.

The next block of code defines the `eval_edges` function, which replaces nodes 2c1–2c4 and 2d1–2d4 in Figure 8-3 and their counterparts in Figure 8-5, by forming the grid cell edges as line segments using grid verts and finding the edge midpoints. Recall that one branch corresponds to one grid cell. Therefore, the function takes one branch (or one grid cell's vertices) and calls `ghc.ShiftList` with `wrap=True` to get the list of vertices shifted by 1 (e.g., (1, 2, 3, 4) becomes (2, 3, 4, 1)). It then calls `ghc.Line` to create line segments between the two sets of verts ((1, 2), (2, 3), (3, 4), and (4, 1), which corresponds to the four grid cell edges counterclockwise starting from the bottom—this is equivalent to 2c1 to 2c4 in Figure 8-3 and their counterparts in Figure 8-5). The next line of code calls `ghc.EvaluateCurve`, which for t = 0.5 retrieves the midpoints of the cell edges (this is equivalent to nodes 2d1 to 2d4 in Figure 8-3 and their counterparts in Figure 8-5).

In the final block of code, we iterate through the two grids (the original and the one above it), visiting one grid cell pair at a time, linking the edge midpoints into the over-under structure to replace the four Line nodes (4a1–4a4) in Figure 8-7. `zip(paths_l, paths_u)` takes one item from each list and pair them off—since `paths_l` and `paths_u` are the grid cells in the original and the upper grid, respectively, `zip(paths_l, paths_u)` gives you the pairings (cell_0 in lower grid, cell_0 in upper grid), (cell_1 in lower grid, cell_1 in upper grid), …etc.

On the next two lines, we call `eval_edges` with each cell in the lower grid and its counterpart in the upper grid to get two sets of corresponding edge midpoints. We then call `ghc.ShiftList` with `shift=-1` to shift the set in the upper cell by –1; therefore, for the lower cell points 1, 2, 3, and 4, the corresponding upper cell points become 4, 1, 2, and 3, which matches the pairings (1-lower, 4-upper), (2-lower, 1-upper), …etc. In other

words, each edge point in the lower cell is matched with its *predecessor* in the upper cell. Calling `ghc.Line` with these two lists of points then produce the same output as nodes 4a1 to 4a4 in Figure 8-7.

Basket Weaving an Input Mesh

We'll now generalize the basket weave generator to create a weave based on an input mesh instead of a rectangular grid. Figure 8-10 shows this modification, where stage 1 reduces the poly count and converts the mesh faces to quads. Stages 2 and 3 use Deconstruct Mesh (2a) followed by Deconstruct Face (2b) to look up the vertices framing each mesh face (which is analogous to using Deconstruct Brep to extract grid cell vertices). Weaverbird's Offset Mesh (3a) is used next to create a scaled-up copy of the mesh that is equidistant everywhere to the original, analogous to moving a copy of the grid along +Z to float above.

You can find the node graph in Figure 8-10 in the file `/Ch8/Figure8-10_basket_weave_bunny.gh` in this chapter's downloaded source, where the Mesh node (1a) is already internalized with the Stanford bunny mesh. We use Quad Remesh (1b) to reduce the bunny's polygon count to 500 (specified via Quad Remesh Settings with Tc = 500) and convert its faces to quads.

In stage 2, we extract the vertices framing each mesh face. Since we've converted the mesh to have quad faces, each face has four verts, thus making it analogous to the four corners of the grid cells and allowing us to reuse the portion of the graph for finding edge midpoints. We start by linking Quad Remesh (1b) to Deconstruct Mesh (2a) to get the list of all vertices via the V output—this is just a flat list with verts listed in no particular order. To access verts on a per face basis, we link Deconstruct Mesh's F output to Deconstruct Face (2b) to retrieve the vert indices framing each face and in turn use them as the i input for List Item (2c1 to 2c4) to look up the vertices.

In stage 3, we use Weaverbird's Offset Mesh (3a) to create a scaled-up copy of the quad mesh that is equidistant everywhere to the original, thus creating the same effect as lifting a copy of the rectangular grid above the original. You tell Weaverbird's Offset Mesh how much to scale a mesh in terms of *how far* the scaled mesh should be when overlaid on top of the original. Therefore, a D of positive value (like the 1.5 we used in Figure 8-10) creates a scaled-up mesh, larger than and enclosing the original, whereas a negative D creates a scaled-down mesh that goes inside the original mesh.

After Weaverbird's Offset Mesh, we duplicate the Deconstruct Mesh-Deconstruct Face-List Item sequence from stage 2, so we can extract the four face corners in the offset mesh.

Figure 8-11 shows the bunny after the poly reduction and conversion to quad faces, along with edge midpoints before and after the scaling by Weaverbird's Offset Mesh. You can see the bunny's faces look very much like a rectangular grid projected onto a mesh.

Figure 8-12 shows how the edge midpoints of the quad mesh and the offset mesh are interconnected into an over-under weave structure, much like their rectangular grid counterpart in Figure 8-8.

Note Where to find the nodes in Figure 8-10: Params ➤ Geometry ➤ Mesh (1a), Mesh ➤ Triangulation ➤ Quad Remesh (1b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (1c), Mesh ➤ Analysis ➤ Deconstruct Mesh (2a), Mesh ➤ Analysis ➤ Deconstruct Face (2b), Sets ➤ List ➤ List Item (2c1, 2c2, 2c3, and 2c4), and **Wb ➤ Transform ➤ Weaverbird's Offset Mesh (3a)**.

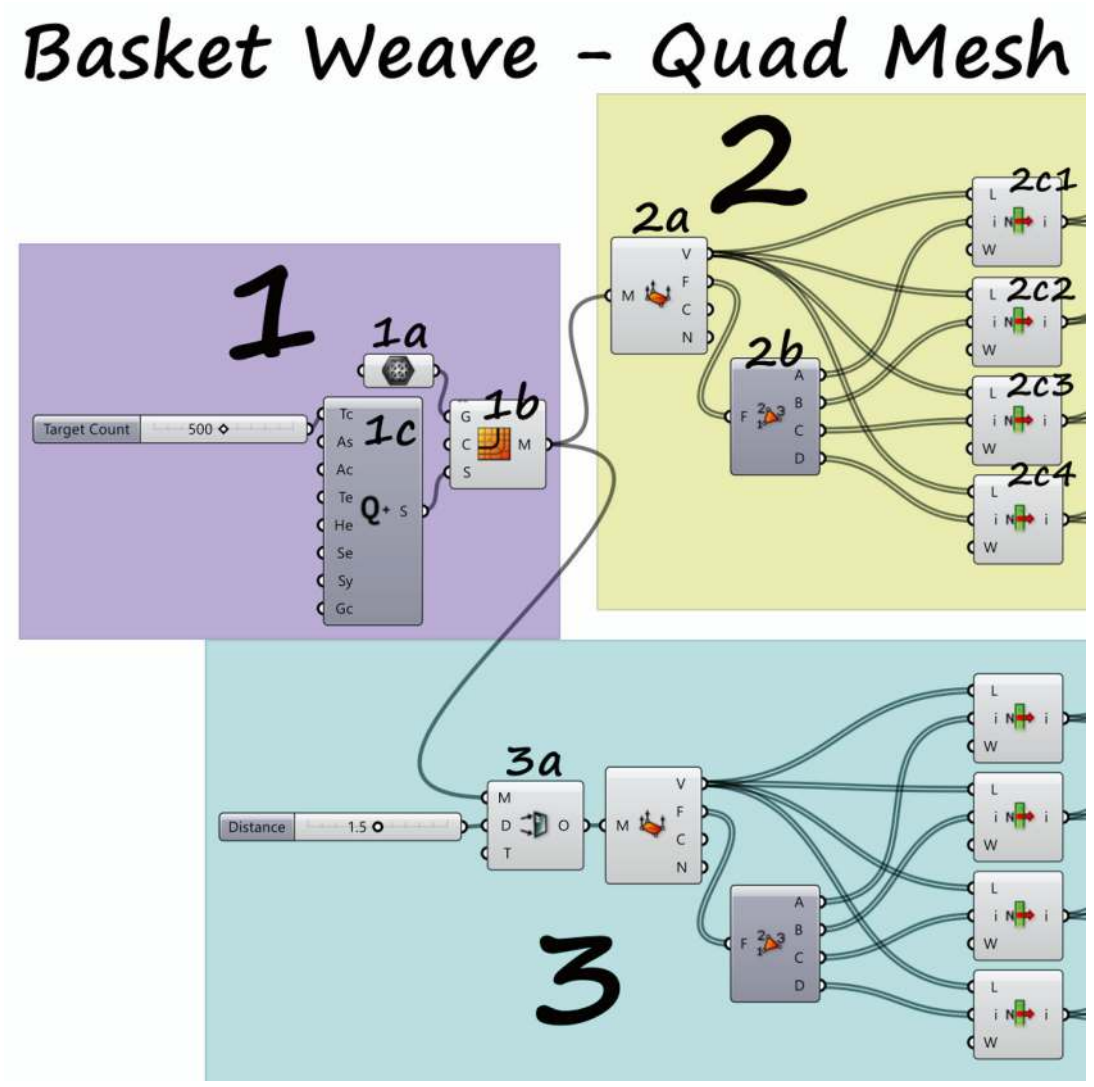


Figure 8-10. *Using a mesh as input instead of a grid to generate a basket weave structure. Stage 1 reduces the poly count and converts the mesh faces to quads. Stage 2 serves a similar function as before and extracts the edge midpoints, except here the edges are the mesh's edges instead of grid cell edges. Stage 3 uses Weaverbird's Offset Mesh (3a) to create a scaled copy of the mesh that is approximately equidistant everywhere to the original as the basis for extracting a second set of edge midpoints that float above the first set*

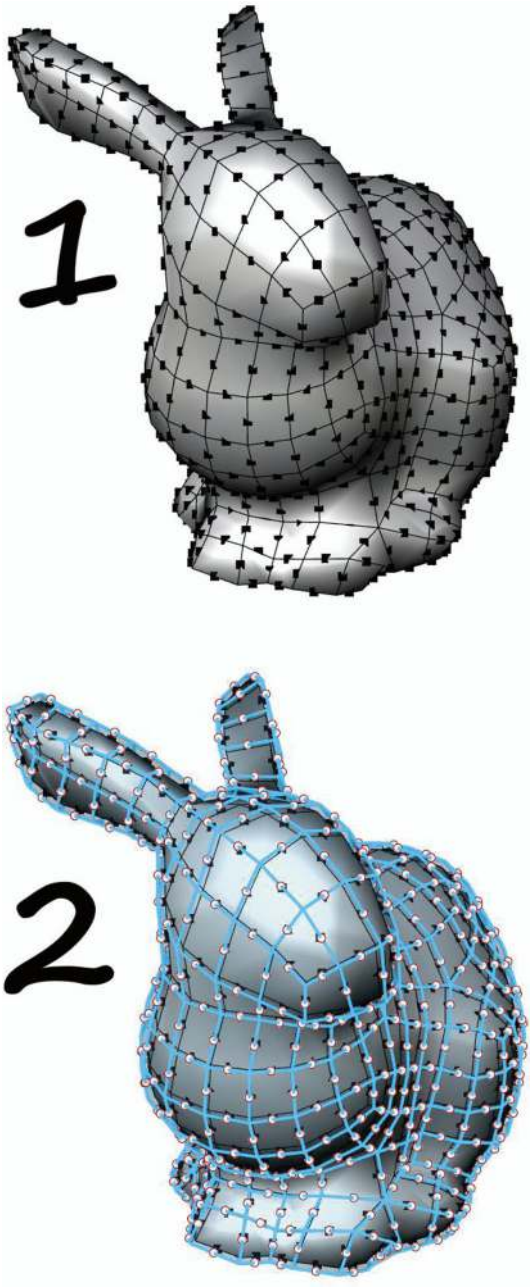


Figure 8-11. *Top: the bunny mesh after poly reduction and Quad Remesh, with its edge midpoints. Bottom: The mesh's scaled copy created by Weaverbird's Offset Mesh overlaid on top of the quad mesh*

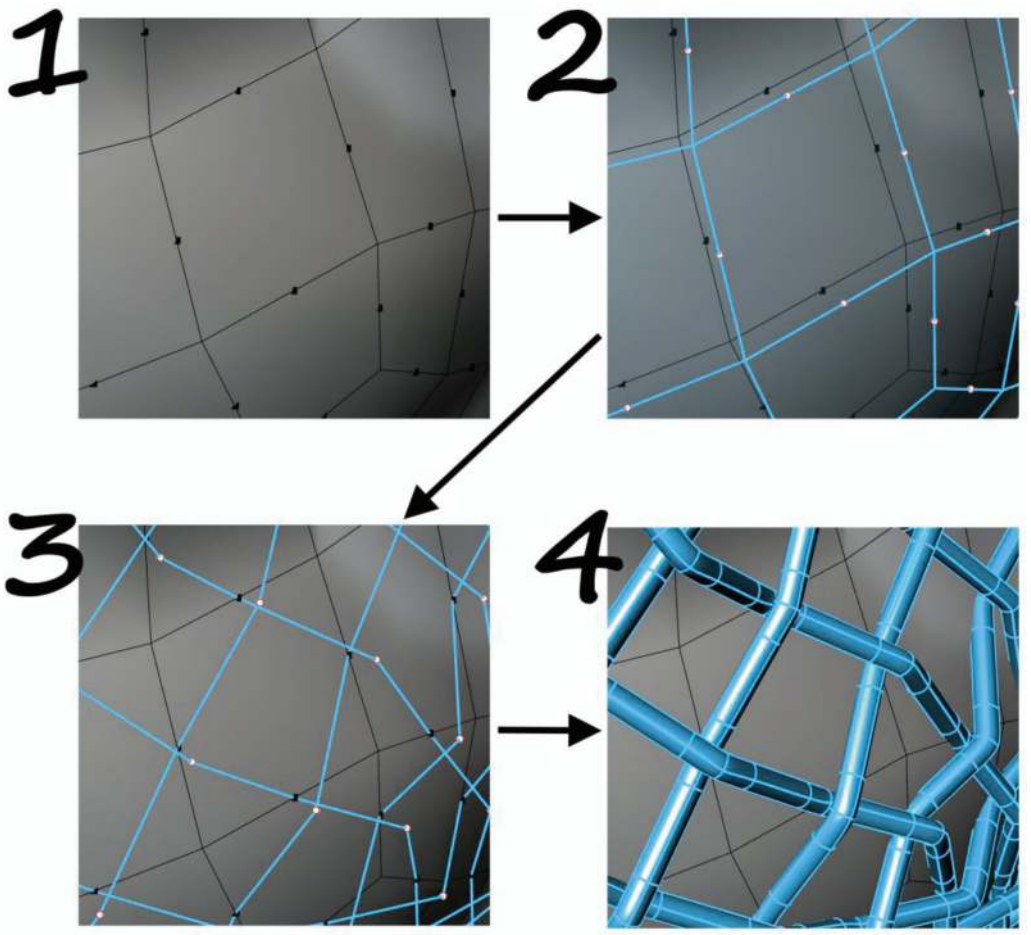


Figure 8-12. *A closer look at how the basket weave structure is formed by linking the edge midpoints between the quad mesh and the offset mesh. (1) The input mesh after poly reduction and Quad Remesh with its edge midpoints. (2) The offset mesh laid over the original with edge midpoints. (3) The two sets of edge midpoints linked to form the basket weave structure. (4) The weave structure MultiPiped to create volume*

Double, Triple, and Twisted Variations of Basket Weave

In this section, we'll explore three variations of basket weave, by doubling the weave threads, tripling the weave threads, and twisting the double weave threads.

Double Weave

Figure 8-13 shows how to modify the Python-3-based basket-weaving graph in Figure 8-9 to generate double weaves (Figure 8-14). The Python 3 Script node (4a) is modified to output the tangent vectors of the lower grid edges (`tangents_l`), which are used to offset `weaved_edges` (also output by 4a) to create the double weave. Recall that `weaved_edges` is created by interconnecting the edge midpoints in the lower grid to their predecessors in the upper grid. The changes to the Python 3 Script node (4a) are summarized in Table 8-3 and Listing 8-2.

Let's take a closer look at the changes to the Python code in Listing 8-2. Recall that `ghc.EvaluateCurve` is called inside the function `eval_edges` to find edge midpoints based on `t`. In addition to edge midpoints, `ghc.EvaluateCurve` also outputs tangent vectors along the edges, which we ignored before as an anonymous variable but change here to a named variable `T` to return from the function.

To find the tangents of all edges, we declare an empty list `tangents_l` just before the `for` loop. Inside the `for` loop body, every time we call `eval_edges`, we add the returned tangents to `tangents_l` by calling `tangents_l.extend`.

Note Where to find the nodes in Figure 8-13: Vector ➤ Grid ➤ Rectangular (RecGrid) (1a), Surface ➤ Analysis ➤ Deconstruct Brep (2a and 3b), Params ➤ Util ➤ Param Viewer (2b), Transform ➤ Euclidean ➤ Move (3a, T1, and T2), Maths ➤ Script ➤ Python 3 Script (4a), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

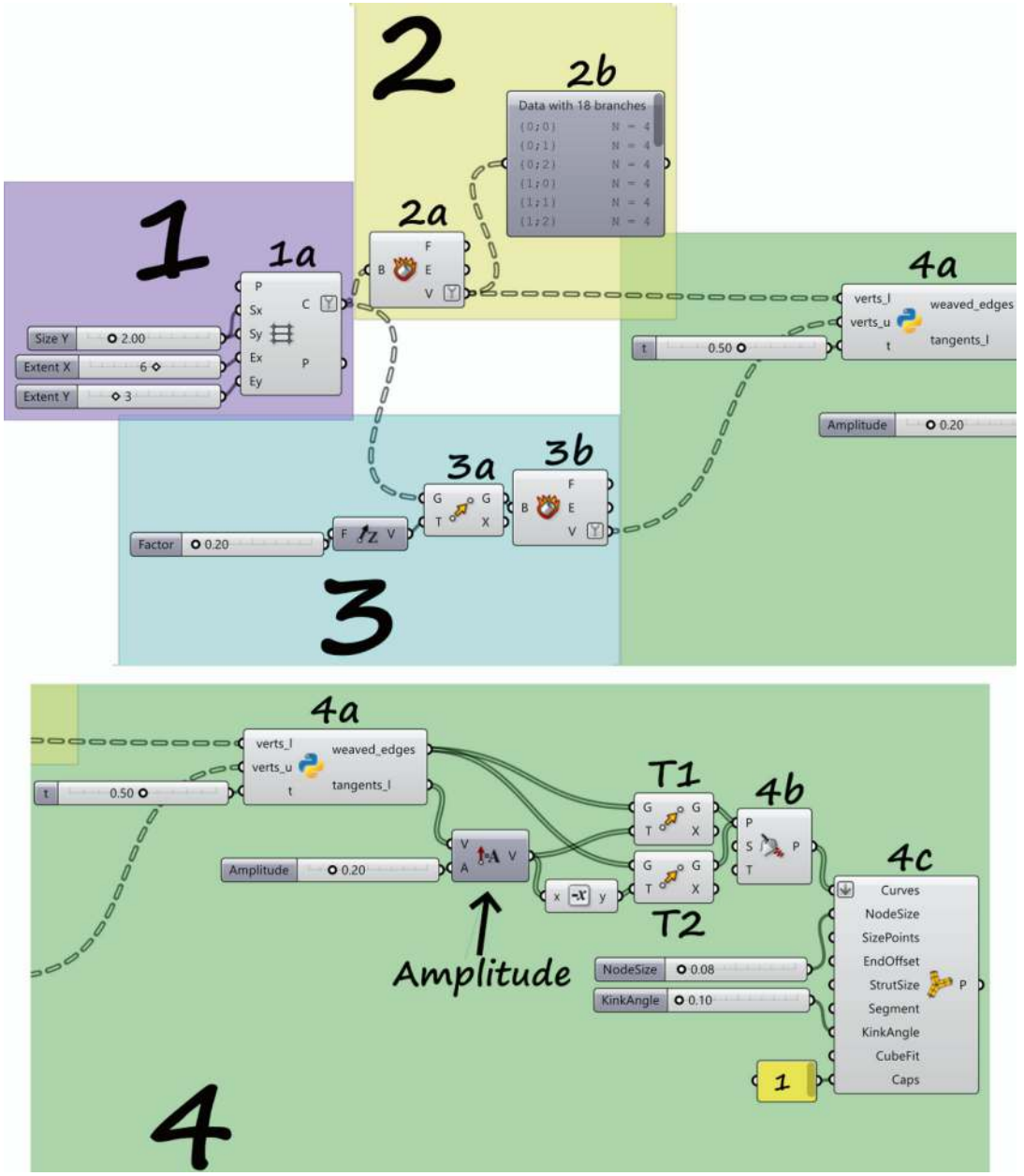


Figure 8-13. *Adapting the Python-3-based basket weave graph (Figure 8-9) to generate a double weave. The Python 3 Script node (4a) is modified to output the tangent vectors of the grid edges, which are in turn used to offset weaved_edges to create the double-weave structure. (weaved_edges are the line segments interconnecting upper and lower grid edge midpoints.)*

Table 8-3. *Access and Type Hints for the Python 3 Script node (4a) in Figure 8-13, with differences from its counterpart in Figure 8-9 shown in bold*

Name	Input or Output	Access	Type Hints	Purpose
verts_l	input	Tree Access	Point3d	Grid cell verts
verts_u	input	Tree Access	Point3d	Grid cell verts moved along +Z to float above the original
t	input	Item Access	float	Amount to interpolate between end points of a grid cell edge (by default 0.5, to retrieve the midpoint)
weaved_edges	output	N/A	Line	List of interconnected line segments between the two sets of cell edge midpoints, a.k.a. the basket weave structure
tangents_l	**output**	**N/A**	**Vector3d**	**Tangent vectors along grid edges (from start to end point) returned by ghc.EvaluateCurve**

Listing 8-2. Code inside the Python 3 Script node (4a) in Figure 8-13, with differences from Listing 8-1 shown in bold (see `/Ch8/Figure8-13_8-14_double_weave.gh`)

```
import ghpythonlib.components as ghc

paths_l, length_l, count_l = ghc.TreeStatistics(verts_l)
paths_u, length_u, count_u = ghc.TreeStatistics(verts_u)

def eval_edges(path, tree, t):
    b = ghc.TreeBranch(tree, path)
    shifted_b = ghc.ShiftList(list=b, shift=1, wrap=True)
    edges = ghc.Line(b, shifted_b)
    P, T, _ = ghc.EvaluateCurve(edges, t)
    return P, T
```

```
weaved_edges = []
tangents_1 = []
for pl, pu in zip(paths_l, paths_u):
    pts_l, T_1 = eval_edges(pl, verts_l, t)
    tangents_1.extend(T_1)
    pts_u, _ = eval_edges(pu, verts_u, t)
    shifted_pts_u = ghc.ShiftList(list=pts_u, ↩
        shift=1, wrap=True)
    weaved_edges.extend(ghc.Line(pts_l, shifted_pts_u))
```

We'll now return to Figure 8-13 and examine the nodes inserted between Python 3 Script (4a) and Smooth Polyline (4b) responsible for offsetting `weaved_edges` to create the double weave. We start by linking `tangents_l` to Vector ➤ Vector ➤ Amplitude so the tangent vectors' lengths can be adjusted with a Number Slider. We then use two Move nodes (T1 and T2) to offset `weaved_edges` in opposite directions, with T1 using the length-adjusted `tangents_l` directly (thus shifting the lines "inward"), and T2 using a negative copy (thus shifting the lines "outward"). T1 and T2 are then both Shift-dragged to Smooth Polyline (4b) to combine and undergo smoothing, followed by MultiPipe (4c) to create volume. A visualization of these steps is shown in Figure 8-14.

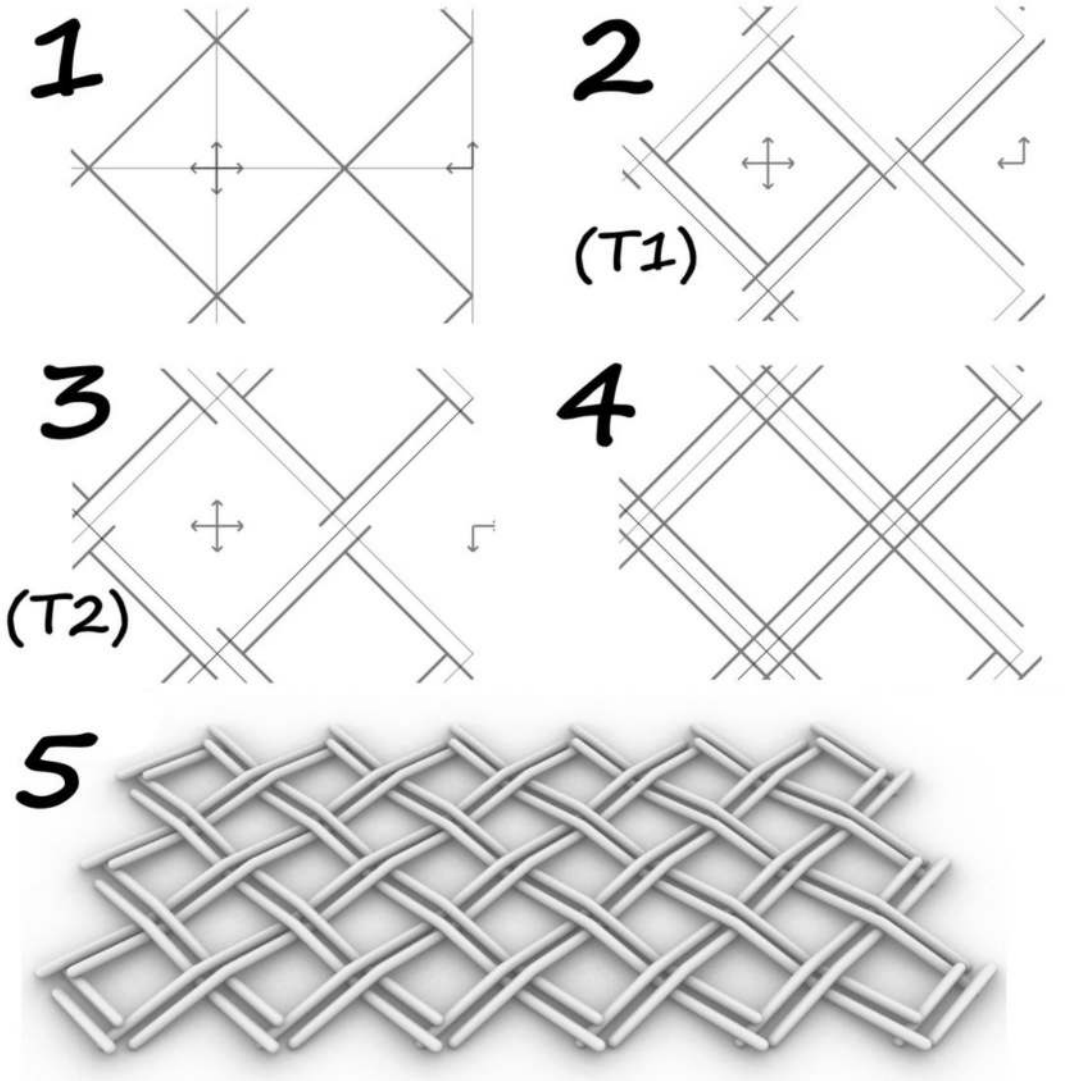


Figure 8-14. *Steps taken by the node graph in Figure 8-13 to generate double basket weaves. (1) shows the rectangular grid with weaved_edges and tangents_l output by Python 3 Script (4a). (2) shows weaved_edges moved along tangents_l (T1). (3) weaved_edges moved along negative of tangents_l (T2). (4) T1 and T2 combined. (5) Final output of Figure 8-13*

Triple Weave

We can make a simple addition to the double-weave node graph in Figure 8-13 to create triple weaves, by adding a wire from the `weaved_edges` output of Python 3 Script (4a) to Smooth Polyline (4b), as shown in Figure 8-15—this adds the original weave to the double weave, thus creating the effect shown in Figure 8-16.

Note Where to find the nodes in Figure 8-15: Maths ➤ Script ➤ Python 3 Script (4a), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

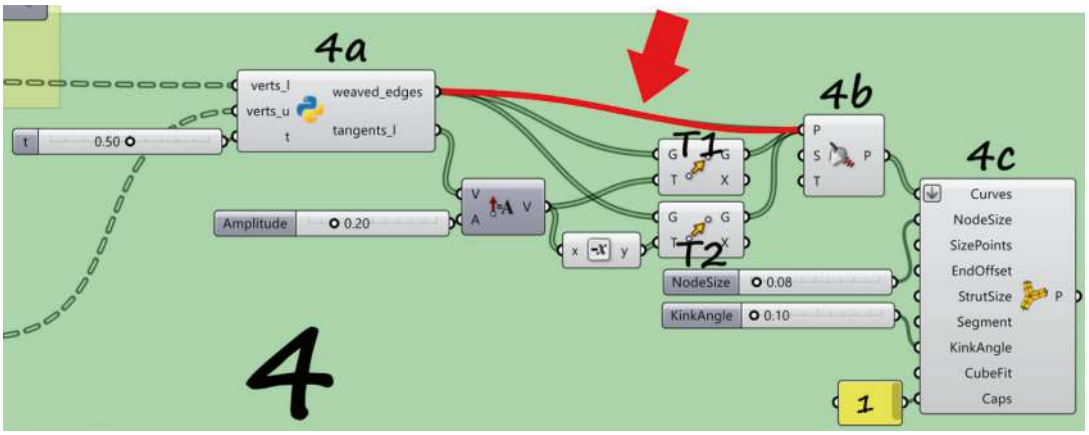


Figure 8-15. *Modifying the double-weave node graph from Figure 8-13 to generate triple weaves. An extra link is added between the weaved_edges output of Python 3 Script (4a) and Smooth Polyline (4b) to create an extra "thread" between the two threads of the double weave*

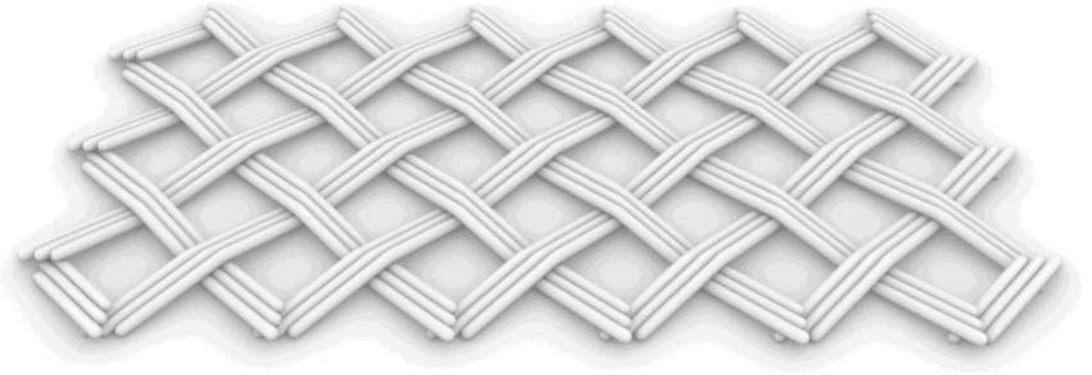


Figure 8-16. *The output of the triple-weave node graph in Figure 8-15. An extra "thread" is added between the double-weave threads*

Twisted Double Weave

Another variation we can create from the Python-3-based basket-weave graph (Figure 8-9) is a double twisted weave. We create this effect by duplicating the Python 3 Script node as shown in Figure 8-17 (4a1, 4a2) with t set to slightly different values (as shown, 0.40 and 0.60, respectively). We then Shift-drag both script nodes to Smooth Polyline (4b) to combine. The slight offsets between the two sets of grid edge points create a twisted effect around one another, as shown in Figure 8-18.

Note Where to find the nodes in Figure 8-17: Maths ➤ Script ➤ Python 3 Script (4a1, 4a2), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

You can click a node to select it, Ctrl-C to copy, and Ctrl-V to paste. You can also find Copy and Paste under Edit in the top menu bar of the GH editor.

Tip You can experiment further by changing the two t values (for 4a1 and 4a2 in Figure 8-17) and see what effects they produce on the weave.

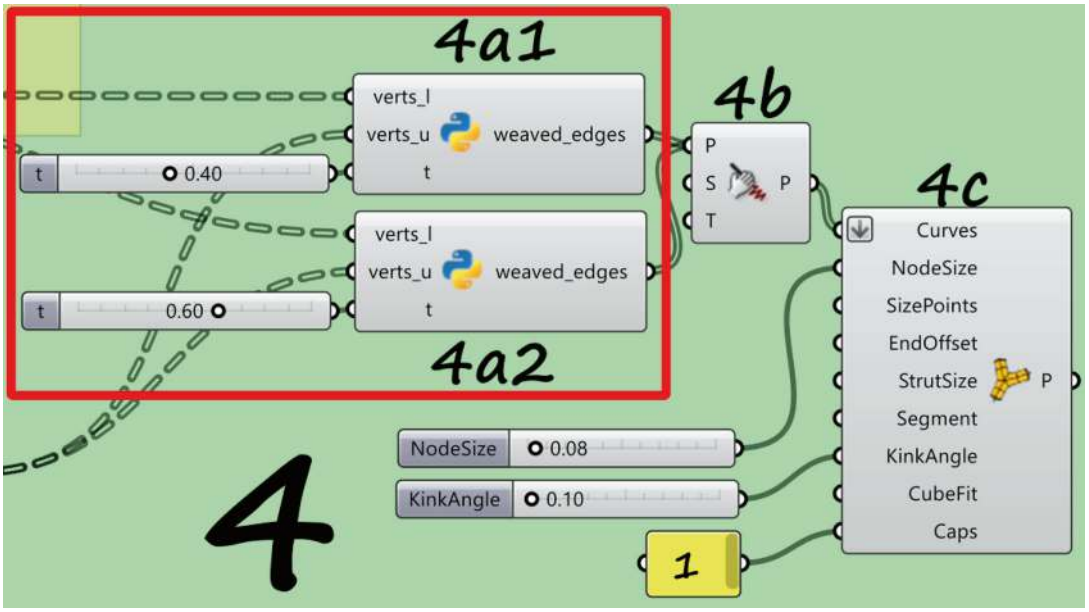


Figure 8-17. *Adapting the Python-3-based basket weave node graph (Figure 8-9) to create a double twisted weave instead. Two copies of the Python 3 Script node are used (4a1 and 4a2), with t set to 0.4 and 0.6, respectively, to create two sets of edge points that are slightly offset from each other*

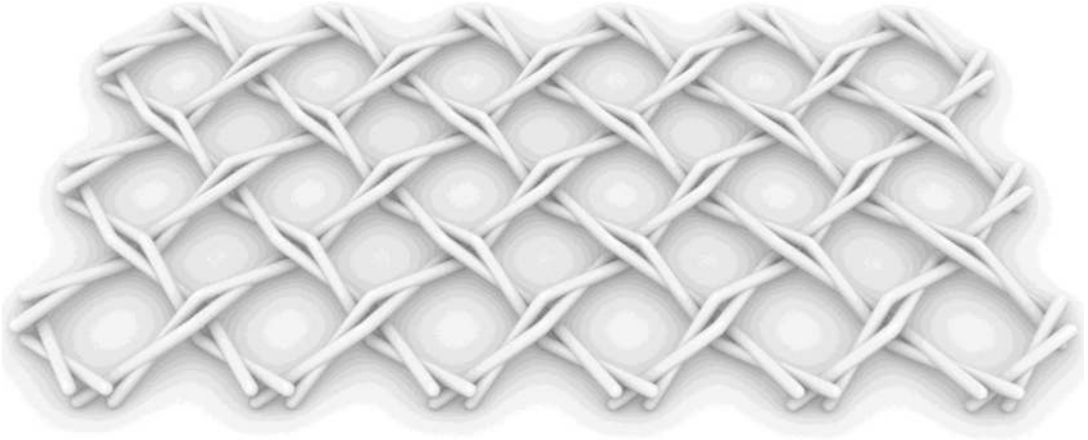


Figure 8-18. *The twisted double-weave structure generated by the node graph in Figure 8-17*

Triaxial Weave from a Triangular Grid

So far, we've explored variations of basket weaves, which use threads along two axes based on an underlying rectangular grid. In this section, we'll discuss triaxial weaves, which use *three* axes with an underlying *triangular* grid.

Figure 8-19 shows the node graph for creating a triaxial weave based on a triangular grid. You'll notice that the overall graph is almost identical to the basket weave graph (Figures 8-2 to 8-8), except for RecGrid (1a) is replaced by TriGrid, and we now use sets of three Line and Evaluate Curve nodes instead of four for finding grid edge midpoints (2c1 to 2c3, 2d1 to 2d3) and linking lines between the lower and upper grids (4a1 to 4a3).

Just like basket weaving, we link the edge midpoints between the two grids by linking each point in the lower grid with its *predecessor* in the upper grid. Each cell in the triangular grid has three edge midpoints—we'll call the points in the lower grid L1, L2, and L3 and the corresponding points in the upper grid (which are directly above) U1, U2, and U3. The points are then paired up as (L1, U3), (L2, U1), and (L3, U2). Note that the points' indices wrap around (1-2-3, 2-3-1, and 3-1-2); therefore, 1's predecessor is 3.

Figure 8-20 shows the weaving steps in images. (1) shows the triangular grid with its cell edge midpoints (1a and 2d1 to 2d3 in Figure 8-19). (2) shows a second set of cell edge midpoints that float directly above the first set, created by copying the first set and moving it along +Z. (3) shows the two sets of cell edge midpoints interconnected into the triaxial pattern, such that each point in the lower grid is linked to its predecessor in the upper grid (4a1 to 4a3 and 4b). (4) shows the final output of the graph in Figure 8-19, where the weaved line segments have been MultiPiped (4c) to create volume.

Figure 8-21 shows a rendering of the triaxial weave. You can see that the weave threads frame hexagon-shaped holes. If you start at a hole and zoom out a bit to include the surrounding thread intersections, six-point stars begin to emerge. You can also focus on the thread intersections, which look like equilateral triangles.

Note Where to find the nodes in Figure 8-19: **Vector ➤ Grid ➤ Triangular (TriGrid) (1a)**, Surface ➤ Analysis ➤ Deconstruct Brep (2a), Sets ➤ List ➤ List Item (2b), Curve Primitive Line (2c1, 2c2, and 2c3), and Curve ➤ Analysis ➤ Evaluate Curve (2d1, 2d2, and 2d3), Transform ➤ Euclidean ➤ Move (3a), Surface ➤ Analysis ➤ Deconstruct Brep (3b), and Sets ➤ List ➤ List Item (3c). Curve ➤ Primitive ➤ Line (4a1, 4a2, and 4a3), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

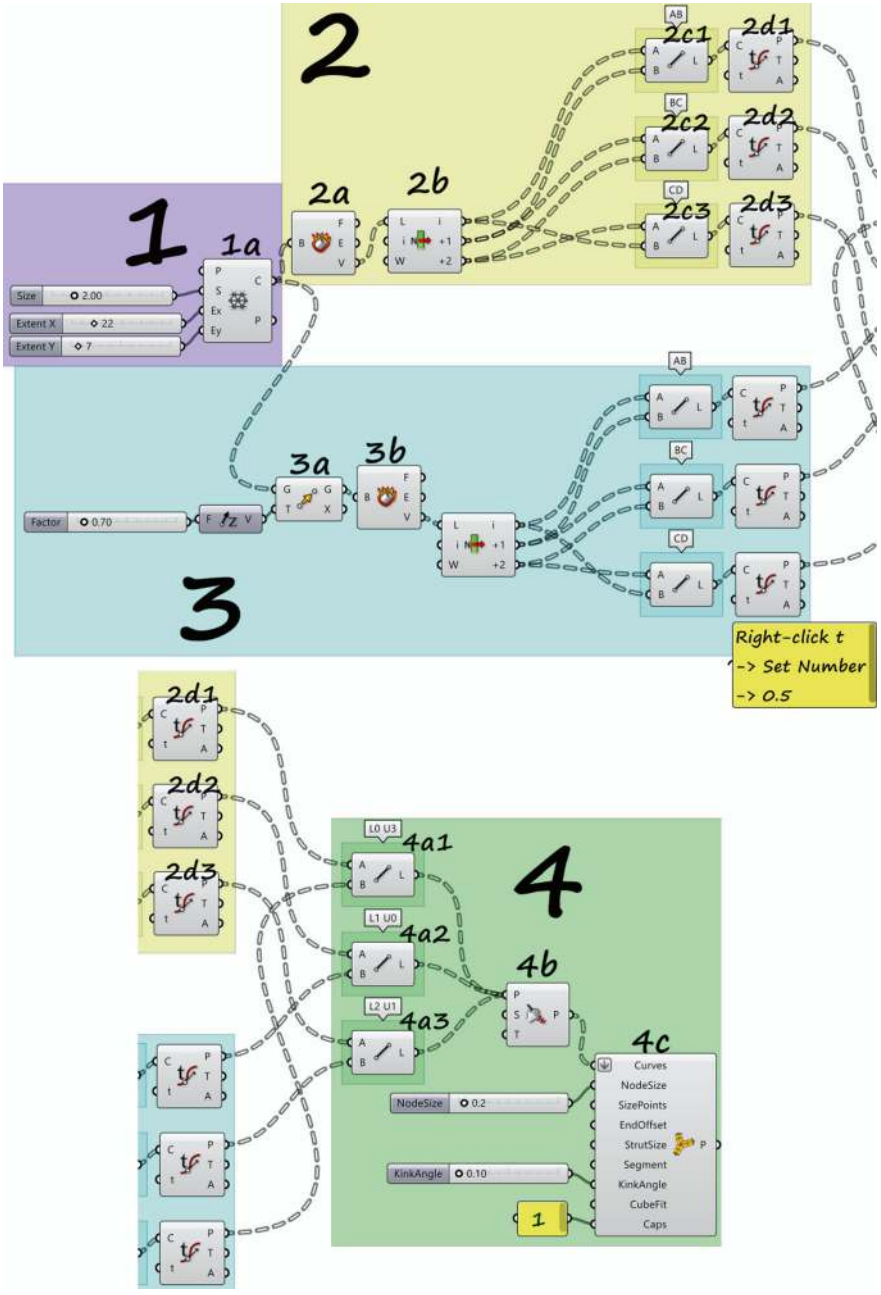


Figure 8-19. *Node graph for generating a triaxial weave from a triangular grid. The underlying principle is largely unchanged from the basket-weaving graph (Figures 8-2 to 8-8), except a triangular grid is used instead of a rectangular grid, and sets of three Line and Evaluate Curve nodes are used to link the line segments between the lower and upper grids instead of four.*

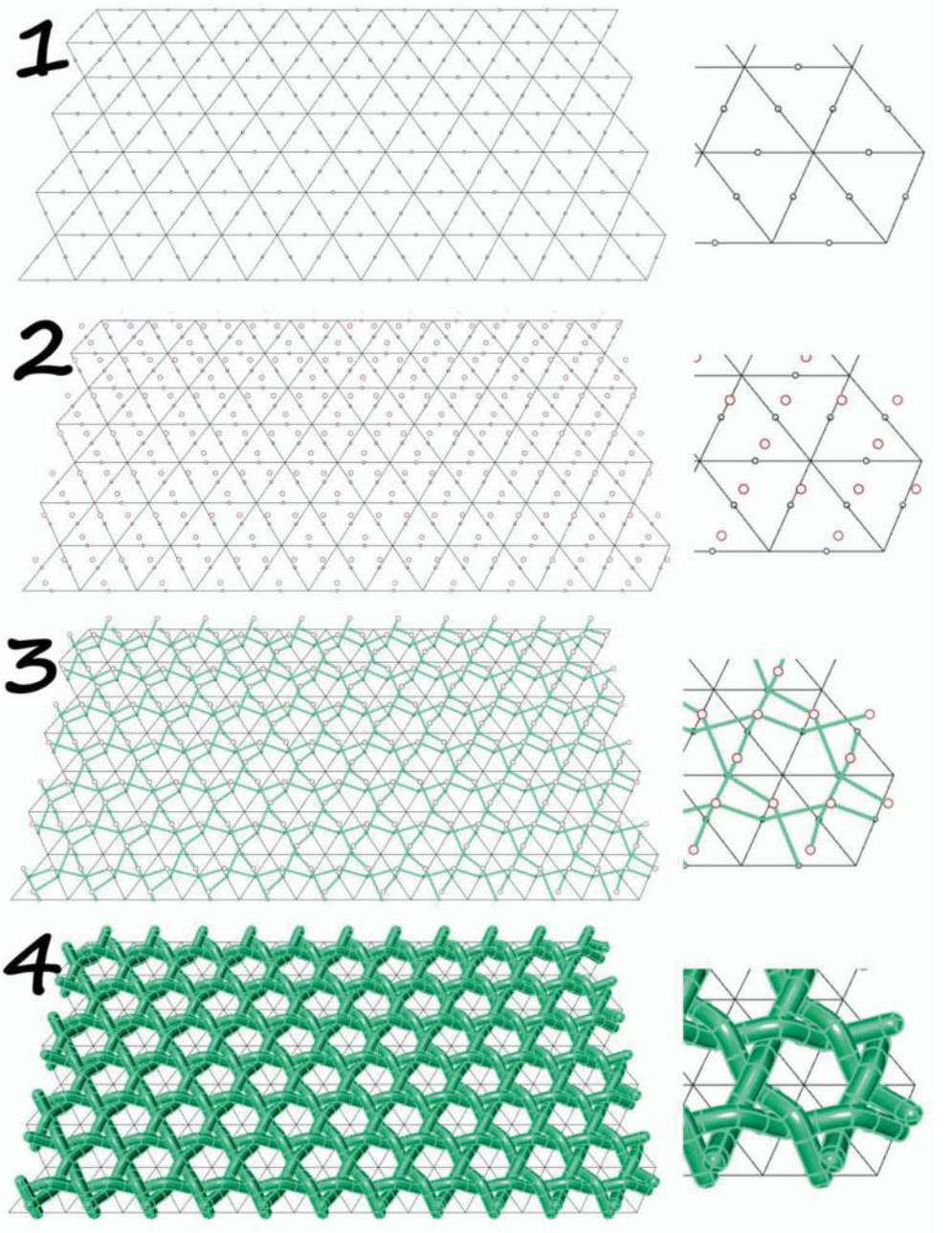


Figure 8-20. *Steps for triaxial weaving visualized. (1) shows a triangular grid with edge midpoints of its cells. (2) A Second set of edge midpoints are create to float above the first set. (3) The two sets of points are linked into the triaxial weave pattern, such that each point in the lower grid is connected to its predecessor in the upper grid. (4) The weave is MultiPiped to create volume*

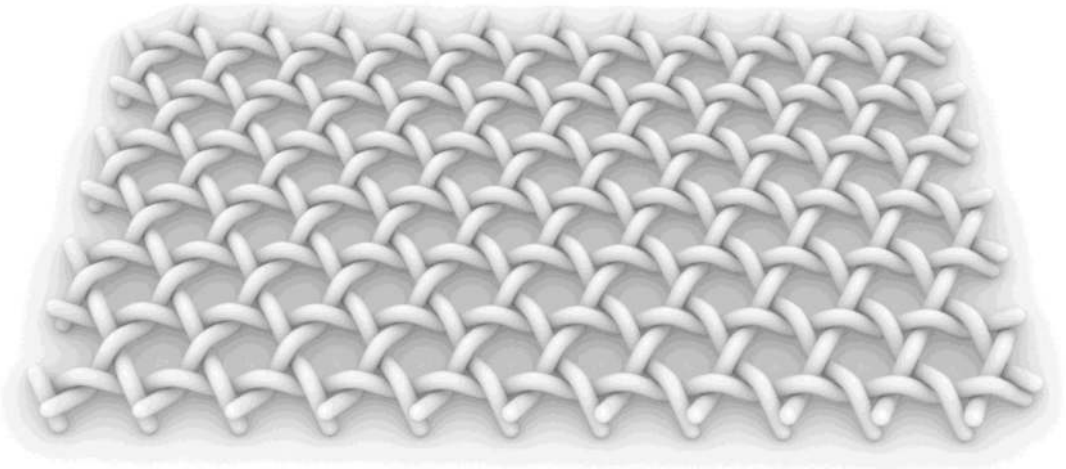


Figure 8-21. *The triaxial weave generated by the node graph in Figure 8-19*

Using Python to Form the Triaxial Weave Structure

Like the case with basket weaving, where we simplify the node graph by replacing the Line and Evaluate curves nodes (Figures 8-2 to 8-8) with Python 3 Script (Figure 8-9), we can simplify the triaxial weaving graph (Figure 8-19) using the same strategy. Since we've structured the Python code in Listing 8-1 to work with grids of any shape, it turns out we can simply replace the Rectangular (RecGrid) node (1a) in Figure 8-9 with Triangular (TriGrid), as shown in Figure 8-22 with everything else unchanged, to produce a triaxial weave instead of a basket weave.

Note Where to find the nodes in Figure 8-22: **Vector ➤ Grid ➤ Triangular (TriGrid) (1a)**, Surface ➤ Analysis ➤ Deconstruct Brep (2a and 3b), Params ➤ Util ➤ Param Viewer (2b), Transform ➤ Euclidean ➤ Move (3a), Maths ➤ Script ➤ Python 3 Script (4a), Curve ➤ Division ➤ Smooth Polyline (4b), and Surface ➤ SubD ➤ MultiPipe (4c).

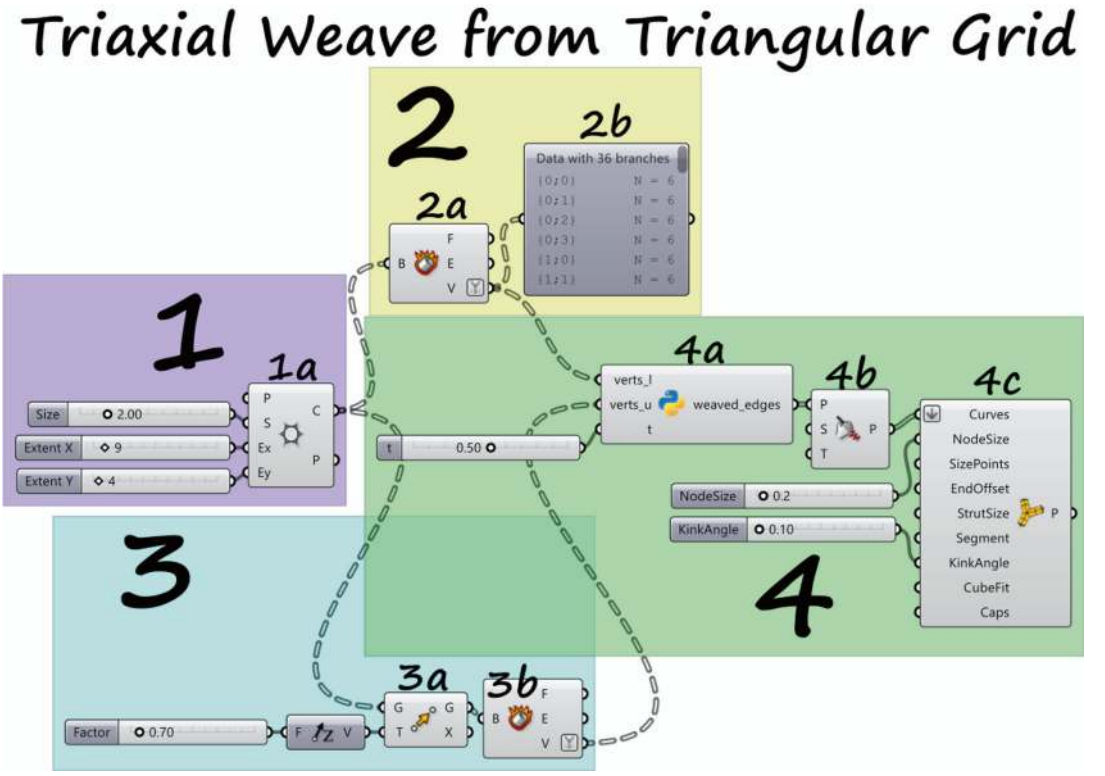


Figure 8-22. *Python-3-based node graph for generating a triaxial weave from a triangular grid, created by replacing the Rectangular (RecGrid) node in Figure 8-9 with Triangular (TriGrid, 1a) while leaving everything else unchanged*

Triaxial Weaving an Input Mesh

Recall that to basket weave an input mesh, we preprocessed it by reducing polys and converting its faces to quads so its surface becomes a rectangular grid. Similarly, to triaxial weave a mesh, we'll first reduce its polys and convert its faces to *triangles* so its surface becomes a *triangular* grid.

We'll modify the node graph in Figure 8-19 as follows: First, replace stage 1 with the new stage 1 shown in Figure 8-23, where we link Mesh (1a) to Quad Remesh (1b) and set its S input using Quad Remesh Settings (1c) with Tc = 500—this converts the mesh's faces to quads and reduces its polygon count to 500. A poly count of 500 works well for

the Stanford bunny mesh internalized in the provided GH definition `/Ch8/Figure8-23_to_8-25_triaxial_bunny_weave.gh`. For a different mesh, you may need to experiment with different Tc values.

Next, we link Quad Remesh (1b) to TriRemesh (1d) with Length = 6 to convert the mesh's faces to triangles with an approximate edge length of six. Again, this is a value that works well with the Stanford bunny mesh, which you may need to adjust if you are using a different mesh.

With the mesh triangulated, the next step is to extract its vertices on a per-face basis, similar to finding the corners of each cell on a triangular grid. We start by linking TriRemesh (1d) to Deconstruct Mesh (2a) to get an unsorted list of vertices (via V). To look up the verts on a per-face basis, we link Deconstruct Mesh's F output to Deconstruct Face (2b) to get the vert indices framing each face and in turn use them with List Item to index the vertices. The rest of stage 2 where we derive the edge midpoints with Line and Evaluate Curve nodes remains the same.

In stage 3, we link TriRemesh (1d) to Weaverbird's Offset Mesh (3a) to make a larger copy of the mesh. You specify how much to scale the mesh in terms of *how far* (D) the scaled copy should sit above the surface of the original when overlaid on top. A positive D (like the 1.5 we set in Figure 8-23) creates a larger mesh that encloses the original, whereas a negative D creates a smaller mesh that goes inside the original. After Weaverbird's Offset Mesh (3a), we use the same setup from stage 2, with Deconstruct Mesh for extracting the larger mesh's vertices, followed by Deconstruct Face for finding the vert indices framing each face to access the verts on a per-face basis.

Figure 8-24 demonstrates the steps for triaxial weaving a mesh visually. (1) shows the mesh following poly reduction and triangulation with edge midpoints in black. The larger copy created by Weaverbird's Offset Mesh is superimposed over the original in blue (2) shows a portion of the two meshes zoomed in—you can see the mesh faces resemble triangular grids, and the edge midpoints of the larger mesh float above the edge midpoints of the smaller mesh. (3) shows the two sets of edge midpoints interconnected into the triaxial pattern. If you follow the edge midpoints framing each face, each point in the lower triangle (on the smaller mesh) is linked to its predecessor in the upper triangle (on the larger mesh). For example, if the three points are indexed L1, L2, and L3 on the smaller mesh, with U1, U2, and U3 directly above them on the larger

mesh, the links between them are (L1, U3), (L2, U1), and (L3, U2). (4) shows the final result after MultiPipe creates volume for the weaved line segments. Figure 8-25 shows the triaxial weaved Stanford bunny output by the node graph in Figure 8-23 after baking.

Note Where to find the nodes in Figure 8-23: Params ➤ Geometry ➤ Mesh (1a), Mesh ➤ Triangulation ➤ Quad Remesh (1b), Mesh ➤ Triangulation ➤ Quad Remesh Settings (1c), **Mesh ➤ Triangulation ➤ TriRemesh (1d)**, Mesh ➤ Analysis ➤ Deconstruct Mesh (2a), Mesh ➤ Analysis ➤ Deconstruct Face (2b), Sets ➤ List ➤ List Item (2c1, 2c2, 2c3, and 2c4), and **Wb ➤ Transform ➤ Weaverbird's Offset Mesh (3a)**.

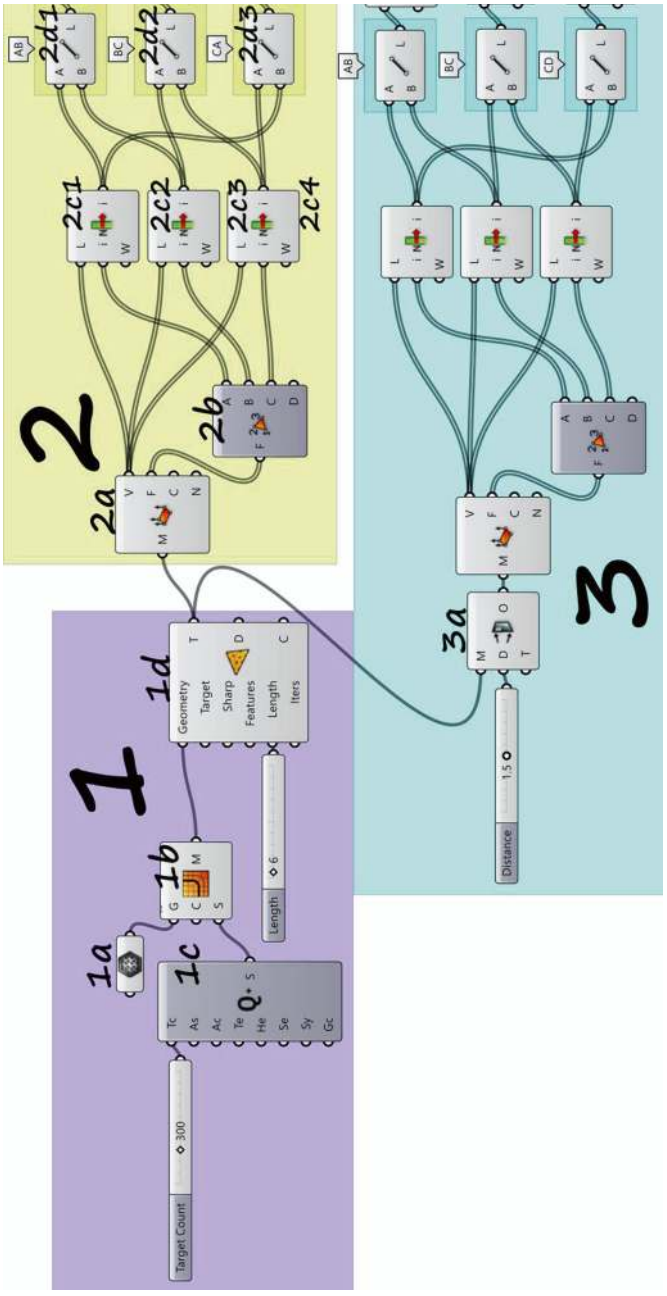


Figure 8-23. *Generating a triaxial weave based on an input mesh. In stage 1, the mesh undergoes a poly reduction followed by triangulation. Stage 2 derives the mesh's edge midpoints. Stage 3 makes a larger copy of the mesh and finds the midpoints of its edges, which float above the edge midpoints of the original*

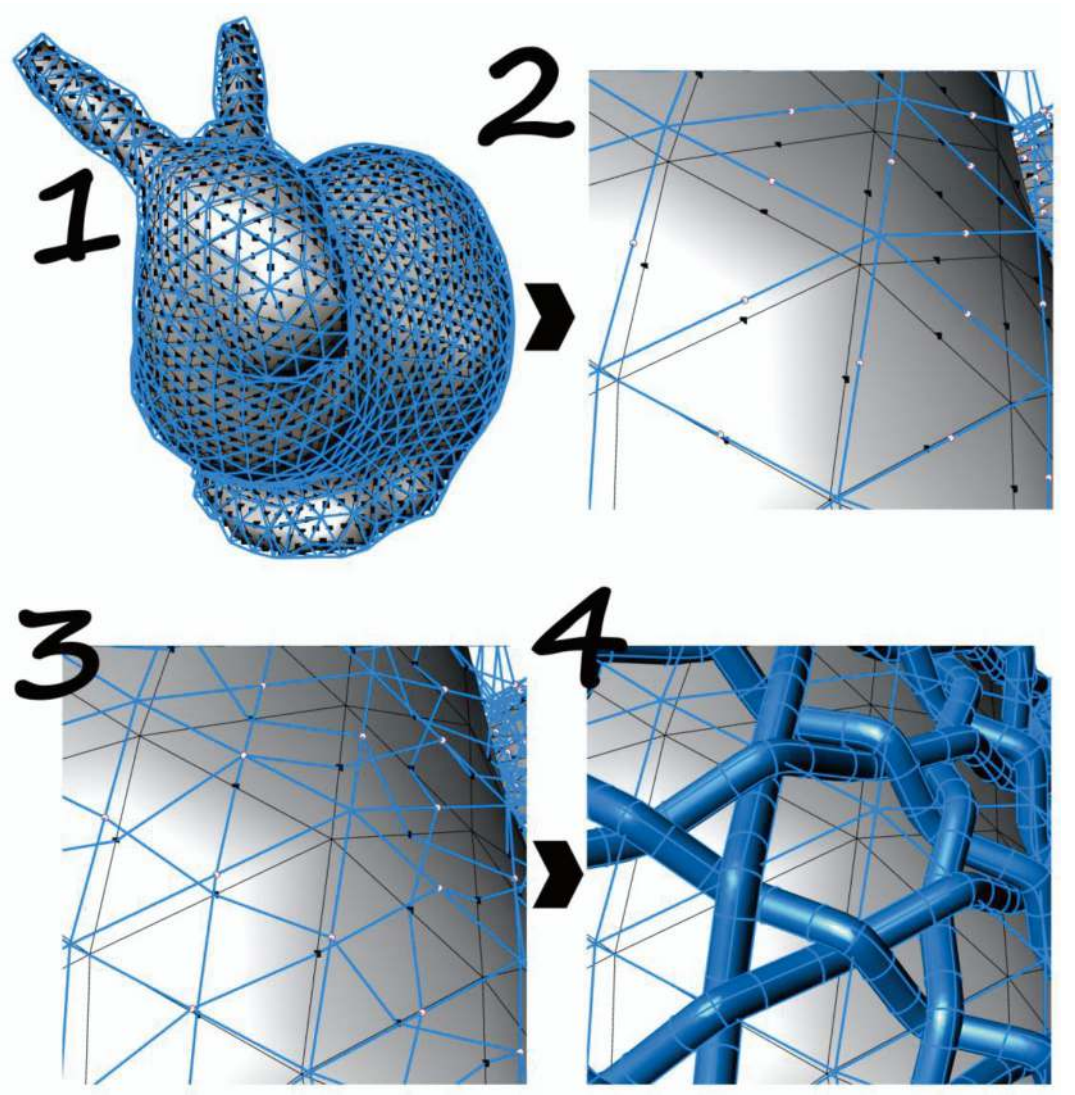


Figure 8-24. *Steps for generating a triaxial weave based on an input mesh. (1) The Stanford bunny mesh after poly reduction and triangulation, with a scaled copy made with Weaverbird's Offset Mesh overlaid on top (in blue). (2) An area of (1) zoomed in to show edge midpoints of both meshes. (3) The two sets of edge midpoints linked into a triaxial weave structure, where on a per-face basis, each point on the lower mesh is linked to its predecessor on the upper mesh. (4) The weave structure MultiPiped to create volume*

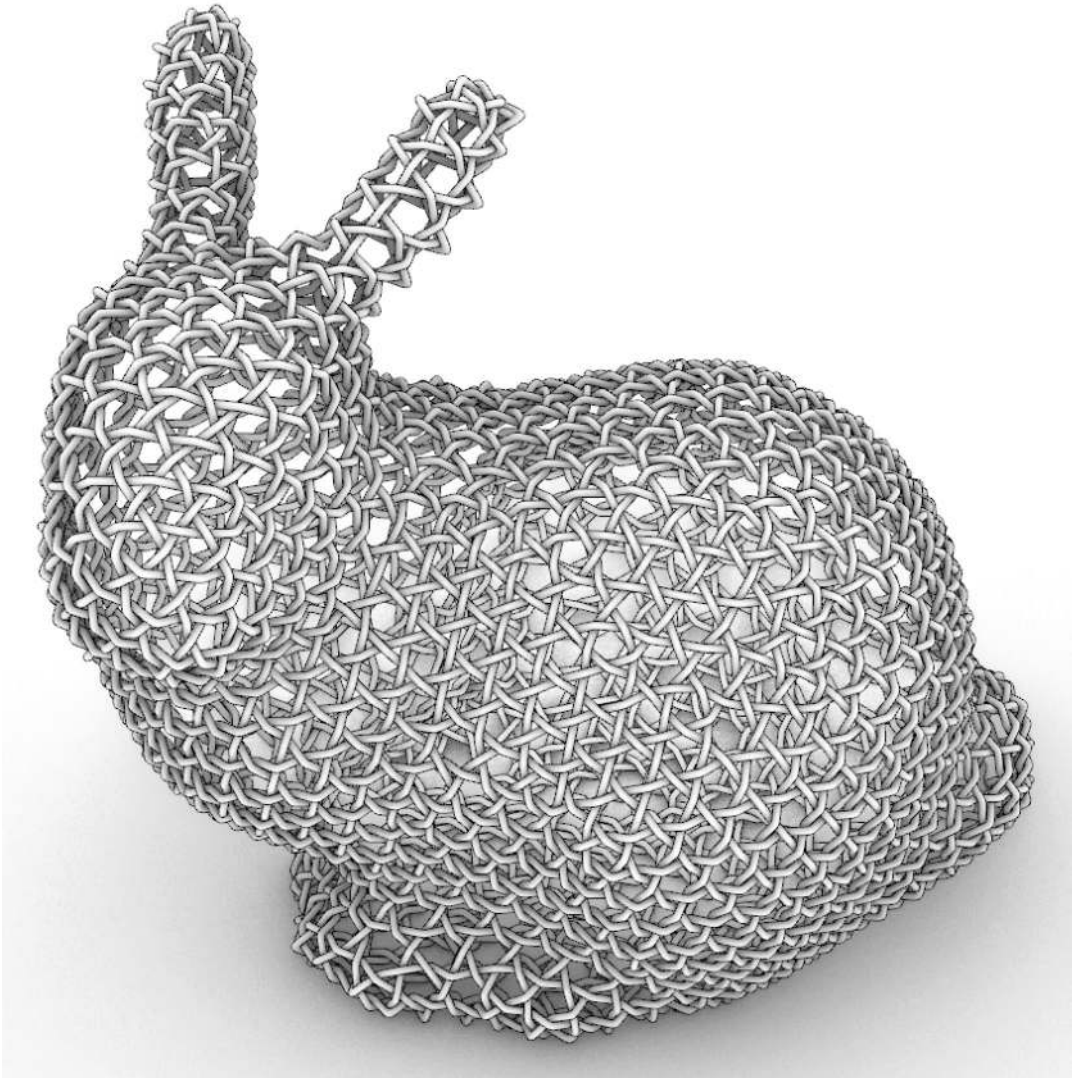


Figure 8-25. *The triaxial weaved Stanford bunny generated by the node graph in Figure 8-23*

Summary

In this chapter, you worked through two categories of weaving—basket weaves and triaxial weaves. Basket weaves interlock threads along two axes in a repeating over-under pattern, whereas triaxial weaves use threads along three axes.

The first part of the chapter took you through implementing a Grasshopper definition that generated basket weaves based on rectangular grids, by finding the grid's edge midpoints, creating a copy that float above, and linking each edge midpoint in the lower grid with its predecessor in the upper grid to form weaved line segments that are MultiPiped to create volume. You then learned how to replace the sets of Line and Evaluate Curve notes for forming weaved line segments with a Python 3 Script node, thus simplifying the node graph significantly.

With basket-weaving basics down, you extended the implementation to weave from an input mesh rather than a grid. The steps involved preprocessing the mesh to reduce its polygon count and converting its faces to quads, resembling a rectangular grid projected onto the mesh. Weaverbird's Offset Mesh is used next to make a scaled-up copy of the quad mesh so it fully encloses and is equidistant everywhere to the smaller mesh. Each edge midpoint on the smaller mesh is then linked to its predecessor on the larger mesh.

The next part of the chapter showed how to create three variations of basket weaves. The first is a *double weave* created by combining weave threads shifted along and opposite of grid edge tangents. The second is a *triple weave* produced by combining the double weave and the original basket weave. The third is a *twisted double weave* created by combining two sets of weaves with slightly different t values for sliding edge points.

In the following part of the chapter, you shifted your focus to a different weave category—triaxial weaves, which uses three axes instead of two in an over-under pattern, requiring a triangular grid instead of rectangular. Other than that, the approach for forming the weave structure is largely the same as basket weave, where the grid edge midpoints are derived, a second copy of the points is offset to float above, and each point in the lower grid is linked to its predecessor in the upper grid to create the weave. The weave is passed through MultiPipe to thicken.

Lastly, you extended the triaxial weaving graph to weave an input mesh instead of a grid. The process is largely the same as basket weaving an input mesh, except an extra step to triangulate the mesh is added after Quad Remesh so the mesh faces are triangular.

Reference

Wikipedia. "Stanford bunny." *Wikipedia*, 29, August 2025, CC BY-SA 4.0, https://en.wikipedia.org/wiki/Stanford_bunny.

CHAPTER 9

Sliceforms

In this chapter, we'll build a sliceform generator that creates two sets of interlocking cross-sectional slices based on an input mesh or Brep. We'll explore variations by changing distances between slices as well as angles the slices are cut at. We'll then expand the generator to create hollow sliceforms. Lastly, we'll look at two examples of creating "mock sliceforms" by creating a 3D spiral using the Twist node and composing the spiral with a rotated copy of itself.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper. Download the files and navigate to the /Ch9 folder, where you'll find *.gh files named after the figures in which they appear, along with a short description. For example, /Ch9/Figure9-3_ComputingBoundingRadius.gh. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, build your own node graphs by following the steps described in the text or start with the downloaded *.gh files and modify their parameters to experiment.

Installing the Weaverbird Plugin

Some examples in this chapter make use of the free GH plugin Weaverbird developed by Giulio Piacentino. To install Weaverbird, go to https://www.giuliopiacentino.com/weaverbird/ and locate the instructions for your Rhino version. For Rhino 8 and 9, the process is simple and only requires you to download and open a *.gh file in Grasshopper—the Package Manager will handle the rest.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_9

Attribution to the Stanford Bunny

The Grasshopper files listed below are internalized with the Stanford bunny mesh, a computer graphics test model developed by Greg Turk and Marc Levoy in 1994 at Stanford University (Wikipedia, 2025). The version included is modified by the author to have a closed bottom.

- /Ch9/Figure9-3_ComputingBoundingRadiusgh
- /Ch9/Figure9-4_to_9-5_CreatingAStackOfPlanesForSlicing.gh
- /Ch9/Figure9-6_to_9-7_FormingSlices.gh
- /Ch9/Figure9-8_to_9-9_ExtrudingSlicesForThickness.gh
- /Ch9/Figure9-10_to_9-11_SlicingAtTwoAngles.gh
- /Ch9/Figure9-12_to_9-13_Sliceform_BunnyMesh_Variations.gh
- /Ch9/Figure9-16_9-17_Sliceform_BunnyMesh_Hollow.gh

What Is a Sliceform?

A *sliceform* is a type of geometry made from interlocking two sets of cross-sectional slices of a model (Lupiani, 2025). The two sets of slices are cut at different angles and often 90 degrees to one another. When made in a small scale, sliceforms are often manufactured in cardboard or wood, like the hemisphere model made by the author in Figure 9-1.

The slits allowing slices to slot together can be sized just wide enough to make the assembled model rigid or wider to allow flex. A rigid sliceform creates a great modern aesthetic for a permanent structure. A notable example is the Setas de Sevilla ("Mushrooms of Seville"), also known as Las Setas or Metropol Parasol, as shown in Figure 9-2. Nonrigid sliceform can be used to create collapsible structures like those used in pop-up cards and books or deployable architecture which can be easily transported and constructed on site. The flat nature of the slices also makes them easy to manufacture and store.

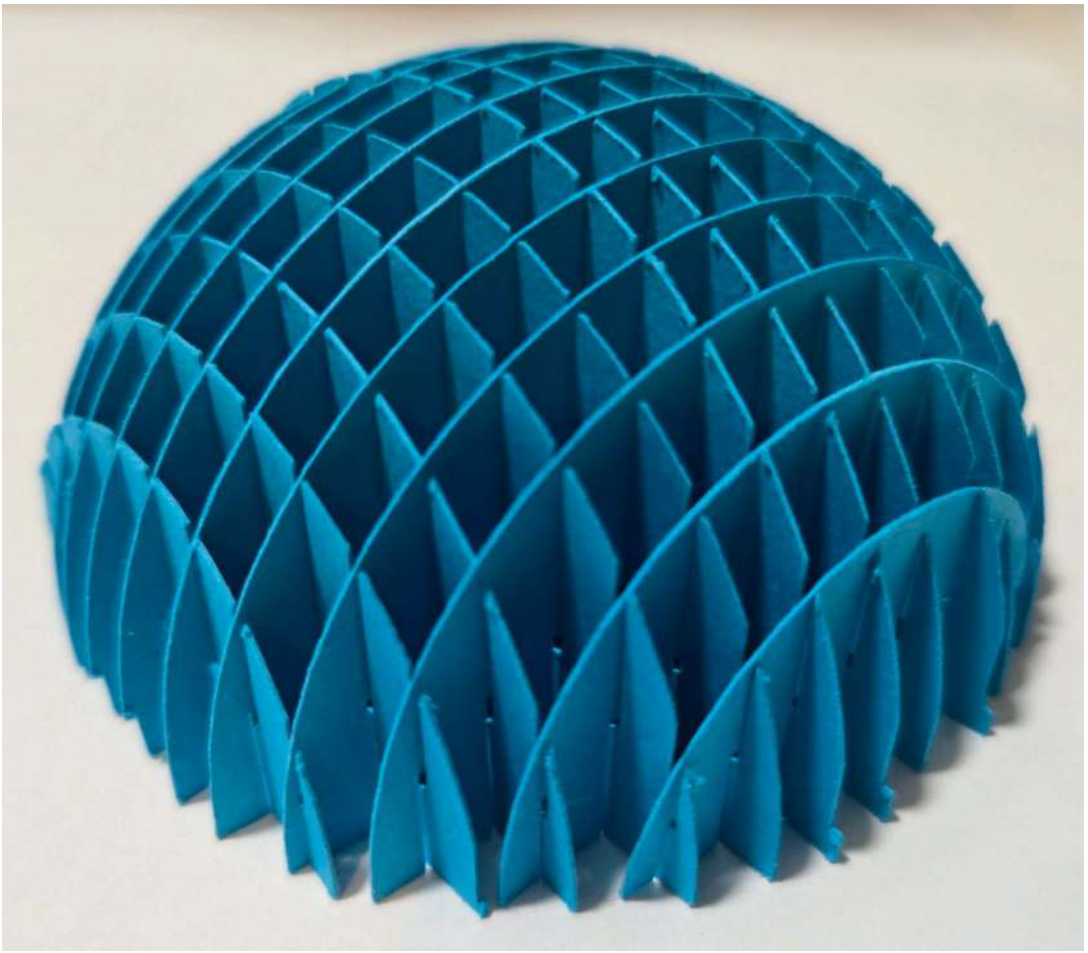


Figure 9-1. *Hemisphere sliceform model made from cardstock by the author. Photo also by the author*



Figure 9-2. *Setas de Sevilla ("Mushrooms of Seville"), also known as Las Setas or Metropol Parasol, is a large structure in Seville, Spain, that resembles a sliceform model. Photo by Anual, 8 November 2011* https://commons.wikimedia.org/wiki/File:Metropolparasolnov2011001.jpg*, CC BY-SA 3.0, via Wikimedia Commons*

Generating a Sliceform from an Arbitrary Input Mesh

We'll start our exploration of sliceforms by building a node graph that takes an arbitrary mesh as input and generates a sliceform version of it. This is the most general case as we impose no constraint on the mesh. The only limitation is that the distance between slices needs to accommodate the size of the mesh and capture enough detail. We'll structure the generator such that either a mesh or a brep can be used as input.

The generator will be gradually built up in stages. We'll first slice in one direction and bring the graph to its completion and later extend the graph to slice in two angles that can be parametrically set. We'll compute the bounding radius of the input model (stage 1)

and based on which create evenly spaced planes along one axis (stage 2). Slices are formed next (stage 3) by creating surfaces from intersection curves between the planes and the model. The slices are then extruded for thickness (stage 4). With slicing in one direction working, we circle back and extend stage 2 to create two stacks of planes for slicing instead of one.

Stage 1: Computing the Bounding Radius of the Input Mesh

Our goal in stage 1 (Figure 9-3) is to compute the radius of the input mesh's bounding sphere so we know the extent to create the stack of planes for slicing, no matter the slicing angle. Since we impose no assumptions nor restrictions on the input mesh, its size could vary wildly in each dimension. Therefore, we use the model's bounding sphere diameter as the central axis for creating the stack of planes for slicing. Since the bounding sphere will enclose the model in *all* directions, we are guaranteed to have enough planes to intersect the model with no matter which direction we take.

You can find the node graph in Figure 9-3 in the file `/Ch9/Figure9-3_ComputingBoundingRadius.gh` in this chapter's downloaded source, which comes with the Mesh node (1a) already internalized with the Stanford bunny mesh. To change the mesh, open the Rhino (`*.3dm`) file containing the mesh, select the mesh in the Rhino viewport, and right-click 1a ➤ Set one Mesh in Grasshopper. Once the new mesh is loaded, you can right-click 1a ➤ Internalise data to save the mesh in the Grasshopper definition so you won't need to open the Rhino file with it each time.

We can estimate the mesh's bounding radius, or radius of its bounding sphere—the smallest sphere that encloses the entire mesh—with the following formula:

$$r = \frac{\sqrt{X^2 + Y^2 + Z^2}}{2}$$

where X, Y, and Z are the extents of the mesh along those three axes. To translate this to Grasshopper, we link Mesh (1a) to Bounding Box (1b) followed by Deconstruct Box (1c) to extract its X, Y, and Z extents as domains. We then use Deconstruct Domain (1d) to find the start (S) and end (E) of each domain, followed by E – S (1e) to find the length of each domain, or the size of the mesh along each direction (X, Y, and Z). After that, the Square node (1f) gives us the list

$$\left[X^2, Y^2, Z^2 \right]$$

which we can sum up using Mass Addition (1g) to get

$$X^2 + Y^2 + Z^2$$

We then take the Square Root (1h) to get

$$\sqrt{X^2 + Y^2 + Z^2}$$

And multiply 1h by 0.5 (1i), thus divide it in half, to find the bounding radius:

$$r = \frac{\sqrt{X^2 + Y^2 + Z^2}}{2}$$

Note Where to find the nodes in stage 1 of Figure 9-3: Params ➤ Geometry ➤ Mesh (1a), Surface ➤ Primitive ➤ Bounding Box (1b), Surface ➤ Analysis ➤ Deconstruct Box (1c), Maths ➤ Domain ➤ Deconstruct Domain (1d), Maths ➤ Operators ➤ Subtraction (1e), Maths ➤ Polynomials ➤ Square (1f), Maths ➤ Operators ➤ Mass Addition (1g), Maths ➤ Polynomials ➤ Square Root (1h), and Maths ➤ Operators ➤ Multiplication (1i).

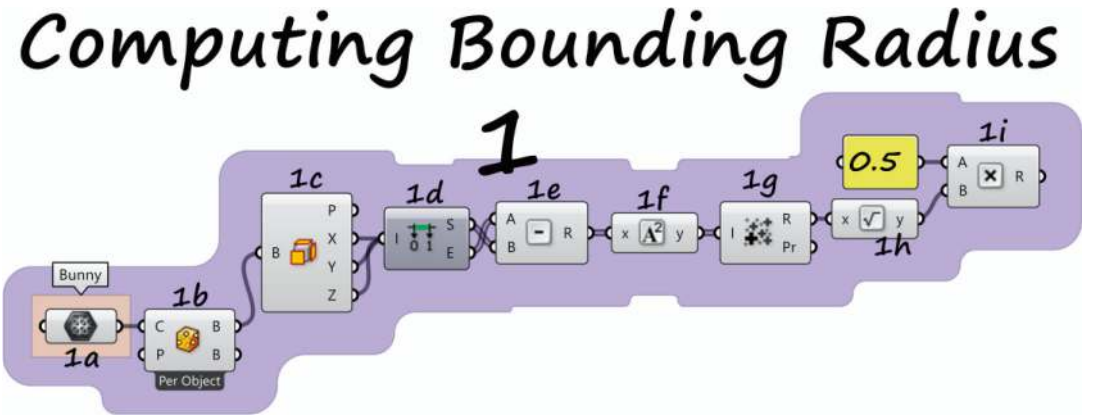


Figure 9-3. *Computing the bounding sphere radius for the input model. We use Bounding Box (1b) to find the model's extent along X, Y, and Z and take the square root (1h) of the sum (1g) of the extent's squares (1f) to find the bounding diameter—the half (1i) of which is then the bounding radius*

Stage 2: Creating a Stack of Planes for Slicing the Mesh

In stage 2 (Figure 9-4), we'll create a stack of planes along Z that spans the mesh bounding sphere's entire *diameter* through the sphere's center. Using the mesh's bounding sphere as reference for creating the slicing planes here ensures a complete and even coverage of the entire mesh.

We will use Linear Array (2i) to create the stack of planes. Since we are orientating the stack to align with the global Z axis, we need to start the stack at the bottom most Z of the bounding sphere. We link Bounding Box (1b) to Area (2a) to find the mesh's bounding box center. We then find the center's Z with Deconstruct (2b) and subtract (2c) the bounding radius (1i) from it to get the bounding sphere's bottom most Z. The three-node sequence 2c, 2d, and 2e finds the bottom-most Z, creates the vector (0, 0, bottom-most Z), and creates a plane parallel to XY centered at (0, 0, bottom-most Z). We will use this plane as the G input for Linear Array (2i) or the starting element of the stack.

Next, we use a Number Slider (2f) to control the spacing between neighboring slices—this is the distance between slices if the slices have no thicknesses. We will use a separate slider later to adjust the slice thickness. The number of planes (N input for Linear Array (2i)) is the diameter (1h) of the bounding sphere divided by the slice spacing (2f). The direction and interval to repeat the plane (D input for Linear Array (2i)) is then a Z vector with the slicing spacing as its length, which is created by linking the slice spacing (2f) to Unit Z.

Figure 9-5 shows the steps of constructing the stack of planes visually. We find the mesh's bounding box (A) and from which the bounding sphere (B). The stack of planes (D) is then created along Z to span the bounding sphere's entire diameter and go through its center.

Note Where to find the nodes in stage 2 of Figure 9-4: Surface ➤ Analysis ➤ Area (2a), Vector ➤ Point ➤ Deconstruct (2b), Maths ➤ Operators ➤ Subtraction (2c), Vector ➤ Vector ➤ Unit Z (2d), Vector ➤ Plane ➤ XY Plane (2e), Params ➤ Input ➤ Number Slider (2f), Maths ➤ Operators ➤ Division (2g), Maths ➤ Util ➤ Round (2h), and Transform ➤ Array ➤ Linear Array (2i).

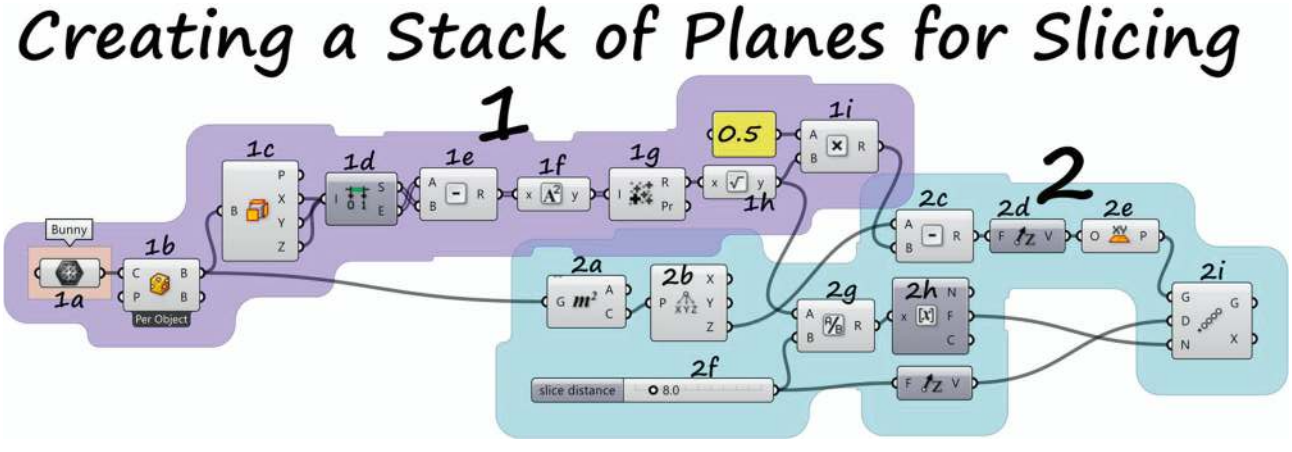


Figure 9-4. *In stage 2, we use Linear Array (2i) to create an evenly spaced stack of planes spanning the bounding diameter of the input model along one direction, to be used for intersecting with the model ("slicing") later*

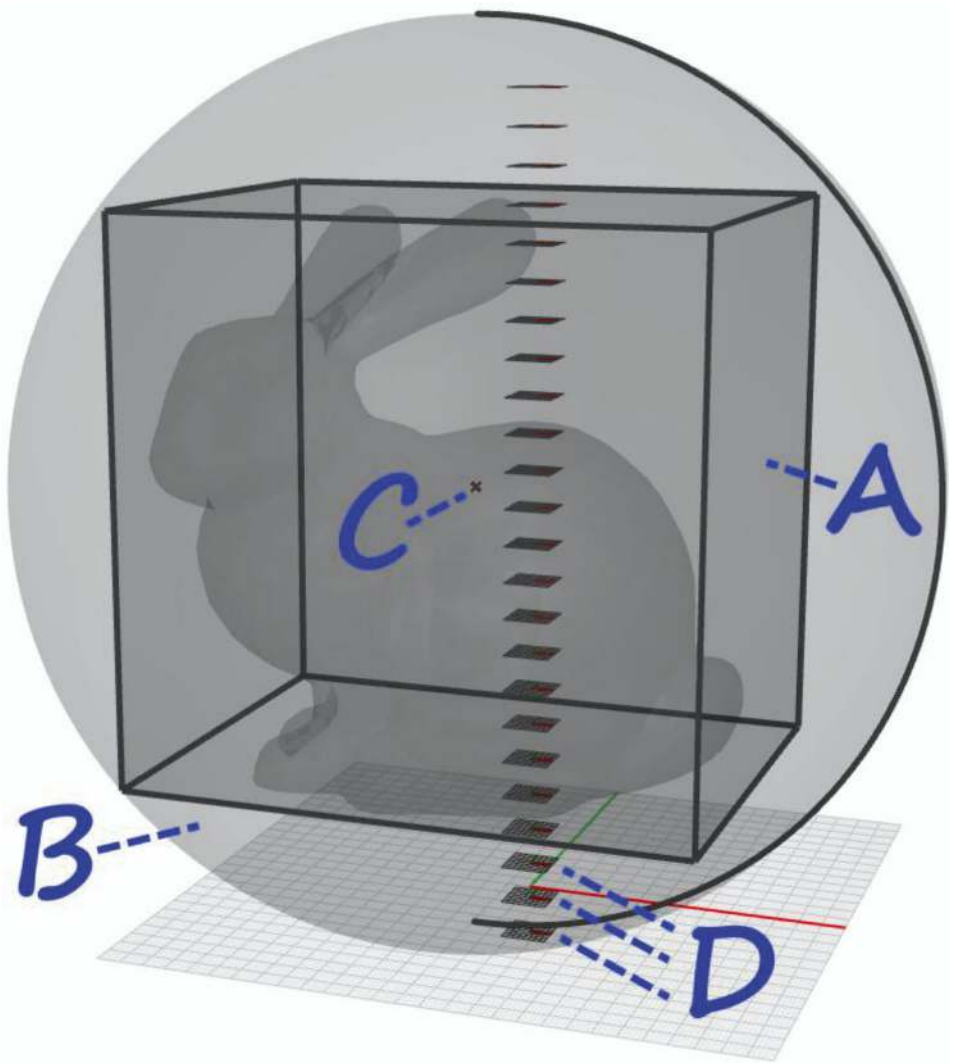


Figure 9-5. *Visualizing how the stack of planes for slicing is constructed. We use the dimensions of the input model's bounding box (A) to compute the radius for its bounding sphere (B). We then create an evenly spaced stack of planes (D) that spans the bounding radius on both sides of the sphere's center (C)*

Stage 3: Forming Slices in One Direction

In stage 3 (Figure 9-6), we create slice surfaces by filling in the intersection curves between the stack of planes and the input mesh.

We use Mesh | Plane (3a) to find the intersection curves between the stack of planes (2i) and the input mesh. Mesh | Plane (3a) outputs a data tree with one curve per branch, which is not necessary, so we right-click ➤ Flatten to simplify it into a list. The Closed node (3b) tells us which intersection curves are closed in a list of True/False values, which we use as the P input for Dispatch (3c) to isolate the closed curves into a list (A output). We then fill in the closed intersection curves using Boundary Surfaces (3d) so they become trimmed surfaces or "slices" along one direction *without* thickness, as shown in Figure 9-7.

Note Where to find the nodes in stage 3 of Figure 9-6: Intersect ➤ Mathematical ➤ Mesh | Plane (3a, right-click output ➤ Flatten), Curve ➤ Analysis ➤ Closed (3b), Sets ➤ List ➤ Dispatch (3c), and Surface ➤ Freeform ➤ Boundary Surfaces (3d).

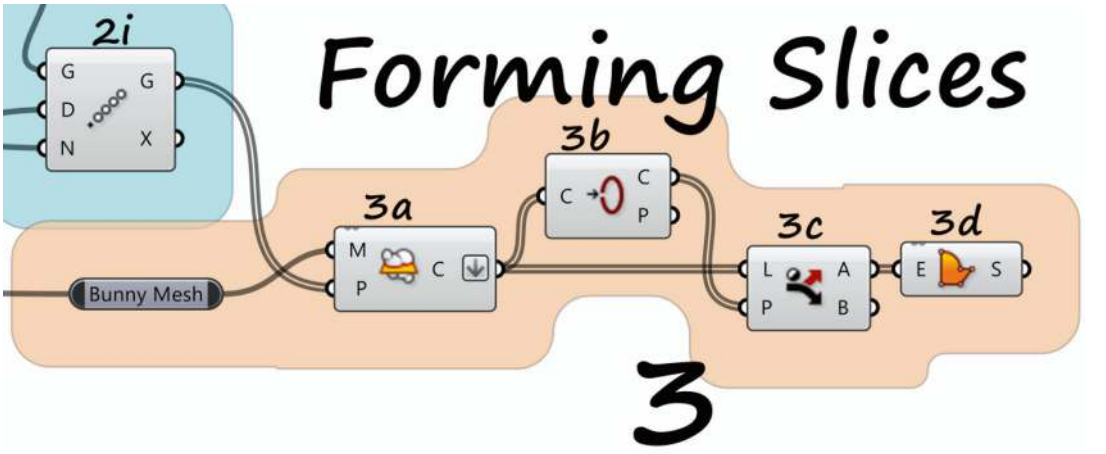


Figure 9-6. *To "slice" the model, we use Mesh | Plane (3a) to find the intersection curves of the model and the stack of evenly spaced planes (2i) and fill in only the Closed (3b) curves (with Boundary Surfaces, 3d) to create slices*

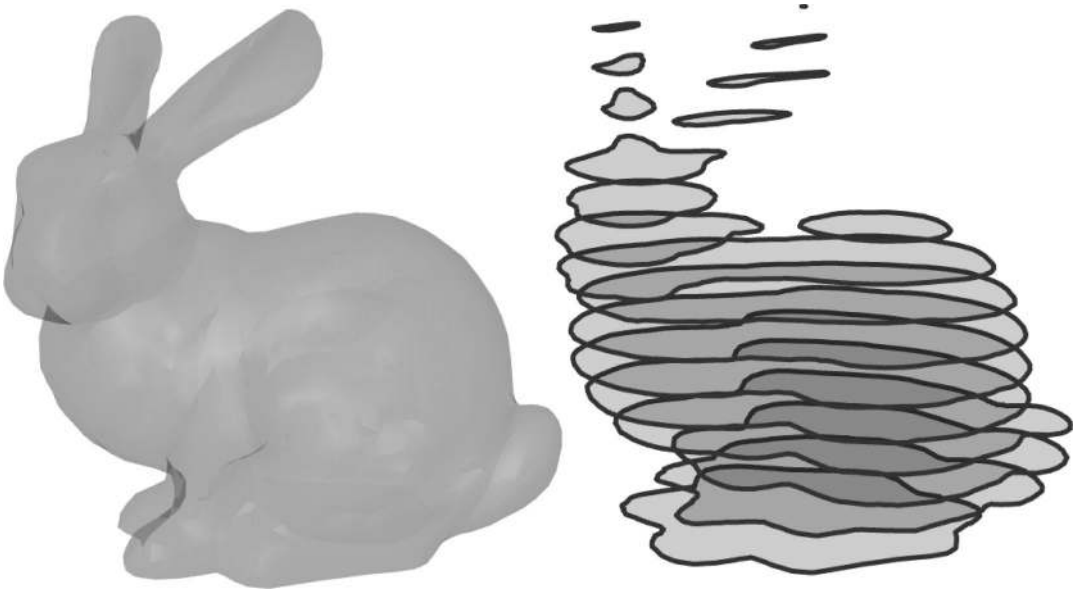


Figure 9-7. *Comparing the input model (Stanford bunny mesh, left) with its slices along Z at eight units apart (right, output of Boundary Surfaces 3d from Figure 9-6)*

Stage 4: Extruding Slices for Thickness

In stage 4 (Figure 9-8), we'll extrude the slices (trimmed surfaces) from stage 3 along their normal vectors so they have thicknesses.

So far, we've created slices parallel to the XY plane with their normal vectors pointing toward +Z, so we could simply extrude slices along +Z; however, we want to leave room for expanding the node graph so that we could slice at two different angles to create interlocking slices, as well as specify how each set of slices are rotated. Therefore, the direction for extrusion will need to "auto-adjust" based on how the slices are oriented. Luckily, there is an easy way to do this, which is using the normal vectors of the slices. The *normal vector* of a surface is a vector that points away from and is perpendicular to the surface. Therefore, by extruding along the slices' normals, we are extruding 90 degrees from each slice, no matter how the slice is oriented, which is precisely what we need.

We can use Evaluate Surface (4a) to find the normal vector of a surface at a given uv point. Recall that the uv coordinate space is formed by dividing a surface evenly into a 2D grid (which need not lay flat). You can intuitively think of u as the "X axis" and v the "Y axis." Therefore, the uv coordinates {0.5, 0.5, 0} (4b) is the point halfway along each direction or the point at the center of the surface (note that to use a Panel (4b) to specify a point, you need to format the coordinates in a comma-separated list of numbers surrounded with curly braces { }). In other words, we are asking Evaluate Surface (4a) to find properties about the surface at its center. Since our slices are flat, the normal vectors anywhere on a slice all point in the same direction; therefore, {0.5, 0.5, 0} is an arbitrary choice, since you must give Evaluate Surface (4a) a point to operate on.

The normal vectors (N) output by Evaluate Surface (4a) are unit vectors (i.e., vectors of length 1), which when used directly as the D input of Extrude (4d) will extrude the surfaces 1 unit. To make the extrusion amount (therefore slice thickness) adjustable, we insert an Amplitude node (4c) between Evaluate Surface (4a) and Extrude (4d) so we can optionally change A to scale the thickness.

Figure 9-9 shows the slices after they've been extruded one unit along their normals.

Note Where to find the nodes in stage 4 of Figure 9-8: Surface ➤ Analysis ➤ Evaluate Surface (4a), Params ➤ Input ➤ Panel (4b), Vector ➤ Vector ➤ Amplitude (4c), and Surface ➤ Freeform ➤ Extrude (4d).

The *normal vector* of a surface is a vector that is perpendicular and points away from that surface.

To specify a point via a Panel, you need to format the coordinates in a comma-separated list surrounded by curly braces { }. For example, the point at (1, 2, 3) is specified as {1, 2, 3}.

A *unit vector* is a vector of length = 1.

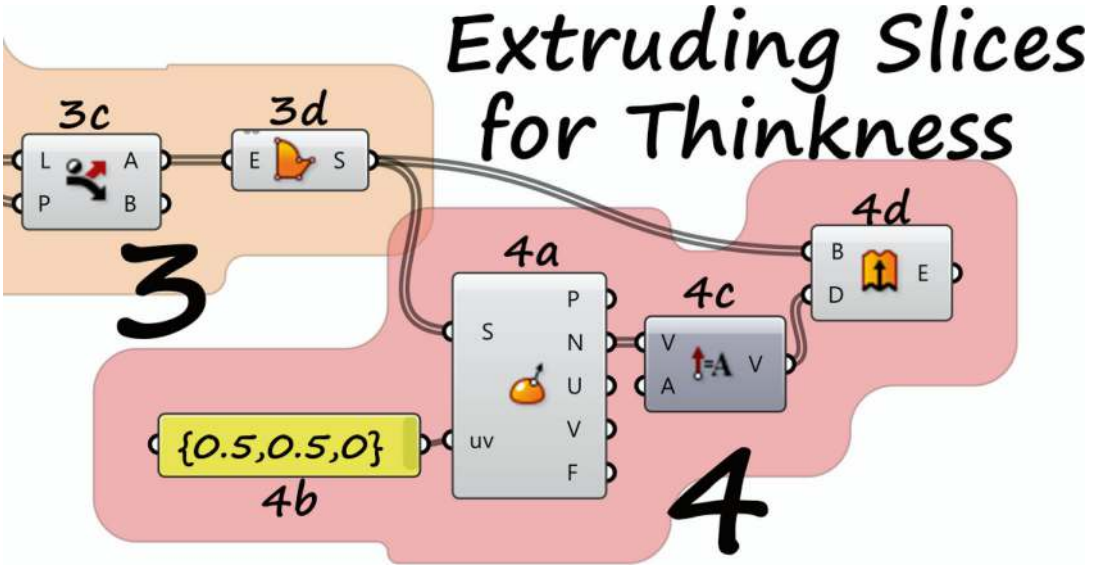


Figure 9-8. *We use Evaluate Surface (4a) to find the surface normals (N) at the centers (4b) of the slices (3d), which we use as the direction (D) to Extrude (4d) for thickness. As shown, the thickness, or Amplitude (4c) of the extrusion vector, is set to 1.0, which we can optionally adjust*

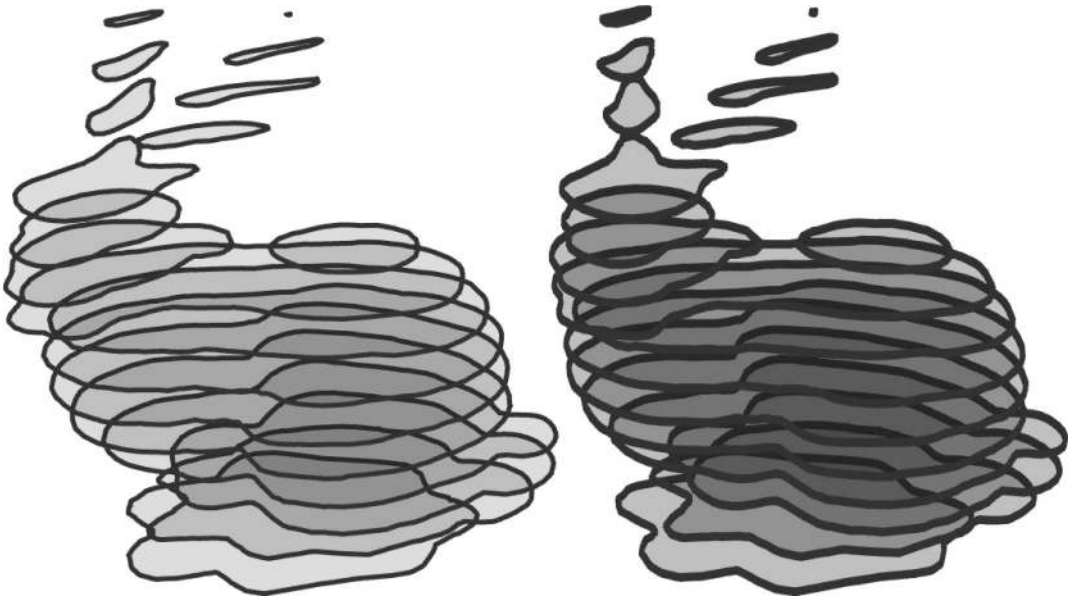


Figure 9-9. *Before (left) and after (right) the slices are extruded by (4d) in Figure 9-8. The effect is subtle since we set the thickness of the slices (or amplitude of the extrude directional vectors) to 1.0*

Extending Stage 2: Slicing in Two Directions

So far, we've successfully created slices with thickness along one direction. To create a sliceform, however, we need two sets of slices that interlock. To do so, we'll extend stage 2 of the node graph (Figure 9-4) to duplicate the stack of planes and rotate them at an angle relative to the original stack, using the node group marked "2b" in Figure 9-10.

We link the stack of planes (Linear Array, 2i) to two Rotate 3D nodes (2b1, 2b2) to create two stacks, each rotated at a different angle, with the second rotation at a relative offset from the first—these two rotations control how the two sets of slices will interlock.

We use a Number Slider (2b3) linked to Pi to create a multiple of Pi as the angle (A) for the first Rotate 3D node (2b1)—as shown, the angle is set to 0.25 Pi, or 0.25*180=45 degrees. The angle for the second Rotate 3D node (2b2) is the angle for the first (2b3) *plus* 0.5 Pi (2b5)—in other words, the second rotation is 90 degrees relative to the first, so the two sets of slices are orthogonal to one another. We finish up by Shift-dragging both Rotate 3D nodes (2b1, 2b2) to Mesh | Plane (3a), so the two sets of planes are both used to intersect with the mesh.

The bounding box center (2a) is used as the center for rotation (C) and **Unit Y** (2b4) as the rotation axis (X) for both Rotate 3D nodes (2b1, 2b2)—this creates slices orientated as shown in Figure 9-11, where the "grid holes" created by the interlocking slices are "front to back" along the mesh or parallel to the **global Y axis**.

Note Where to find the nodes in Figure 9-10: Params ➤ Util ➤ Relay (2a, 2i, 2b4), Transform ➤ Euclidean ➤ Rotate 3D (2b1, 2b2), Params ➤ Input ➤ Number Slider (2b3), Vector ➤ Vector ➤ Unit Y (2b4), Maths ➤ Util ➤ Pi (2b5), and Maths ➤ Operators ➤ Addition (2b5).

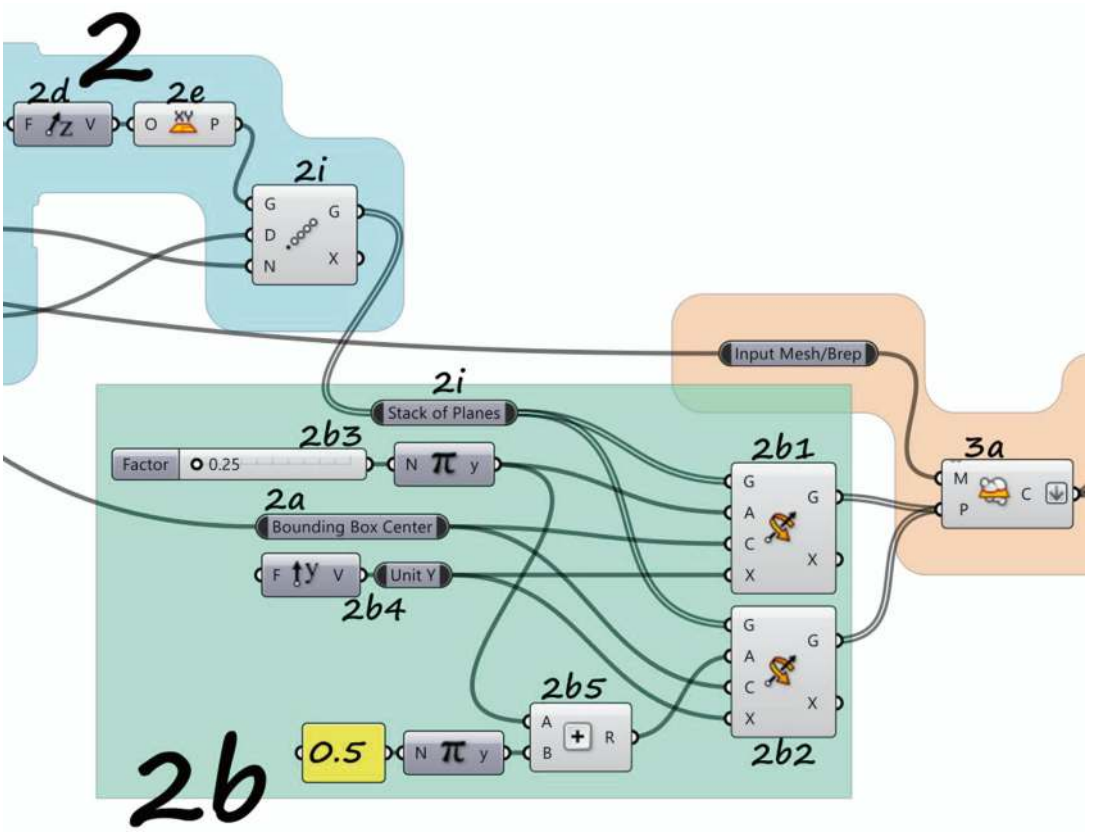


Figure 9-10. *Inserting nodes (marked 2b) after the initial stack of planes to copy and edit them into two sets of planes with rotations relative to one another. Two "Rotate 3D" nodes (2b1 and 2b2) are used to copy and rotate the stack of planes, with 0.25 Pi (2b3) as the angle (A) for the first node (2b1) and 0.25 Pi + 0.5 Pi (2b5) as the angle (A) for the second node (2b2). The rotations are about the Y axis (2b4) and centered at the input model's bounding box center (2a), with the first set of planes rotated 180*0.25 = 45 degrees and the second set an additional 90 degrees relative to the first set (for a total of 135 degrees)*

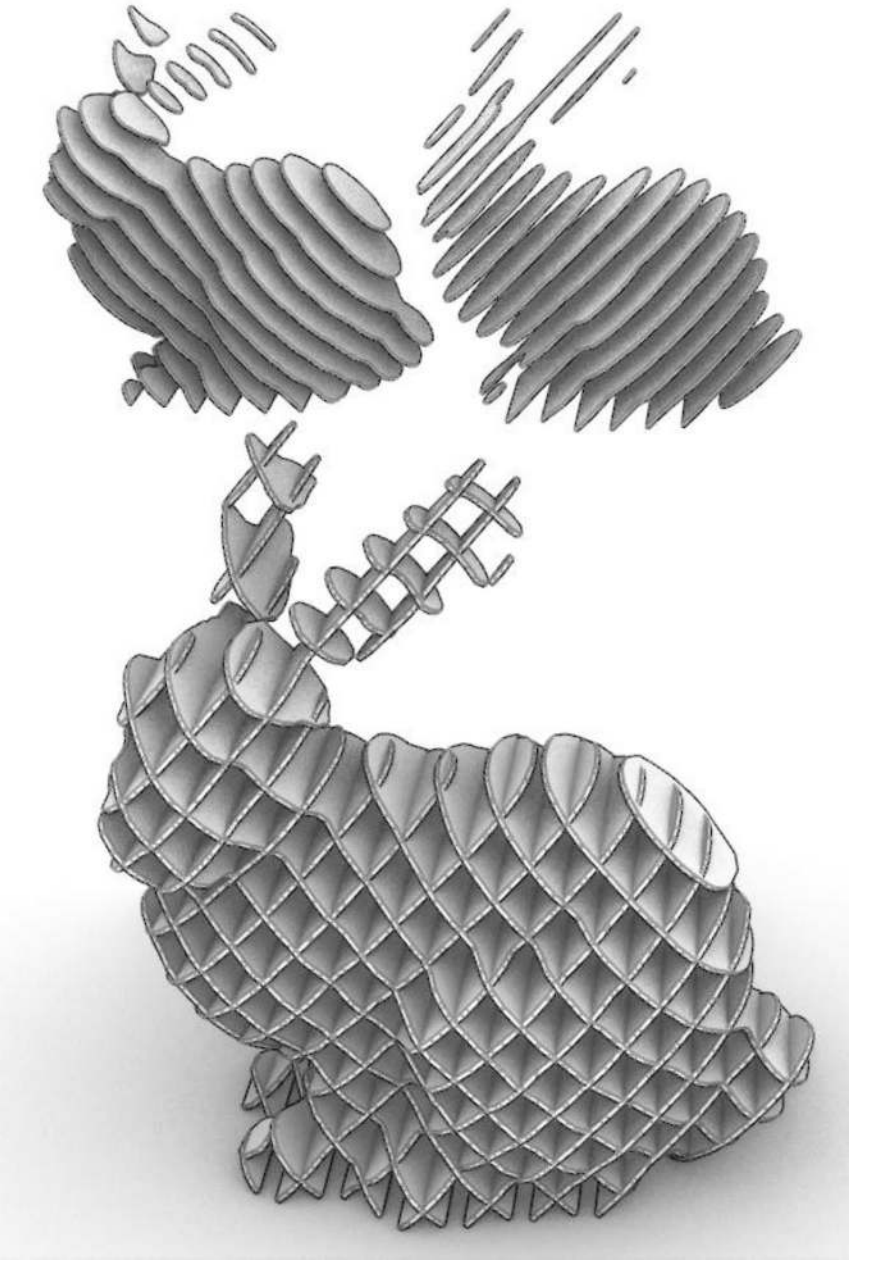


Figure 9-11. *Slices produced by stage 2b in Figure 9-10 and what they look like assembled. Top left and right depict the slices made from the two sets of planes output by the Rotate 3D nodes (2b1, 2b2) in Figure 9-10. The bottom shows the final sliceform after the two sets of slices have been extruded and combined*

Adjusting Inter-Slice Distances and Angles to Create Variations

We can create some interesting variations by changing the three parameters highlighted in Figure 9-12:

1. *The Number Slider (2f) controlling the length of the Unit Z vector as the direction (D) input for Linear Array (2i)*: This changes the distance between slices.
2. *The Number Slider and Pi node combo (2b3) that controls the angle (A) input for the first Rotate 3D node (2b1)*: This changes the rotation of the first set of slices.
3. *The Panel-Pi-Addition node sequence (2b5) that controls the angle (A) input for the second Rotate 3D node (2b2)*: This changes the rotation of the second set of slices, relative to the first set.

Table 9-1 summarizes three sets of parameter values which produce the sliceform variations shown in Figure 9-13 (labeled 1, 2, and 3). Both 1 and 3 have interlocking slices that are perpendicular, which resemble a squared grid when viewed from the front. Notably, 1 has the first set of slices rotated 45 degrees and the second set an additional 90 degrees, therefore producing a diamond grid. The inter-slice distance decreases from 1, 2, to 3, creating a finer and finer grid. Moreover, 2 shows the effect of offsetting the two sets of slices at an angle other than 90 degrees.

There is a fourth parameter that you could experiment with—in Figures 9-10 and 9-12, both the Rotate 3D nodes (2b1, 2b2) use Unit Y as the axis of rotation (X input, 2b4), creating the "grid tunnels" parallel to the global Y axis and "grid holes" from front to back of the mesh (Figure 9-13). Try changing 2b4 to use Unit X as the X input for Rotate 3D instead and see what effects you get.

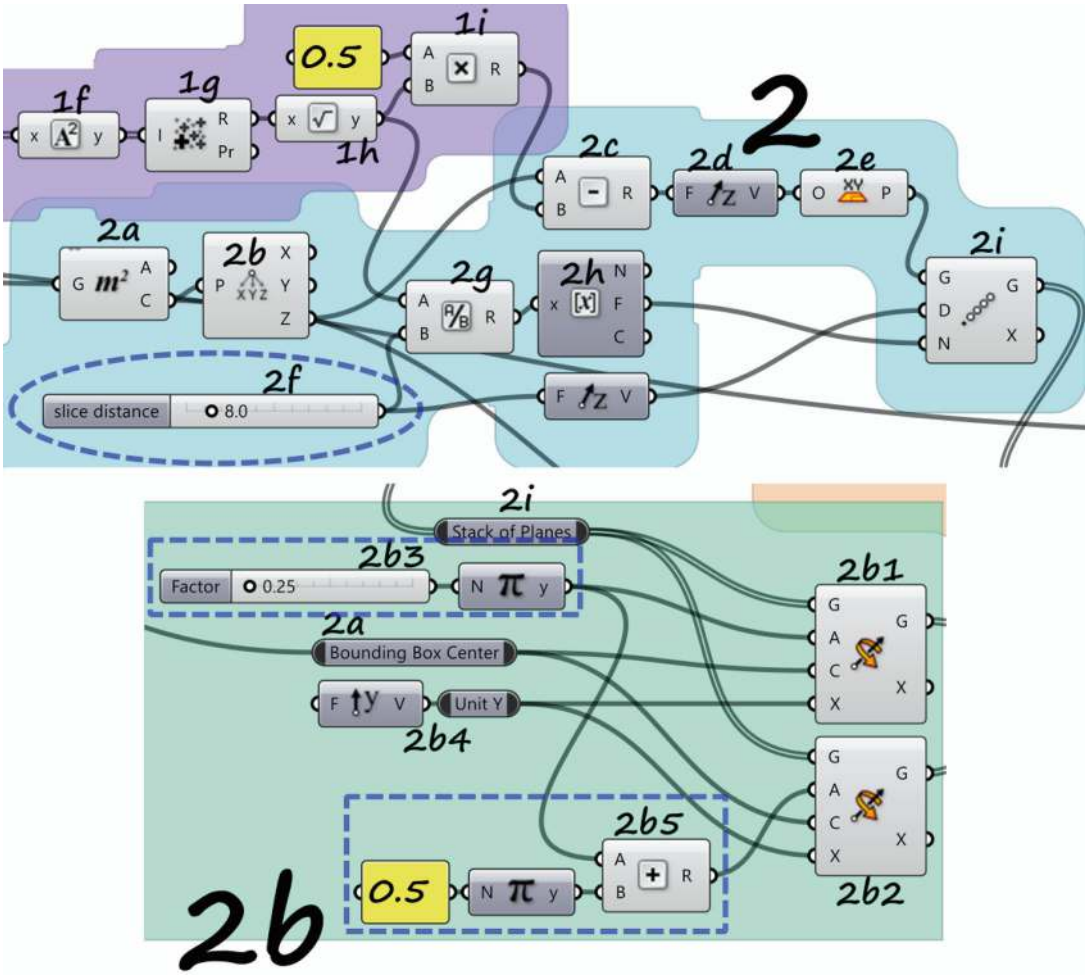


Figure 9-12. *Three parameters to adjust for variations: (1) Number Slider (2f) that controls the inter-slice spacing (this refers to the spacing between slices before they are extruded). (2) The angle to rotate the first set of planes for slicing (2b3). (3) The angle between the two sets of planes for slicing, which is typically 90 degrees, but doesn't have to be*

Table 9-1. *Parameter values that produce the three sliceform variations shown in Figure 9-13 based on the Stanford bunny mesh*

Node(s) Name	Node(s) Type	Purpose	Value (1 in Figure 9-13)	Value (2 in Figure 9-13)	Value (3 in Figure 9-13)
2f, Figure 9-4	Number Slider	Inter-slice spacing (without thickness)	8.0	5.0	3.0
2b3, Figure 9-10	Number Slider + Pi	Angle to rotate first set of slices	0.25 Pi	0.6 Pi	0.0
2b5, Figure 9-10	Panel + Pi + Addition	Angle to rotate second set of slices	0.75 Pi	0.8 Pi	0.5 Pi

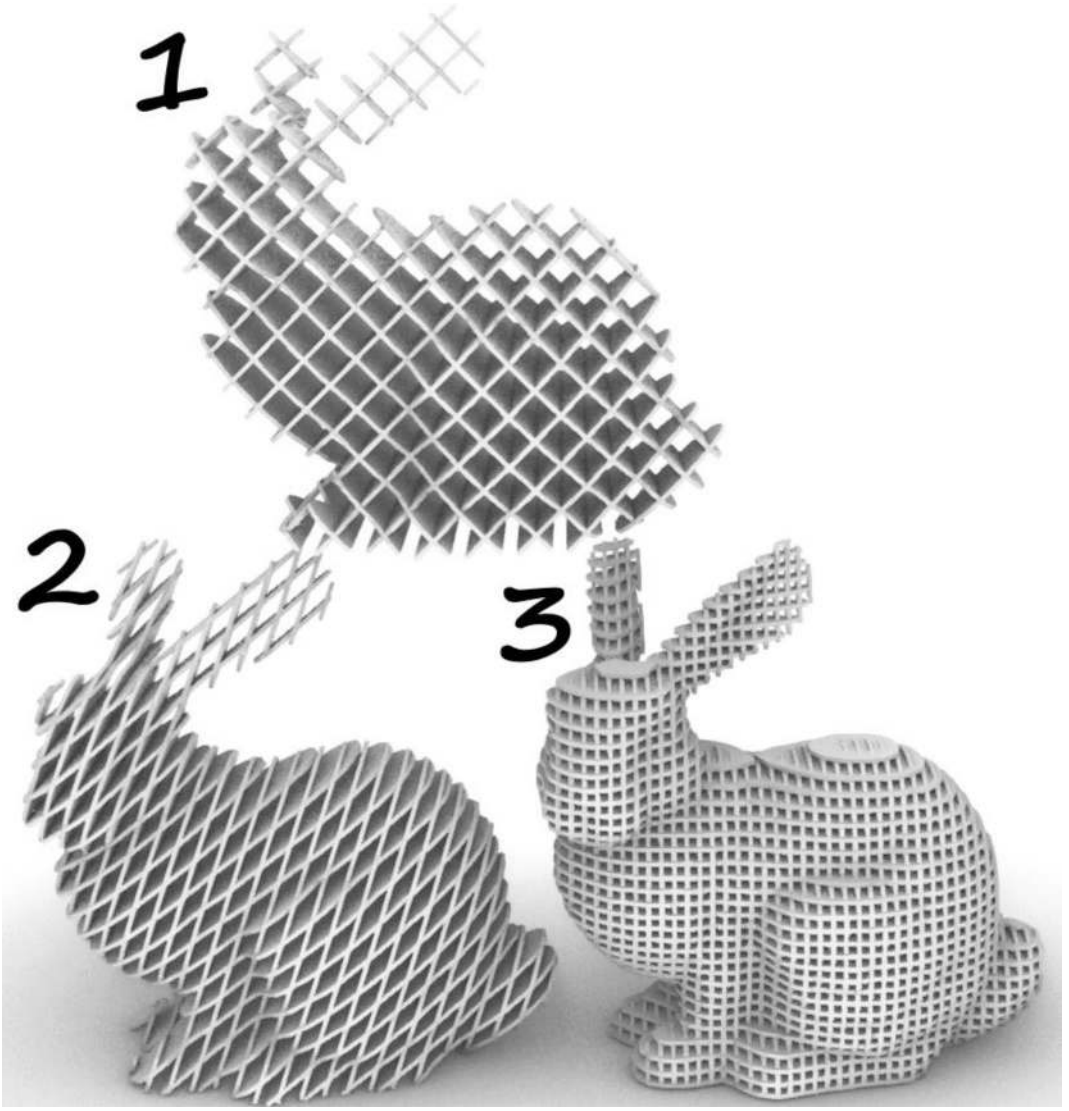


Figure 9-13. *Three sliceform variations based on the Stanford bunny mesh. (1) Inter-slice distance = 8.0, first set of slices rotated 0.25 Pi (45 deg), and second set rotated 0.75 Pi (135 deg). (2) Inter-slice distance = 5.0, first set rotated 0.6 Pi, and second set rotated 0.8 Pi. (3) Inter-slice distance = 3.0, first set of slices unrotated (angle = 0 Pi), and second set rotated 0.5 Pi*

Using Breps As Input Instead of Meshes for Sliceform Generation

So far, we've used a mesh as the input geometry for our sliceform generator (Figures 9-3 to 9-10). Since all the nodes we've used operate on both meshes and breps, we can use either as input. You can try this by replacing the Mesh node (1a) in Figure 9-3 with a Sphere node, as shown in Figure 9-14, and use a Number Slider to control its radius (R input). With this modification, the three sets of parameter values from Table 9-1 will produce the three sphere sliceform variations shown in Figure 9-15.

Note Where to find the nodes in Figure 9-14: Surface ➤ Primitive ➤ Sphere (1a).

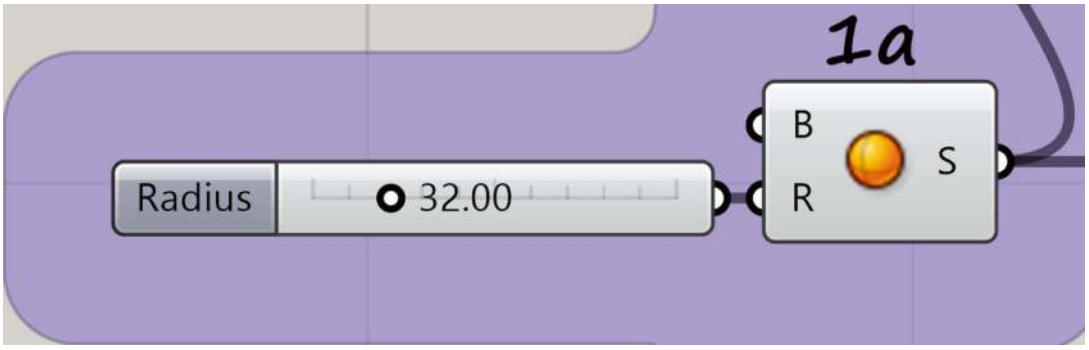


Figure 9-14. *Swapping out the Stanford bunny mesh with a Sphere (1a) Brep whose radius is parametrically controlled by a Number Slider*

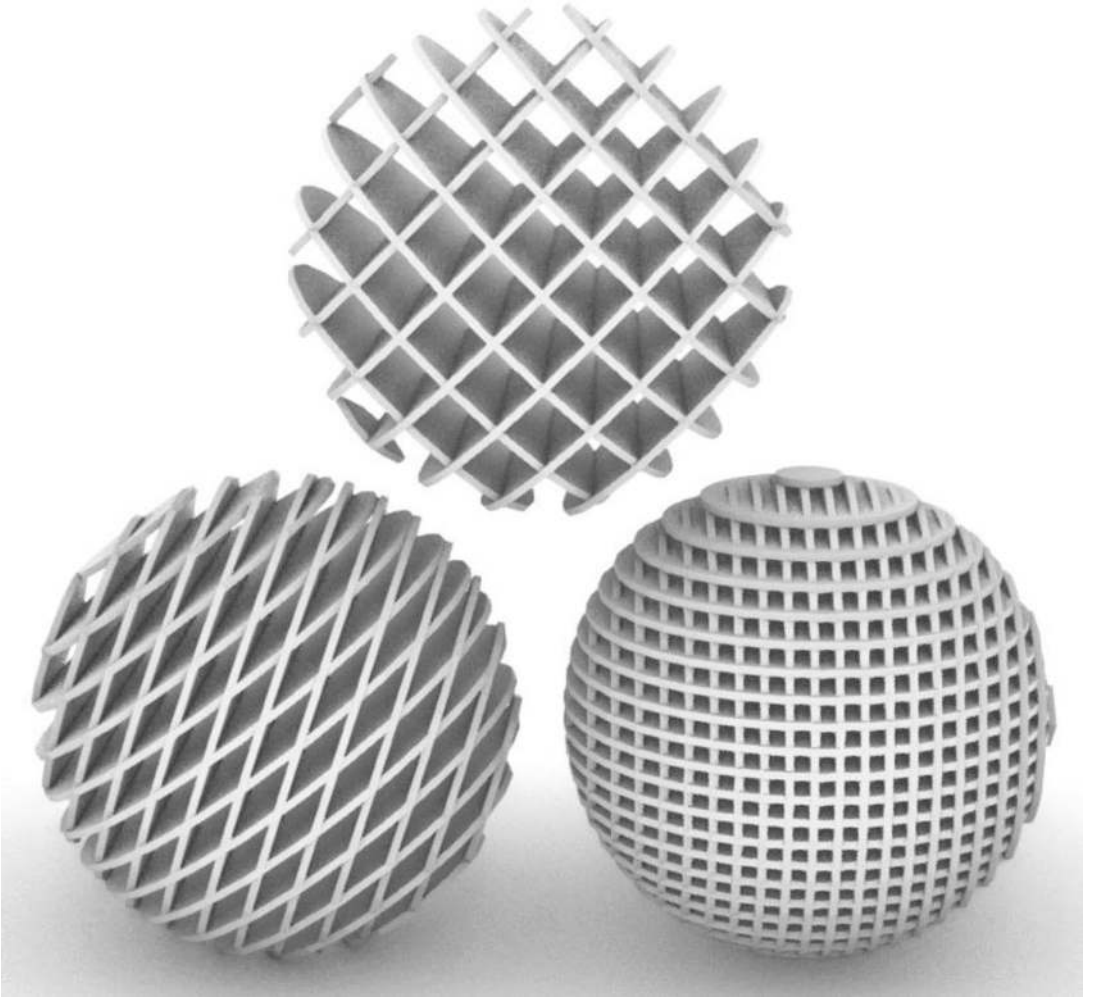


Figure 9-15. *Three sliceform variations based on a Sphere Brep with radius = 32. The parameter values used to produce the variations (inter-slice distance and angles for rotating the two sets of slices) are identical to those listed in Table 9-1 for the bunny variations.*

Generating Hollow Sliceforms

Let's see how we can modify the sliceform generator (Figures 9-3 to 9-10 and 9-14) further to create hollow versions of the sliceforms. We'll look at how to do this with meshes first, followed by Breps.

Hollow Sliceforms from Input Meshes

To create a hollow sliceform, we'll preprocess the input mesh to create a hollowed version such that it is approximately equal distance everywhere from the cavity to the mesh surface. The group of nodes marked "0-Mesh" in Figure 9-16 implements this process.

We start by linking the Mesh node (0a, formally 1a in Figure 9-3) to Unify Mesh (0b) to ensure its normals are consistent—this is important since the directions of normals are what Rhino uses to tell what is on the outside versus inside of the mesh. We then link Unify Mesh (0b) to Quad Remesh (0c) to turn its faces into quads and reduce the polygon count to a reasonable level. A Quad Remesh Settings node (0d) with default parameter values is linked to the S input for Quad Remesh (0c) to specify its various settings.

Next, we link 0c to Weaverbird's Offset Mesh (0e) to create a scaled copy of the mesh with equal distance everywhere to the original. By using a Number Slider set to a positive value (3.0 as shown) as the distance (D) input, Weaverbird's Offset Mesh (0e) will create a *scaled-up* copy (using a negative distance will create a scaled-down copy).

We can now create the hollowed-out mesh with Mesh Difference (0f), using the scaled-up mesh (0e) as A input and the original-sized mesh (0c) as B input. The last step is to pass the mesh through Unify Mesh (0g) again to ensure the normals are consistent after the "cutting."

Note Where to find the nodes in Figure 9-16: Params ➤ Geometry ➤ Mesh (0a), Mesh ➤ Util ➤ Unify Mesh (0b), Mesh ➤ Triangulation ➤ Quad Remesh (0c), Mesh ➤ Triangulation ➤ Quad Remesh Settings (0d), ***Wb ➤ Transform ➤ Weaverbird's Offset Mesh (0e)***, Intersect ➤ Shape ➤ Mesh Difference (0f), and Mesh ➤ Util ➤ Unify Mesh (0g).

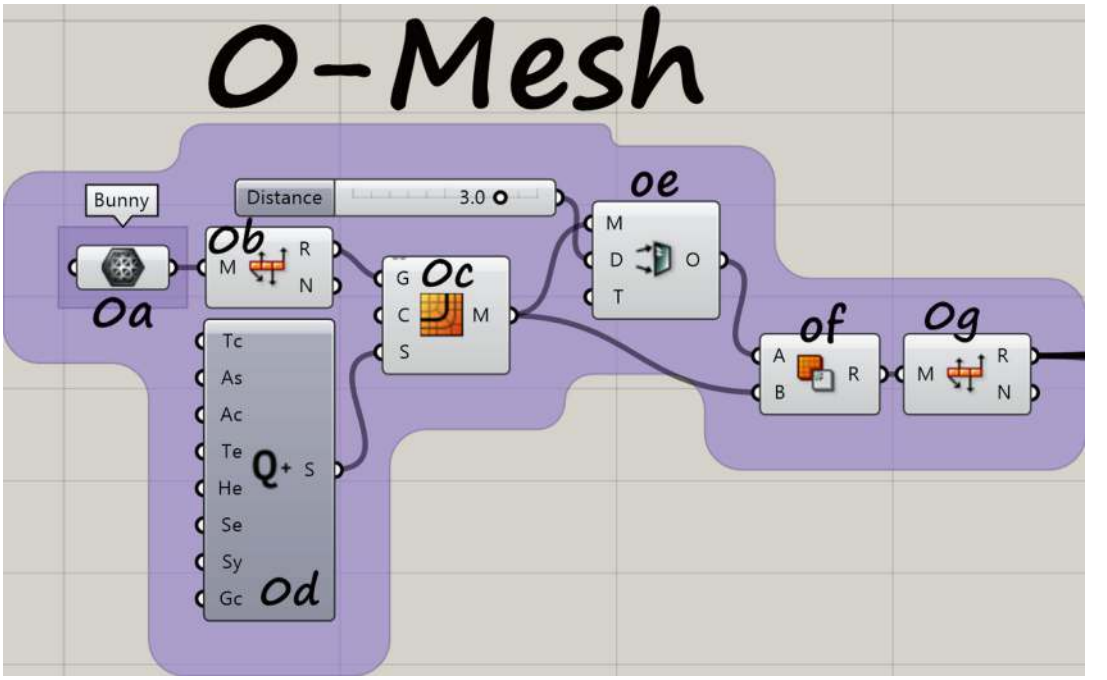


Figure 9-16. *Creating a hollow version of the bunny mesh (0a) with Weaverbird's Offset Mesh (0e) before using it as input for the sliceform generator*

Table 9-2 shows three sets of parameter values for inter-slice distance and angles for rotating the two sets of interlocking slices, which produce the three hollow sliceform variations in Figure 9-17.

Table 9-2. *Parameter values that produce the three hollow sliceform variations shown in Figure 9-17*

Node(s) Name	Node(s) Type	Purpose	Value (Top)	Value (Bottom Left)	Value (Bottom Right)
2f, Figure 9-4	Number Slider	Inter-slice spacing (without thickness)	8.0	10.0	6.0
2b3, Figure 9-10	Number Slider + Pi	Angle to rotate first set of slices	0.25 Pi	0.6 Pi	0.0
2b5, Figure 9-10	Panel + Pi + Addition	Angle to rotate second set of slices	0.75 Pi	0.8 Pi	0.5 Pi

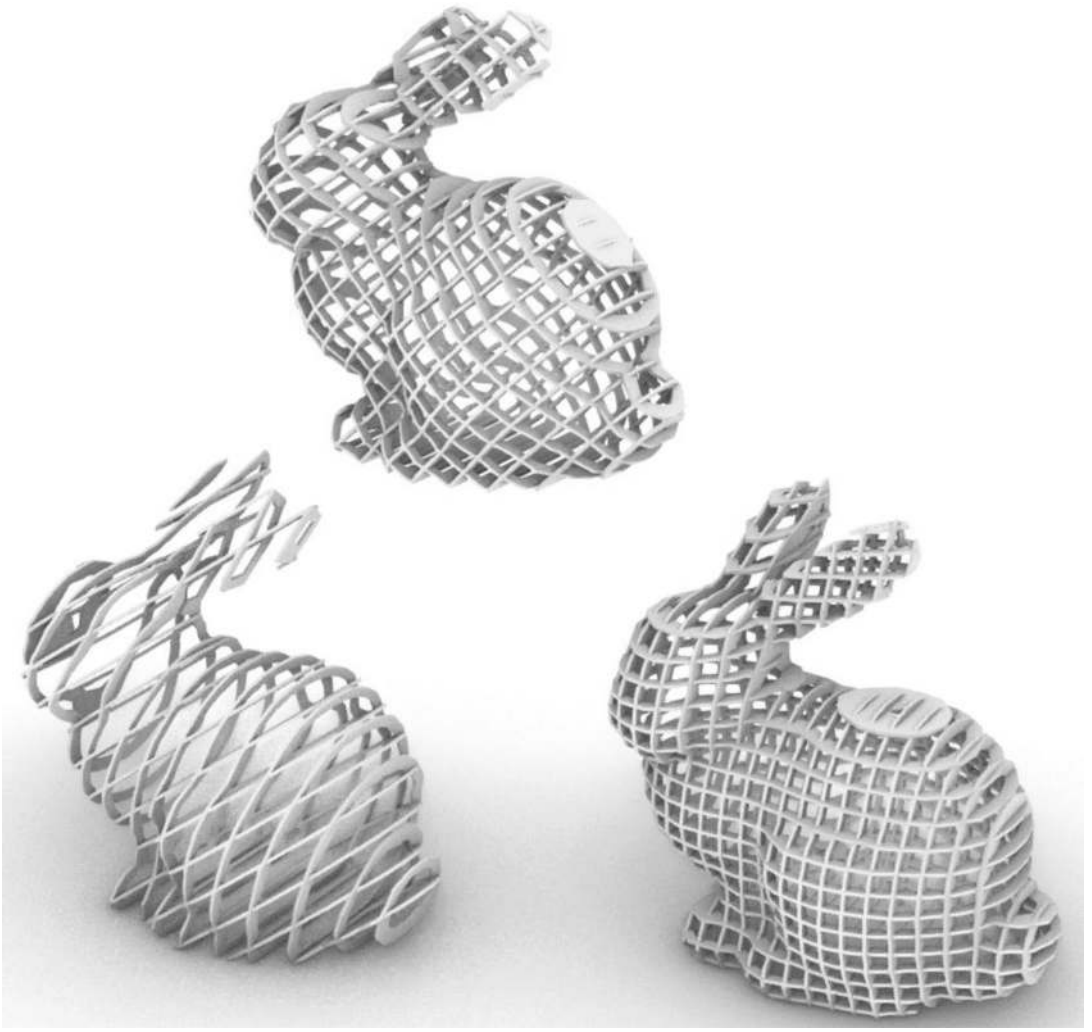


Figure 9-17. *Three hollow sliceform variations created by cutting an interior hole in a scaled copy of the bunny mesh with Weaverbird's Offset Mesh (Figure 9-16) before using it as input for the sliceform generator (Figures 9-3 to 9-10). Parameter values for inter-slice distance and angles for slicing used for these variations are summarized in Table 9-2*

Hollow Sliceforms from Input Breps

Just like meshes, we can hollow out Breps before using them as inputs for the sliceform generator to create hollow sliceforms. Figure 9-18 shows this modification as the group of nodes "0-Brep," which replaces 1a in Figure 9-14.

We'll start by increasing the radius of the Sphere (0a) from 32 to 60, to make the hollow effect easier to see. We then use Scale (0c) to create a copy of the sphere that is 80% its size, by using a Number Slider set to 0.8 as the F input. After that, we convert both the original (0a) and the scaled-down (0c) Breps to meshes using Mesh Brep (0b, 0d). Finally, we create the hollow mesh by taking the Mesh Difference (0e) between the two, using the original sized mesh as the A input and the scaled mesh as the B input.

Note Where to find the nodes in Figure 9-18: Surface ➤ Primitive ➤ Sphere (0a), Mesh ➤ Util ➤ Mesh Brep (0b), Transform ➤ Affine ➤ Scale (0c), Mesh ➤ Util ➤ Mesh Brep (0d), and Intersect ➤ Shape ➤ Mesh Difference (0e).

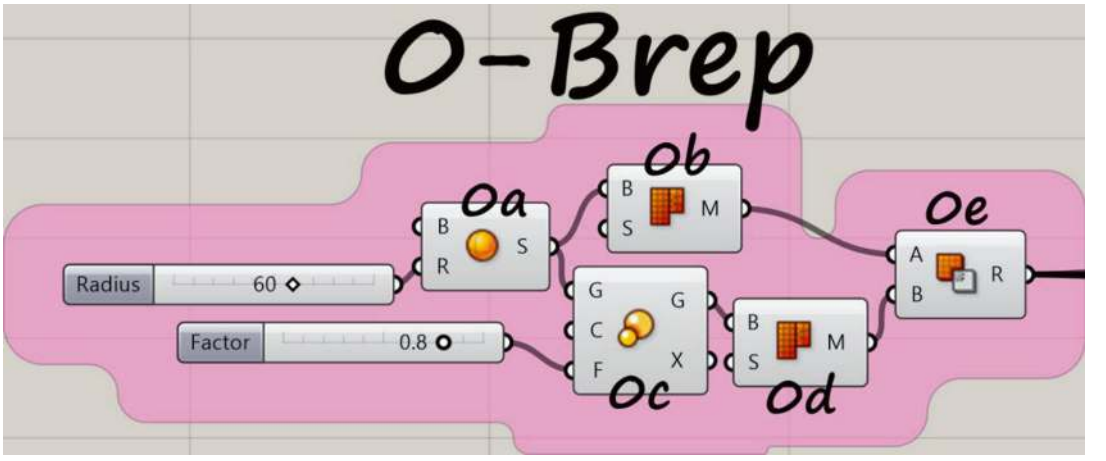


Figure 9-18. *To create a hollow sliceform from a Brep, we create a scaled down copy of the Brep (0c), convert both the original and the copy to meshes (0b, 0d), and use Mesh Difference (0e) to obtain the hollowed out mesh, before using that as input for the sliceform generator (Figures 9-3 to 9-10 and 9-14)*

Table 9-3 lists three sets of parameter values for inter-slice distance and angles for rotating slices that are used to produce the hollow sphere variations in Figure 9-19. An additional parameter you could experiment with is the Number Slider linked to the Scale node (0c) in Figure 9-18, which controls the size of the portion carved out of the sphere, therefore the width of the sliced "rings."

Table 9-3. *Parameter values used to produce the three hollow sphere variations shown in Figure 9-19*

Node(s) Name	Node(s) Type	Purpose	Value (Top)	Value (Bottom Left)	Value (Bottom Right)
2f, Figure 9-4	Number Slider	Inter-slice spacing (without thickness)	12.0	15.0	9.0
2b3, Figure 9-10	Number Slider + Pi	Angle to rotate first set of slices	0.25 Pi	0.6 Pi	0.0
2b5, Figure 9-10	Panel + Pi + Addition	Angle to rotate second set of slices	0.75 Pi	0.8 Pi	0.5 Pi

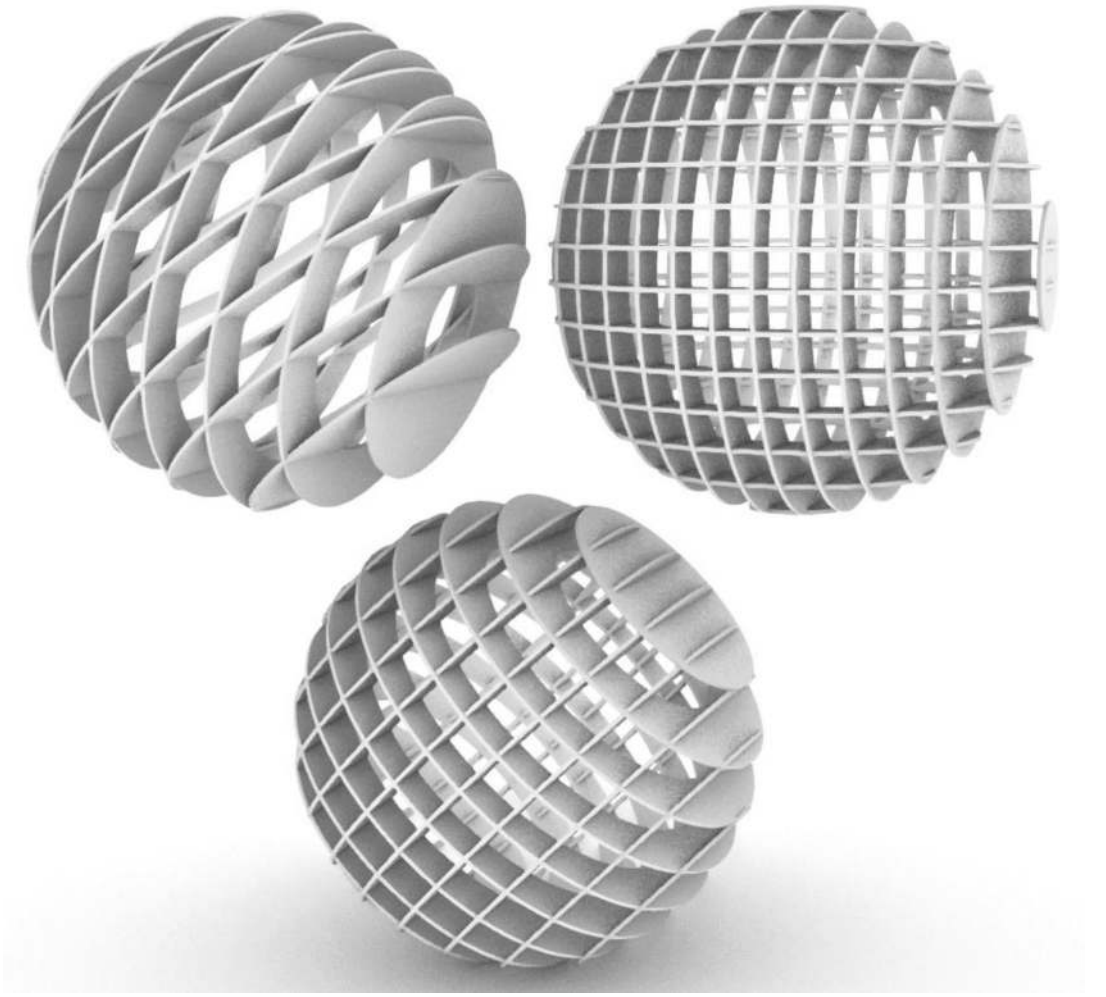


Figure 9-19. *Three hollow sliceform variations created by cutting an interior hole in a sphere Brep (Figure 9-18) before using it as input for the sliceform generator (Figure 9-3 to 9-10). The parameter values for inter-slice distance and slice rotations that produced these variations are summarized in Table 9-3*

Make Mock Hollow Sliceforms from Interlocking Spiral Ribbons

Instead of making a "true" sliceform consisting of two sets of interlocking slices, we can make a "mock" sliceform by turning an input Brep into two spiral ribbons that are orthogonal to one another, as shown in Figure 9-21. We'll explore two examples of this in this section.

Interlocking Spherical Ribbons

We begin with the simplest case and make a mock sphere sliceform. We start by making a trimmed surface in the shape of a flat donut using the following steps (Figure 9-20): use Circle (1a) to create a circular curve, Scale (1b) to create a copy 1.25 times larger, and Boundary Surfaces (1c) to fill in the space between the two curves to create the surface.

Next, we use Twist (1g) to turn the flat ring (1c) into a 3D spiral. We create a line segment that starts at the origin (0, 0, 0) and goes one unit along +X as the axis (X) for Twist, using Construct Point (1e) with default parameters to create the starting point (S) at (0, 0, 0) and Unit X (1f) as the direction (D) for Line SDL (1d). To make the spiral into a ball shape, we set the A input for Twist (1g) to 2Pi (1h) and the Infinite (I) input to True with a Boolean Toggle (1i).

Lastly, we compose the 3D spiral with a copy of itself rotated 90 degrees to create a mock sliceform. Linking Twist (1g) to Rotate (1j) with default parameters rotates the spiral 90 degrees, and Shift-dragging Twist (1g) and Rotate (1j) to Surface (1k) combines them.

Figure 9-21 shows four intermediate steps for constructing the mock sliceform. That is, 1 is the flat ring surface (1c) made from a circular curve and its scaled copy, 2 is the spiral created by linking 1c to Twist, 3 is the spiral rotated 90 degrees, and 4 is the spiral and its copy combined into a mock sliceform.

Note Where to find the nodes in Figure 9-20: Curve ➤ Primitive ➤ Circle (1a), Transform ➤ Affine ➤ Scale (1b), Surface ➤ Freeform ➤ Boundary Surfaces (1c), Curve ➤ Primitive ➤ Line SDL (1d), Vector ➤ Point ➤ Construct Point (1e), Vector ➤ Vector ➤ Unit X (1f), Transform ➤ Morph ➤ Twist (1g), Maths ➤ Util ➤ Pi (1h), Params ➤ Input ➤ Boolean Toggle (1i), Transform ➤ Euclidean ➤ Rotate (1j), and Params ➤ Geometry ➤ Surface (1k).

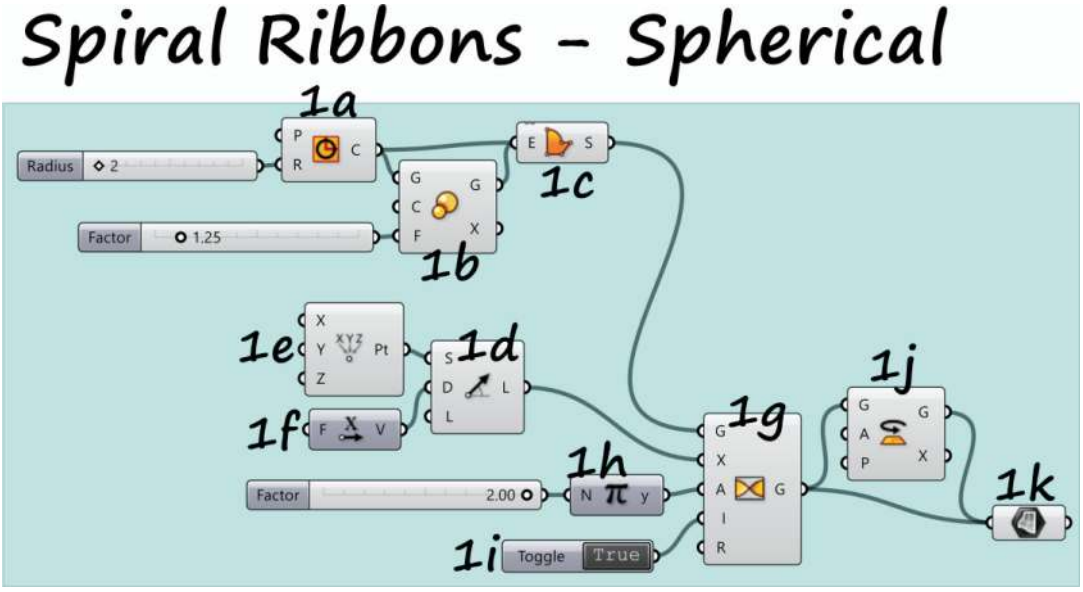


Figure 9-20. *Using the Twist node (1g) to create a spiral out of a surface in the shape of a circular ribbon and composing the spiral with a copy of itself rotated 90 degrees to create a "mock sliceform"*

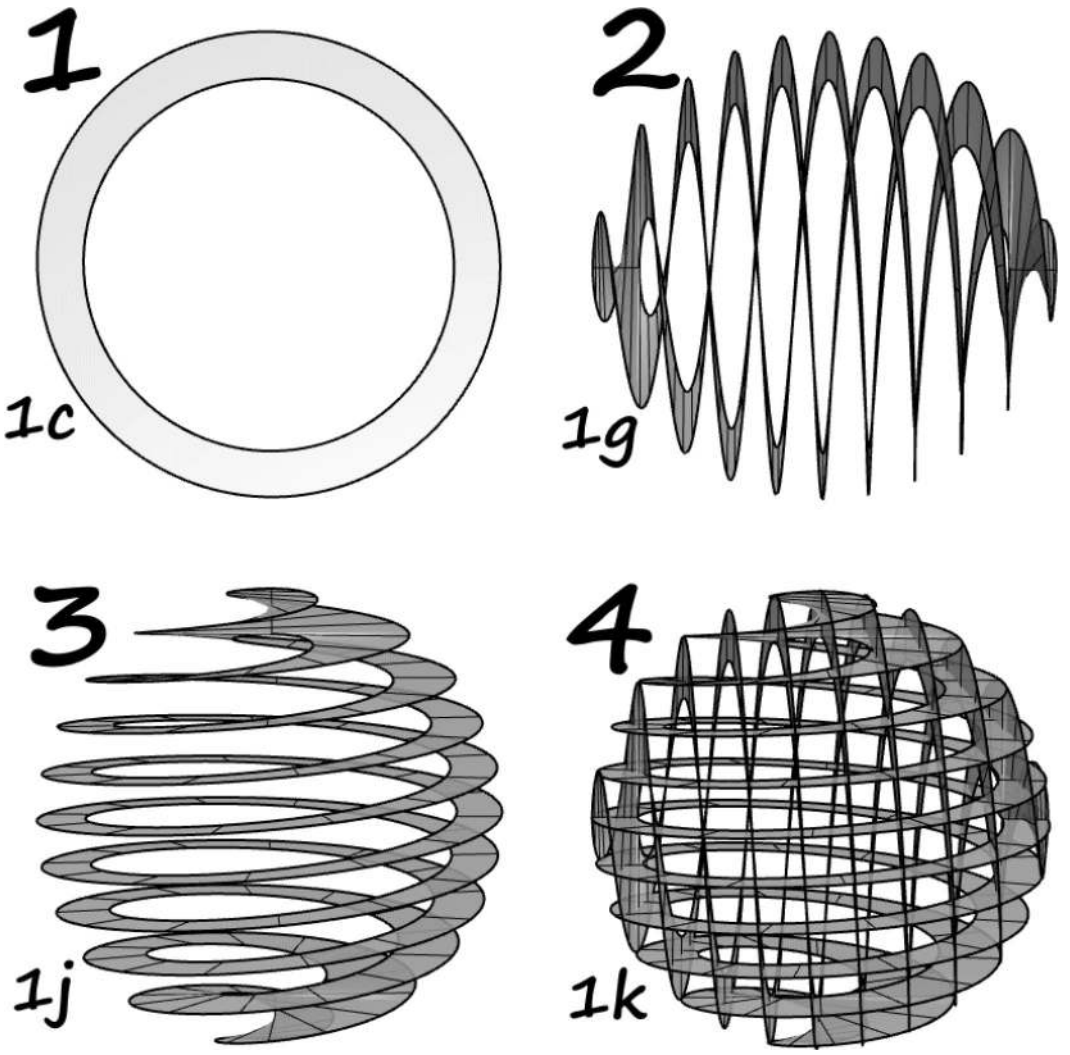


Figure 9-21. *Outputs of nodes in Figure 9-20, illustrating how the spherical "mock sliceform" is assembled. The visuals labeled 1 to 4 correspond to outputs of the nodes 1c, 1g, 1j, and 1k from Figure 9-20, respectively. (1) A surface shaped like a circular band is created using a circular curve and a scaled-up copy as input for Boundary Surfaces (1c). (2) The Twist node (1g) is used to turn the circular band into a spiral. (3) A rotated copy of the spiral (1j) is created next. (4) The original spiral and its rotated copy are Shift-dragged to a Surface node (1k) to combine*

Interlocking Polygonal Ribbons

We can make a variation of the mock sliceform by replacing the Circle node (1a) in Figure 9-20 with Polygon (1a), as shown in Figure 9-22. We will again use a Number Slider to control its radius and add a second Number Slider to set its Segments (S) to 4.

Figure 9-23 shows the intermediate node outputs of Figure 9-22. That is, 1 is the diamond-shaped band created with Boundary Surfaces (1c) from the square curve (1a) and its scaled-up copy (1b), 2 is the spiral created using Twist (1g) on the diamond band, 3 is the spiral rotated 90 degrees (1j), and 4 is the two spirals (1g and 1j) Shift-dragged to a Surface node (1k) to combine into the mock sliceform.

Note Where to find the nodes in Figure 9-22: Curve ➤ Primitive ➤ Polygon (1a), Transform ➤ Affine ➤ Scale (1b), Surface ➤ Freeform ➤ Boundary Surfaces (1c), Curve ➤ Primitive ➤ Line SDL (1d), Vector ➤ Point ➤ Construct Point (1e), Vector ➤ Vector ➤ Unit X (1f), Transform ➤ Morph ➤ Twist (1g), Maths ➤ Util ➤ Pi (1h), Params ➤ Input ➤ Boolean Toggle (1i), Transform ➤ Euclidean ➤ Rotate (1j), and Params ➤ Geometry ➤ Surface (1k).

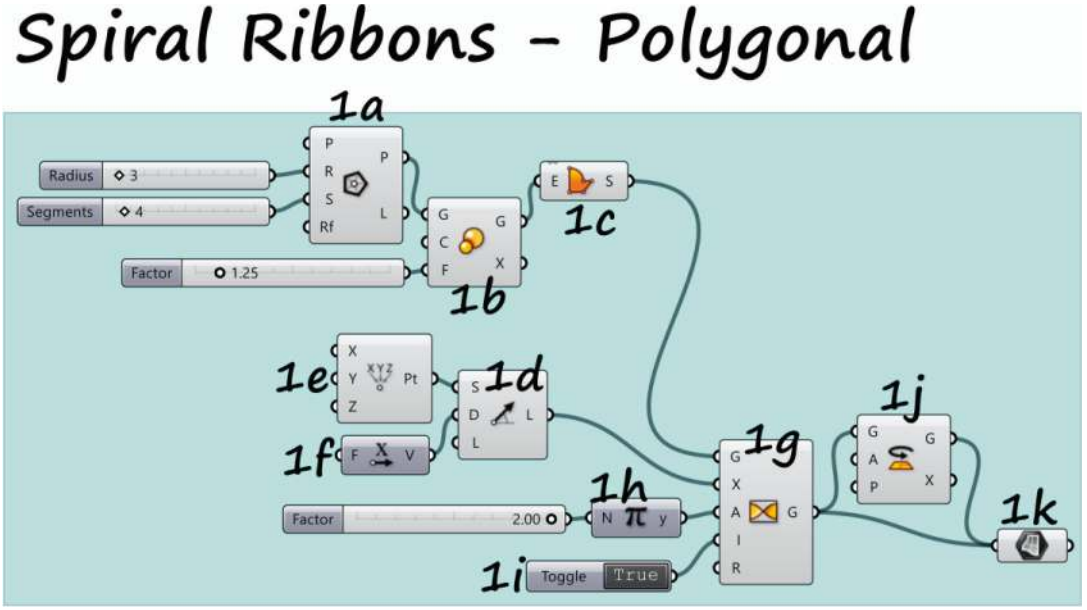


Figure 9-22. *We can make a variation of the "mock sliceform" by swapping out the Circle node with Polygon (1a) to create a square-shaped band instead of a circular one, before turning it into a spiral with Twist (1g)*

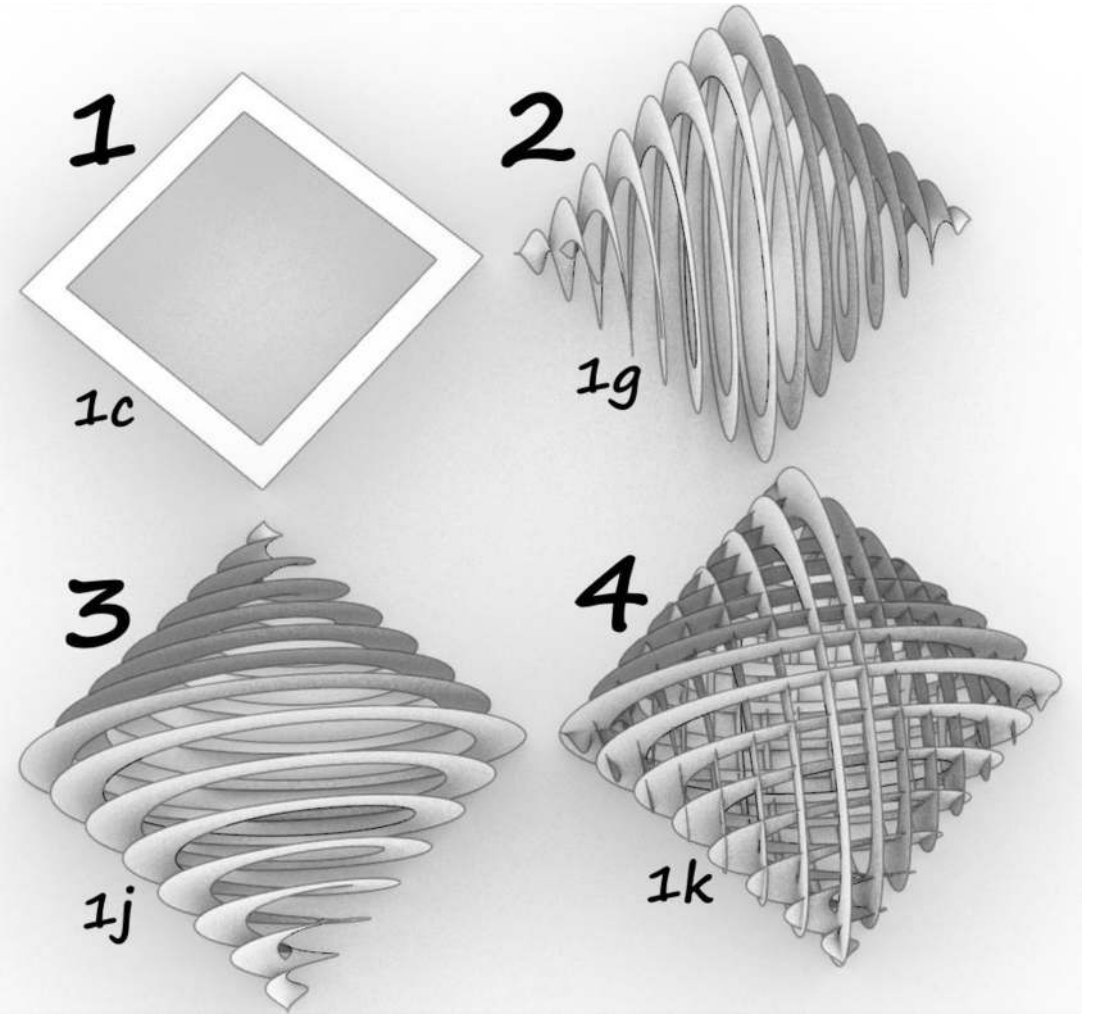


Figure 9-23. *Outputs of nodes in Figure 9-22, showing how the polygonal "mock sliceform" is assembled. The visuals labeled 1 to 4 correspond to outputs of the nodes 1c, 1g, 1j, and 1k from Figure 9-22, respectively. (1) A surface shaped like a diamond ribbon is created by using a Polygon curve and a scaled-up copy as inputs for Boundary Surfaces (1c). (2) The Twist node (1g) is used to turn the ribbon into a spiral. (3) A rotated copy of the spiral (1j) is created next. (4) The original spiral and its rotated copy are Shift-dragged to a Surface node (1k) to combine*

Summary

A *sliceform* is a type of geometry created by interlocking two sets of cross-sectional slices of a model. The two sets of slices are cut at different angles, often 90 degrees from each other. Thanks to their unique aesthetics and options to be made rigid or collapsible, sliceforms are suitable for a wide range of applications, both permanent and deployable. The flat slices also make them relatively easy to manufacture and transport.

In the first part of the chapter, you built a basic sliceform generator from the ground up that slices an input mesh at a single fixed angle. You then expanded this design to create two interlocking sets of slices, whose angles can be parametrically varied. After that, you learned how to carve out a mesh or a Brep before using it as input for the generator to create hollow sliceforms. Lastly, you learned how to create a 3D spiral out of a flat, "ring-like" surface using the Twist node and compose a spiral with a rotated copy to create hollow "mock sliceforms."

References

Lupiani, Isabel. *Procedural Content Generation for Games: Automate 3D Asset and Environment Creation with Blender Python and Geometry Nodes*. First Edition, Apress, 2025.

Wikipedia. "Stanford bunny." *Wikipedia*, 29, August 2025, CC BY-SA 4.0, https://en.wikipedia.org/wiki/Stanford_bunny

CHAPTER 10

Generating Models from Images

In this chapter, we'll discuss two approaches for generating tiles from images—the first using the Image Sampler node and the second using Import Image in combination with Python 3. Both approaches will create relief surfaces whose heights are proportional to the intensities of their corresponding image pixels. After that, we'll look at three applications for the generated image tiles—projecting tiles onto other surfaces as decorative elements, creating 3D printable molds, and 3D printed lithophanes.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch10` folder, where you'll find `*.gh` files named after the figures in which they appear, along with a short description. For example, `/Ch10/Figure10-1_to_10-9_TileFromImage_ImageSampler.gh`. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, build your own node graphs by following the steps described in the text or start with the downloaded `*.gh` files and modify their parameters to experiment.

Attribution to Image Used As Inputs for Node Graphs

The image used as inputs for the node graphs throughout this chapter (`/Ch10/succulent.jpg`) is taken by the author.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_10

Generating Tiles from Images Using the "Image Sampler" Node

In this section, we'll build a node graph to generate a 3D printable tile by sampling an input image using the Image Sampler node. The graph encompasses four stages as depicted in Figure 10-1. Stage 1 creates a surface from points whose Z coordinates are proportional to the intensities of their corresponding pixels in the input image. Stage 2 generates a flat bottom for the tile. Stage 3 creates the four vertical sidewalls for the tile, and stage 4 combines the top, sides, and bottom of the tile into a single mesh.

Caution Bear in mind the larger the image, the longer it will take for Image Sampler to process it.

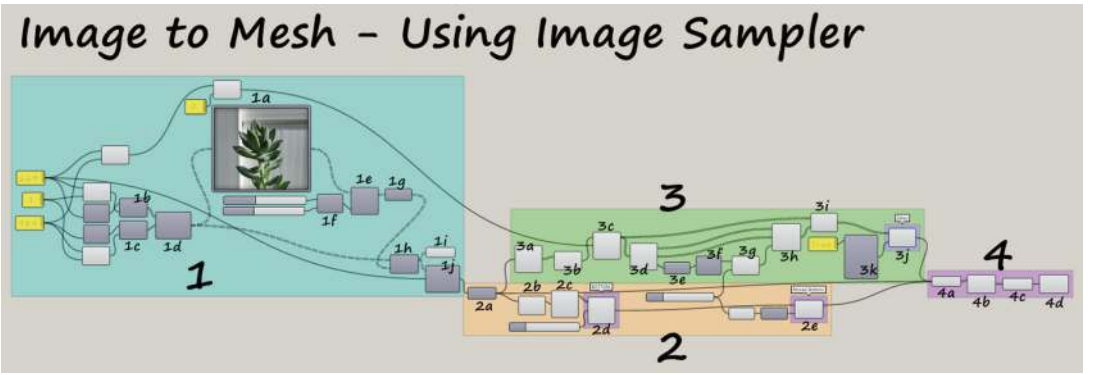


Figure 10-1. *Bird's-eye view of the node graph for generating tiles from images. Stage 1 samples an input image using the Image Sampler node and generates a surface from points whose heights are determined by the corresponding pixel intensities. Stages 2 and 3 create the bottom and sidewalls for the tile, respectively, while stage 4 combines the top, sides, and bottom into a single mesh*

Stage 1: Creating Relief Surface from Points with Heights Proportional to Pixel Intensities

Our goal in stage 1 (Figure 10-2) is to create a surface from points whose heights are proportional to the intensities of their corresponding pixels.

You start by dragging an Image Sampler node (1a) to the canvas, which appears initially as an empty square with an exclamation point. You then double-click the node to open its Settings editor (top of Figure 10-3). To load an image, click the "..." button to the right of "File path" (2 in Figure 10-3) to invoke the file open dialog, navigate to the Ch10 folder in the downloaded source files for the book, and select "succulent.jpg". Click the "Use image pixel dimensions" button (3) to auto-detect the X and Y domains of the image (4), which are 1 to 664 and 1 to 489, respectively, since the image is 664 pixels wide and 489 pixels tall. Leave the other settings at default—Channel as RGBA Colours, with "Auto update" and "Save in file" checked, to internalize the image data in the Grasshopper definition and automatically refresh when the source image file changes. Click "OK" to finish setting up the node and dismiss the dialog. Afterward, the Image Sampler node will show a preview of the loaded image, as shown in Figure 10-2.

The next step is to create input coordinates for Image Sampler to indicate which pixels to read from. Since we want to sample the entire image (664 by 489), we need an array with 664 columns and 489 rows of points, with coordinates (1, 1, 0), (1, 2, 0), ..., to read pixels at position "column 1 row 1," "column 1 row 2," and so on. If you take a closer look at the preview image inside the Image Sampler Settings dialog in Figure 10-3 (brought on by double-clicking the node), you'll notice that the column indices increase from left to right (widthwise), while the rows increase from bottom to top (height-wise). As a result, the coordinates (1, 1, 0), (1, 2, 0), ... will read the pixels one column at a time from left to right and from bottom to top within each column.

We use two Construct Domain nodes to create domains from 1 to 664 and 1 to 489, respectively. We then use the two domains as the D input for two Range nodes (1b and 1c in Figure 10-2), with their N (Steps) set to the high ends of the domains minus one. Recall that if you want to create ten values with Range (say within the domain 1 to 10), you need to set N = 10 - 1 = 9. Therefore, the first Range node (1b) has D = 1 to 664 and N = 663, which generates the numbers 1, 2, 3, ..., 664 (a total of 664 numbers). The second Range node (1c) has D = 1 to 489 and N = 488, which generates the list 1, 2, 3, ..., 489 (a total of 489 numbers).

Up to this point, we've generated separate lists of X and Y values, which are 1, 2, 3, ..., 664 and 1, 2, 3, ..., 489, respectively. We want to cross-reference them to create the coordinates (1, 1, 0), (1, 2, 0), ..., (664, 489, 0). To do so, we link the two Range nodes (1b and 1c) to the X and Y inputs of Construct Point (1d) and leave Z unconnected so it has the default value of 0—with this setup, Construct Point (1d) outputs a list based on one-to-one pairings of X and Y, which is a diagonal line and not yet what we want. To make Construct Point (1d) output a 2D matrix of points, such that each value of X is paired up with *all* values of Y, right-click X ➤ Graft. If you move the mouse cursor over 1d's output now, it should read

```
324696 locally defined values...
{0;0;0} (N = 489)
{0;0;1} (N = 489)
{0;0;2} (N = 489)
 ↓
{0;0;663} (N = 489)
```

which tells us 1d's output is a data tree with 664 branches (0 to 663), with 489 points per branch. We can link a panel to 1d's output to verify the generated point coordinates, which are

```
(1, 1, 0)
(1, 2, 0)
(1, 3, 0), ..., and so on, as expected.
```

We can now link Construct Point (1d) to Image Sampler (1a) to retrieve the pixels' RGB values. To use the RGB values to vary the heights (Z) of the relief surface that we'll build next, we need to use Remap Numbers (1e) to convert the values to between the lowest and highest points of the surface.

To do so, we use Image Sample (1a) as the Value (V) input for Remap Numbers (1e) and create a domain from 6 to 0 (with Construct Domain (1f)) as the Target (T) input, which maps the pixel values from 0–255 to 6–0, thus making the darker pixels *taller* in the surface (conversely, a Target (T) of 0–6 will make the *lighter* pixels taller).

Next, we link the remapped values to Unit Z (1g) to use as Motion (T) for the Move node (1h) to offset the points we've already made with XY pixel coordinates (1d). We can preview the point cloud by linking (1h) to a Point node (1i, right-click ➤ Flatten) as shown in Figure 10-4. When zoomed out, the point cloud resembles the image. The more you zoom in, the more the point cloud looks like a grid of points with even spacing along XY. You can see the Z variations from the side and gauge how the relief surface will contour.

The last step is to create a relief surface from the point cloud. We do so by linking the output of Move (1h) to Surface from Points (1j) and use the image width (664) directly as the U count.

Note Where to find the nodes in stage 1 of Figure 10-2: Params ➤ Input ➤ Image Sampler (1a), Sets ➤ Sequence ➤ Range (1b, 1c), Vector ➤ Point ➤ Construct Point (1d), Maths ➤ Domain ➤ Remap Numbers (1e), Maths ➤ Domain ➤ Construct Domain (1f), Vector ➤ Vector ➤ Unit Z (1g), Transform ➤ Euclidean ➤ Move (1h), Params ➤ Geometry ➤ Point (1i), and Surface ➤ Freeform ➤ Surface from Points (1j).

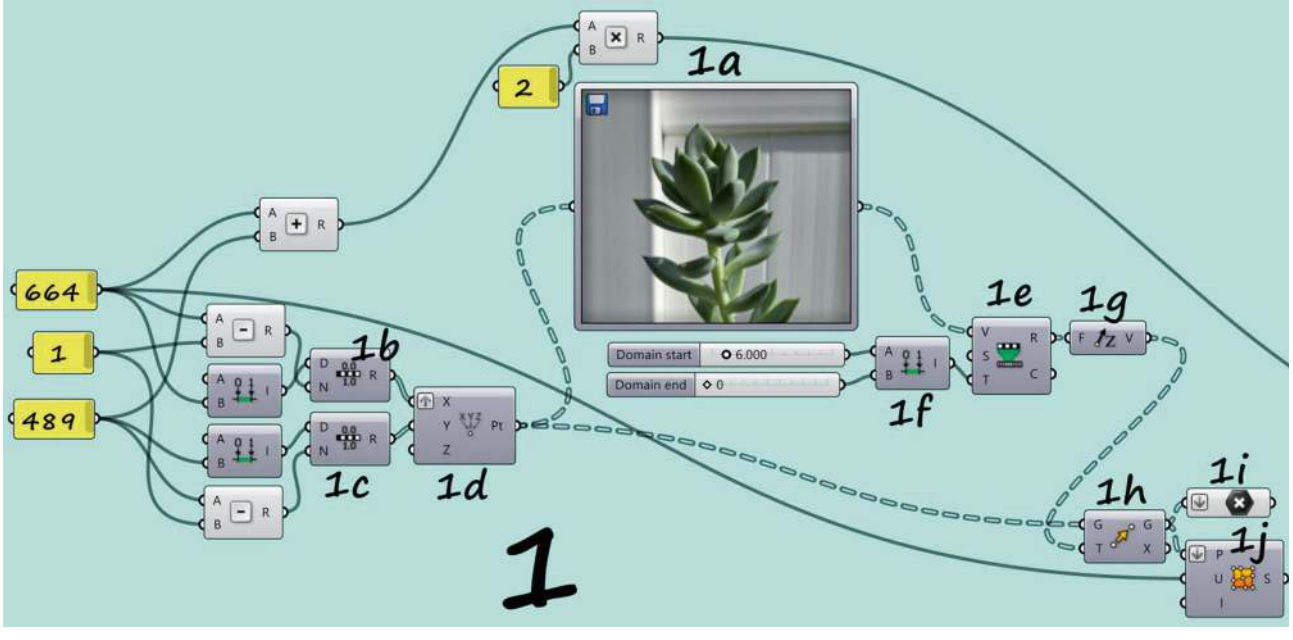


Figure 10-2. *Stage 1 of the node graph for generating a tile from image, where we use Image Sampler (1a) to read the pixel values (RGB) and Remap Numbers (1e) to convert the RGB values to a range between 6 and 0, with which to offset points along Z to construct a relief surface*



Figure 10-3. *When you first drag an Image Sampler node (1) to the canvas, it looks blank with an exclamation point in the middle. You double-click the node to open its Settings editor (top). To load an image, click the "..." button (2) to the right of File path to invoke the file open dialog and select the file. Click "Use image pixel dimensions" (3) to auto-detect the X and Y Domain of the image (4) and leave the other settings at default ("Channel" as RGBA Colours and "Auto update" and "Save in file" checked to internalize the image data in the GH definition and auto refresh when the source file changes). Finally, click "OK" to load the image*

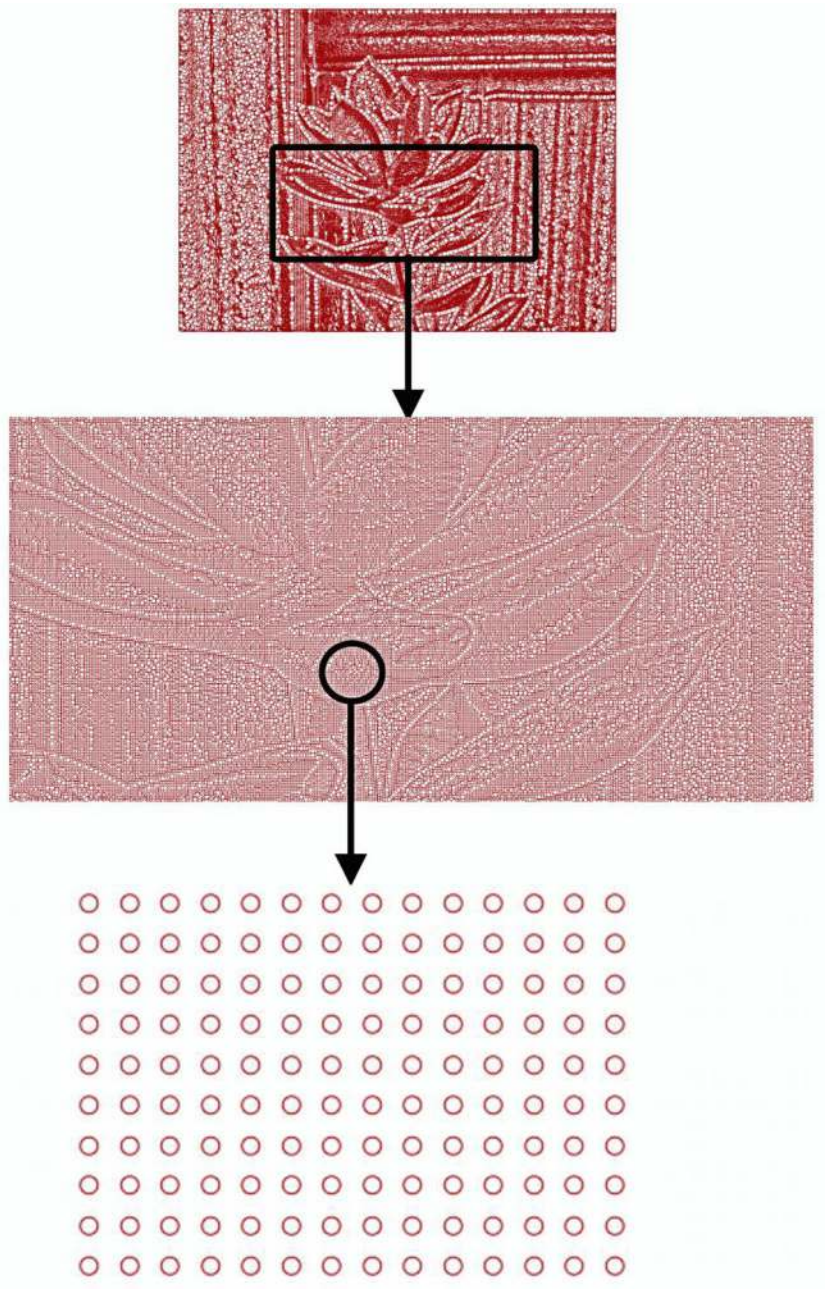


Figure 10-4. *Points created from pixel XY coordinates with Z proportional to pixel intensities (0–255 mapped to 6–0, such that darker pixels are taller). When zoomed out (top), the point cloud looks like the original image. As you zoom in more and more, the point cloud increasingly resembles a grid of points with even spacing along XY*

Stage 2: Creating a Flat Bottom for the Tile

If you imagine the relief surface from stage 1 as the top of the tile, stage 2 (Figure 10-5) creates its bottom.

We start by linking the relief surface (1j) to Flip Matrix (2a) to swap its rows and columns—this is to put its edges in order so we can start at one corner of the surface and walk all the way around, therefore easily isolating its perimeter. Next, we link Flip Matrix (2a) to Bounding Box (2b) to get the minimal box enclosing the surface, followed by Deconstruct Brep (2c) to extract its faces. The bounding box has six faces total with indices 0 to 5—we isolate the face with index 4, which is the bottom of the box, using List Item (2d) with a Number Slider dialing its Index (i) input to 4. We'll use the bounding box's bottom face as the tile's bottom.

All that's left to do is to move the bottom to the correct Z offset, which we won't know yet, since we want to allow the thickness of the tile (therefore the height of the sidewalls) to be parametrically varied. The Move node (2e) at the end of stage 2 does just that—which we'll discuss in stage 3, once the logic for varying the tile thickness is established.

Note Where to find the nodes in stage 2 of Figure 10-5: Sets ➤ Tree ➤ Flip Matrix (2a), Surface ➤ Primitive ➤ Bounding Box (2b), Surface ➤ Analysis ➤ Deconstruct Brep (2c), Sets ➤ List ➤ List Item (2d), and Transform ➤ Euclidean ➤ Move (2e).

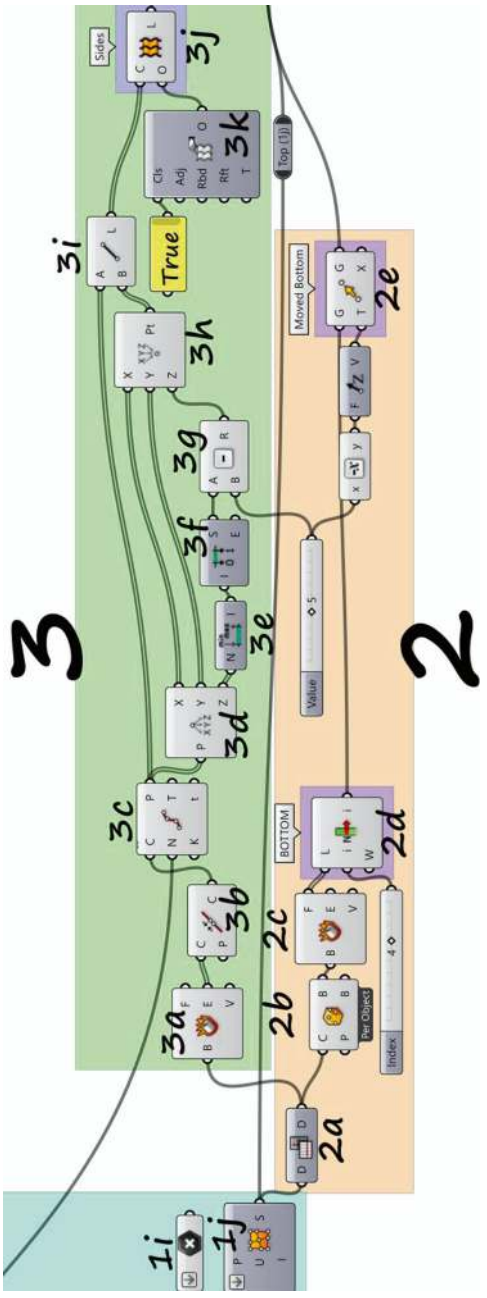


Figure 10-5. *Stages 2 and 3 of the node graph for generating a tile from image. If the relief surface from stage 1 is the top of the tile, stage 2 creates its bottom from the bottom face of its bounding box and stage 3 creates the tile's sidewalls whose height can be parametrically varied*

Stage 3: Creating the Sidewalls of the Tile

In Stage 3 (Figure 10-5), we construct the sidewalls of the tile by dividing the perimeter curve of the relief surface, offsetting a copy of the division points, linking the two sets of points pairwise into vertical line segments, and lofting through the segments to create the walls, as visualized in Figure 10-6.

Recall that at the beginning of stage 2, we used Flip Matrix (2a) to swap the rows and columns of the relief surface from stage 1, reordering its edges such that one can start at one corner of the surface and follow the direction of the edges to go all the way around. Therefore, we can simply link Flip Matrix (2a) to Deconstruct Brep (3a) to extract its perimeter edges, followed by Join Curves (3b) to combine them into a single closed curve and Divide Curve (3c) to get evenly spaced points along that curve. Since we know the dimension of the image (664 by 489), we can calculate how many points to divide the curve into (N input of Divide Curve (3c)) so that the points are one unit apart. The number of points all the way around is (664 + 489) * 2 = 2306.

Since the relief surface is not flat, we'll define the tile's thickness as measured from the lowest Z of the relief surface to the tile's bottom. To find the lowest Z of the relief surface, we link the P output of Divide Curve (3c) to Deconstruct (3d) to separate their Z coordinates, followed by Bounds (3e) to find the minimum (S) and maximum (E). We find the Z value at the bottom of the sidewalls by subtracting (3g) the thickness from the minimum Z of the relief surface. In Figure 10-5, we set the thickness of the tile to 5 using a Number Slider, which we'll also use to offset the tile bottom in stage 2. Since we want the bottom to move downward relative to the relief surface, we insert a Negative node between the Number Slider and Unit Z before linking it to Move (2e) to offset the bottom. We create the points at the bottom of the sidewalls with Construct Point (3h), using the same XY from points framing the relief surface (3d) and Zs we've just derived (S of 3f).

Next, we'll create vertical line segments evenly spaced around the perimeter so they can be lofted into walls. We create the line segments by linking the division points of the relief surface's perimeter curve and points at the bottom of the sidewalls we've just derived, as shown in Figure 10-6 (3i). The last step is to Loft (3j) through the line segments (3i). To make the sidewalls go all the way around, we link a Loft Options node (3k) with Cls (Closed) = True to the O input of Loft (3j).

Note Where to find the nodes in stage 3 of Figure 10-5: Surface ➤ Analysis ➤ Deconstruct Brep (3a), Curve ➤ Util ➤ Join Curves (3b), Curve ➤ Division ➤ Divide Curve (3c), Vector ➤ Point ➤ Deconstruct (3d), Maths ➤ Domain ➤ Bounds (3e), Maths ➤ Domain ➤ Deconstruct Domain (3f), Maths ➤ Operators ➤ Subtraction (3g), Vector ➤ Point ➤ Construct Point (3h), Curve ➤ Primitive ➤ Line (3i), Surface ➤ Freeform ➤ Loft (3j), and Surface ➤ Freeform ➤ Loft Options (3k).

Stage 4: Combining the Top, Sides, and the Bottom of the Tile

We now have all the pieces of the tile constructed (top, bottom, and sides). We'll combine the pieces into one mesh in stage 4 (Figure 10-7).

We Shift-drag the relief surface (1j), sidewalls (3j), and bottom (2e) to a Brep node (4a), followed by Mesh Brep (4b) to convert them into three meshes, Mesh Join (4c) to combine them into a single mesh, and Unify Mesh (4d) to ensure the mesh's normals are consistent.

Figure 10-8 shows how the relief surface (1j), sidewalls (3j), and the bottom (2e) fit together, while Figure 10-9 shows the combined and normal-corrected mesh (4d).

Note Where to find the nodes in stage 4 of Figure 10-7: Params ➤ Geometry ➤ Brep (4a), Mesh ➤ Util ➤ Mesh Brep (4b), Mesh ➤ Util ➤ Mesh Join (4c), and Mesh ➤ Util ➤ Unify Mesh (4d).

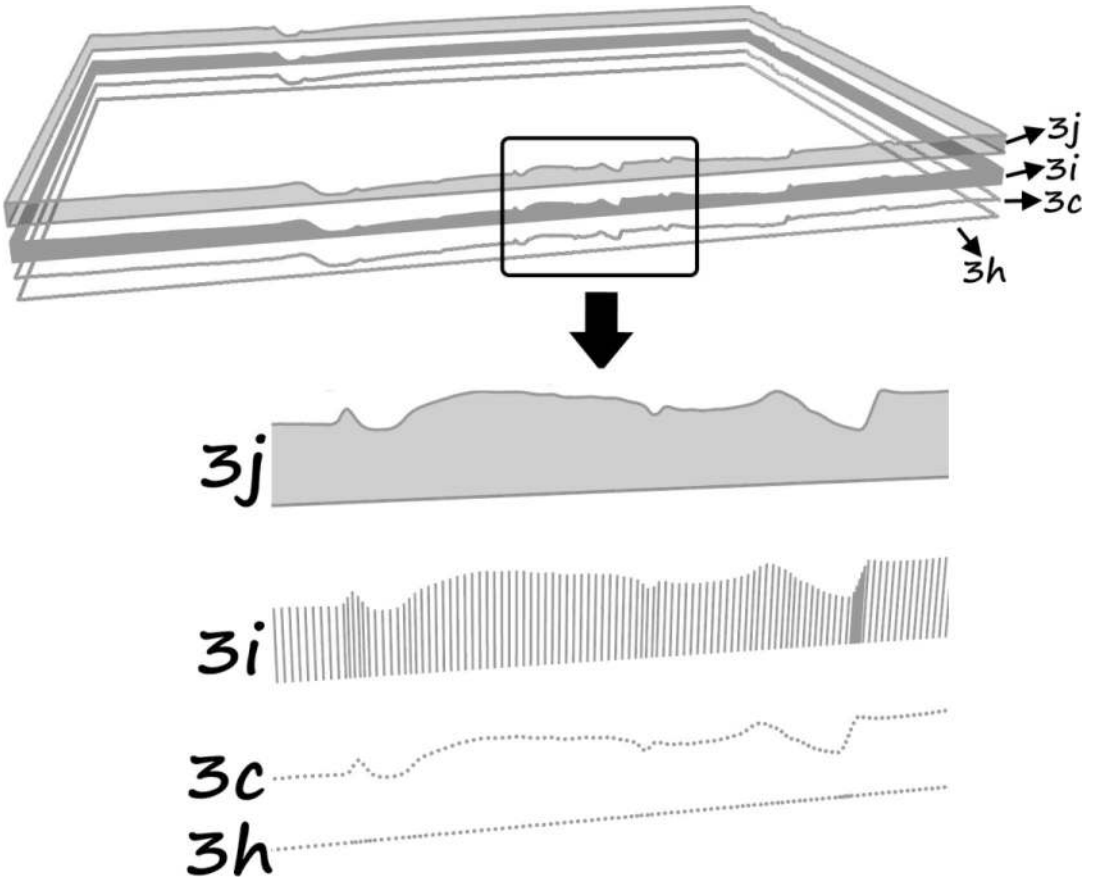


Figure 10-6. *A visual breakdown of how the sidewalls of the tile are constructed. We divide the perimeter of the relief surface (3c), offset a copy along –Z to create the points framing the bottom (3h), link those two sets of points into vertical line segments (3i), and loft through the line segments into the sidewalls (3j). The node labels refer to the node graph shown in Figure 10-5*

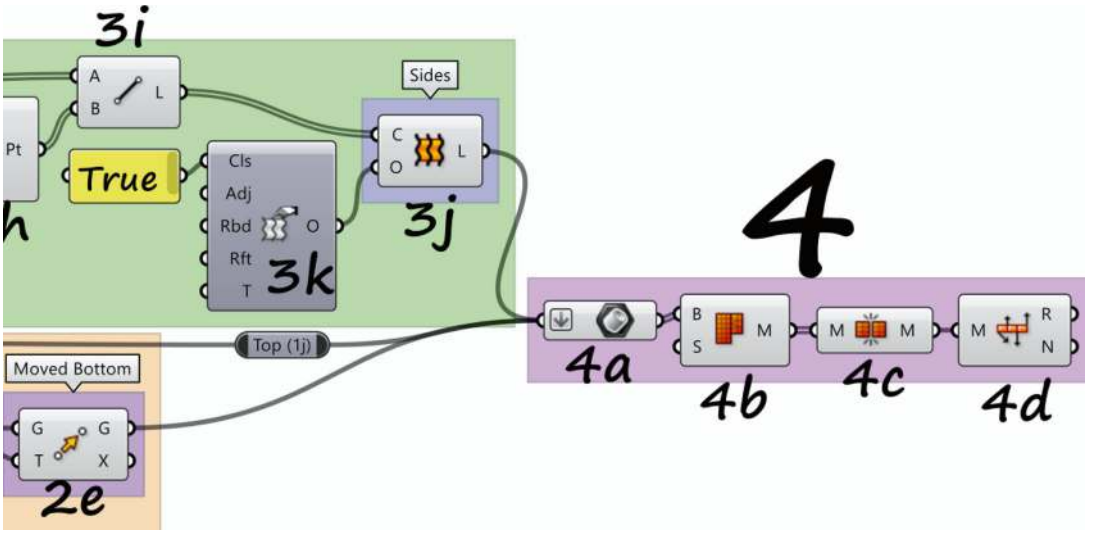


Figure 10-7. *Stage 4 of the node graph that generates a relief tile from image using Image Sampler, where we combine the top (1j), sides (3j), and bottom (2e) into a single mesh (4c) and ensure its normals are consistent (4d)*

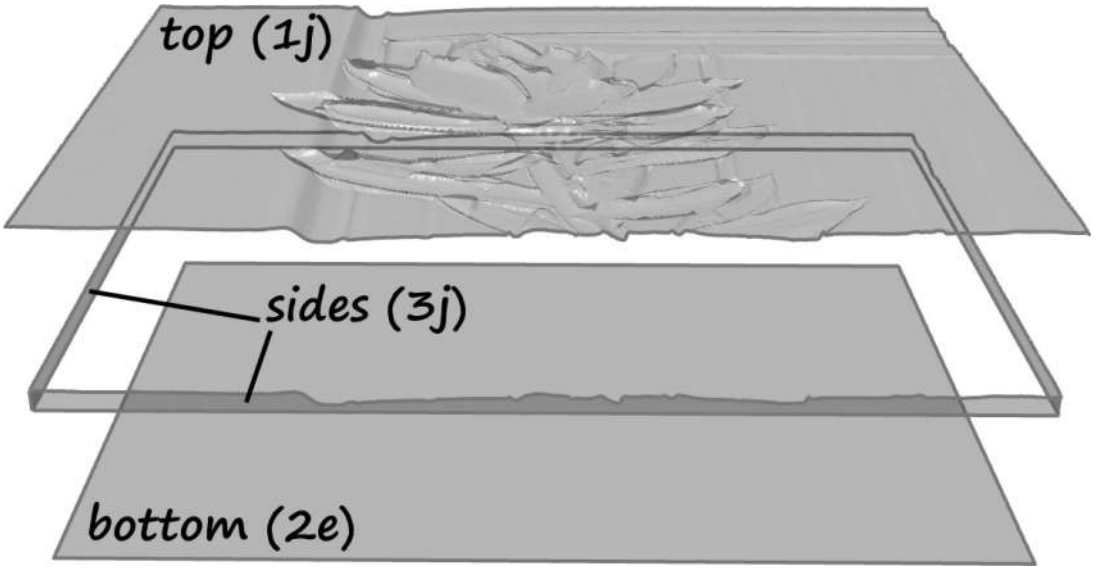


Figure 10-8. *Visualization of how the relief surface (or the top of the tile, 1j), the sides (3j), and bottom (2e) fit together. The node labels 1j, 3j, and 2e refer to the node graph shown in Figures 10-2 and 10-5*

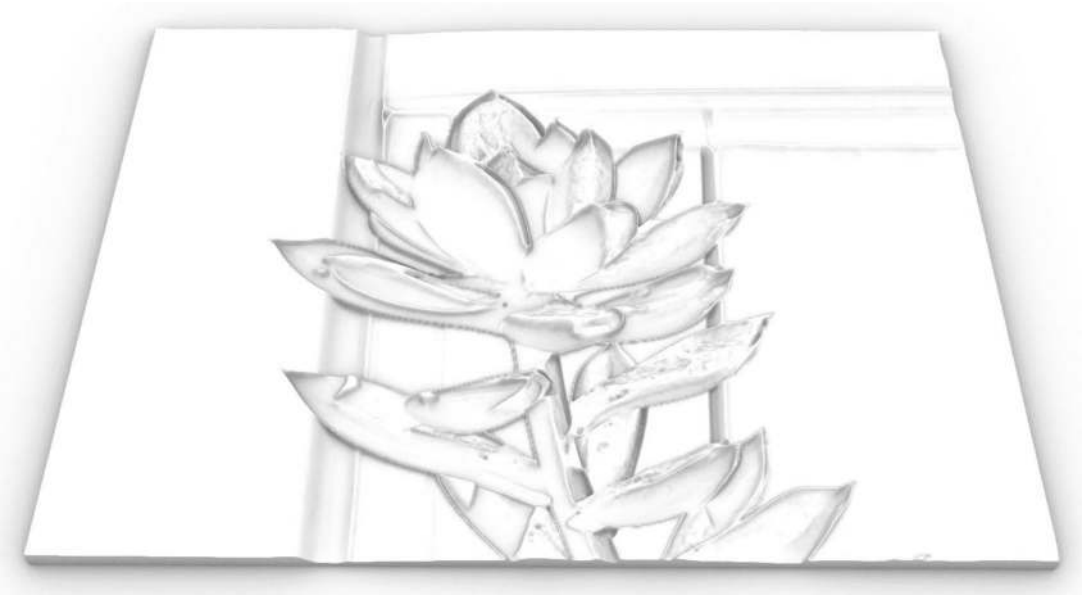


Figure 10-9. *Sample output of node 4d in Figure 10-7, where the top, bottom, and sides of the image tile are combined into a single mesh with its normals corrected*

Generating Tiles from Images Using the "Import Image" Node

In this section, we'll explore a different strategy for generating tiles from images, by using Import Image to convert an input image into a mesh and using pixel intensities to vary vertex heights.

Figure 10-10 shows the approach in four stages. Stage 1 samples the image with Import Image and varies the vertex heights based on pixel intensities. Stage 2 generates the sidewalls for the tile with the help of Python 3. Stage 3 creates the bottom of the tile. Stage 4 puts it altogether by combining the top, sides, and bottom of the tile into a single mesh.

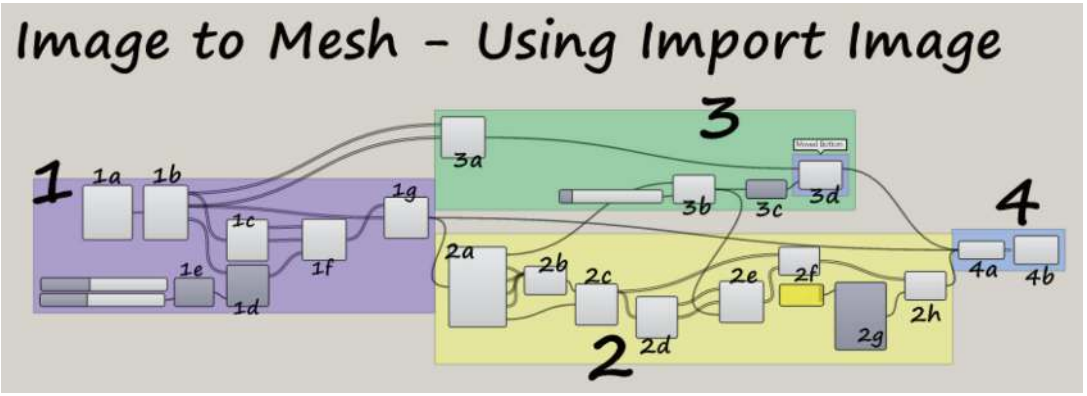


Figure 10-10. *Node graph for generating image tiles using an alternative approach, by creating a mesh with Import Image (stage 1) and varying its vertex heights based on pixel intensities. Stages 2, 3, and 4 serve similar functions as Figure 10-2 by creating the sides and bottom of the tile and combining all the pieces into a single mesh*

Stage 1: Creating a Relief Mesh from Pixel Intensities with the Import Image Node

Our goal in stage 1 (Figure 10-11) is to create a mesh from an input image with one vertex per pixel and vary vertex heights using pixel intensities.

We start by using the Import Image node (1a) to read the intensity values from an input image and generate a flat mesh with one vertex per pixel. To select the image to load, right-click Import Image's F input ➤ Select one existing file and open `/Ch10/succulent.jpg`. Ensure that right-click F ➤ Synchronize is toggled on so the data in the GH definition will be refreshed when the source image file changes.

Import Image (1a) provides a single mesh as output with color info of the pixels included. To create a relief surface like we did in Figure 10-2, we'll use Deconstruct Mesh (1b) to extract the vertices of the mesh, offset them along Z based on pixel intensities, and put them back together into a mesh. Here are the steps: We link the V output of Deconstruct Mesh (1b) to Deconstruct (1c) to separate the vertices' X, Y, and Z coordinates. We then link the C output of Deconstruct Mesh (1b) to Remap Numbers (1d) so the pixel values fall with the domain from 6 to 0 (1e); in other words, the darker pixels are taller (lithophane). With the new Z values computed, we reconstruct the vertices using Construct Point (1f) by reusing the original XY coordinates

(1c) plus the new Z values (1d). We then put the mesh back together using Construct Mesh (1g) with V set to the reconstructed vertices (1f) and F set to the faces from the original mesh output by Import Image (1a).

Figure 10-12 visualizes the output of 1g. You can see that the mesh generated this way has a very clean topology with all quad faces. The vertices corresponding to darker pixels have larger Z.

Note Where to find the nodes in stage 1 of Figure 10-11: Params ➤ Input ➤ Import Image (1a), Mesh ➤ Analysis ➤ Deconstruct Mesh (1b), Vector ➤ Point ➤ Deconstruct (1c), Maths ➤ Domain ➤ Remap Numbers (1d), Maths ➤ Domain ➤ Construct Domain (1e), Vector ➤ Point ➤ Construct Point (1f), and Mesh ➤ Primitive ➤ Construct Mesh (1g).

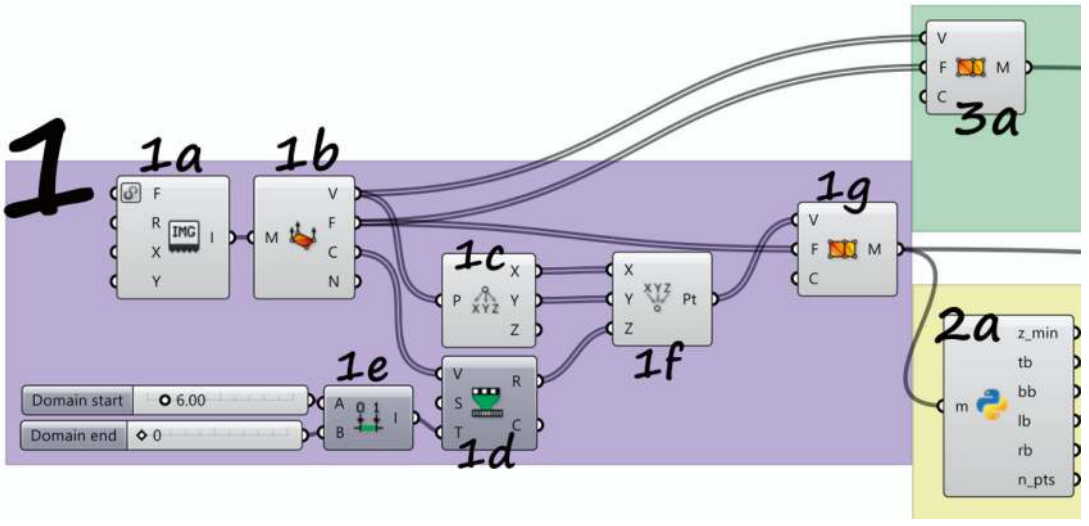


Figure 10-11. *Sample pixel values and generate a flat mesh with one vertex per pixel using the Import Image node (1a), deconstruct the mesh, vary the vertex heights based on pixel intensities, and reconstruct the mesh using the updated vertices (1g)*

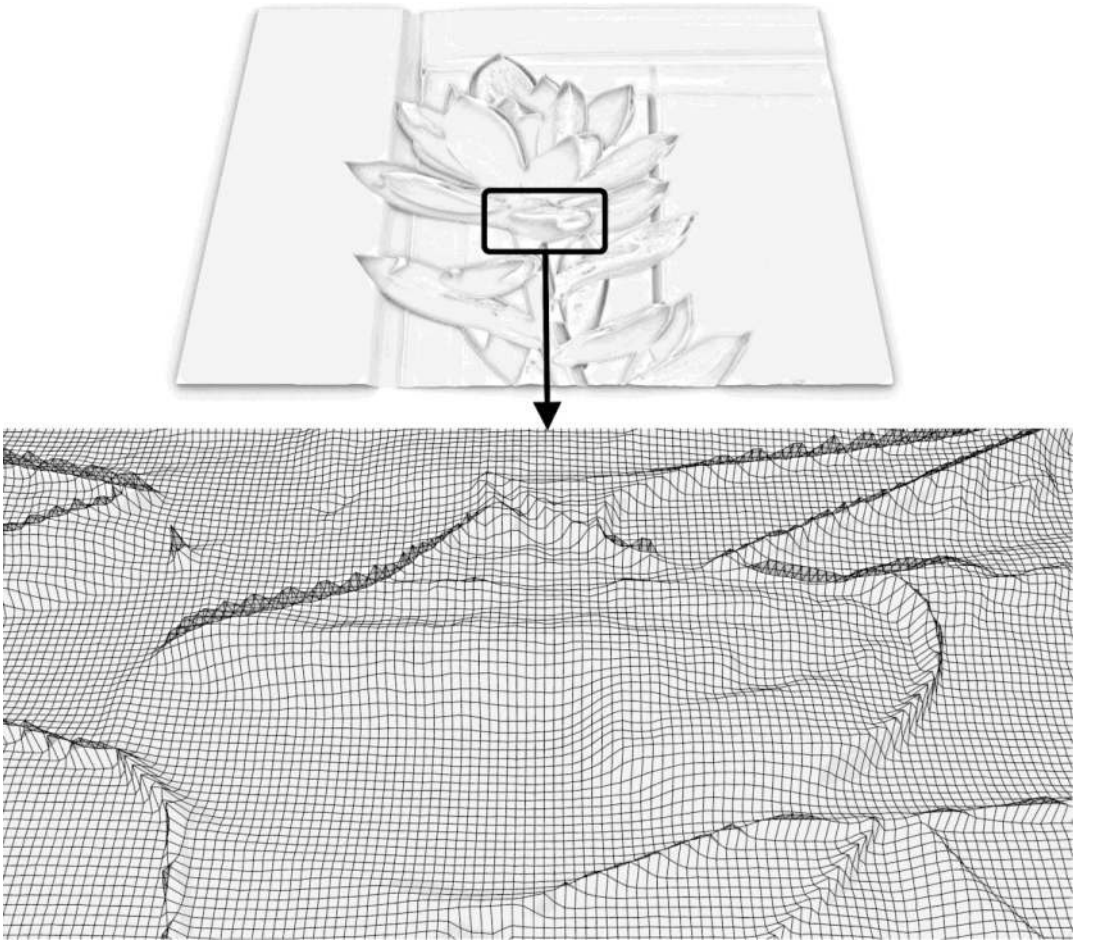


Figure 10-12. *The relief mesh created by varying Z coordinates of the mesh generated with Import Image based on pixel intensities. You can see that the mesh has a very clean topology with all quad faces*

Stage 2: Generating the Sidewalls of the Tile with the Help of Python 3

In stage 2 (Figure 10-14), we implement a Python 3 Script node (2a) to calculate the minimum Z of the relief mesh, the four curves constituting the perimeter of the mesh, and the total number of vertices (pixels).

The Access and Type Hints for the Python 3 Script node (2a) are summarized in Table 10-1, with the corresponding Python code shown in Listing 10-1. The node takes the mesh generated by Import Image (1a in Figure 10-11) as input (`m`) and outputs the minimum Z of `m`'s vertices (`z_min`), the four curves framing m's outer border (`tb`, `bb`, `lb`, and `rb`), and the number of vertices (`n_pts`) in `m`. If you imagine `m` as a rectangle, the four output curves `tb`, `bb`, `lb`, and `rb` correspond to the rectangle's top, bottom, left, and right edges when viewed from the top.

Table 10-1. *Access and Type Hints for the Python 3 Script node (2a) in Figure 10-14*

Name	Input or Output	Access	Type Hints
m	input	Item Access	Mesh
z_min	output	N/A	float
tb	output	N/A	Curve
bb	output	N/A	Curve
lb	output	N/A	Curve
rb	output	N/A	Curve
n_pts	output	N/A	int

Let's take a closer look at the Python code in Listing 10-1. We start by importing `ghpythonlib.components` as the shorthand `ghc`, so we can call Grasshopper components as Python functions. To find the minimum Z of the input mesh, we first call `ghc.BoundingBox` to find its bounding box, followed by `ghc.DeconstructBox` to find the box's domains along X, Y, and Z. We then call `ghc.DeconstructDomain` to extract the start and end of each of the three domains, to give us the minimum and maximum of the box's extent along X, Y, and Z.

The next block of code derives the top, bottom, left, and right sides of the mesh's border as curves. We start by finding the `border_edges` of the input mesh by calling `ghc.MeshEdges(m)` (note that this corresponds to the Naked Edges (E1) output of the Mesh Edges node). The next four lines of code find the four sides of the border by extracting sublists of continuous edges from `border_edges`.

The order of edges in `border_edges` is as follows (and visualized in Figure 10-13).

There are `x_max` edges along the top (`tb`) from left to right, with indices `0` to `x_max-1`.

The edges along the left and right are interweaved in the list `v_edges`. Each side has `y_max` edges, so `v_edges` has `y_max*2` edges, with indices from `x_max` to `x_max+y_max*2-1`.

The bottom side (`bb`) contains the last `x_max` edges, with indices starting at *1 past* the end of `v_edges`, which is `x_max+y_max*2`, to the end of the list, which is the length of `border_edges` minus 1 (`ghc.ListLength(border_edges)-1`)).

Since `v_edges` interweaves the left and right border edges in an alternating fashion (left, right, left, right, ...), we can call `ghc.Dispatch(v_edges, [True, False])` to separate it into two lists with the pattern `[True, False]`.

The bottom of Figure 10-13 shows a worked example with x_max=4 and y_max=2 (or a 4 by 2 rectangle). The top of the rectangle has edges with indices 0 to 4-1=3, the left and right sides have the edges with indices from 4 to 4+2*2-1 = 7, and the bottom has edges with indices from 8 to 12-1=11 (since the whole rectangle has (4+2)*2=12 edges).

The last thing to do is calculate the total number of vertices (`n_pts`), which is two times (`x_max + y_max`). We can reason about this in two ways. The first is the rectangle has an equal number of edges and vertices (you can convince yourself by counting the edges and vertices in the worked example at the bottom of Figure 10-13). The other is `x_max` is the number of edges along the top (or bottom), and the number of vertices is *one more* than the number of edges. Analogously, y_max is the number of edges along the left (or right), and the number of vertices is one more. However, if you count the vertices one side at a time, they overlap at the corners (top and left, top and right, bottom and left, bottom and right). Therefore, the number of vertices is (x_max+1 + y_max+1)*2 – 4 = (x_max+y_max)*2.

Listing 10-1. Code for the Python 3 Script node in Figure 10-14, where we derive the minimum Z of the relief mesh, border edges of the mesh, and the total number of vertices (see `/Ch10/Figure10-10_to_10-17_TileFromImage_ImportImage.gh`)

```
import ghpythonlib.components as ghc

mbbox, _ = ghc.BoundingBox(m,
    ghc.XYPlane(ghc.ConstructPoint(0,0,0)))
_, dom_x, dom_y, dom_z = ghc.DeconstructBox(mbbox)

_, x_max = ghc.DeconstructDomain(dom_x)
```

```
_, y_max = ghc.DeconstructDomain(dom_y)
z_min, _ = ghc.DeconstructDomain(dom_z)

border_edges, _, _ = ghc.MeshEdges(m)
tb, _ = ghc.SubList(border_edges, ghc.ConstructDomain(0,
    x_max-1), True)
v_edges, _ = ghc.SubList(border_edges,
    ghc.ConstructDomain(x_max, x_max+y_max*2-1), True)
rb, lb = ghc.Dispatch(v_edges, [True, False])
bb, _ = ghc.SubList(border_edges,
    ghc.ConstructDomain(x_max+y_max*2,
    ghc.ListLength(border_edges)-1), True)

n_pts = (x_max + y_max)*2
```

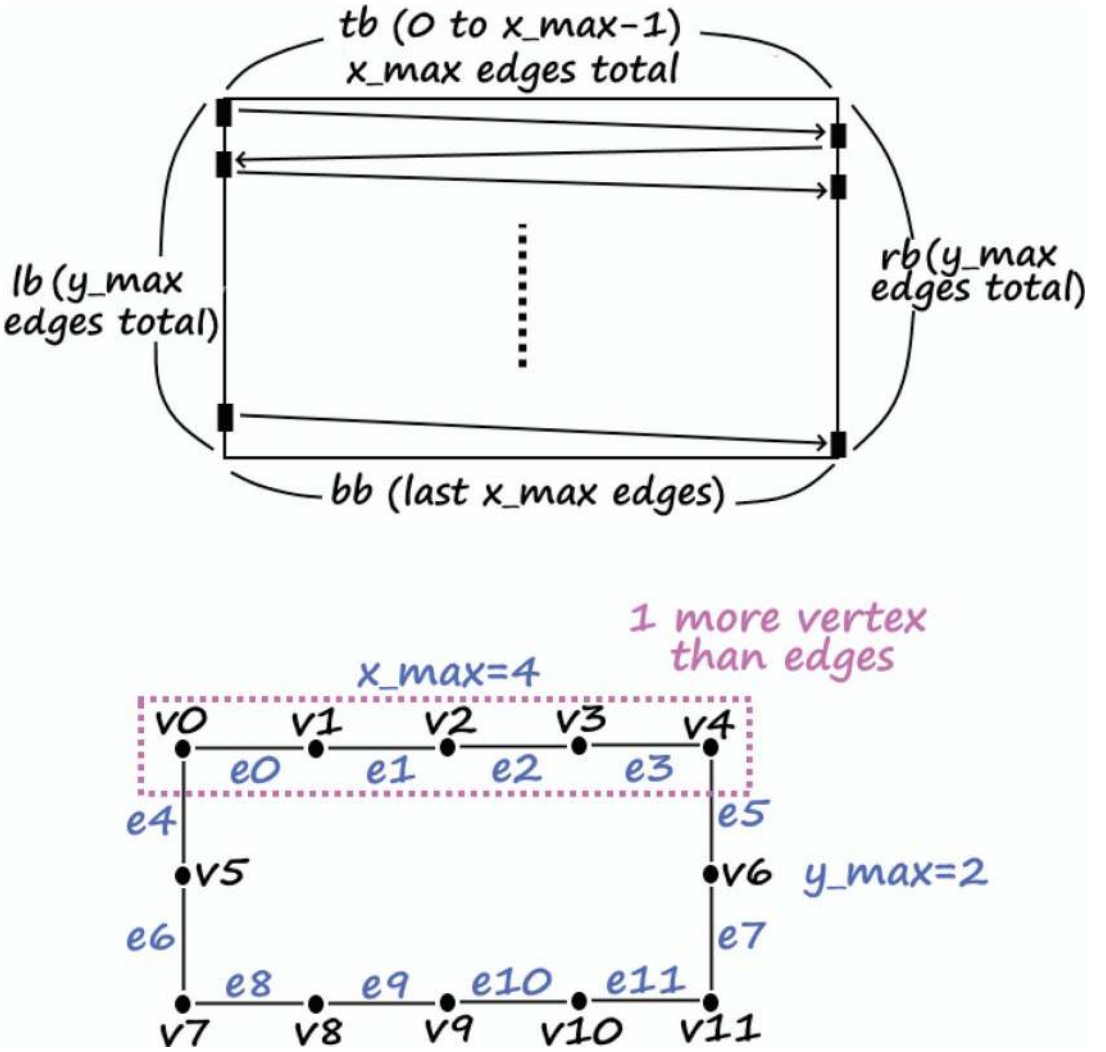


Figure 10-13. *Visualizing indices of the input mesh's border edges for the Python 3 Script node (2a in Figure 10-14). The breakdown of indices is based on variable names in Listing 10-1. The bottom shows a worked example for a 4 by 2 rectangle, where x_max=4 and y_max=2*

Note Recall that for a Python 3 Script node, you right-click an input to change its name and set its Access (Item, List, or Tree Access) and Type Hints.

You also right-click an output to set its name and Type Hints. There is no Access Hint for an output.

Next, we use Join Curves (2b) to combine the four border curves output by the Python 3 Script node (2a) into a single closed curve and break up this curve into evenly spaced points one unit apart using Divide Curve (2c) with N = the number of border vertices (`n_pts`) output by 2a.

The next step is to separate the X, Y, and Z values of these points with Deconstruct (2d), so we can reuse the same XY values to create a second set of points whose Z are the minimum Z of the relief mesh (`z_min` by 2a) minus five, so they sit five units below. We link the two sets of points pairwise (2f) to create vertical line segments all around, which we Loft (2h) into the tile's sidewalls. Loft Options (2g) with Cls set to True is used for the 0 input of Loft (2h) to ensure the lofted surface is closed (therefore goes all the way around the tile).

Note Where to find the nodes in stage 2 of Figure 10-14: Maths ➤ Script ➤ Python 3 Script (2a), Curve ➤ Util ➤ Join Curves (2b), Curve ➤ Division ➤ Divide Curve (2c), Vector ➤ Point ➤ Deconstruct (2d), Vector ➤ Point ➤ Construct Point (2e), Curve ➤ Primitive ➤ Line (2f), Surface ➤ Freeform ➤ Loft Options (2g), and Surface ➤ Freeform ➤ Loft (2h).

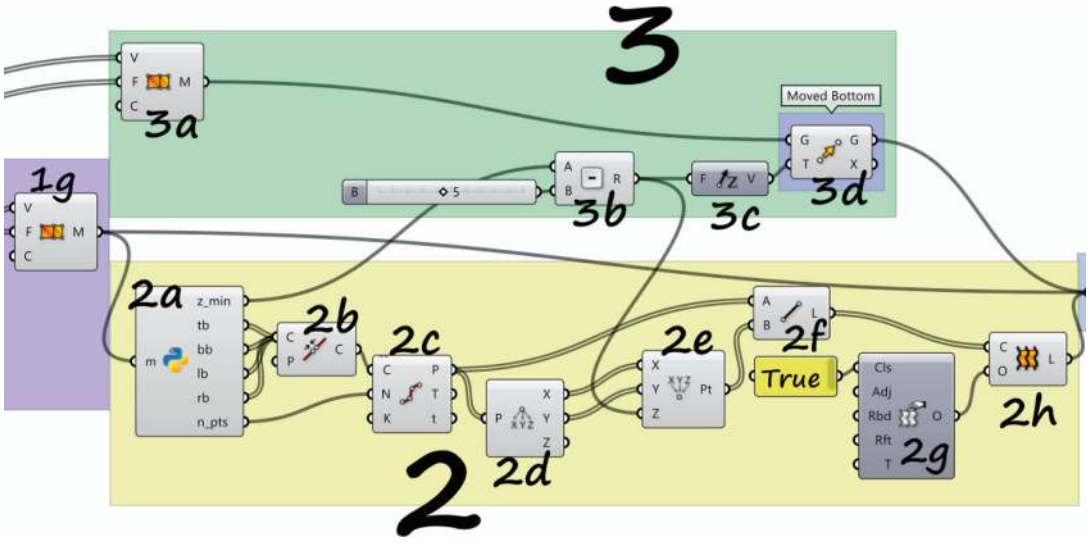


Figure 10-14. *Stages 2 and 3 of the node graph from Figure 10-10. After using Import Image in stage 1 (Figure 10-11) to sample the image and create a relief mesh, stage 2 uses Python 3 to extract the border curves to loft into the tile's sidewalls, while stage 3 creates the tile's bottom*

Stage 3: Creating a Flat Bottom for the Tile

In stage 3 (Figure 10-14), we create the flat bottom for the tile, by making a copy of the mesh output by Import Image (1a in Figure 10-11) without the color info (3a), moving the copy (3d) to five units below the lowest Z of the relief mesh (`z_min` output by 2a). The same Z offset is used in stage 2 to construct the bottom points for the sidewalls.

Figure 10-15 shows how the relief mesh (1g in Figure 10-11), the sides of the tile (2h), and the bottom of the tile (3d) fit together.

Note Where to find the nodes in stage 3 of Figure 10-14: Mesh ➤ Primitive ➤ Construct Mesh (3a), Maths ➤ Operators ➤ Subtraction (3b), Vector ➤ Vector ➤ Unit Z (3c), and Transform ➤ Euclidean ➤ Move (3d).

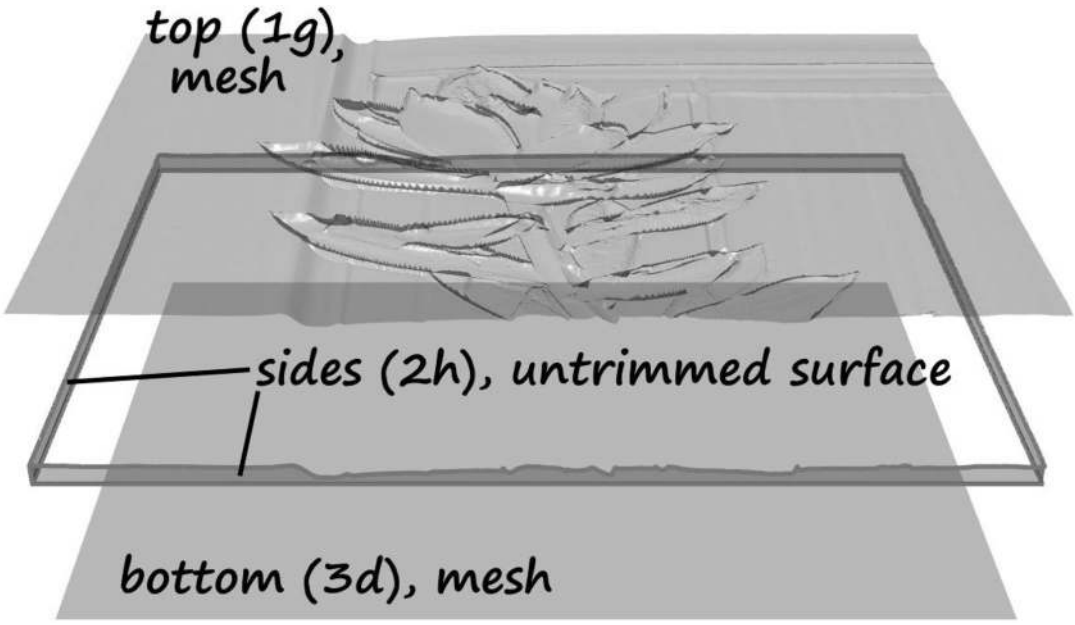


Figure 10-15. *Visualizing how the top (1g in Figure 10-11), sides (2h in Figure 10-14), and bottom (3d in Figure 10-14) are combined into a solid tile*

Stage 4: Combining the Top, Sides, and Bottom of the Tile and Converting It into a Mesh

In Stage 4 (Figure 10-16), we use Mesh Join (4a) to combine the relief mesh (1g in Figure 10-11), the sidewalls of the tile (2h), and the bottom mesh (3d) into a single mesh. Note that even though 1g and 3d are meshes while 2h is an untrimmed surface, we can link all three to the same Mesh Join node (4a) since the surface gets implicitly converted to the correct format. Finally, we use Unify Mesh (4b) to ensure that the combined mesh's normals are consistent.

Note Where to find the nodes in stage 4 of Figure 10-16: Mesh ➤ Util ➤ Mesh Join (4a) and Mesh ➤ Util ➤ Unify Mesh (4b).

Comparing Tiles Generated Using Image Sampler vs. Import Image

Figure 10-17 shows a comparison of the tiles generated using the node graph from Figures 10-1 to 10-9 (top) and Figures 10-10 to 10-17 (bottom). Recall that Figures 10-1 to 10-9 use Image Sampler while Figures 10-10 to 10-17 use a combination of Import Image and Python 3. You can see that the results are nearly indistinguishable.

Image Sampler is more intuitive to use and provides a visual preview. However, you must manually change the Construct Domain nodes defining the height range for creating the relief surface, as well as the relief surface's U count. With Import Image, the dimension of the image is implicitly derived. The downside is that it is harder to diagnose when something is wrong. Having to supplement with Python code to extract boundary curves also makes the second approach more complex.

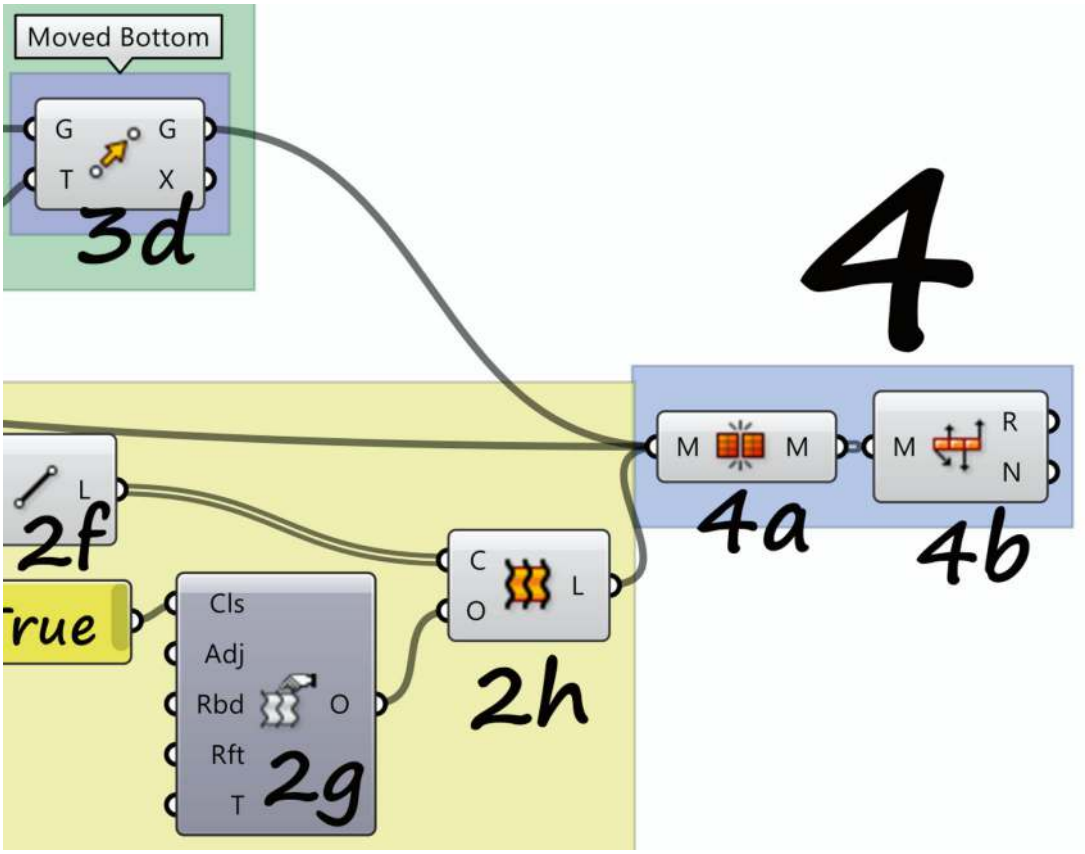


Figure 10-16. *Stage 4 of the node graph from Figure 10-10, where we combine the relief mesh (1g), the sidewalls (2h), and the bottom (3d) of the tile into a single mesh (with Mesh Join, 4a) and ensure the joined mesh's normals are consistent (with Unify Mesh, 4b)*

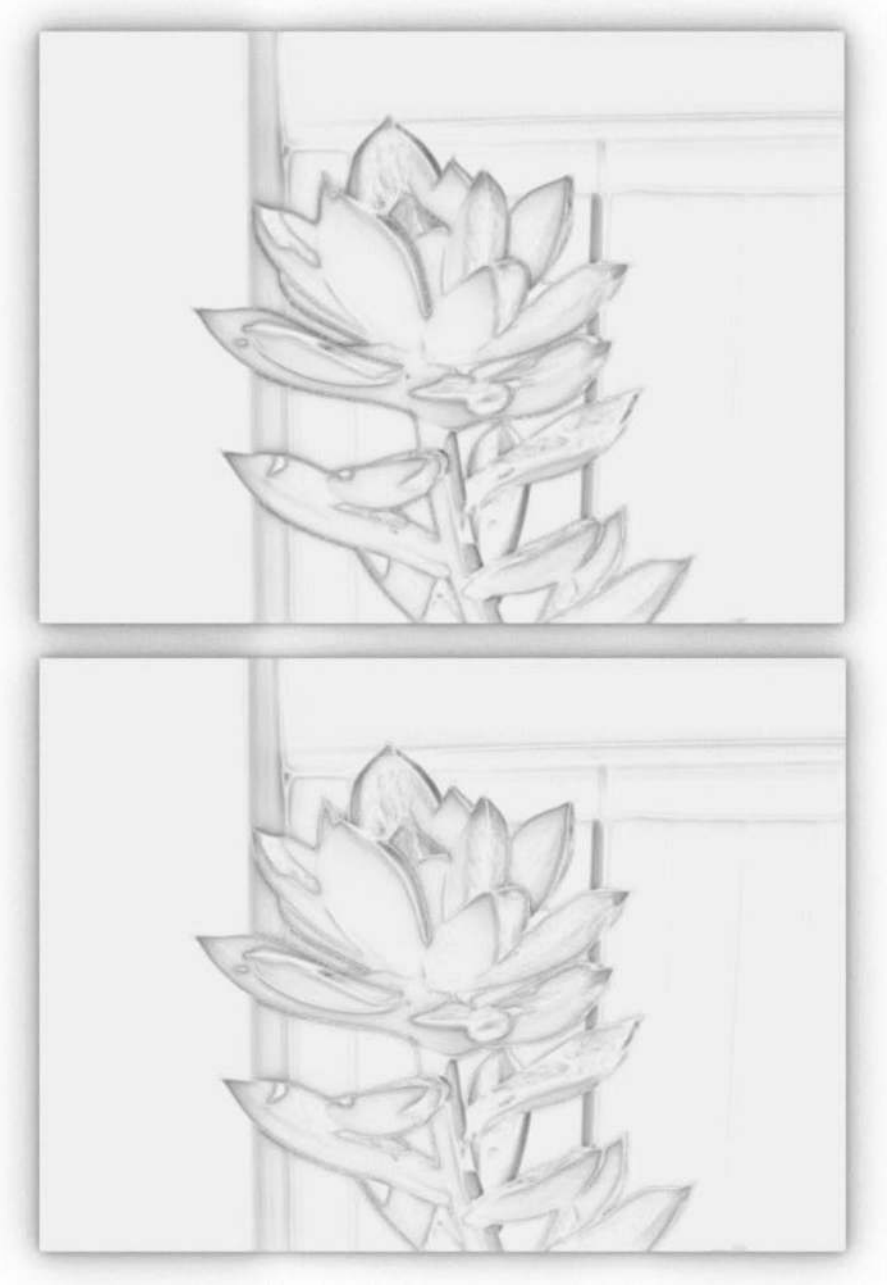


Figure 10-17. *Comparison of tiles generated using the Image Sampler node (top) vs. the Import Image node (bottom). The top is the output of 4d in Figure 10-7 and identical to the tile depicted in Figure 10-9. The bottom is the output of 4b in Figure 10-16*

Projecting Image Tiles onto Other Surfaces As Decorative Elements

With two approaches for generating tiles from images under your belt, in this section, we'll discuss how to project tiles onto other surfaces as decorative elements.

Projecting an Image Tile onto a Sphere

The first example we'll look at is projecting an image tile onto a sphere as shown in Figure 10-19. To do so, we'll add a fifth stage onto our first node graph for generating image tiles using Image Sampler (Figures 10-1 to 10-9), as shown in Figure 10-18.

To start, we'll combine the top (1j in Figure 10-2) and the sides (3j in Figure 10-5) into an open-back tile using a three-node sequence: Mesh Brep (5a) to turn each piece into a mesh, Mesh Join (5b) to combine them into a single mesh, and Unify Mesh (5c) to ensure the mesh's normal vectors are consistent.

Next, we'll approximate the center of the tile from its bounding box, which we'll use as the Plane input for Splop (5h) shortly to perform the projection, as well as position the sphere we'll project onto (5g). We link the tile mesh to Bounding Box (5d), followed by the Area node (5e) to get its center (C output).

To create the sphere to project onto, we first calculate the radius based on the image dimension using an Expression node (5f) with two inputs, x and y, and the following formula:

$$\sqrt{\left(\frac{x}{2}\right)^2+\left(\frac{y}{2}\right)^2}$$

To enter the formula into the Expression node (5f), double-click the node to open the "Expression Designer" window, and enter the following line in the Expression field:

$$\mathrm{sqrt}((x/2)^2+(y/2)^2)$$

You can use the A^2 button to add the "square" in the expression. The button is located on the second row of Operators above the Expression field. With the Expression node set up, you'll link the image width and height to its x and y inputs, respectively, which are 664 and 489 for the provided `/Ch10/succulent.jpg`. Finally, we use the

Sphere node (5g) to create the sphere to project onto, by using the tile's bounding box center (5e) as the B input, which is the center of the sphere, and the output of Expression (5f) as the R input, which is the radius.

Next, we use Transform ➤ Morph ➤ Splop (5h) to project the tile onto the sphere, which will conform the tile to the contour of the sphere but keep it as a separate piece—we'll combine the two at the end.

To set up Splop (5h), we link the open-back tile (5c) to the G input and the center of the tile (5e) to the P input—which will implicitly create a plane centered at that point parallel to the XY plane as the source plane for the projection. The sphere (5g) is the S input, which is the target surface to project onto, while uv is where on the target surface to center the projection—we used (0.5, 0, 0) in this case (5i), which produces the projection shown in Figure 10-19, halfway between the north and south poles of the sphere. Finally, we combine the morphed tile (5h) and the sphere (5g) into a single mesh with Mesh Join (5j).

Note Where to find the nodes in stage 5 of Figure 10-18: Mesh ➤ Util ➤ Mesh Brep (5a), Mesh ➤ Util ➤ Mesh Join (5b), Mesh ➤ Util ➤ Unify Mesh (5c), Surface ➤ Primitive ➤ Bounding Box (5d), Surface ➤ Analysis ➤ Area (5e), Maths ➤ Script ➤ Expression (5f), Surface ➤ Primitive ➤ Sphere (5g), Transform ➤ Morph ➤ Splop (5h), Params ➤ Input ➤ Panel (5i), and Mesh ➤ Util ➤ Mesh Join (5j).

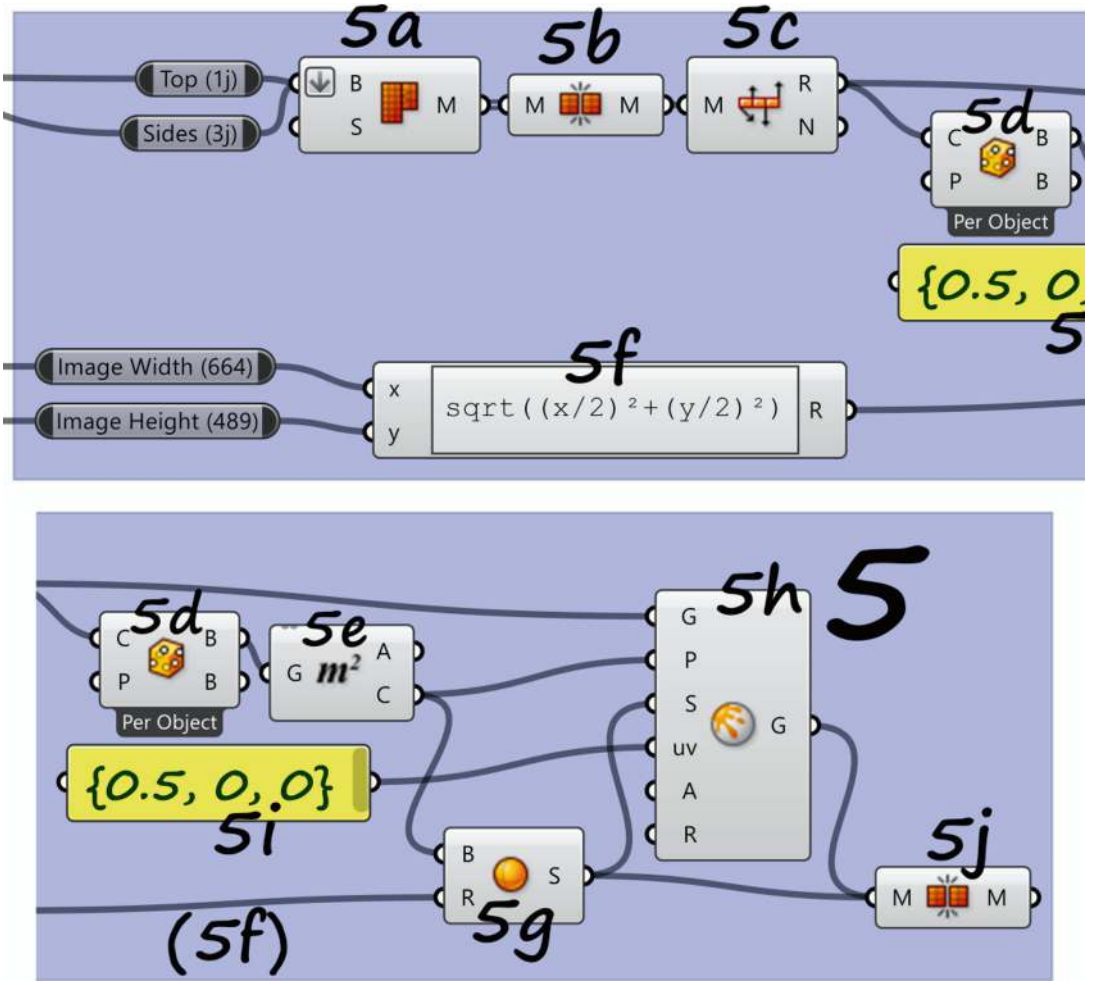


Figure 10-18. *A fifth stage added to the node graph from Figures 10-1 to 10-9 to project a generated image tile onto a sphere. We start by combining the top and sides to make an open-back tile (5a to 5c) and find its center. We then create a sphere (5g) whose radius is computed using an Expression (5f) based on the image's dimension (664 by 489). Splop (5h) is used to morph the open-back tile onto the sphere. Finally, the morphed tile and the sphere are combined into a single mesh (5j)*



Figure 10-19. *Image tile generated using the provided image* `/Ch10/succulent.jpg` *and projected onto the surface of a sphere using Transform ➤ Morph ➤ Splop*

Projecting Image Tile onto a Cylinder

We can easily modify the node graph from Figure 10-18 to project onto a cylinder instead of a sphere, by replacing the Sphere node (5g) with the group of nodes labeled "6" in Figure 10-20.

To make the cylinder large enough to fit the entire tile, we link the tile's bounding box (5d) to Deconstruct Box (6a) to get the tile's height (Y) and use it to set the Cylinder's height (6b, L input). To make the tile wrap roughly halfway around the side of the cylinder, we use half (6c) the Sphere's (5g) radius from Figure 10-18 as the Cylinder's radius (6b, R input). The tile's bounding box center (5e) is used as the Cylinder's center (6b, B input). The Cylinder is by default open-ended—here we use Cap Holes (6d) to close its top and bottom.

We reuse the Splop node (5h) from Figure 10-18 to project the open-back tile (5c) onto the cylinder, by replacing Splop's S input with Cylinder (6b) while keeping all other inputs the same (including uv = {0.5, 0, 0}). We then combine the morphed tile and the cylinder into a single mesh with Mesh Join (5j), which is shown in Figure 10-21.

Note Where to find the nodes in stage 6 of Figure 10-20: Surface ➤ Analysis ➤ Deconstruct Box (6a), Surface ➤ Primitive ➤ Cylinder (6b), Maths ➤ Operators ➤ Multiplication (6c), and Surface ➤ Util ➤ Cap Holes (6d).

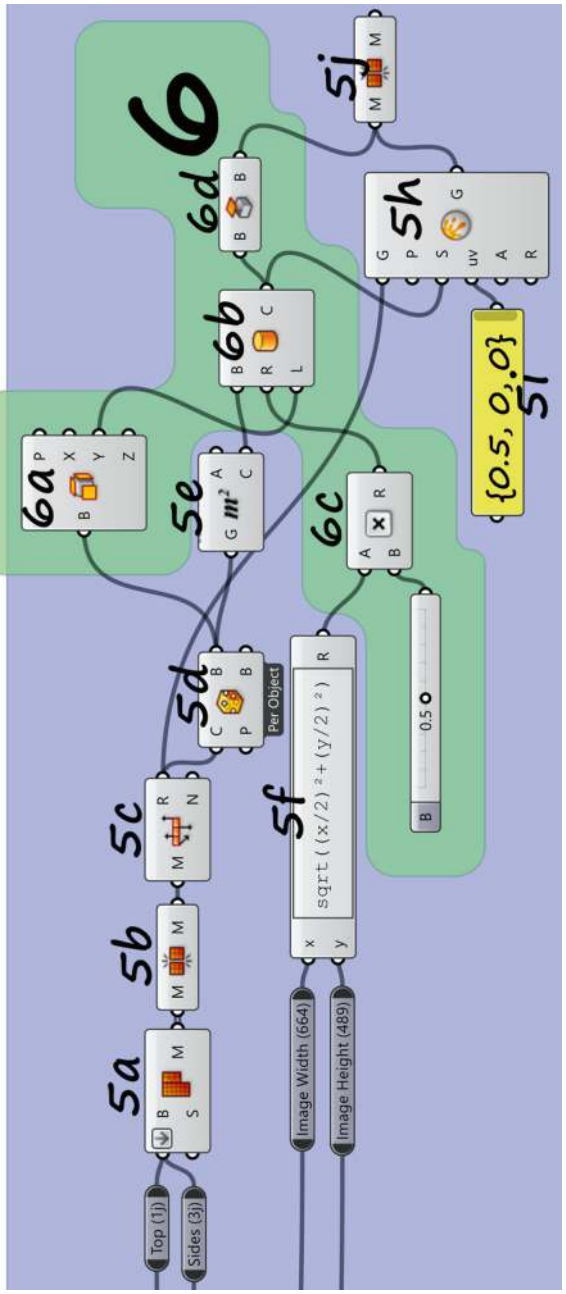


Figure 10-20. *Replacing Sphere (5g) with the group of nodes labeled "6" as shown to project the open-back tile (5c) onto a Cylinder (6b) instead of a sphere. The Cylinder's height (L input) is set to match the image's height (Deconstruct Box, 6a, Y), while its radius is set to half the radius used for the sphere to make the tile wrap roughly halfway around*



Figure 10-21. *The image tile generated from the provided image* `/Ch10/succulent.jpg`*, projected onto a cylinder by running the node graph from Figure 10-20*

Generating Mold Boxes from Image Tiles

In addition to using image tiles as decorative elements, we can use them to create mold boxes that can be 3D printed with a flexible filament such as TPU to cast copies of the tiles in different materials, like plaster, cement, etc. To turn an image tile into a mold box, we use the tile as the bottom of the box with the contoured side facing up and create four walls wrapped around the tile, as shown in Figure 10-25.

We extend stage 2 of the node graph for generating image tiles using Image Sampler (Figures 10-1 to 10-9) to add the four walls for the mold box. Figure 10-22 shows this extension as stage "2b." We make a scaled copy of the tile's bottom perimeter curve based on how thick we want the walls for the mold box to be and create a surface between the original perimeter and the scaled-up perimeter (imagine this surface as the footprint of the walls). We then Extrude (2b8) this footprint upward into walls based on how tall we want the mold box to be.

We can find the amount to scale the tile's bottom perimeter by how much longer we want it to be. If we add 10 units to each side of the rectangle, we add a total of 10 * 4 = 40 units (2b1) to the perimeter Length (2b2). We then find the scale factor by dividing (2b4) the increased length (2b3) by the original length (2b2). We Scale (2b6) the perimeter curve about its center (found via Area (2b5)) using the scale factor (F) we've just calculated (2b4).

Next, we create the footprint of the walls by Shift-dragging the original and the scaled perimeter curves to Boundary Surfaces (2b7), which creates a thin ring-like surface between the two curves as shown in Figure 10-23. We then Extrude (2b8) the footprint into walls with adjustable height using a Number Slider linked to Unit Z (2b9) as the extrude direction (D, 100.00 as shown).

Note Where to find the nodes in stage 2b of Figure 10-22: Maths ➤ Operators ➤ Multiplication (2b1), Curve ➤ Analysis ➤ Length (2b2), Maths ➤ Operators ➤ Addition (2b3), Maths ➤ Operators ➤ Division (2b4), Surface ➤ Analysis ➤ Area (2b5), Transform ➤ Affine ➤ Scale (2b6), Surface ➤ Freeform ➤ Boundary Surfaces (2b7), and Surface ➤ Freeform ➤ Extrude (2b8).

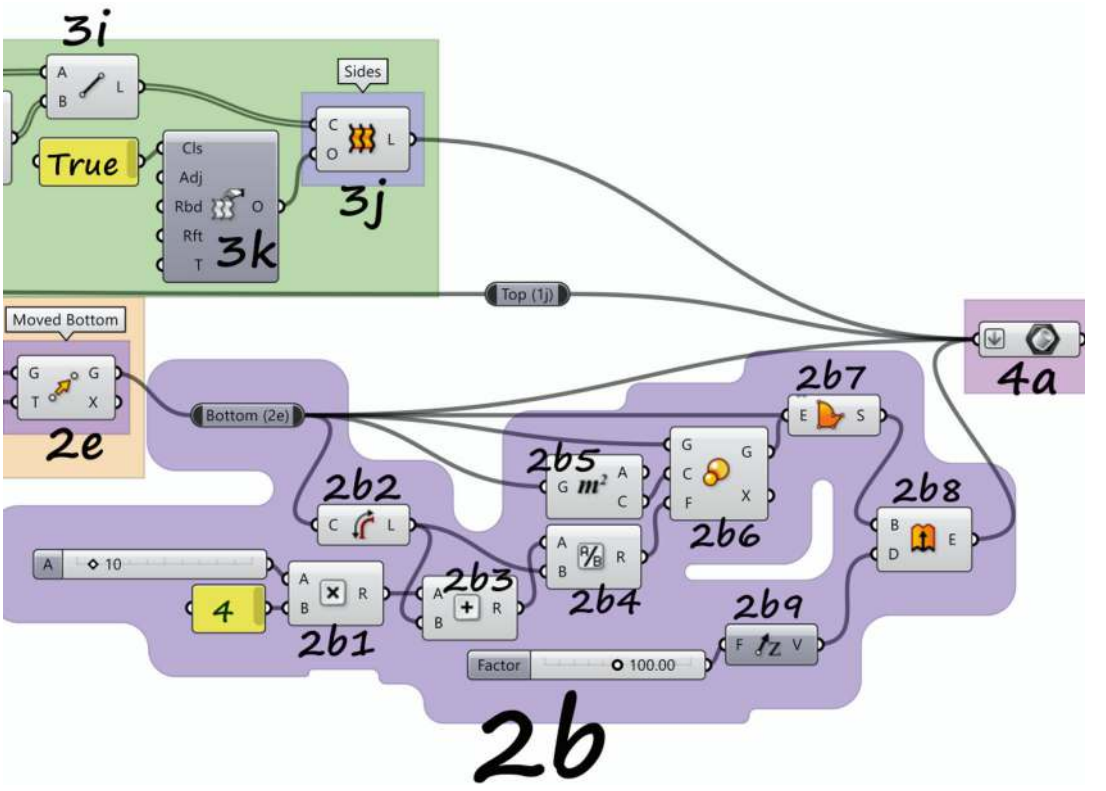


Figure 10-22. *Extending the graph for generating image tiles using Image Sampler (Figures 10-1 to 10-9) to turn the tiles into 3D printable mold boxes. The partial graph marked "2b" is added after stage 2 to create the four walls for the box. The walls are extruded from a surface formed between the tile's bottom perimeter and a scaled-up copy of the perimeter*

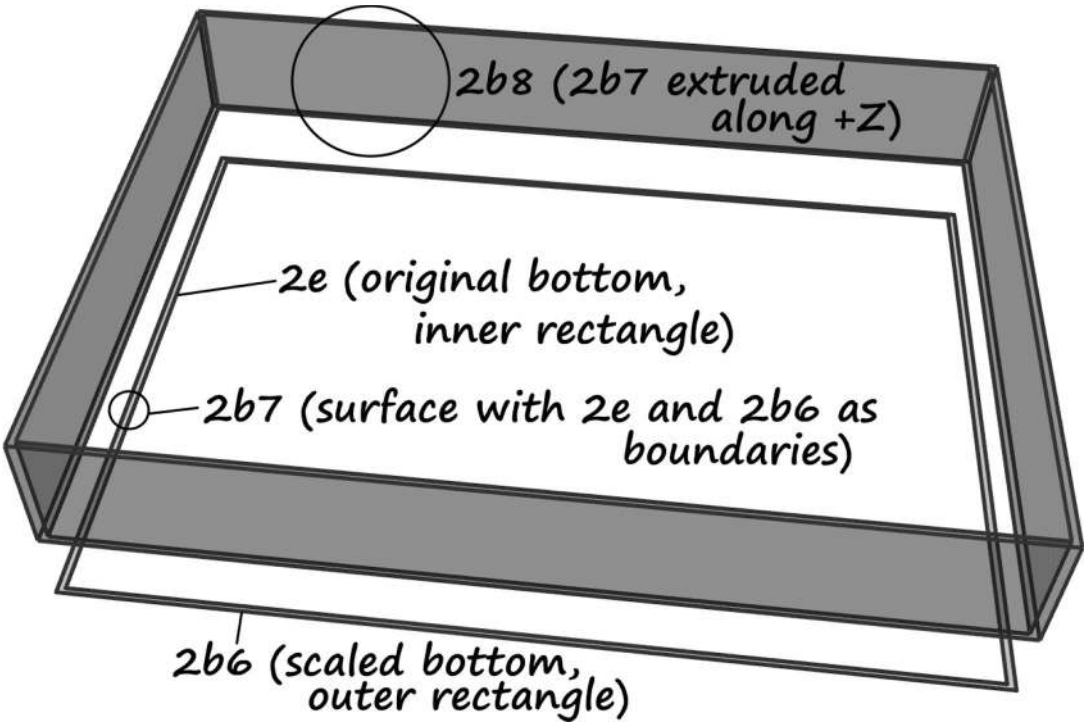


Figure 10-23. *Visualization of node outputs from Figure 10-22. Boundary Surfaces (2b7) is used to create the footprint of the walls using the tile's bottom perimeter curve (2e, Figure 10-5) and a scaled copy of the perimeter (2b6). The footprint is then extruded (2b8) along +Z to create the walls for the mold box*

Inverting Heights of the Relief Surface for the Mold

For both approaches of generating image tiles (using Image Sampler (Figures 10-1 to 10-9) and using Import Image (Figures 10-10 to 10-17)), we set the graphs to produce lithophanes, such that darker pixels in the image are taller in the tile's relief surface, therefore thicker and blocking more lights when 3D printed. For the purpose of a mold, you want the heights of the relief surface to be inverted, so that the casted piece will have the "correct" contour.

Figure 10-24 shows part of stage 1 from Figure 10-2 where the output of Image Sampler (1a) is converted from RGB values to the height range of the relief surface using Remap Numbers (1e). To invert the range such that lighter pixels are taller, we

simply swap the values of the two Number Sliders defining the target domain for Remap Numbers (1e) so it goes from 0 to 6 instead of 6 to 0, that is, "low to high" instead of "high to low."

Figure 10-25 shows a side-by-side comparison of two mold boxes generated with height ranges 6 to 0 (darker pixels are taller) and 0 to 6 (lighter pixels are taller), respectively.

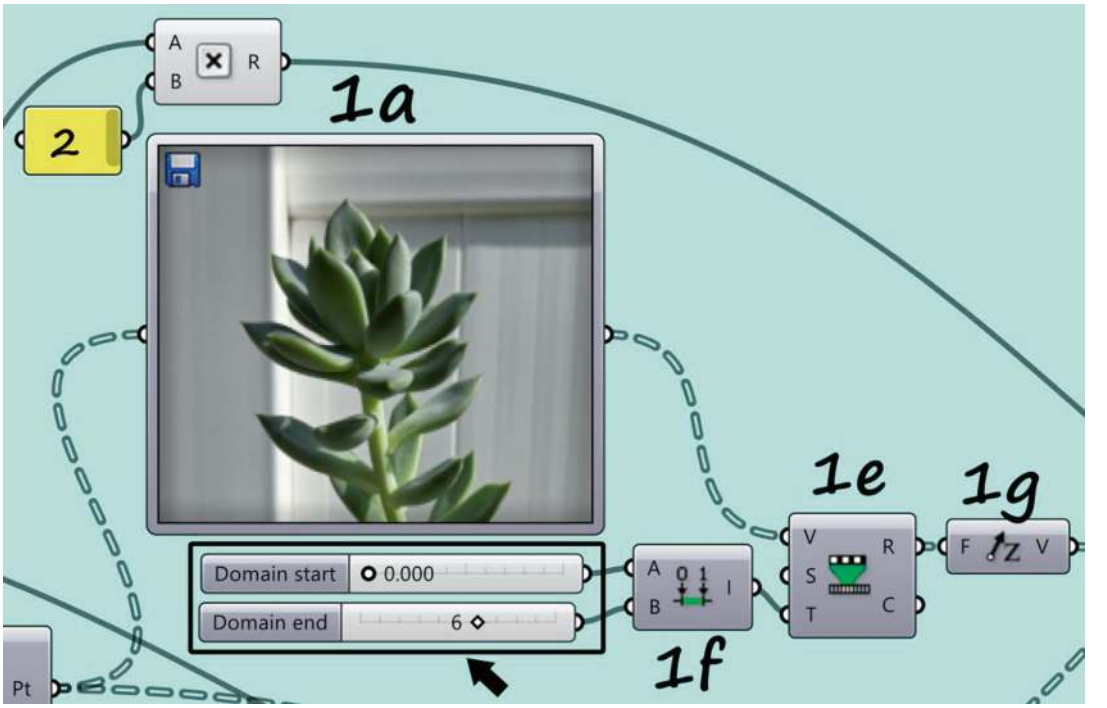


Figure 10-24. *Inverting the range for mapping pixel intensities so lighter pixels correspond to taller points on the relief surface. A partial stage 1 of the node graph using Image Sampler (Figures 10-1 to 10-9) is reproduced here. As shown, the range for remapping pixel intensities is set to 0–6*

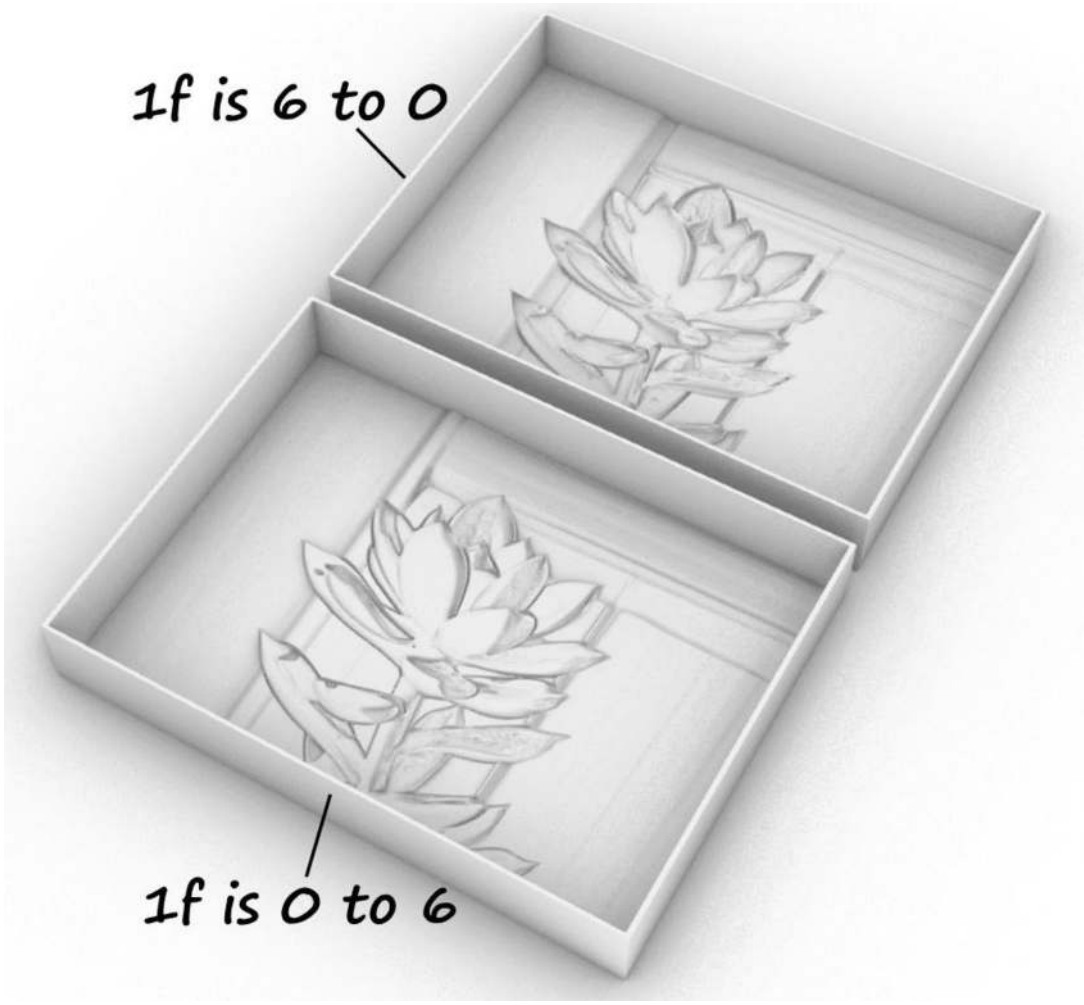


Figure 10-25. *Side-by-side comparison of two mold boxes generated by the node graph in Figure 10-22. The top has pixel intensities mapped to the range of 6–0 (therefore, darker pixels are taller). The bottom has pixels intensities mapped to the range of 0–6 (therefore, lighter pixels are taller)*

3D-Printed Lithophane (Darker Pixels Are Taller)

I have 3D printed a lithophane produced by the first tile generator (Figures 10-1 to 10-9) using the provided image `/Ch10/succulent.jpg`, as shown in Figure 10-26. The input sliders for Remap Numbers are set to 6–0 so darker pixels are taller on the relief surface.

As you can see, the darker (more shadowy) parts of the leaves are taller in the physical printed tile. Figure 10-27 shows what the same tile looks like when held up to a light. Since the thicker parts allow less light to shine through, they appear darker when back-lit. The lithophane was 3D printed on a desktop FDM using white PLA filament.



Figure 10-26. *Unlit lithophane, 3D printed from a tile generated with pixel intensities mapped to 6–0 (darker pixels are taller) using the provided succulent image. Printed on a desktop FDM printer using white PLA filament*



Figure 10-27. *3D-printed lithophane from Figure 10-26 that has been backlit. Since we mapped the pixel intensity values to 6–0, darker pixels are taller, which means the corresponding part of the print is thicker, allowing less light to shine through*

Summary

In this chapter, we explored two approaches for generating a relief tile from an input image, by using Image Sample and using Import Image with the help of Python 3, respectively. We then discussed several ways to use the image tiles, including projecting them onto other surfaces as decorative elements, using them as the bases for creating mold boxes, and 3D printing lithophanes that appear as 3D photos when backlit.

The first image tile generator uses the Image Sampler node to parse the input image into a 2D array (data tree) of pixel intensity values, which are then mapped to a custom height range for building a relief surface. A bottom is created by filling in the perimeter curve of the tile rectangle. The sidewalls are created by creating vertical segments all around the tile and lofting them.

The second image tile generator uses the Import Image node to convert an image into a flat mesh structure with embedded color information, which is then used to create a relief mesh with vertices offset along Z based on their corresponding pixel intensities. A Python 3 Script node is implemented to extract the perimeter of the relief mesh, which is then used to create the tile bottom and sidewalls.

The next part of the chapter discussed how to project an image tile onto a sphere or a cylinder as decorative elements using Splop. After that we went over how to extend the first tile generator to create a mold box, by inverting the relief surface such that lighter pixels are taller, and generating walls for the mold box by creating a surface between the tile's bottom perimeter and a scaled-up copy, and extruding it along Z.

Finally, we saw a 3D-printed example of a lithophane, which can be generated by either tile generator we created in this chapter, with a range of *higher to lower* for mapping pixel intensities to relief surface heights, such that darker pixels are taller or thicker in the printed tile, therefore allowing less light to shine through.

CHAPTER 11

L-System

This chapter is all about L-System—a family of grammars naturally suited for representing self-repeating patterns like fractals and trees. You'll start by rewriting L-System strings using pen and paper, before implementing the process as Python code. You'll then create your own turtle graphics style object to generate 2D curves from L-System strings. Using 2D trees as examples, you'll investigate how to modify grammars to influence the type of curves they'll generate. After that, you'll take the turtle from 2D to 3D and generate 3D trees. Finally, you'll look at how to generate a canopy which resembles an abstracted foliage to sit atop a tree.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Download the files and navigate to the `/Ch11` folder, where you'll find `*.gh` files named after the figures in which they appear along with a short description. For example, `/Ch11/Figure11-5_rewriting.gh`. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, feel free to build your own node graphs by following the steps described in the text or start with the downloaded `*.gh` files and modify their parameters to experiment.

Some source code and figures in this chapter are derived from Chapter 7 of the book *Procedural Content Generation for Games* also by the author (Lupiani, 2025).

What Is an L-System?

L-System is a type of grammar used to encode construction rules for self-repeating structures that consist of increasingly smaller copies of itself. For example, the fern in Figure 11-1 exhibits the same branching pattern in its main trunk as well as side

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_11

branches when repeatedly zoomed in. Many organisms in nature have self-repeating patterns, including trees, flowers, fruits, etc., which make L-System a great mechanism for encoding rules for their construction.

An L-System grammar consists of an axiom, the starting string, and one or more rewriting rules that tell how to expand that axiom. Each rewriting rule has a *predecessor* (left-hand side), which is a character, and a *successor* (right-hand side) which may be a character or a string, for example, a → ab.

Rewriting an L-System for N iterations means to start with the axiom and repeatedly apply the rewriting rules N times. In each iteration, you identify *all matches* in the current string to the predecessor of any rule and replace them with the successor of that rule—a process called *simultaneous rewriting*. The string at the end of rewriting is called an "expansion," which consists of symbols from the grammar's *alphabet* and is typically longer than the axiom, though not necessarily.

By itself, the expansion string does not mean much. It is up to the artist to *interpret* the string by attributing meaning to each symbol. For example, as we'll see shortly, the string "F+F" means to draw a straight line that is one unit long, turn left, and draw another line that is one unit long. In this chapter, we'll be implementing a *turtle graphics* style object in Python that interprets an expansion string and generates 2D or 3D geometry from it in Grasshopper.

A turtle graphics object originates from a drawing tool introduced by the Logo programming language, which is later adapted to other languages including Python (in the module `turtle`). In turtle-style drawing, you can move forward a fixed distance per step, whether laying down a polyline or not, or turn, pitch, or roll to change direction. We will not be using Python's `turtle` in this chapter but implementing our own.

One thing to note is although you are free to assign meaning to L-System symbols any way you like, there are symbols commonly used in turtle graphics systems with reserved meanings, such as + for turning left, that you should avoid changing to not confuse your users.

Note There are other grammars that use "sequential rewriting," which replaces one match in the current string at a time instead of all matches.



Figure 11-1. *A fern plant exhibiting the same branching pattern at increasingly smaller scales. Left shows the whole plant, middle zooms in on one of the five major branches, and the right zooms in further on one of the minor branches. You can see the same pattern at all three levels. Photo by the author*

Rewriting Strings

In this section, we'll start by rewriting a L-System grammar on pen and paper followed by automating the process with a custom Python 3 Script node.

Using Pen and Paper to Rewrite the String for a Koch Curve Variation

Let's practice rewriting the following L-System grammar, which defines a variation of the Koch curve:

- *Axiom*: F
- *Rewriting rule*: F → F+F--F+F

This L-System has just one rewriting rule. F means to move forward one unit while laying down a line (when used as the axiom, it means to start with a line segment one unit long). Furthermore, + and - mean to turn left and right in place, respectively.

To produce the iterations shown in Figure 11-2, we'll use an angle of 75 degrees for each turn, although in general, you can adjust the angle to any amount. Typically, one angle is used as the unit for all turns, pitches, and rolls. For example, theta = 75 degrees means every time the turtle encounters a symbol to turn right, pitch down, etc.; it should do so by 75 degrees relative to its current heading and orientation.

You may have noticed we said move "forward" without specifying a direction. "Forward" at the starting point (or the turtle's initial heading) is specified parametrically by the user (+Y is used in Figure 11-2). Every F thereafter means to move forward along the turtle's current heading, whichever way it happens to be at that point in time.

With the guidelines in place, let's start rewriting the Koch string step by step for a total of four iterations. The corresponding curves are shown in Figure 11-2.

Axiom: The axiom is F, which is a line segment of unit length. We use +Y as the initial heading (which is an arbitrary choice).

Iteration 1: We apply the rewriting rule to replace the single F in the axiom with F+F--F+F. Since we define + and - as "turn left/right 75 degrees," this interprets the expansion as "forward 1 step, turn left 75 degrees, forward 1 step, turn right 75 degrees, turn right 75 degrees, forward 1 step, turn left 75 degrees, forward 1 step," as visualized in Figure 11-2.

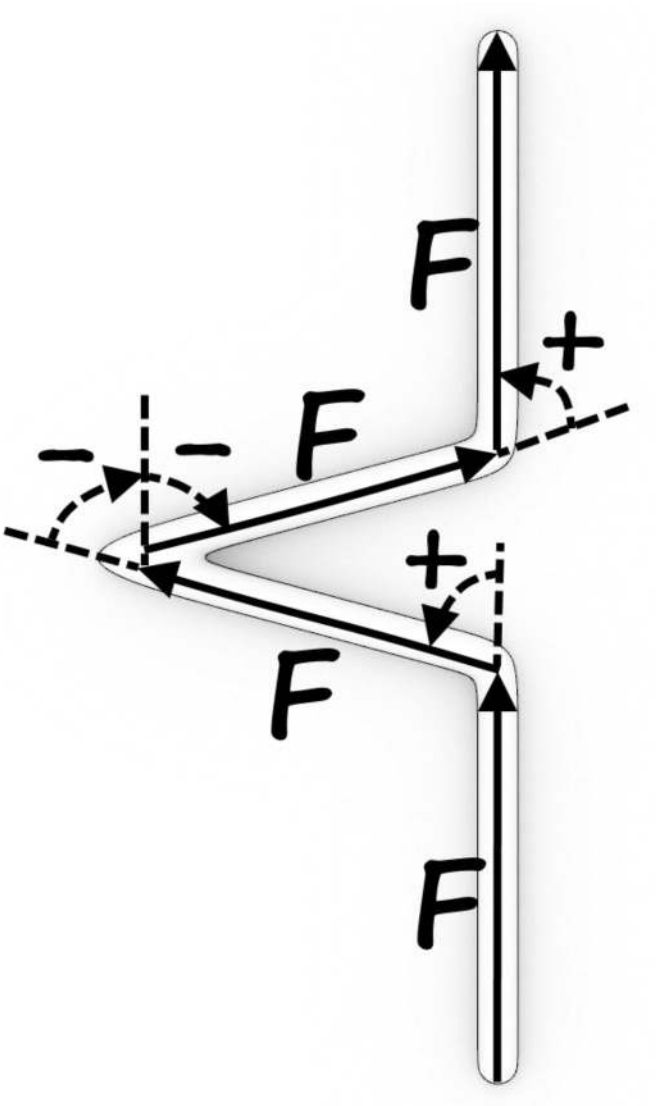


Figure 11-2. *Visualizing how the turtle interprets the string F+F--F+F. The turtle draws a one-unit forward segment, turns left 75 degrees, draws a one-unit forward segment, turns right 75 degrees, turns right 75 degrees, draws one-unit forward segment, turns left 75 degrees, and draws a one-unit forward segment*

Iteration 2: We start with F+F--F+F and replace every F with F+F--F+F to get

F+F--F+F+**F+F--F+F**--**F+F--F+F**+**F+F--F+F**

which corresponds to replacing every unit line segment in the iteration #1 polyline with a copy of itself, as shown in Figure 11-3.

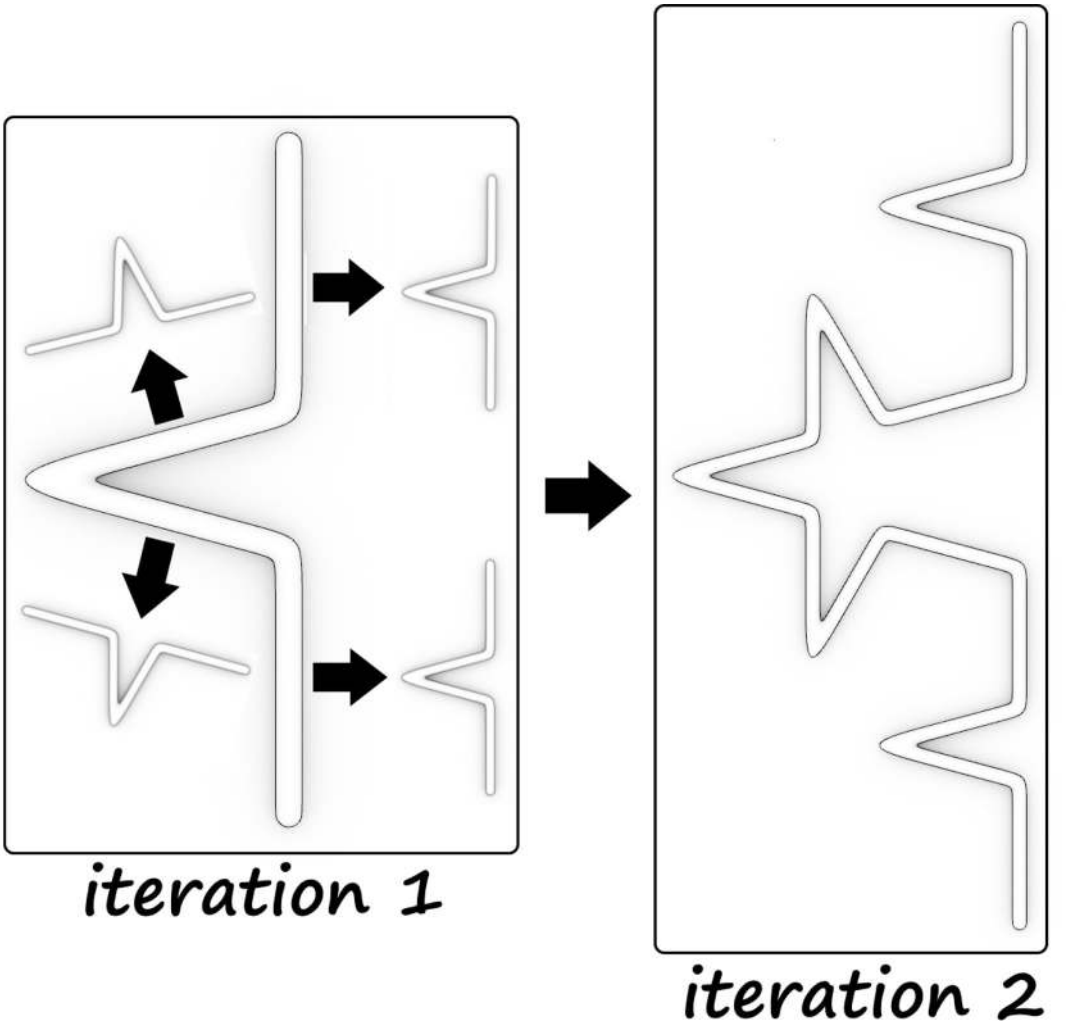


Figure 11-3. *Replacing every unit segment in iteration 1 with the right-hand side of the rewriting rule to derive iteration 2*

Iteration 3: We start with F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F and replace every F with F+F--F+F to get

F+F--F+F +**F+F--F+F**--**F+F--F+F**+**F+F--F+F**+**F+F--F+F**+**F+F--F+F**--**F+F--F+F**+**F+F--F+F**--**F+F--F+F**+**F+F--F+F**--**F+F--F+F**+**F+F--F+F**--**F+F--F+F**+**F+F--F+F**+**F+F--F+F**+**F+F--F+F**--**F+F--F+F**+ **F+F--F+F.**

You can see by now it's become difficult to apply the rewriting rule by hand. It is still relatively intuitive to visualize, however, by replacing every unit segment by a copy of iteration #1, as shown in Figure 11-4.

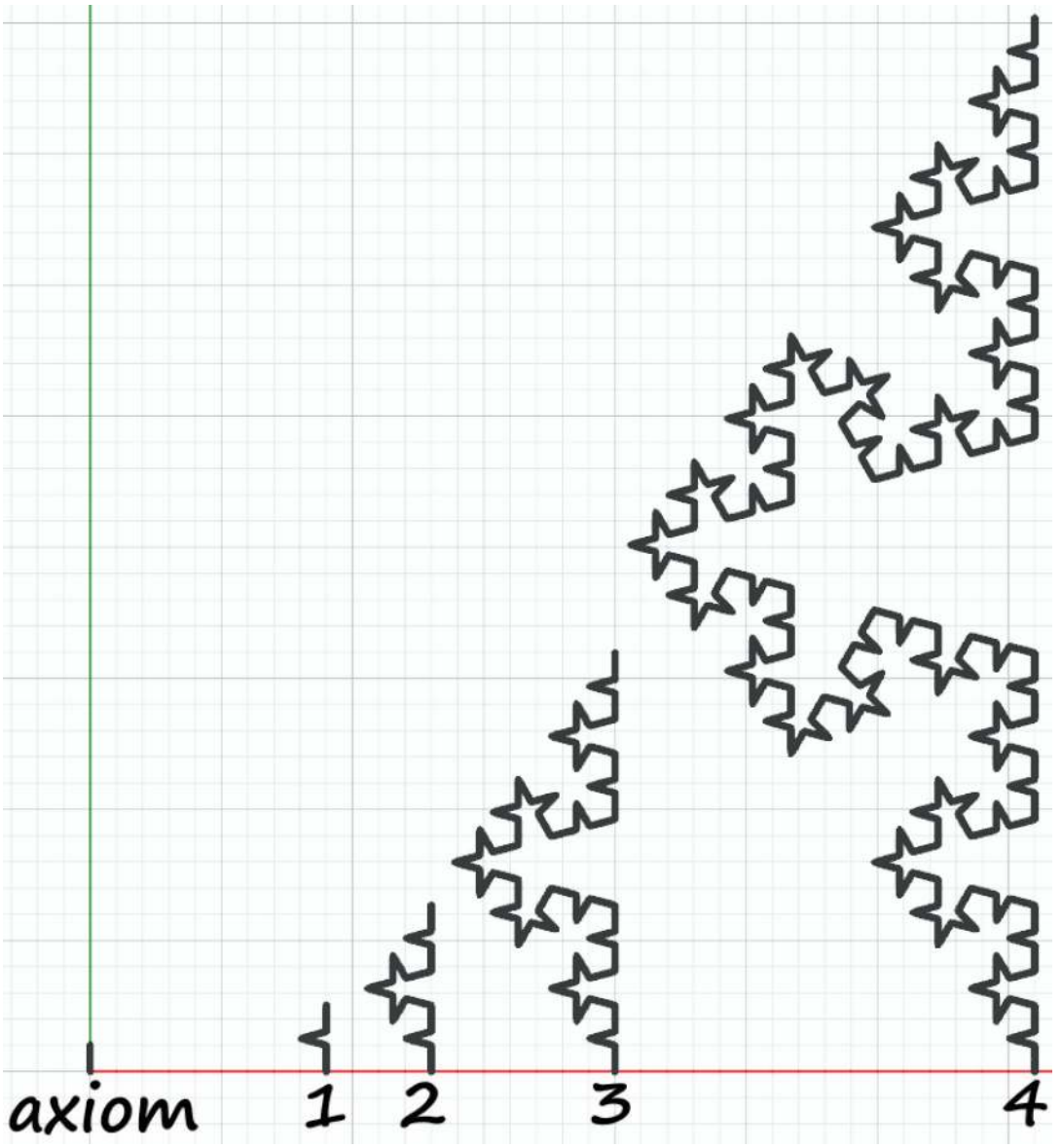


Figure 11-4. *The interpreted polylines corresponding to the axiom, the first, second, third, and fourth iteration of the modified Koch curve with 75-degree turn angle and step distance of 1*

Iteration 4: We start with F+F--F+F +F+F--F+F--F+F--F+F+F+F--F+F+F+F--F+F+F+ F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F+F+F--F+F+F+ F--F+F--F+F--F+F+ F+F--F+F and replace every F with F+F--F+F, which gives us

F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F + F+F--F+F + F+F--F+F -- F+F--F+F + F+F--F+F. The corresponding polyline is shown in Figure 11-4.

Phew! You can see that in as few as four iterations, it has become impractical to rewrite strings by hand. In the next section, we'll discuss how to automate the rewriting process with Python.

Using Python to Rewrite Strings

We'll now use Python to automate the rewriting process. Figure 11-5 shows the Python 3 Script node along with sample inputs and outputs. The node takes three inputs:

1. `axiom`: As a string (Type Hint: str), with or without quotes. Here, we set it to the string b using a Panel.
2. `rewriting_rules`: Formatted as a Python dictionary but given as a string (Type Hint: str), with each item surrounded by double quotes. For example, as shown in Figure 11-5, the dictionary contains two rules, a → ab, given as "a": "ab", and b → a, given as "b": "a". The rules are listed in order, separated by comma(s), and surrounded by curly braces: {"a": "ab", "b": "a"}. This formatting is required for the `json` module to properly parse the string into a Python dictionary.
3. `iterations`: As an integer (Type Hint: int), to indicate the total number of iterations to rewrite the string. Here, we set it to five using a Number Slider.

The node has a single output, `str_output`, which is the expansion after the given number of iterations (as shown, abaababa). We retain the `out` port that comes default with the Python 3 Script node when you first deploy it onto the canvas, so we can print

the rewritten string after each iteration and verify their accuracy. The Type and Access Hints for the Python 3 Script node are summarized in Table 11-1.

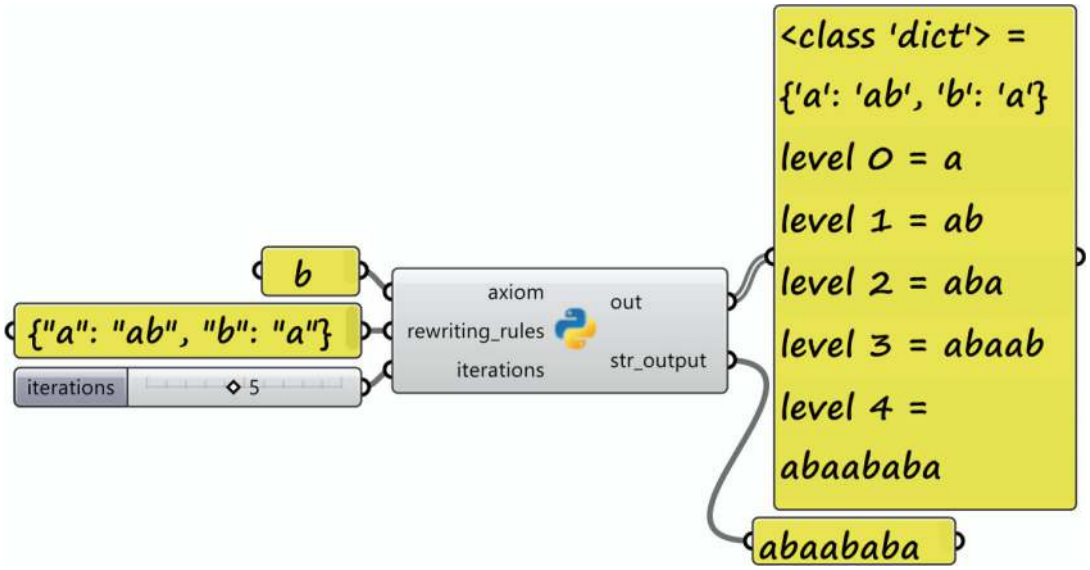


Figure 11-5. *Python 3 Script node that rewrites an L-System grammar for a given number of iterations. The node takes the grammar's axiom (b) and rewriting rules (a → ab, b → a) as strings, with the number of iterations (an integer) set via a Number Slider. As shown, the node prints the rewritten string after each iteration via the standard out stream (out). The expansion after four iterations is abaababa*

Table 11-1. *Access and Type Hints for the Python 3 Script node in Figure 11-5*

Name	Input or Output	Access	Type Hints	Purpose
axiom	input	Item Access	str	Starting string for the L-System grammar. Can be given with or without double quotes
rewriting_rules	input	Item Access	str	Set of rewriting rules as a string with each term enclosed in double quotes, to be parsed by the `json` module into a Python dictionary
iterations	input	Item Access	int	Number of iterations to rewrite the string
str_output	output	N/A	str	The expansion

The code inside the Python 3 Script node (Figure 11-5) is divided into two parts, with Listing 11-1 showing the `rewrite` function which expands a string once based on the set of rewriting rules and Listing 11-2 showing the `rewrite_recur` function that repeatedly applies the set of rewriting rules for the total number of iterations. Listing 11-2 also contains the block of code that parses `rewriting_rules` into a Python dictionary and calls `rewrite_recur` to kick off the rewriting process.

Let's look more closely at the `rewrite` function in Listing 11-1. We start by creating an empty string `str_out` to accumulate the rewritten bits. We then iterate the input string in a `for` loop and check each character against every rule in a second `for` loop nested under it. If the character matches a rule's predecessor (left-hand side), we add the rule's successor to the end of `str_out`. As soon as we've applied one matching rule for the current character, we can break out of the nested loop and continue with the rest of the input string. If no matching rule is found (`not rewritten`), we simply append the character as is to the end of `str_out`.

Listing 11-1. Function to rewrite a string one time based on a given set of rewriting rules, from inside the Python 3 Script node in Figure 11-5 (see `/Ch11/Figure11-5_rewriting.gh`)

```
def rewrite(str_to_rewrite: str, rewriting_rules: dict):
    str_out = ""
    for c in str_to_rewrite:
        rewritten = False
        for predecessor, successor ↩
            in rewriting_rules.items():
            if c == predecessor:
                str_out = "".join((str_out, successor))
                rewritten = True
                break
        if not rewritten:
            str_out = "".join((str_out, c))

    return str_out
```

Note When no matching rewriting rule is found, we simply copy the character as is from the input string to the rewritten string.

We'll now take a closer look at `rewrite_recur` in Listing 11-2. `rewrite_recur` is a *recursive* function—which is a function that calls itself over and over until some terminating condition is met. The block of code at the end of Listing 11-2 calls `rewrite_recur` the first time when the Python 3 Script node is run. After that, `rewrite_recur` calls itself repeatedly until it reaches the total number of iterations (`level == max_level`).

Each time `rewrite_recur` is called, it calls `rewrite` (Listing 11-1) to rewrite the current string, prints the result to the standard output stream (`out`), and increments the iteration number by one (`level += 1`).

The block of code at the bottom of Listing 11-2 (outside of the definition of `rewrite_recur`) serves as the entry point into the Python 3 Script node. The code uses the `json` module (imported at the top via `import json`) to parse the `rewriting_rules` input into a Python dictionary (`rr`). The type and contents of the dictionary are printed next (to `out`), which should read "class <'dict'> = {'a': 'ab', 'b': 'a'}". The next line of code kicks off the rewriting process by calling `rewrite_recur` with an initial `level` of 0 and assigns the expansion returned by `rewrite_recur` to `str_output`, the output of the Python 3 Script node.

Listing 11-2. Rewriting a string recursively for a given number of iterations, from inside the Python 3 Script node in Figure 11-5 (see `/Ch11/Figure11-5_rewriting.gh`)

```
import json

def rewrite_recur(str_to_rewrite: str, ↩
    rewriting_rules: dict, level: int, max_level: int):
    if level == max_level:
        return str_to_rewrite

    str_to_rewrite = rewrite(str_to_rewrite, rewriting_rules)
    print("level " + str(level) + " = " + str_to_rewrite)
    level += 1
    return rewrite_recur(str_to_rewrite, ↩
        rewriting_rules, level, max_level)
```

```
rr = json.loads(rewriting_rules)
print(str(type(rr)) + " = " + str(rr))
str_output = rewrite_recur(axiom, rr, 0, iterations)
```

Implementing a Turtle Graphics Object to Create 2D Polylines (Continuous Line Drawing) Based on Expansion Strings

In this section, we'll implement a second Python 3 Script node to create 2D polylines based on strings rewritten from L-System grammars. Table 11-2 summarizes the Access and Type Hints for the node, and Listing 11-3 shows the code.

Table 11-2. *Access and Type Hints for the Python 3 Script node introduced in Figure 11-6*

Name	Input or Output	Access	Type Hints	Purpose
rewritten_str	input	Item Access	str	String rewritten for a desired number of iterations based on a given L-System grammar, ready to be interpreted by the turtle graphics object
theta	input	Item Access	float	Angle to turn left or right whenever the turtle encounters a + or -
step_dist	input	Item Access	float	Whenever the turtle encounters a F (or f), the amount of distance to move forward
polyline	output	N/A	**ghdoc Object**	**Polyline created by the turtle, as an accumulation of line segments from F symbols encountered in the rewritten string**

Listing 11-3. Code for the Python 3 Script node in Table 11-2 and Figure 11-6, which implements a turtle_2D class for creating 2D polylines based on strings rewritten from L-System grammars (see /Ch11/Figure11-6_turtle2D.gh)

```
import Rhino.Geometry as rg
from math import radians

class turtle_2D:
    def __init__(self, rewritten_str: str, ↩
        theta: float,  step_dist: float):
        self.rewritten_str = rewritten_str
        self.theta = theta
        self.step_dist = step_dist

        self.polyline = [rg.Polyline()]
        self.polyline[-1].Add(0.0, 0.0, 0.0)
        self.lcs = rg.Transform.Identity
        self.heading = rg.Vector3d(rg.Point3d(0.0, 1.0, 0.0))

    def update_heading(self, mat_rot):
        self.lcs = self.lcs * mat_rot
        self.heading = rg.Vector3d( ↩
            rg.Point3d(self.lcs.M10, self.lcs.M11, ↩
            self.lcs.M12))
        self.heading.Unitize()

    def draw(self):
        for c in self.rewritten_str:
            last_pt_index = self.polyline[-1].Count-1
            start_pt = self.polyline[-1].PointAt( ↩
                last_pt_index)

            if c == "F":
                self.one_step_forward(start_pt)
            elif c == "+":  # Turn LEFT
                mat_rot = rg.Transform.Rotation( ↩
                    radians(-self.theta), rg.Vector3d( ↩
                    rg.Point3d(self.lcs.M20, self.lcs.M21, ↩
```

```
                    self.lcs.M22)), start_pt)
                self.update_heading(mat_rot)
            elif c == "-":  # Turn RIGHT
                mat_rot = rg.Transform.Rotation( ↩
                    radians(self.theta), rg.Vector3d( ↩
                    rg.Point3d(self.lcs.M20, self.lcs.M21, ↩
                    self.lcs.M22)), start_pt)
                self.update_heading(mat_rot)

    def one_step_forward(self, start_pt):
        trans_vec = self.heading * self.step_dist
        end_pt = start_pt + trans_vec
        self.polyline[-1].Add(end_pt[0], ↩
            end_pt[1], end_pt[2])

t = turtle_2D(rewritten_str, theta, step_dist)
t.draw()
polyline = t.polyline
```

Let's take a closer look at the code in Listing 11-3. The class `turtle_2D` implements our turtle graphics object, which creates polylines based on a given string rewritten from an L-System grammar.

The `__init__` method takes the rewritten string (`rewritten_str`), the angle for turning (`theta`), and the unit distance for moving forward (`step_dist`) as inputs and saves them as member variables. The next line creates a list for the turtle to store the polylines it will create (`self.polyline`), initialized with an empty Polyline object (`rg.Polyline()`). The following line adds the point (0.0, 0.0, 0.0) to the Polyline object. Next, a matrix is created to save the turtle's local coordinate system (`self.lcs`, or current orientation), initially as a 3x3 identity matrix (`rg.Transform.Identity`). Finally, a `Vector3d` object initialized to Unit Y is created to keep track of the turtle's current heading (`self.heading`).

The `update_heading` method is called to update the turtle's orientation (`self.lcs`) and heading (`self.heading`) whenever it turns. First, `self.lcs` is multiplied with the rotation matrix passed in (`mat_rot`) to form the new orientation. The +Y of the new orientation (`self.lcs.M10, self.lcs.M11, self.lcs.M12`) then becomes the new heading. Finally, the new heading is unitized such that its length becomes one, so whenever the turtle needs to move forward, we can simply multiply its current heading (`self.heading`) with the unit distance (`self.step_dist`) to find the new position.

We'll look at the `one_step_forward` method next, which is called to update the turtle's position whenever it needs to move forward one step. The method takes a starting point (`start_pt`) as input and adds the movement computed from multiplying the turtle's heading (`self.heading`) with the step distance (`self.step_dist`) to find the new position (`end_pt`). It then adds the new position to the end of the last polyline. Since we've not added the ability for the turtle to skip yet (move without leaving a mark), the turtle will create only a single polyline—however, setting the turtle up preemptively to expect more than one polyline will prepare us to add skipping later with ease. Note that to add a point to a Polyline object, you must call its `Add` method with the point's XYZ coordinates as three numbers (`end_pt[0], end_pt[1], end_pt[2]`).

The `draw` method creates the polyline(s) based on the rewritten string. It iterates through the string in a `for` loop. For each character, it looks up the last point of the most recent polyline to use as the starting point. Depending on what the character is, it either moves forward one unit (for F) by calling `one_step_forward`, turns left (for +), or turns right (for -). Since we use +Y of the turtle's local coordinate system (`self.lcs`) as its heading, to turn left or right, the turtle should rotate about +Z of `self.lcs`, which is specified via the line

```
rg.Vector3d(rg.Point3d( ↩
    self.lcs.M20,self.lcs.M21,self.lcs.M22)
```

To turn left (for +), the angle is `-self.theta` (note the negative sign), and to turn right, the angle is `self.theta`. In either case, you call `rg.Transform.Rotation` to create the rotation matrix with the parameters we've just discussed and use it to call `update_heading` to perform the turn. Note that `self.theta` is in degrees but `rg.Transform.Rotation` expects angles in *radians*, so we must convert `self.theta` to radians first.

Figure 11-6 shows two node graphs composed with the Python 3 Script node for rewriting strings (Figure 11-5) and the node implemented in this section. The top graph has the grammar for drawing a left-leaning square as input:

- *axiom*: F
- rewriting_rules: {"F": "F+F+F+F"}
- *iterations*: 1
- *theta*: 90 degrees
- step_dist: 1

Figure 11-7 shows the square traced by the turtle step by step, as it interprets each character and either moves forward or turns.

The bottom graph in Figure 11-6 has the grammar for the modified Koch curve we've used in the pen-and-paper example for rewriting strings (Figure 11-4):

- *axiom*: F
- *rewritin_rules*: {"F": "F+F--F+F"}
- *iterations*: 4
- *theta*: 75 degrees
- step_dist: 1

Note The *identify matrix* is a matrix of any size (2x2, 3x3, etc.) with all 1's along its diagonal and all 0's otherwise. If you multiply any matrix M with the identity matrix, you get M unchanged as the result.

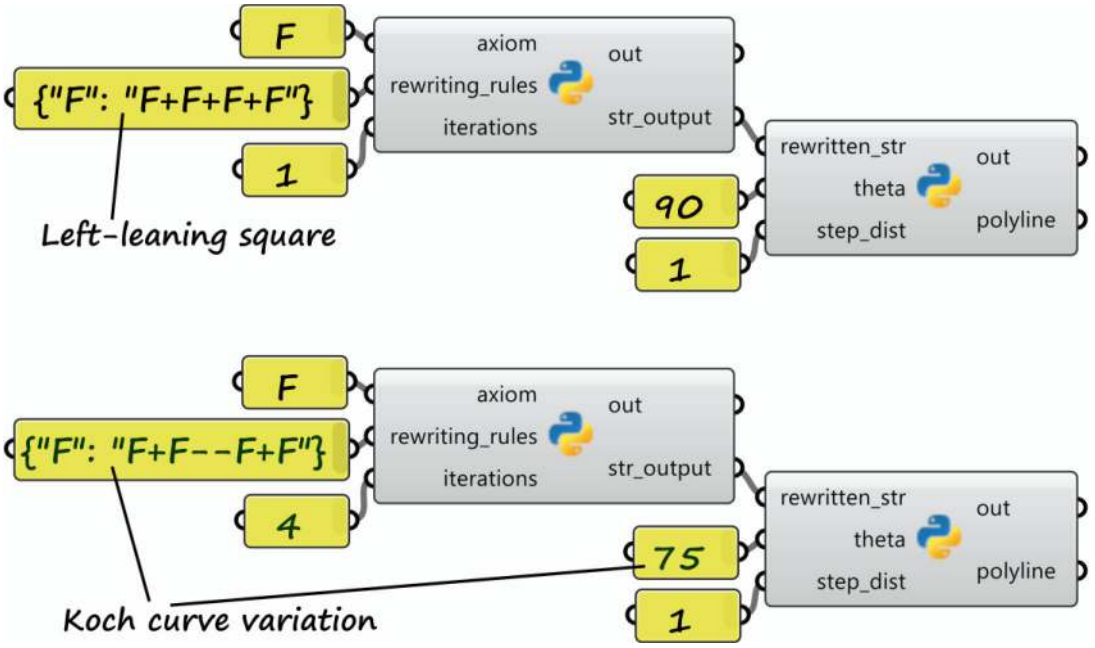


Figure 11-6. *Composing the Python 3 Script node for rewriting strings (Figure 11-5) with the Python 3 Script node for drawing continuous 2D polylines. Top shows the grammar for a left-leaning square as input (axiom = F; rewriting_rules = {"F": "F+F+F+F"}). Bottom shows the grammar for the modified Koch curve from the pen-and-paper example as input (axiom = F; rewriting rules = {"F": "F+F--F+F"})*

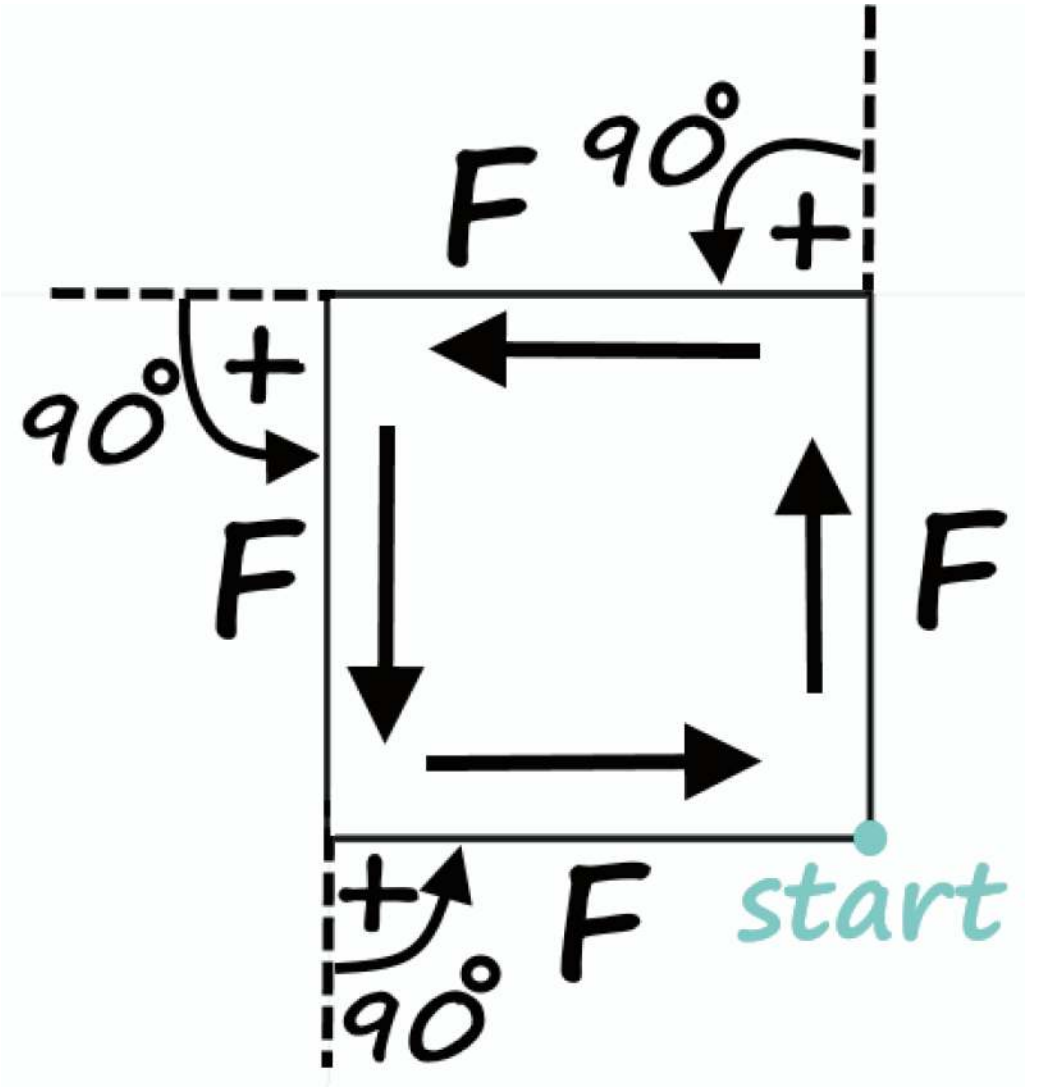


Figure 11-7. *Creating the polyline step by step based on the rewritten string F+F+F+F, step distance of one, and turn angle of 90 degrees*

Two-Dimensional Continuous Line Variations

Table 11-3 lists eight examples of L-System grammars that use only the symbols F, +, and/or - (Prusinkiewicz et al., 1990), which produce continuous line drawings, that is, a single continuous polyline, where the turtle keeps the pen on the paper and always leaves a mark whenever it moves forward. The corresponding polylines are shown in

Figures 11-8 and 11-9. You can find the Grasshopper definitions containing all eight examples in the downloaded source files for the chapter (`/Ch11/Figure11-8_turtle2D_variations1.gh` and `/Ch11/Figure11-9_turtle2D_variations2.gh`). You can see that although all the grammars in Table 11-3 are relatively simple, they produce a wide variety of complex curves, which shows the power and range of L-System grammars.

Table 11-3. *Parameters for the turtle_2D class (node graph in Figure 11-6) responsible for producing the "continuous line drawings" in Figures 11-8 and 11-9 (see `/Ch11/Figure11-8_turtle2D_variations1.gh` and `/Ch11/Figure11-9_turtle2D_variations2.gh`)*

Axiom	Rewriting Rules	No. Iterations	Angle	Step Dist.	Figure
F-F-F-F	{"F": "FF-F-F-F-FF"}	4	90	1	11-8, 1
F-F-F-F	{"F": "F-F+F+FF-F-F+F"}	3	90	1	11-8, 2
F-F-F-F	{"F": "FF-F+F-F-FF"}	3	90	1	11-8, 3
F-F-F-F	{"F": "FF-F-F-F-F-F+F"}	4	90	1	11-8, 4
F-F-F-F	{"F": "FF-F--F-F"}	4	90	1	11-9, 1
F-F-F-F	{"F": "F-F+F-F-F"}	4	90	1	11-9, 2
F-F-F-F	{"F": "F-FF--F-F"}	5	90	1	11-9, 3
F-F-F-F-F	{"F": "F+F--F+F"}	4	60	1	11-9, 4

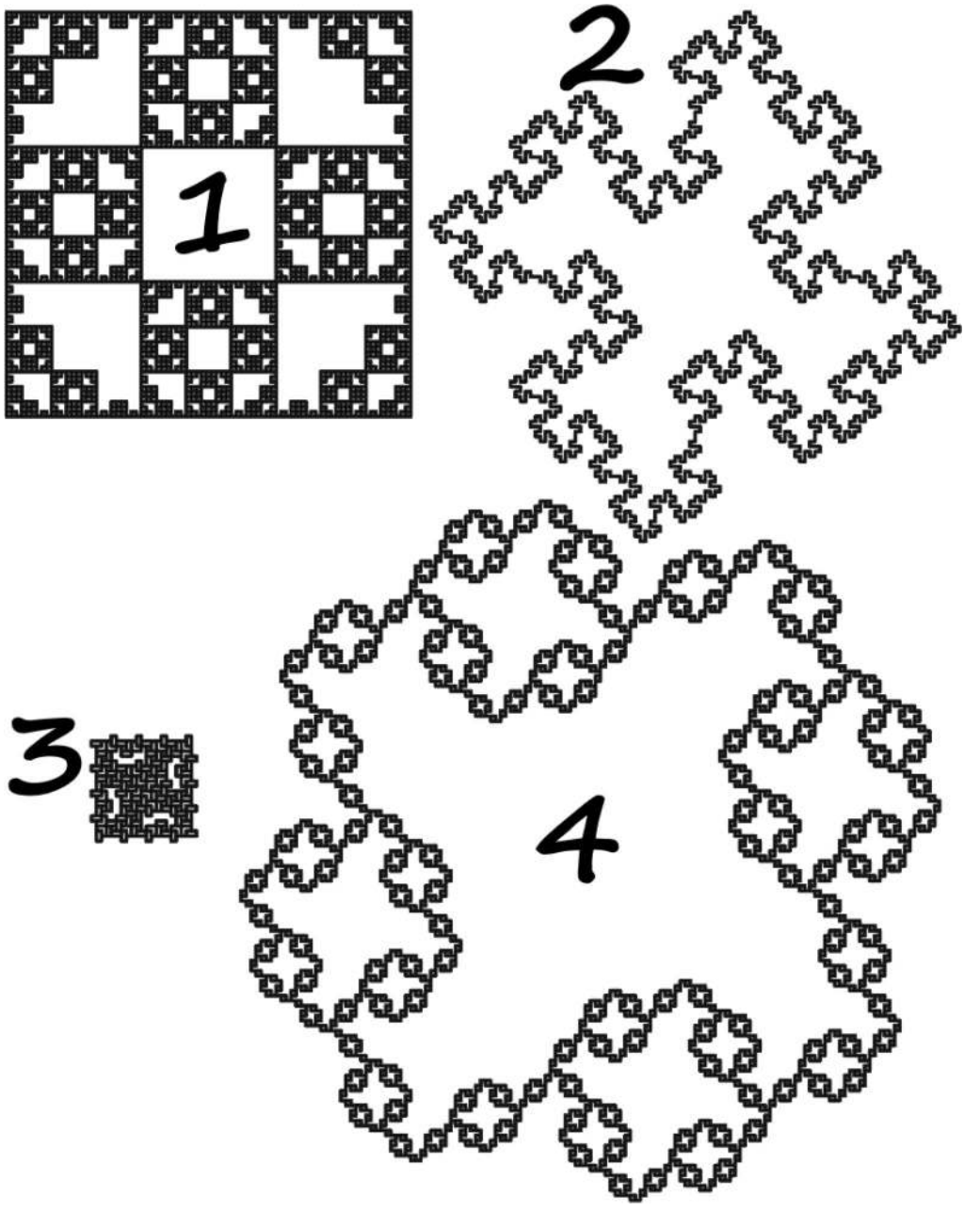


Figure 11-8. *Polylines produced from the grammars listed in the first four rows of Table 11-3. Each is a single continuous polyline (see* `/Ch11/Figure11-8_turtle2D_variations1.gh`*)*

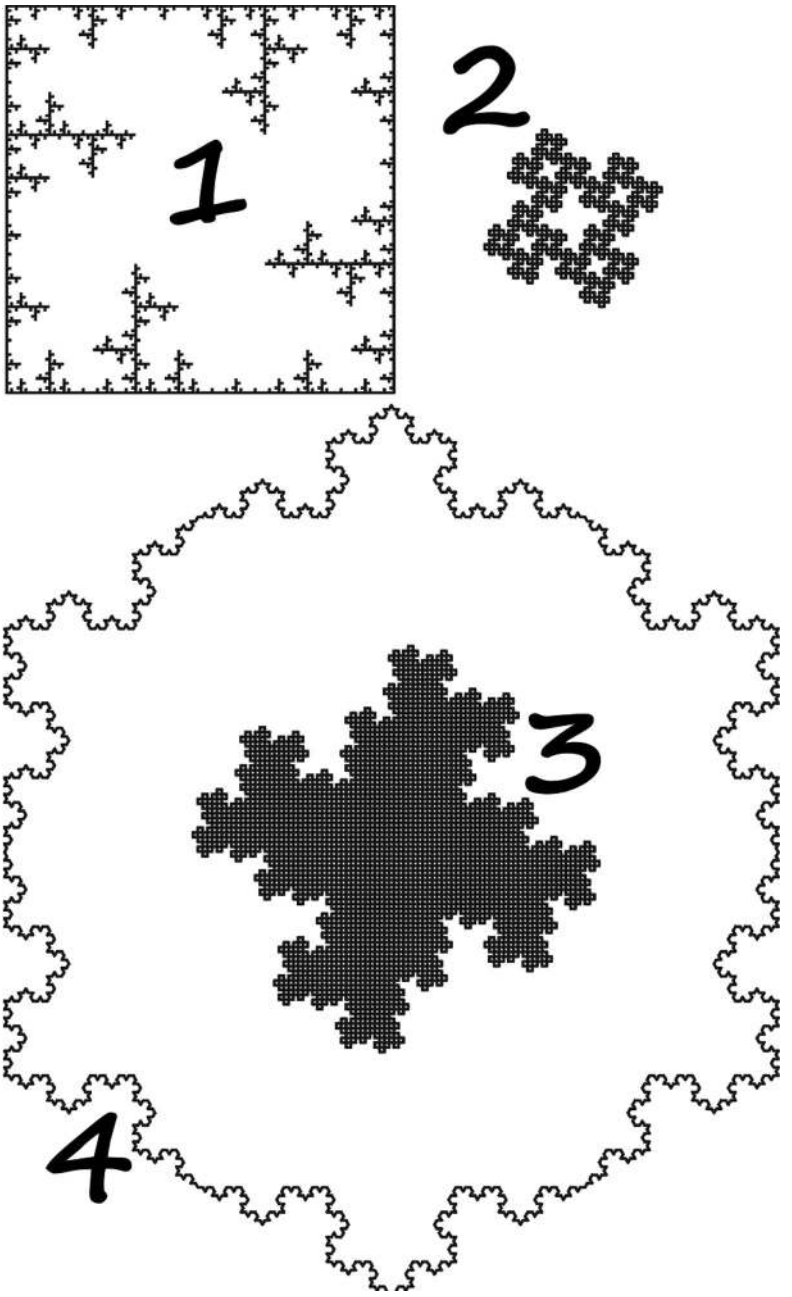


Figure 11-9. *Polylines produced from the grammars listed from the fifth to eighth rows from the top of Table 11-3. Each is a single continuous polyline (see* `/Ch11/Figure11-9_turtle2D_variations2.gh`*)*

Adding Ability to the 2D Turtle to Lift the Pen Off the Page

In this section, we'll add a simple addition to the turtle to allow it to "lift the pen off the page," giving it the option to move forward *without* leaving a mark or without extending the current polyline. This addition allows the turtle to create drawings with disjoint parts, like the "islands and lakes" fractal shown in Figure 11-12 (Prusinkiewicz et al., 1990). The code additions are shown in Listing 11-4.

Listing 11-4. Adding the ability to the turtle to "lift the pen off the page" or move forward without extending the current polyline (see `/Ch11/Figure11-10_11-11_turtle2D_skips.gh`)

```
import Rhino.Geometry as rg
import ghpythonlib.components as ghc
from math import radians

class turtle_2D:
    #### SNIP ####

    def draw(self):
        for c in self.rewritten_str:
            last_pt_index = self.polyline[-1].Count-1
            start_pt = self.polyline[-1].PointAt( ↩
                last_pt_index)
            if c == "F":
                self.one_step_forward(start_pt, True)
            elif c == "f":
                self.one_step_forward(start_pt, False)
            elif c == "+":
                #### SNIP ####
            elif c == "-":
                #### SNIP ####

    def one_step_forward(self, start_pt, ↩
        should_draw: bool = True):
```

```
        if not should_draw:
            self.polyline.append(rg.Polyline())
        #### SNIP ####

#### SNIP ####
```

We modify the `draw` method such that the letter "f" will move the turtle forward without extending the polyline. Currently, `one_step_forward` always extends the polyline; therefore, we must change it to take an extra Boolean argument `should_draw` to distinguish between a forward movement caused by F (extend the line) or f (not extend the line). When `should_draw` is `False` (therefore, "`not should_draw`" is `True`), we add a new Polyline object (`rg.Polyline()`) to the end of the list of polylines (`self.polyline`), so the next point we create will be at the beginning of the new polyline.

Figure 11-10 shows a node graph with the grammar for the "islands and lakes" fractal (Prusinkiewicz et al., 1990) as input:

- *axiom*: F+F+F+F
- *rewriting_rules*: {"F": "F+f-FF+F+FF+Ff+FF-f+FF-F-FF-Ff-FFF", "f": "ffffff"}
- *iterations*: 2
- *theta*: 90 degrees
- step_dist: 10

The set of polylines produced by Figure 11-10 are shown in Figure 11-12. You can see that the f symbols in the grammar have created many disjoint polylines. Figure 11-11 shows the turtle's step-by-step movement for tracing out the axiom and the first iteration.

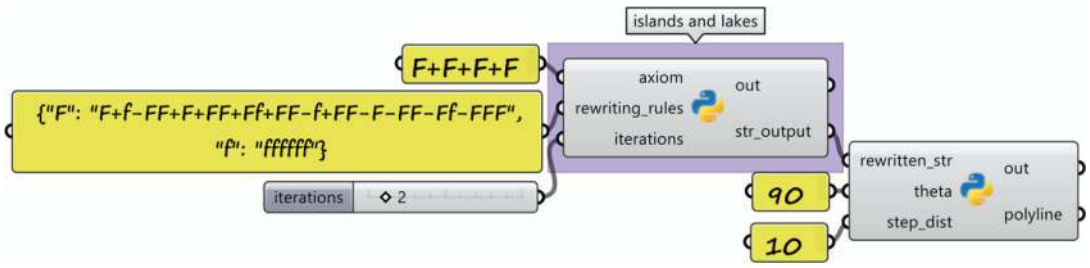


Figure 11-10. *Node graph using the modified Python 3 Script node containing Listing 11-4 and the "islands and lakes" fractal grammar (Prusinkiewicz et al., 1990) as input, with iterations = 2, theta = 90 degrees, and step_dist = 10, which produces the set of polylines depicted in Figure 11-12*

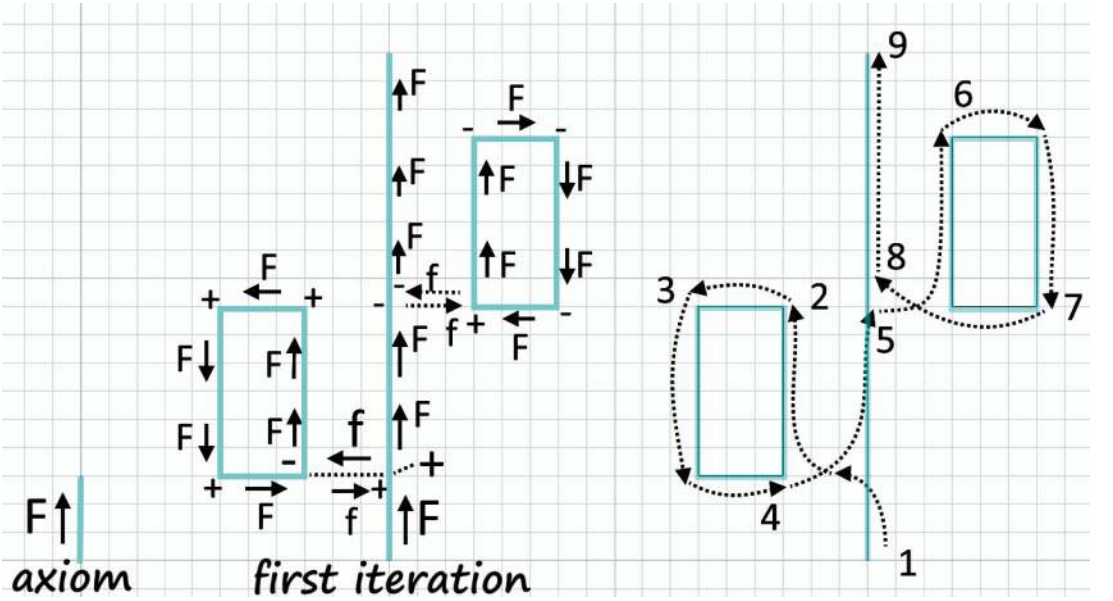


Figure 11-11. *The movement of the turtle tracing out the axiom (left) and the first iteration (middle) of the "islands and lakes" fractal (Prusinkiewicz et al., 1990). The right shows a numbered sequence of the turtle's positions in time instead of per symbol. You can see the turtle draws one unit of the central line, hops over to the left to trace the rectangle counterclockwise, hops back to the middle and advances the central line, and hops to the right to trace the rectangle clockwise and finally back to the middle to finish the central line*

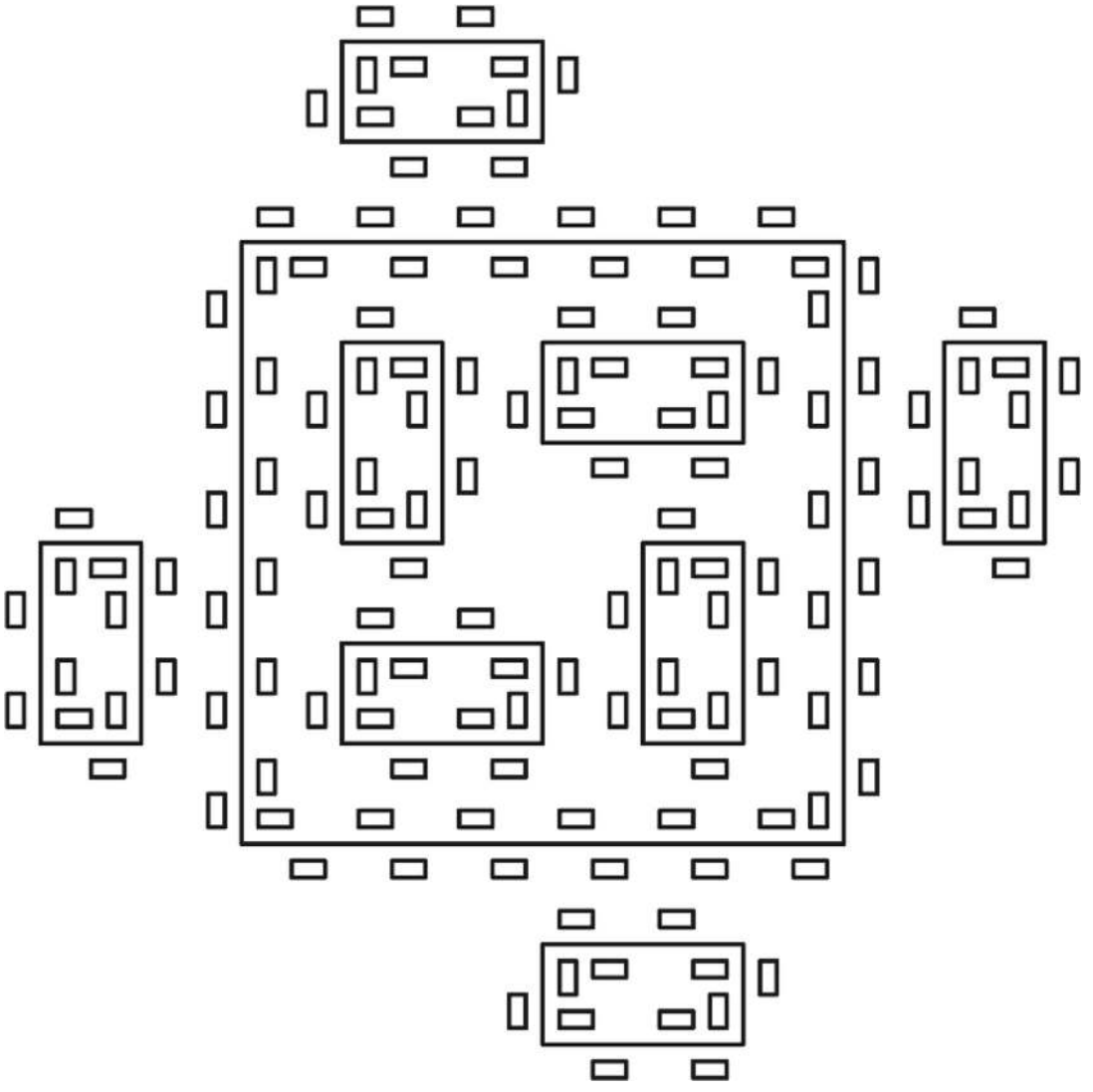


Figure 11-12. *The set of polylines produced by the node graph in Figure 11-10, using the "islands and lakes" fractal grammar (Prusinkiewicz et al., 1990) as input*

Adding Ability to the 2D Turtle to Create Branches

So far, our turtle is able to either extend an existing polyline or create a new polyline by continuously moving forward through space with optional in-place rotations. To create branches for a tree-like structure, however, we must add a new ability to the turtle—to backtrack to a previous part of a polyline and extend the line from there.

Imagine drawing a tree on a piece of paper with the trunk and branches as polylines. Suppose you have just finished drawing a branch B and are now adding a new branch from B. Once you get to the end of the branch (we'll call it B1), you must backtrack to B and pick another spot somewhere along B to start a new branch. The turtle isn't yet capable of doing this, however, since currently its only options are to turn in-place or move in one-step increments from the end of B1. You could argue that it is possible for the turtle to turn until it faces the opposite way, and move as many steps necessary to get back to the beginning of B1, and go from there—while this is possible, it gets unmanageable very quickly when you want to create more complex structures. There is a better way, which is what we'll explore in this section.

Listing 11-5 shows the code additions for the turtle to backtrack and create branches. By "backtracking" we mean to teleport back to an earlier point on the current or earlier polyline and resume the turtle's heading and orientation while it was last at that point in time. There are two symbols we'll add to the L-System alphabet for branching—"[" and "]".

Each time the turtle encounters a "[", it takes a snapshot of its current state, which includes the following:

1. Current polyline index
2. Point index along the current polyline
3. Heading
4. Local coordinate system (orientation)

The turtle will accumulate these snapshots on a stack (`self.stack`)—a data structure resembling a stack of books, where you add and remove only from the top, one item at a time.

Each "]" tells the turtle to pop the stack and remove the most recent snapshot from the top (which we store in `self.prev_popped_pt`). Since we read one character at a time, we interpret "]" and pop the stack in the current iteration, but teleport the turtle at the start of the *following iteration*. Therefore, a Boolean flag `self.prev_step_popped` is needed to track whether a snapshot is popped in the previous iteration so we know to teleport in the current iteration. For convenience, when unpacking a snapshot, we also use a list (`self.last_pt_stats`) to store the polyline index and point index so they are together in a pair. Figure 11-13 visualizes how the snapshots are organized in memory.

Let's go over the changes in the `draw` method. If `self.prev_step_popped` is `True` (due to the previous character being "]"), we unpack the popped snapshot (`self.prev_popped_pt`) into `self.last_pt_stats`, `self.heading`, and `self.lcs`, which contain the

polyline and point indices, turtle's current heading, and turtle's current local coordinate system, respectively; this "teleports" the turtle since we've replaced these parameters with the values unpacked from the snapshot. Afterward, we set `self.prev_step_popped` to `False` and empty the `self.prev_popped_pt` list. We then use the polyline and point indices from the snapshot (`self.last_pt_stats`) to look up the starting point for the turtle to continue drawing (`start_pt`).

The next block of code in `draw` handles the "[" and "]" symbols. For "[", the turtle pushes a copy of the snapshot onto the stack with the line starting with `self.stack.append`. Note that the heading (`self.heading`) and the orientation (`self.lcs`) must each be copied by calling the `Clone()` method (e.g., `self.heading.Clone()`). Next, to simulate the effect of creating a branch, we add a new Polyline object with `start_pt` as its first point. We then update `self.last_pt_stats` with this new Polyline's index (which is the number of polylines minus one, or `len(self.polyline)-1`), and the current point index—which is zero, since we've just started a new polyline. For "]", the turtle simply pops the stack (`self.stack.pop()`), stores the popped item in `self.prev_popped_pt`, and sets `self.prev_step_popped` to `True` so it knows to teleport in the following iteration.

Some changes are also required in the `one_step_forward` method to update `self.last_pt_stats`. For "F" (`should_draw` is `True`), the turtle adds the new point (`end_pt`) resulting from the movement to the end of the polyline with index `self.last_pt_stats[0]` or the first entry in `self.last_pt_stats`. The point index (`self.last_pt_stats[1]`) is also updated after the addition, to the number of points minus one (`self.polyline[self.last_pt_stats[0]].Count-1`). For "f," after a new Polyline object is added, we update `self.last_pt_stats` so that the Polyline index is at number of lines minus one and the point index is at zero.

Listing 11-5. Modifying the turtle_2D class so it can take snapshots of the turtle's position, orientation, and heading, as well as teleporting back to the snapshot (see */Ch11/Figure11-12_11-13_turtle2D_branching.gh*)

```
class turtle_2D:
    def __init__(
        self, rewritten_str: str, theta: float, ↩
        step_dist: float):
        #### SNIP ####
        self.stack = []
```

```
        self.prev_popped_pt = []
        self.prev_step_popped = False
        self.last_pt_stats = [0, 0]

    #### SNIP ####

    def draw(self):
        for c in self.rewritten_str:
            if self.prev_step_popped:
                self.last_pt_stats[0], ↩
                self.last_pt_stats[1], ↩
                self.heading, self.lcs = self.prev_popped_pt

                self.prev_step_popped = False
                self.prev_popped_pt.clear()

            start_pt = self.polyline[↩
                self.last_pt_stats[0]].PointAt(↩
                    self.last_pt_stats[1])

            if c == "[":
                self.stack.append([self.last_pt_stats[0], ↩
                    self.last_pt_stats[1], ↩
                    self.heading.Clone(), ↩
                    self.lcs.Clone()])
                self.polyline.append(rg.Polyline())
                self.polyline[-1].Add(↩
                    start_pt[0], start_pt[1], start_pt[2])
                self.last_pt_stats = ↩
                    [len(self.polyline)-1, 0]
            elif c == "]":
                if len(self.stack) > 0:
                    self.prev_popped_pt = self.stack.pop()
                    self.prev_step_popped = True
            #### SNIP ####
```

```
    def one_step_forward(self, start_pt, ↩
        should_draw: bool = True):
        #### SNIP ####
        if should_draw:
            self.polyline[self.last_pt_stats[0]].Add( ↩
                end_pt[0], end_pt[1], end_pt[2])
            self.last_pt_stats[1] = ↩
                self.polyline[self.last_pt_stats[0]].Count-1
        else:
            #### SNIP ####
            self.last_pt_stats = [len(self.polyline) - 1, 0]

#### SNIP ####
```

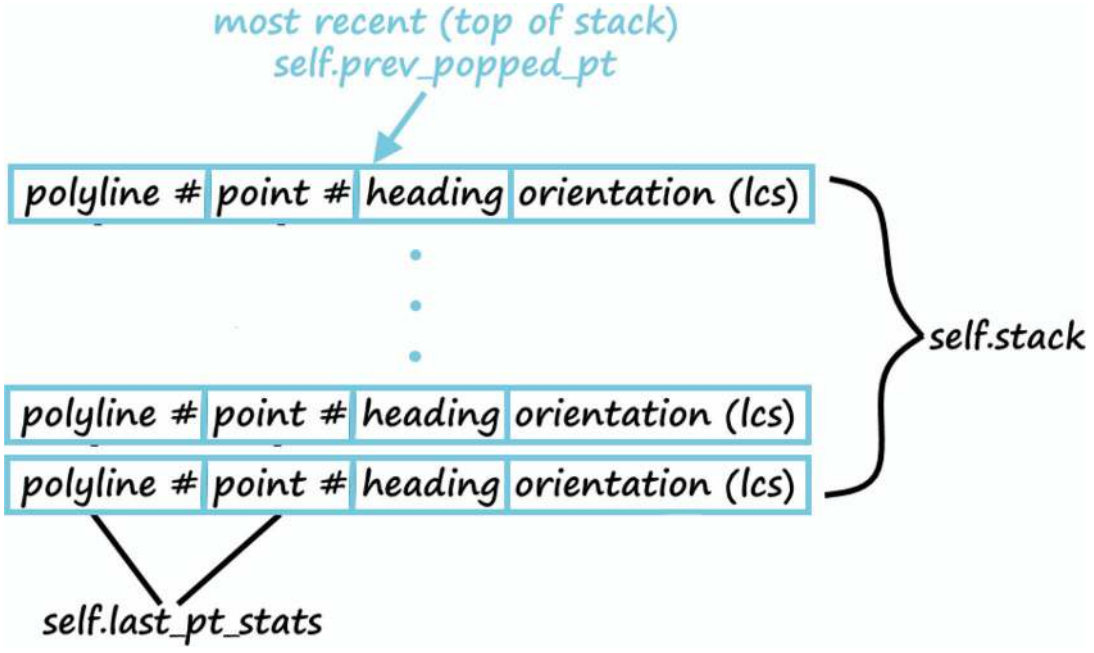


Figure 11-13. *A visualization of the turtle's stack for keeping track of snapshots. Each snapshot is made of a list of four items: the polyline index, point index along the polyline, heading, and local coordinate system (orientation). Each time the turtle encounters a "[" while iterating the rewritten string, it pushes a snapshot onto the stack before starting a new branch. Each time the turtle encounters a "]", it pops a snapshot from the stack to be used in the following iteration to teleport to*

Two-Dimensional Branching Tree Variations

In this section, we'll experiment with several ways of generating 2D tree variations. Starting with a basic branching tree symmetrical about its trunk, we'll discuss how to modify the grammar to create longer branches overall, left or right leaning branches, more branches developing on one side, and clustering branches resembling a dandelion bulb.

To isolate the effects resulting from modifying the grammar, we'll keep the following parameters constant:

- *iterations*: 7
- *theta*: 22.5 degrees
- step_dist: 1

Base Tree

We'll start with a basic 2D tree with two rewriting rules that produce mirror symmetry about its trunk:

- *axiom*: X
- rewriting_rules: {"X": "F[+X][-X]FX", "F": "FF"}

Let's look at the first rewriting rule more closely. The symbol "X" is not interpreted by the turtle; therefore, it only serves as a placeholder for substitution during string rewriting. At a high level, each substring enclosed by a pair of square brackets ("[" and "]") is one "branch." So "[+X][-X]" gives you one branch to the left and one branch to the right stemming from the same spot on the trunk. The portion of the rule before and after the "branches" lengthens the trunk. The X at the end repeats the same structure all over again. Therefore, overall, the rule can be read as: grow the trunk, branch to the left, branch to the right, and grow the trunk more, repeat.

Since F is one forward segment, the second rule repeatedly replaces each forward segment with one that is twice its length, hence growing the structure by a power of 2. The tree produced by this rule set is shown in Figure 11-14.

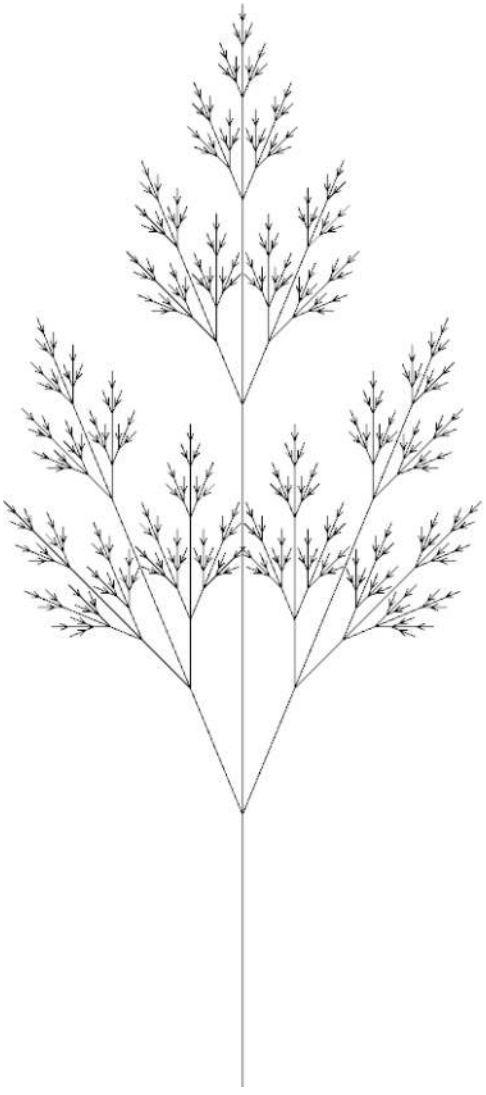


Figure 11-14. *Basic 2D tree with mirror symmetry about its trunk and branches lengthening at the rate of the power of 2*

Variation #1: Longer Branches

The grammar for the first variation is almost identical to the base tree, except the successor of the second rewriting rule is now "FFF" instead of "FF," which has the effect of lengthening the branches by a power of 3 instead of 2. As a result, the variation has

the same branching pattern as the base tree but with its trunk and branches "stretched" longer, as shown in Figure 11-15:

- *axiom*: X
- rewriting_rules: {"X": "F[+X][-X]FX", **"F": "FFF"**}

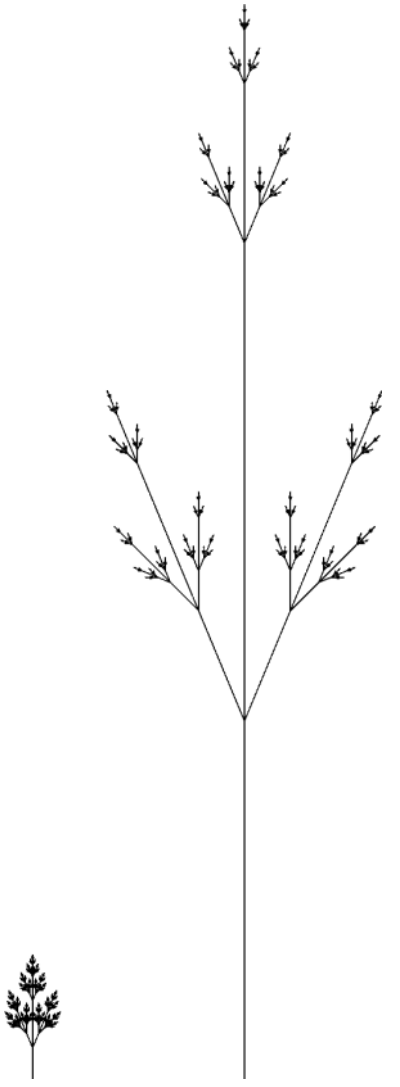


Figure 11-15. *Using "F": "FFF" (right) instead of "F": "FF" (left) for the second rewriting rule lengthens the branches by a power of 3 instead of 2*

Variation #2: Left- or Right-Leaning Branches

For the second variation, we'll modify the first rewriting rule by replacing [+X] with [+X+X]. Since everything inside a pair of square brackets is "one branch"—in other words, the turtle does not backtrack to the trunk until it hits the "]" at the end; it grows a left-leaning branch at the *tip* of a left-leaning branch, creating a willow-like drape toward the left, as shown in Figure 11-16:

- *axiom*: X
- rewriting_rules: {"X": "F**[+X+X]**[-X]FX", "F": "FF"}

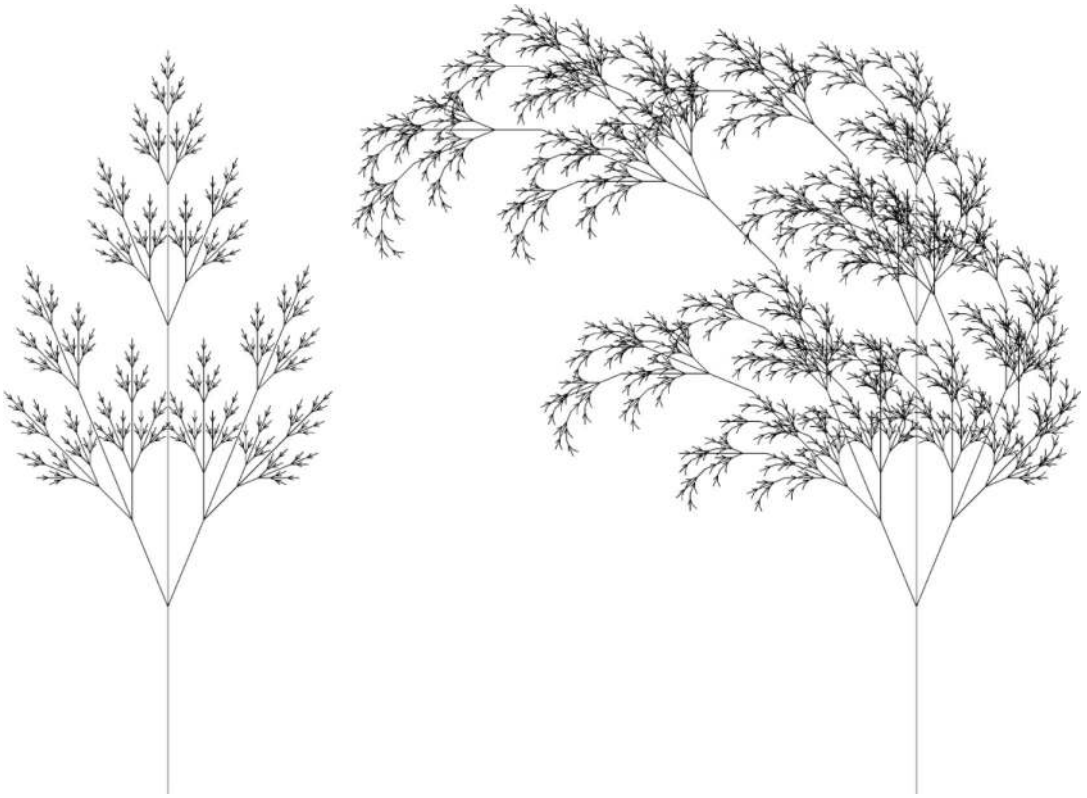


Figure 11-16. *Left shows the base tree, and the right shows the effect of replacing the [+X] in the first rewriting rule with [+X+X], which grows a left-leaning branch at the tip of a left-leaning branch, producing a willow-like drape*

Variation #3: More Branching on One Side

For the third variation, we'll insert a **[--X]** in the first rewriting rule after the original pair of branches, thus creating a third branch. Since theta = 22.5 degrees, each – means turning right 22.5 degrees. With the sequence [-X][--X], we're saying: "Branch to the right at 22.5 degrees, backtrack to the trunk, and branch to the right at 22.5 + 22.5 = 45 degrees." Therefore, we're adding *twice as many* right-leaning branches at the *same spot* on the trunk, creating more densely packed foliage *without* the drape, as shown in Figure 11-17:

- *axiom*: X
- rewriting_rules: {"X": "F[+X][-X]**[--X]**FX", "F": "FF"}

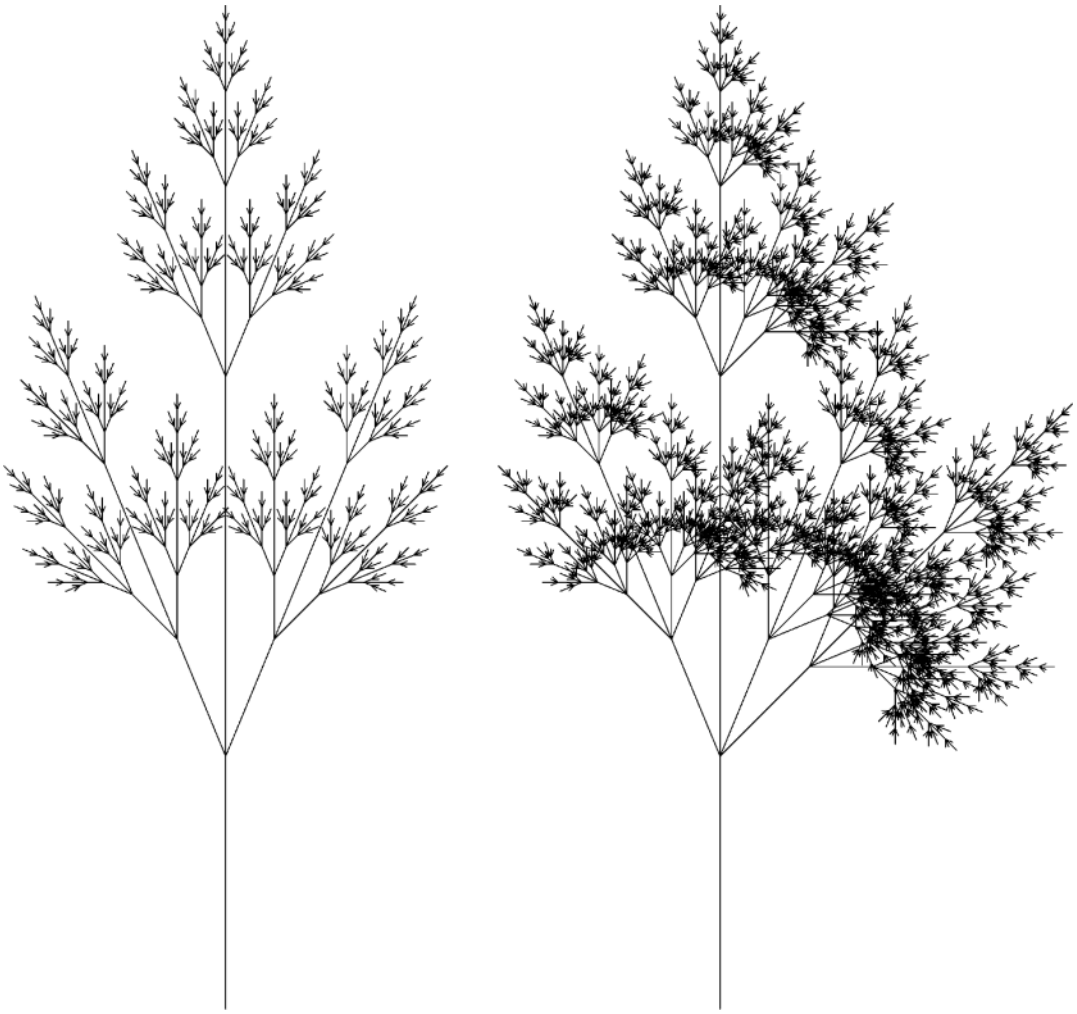


Figure 11-17. *Left: base tree. Right: using [+X][-X][--X] instead of [+X][-X] in the first rewriting rule, thus adding a third right-leaning branch stemming out of the same spot as the first two branches, creates a more densely packed foliage without the drape*

Variation #4: Bulb-Like (Dandelion) Trees

For the final variation, we'll derive from the third variation by enclosing *everything* in the successor of the first rewriting rule before the final "FX" in a pair of square brackets. This means each time we create a group of branches, we backtrack to the base of those

branches—which creates the clustering effect shown in Figure 11-18. If you read the modified rule out aloud, it goes like "Forward, branch left, branch right, branch more right, ***back to the beginning***. Forward, and repeat *everything*."

- *axiom*: X
- rewriting_rules: {"X": "**[F[+X][-X][--X]]**FX", "F": "FF"}

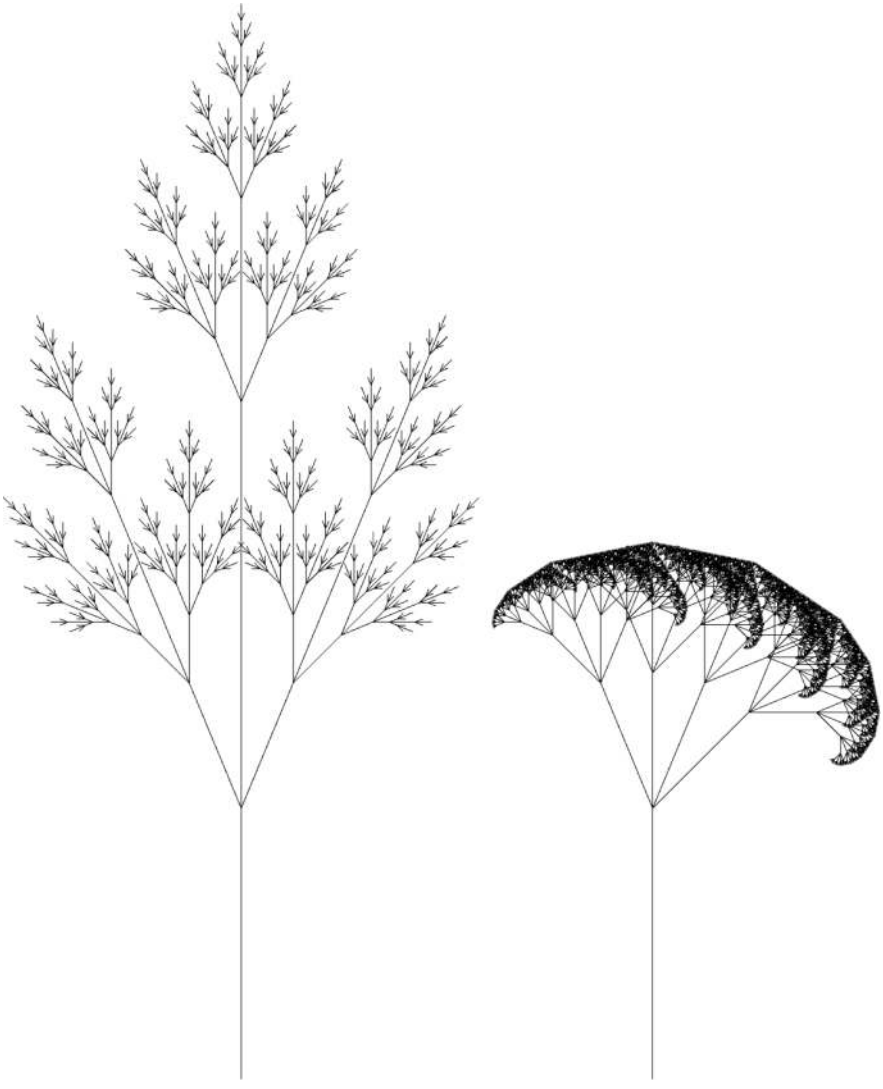


Figure 11-18. *Left: base tree. Right: Enclosing the entire branching portion of the first rewriting rule in a pair of square brackets makes all branches stem from the same spot, thus producing a "dandelion bulb" like clustering effect*

Note You can find the node graphs for all the 2D tree variations from this section in the file /Ch11/Figure11-14_to_11-18_2D_tree_variations.gh.

Taking the Turtle to 3D

In this section, we'll take our turtle class from 2D to 3D. All it takes is adding some code in the draw method to interpret symbols for pitching, rolling, and turning around, which are just rotations around different axes.

Listing 11-6 shows the necessary code changes. The symbols "&" and "^" mean to pitch down and up, respectively. The symbols "\" and "/" mean to roll left and right (note that the backslash must be "escaped" by prepending it with another backslash in the Python code, making it "\\"). The symbol "|" means to turn around (or turn 180 degrees).

Recall that we use +Y of the turtle's local coordinate system as its forward heading; therefore, turning left or right is rotating around its local Z axis, or (self.lcs.M20, self.lcs.M21, self.lcs.M22).

Pitching down ("&") or up ("^") thus corresponds to rotation around the local X axis, or (self.lcs.M00, self.lcs.M01, self.lcs.M02), with "down" as theta and "up" as negative theta.

Rolling left ("\\") or right ("/") corresponds to rotation around the local Y axis, or (self.lcs.M10, self.lcs.M11, self.lcs.M12), with "left" as negative theta and "right" as theta.

Turning around ("|") is similar to turning left ("+") or right ("-"), except we turn left or right 180 degrees instead of theta. If we are turning around, it does not matter if we make a left or right since we'll end up facing the same way. Therefore, we'll go with left or turn –180 degrees around the turtle's local Z axis.

Listing 11-6. Modifying the turtle class (renamed to turtle_3D) to handle pitching, rolling, and turning around in 3D (see /Ch11/Figure11-19_11-20_turtle3D.gh)

```
import Rhino.Geometry as rg
import ghpythonlib.components as ghc
from math import radians
```

```
class turtle_3D:
    #### SNIP ####

    def draw(self):
        for c in self.rewritten_str:
            #### SNIP ####
            if c == "[":
                #### SNIP ####
            elif c == "&":  # Pitch down
                mat_rot = rg.Transform.Rotation( ↩
                    radians(self.theta), ↩
                    rg.Vector3d(rg.Point3d( ↩
                    self.lcs.M00, self.lcs.M01, ↩
                    self.lcs.M02)), start_pt)
                self.update_heading(mat_rot)
            elif c == "^":  # Pitch up
                mat_rot = rg.Transform.Rotation( ↩
                    radians(-self.theta), ↩
                    rg.Vector3d(rg.Point3d( ↩
                    self.lcs.M00, self.lcs.M01, ↩
                    self.lcs.M02)), start_pt)
                self.update_heading(mat_rot)
            elif c == "\\":  # Roll left
                mat_rot = rg.Transform.Rotation(
                    radians(-self.theta),
                    rg.Vector3d(rg.Point3d(
                    self.lcs.M10, self.lcs.M11,
                    self.lcs.M12)), start_pt)
                self.update_heading(mat_rot)
            elif c == "/":  # Roll right
                mat_rot = rg.Transform.Rotation(
                    radians(self.theta),
                    rg.Vector3d(rg.Point3d(
                    self.lcs.M10, self.lcs.M11,
                    self.lcs.M12)), start_pt)
                self.update_heading(mat_rot)
```

```
            elif c == "|":  # Turn around
                mat_rot = rg.Transform.Rotation(
                    radians(-180), rg.Vector3d(rg.Point3d(
                    self.lcs.M20, self.lcs.M21,
                    self.lcs.M22)), start_pt)
                self.update_heading(mat_rot)

    #### SNIP ####

#### SNIP ####
```

Exploring 3D Tree Variations

In this section, we'll look at how to create a few variations of the 3D tree.

Basic Tree with Random-ish Branching

Here is an example of a basic 3D tree with somewhat random branching. To create a 3D structure with branches going in all directions, one must not only turn but pitch and roll relative to existing branches. As a result, the grammar becomes very complex very quickly:

- *axiom*: F
- rewriting_rules: {"F": "F[-&\\F][\\++&F][/--^F]||F[--&/F][++^\\F][+&F]"}
- *iterations*: 3
- *theta*: 22.5 degrees
- step_dist: 1

Figure 11-19 shows a node tree set up with the above grammar and the 3D turtle class from Listing 11-6 (b). Since we arbitrarily picked global +Y as the turtle's initial heading, the tree grows toward +Y. Therefore, we'll use Rotate 3D (c) with Unit X as the rotational axis (X input) and the default A = 0.5 Pi as the angle to rotate the tree so it sits upright on the XY plane with global +Z as the up axis. We link Rotate 3D to MultiPipe (d) next with radius = 0.1 (NodeSize) and flat end caps (Caps = 1) to create the tree shown in Figure 11-20.

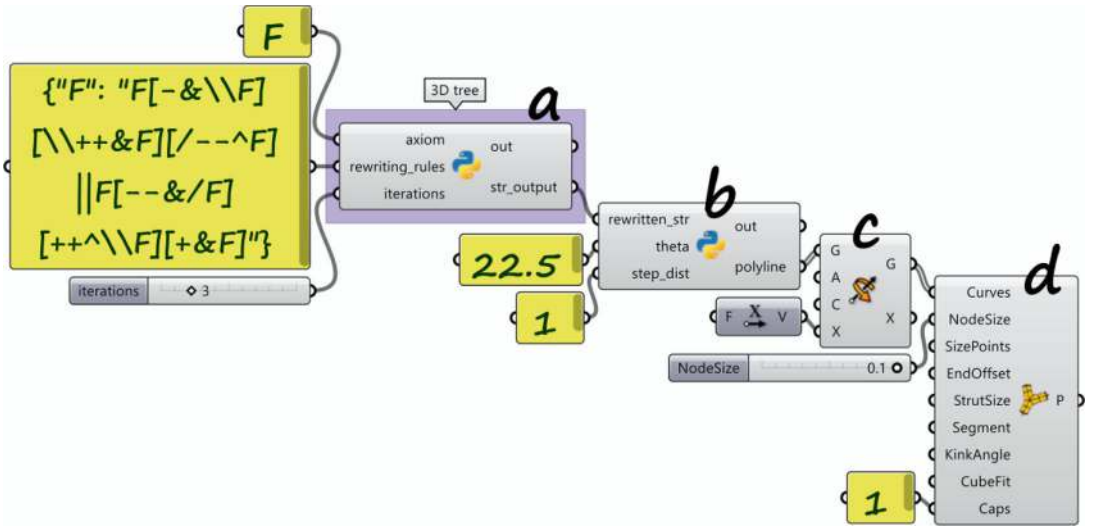


Figure 11-19. *Node graph with the turtle_3D class (Listing 11-6) and the basic 3D tree grammar as input, which produces the tree shown in Figure 11-20. The first Python 3 Script node (a) rewrites the grammar for three iterations (Listings 11-1 and 11-2), which is interpreted by the turtle_3D class (b) with a turn angle of 22.5 degrees and step distance of one. Since the turtle has an initial heading of +Y, Rotate (c) is used to turn the tree 90 degrees around X so it sits upright on the XY plane. Finally, MultiPipe (d) with a NodeSize of 0.1 and flat end caps (Caps = 1) is used to give the tree branches volume*

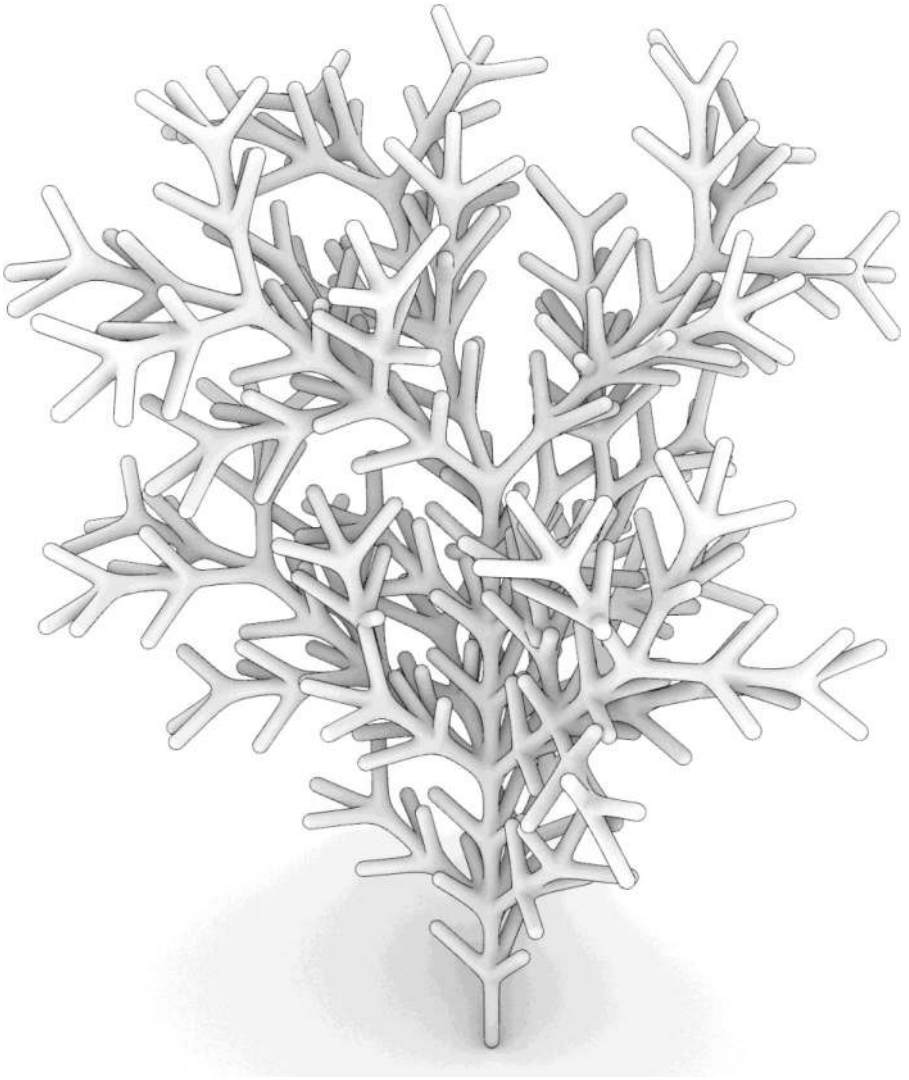


Figure 11-20. *The basic 3D tree produced by the node graph shown in Figure 11-19*

Trees with Fourfold Symmetry

Random branching resembles trees in nature and produces a realistic look. However, sometimes we want symmetry on purpose to create a geometric look. We can do so by repeating a substring encoding for a branching pattern back-to-back in the same

rewriting rule, as the example shown below. The substring "[&&FX]" makes one branch. Repeating it four times in a single rule with the "rolling right" maneuver ("////") in between produces the pattern with fourfold symmetry shown at the top of Figure 11-21 (recall that fourfold symmetry means you get the original pattern back with each rotation of 360/4 = 90 degrees).

- *axiom*: FX
- rewriting_rules: {"X": "[&&FX]////[&&FX]////[&&FX]////[&&FX]", "F": "FF"}
- *iterations*: 3
- *theta*: 22.5 degrees
- step_dist: 3

You can use the trick of repeating a "branch" substring to create different types of symmetry. For example, a repetition of three creates threefold symmetry (360/3 = 120 degrees). You will need to experiment with different `theta` and `step_dist` values to find ones compatible with the number of branch repetitions. For example, in the sample grammar above, we repeat a branch four times, with "////" (rolling 22.5*4 = 90 degrees) in between each adjacent pair of branches; this works because 90*4 = 360, so for each iteration, the branches fill up one "revolution." The value of `step_dist` needs to be small enough such that the branches do not overlap each other. I found a `step_dist` = 3 works with this grammar through experimentation.

Creating Variations by Adjusting MultiPipe Parameters

By adjusting NodeSize and StrutSize values for the MultiPipe node used to give the tree polylines volume, we can create some intriguing variations of the fourfold symmetric tree as shown in Figure 11-21. The variations are numbered 1 through 5 with their corresponding parameters summarized in Table 11-4. You can find the complete Grasshopper definition containing all five variations in the file `/Ch11/Figure11-21_3D_tree_variations.gh` in the downloaded source.

Recall that NodeSize sets the pipe radius, whereas StrutSize is a multiplier of NodeSize for controlling the strut between pipes. It turns out by varying only these two parameters while holding all others constant (with Caps = 1 to make flat end caps for the pipes and default values for the rest), we can create some very interesting and diverse forms.

Using a negative NodeSize (#1 in Figure 11-21) makes for hollow pipes, whereas a StrutSize greater than 1 (#3 in Figure 11-21) creates "inverted" struts that resemble a twisted balloon sculpture. If you try setting StrutSize to a negative value that is smaller than –1.0 (#5 in Figure 11-21), the pipes will also look like twisted balloons but with the tips becoming flower-like structures with "cups" overlapping one another.

Table 11-4. *Parameters for MultiPipe used to produce the five variations of the fourfold symmetric tree shown in Figure 11-21 (see* `/Ch11/Figure11-21_3D_tree_variations.gh`*)*

Variation # (in Figure11-21)	NodeSize	StrutSize	Caps
1	-1.00	0.7	1
2	0.50	0.5	1
3	0.50	3.0	1
4	0.50	0.05	1
5	0.50	-5.0	1

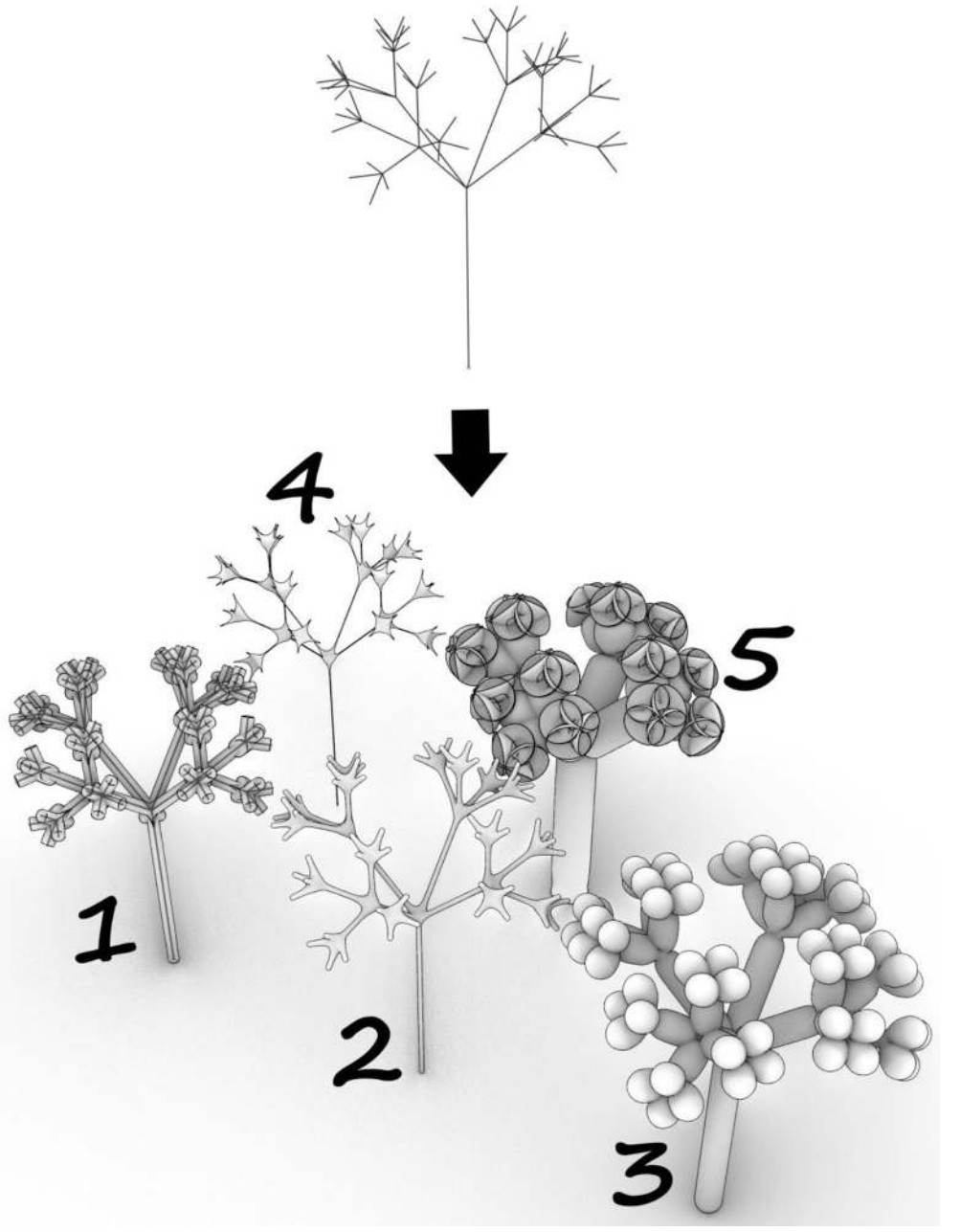


Figure 11-21. *We can create some very interesting variations of the fourfold symmetric tree simply by varying the NodeSize and StrutSize inputs for MultiPipe (recall that NodeSize is pipe radius and StrutSize is a multiplier of NodeSize that controls struts between pipes). Using a negative NodeSize gives #1 hollow pipes. A StrutSize > 1.0 makes #3 look like a twisted balloon sculpture. A StrutSize < -1.0 also makes bulging pipes, but makes the unconnected pipe tips look like cups and overlap (#5)*

Generating a Canopy for the Fourfold Symmetric 3D Tree

In this section, we'll generate a canopy for the fourfold symmetric 3D tree that resembles an abstracted version of foliage. Figure 11-22 shows the complete node graph. Nodes a through e are a copy of Figure 11-19, which generates the base fourfold 3D tree (shown at the top of Figure 11-21). Nodes f through o create the canopy by finding the points of the tree above the trunk, deriving the 3D convex hull and ShrinkWrap mesh from those points and taking the Mesh Difference between the two. We'll look at each step in detail next.

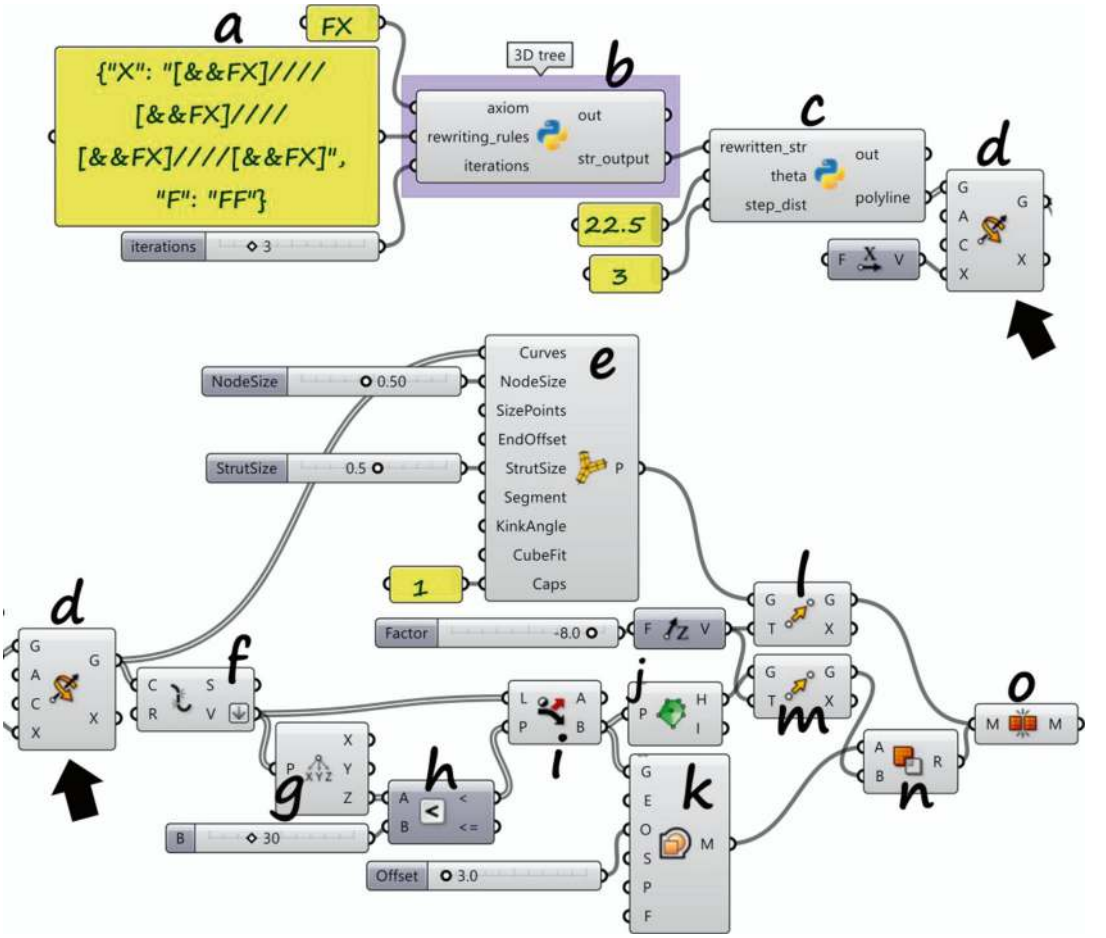


Figure 11-22. *Node graph for generating a canopy for the fourfold symmetric tree. Nodes a through e are reproduced from Figure 11-19 and generate the base fourfold tree (shown at the top of Figure 11-21), while nodes f through o generate the canopy by taking the difference between the ShrinkWrap mesh and the 3D convex hull of the tree branches*

Note Where to find the nodes in Figure 11-22: Params ➤ Input ➤ Panel (a), Maths ➤ Script ➤ Python 3 Script (b, c), Transform ➤ Euclidean ➤ Rotate 3D (d), Surface ➤ SubD ➤ MultiPipe (e), Curve ➤ Util ➤ Explode (f), Vector ➤ Point ➤ Deconstruct (g), Maths ➤ Operators ➤ Smaller Than (h), Sets ➤ List ➤ Dispatch (i), Mesh ➤ Triangulation ➤ Convex Hull 3D (j), Mesh ➤ Triangulation ➤ ShrinkWrap (k), Transform ➤ Euclidean ➤ Move (l, m), Intersect ➤ Shape ➤ Mesh Difference (n), and Mesh ➤ Util ➤ Mesh Join (o).

Step 1: Isolating the Points Defining the Branches (Minus the Trunk)

Our first step is to isolate the points above the trunk so they can be used as the basis for deriving the convex hull. Figure 11-23 shows the step visually.

Recall that Rotate 3D (d) outputs the tree curves that have been rotated to stand upright on the XY plane (1 in Figure 11-23). Rotate 3D (d) is linked to Explode (f) to extract the points constituting these polylines (2 in Figure 11-23).

Dispatch (i) is used next to divide the points into two groups base on their Z values, such that those with Z < 30 go in the first group (A) and the rest (with Z >= 30) go in the second group (B). Here is how to set up the node sequence: we link Explode (f) to Deconstruct (g) to separate the X, Y, and Z coordinates of the points and link Z to the A input of Smaller Than (h) with B set to 30 to obtain a list of True or False values indicating which Zs are less than 30 (via <). This list is in turn used for the P input of Dispatch (i) as the pattern for separating the points (f).

In Figure 11-23, 3 and 4 show the A and B outputs of Dispatch (i), respectively. Moreover, 3 is the group of points with Z less than 30 and 4 the group with Z ≥ 30. I found the value 30 as the threshold for separating the points experimentally.

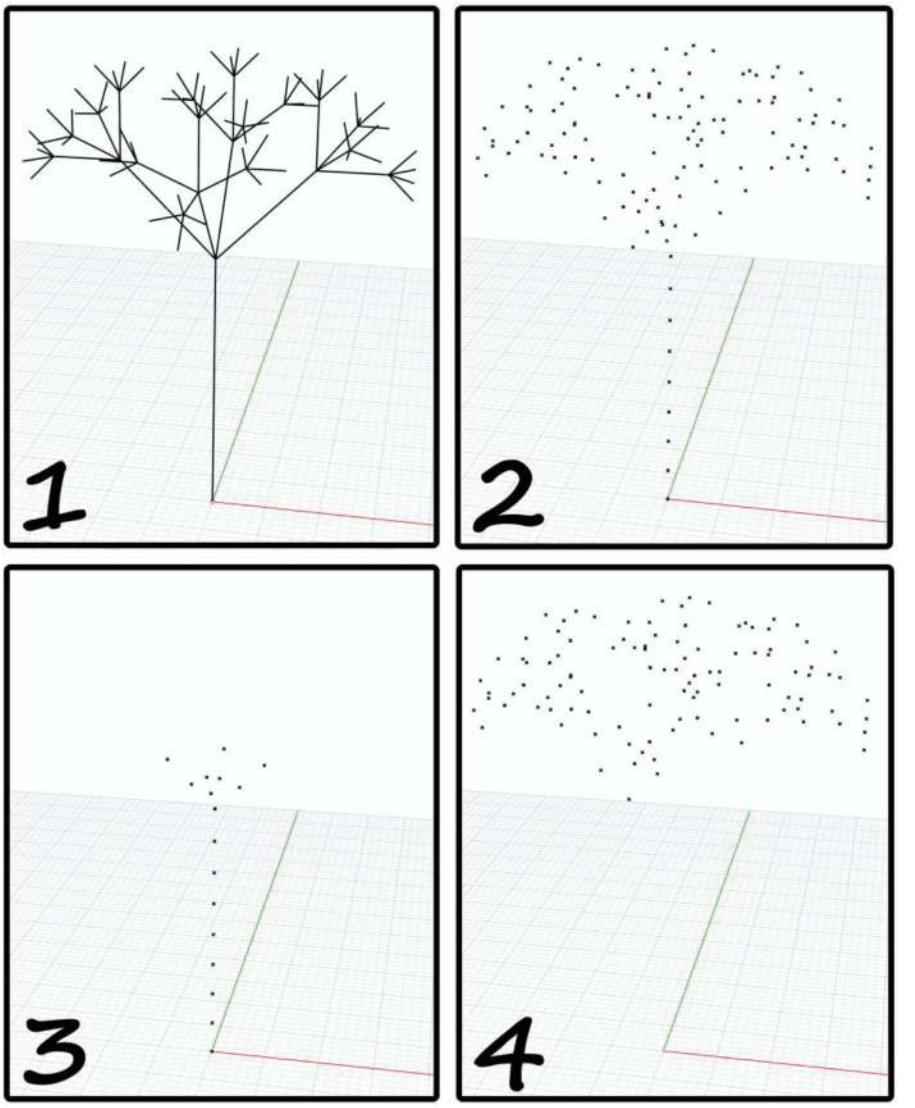


Figure 11-23. *Separating the points in the fourfold tree into two groups, such that one group contains points with Z < 30 and the other Z ≥ 30. (1) is the output of Rotate 3D (d in Figure 11-22), which are the tree polylines rotated to stand upright on the XY/ plane. (2) shows the points constituting the tree polylines (Explode, f in Figure 11-22). (3) shows the points with Z < 30 (A output of Dispatch, i in Figure 11-22), and (4) the points with Z >= 30 (B of Dispatch)*

Step 2: Finding the Convex Hull of the Branches

Now that we've isolated the points making up the branches of the tree, the next step is to find their convex hull—the minimal convex polygon fully enclosing them. We do so by linking the B output of Dispatch (i), which are the points with $Z \geq 30$, to Mesh ➤ Triangulation ➤ Convex Hull 3D (j). Figure 11-24 shows the convex hull with (right) and without (left) the tree polylines for reference.

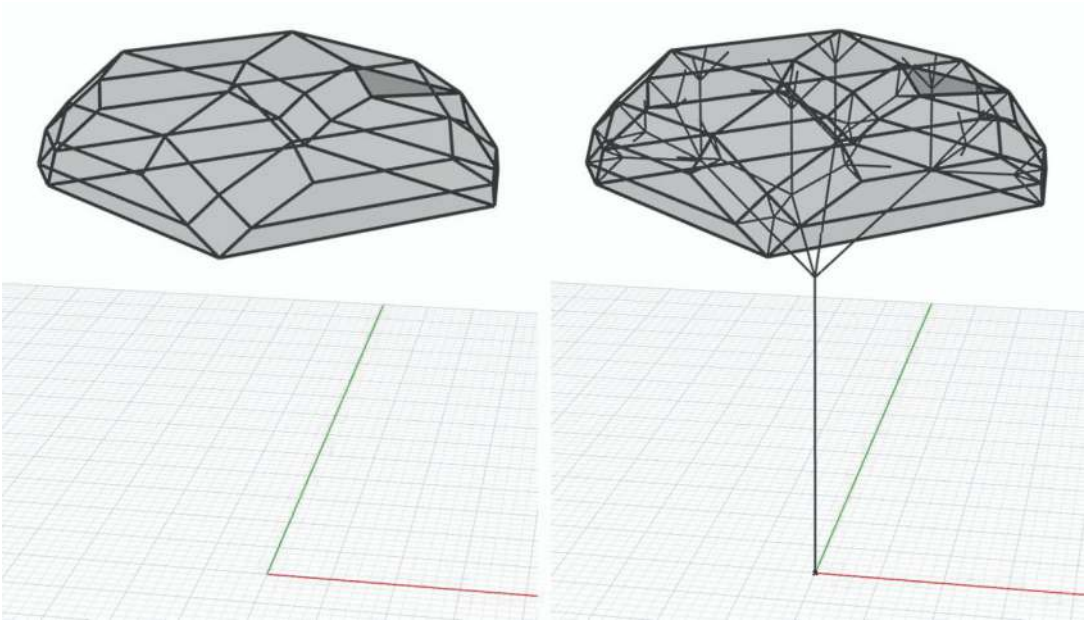


Figure 11-24. *Left: Output of Convex Hull 3D (j in Figure 11-22), which is the minimal convex 3D polygon that encloses all the tree branch points. The right shows the convex hull composed with the tree polylines for reference*

Step 3: Using "Shrink Wrap" to Create a Shell Hugging the Branches

The convex hull alone could make a canopy; however, we can create something more organic-looking by using Mesh ➤ Triangulation ➤ ShrinkWarp (k) with an Offset of 3.0, which produces the curved shell shown in Figure 11-25 that resembles an abstracted foliage.

ShrinkWrap creates a mesh fully enclosing a given piece of geometry with an optional offset. Unlike Convex Hull 3D, ShrinkWrap does not necessarily output a convex mesh. The larger the positive Offset, the further ShrinkWrap (k) grows the mesh outward. The right side of Figure 11-25 shows the output of ShrinkWrap (k) composited with the tree curves for reference.

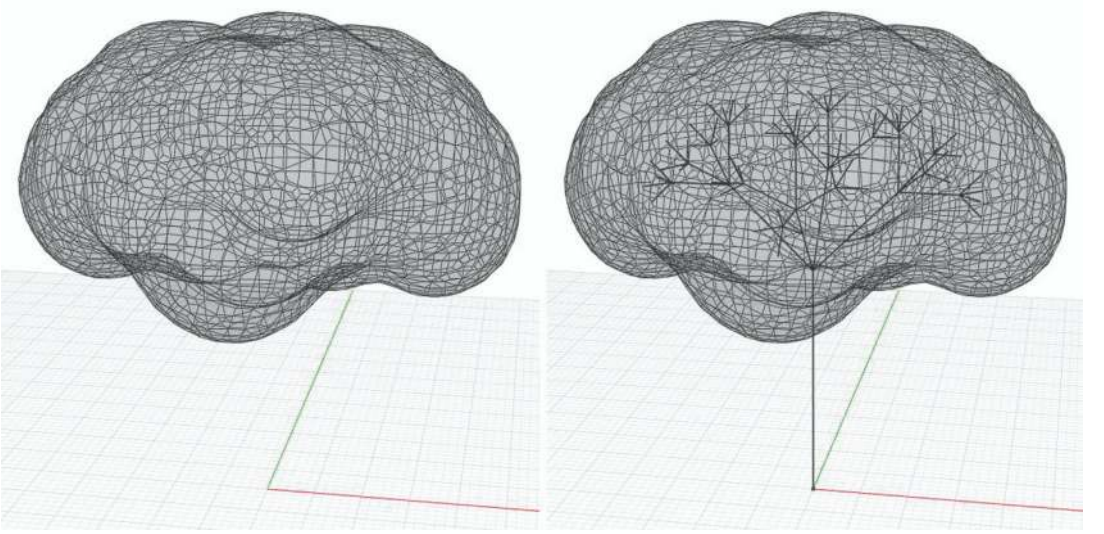


Figure 11-25. *The output of ShrinkWrap (k) with Offset = 3.0, which creates a rounded shell around the tree branch points, resembling an abstracted foliage. The right shows the same mesh composed with the tree polylines as reference*

Step 4: Using "Mesh Difference" to Carve the Convex Hull from the Shrink-Wrapped Shell to Create the Canopy

The next step is to carve a hole at the bottom of the ShrinkWrap (k) mesh just large enough to fit the MultiPiped (e) tree branches—this is where the convex hull comes in. Recall that Convex Hull 3D (j) gives us the minimal polygons fully enclosing the points making up the tree branches. Therefore, we can use Convex Hull 3D (j) as the template for cutting the hole from the ShrinkWrap (k) mesh.

There is a slight problem, however; as shown in Figure 11-26, both the ShrinkWrap (k) mesh and the convex hull center on the point cloud. As a result, we must use a Move node (m) to offset the output of Convex Hull 3D (j) downward so it sits at

the bottom of the ShrinkWrap (k) mesh. After that, we can simply use Mesh Difference (n) to perform the carving, by linking ShrinkWrap (k) to its A input and Move (m) to the B input.

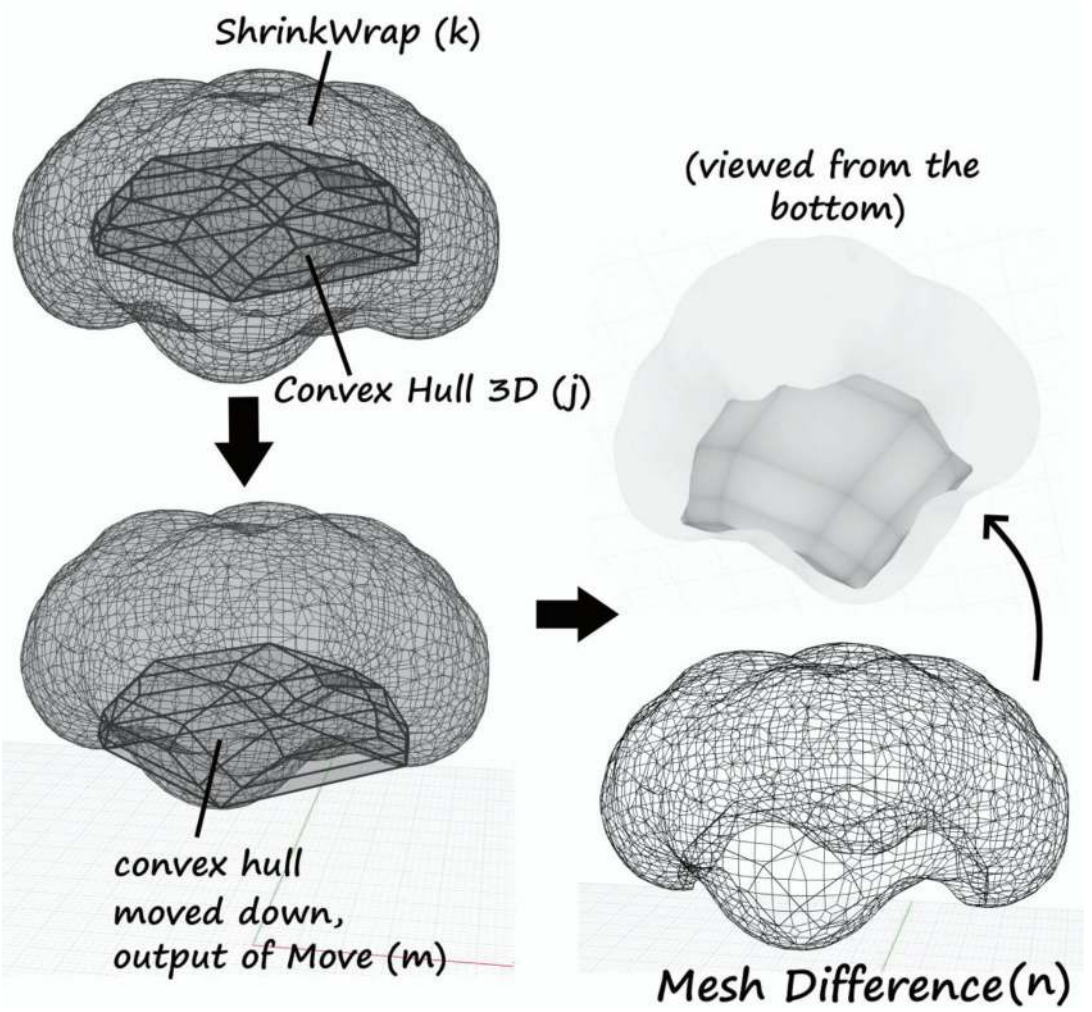


Figure 11-26. *To create the canopy, we offset the output of Convex Hull 3D (j) downward using a Move (m) node so it sits at the bottom of the ShrinkWrap (k) mesh and perform a Mesh Difference (n) between the two*

Step 5: Combining the Tree with the Canopy

With the canopy formed, the final step is to combine the tree and the canopy into a single object. During the previous step, we moved the convex hull downward (m) to align with the bottom of the ShrinkWrap (k) mesh. As a result, we must use a second Move node (l) to offset the output of MultiPipe (e) the same amount as the convex hull to make the top of the branches line up with the interior of the canopy.

Once the piped tree is in place, the final step is to Shift-drag the tree (l) and the canopy (n) to a Mesh Join node (o) to combine. The complete model is shown in Figure 11-27.

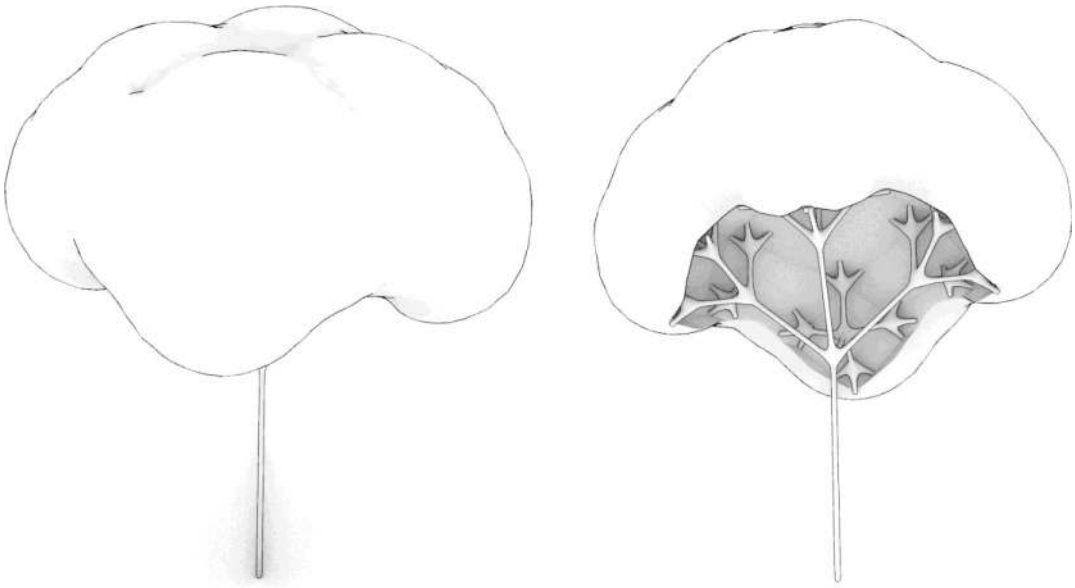


Figure 11-27. *The canopy combined with the MultiPiped tree into a single model*

Summary

In this chapter, you learned all about L-System grammars—their structure, how to use them for encoding various types of objects, and how to generate curves and meshes from them.

An L-System grammar consists of an axiom, which is a starting string, and a set of one or more rewriting rules that tell how to substitute characters in the string at hand. In each iteration, all occurrences of a character matching the left-hand side (predecessor) of a rule are replaced at once by the rule's right-hand side (successor)—a process called *simultaneous* rewriting. We practiced rewriting strings with pen and paper and then implemented the process as Python code.

The second part of the chapter took you through implementing a turtle graphics style Python class that generates 2D polylines from strings rewritten using L-System grammars. You started with the simplest turtle, which creates continuous-line drawings by either laying down forward strokes or turning in-place in fixed angles. You then gradually upgraded the turtle to give it the ability to skip (move without laying down a stroke) and to backtrack to a previous spot by teleporting, which allows it to create tree branches.

In the third part of the chapter, you experimented by modifying grammars in different ways to influence their effects on the generated 2D trees. Some of the tricks you learned include using a longer sequence of the same letter in the successor of the rule (e.g., "FFF" instead of "FF") to create more elongated branches. Since a pair of square brackets creates one branch, repeating a substring *within* a pair of brackets makes branches that stack tip-to-tip (e.g., "[+X+X] instead of [+X]"), while adding a substring *with brackets* (e.g., [-X][--X] instead of [-X]) makes multiple branches stem from the same spot, creating denser foliage. Finally, by nesting the entire branching sequence inside an outer pair of brackets, you make the turtle backtrack to the same spot each time a set of branches is made, therefore clustering all the branches at the same place and creating a dandelion-bulb-like cluster.

In the fourth part of the chapter, you took the turtle from 2D to 3D by giving it the ability to pitch, roll, and turn around. You then generated a basic 3D tree as well as a tree with fourfold symmetry and explored variations by changing the NodeSize (pipe radius) and StrutSize for MultiPipe.

In the last part of the chapter, you generated an abstract canopy for the fourfold tree by isolating the points constituting the branches at the top, creating a shell around them using ShrinkWrap, and using the points' 3D convex hull to carve a hole at the bottom of the shell just large enough to fit the tree branches.

References

Lupiani, I. (2025). Procedural Content Generation for Games: Automate 3D Asset and Environment Creation with Blender Python and Geometry Nodes. Apress.

Prusinkiewicz, P., Lindenmayer, A., Hanan, J. S., Fracchia F. D., Fowler, D., de Boer, M. J. M., Mercer, L. (1990). The Algorithmic Beauty of Plants. Springer-Verlag, New York.

CHAPTER 12

Generating City Skyline Models from Building Footprint (GIS) Data

Building footprint data is a type of GIS (Geographic Information System) data comprised of 2D outlines of building cross-sections along with relevant statistics like roof heights and ground elevations. In this chapter, we'll work through the process of sourcing footprint data, editing it in QGIS, analyzing it with Python 3, and generating models from it in Grasshopper.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper. Download the files and navigate to the /Ch12 folder, where you'll find *.gh files named after the figures in which they appear, along with a short description. For example, /Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh. The sample files have been tested in Rhino 9 on Windows.

As you work through the chapter, build your own node graphs by following the steps described in the text or start with the downloaded *.gh files and modify the parameters to experiment.

Some source code, examples, and figures in this chapter are derived from Chapter 8 of the book *Procedural Content Generation for Games* also by the author (Lupiani, 2025).

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_12

Installing Required Python Packages

The examples in this chapter require you to install the following third-party Python packages from the PyPl server, which can be done directly inside a Python 3 Script node:

1. pyshp v3.0.3
2. pandas v3.0.1
3. matplotlib v3.10.8
4. openpyxl v3.1.5

Figure 12-1 shows the step-by-step process. Place a Python 3 Script node (1) on the canvas, double-click the node to open its Script Editor (2), and click the "Install Package" button (3) at the top. In the window that appears (4), type in the name of the package you want to install without the version number (5). For example, `pyshp` instead of `pyshp v3.0.3`, so the search will include all available versions of that package. Click to select the latest version from the list (6). Check the "Add Package Reference to Script" box (7), and click the "Install" button to install the package.

Note Just like a GH plugin, each Python package only needs to be installed once per Rhino installation.

Enabling the "**Add Package Reference to Script**" option will trigger Grasshopper to automatically check for and install missing packages when the GH definition is opened—great for sharing the file with other users. For example, the option is enabled for all `*.gh` files for this chapter, so when you open them on your machine, GH automatically checks for and installs any missing packages from the list above. If you choose to follow the chapter by creating GH definitions from scratch, be sure to install each package by following the steps in Figure 12-1.

The observant reader may also notice that some of the Python 3 code in this chapter uses the package `numpy` even though it is not explicitly installed—this is due to the fact that some packages from the above list have `numpy` as a dependency, so installing those packages also triggers the installation of `numpy`.

We will *not* be calling functions from `openpyxl` directly; however, it is installed as a dependency for some of the other packages.

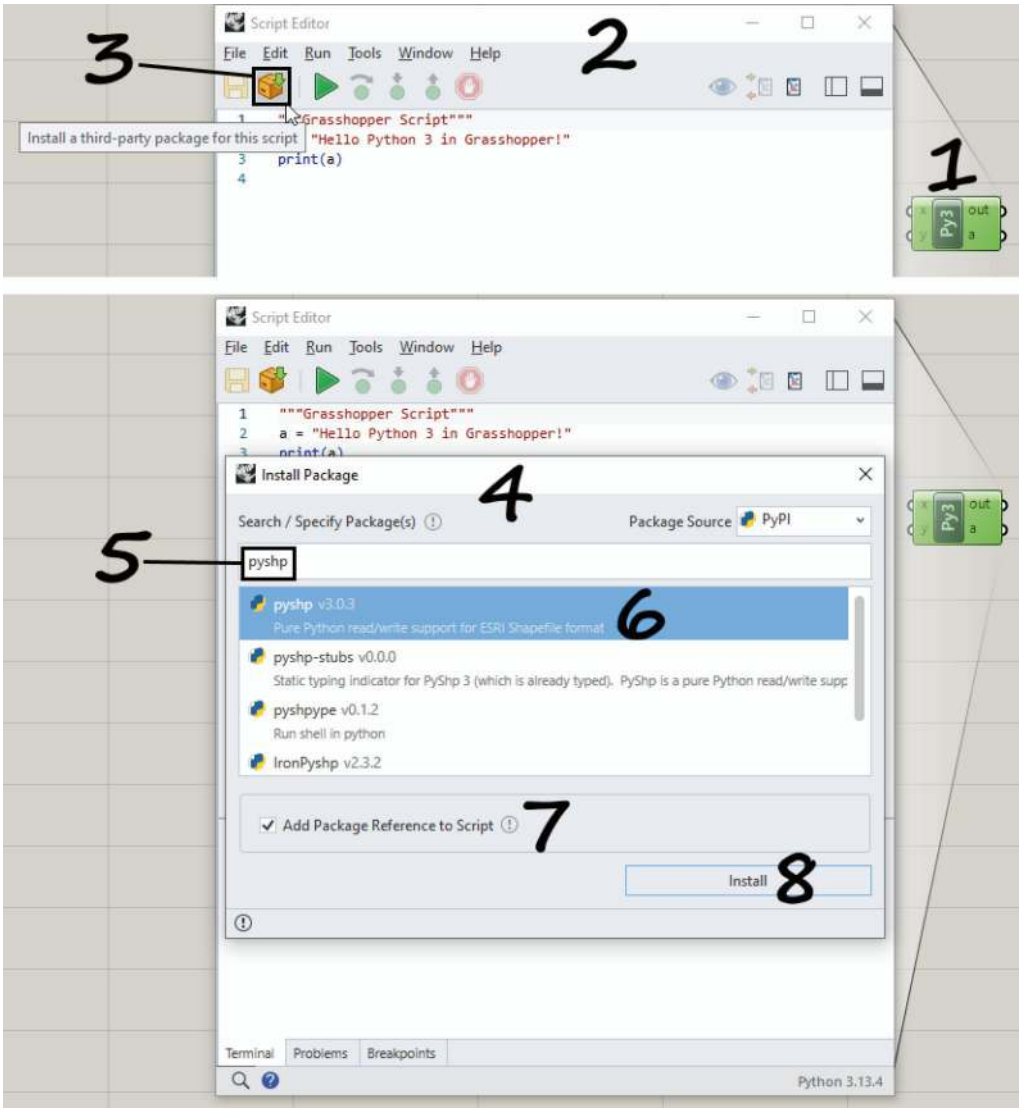


Figure 12-1. *Steps for installing a third-party Python package from the PyPl server. (1) Double-click a Python 3 Script node to open its (2) Script Editor. (3) Click the "Install Package" button at the top to open the (4) Install Package window. (5) Type in the name of the package (as shown, pyshp). (6) Click to select the latest version in the search results. (7) Check the "Add Package Reference to Script" box. (8) Click the "Install" button*

Installing QGIS

The provided sample building footprint data sets are already converted to the required formats. To edit your own data sets, please download the Long-Term Release of QGIS for your operating system from https://qgis.org/download/ and follow the installation steps. The screenshots in this chapter are based on QGIS 3.44.8 "Solothurn" on Windows 11.

Attribution to the Provided Data Sets

The provided building footprint data set of Battery Park, NYC, contains the following four files:

1. /Ch12/qgis_battery_park.shp
2. /Ch12/qgis_battery_park.xlsx
3. /Ch12/qgis_battery_park.dbf
4. /Ch12/qgis_battery_park.shx

The data is downloaded from NYC OpenData (https://data.cityofnewyork.us/City-Government/Building-Footprints-Map-/3g6p-4u5s) and edited by the author in QGIS.

The provided building footprints data set of Havana, Cuba, contains the following four files:

1. /Ch12/havana.shp
2. /Ch12/havana.xlsx
3. /Ch12/havana.dbf
4. /Ch12/havana.shx

The data is downloaded from Google Research Open Buildings (https://sites.research.google/gr/open-buildings/, CC By 4.0 https://creativecommons.org/licenses/by/4.0/deed.en) and edited by the author in QGIS.

What Is Building Footprint Data?

A building footprint data set is a type of GIS (Geographic Information System) data that describes the shapes and dimensions of buildings in a municipal area, like a city or county. When you download a data set, it is usually in the `*.csv` format, which must be post-processed in GIS software like QGIS to isolate the area of interest to be exported out as a *shapefile* (`*.shp`) with a corresponding *spreadsheet* (`*.xlsx`). The shapefile contains polylines that represent the outlines of the buildings' cross-sections, while the spreadsheet contains their roof heights, ground elevations, etc. During the export to shapefile, two additional files with the extensions `*.dbf` and `*.shx` are also generated. These four files (`*.shp`, `*.xlsx`, `*.dbf`, and `*.shx`) are what we'll need to generate building models in Grasshopper.

Sourcing Building Footprint Data Online

You can easily find building footprint data sets for free online, although their quality and formats vary. Below are two suggestions to get you started. Bear in mind that different sources offer data in different formats, such as `*.csv`, `*.geojson`, etc., which require different types of post-processing. In this chapter, we'll go over how to edit and convert s `*.csv`, files, which is one of the most common file types for building footprint sets.

- Local government websites for municipal areas like cities, counties, etc.
  - For example, the provided data set of the NYC Battery Park neighborhood is downloaded from NYC OpenData https://data.cityofnewyork.us/City-Government/Building-Footprints-Map-/3g6p-4u5s
- Google Research Open Buildings https://sites.research.google/gr/open-buildings/

Example: Downloading Building Footprint Data from NYC OpenData

Let's head over to NYC OpenData and download a building footprint data set as shown in Figure 12-2. Once at the site, click the "Export" button at the upper right. In the dialog that appears, select "Download file", click to expand the dropdown menu, and select

"CSV for Excel". Check the "All data" option and click the Download button. At the time of writing, this downloads a file named "BUILDING_20260324.csv" that is over 500 MB, as it contains data for a large area.

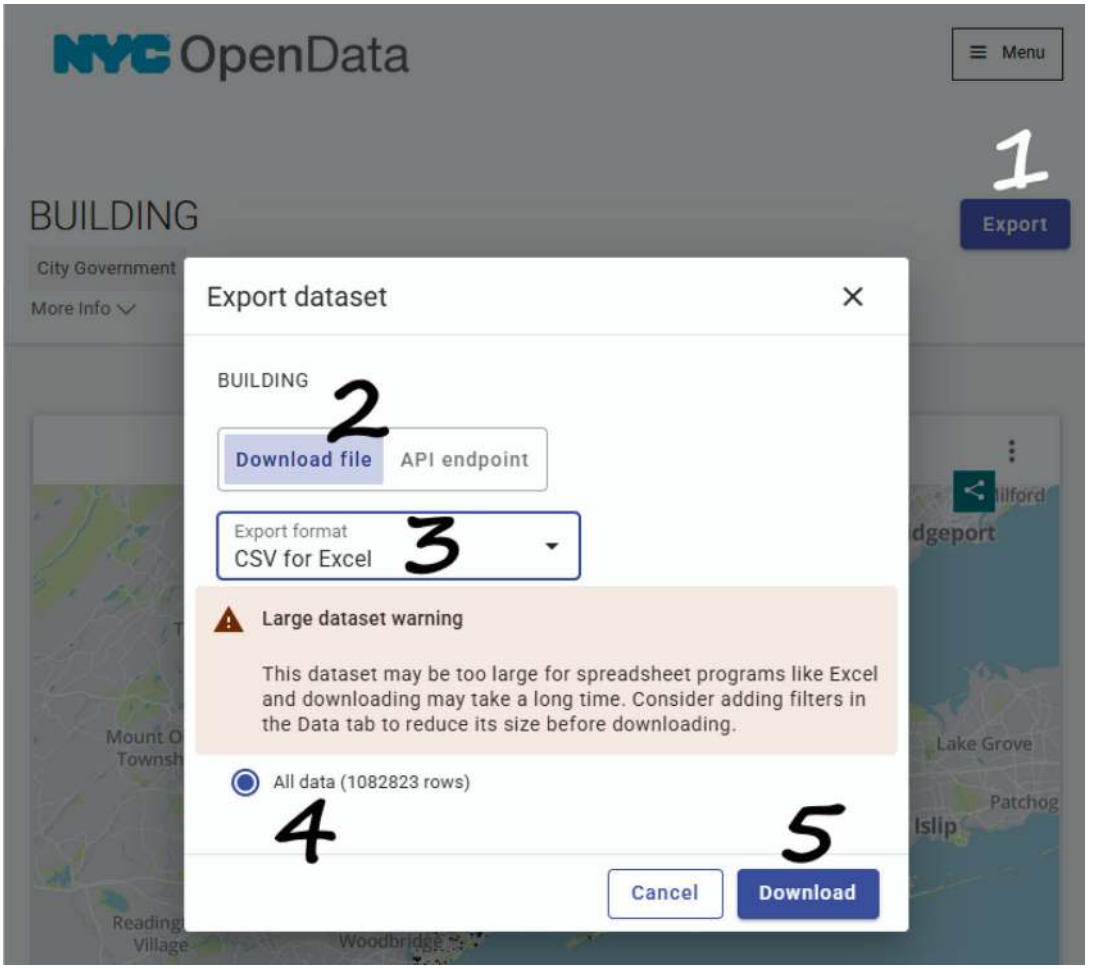


Figure 12-2. *Downloading building footprint data from NYC OpenData. (1) Click the "Export" button. In the dialog that appears, select (2) Download file and (3) Export format CSV for Excel and (4) check "All data". Click the (5) Download button to download*

Post-Processing Building Footprint Data in QGIS

With a data set downloaded, let's walk through how to isolate an area of interest in QGIS and export it.

Step 1: Importing the *.csv File in QGIS As a Delimited Text Layer

To get started, open QGIS and create a new project with Project → New (Ctrl-N). Follow the steps below to import the downloaded *.csv file as a Delimited Text layer as shown in Figure 12-3:

1. Click Layer ➤ Add Layer ➤ Add Delimited Text Layer....
2. Click the "..." button to the right of the File name field ((1) in Figure 12-3) to bring up the file open dialog.
3. Select the downloaded BUILDING_20260324.csv file to open it.
4. A message will appear in the Sample Data area within the Add Layer dialog, informing you the text scanning is in process. Wait a few seconds for it to finish.
5. Leave all auto-detected settings at default (File format will be CSV and Geometry Definition will be Well known text (WKT)).
6. Click the Add button to add the layer.

After clicking the Add button, it may take QGIS a few minutes to fully create the layer since the *.csv file covers a large area. Once the layer appears in the editor and the Add button is greyed out, you can close the dialog. The layer will look like Figure 12-4, which is initially very zoomed out with all the buildings appearing as dots. You can type in a new scale (1:5000) at the bottom of the editor, as shown in Figure 12-5, or scroll the mouse wheel to zoom in until building outlines begin to appear.

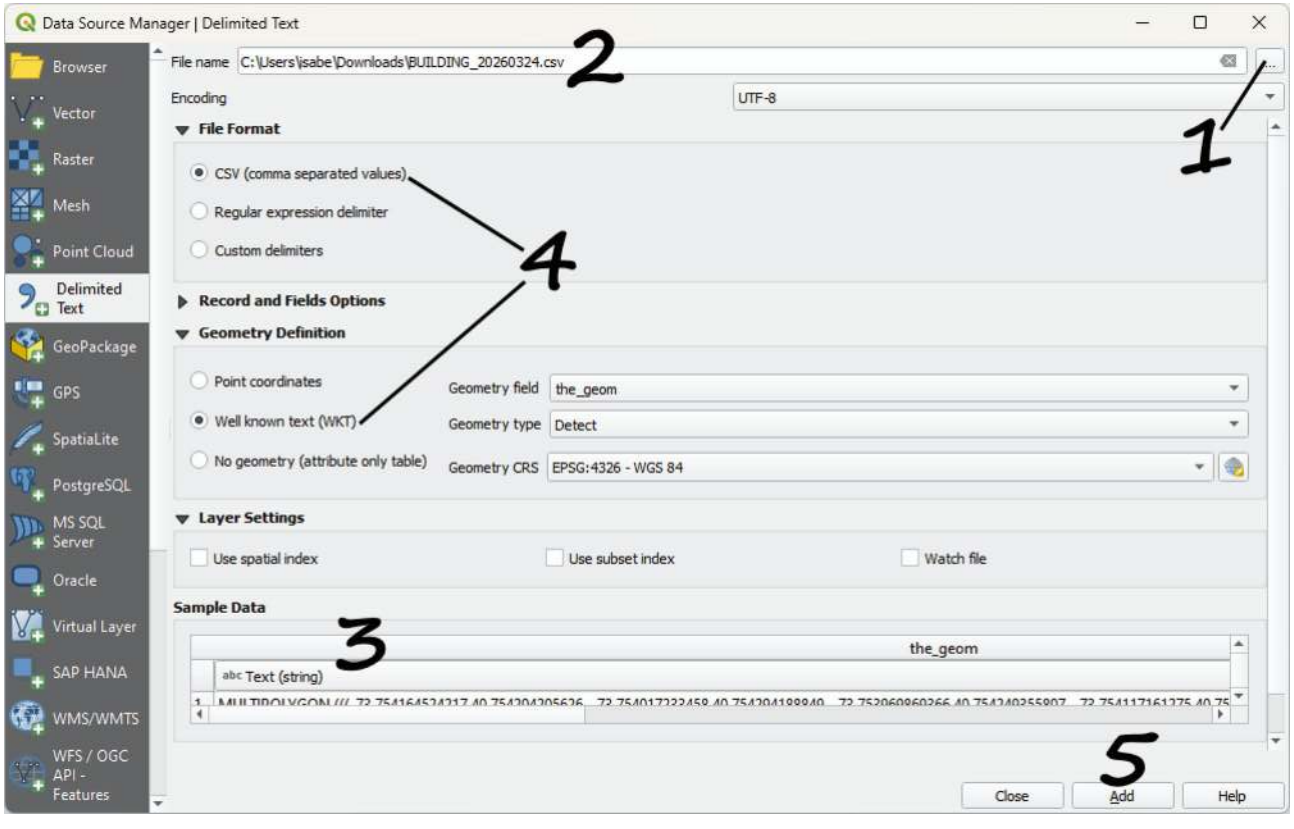


Figure 12-3. *Adding the *.csv file as a Delimited Text layer in QGIS. Click Layer ➤ Add Layer ➤ Add Delimited Text Layer... to invoke the dialog as shown. Click the (1) "..." button and (2) select the *.csv file. (3) A message will appear in the "Sample Data" area to let you know that text scanning is in progress. Once scanning is complete, (4) QGIS should detect that the File Format is CSV and auto select Well known text (WKT). (5) Click Add to create the layer*

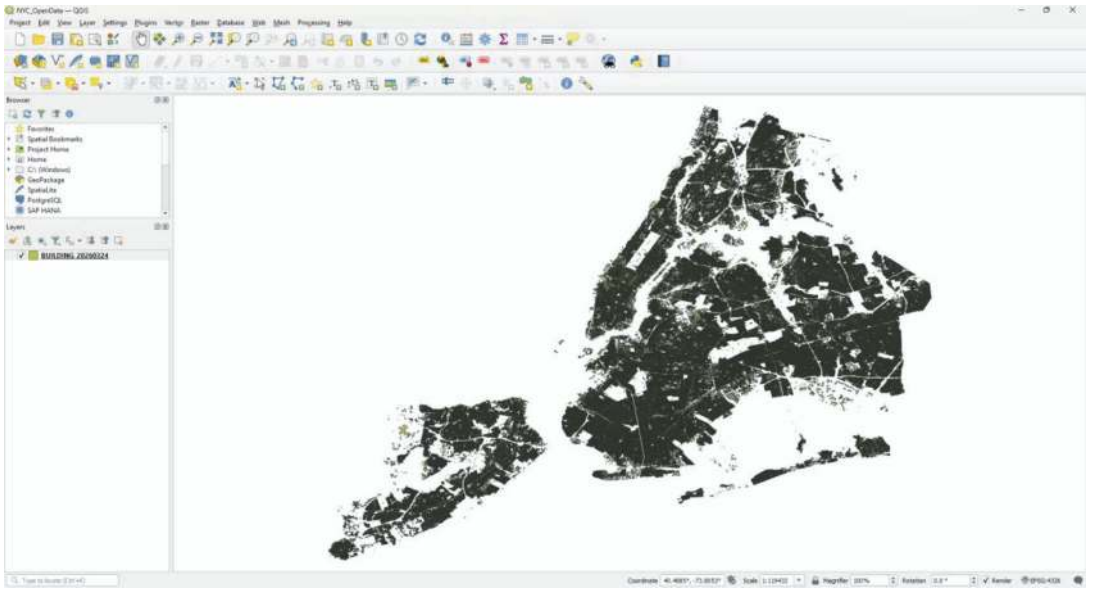


Figure 12-4. *The new layer created from the* `*.csv` *file covers a large area and is initially very zoomed out—so much so that building footprints appear as dots*

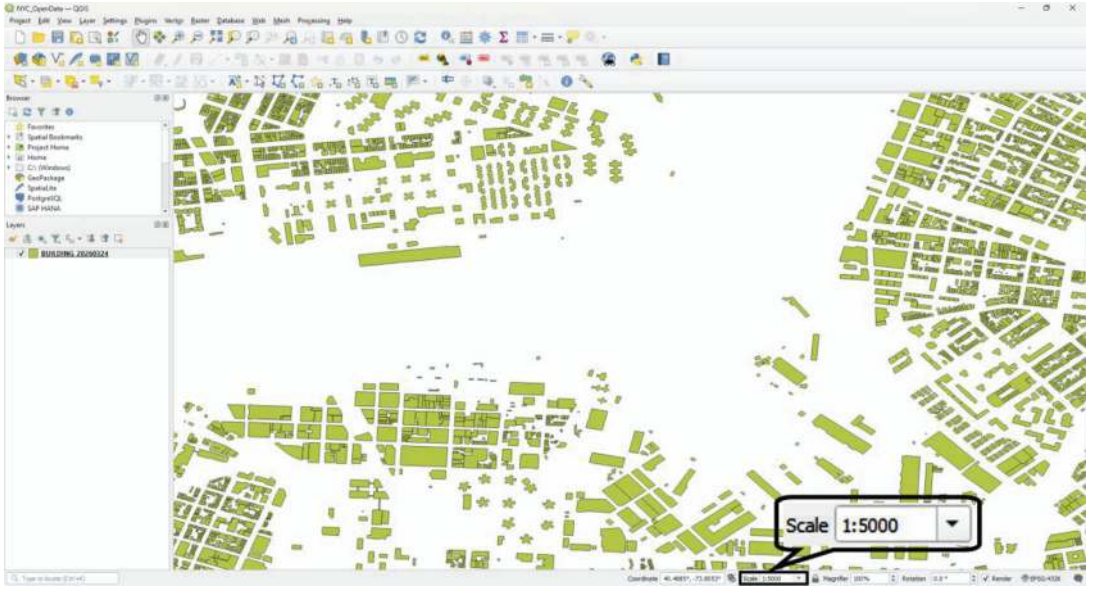


Figure 12-5. *You can type in a scale at the bottom of the QGIS editor (as shown, 1:5000) or scroll the mouse wheel to zoom in until bounding outlines start to appear*

Step 2: Selecting an Area of Interest

The next step is to select an area of interest. Since the data set contains buildings in New York City, we'll make a selection over Time Square—a famous landmark. To help locate the area in the QGIS layer, we'll first look up its coordinates (longitude and latitude) in Google Maps. Head over to `https://maps.google.com`, enter "Time Square" in the search box, press enter, and the map should automatically navigate to and center on the Time Square pin. Move your mouse cursor to the pin and right-click to reveal the coordinates, which are 40.75803, -73.98554, as shown in Figure 12-6.

Enter the Time Square coordinates at the bottom of the QGIS editor and press enter to pan the layer to that location, taking care to preserve the "°" (degree) symbols, as shown in Figure 12-7:

$$(40.75803°, -73.98554°)$$

The editor will take a while to refresh as the layer is large, and you might see building footprints slowly appearing. Once the layer is fully refreshed, click Edit ➤ Select ➤ Select Features by Polygon to enable the selection tool. You will select an area of interest by enclosing it with a polygon. You specify the polygon by clicking and laying down vertices one by one in a clockwise order. Once you are finished placing all vertices, right-click to confirm. You will see the building footprints enclosed by the polygon turn yellow, as shown in Figure 12-7.

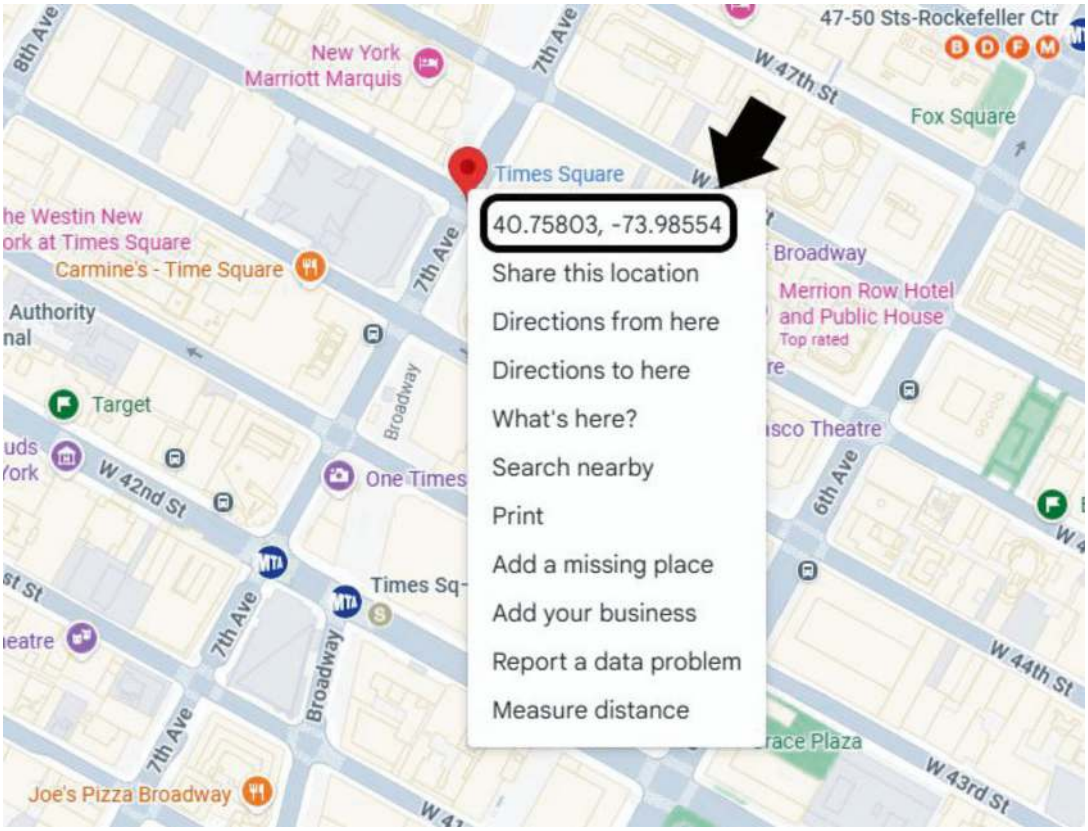


Figure 12-6. *Search for "Time Square" on Google Maps, move the mouse cursor to its pin, and right-click to reveal its coordinates, which are 40.75803, -73.98554*



Figure 12-7. *Enter the coordinates for Time Square at the bottom of the QGIS editor to pan the layer to that location, and use Edit ➤ Select ➤ Select Features by Polygon to select a group of building footprints by enclosing them with a polygon*

Step 3: Exporting the Current Selection to a Spreadsheet

Once a group of shapes are selected, we can export them out to the required file formats. We will export the selection to a spreadsheet (`*.xlsx` file) in this step and a matching shapefile (`*.shp` file) in the following step.

To export a selection, navigate to the "Layers" panel located at the lower left corner of the QGIS editor. Right-click the layer name (there should be only one), and click Export ➤ Save Selected Features As... as shown in Figure 12-8 to bring up the "Save Vector Layer as..." dialog (Figure 12-9).

To export the selection as a spreadsheet, select "MS Office Open XML spreadsheet [XLSX]" in the Format dropdown and click the "..." button to the right of the File name field on the next line to invoke the file open dialog. Navigate to the desired folder, and type in a file name to save the spreadsheet as (e.g., `time_square.xlsx`). Check that the Layer name is correct and CRS is set to "EPSG: 4326 – WGS 84", and click "OK" to export.

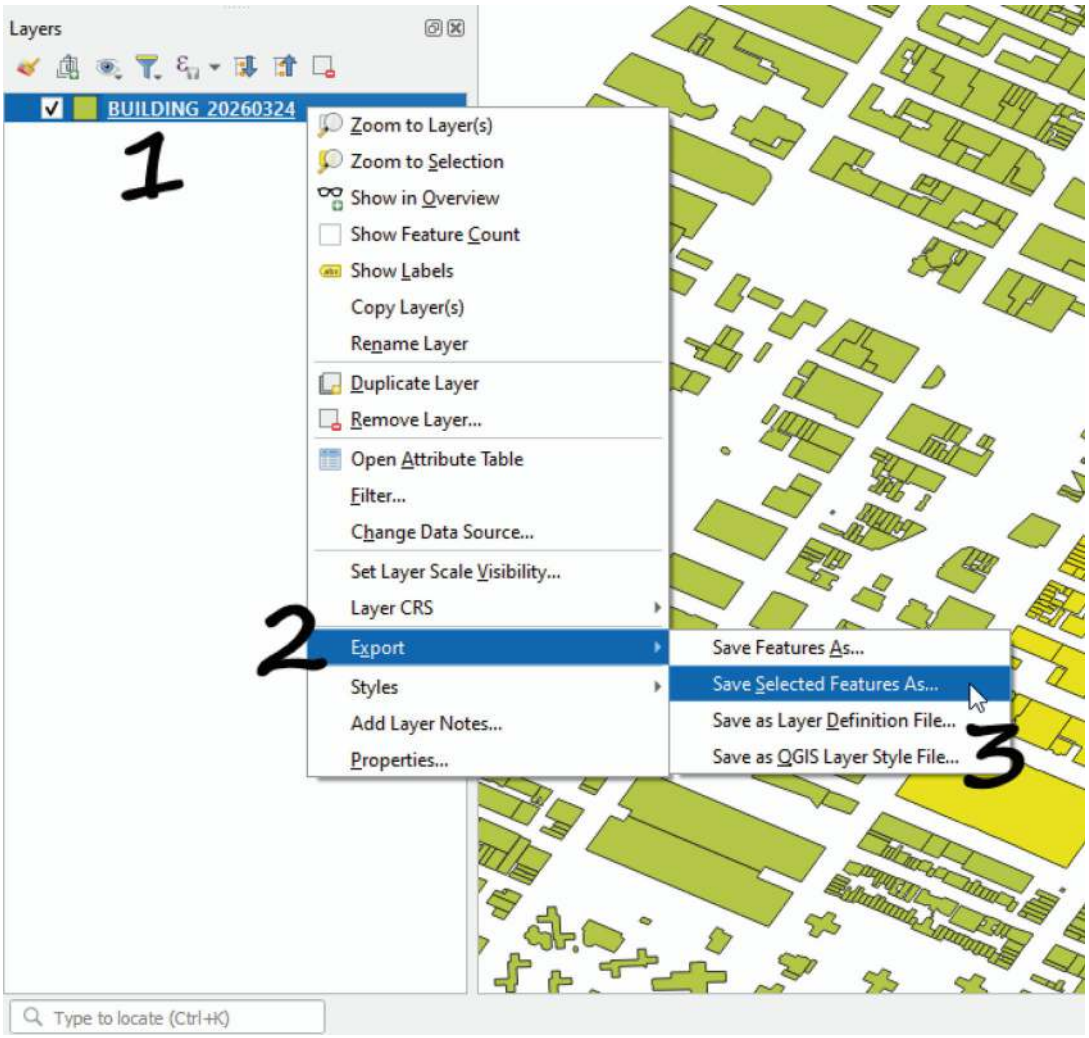


Figure 12-8. *In the "Layers" panel at the lower-left corner of the QGIS editor, (1) right-click the layer name ➤ (2) Export ➤ (3) Save Selected Features As... to bring up the dialog for exporting the current selection to* `*.xlsx` *or* `*.shp` *file*

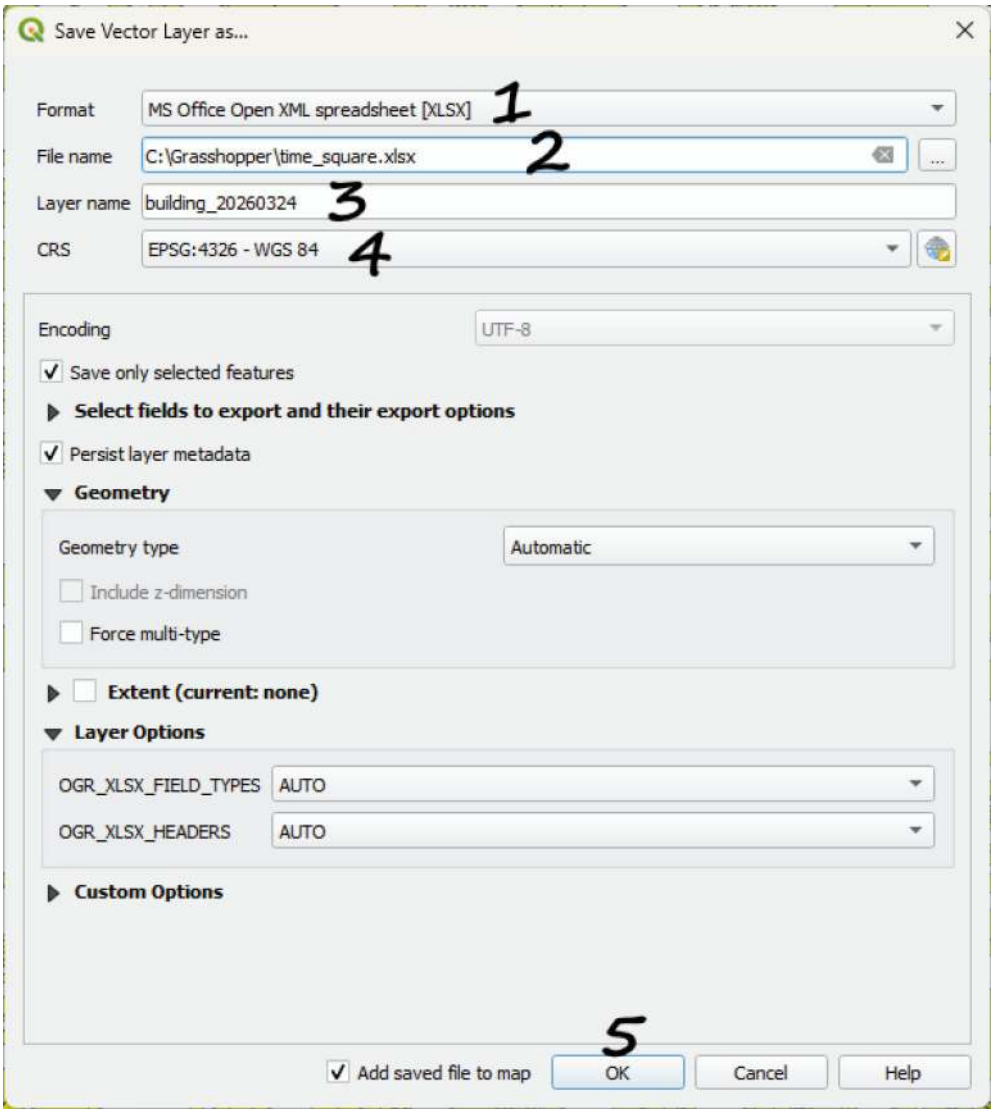


Figure 12-9. *The "Save Vector Layer as..." dialog (invoked by right-clicking a layer's name ➤ Export ➤ Save Selected Features As...). To export the current selection as a spreadsheet, select (1) XLSX as the format, and click the "..." button to the right of the (2) File name field to bring up the file open dialog. Navigate to the desired folder and enter a file name to save as. Check that the (3) Layer name is correct and (4) CRS is WGS 84. Click (5) OK to export*

Step 4: Exporting the Current Selection to a Shapefile

In this step, we'll export the selection to a matching shapefile that coordinates with the spreadsheet exported in the previous step. We will again navigate to the "Layers" panel located at the lower-left corner of the QGIS editor, right-click the layer name (there should be only one), and click Export ➤ Save Selected Features As... as shown in Figure 12-8 to bring up the "Save Vector Layer as..." dialog (Figure 12-10).

To export the selection to a shapefile, select "ESRI Shapefile" in the Format dropdown and click the "..." button to the right of the File name field to invoke the file open dialog. Navigate to the same folder where the spreadsheet is saved. Type in a file name to save the shapefile that matches the name of the spreadsheet (e.g., `time_square.shp`). Check that CRS is set to "EPSG: 4326 – WGS 84" to match the setting for the spreadsheet in the previous step, and click "OK" to export, as shown in Figure 12-10.

Once the export to shapefile is complete, you'll see that a new layer containing the exported shapes is automatically added to the Layers panel. For example, if you've exported the selection to `time_square.shp`, a layer named time_square is added.

If you open the folder containing both the spreadsheet and the shapefile, you'll see that a total of seven files has been exported:

- time_square.cpg
- **time_square.dbf**
- time_square.prj
- time_square.qmd
- **time_square.shp**
- **time_square.shx**
- **time_square.xlsx**

Later in this chapter, we'll read from the `*.shp` and `*.xlsx` files in a Python 3 Script node and use the `*.shp` file as input for the Import SHP node (Params ➤ Input ➤ Import SHP).

Note that Import SHP requires the `*.dbf` and `*.shx` files to reside in the same folder as the `*.shp` file, even though the node only takes the `*.shp` file directly as input.



Figure 12-10. *The "Save Vector Layer as..." dialog (invoked by right-clicking a layer's name ➤ Export ➤ Save Selected Features As...). To export the current selection to a shapefile, select (1) ESRI Shapefile as the format, and click the "..." button to the right of the (2) File name field to bring up the file open dialog. Navigate to the desired folder and enter a file name to save as. Check that (3) CRS is WGS 84 (which matches the XLSX file), and click (4) OK to export*

Parsing and Previewing a Building Footprint Data Set with Third-Party Python Packages in Python 3

Now that you're familiar with the process of downloading and post-processing building footprint data, let's look at how to parse and preview the data in Grasshopper using `pyshp`, `matplotlib`, and `pandas`.

Note You can find a fully implemented copy of the Python 3 Script node for this section in `/Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh`.

Step 1: Configuring Input and Output Ports for the Python 3 Script Node

Follow the steps described in the section "Installing Required Python Packages" to create a new Python 3 Script node with the required references. Zoom in on the Script node until the "+" and "-" buttons appear. Click the "+" buttons to add the inputs and outputs shown in Figure 12-11 and set their Access and Type Hints based on Table 12-1. The purposes of the inputs and outputs are summarized in Table 12-1 as well, which we'll go over in detail shortly.

Note Recall that for a Python 3 Script node, you right-click an input to change its name and set its Access and Type Hints. You also right-click an output to set its name and Type Hints. There is no Access Hint for an output.

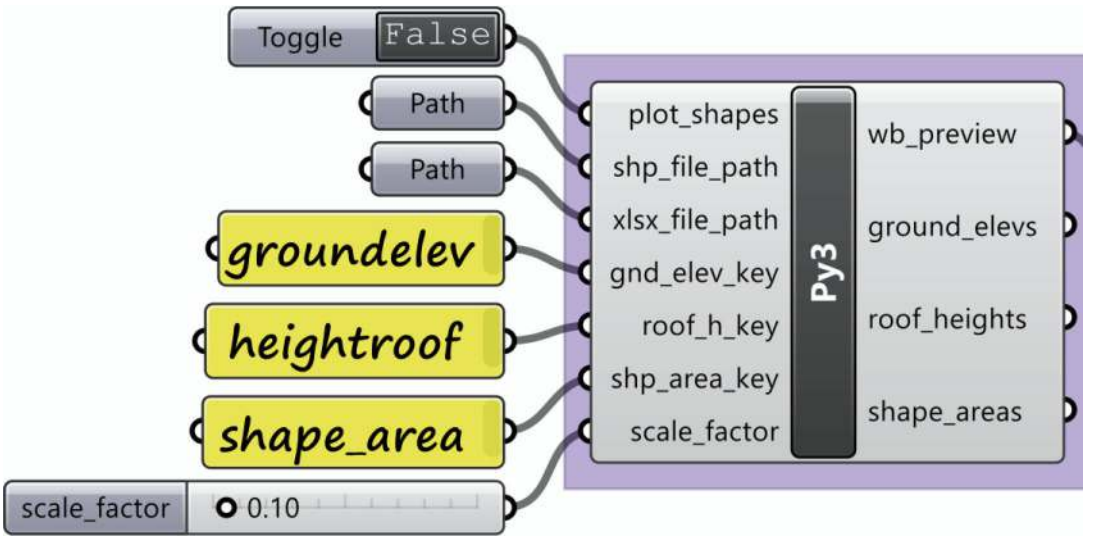


Figure 12-11. *Setting up the inputs and outputs of the Python 3 Script node along with their Access and Type Hints, for parsing and previewing building footprint data using third-party packages*

Table 12-1. *Access and Type Hints for the Python 3 Script node in Figure 12-11*

Name	Input or Output	Access	Type Hints	Purpose
plot_shapes	input	Item Access	bool	True to plot shape polygons
shp_file_path	input	Item Access	File Path	File path to the *.shp file (shapefile)
xlsx_file_path	input	Item Access	File Path	File path to the *.xlsx file (spreadsheet)
gnd_elev_key	input	Item Access	str	Column name for ground elevations in *.xlsx file
roof_h_key	input	Item Access	str	Column name for roof heights in *.xlsx file
shp_area_key	input	Item Access	str	Column name for shape polygon areas in *.xlsx file

(*continued*)

Table 12-1. *(continued)*

Name	Input or Output	Access	Type Hints	Purpose
scale_factor	input	Item Access	float	Scale factor for *.xlsx file contents
wb_preview	output	N/A	str	First five rows of the *.xlsx workbook (link to a Panel to preview)
ground_elevs	output	N/A	float	List of ground elevations
roof_heights	output	N/A	float	List of roof heights
shape_areas	output	N/A	float	List of shape areas

Step 2: Creating Input and Output Values

Once you've configured the input and output ports for the Python 3 Script node, the next step is to create input and output values and link them to the respective ports, as shown in Figure 12-11:

1. Link a Boolean Toggle to *plot_shapes*, which when True triggers plotting of the shape polygons in the *.shp file (shapefile).
2. Link a File Path to *shp_file_path*. Right-click File Path ➤ Select one existing file and select the *.shp file for the building footprint data set. For this example, select /Ch12/qgis_battery_park.shp.
3. Link a File Path to *xlsx_file_path*. Right-click File Path ➤ Select one existing file and select the *.xlsx file (spreadsheet) for the building footprint data set. For this example, select /Ch12/qgis_battery_park.xlsx.
4. Create a Panel, and type in the name of the column containing the ground elevation values in the *.xlsx file (for the Battery Park example, groundelev). Link the Panel to *gnd_elev_key*.
5. Create a Panel, and type in the name of the column containing the roof height values in the *.xlsx file (for the Battery Park example, heightroof). Link the Panel to *roof_h_key*.

6. Create a Panel, and type in the name of the column containing the areas of the shape polygons in the *.xlsx file (for the Battery Park example, shape_area. Link the Panel to *shp_area_key*.
7. Link a Number Slider (set to 0.10) to *scale_factor,* for scaling all values contained in the *.xlsx file uniformly.
8. Link the *wb_preview* output to a Panel to see a printout of the first five rows of the *.xlsx file.
9. Link the *ground_elevs* output to a Panel to preview the list of ground elevation values read from the *.xlsx file.
10. Link the *roof_heights* output to a Panel to preview the list of roof heights read from the *.xlsx file.
11. Link the *shape_areas* output to a Panel to preview the list of shape areas read from the *.xlsx file.

Step 3: Checking References to Packages and Adding Import Statements

Once the input and output connections are established, we'll check that references for the third-party packages are set up properly and add import statements. Double-click the Python 3 Script node to launch the Script Editor window. You should see the first four lines in Listing 12-1 already loaded in the code area—these lines were added by Grasshopper automatically when packages are installed from the PyPl server with the "Add Package Reference to Script" option checked (see Figure 12-1). For example, the first line reads # r "pip: pyshp>=3.0.3", which tells GH to look for pyshp with a *minimum* version of 3.0.3.

The next block of code in Listing 12-1 imports modules and functions from the third-party packages: We import `pandas` as the shorthand `pd` to parse the spreadsheet (`*.xlsx` file). `pyshp` is imported as `shapefile` to parse the `*.shp` file containing horizontal building outlines as polylines. We import two items from the `matplotlib` package, `pyplot` and the Polygon type from `patches`, for plotting polylines from the `*.shp` file to preview them. Lastly, we import `numpy` as the shorthand `np`, to generate random roof heights and ground elevation values in the event they are missing from the spreadsheet.

Listing 12-1. References for third-party packages and import statements for the Python 3 Script node (see `/Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh`)

```
# r "pip: pyshp>=3.0.3"
# r "pip: pandas>=3.0.1"
# r "pip: matplotlib>=3.10.8"
# r "pip: openpyxl>=3.1.5"

import pandas as pd
import shapefile
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
import numpy as np
```

Step 4: Reading Contents of a Shapefile with Python 3 Using `pyshp`

With the imports squared away, next we'll write a Python function `read_shp_file` as shown in Listing 12-2 to read the contents of the shapefile.

The code takes a file path to a shapefile (`shp_file_path`) with which to call `shapefile.Reader()` to retrieve an object (`shp_file`) containing the contents of the file. The call to `shp_file.shapes()` then gives the shape polylines.

The next two lines extract the portion of the shapefile name *without* the extension as follows: take the file path, break it into pieces at either forward ("/") or back slashes ("\\"), grab the last piece (the file name *with* the extension), split it further at the dot ("."), and grab the part *before* the dot.

For example, the file path "`C:/Ch12/qgis_battery_park.shp`" has the forward slash "/" as delimiter, which we use to break into the list of strings ["C:", "Ch12", "qgis_battery_park.shp"]. The last entry of the list (at index –1) is "qgis_battery_park.shp", which when split further with the dot (".") becomes the list of two strings, ["qgis_battery_park", "shp"]—the first entry of which is the portion of the file name *without* the extension.

Lastly, we return the object containing the shapefile content (`shp_file`), the list of polylines (`shapes`), and the shapefile name without the extension (`shp_file_name`). In the next section, we'll use the return value of this function as the basis for plotting the polylines using `matplotlib`.

Listing 12-2. Function to read the contents of a shapefile using pyshp (see /Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh)

```
def read_shp_file(shp_file_path: str):
    shp_file = shapefile.Reader(shp_file_path)
    shapes = shp_file.shapes()

    delimiter = "/" if "/" in shp_file_path else "\\"
    shp_file_name = ↩
        shp_file_path.split(delimiter)[-1].split(".")[0]

    return shp_file, shapes, shp_file_name
```

Step 5: Plotting Shape Polygons with Python 3 Using `matplotlib`

With the shapefile contents parsed, next we'll use `matplotlib` to visualize the polylines. You may wonder why we bother to plot the polylines first—the reason is that the scales of building footprint sets vary widely (feet, meter, km, ..., and everything in between). Often the only way to find out is to dig into the data. By plotting with `matplotlib`, we can immediately find the range of the shape vertex coordinates from the graph axes—which when cross-referenced with real world measurements of the same municipal area, allow us to derive the scale.

In Listing 12-3, we write the function `plot_shp_file` which takes the path to the `*.shp` file as a string, uses it to call `read_shp_file` (Listing 12-2) to parse the shapefile, and plots the shape polygons using `matplotlib` functions.

Listing 12-3. Function to plot shape polylines using matplotlib (see /Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh)

```
def plot_shp_file(shp_file_path: str):
    shp_file, shapes, shp_file_name = read_shp_file(shp_file_path)

    fig, axes = plt.subplots()
    x_min, y_min, x_max, y_max = shp_file.bbox
    axes.set_xlim(x_min, x_max)
    axes.set_ylim(y_min, y_max)
    axes.set_title(shp_file_name)
```

```
for shape in shapes:
    points = shape.points
    polygon = Polygon(points, edgecolor='green', facecolor='lightgreen')
    axes.add_patch(polygon)

plt.show()
```

We start by calling `read_shp_file` (Listing 12-2) to retrieve the object containing the shapefile's contents (`shp_file`), the list of shape polylines (`shapes`), and the name of the shapefile without the extension (`shp_file_name`)—we'll be using all three to construct the plot shortly.

The next block of code creates the plot window along with plot axes and title. `shp_file.bbox` gives the bounding box of the shapes collectively as four values—the minimum X, minimum Y, maximum X, and maximum Y, which we use to set the x and y axes limits on the next two lines (`axes.set_xlim`, `axes.set_ylim`). The plot title is set next using the name of the shapefile without the extension. For example, for "`qgis_battery_park.shp`", the plot title is "qgis_battery_park".

We iterate through the list of polylines with a `for` loop in the next block. For each shape, we call the `Polygon` function to draw it using its vertices (`shape.points`). The edge and fill color are specified via `edgecolor` and `facecolor`, respectively (which are green and light green here). The next line adds the shape to the plot (`axes.add_patch`).

After the `for` loop concludes and all shapes are added to the plot, we call `plt.show()` to display the plot window.

In Figure 12-11, we linked a Boolean Toggle (set to False) to the Python 3 Script node's `plot_shapes` input. In the Python code, we use `plot_shapes` to control whether `plot_shp_file` is called, like this:

```
if plot_shapes:
    plot_shp_file(shp_file_path)
```

Using this setup, the plot is only produced when you double-click to toggle the Boolean Toggle's value to True.

Note `shp_file.bbox` is similar to GH's Bounding Box node set to right-click ➤ Union Box, which gives the smallest box enclosing all the polylines collectively.

Recall that `Polygon` is a type under the `matplotlib.patches` module imported in Listing 12-1.

Deriving Scale from the Preview Plot

The result of calling Listing 12-3 with the provided `/Ch12/qgis_battery_park.shp` file is shown in Figure 12-12. We can now derive the scale of the shapefile from the plot as follows:

$$X\ axis\ spans\ the\ range = -73.995 - \left(74.015\right) = 0.02$$

$$Y\ axis\ spans\ the\ range = 40.72 - 40.70 = 0.02$$

Battery Park has an area of

$$25\ \ acres = 0.0390625\ square\ miles$$

$$\sqrt{0.0390625} = 0.1976423538 \approx 0.2 mile$$

Therefore, a length of 0.02 on the plot corresponds to 0.2 mile.

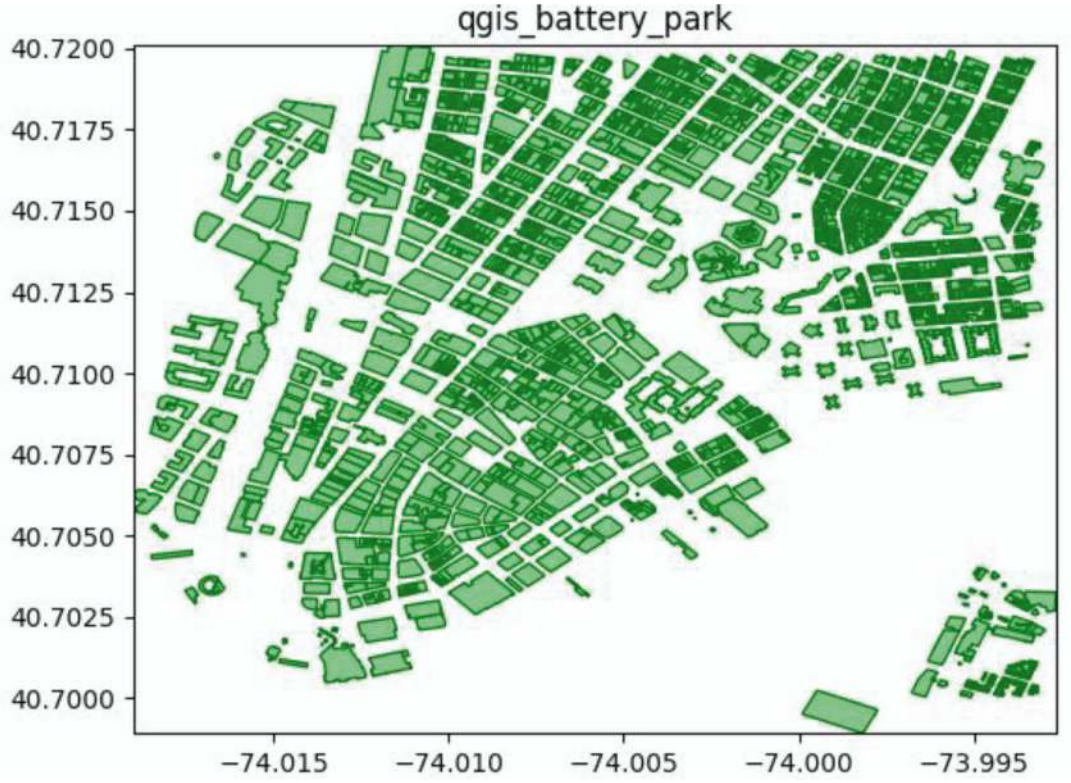


Figure 12-12. *Shapes from the provided shapefile* `/Ch12/qgis_battery_park.shp`*, plotted using the Python 3 code in Listing 12-3*

Step 6: Reading the Contents of a Spreadsheet with Python 3 Using pandas

Next, we'll use `pandas` to parse the contents of the spreadsheet and preview the first few rows in a Panel as shown in Listing 12-4. We imported `pandas` as the shorthand `pd` in Listing 12-1. Here, we call `pd.read_excel` with the input `*.xlsx` file path, which returns an object (`wd`) containing the contents of the spreadsheet. Calling `wb.head()` then gives the first five rows of the spreadsheet, which we assign to the output `wb_preview`. Recall that in the section "Step 2: Creating Input and Output Values," we linked `wb_preview` to a Panel and set `xlsx_file_path` to `/Ch12/qgis_battery_park.xlsx`. As a result, the Panel is now populated with the first five rows of this spreadsheet, as shown in Figure 12-13.

Listing 12-4. Parsing `*.xlsx` file with pandas and previewing the first five lines (see `/Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh`)

```
wb = pd.read_excel(xlsx_file_path)
wb_preview = wb.head()
```

wb_preview

{0}

	bin	cnstrct_yr	doitt_id	...	name	shape_area	shape_len
0	3001592	1899	31218	...	NaN	574.947956	117.845191
1	1005645	1930	79482	...	NaN	6719.739882	329.329988
2	1002184	1920	426267	...	NaN	643.970404	109.754291
3	1077207	1930	37237	...	NaN	3866.526338	301.277681
4	1003296	1915	708848	...	NaN	3520.843385	322.676725

[5 rows x 12 columns]

Figure 12-13. *First five rows of the provided spreadsheet* `/Ch12/qgis_battery_park.xlsx`*, output via the "wb_preview" port of the Python 3 Script node (Figure 12-11) and shown in a Panel*

Reading the Roof Heights, Ground Elevations, and Shape Areas Based on Input Column Names

Next, we'll read the roof heights, ground elevations, and shape areas from the spreadsheet based on their respective column names. Unfortunately, the column names for these values are not standardized across footprint data sets and sometimes missing altogether. The only way to find what the column names are is to either open the file in a spreadsheet editor like Microsoft Excel, Google Sheets, OpenOffice Calc, etc., or to extract them from the `pandas` workbook object (`wb`) that we've created in Listing 12-4 and print them to the Terminal area at the bottom of the Python 3 Script node's Script Editor, using the following line of code

```
print(list(wb.columns))
```

which prints

```
['bin', 'cnstrct_yr', 'doitt_id', 'feat_code', 'groundelev', 'heightroof',
'date_lstmo', 'time_lstmo', 'laststatype', 'name', 'shape_area',
'shape_len']
```

Therefore, in Figure 12-11, we've set the inputs for the Python 3 Script node as follows: `gnd_elev_key` to groundelev, `roof_h_key` to heightroof, and `shp_area_key` to shape_area. Note that these column names are specific to the file `/Ch12/qgis_battery_park.xlsx` and will be different for other data sets you may acquire.

In Listing 12-5, we write the Python code to read from the spreadsheet using these three column names. We first check if a given column name is in the workbook (e.g., `if gnd_elev_key in wb.keys()`)—if so, we proceed to read it from the workbook and multiply it by the input `scale_factor`. If the column name is not in the workbook, we call the function `get_sample` (Listing 12-6) to generate random values for that column, which we'll look at next.

Listing 12-5. Reading the roof heights, ground elevations, and shape areas with pandas and generate random stand-in values if a column doesn't exist (see `/Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh`)

```
num_shapes = len(shapes)
ground_elevs, roof_heights, shape_areas = [], [], []

if gnd_elev_key in wb.keys():
    ground_elevs = list(wb[gnd_elev_key]*scale_factor)
else:
    ground_elevs = get_sample(num_shapes, 10, 30)*scale_factor

if roof_h_key in wb.keys():
    roof_heights = list(wb[roof_h_key]*scale_factor)
else:
    roof_heights = get_sample(num_shapes, 40, 100)*scale_factor

if shp_area_key in wb.keys():
    shape_areas = list(wb[shp_area_key]*scale_factor)
else:
    shape_areas = get_sample(num_shapes, 600, 6000)*scale_factor
```

Generating Random Values for Missing Columns in the Spreadsheet

In Listing 12-6, we implement a simple function `get_sample` to generate random values for a missing column in the spreadsheet. The function has three arguments, the number of items to generate (`num_items`) and the range to generate the numbers in as `max` and `min`.

The `numpy` function `np.random.random_sample` generates a given count of values (`num_items`) between 0 and 1. To map the values to between min and max, we first multiply the values by max-min, which maps them from the range

$$0 \rightarrow 1$$

to the range

$$0 \rightarrow (\max - \min)$$

By adding min on both ends, we shift the range to

$$0 + \min \rightarrow (\max - \min) + \min$$

which is equivalent to

$$\min \rightarrow \max$$

Listing 12-6. Using `numpy` to generate random values for filling in a missing column in the spreadsheet (see `/Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh`)

```
def get_sample(num_items, max, min):
    span = max-min
    return np.random.random_sample(num_items)*span+min
```

Note You can find the complete node graph from this section in the file `/Ch12/Figure12-11_parsing_shp_and_xlsx_files_python3.gh`.

Generating Buildings

Now that we've written the Python 3 code for parsing the spreadsheet and deriving shape scales, let's start building the node graph for generating city skyline Breps as shown in Figure 12-14. The graph consists of four stages. For stage 1, we'll reuse the Python 3 Script node from Figure 12-11 but delete the output Panels since we no longer need the previews. Stage 2 will scale the shape polylines and turn them into trimmed 2D surfaces. Stage 3 will move the surfaces to center at the world origin (0, 0, 0). Stage 4 will extrude the footprint surfaces into a collection of building Breps. We will continue to use the provided Battery Park, NYC, data set for this section.

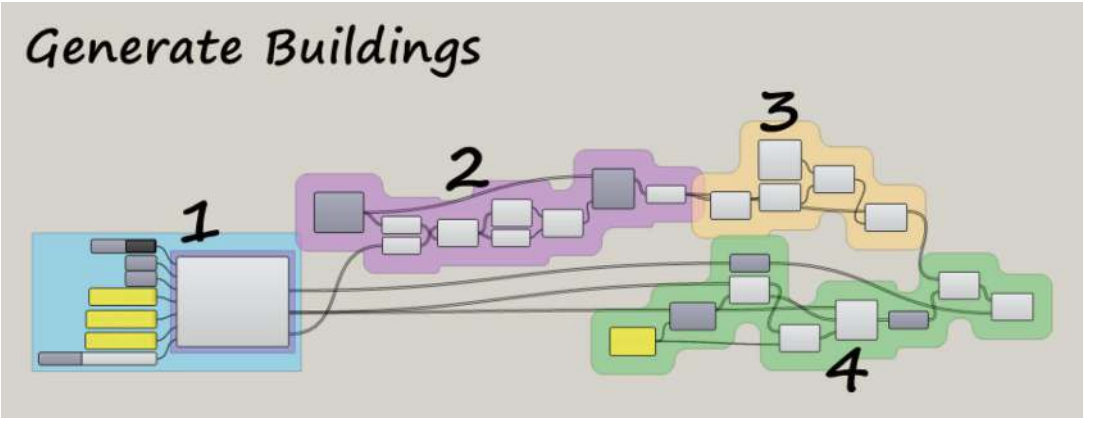


Figure 12-14. *Generating the city skyline Breps in a four-stage process, by reusing the Python 3 Script node setup from Figure 12-11 as stage 1 for parsing the spreadsheet*

Stage 1: Parsing the Spreadsheet and Filling in Missing Values

For stage 1 (Figure 12-15), we'll reuse the Python 3 Script node setup from Figure 12-11 minus the output Panels. Recall that the outputs `ground_elevs`, `roof_heights`, and `shape_areas` are the values for ground elevations, building roof heights, and areas of building cross-sections read from the spreadsheet, respectively.

Note Remember that Figure 12-15 shows the column names for `/Ch12/qgis_battery_park.xlsx`, which you'll need to swap out for a different data set, along with file paths for the `*.shp` and `*.xlsx` files.

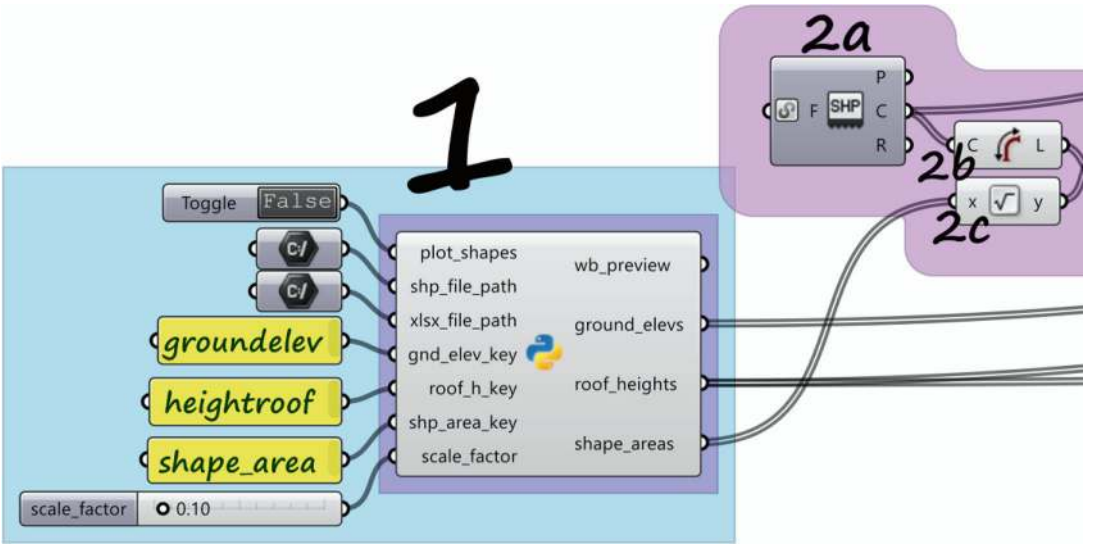


Figure 12-15. *Reusing the Python 3 Script node setup from Figure 12-11 minus the output Panels as stage 1, for parsing the spreadsheet and generating missing values*

Stage 2: Turning the Polylines Outlining Building Footprints into Trimmed Surfaces

In stage 2 (Figure 12-16), we'll use the Import SHP node (2a) to read the shape polylines from `/Ch12/qgis_battery_park.shp`, scale them based on the corresponding shape areas from the spreadsheet, and turn the scaled polylines into trimmed surfaces.

Although we've parsed and previewed the shapefile using `pyshp` in Figure 12-11, it is more convenient to use the Import SHP node (2a) here to read the shapefile since it outputs the shapes directly as GH polyline curves via the C output. Note that Import SHP requires that the `*.dbf` and `*.shx` files exported along with the `*.shp` file reside in the same folder, even though it only takes the `*.shp` file directly as an input. Since we are using `/Ch12/qgis_battery_park.shp` for this example, the files `qgis_battery_park.dbf` and `qgis_battery_park.shx` must be in the same folder as well.

Since the polylines from the shapefile do not necessarily match in scale with their counterparts from the spreadsheet, we'll cross-reference the two and scale the polylines so they have the same order of magnitude. The polygons are irregularly shaped; however, we can get a pretty good estimate by treating them as squares. We can find the side length of a square by taking the square root of its area. Therefore, by taking the Length (2b) of the polyline curves, which are the *perimeters* of the shapes, and dividing (2d) them by the approximated side length from the Square Root (2c) of the shape areas, we get the list of scale factors that will match the two in order of magnitude.

For simplicity, we'll compute a single average scale factor by summing the factors using Mass Addition (2e) and dividing it (2g) by the number of shapes (2f, List Length). We then Scale (2h) all polylines uniformly by this average scale factor. To wrap up, we link the scaled polylines (2h) to a Boundary Surfaces node (2i) to turn them into trimmed surfaces.

Note Where to find the nodes in stage 2 of Figure 12-16: Params ➤ Input ➤ Import SHP (2a), Curve ➤ Analysis ➤ Length (2b), Maths ➤ Polynomials ➤ Square Root (2c), Maths ➤ Operators ➤ Division (2d, 2g), Maths ➤ Operators ➤ Mass Addition (2e), Sets ➤ List ➤ List Length (2f), Transform ➤ Affine ➤ Scale (2h), and Surface ➤ Freeform ➤ Boundary Surfaces (2i).

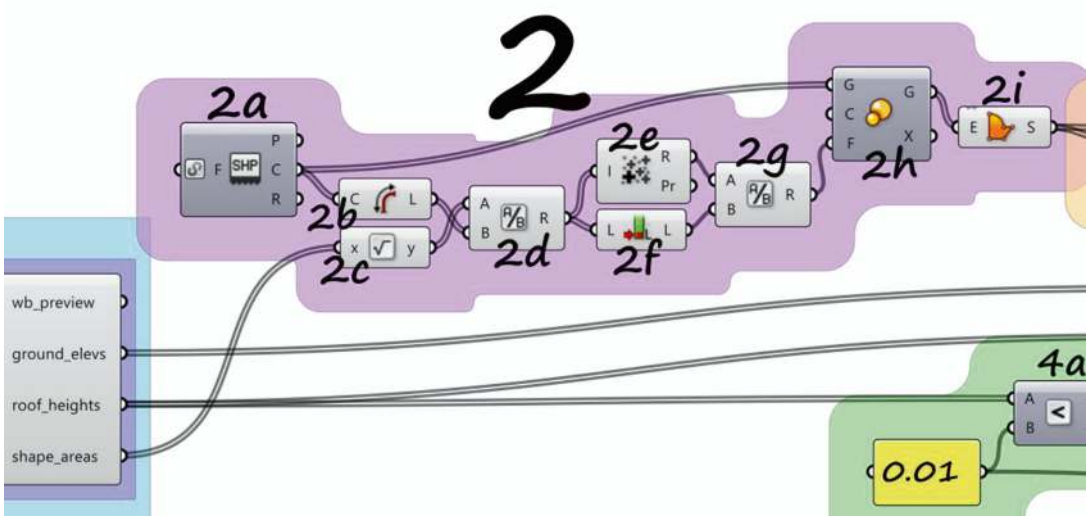


Figure 12-16. *Stage 2 of generating skyline Breps involves scaling shape polylines by cross-referencing their lengths with the areas they enclose from the spreadsheet and turning the scaled polylines into trimmed surfaces*

Stage 3: Moving the Footprint Surfaces to Center at the Origin (0, 0, 0)

In Stage 3 (Figure 12-17), we'll move the scaled shape surfaces to center at the world origin (0, 0, 0) so it's easier to find them in the viewport.

We start by linking Boundary Surfaces (2i) to Bounding Box (right-click ➤ Union Box, 3a) to find the minimal box that will enclose the surfaces as a whole. We then use Area (3c) to find the bounding box center. To compute the vector for moving the bounding box center to the origin, we subtract (3d) the bounding box center from a point at (0, 0, 0) created using Construct Point (3b) with default parameters. Finally, we move the collection of surfaces (2i) using a Move node (3e) with the vector (3d) as its T input. Figure 12-18 shows the surfaces after the movement.

Note Where to find the nodes in stage 3 of Figure 12-17: Surface ➤ Primitive ➤ Bounding Box (right-click ➤ Union Box) (3a), Vector ➤ Point ➤ Construct Point (3b), Surface ➤ Analysis ➤ Area (3c), Maths ➤ Operators ➤ Subtraction (3d), and Transform ➤ Euclidean ➤ Move (3e).

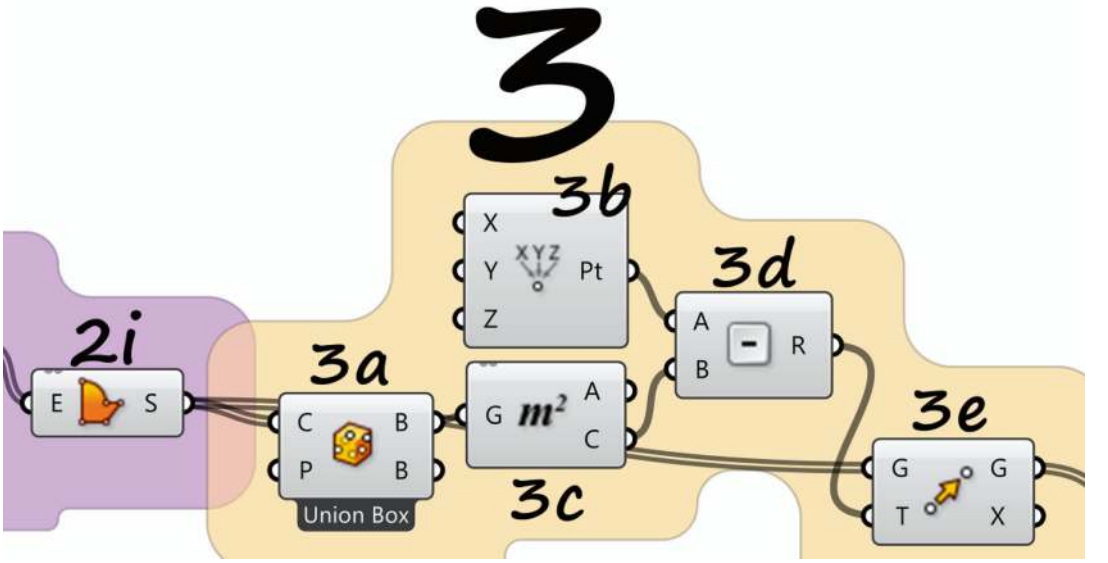


Figure 12-17. *Stage 3 of generating skyline Breps involves moving the collection of building footprint surfaces to center at the world origin (0, 0, 0)*



Figure 12-18. *Trimmed shape surfaces after they've been relocated to center at the world origin (0, 0, 0)*

Stage 4: Extruding Footprint Surfaces into Buildings and Offset Them Along Z Based on Ground Elevations

In stage 4 (Figure 12-19), we'll extrude the footprint surfaces along +Z based on roof heights read from the spreadsheet in stage 1 and offset them based on ground elevation values.

One thing you'll find about working with real-world GIS data is that there could be errors or noises in the set. The provided `/Ch12/qgis_battery_park.xlsx` file indeed contains some zero roof heights, which will cause errors if used directly with Extrude.

To handle this gracefully, we'll use a Dispatch (4b) node to separate the (near) zero values and add 0.01 to them, before weaving them back in (4d) with the rest of the data as follows: We link the Python 3 Script node's `roof_heights` output to the A input of a *Smaller Than* node (4a) with 0.01 as the B input, which gives a list of Boolean values indicating which entries are less than 0.01. We then use the list of Booleans as the Pattern input for Dispatch (4b) to separate the roof heights < 0.01 (A) and ≥ 0.01 (B). We add (4c) 0.01 to A to make everything in that sublist *at least* 0.01, before recombining it with B using the Weave (4d) node. Essentially, we've added 0.01 *only* to those roof heights < 0.01 to ensure that they are at least 0.01.

We're now ready to extrude the footprint surfaces. To do so, we link the adjusted roof heights (4d) to Unit Z (4e) to use as the D input for Extrude (4f) and the moved footprint surfaces (3e) as the B input.

With the footprints extruded into solids, the last step is to move the buildings along Z based on ground elevation values read from the spreadsheet. To do so, we link the Python 3 Script node's `ground_elevs` output to Unit Z (4g) to use as the T input for Move (4h) and link the output of Extrude (4f) to the G input.

The G output of 4h, which is also the final output of the complete node graph (Figures 12-14 to 12-17 and 12-19), is shown in Figure 12-20.

Note Where to find the nodes in stage 4 of Figure 12-19: Maths ➤ Operators ➤ Smaller Than (4a), Sets ➤ List ➤ Dispatch (4b), Maths ➤ Operators ➤ Addition (4c), Sets ➤ List ➤ Weave (4d), Vector ➤ Vector ➤ Unit Z (4e, 4g), Surface ➤ Freeform ➤ Extrude (4f), and Transform ➤ Euclidean ➤ Move (4h).

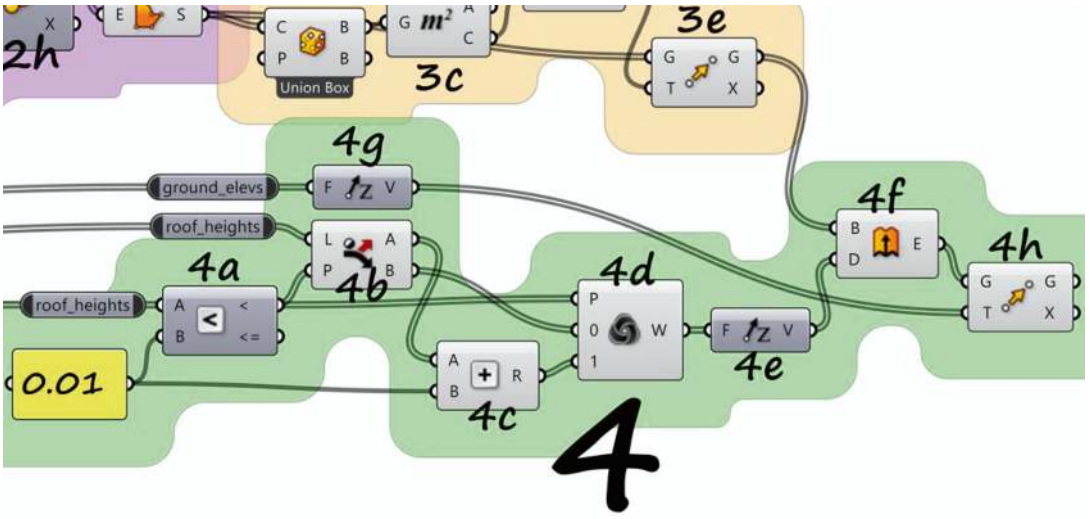


Figure 12-19. *In stage 4 of the skyline generation, we extrude the footprint surfaces into building Breps and offset them along Z based on ground elevation values*

Note You can find the complete node graph from this section (Figures 12-14 to 12-17 and 12-19 in `/Ch12/Figure12-14_to_12-19_generating_buildings.gh`.



Figure 12-20. *Final output of the skyline generator node graph (Figures 12-14 to 12-17 and 12-19) or the G output of the Move node (4h) in Figure 12-19, based on the provided Battery Park, NYC, data set*

Generating the Havana Cityscape

Let's try generating the cityscape of Havana, Cuba, using the second provided building footprints data set. To do so, you will need to modify the parameters of the node graph from Figures 12-14 to 12-17 and 12-19 as follows:

1. Modify the File Path linked to the *shp_file_path* input of the Python 3 Script node. Right-click File Path ➤ Select one existing file and select `/Ch12/havana.shp`.
2. Modify the File Path linked to the *xlsx_file_path* input of the Python 3 Script node. Right-click File Path ➤ Select one existing file and select `/Ch12/havana.xlsx`.
3. Double-click the Panel linked to the *gnd_elev_key* input of the Python 3 Script node and change the text inside to "missing".
4. Double-click the Panel linked to the *roof_h_key* input of the Python 3 Script node and change the text inside to "missing".
5. Double-click the Panel linked to the *shp_area_key* input of the Python 3 Script node and change the text inside to "area_in_ meters".
6. Right-click the input of the Import SHP node ➤ Select one existing file and select `/Ch12/havana.shp`.
7. Right-click the input of the Import SHP node ➤ Synchronize.
8. Set the Number Slider linked to the scale_factor input of the Python 3 Script node to 0.02.

Note that the ground elevations and roof heights are missing in `/Ch12/havana.xlsx`; therefore, we use "missing" for their column name inputs and rely on the Python 3 Script node to generate random values for these two columns. The portion of the node graph containing all the changes described in this section is shown in Figure 12-21. The preview plot of `/Ch12/hanava.shp` is shown in Figure 12-22 (the plotting of which is triggered by double-clicking the Boolean Toggle linked to the `plot_shapes` input of the Python 3 Script node in Figure 12-21). Figure 12-23 shows the cityscape generated (or the G output of the Move node (4h) in Figure 12-19).

Note You can find the node graph containing the complete modification from this section in `/Ch12/Figure12-21_to_12-23_generating_buildings_havana.gh`.

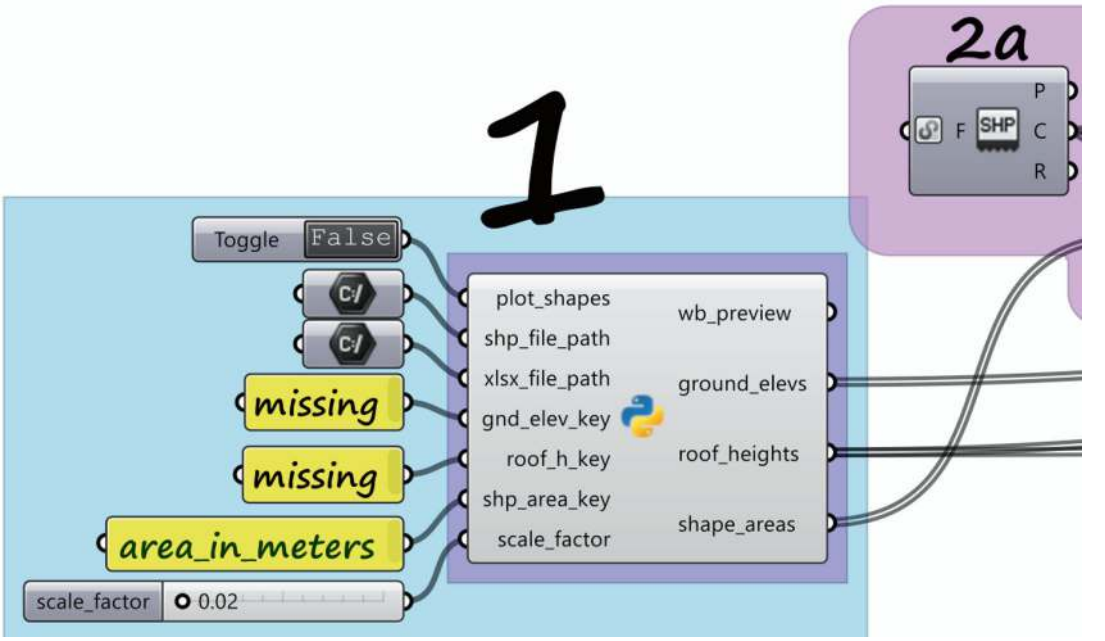


Figure 12-21. *Input parameters for the Python 3 Script node from Figure 12-15 modified according to the provided Havana, Cuba, data set*

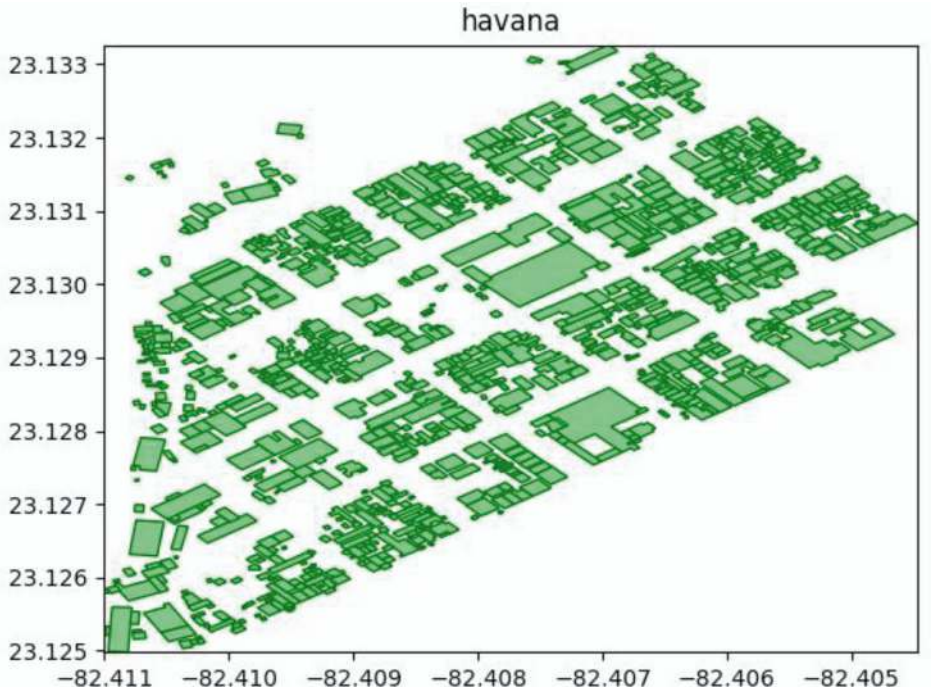


Figure 12-22. *The provided /Ch12/havana.shp file parsed with pyshp and plotted using matplotlib (double-click the Boolean Toggle linked to the plot_shapes input of the Python 3 Script node in Figure 12-21 to trigger the plotting)*



Figure 12-23. *The cityscape generated by the node graph from Figures 12-14 to 12-17 and 12-19 modified to use the provided Havana, Cuba, data set*

Summary

Building footprints is a form of GIS (Geographic Information System) data that contains 2D cross-sections of buildings in the form of polylines along with related statistics like roof heights and ground elevations. Footprint data sets are widely available online and usually contain data for a municipal area like a city or a county. In this chapter, we worked through the process of sourcing building footprint data online, post-processing it in QGIS, previewing and analyzing the data in Python, and generating cityscape models from the data.

In the first part of the chapter, you downloaded a data set from NYC OpenData as a `*.csv` file, which is the most common format for footprint data. You then learned how to import the data into QGIS, isolate an area of interest, and export it in formats required for the purpose of model generation in Grasshopper (`*.shp`, `*.dbf`, `*.shx`, and `*.xlsx`).

In the second part of the chapter, you implemented a Python 3 Script node to preview building footprint polylines and extract data like roof heights and ground elevations from the corresponding spreadsheet, using third-party Python packages `pyshp`, `pandas`, `matplotlib`, and `numpy`. You also learned about the inconsistencies between footprint data sets, how to deal with missing fields, and how to derive scale by cross-referencing the data with real-world measurements.

In the final part of the chapter, you built a Grasshopper definition to generate skyline Breps based on building footprints data and tested the definition using two provided data sets (Battery Park, NYC, and Havana, Cuba).

Reference

Lupiani, Isabel. *Procedural Content Generation for Games: Automate 3D Asset and Environment Creation with Blender Python and Geometry Nodes*. First Edition, Apress, 2025.

CHAPTER 13

Packaging Rhino Python and Grasshopper Scripts As Plugins for Publication

This chapter is all about building and publishing Grasshopper plugins and Rhino commands from Grasshopper definitions containing Python 3 Script nodes. The first part of the chapter guides you through creating a GH plugin in two ways—as one component per Python 3 Script node or with all scripts combined into a single component. Next, you'll learn how to package and publish your plugins to Yak—the Rhino Package Server. After that, the chapter shows you how to build an interactive Rhino command which prompts users for parameter values. Finally, we'll go over how to share GH definitions with others without building plugins by using the Remote Control Panel to expose parametric controls.

Running This Chapter's Examples

The source code and color figures for this book are available on GitHub via the book's product page, located at `https://github.com/Apress/Parametric-and-Generative-Design-with-Rhino-Python-and-Grasshopper`. Since this chapter is about building Grasshopper plugins and Rhino commands, the sample files are organized such that each folder contains the source, projects, `*.gha` (GH plugin) and `*.rhp` (Rhino command) installer files, as well as their `*.yak` packages for distribution via the Rhino Package Server. I suggest that you create your own files by following the chapter text and use the provided files to double-check your work after.

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2_13

Building a Grasshopper Plugin

In this section, we'll build a 2D L-System Grasshopper plugin using the GH definitions we've developed in Chapter 11: L-System (Lupiani, 2025; Prusinkiewicz et al., 1990). We will discuss how to structure the plugin in two ways—first as a panel of two components (RewriteString and Turtle2D) and then with everything combined into a single component.

Note You can find all the files for this section in the `/Ch13/BuildingGrasshopperPlugin` folder in this chapter's downloaded source.

Creating a ScriptEditor Project

Rhino 8+ provides a feature called *ScriptEditor* that lets you easily create Grasshopper plugins and Rhino commands from Grasshopper definitions containing Python 3 Script nodes. Enter "ScriptEditor" in the Rhino command line as shown in Figure 13-1 to bring up the ScriptEditor window, followed by File ➤ Create Project to create a new project.

Upon clicking Create Project, a window will appear that prompts you to enter settings for the project (Figure 13-2) with the required values shown in bold. The **Id** number is a unique identifier automatically assigned by Rhino that should ***not*** be changed under normal circumstances. You can however enter a custom **Name**, **Description**, and **Version** number at will. The Version number follows the format *major.minor.patches* with a "PreRelease" option (which when checked appends "-beta" to the version, e.g., 1.0.0-beta). One or more **Author(s)** is also required, while Copyright, License, Keywords, and URL are optional and can be left blank.

Notice that there are two dropdown menus at the upper-right corner (shown as 10 and 11 in Figure 13-2) for specifying a light theme (left) and dark theme (right) icon for the project, respectively. Once all the details are filled in, click Save Changes to finish creating the project.

Note Using "major.minor.patches" to version software is a practice called Semantic Versioning. The "patches" field denotes bug fixes that do not change functionality. "minor" may add new features but preserve existing interfaces (e.g., a function may still be called the same way but its underlying implementation may have changed). "major" introduces large, breaking changes that are not backward compatible.

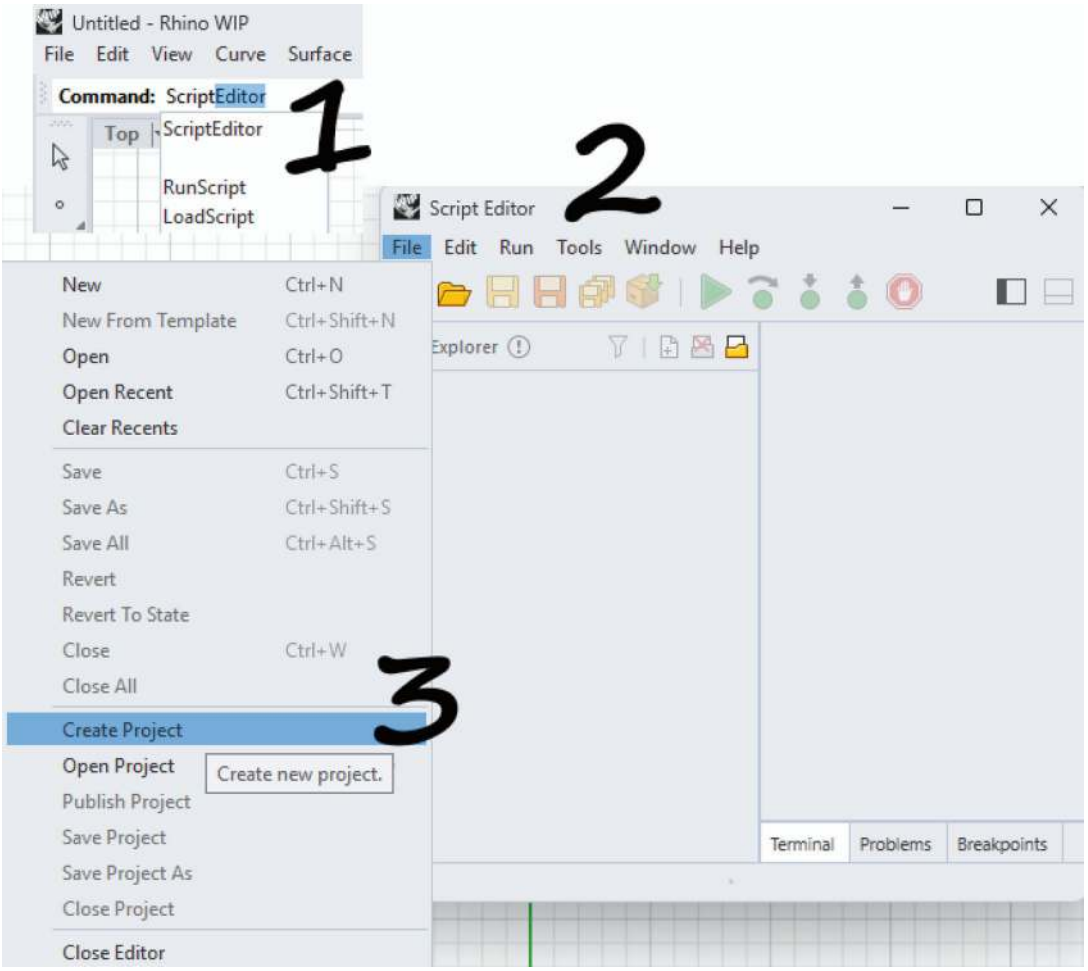


Figure 13-1. *Enter "ScriptEditor" in the Rhino command line (1) to open the ScriptEditor window (2) and click File ➤ Create Project (3) to create a new project (which brings up the project settings window shown in Figure 13-2)*

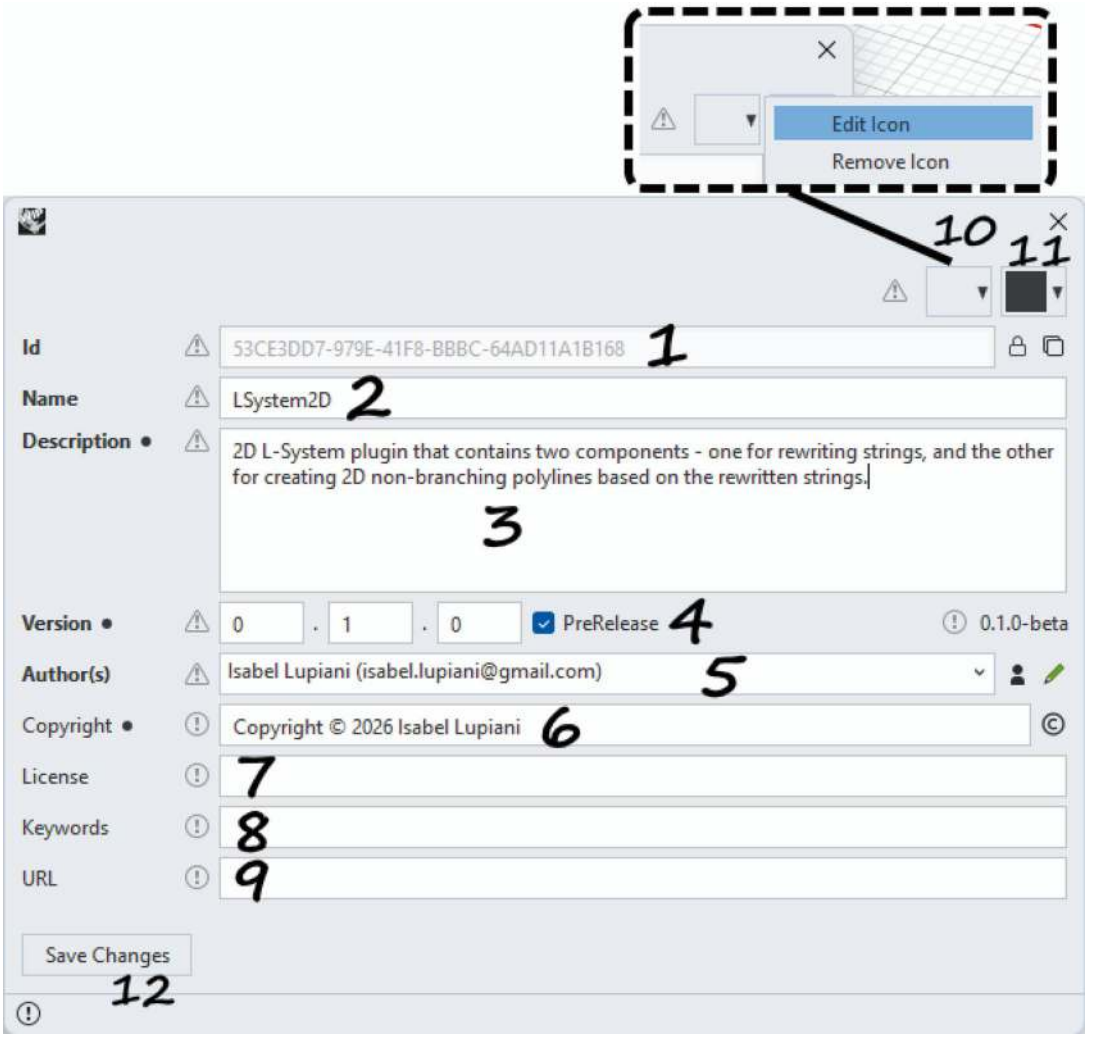


Figure 13-2. *The project settings window brought up by clicking File ➤ Create Project in ScriptEditor. Rhino automatically generates a unique Id (1) for a new project which should not be changed. You can however enter a custom Name (2), Description (3), and Version number (4, in the format of major.minor.patches; the "PreRelease" option appends "-beta"). One or more Author(s) (5) is also required, and you can optionally enter Copyright (6), License (7), Keywords (8), and a URL (9). The dropdowns 10 and 11 are for selecting a light and dark theme icon for the project. Once all the details are filled in, click Save Changes (12) to finish*

Creating SVG Files As Icons

Let's look at how to create icons in more detail. An icon is specified as a pair of SVG files—one for light theme and one for dark theme. You designate an icon for the project itself as well as for each individual Grasshopper component or Rhino command you add to the project. The interface for selecting icons is the same everywhere and consists of two dropdowns—one for light theme (left) and one for dark theme (right), as the example shown in Figure 13-2 (10, 11). To select an icon for a theme, click on the downward arrow ➤ Edit Icon to launch the Svg Editor (Figure 13-3). You can then use File ➤ Import to select an existing `*.svg` file or create one right inside the editor. The bottom-right corner of the editor shows a preview of the same icon at various sizes. Click OK to dismiss the Svg editor and save the icon once you are satisfied with it.

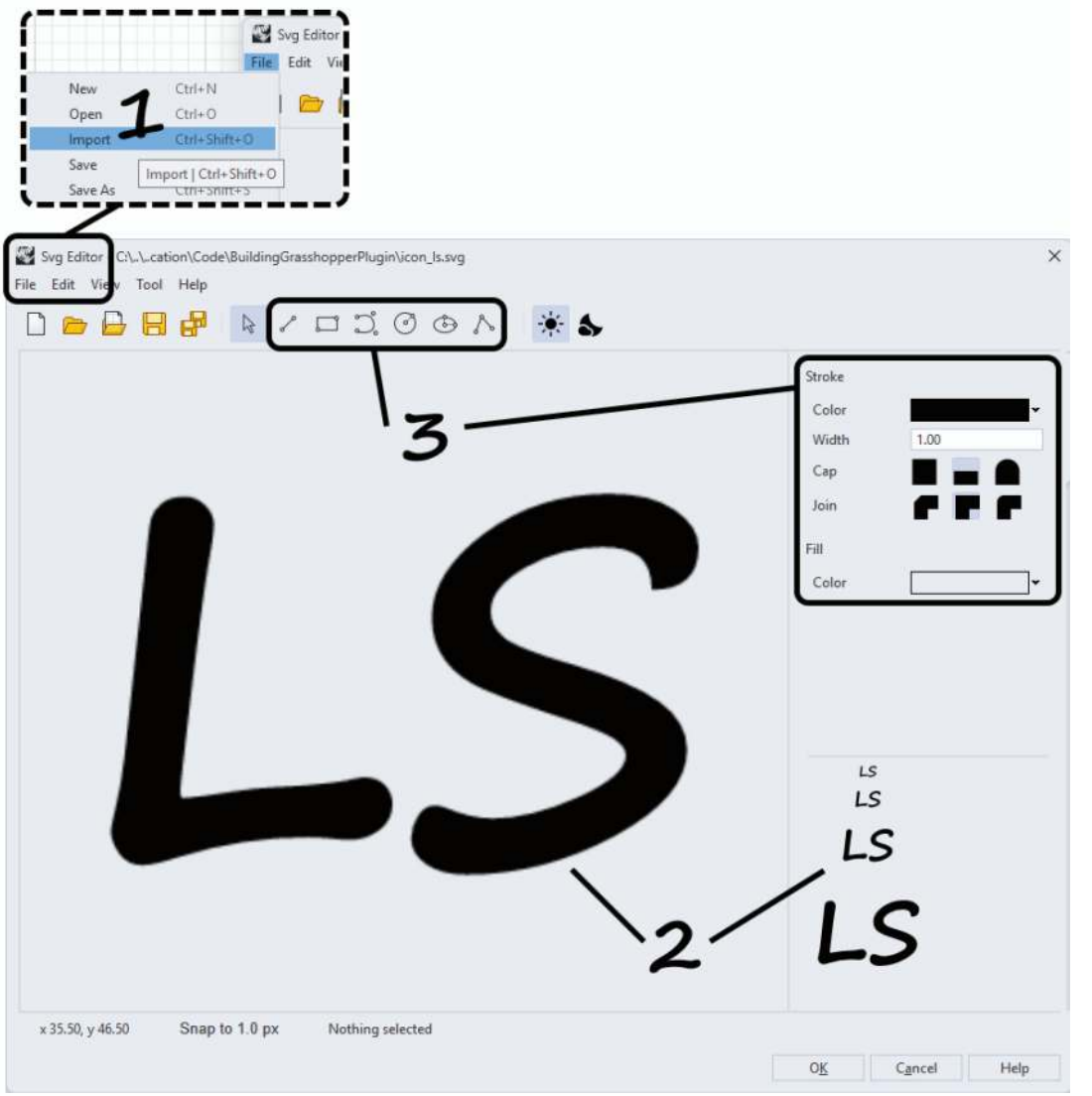


Figure 13-3. *The Svg editor window summoned by clicking an icon selection dropdown ➤ Edit Icon(10, 11 in Figure 13-2). You can either use File ➤ Import (1) to load an existing* `*.svg` *file or use the built-in tools (3) to create a new image inside the editor. Either way, the image is shown in the main editing area (2), with previews of that image at various sizes shown on the lower right*

I've created four pairs of `*.svg` files in the open-source vector graphics editor Inkscape to be used as icons for the plugins and command we'll build throughout the chapter. When creating custom icons, I recommend an image size of 24 by 24 pixels (px).

You can set the image size to 24 x 24 px in Inkscape by right-clicking a document's background ➤ Document Properties... (Figure 13-4). Under "Format", select Custom px (1) and enter 24 for both Width and Height (2). For the light theme, you'll set the Page's color to white (by right-clicking (3)) and for the dark theme black. To make the icon's background transparent, check the "Checkerboard" (4) box.

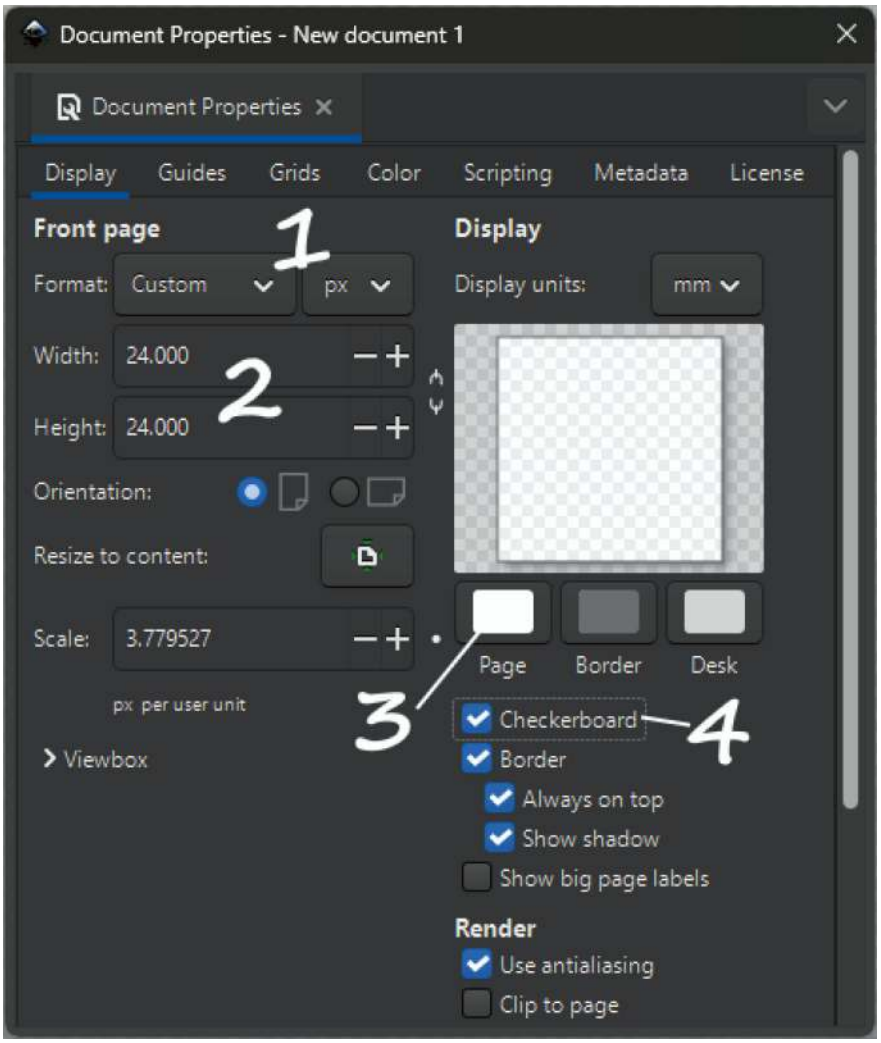


Figure 13-4. *To set a *.svg drawing's size to 24x24, right-click a document's background in the Inkscape editor ➤ Document Properties... and set Format to Custom px (1) and both Width and Height to 24.000 (2). Right-click Page (3) to set the background color to white for the light theme icon and black for the dark theme icon, and check the Checkerboard box (4) to make the background transparent*

Scripts As Individual Components

Figure 13-5 shows the L-System node graph for creating 2D continuous line drawings from Figure 11-6. As shown, it rewrites and creates polylines for a unit square. We will use Figure 13-5 to create a Grasshopper plugin. As we'll see shortly, since Figure 13-5 contains two Python 3 Script nodes, when the GH definition is added as a component under a ScriptEditor project, two GH components will be created when the project is built into a GH plugin, one component per script node.

Note You can find Figure 13-5 in `/Ch13/BuildingGrasshopperPlugin/turtle2D_with_tooltips.gh` in the downloaded source files for the chapter.

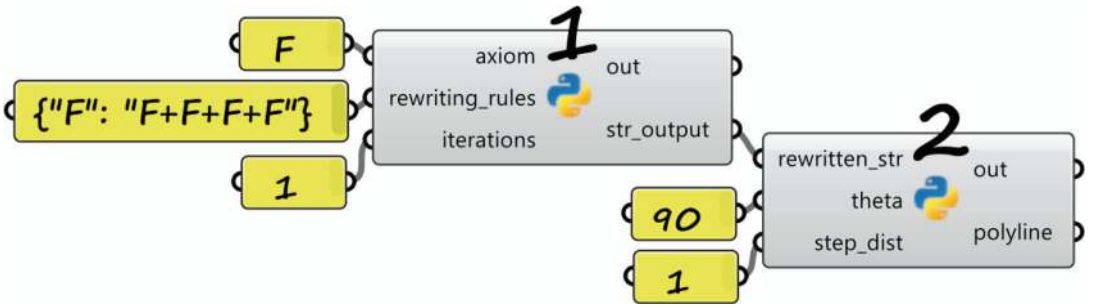


Figure 13-5. *The node graph for rewriting and generating 2D continuous line drawings from L-Systems (see `/Ch13/BuildingGrasshopperPlugin/turtle2D_with_tooltips.gh`). As shown, the grammar traces a square counterclockwise. The graph consists of two Python 3 Script nodes, (1) for rewriting a given grammar and (2) for creating polylines based on the expansion string*

Preparing the Source Grasshopper Definition

By now, you've probably noticed that built-in nodes in Grasshopper come standard with both default and human-friendly names for each input/output as well as a tooltip descripting its purpose. To make our custom GH components more user-friendly, we'll modify the source script before adding it to the ScriptEditor project.

Figure 13-6 shows how we'll augment each Python 3 Script node from Figure 13-5. When you right-click an input (or the node itself), the name that you type in at the top (1) is the default name, "Name (for humans, optional)" (2) is a human-friendly nickname, while "Tooltip (optional)" adds a description. Using the first input of Rectangular Array as an example, these three options correspond to "G," "Geometry," and the message "Base geometry" in the tooltip (4, 5, and 6 in Figure 13-6). As you'll see shortly, the icon you'll specify for the component in the ScriptEditor project will show up as the icon in the center of the node (7 in Figure 13-6).

In addition, you will right-click each input for both script nodes and check "Required," to make Grasshopper show the component as orange and display a warning to the user when any input value is missing.

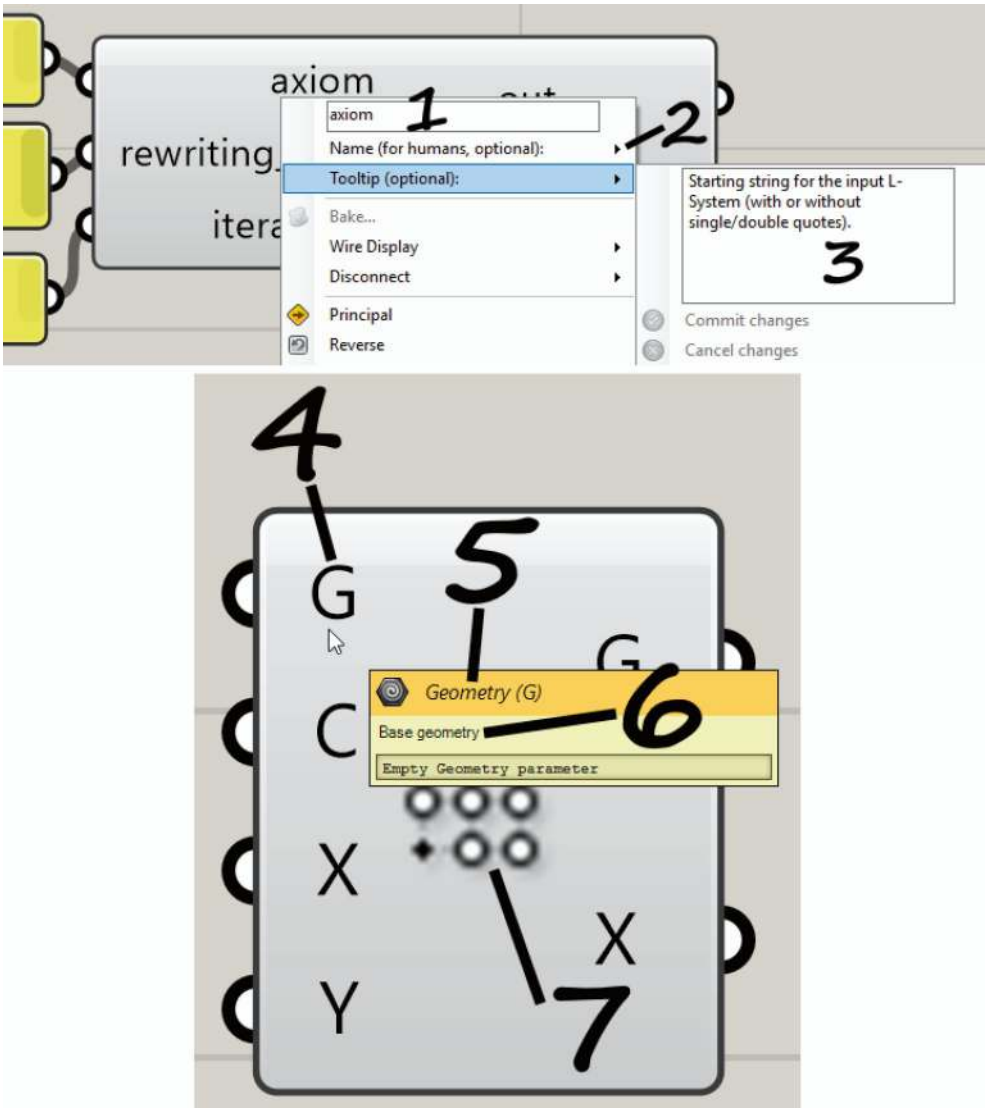


Figure 13-6. *Customizing a Python 3 Script node to make a more user-friendly component. The top shows right-clicking an input (or node) to modify its (1) default name, (2) human-friendly name, and (3) tooltip content. Using the first input of Rectangular Array as an example, "G" is the default name (4), "Geometry" the human-friendly name, and "Base geometry" (6) the tooltip message. As you'll see shortly, the icon you'll specify in ScriptEditor for the component will show up as the image in the center of the node (7)*

Note You can find Rectangular Array under Transform ➤ Array.

Table 13-1 summarizes the names and tooltips that I've set up for the two script nodes in `/Ch13/BuildingGrasshopperPlugin/turtle2D_with_tooltips.gh`. I did not create a human-friendly name for every input/output since it did not feel helpful. Ultimately, the decision is up to you. If only a default name is given for an input or output, that name is displayed regardless of what the display settings are in Grasshopper (Display ➤ Draw Icons or Draw Full Names). If both a default name and a human-friendly name are given, the default name will show for Draw Icons and the human-friendly name if Draw Full Names is selected.

Table 13-1. *Custom names and tooltips for the two Python 3 Script nodes in Figure 13-5*

Node #	Name	Name (for human, optional)	Tooltip (optional)
1	axiom	N/A	Starting string for the input L-System (with or without single/double quotes).
1	rewriting_rules	N/A	One or more rewriting rules for the input L-System formatted as a string representation of a Python dictionary, with each rule as one entry and the predecessor of the rule as the key and successor as the item, for example, {"F":"F+F+F+F"}. Each key or item must be surrounded by double quotes.
1	iterations	N/A	Number of iterations to rewrite the L-System.
1	str_output	expansion	The rewritten string for the given number of iterations.
2	rewritten_str	Expansion String	An expansion string for an L-System, ready to be interpreted as a (collection of) polyline(s).
2	theta	Step Angle	Unit angle in degrees for turn, pitch, or roll.
2	step_dist	Step Distance	Unit distance for one step forward (F or f).
2	polyline	N/A	Polyline(s) constructed based on the expansion string.

Linking the GH Definition As Individual Components

With the script nodes set up with appropriate names and tooltips, the next step is to link the GH definition to the ScriptEditor project. To do so, right-click Components/ ➤ Link Grasshopper 1 File and select `/Ch13/BuildingGrasshopperPlugin/turtle2D_with_tooltips.gh` from the chapter's downloaded source, as shown in Figure 13-7. After linking the file, you should see the script's name appear with two items titled "Default/" nested under it (2 in Figure 13-7).

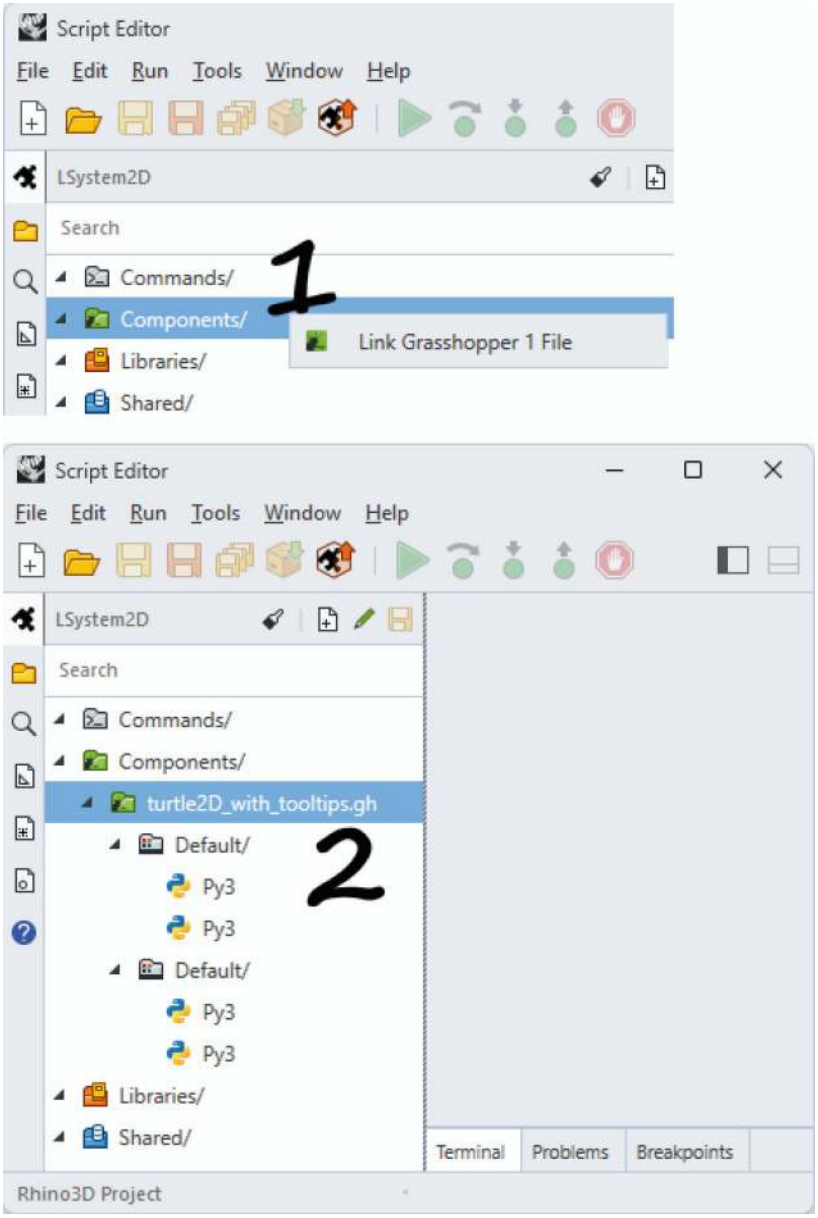


Figure 13-7. *To link a Grasshopper definition to a ScriptEditor project as the source for a Grasshopper plugin, right-click Components/ (1) and navigate to and select the desired file (in this case,* `/Ch13/BuildingGrasshopperPlugin/turtle2D_with_tooltips.gh`*). You should see the name of the file appear (2) along with two items named "Default/" nested under it*

Editing the Description of Each Component

Next, we'll edit each component's descriptions. As shown in Figure 13-8, under Components/, click the first Py3 in the list (1) followed by the Edit icon (3) button to its right to bring up the "Edit Component" window to change settings for how and where the RewriteString component will appear in the finished plugin.

Name, *NickName*, and *Description* correspond to the items shown inside the tooltip when you hover over the component. *Sub-Category* controls which panel the component will be listed under (the *Sub-Category*'s name will be shown on the black clickable tab at the bottom of that panel). *Exposure* controls where inside the panel the component goes (the option Normal (Panel Top Section) says the component will be in the visible portion of the panel under the plugin's tab, without the user having to click the black tab at the bottom of the panel to expand it).

As shown in Figure 13-8, for the RewriteString component (made from the first Python 3 Script node in Figure 13-5), we'll enter RewriteString as its *Name*, RW as its *NickNam*e, and "Rewriting a given L-System for the desired number of iterations" for its *Description*.

We'll select "Default" for *Sub-Category* and "Normal (Panel Top Section)" for *Exposure*, so the component will appear in a panel that says "Default" at the bottom and remain visible under the plugin's tab without the user having to click to expand the panel.

Note A complete copy of the ScriptEditor project based on this section is provided in `/Ch13/BuildingGrasshopperPlugin/LSystem2D.rhproj`.

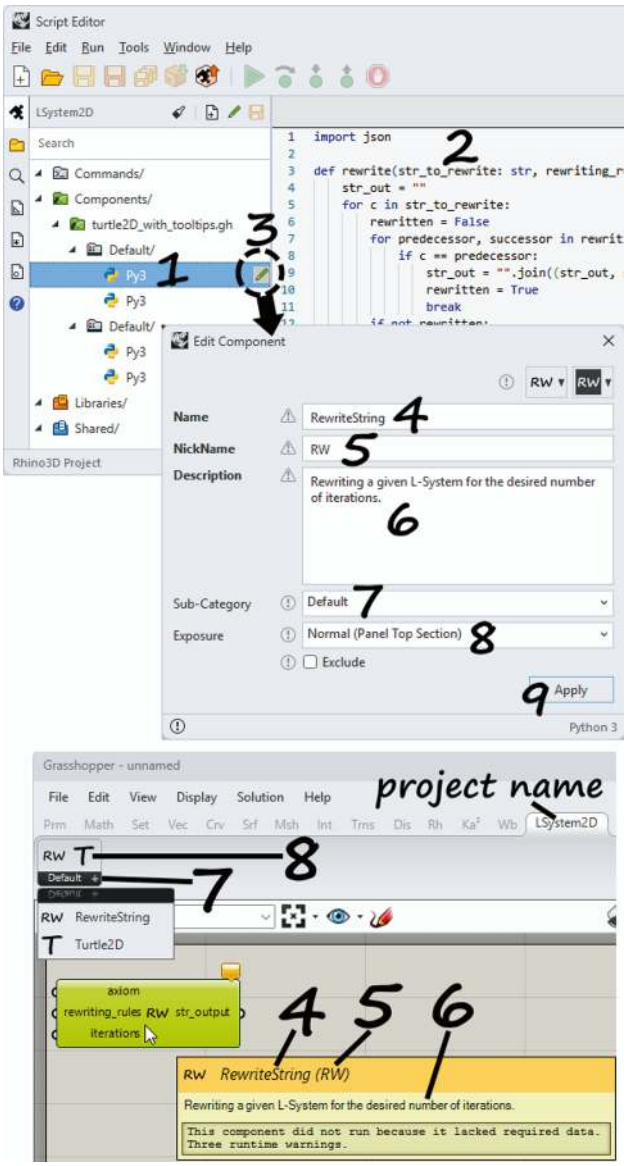


Figure 13-8. *Customizing how and where a component will appear in the finished plugin. To customize the RewriteString component, click the first Py3 (1) followed by the Edit icon (3) button to launch the Edit Component window. Enter the Name (4), NickName (5), and Description (6) to customize how the tooltip will read when you hover over the node. Sub-Category (7) controls which panel the node will go under, while Exposure (8) specifies where in the panel the component will appear (Normal means visible without the user having to click to expand the panel). Note that when you click a linked script (1) in ScriptEditor, its contents are displayed on the right (2)*

We will customize the Turtle2D component (made from the second Python 3 Script node in Figure 13-5) in a similar fashion. As shown in Figure 13-9, click on the second Py3 under Components/ followed by the Edit icon button to bring up the Edit Component window. Enter Turtle2D and T2D for its *Name* and *NickName* and "2D turtle graphics object that creates 2D polylines based on input L-System expansion strings. Allows skips but not branching." for its *Description*. We'll select Default for *Sub-Category* and Normal for *Exposure*, which match the settings for RewriteString, so both nodes get grouped under the same location inside the same panel, as shown on the bottom of Figure 13-8.

Note that the *Name* you previously entered for the ScriptEditor project will appear as the plugin's tab name in Grasshopper. For example, we entered "LSystem2D" in Figure 13-2 as the project's *Name*; therefore, the finished plugin shows "LSystem2D" on its tab once installed in Grasshopper, as shown on the bottom of Figure 13-8.

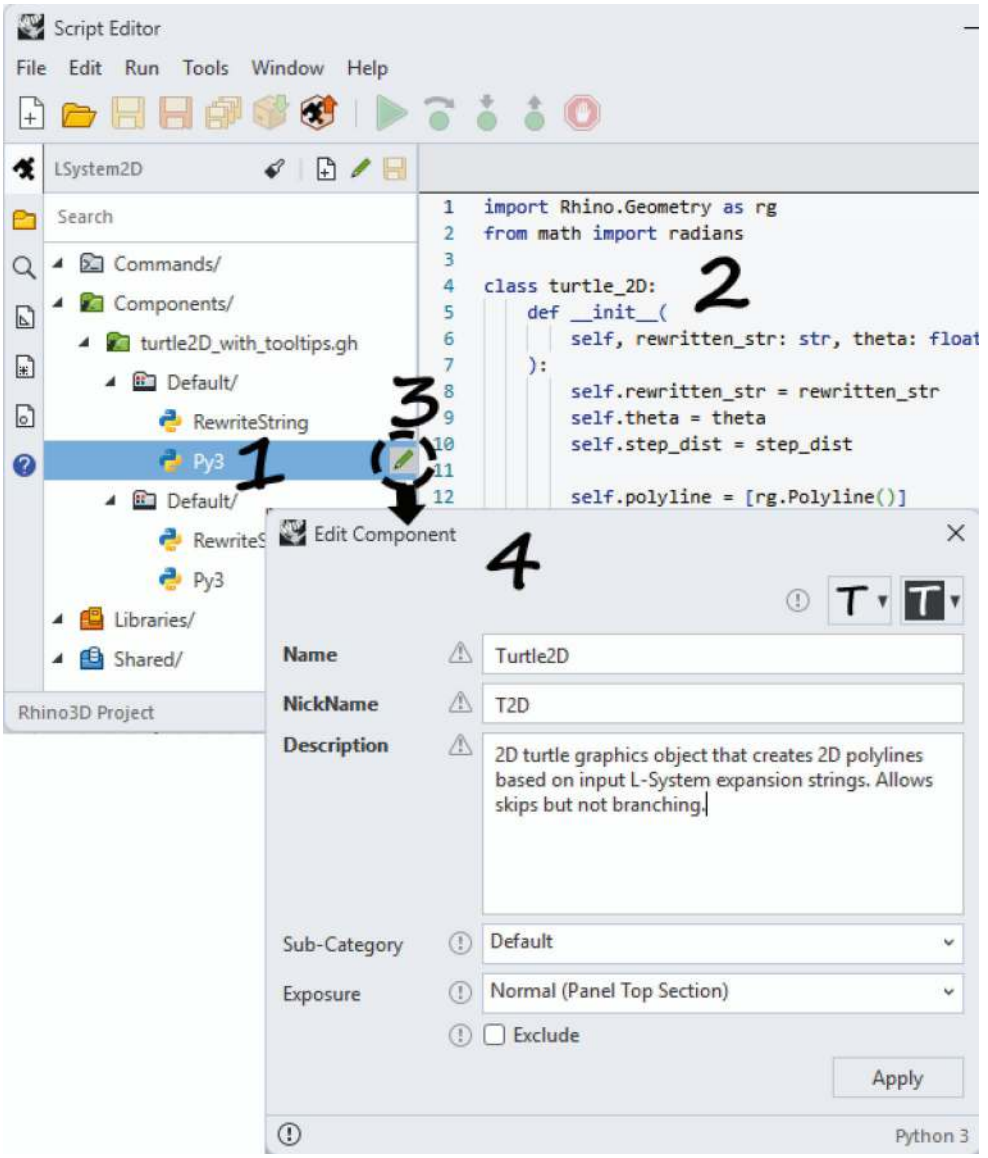


Figure 13-9. *To customize the Turtle2D component, click on the second Py3 (1) followed by the Edit icon (3) button to launch the Edit Component window (4) and enter the various fields as described in the text. You can verify that the correct script is selected by checking its contents on the right (2)*

Creating Installers and Packages

With both components configured, we are ready to build the project and generate an installer (`*.gha` file) for the Grasshopper plugin along with a Yak package (`*.yak` file) for distributing it. Yak is the name of Rhino's Package Manager that lets you discover and install plugins right inside Rhino. It simplifies the distribution process for developers as you can publish and update plugins by simply uploading files to the Yak server right from the command line.

To initiate the build process, click the "Publish active project" button at the top of ScriptEditor (1 in Figure 13-10) to launch the Publish Package window shown on the bottom of Figure 13-10. Most of the parameters in this window should already be pre-populated based on your project settings. For "Server," select "McNeel Yak Server (yak.rhino3d.com)" (2). For "Target" (3), select a Rhino version based on the operating system and *minimum* version of Rhino you are targeting. The dropdown includes your current version of Rhino and up to one major release prior or later, for both your operating system and the same version without OS designation. For example, Rhino 3D (8.* - Windows) and Rhino 3D (8.*) mean "any version of Rhino 8 on Windows" and "any version of Rhino 8, on any OS", respectively. Note that this book is written against Rhino 9 WIP. At the time of writing, even though the dropdown includes Rhino 3D (9.* - Windows) and Rhino 3D (9.*) as available options, ScriptEditor errors out if you attempt to build with those targets—if you are using Rhino 9 WIP, be sure to build only up to Rhino 8+ but not Rhino 9 (I suggest building against Rhino 3D (8.*) for your OS). The provided files are built against Rhino 3D (8.* - Windows).

The next step is to select a Path (4) to save the build. This path is relative to the directory the project is saved in. For example, build/LSystem2D is actually

```
<your project dir>/build/LSystem2D
```

Once you've entered and reviewed all settings, click "Save Changes" (5) so the current set of settings are saved with your project and auto loaded next time you click "Publish active project". If saved successfully, the bottom of the window will turn green and display the message "Project Saved" (6).

All that's left is to build. Click "Build Package" (7) to kick off the process. You should see a series of messages running through the bottom of the window. If the build is successful, the bottom will turn green and display the message "Package Created Successfully" (8). Otherwise, it will turn red—click "Log" to find out what's wrong.

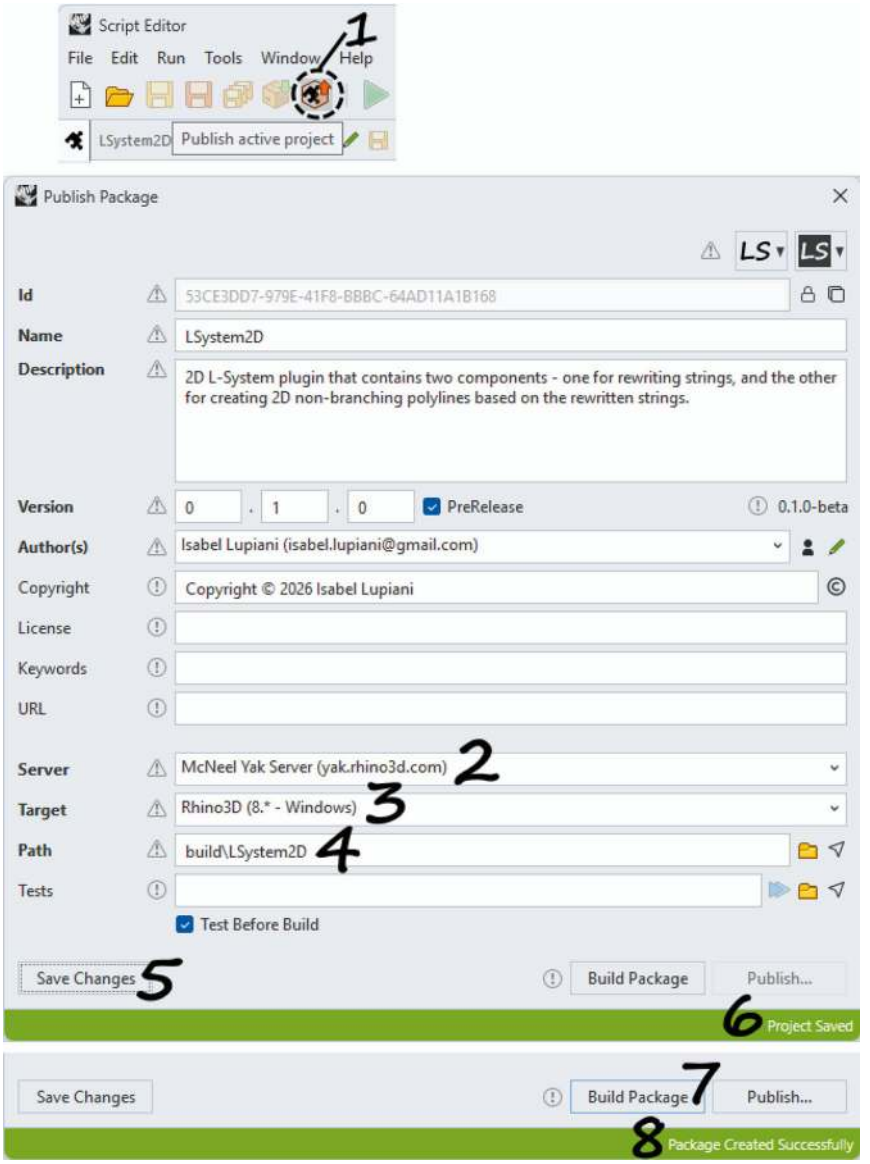


Figure 13-10. *To build the project, click the (1) "Publish active project" button to bring up the "Publish Package" window (bottom) and fill in the build parameters (most of which should be pre-populated based on your project settings). For Server (2), select "McNeel Yak Server". For Target (3), select the OS and minimal Rhino version combo you are targeting (for Rhino 9 WIP, select Rhino 3D (8.*)). For Path (4), enter a relative path from the project's directory. Click Save Changes (5) to save the current set of build settings with the project (if save is successful, the bottom will turn green (6) and say "Project Saved"). Finally, click "Build Packages" to build (7). The bottom of the window should turn green upon a successful build and say "Package Created Successfully" (8)*

Once the build is complete, navigate to the folder that you've designated as the Path. You'll find that Rhino has automatically created a folder under it with the name of the target OS. For example, I selected Rhino 3D (8.* - Windows) for my build, with `build/LSystem2D` as Path, and Rhino saved the files to `build/LSystem2D/rh8-win` (1 in Figure 13-11). For a project containing only Grasshopper components, you should see a list of five items as shown in Figure 13-11. The "src" folder (1) contains a Visual Studio solution (`*.sln`) with various C Sharp (`*.cs`) files for the plugin if you wish to modify it further at the source level. `icon.png` (2) is an export of the plugin's light theme icon as a PNG image. The `*.gha` file (3) is the Grasshopper installer for the plugin. The `*.yak` file (4) is the package for publishing the plugin on the Rhino Package Manager server. `manifest.yml` (5) is metadata for the Yak package which you could optionally edit by hand and use when building with the Yak command line tool.

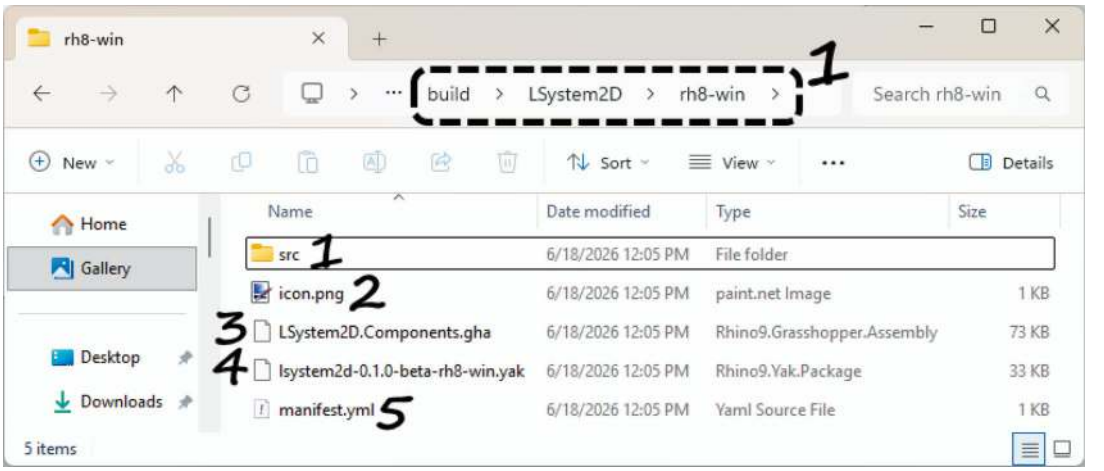


Figure 13-11. *Rhino automatically creates a folder named after the target Rhino version and OS under the Path specified for the build (1). For a GH plugin, you should find five items in this folder: (1) a src folder containing CSharp source for the plugin, (2) an exported PNG of the plugin icon, (3) a GH installer (`.gha`) for the plugin, (4) a package (`.yak`) for publishing the plugin on the Rhino Package Manager server, and (5) metadata (`manifest.yml`) for the package*

Installing the Grasshopper Plugin You've Just Created

The best way to verify your plugin works as intended is to install it and try it out. There are two ways to install the plugin. The first is to drag and drop the `*.gha` file onto the Grasshopper canvas. The second is to copy and paste the `*.gha` file to the folder where

GH stores the plugins—to locate this folder, open Grasshopper and go to File ➤ Special Folders ➤ Components Folder (Figure 13-12) to open the folder in Files Explorer on Windows (or Finder on Mac). Close both Rhino and Grasshopper and copy and paste the *.gha file to this folder. Relaunch Rhino and Grasshopper; you should see the plugin's tab appear (in this case, LSystem2D).

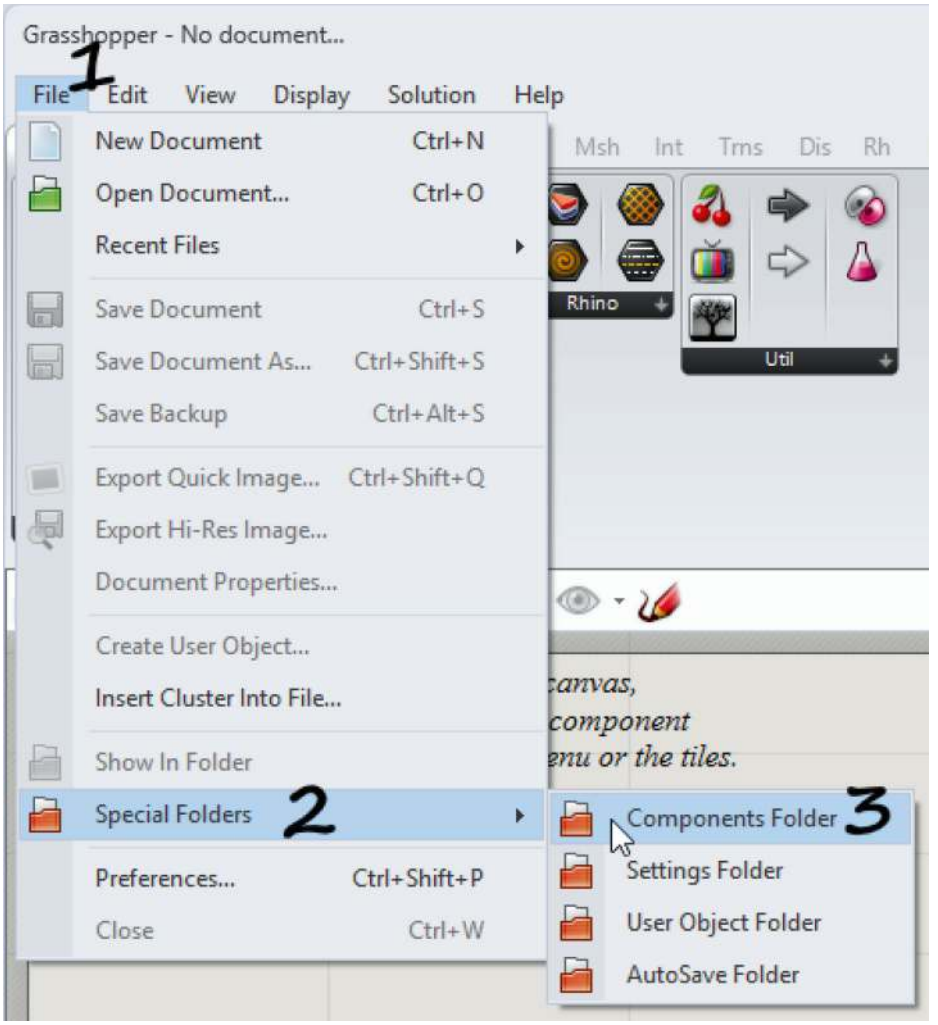


Figure 13-12. *You can find the folder where Grasshopper stores the plugins via File (1) ➤ Special Folders (2) ➤ Components Folder (3), which will open the folder in Files Explorer on Windows or Finder on Mac*

Verifying the Grasshopper Plugin

We will verify the plugin works correctly by creating a definition to generate the modified Koch curve from Chapter 11: L-System. As shown in Figure 13-13, from the "LSystem2D" tab (1), we'll drag down a RW node (2) to the canvas and set it up with the following inputs:

- *axiom*: F (in a Panel)
- rewriting_rules: {"F":"F+F--F+F"} (in a Panel)
- *iterations*: 4 (using a Number Slider)

We then drag a T node (3) to the canvas and set it up with the following inputs:

- rewritten_str: Output of RW
- *theta*: 75 (Panel or Number Slider)
- step_dist: 1 (Panel or Number Slider)

If all goes well, the T node should output the polyline shown in Figure 13-14, which is the same modified Koch curve with four iterations we've worked out in Chapter 11. Notice that both the RW and T nodes are grouped in the panel named "Default" as we configured (remember in Figures 13-8 and 13-9, we set Sub-Category to Default in the Edit Component window for both components?).

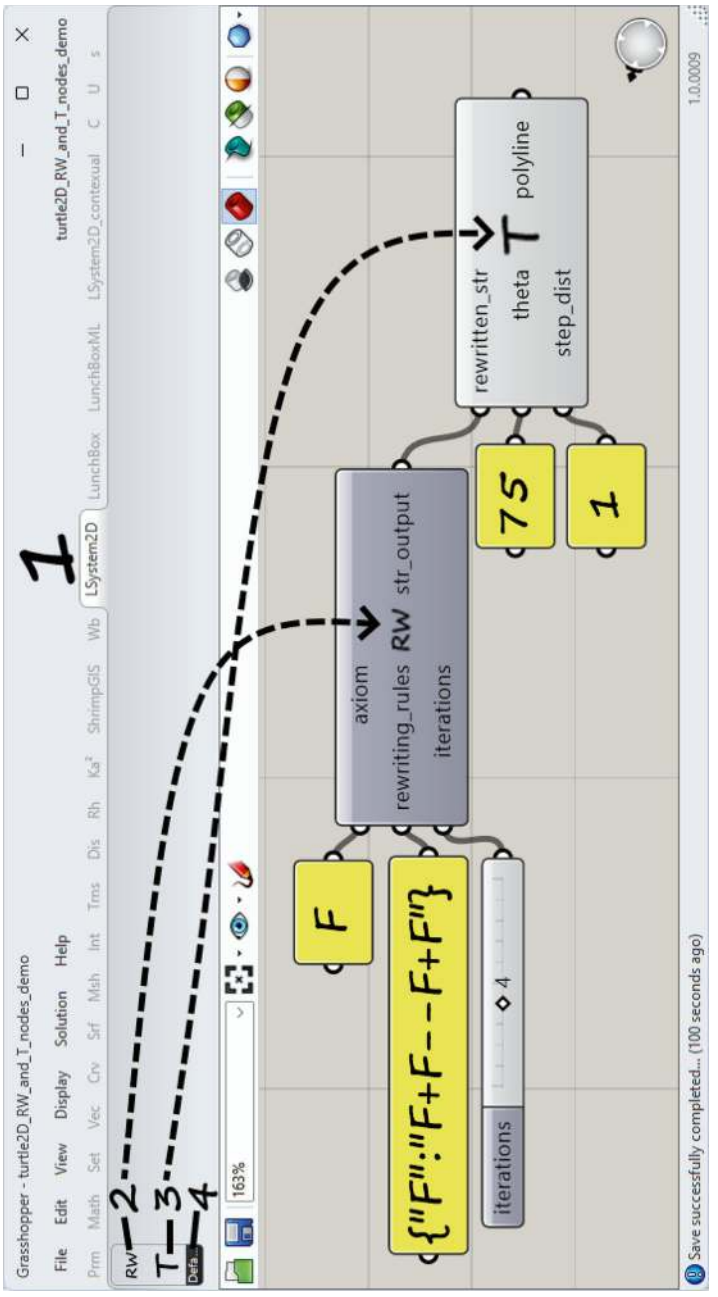


Figure 13-13. *To test the plugin, we recreate the node graph from Chapter 11 (Figure 11-6) with the RW (2) and T (3) nodes under the LSystem2D tab (1) to generate the modified Koch curve. Notice that both RW and T are grouped under a panel named "Default" (4) since we've set Sub-Category to Default in Edit Component for both nodes (Figures 13-8 and 13-9)*

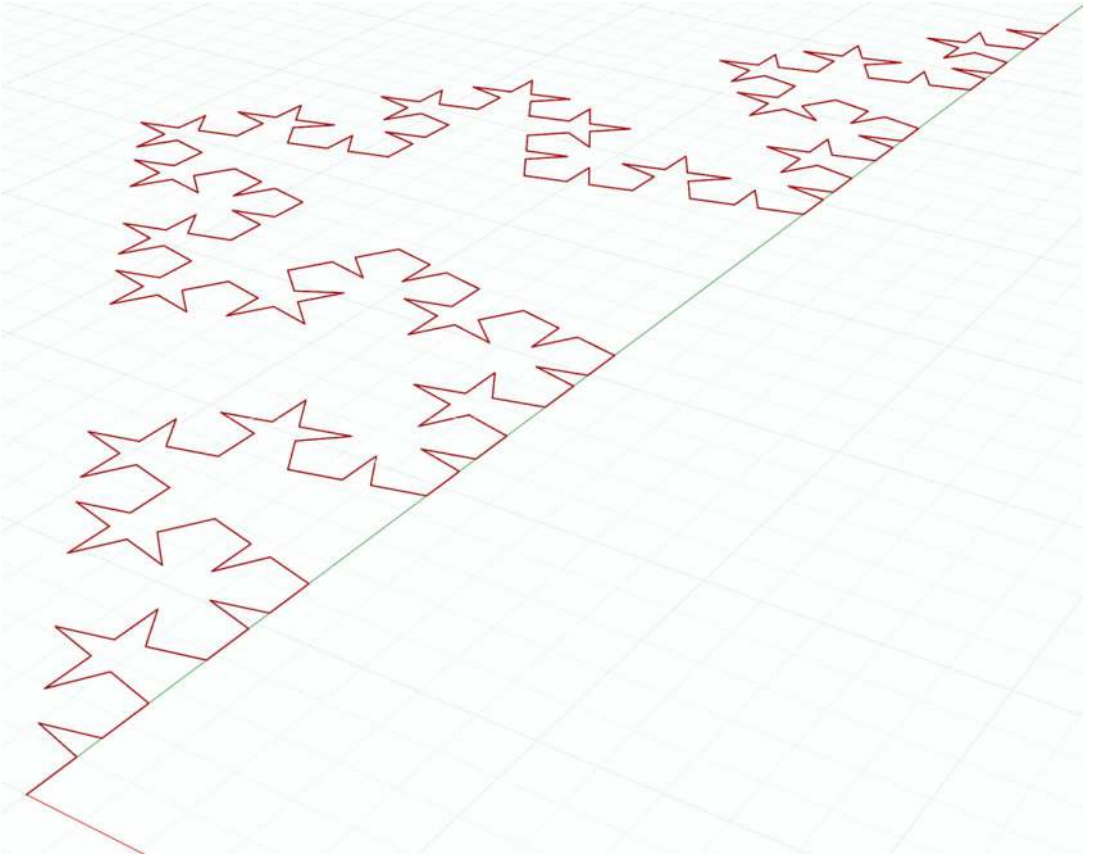


Figure 13-14. *The modified Koch curve (four iterations) generated by the node graph in Figure 13-13, using the RW and T nodes from our plugin*

Combining Multiple Script Nodes into One Component

In this section, we'll discuss a different way of building a plugin from a Grasshopper definition—by combining multiple Python 3 Script nodes into a single Grasshopper component for the plugin.

Adding Contextual Inputs and Output(s)

To tell ScriptEditor to combine multiple Python 3 Script nodes, we'll insert contextual inputs of the appropriate data types between the original input values and the Script nodes and link the final output of the script to a contextual output. The contextual inputs/output(s) will become the inputs/output(s) of the Grasshopper component that ScriptEditor will build.

"Contextual inputs" are any of the "Get <data type>" nodes under Params ➤ Util as shown in Figure 13-15, such as Get Boolean, Get Number, Get Geometry, etc. You will choose a contextual input that is the same type as your original input. For example, 5 is Get Integer, a surface is Get Geometry, and 8.5 is Get Number, and so on. There are two "contextual outputs," which are Params ➤ Util ➤ Context Print for text (strings), and Context Bake for all other data types.

We will modify the L-System node graph as shown in Figure 13-16. For RewriteString, we will insert Get String (1) before axiom, Get String (2) before rewriting_rules, and Get Integer (3) before iterations. For Turtle2D (second Python 3 Script node), we'll insert Get Number (4) before theta and Get Number (5) before step_dist. We'll also link Turtle2D's polyline output to Context Bake (6). Note that we're using Get Integer for RewriteString's iterations input since that must be whole numbers and Get Number for Turtle2D's theta and step_dist since those could be decimal numbers.

Note You can find the node graph with all the modifications from this section in `/Ch13/BuildingGrasshopperPlugin/turtle2D_contextual_raw.gh` in this chapter's downloaded source files.

Note that the original inputs in the source GH definition become the default values for those inputs in the Grasshopper component that will be generated for the plugin by ScriptEditor.

To create a plugin from the modified GH definition, you'll simply create a new project in ScriptEditor (File ➤ Create Project) and add the modified GH definition from this section by right-clicking Components/ ➤ Link Grasshopper 1 File. When built, the plugin will contain the single GH component depicted in Figure 13-17, with five inputs and one output, matching the number of contextual inputs and output we've set up in Figure 13-16.

The right side of Figure 13-17 (3) shows what happens if you only set up contextual inputs but forget to add contextual output(s). The generated Grasshopper component will have no output!

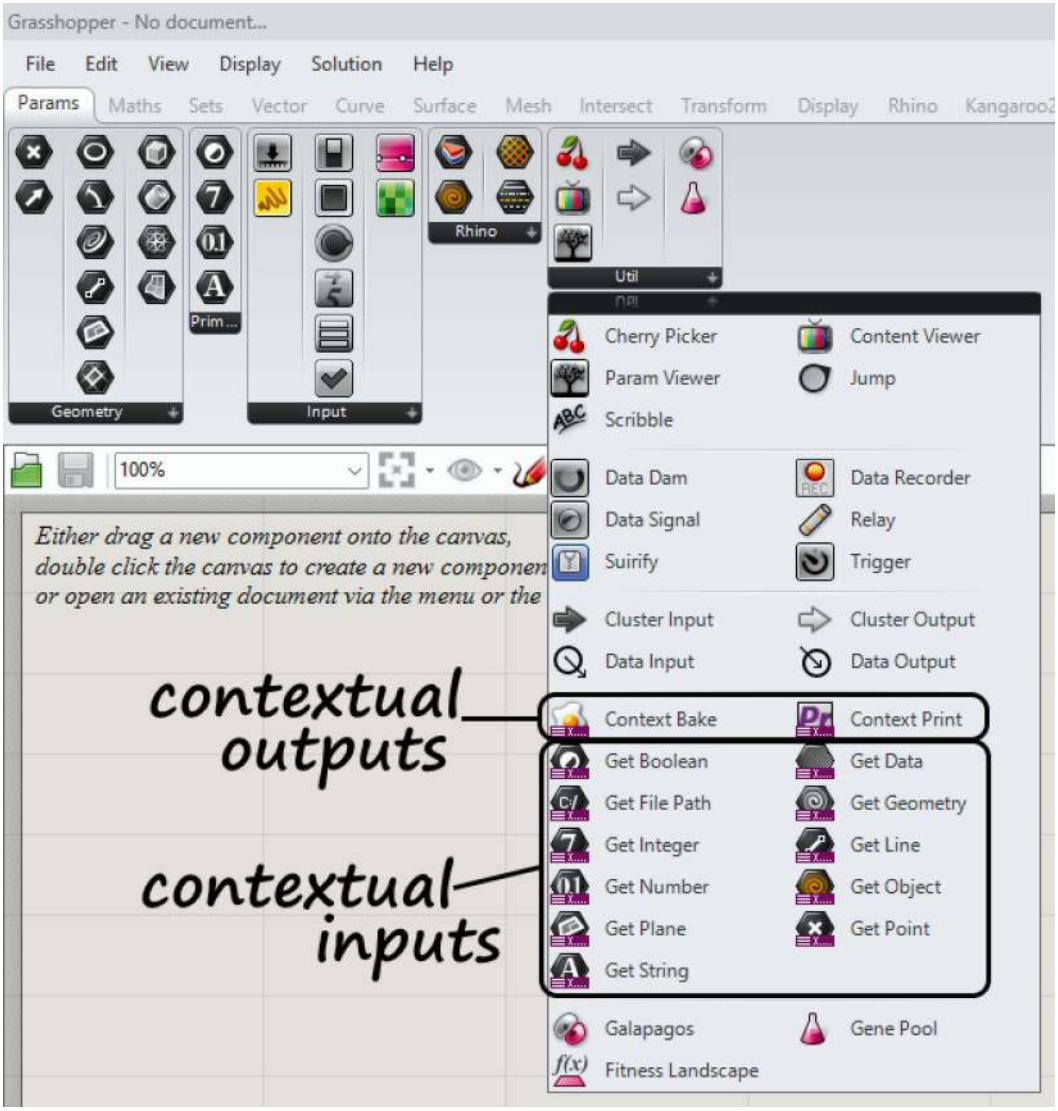


Figure 13-15. *You can find the contextual inputs/outputs under Params ➤ Util. The contextual inputs include all the "Get <data type>" nodes, such as Get Boolean, Get Geometry, etc. The contextual outputs include Context Print (for text) and Context Bake (for everything else)*

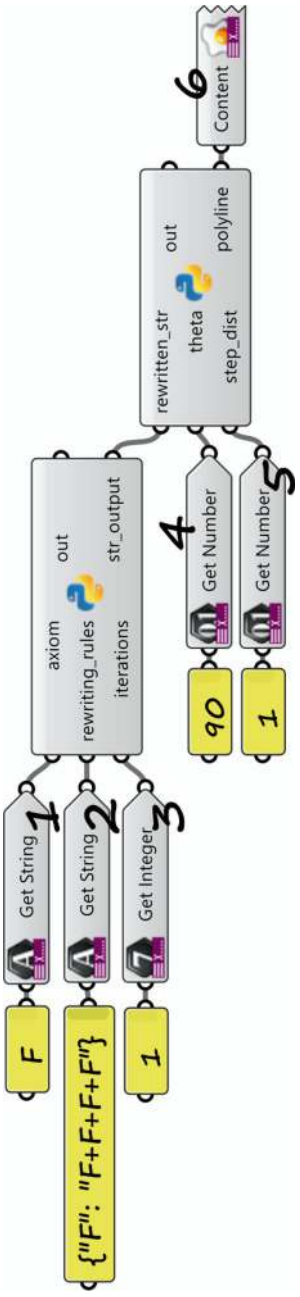


Figure 13-16. *Modifying the L-System node graph (from Figure 13-5) by inserting the appropriate contextual inputs and output. As shown, for RewriteString, we use Get String (1) for axiom, Get String (2) for rewriting_rules, and Get Integer (3) for iterations. For Turtle2D, we use Get Number (4) for theta and Get Number (5) for step_dist and link its polyline output to Context Bake (6)*

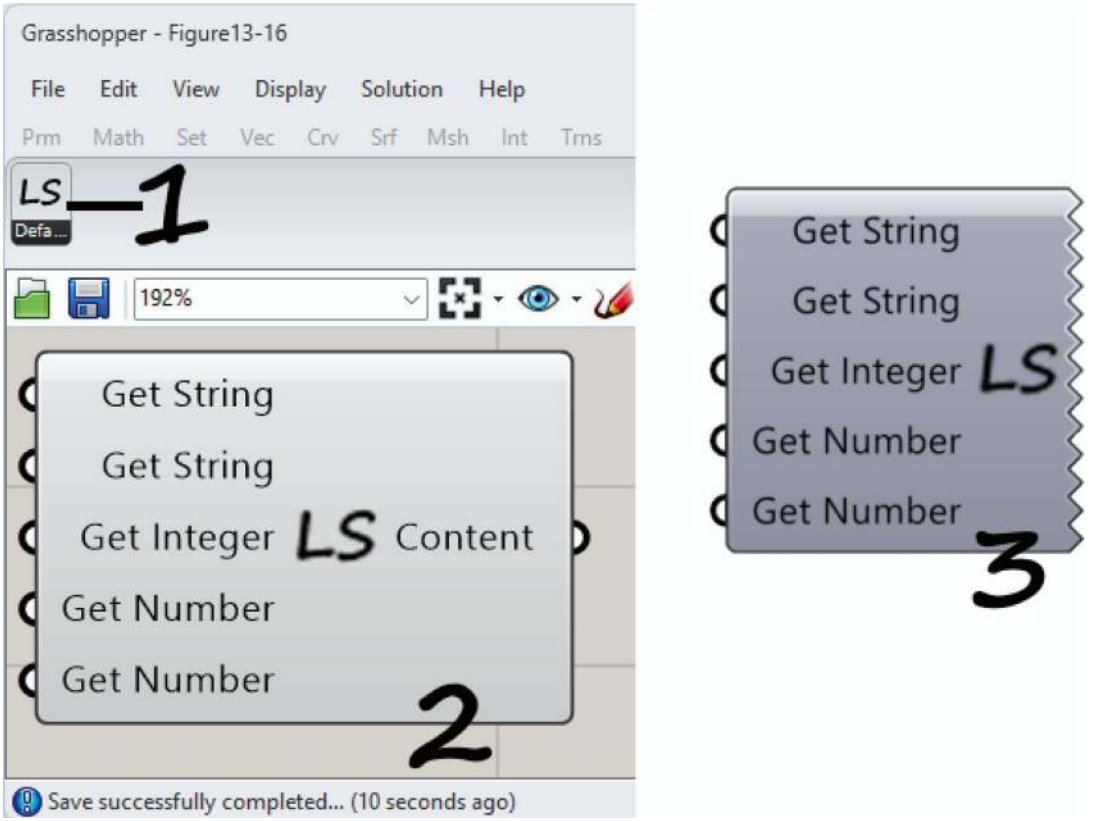


Figure 13-17. *The plugin created from the GH definition in Figure 13-16. (1) The plugin will now contain only a single GH component, which when dragged down to the canvas (2) is a node with five inputs and one output, matching the contextual inputs/output we've set up in Figure 13-16. Note that if you've forgotten to add the contextual output(s), the generated GH component will have no output (3)*

Customizing Contextual Inputs and Output(s)

While we've combined the Python 3 Script nodes into a single component (Figure 13-17), the inputs and outputs are not very descriptive, which we'll remedy in this section. For each contextual input/output, we'll right-click and modify its Name (1), Prompt... (2), and optionally the Minimum... and Maximum... (3), as shown in Figure 13-18. The values we'll set for all the contextual nodes are summarized in Table 13-2.

The Name (1) that you enter here will become the name of that input (or output) in the final Grasshopper component in the plugin as shown at the bottom of Figure 13-18. The Prompt... (2) will become the message displayed at the top of the tooltip. If Minimum... and Maximum... (3) are entered, they will be shown as a range in the tooltip. For example, for the "iterations" input, we set a Minimum... of 1 and Maximum... of 100, which are shown as "Range: 1 to 100" in the tooltip.

After all the contextual inputs and output have been edited based on Table 13-2, the node graph will look like Figure 13-19.

Note You can find a copy of Figure 13-19 in the file `/Ch13/BuildingGrasshopperPlugin/turtle2D_contextual.gh`.

A sample ScriptEditor project based on Figure 13-19 is provided in `/Ch13/BuildingGrasshopperPlugin/LSystem2D_contextual.rhproj`.

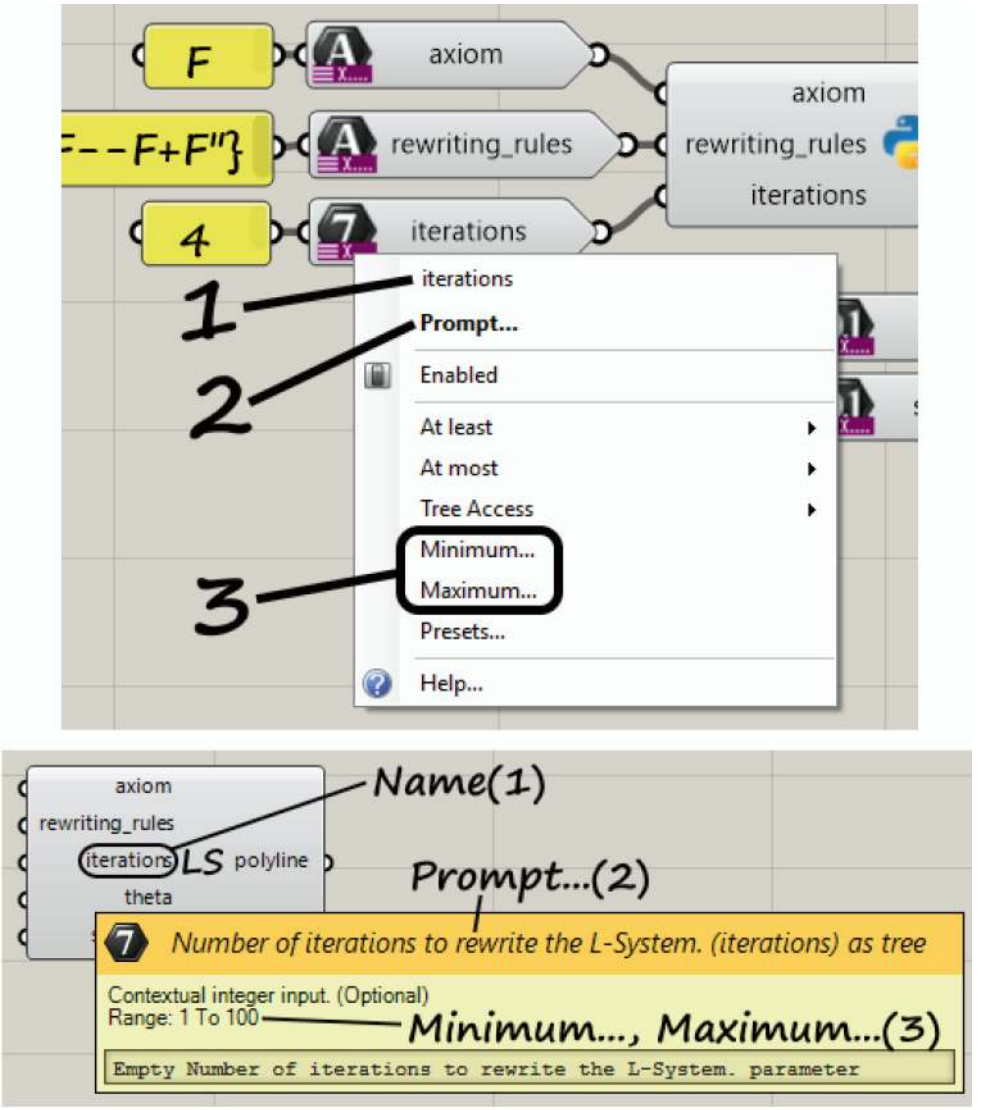


Figure 13-18. *You can right-click a contextual input or output to customize it by giving it a descriptive (1) Name, (2) Prompt... (tooltip message), and an optional range for its value (3). As shown, the "iterations" input is edited with Minimum... = 1 and Maximum... = 100, which shows up as "Range: 1 To 100" in its tooltip in the generated GH component*

Table 13-2. *Parameter values for customizing the contextual inputs/output in Figure 13-16*

# in Figure 13-16	Type	Name	Minimum...	Maximum...	Prompt...
1	Get String	axiom	N/A	N/A	Starting string for the input L-System (with or without single/ double quotes).
2	Get String	rewriting_rules	N/A	N/A	One or more rewriting rules for the input L-System formatted as a string representation of a Python dictionary, with each rule as one entry and the predecessor of the rule as the key and successor as the item, for example, {"F":"F+F+F+F"}. Each key or item must be surrounded by double quotes.
3	Get Integer	iterations	1	100	Number of iterations to rewrite the L-System.
4	Get Number	theta	5	120	Unit angle in degrees for turn, pitch, or roll.
5	Get Number	step_dist	1	100	Unit distance for one step forward (F or f).
6	Context Bake	polyline	N/A	N/A	N/A

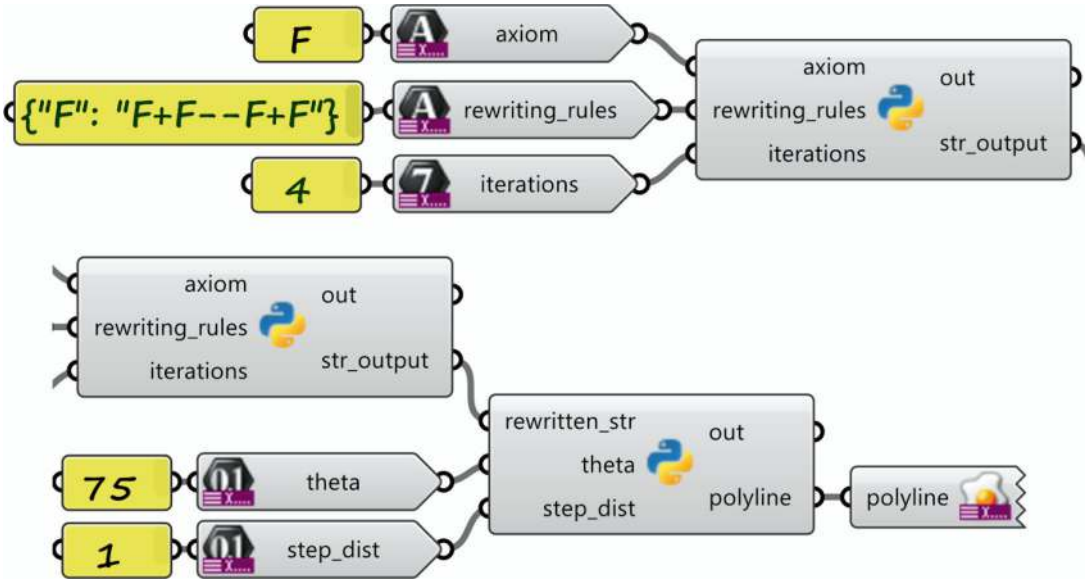


Figure 13-19. *The node graph from Figure 13-16 with all contextual inputs and output customized with descriptive names (as summarized in Table 13-2)*

Recreating the Plugin with Descriptive Inputs/Output

Once you've finished customizing the contextual inputs and output, you can update the ScriptEditor project by right-clicking Components/ ➤ Relink and selecting the modified Grasshopper definition. If you build and install the plugin again, you should see that the component has been updated with descriptive inputs/output, as shown in Figure 13-20.

Figure 13-21 shows the component set up with one of the L-System fractals from Chapter 11: L-System (Lupiani, 2025; Prusinkiewicz et al., 1990). You can find the original in Figure 11-8 in the section titled "2D Continuous Line Variations." The inputs are as the following:

- *axiom*: F-F-F-F (Panel)
- rewriting_rules: {"F": "FF-F+F-F-FF"} (Panel)
- *iterations*: 3 (Number Slider)
- *theta*: 90 (Panel)
- step_dist: 1 (Panel)

The produced polyline is shown on the bottom left of Figure 13-21.

Note You can find the file containing the GH definition from Figure 13-21 in `/Ch13/BuildingGrasshopperPlugin/ turtle2D_contexual_demo_fractal.gh`.

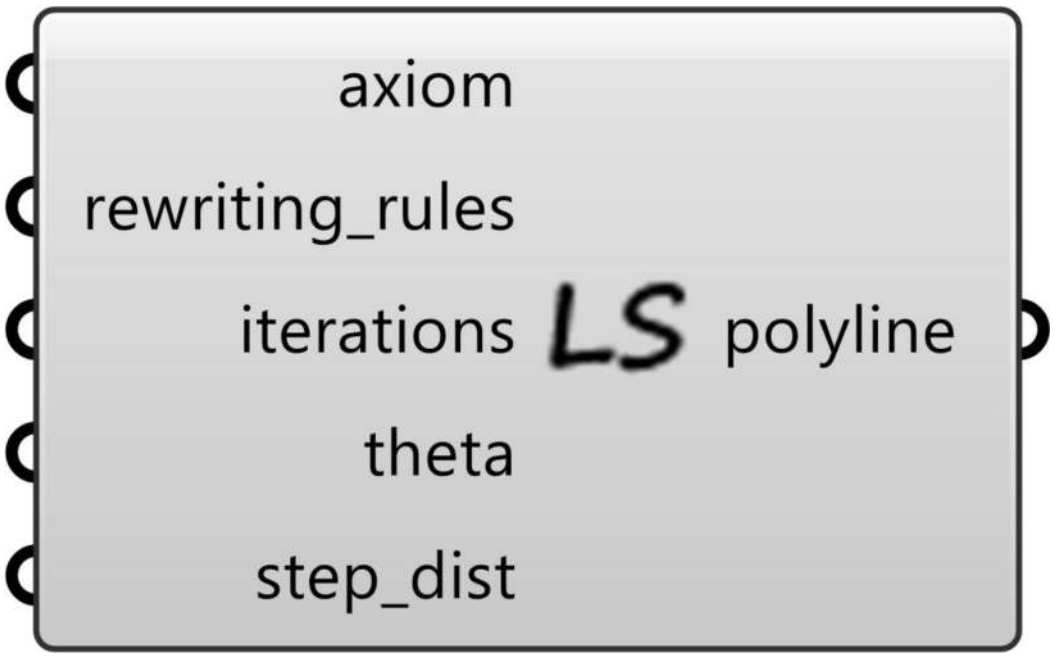


Figure 13-20. *The GH component generated from the GH definition in Figure 13-19 with customized contextual inputs/output*

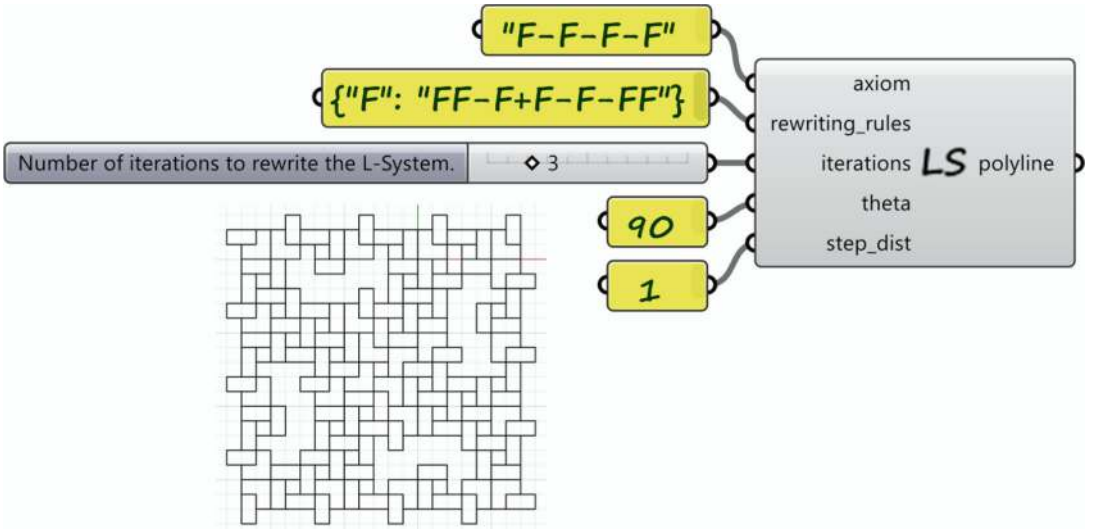


Figure 13-21. *Testing the LSystem2D component with a 2D fractal, with axiom = "F-F-F-F", rewriting_rules = {"F": "FF-F+F-F-FF"}, iterations = 3, theta = 90, and step_dist = 1 (Lupiani, 2025; Prusinkiewicz et al., 1990). Bottom left shows the produced polyline output*

Pushing to the Rhino Package Server (Yak)

Publishing your plugin with the Rhino Package Manager is easy—all it takes is to push (upload) the `*.yak` file built from your ScriptEditor project to the Rhino Package Server. In this section, we'll push to the *test server* as practice, which is a server that is wiped nightly. The entire process will be carried out in a terminal. If you are a Windows user, Search for "Command Prompt"; right-click ➤ Run as Administrator.

As shown in Figure 13-22, in the terminal, enter the following command at the prompt:

```
"<dir of Yak.exe>" login
```

For example, on my Windows 11 system, `Yak.exe` is found at

```
C:\Program Files\Rhino 9 WIP\System\Yak.exe
```

Therefore, I will type the following at the prompt and hit enter:

```
"C:\Program Files\Rhino 9 WIP\System\Yak.exe" login
```

Note that the path to Yak.exe must be enclosed in double quotes on Windows. The login command will trigger the OS to open a browser tab with the login page to Yak, which looks like the bottom of Figure 13-22. Go ahead and login with your Rhino account credentials. If the login is successful, you'll see a message like the following back at the terminal:

```
"Login is successful. Token stored in <app data dir>\yak.yml. Logged in as
<your Rhino username> on https://yak.rhino3d.com/."
```

Note that the login token is valid for 30 days. So Yak will ask you to login again periodically.

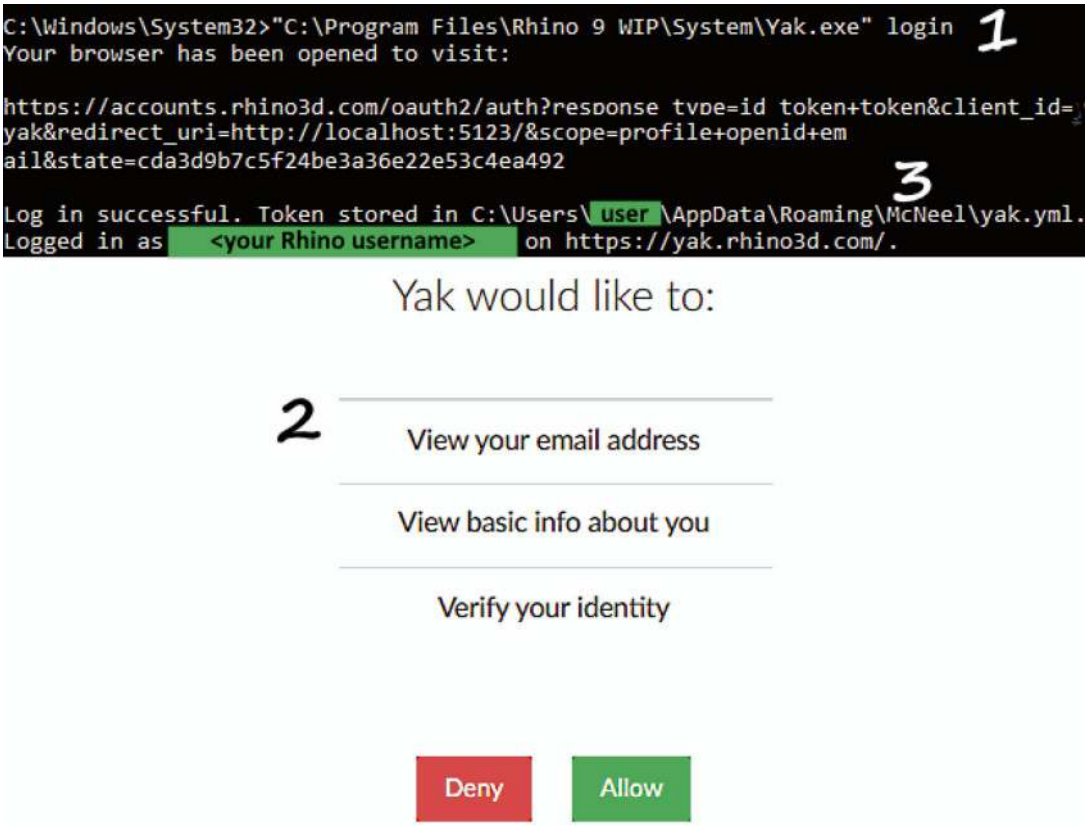


Figure 13-22. *Logging onto the Yak Package Server. At a terminal (e.g., Command Prompt ➤ Run as Administrator on Windows), (1) type the path to Yak.exe enclosed in double quotes followed by login, and press Enter. This will trigger your OS to open a browser tab with the (2) Yak login page. Go ahead and login with your Rhino account credentials. If successful, you should see a message similar to (3) telling you a token is created successfully*

Next, switch to the directory where the *.yak file resides, by entering the following at the prompt:

```
cd <path to *.yak file>
```

Pushing to the Yak Test Server

We'll push the *.yak file to the test server first as practice, by entering the following command at the next prompt:

```
"C:\Program Files\Rhino 9 WIP\System\Yak.exe" push --source https://test.
yak.rhino3d.com ./<yak filename>
```

For example, to push the file lsystem2d-0.1.0-beta-rh8-win.yak inside the current directory, you'll enter

```
"C:\Program Files\Rhino 9 WIP\System\Yak.exe" push --source https://test.
yak.rhino3d.com ./lsystem2d-0.1.0-beta-rh8-win.yak
```

If the push is successful, you'll see a message similar to the following:

```
Successfully published LSystem2D (0.1.0-beta) to https://test.yak.
rhino3d.com/

Details
=======

name:          LSystem2D
version:       0.1.0-beta (prerelease)
rhino version: rh8
platform:      win
size:          32.59 KB
```

You can double-check that the package is indeed on the test server by doing a search. For example, to look up the LSystem2D package, you'll enter the following command at the prompt:

```
"C:\Program Files\Rhino 9 WIP\System\Yak.exe" search --source https://test.
yak.rhino3d.com --all --prerelease LSystem2D
```

If found, the terminal will echo back the following message:

```
LSystem2D (0.1.0-beta)
```

Pushing (for Real) to the Yak Package Server

To push the *.yak file to the Yak Package Server, from the directory with the *.yak file, you'll enter the following command at the prompt:

```
"C:\Program Files\Rhino 9 WIP\System\Yak.exe" push ./lsystem2d-0.1.0-beta-rh8-win.yak
```

Hosting on Food4Rhino

To host a Rhino or Grasshopper plugin on Food4Rhino, login to your account on www.food4rhino.com, and navigate to My Content, under which you'll find the following options:

- "Add App", which lets you upload the files for the plugin directly
- "Add Resource", which lets you upload useful files other than plugins such as scripts and tutorials
- "Add App from Package Manager", which will let you link a package on the Yak Package Server directly to Food4Rhino.

Building a Rhino Command

Now that you are familiar with building Grasshopper plugins, in this section, we'll shift gears and build a Rhino command. With ScriptEditor, you can build both Grasshopper plugins and Rhino commands from GH definitions containing Python 3 Script nodes in a largely similar process, except in the case of Rhino commands, you must add contextual inputs and outputs to the GH definition if the command is to be interactive and prompt users for input values.

Converting a Node Graph to a Python 3 Script Node

Recall that in Figure 2-3, we implemented a node graph to generate a series of nested arc walls with fixed angle increments, as reproduced in Figure 13-23. In this section, we'll convert this graph to a Python 3 Script node, using the inputs and output summarized in Table 13-3.

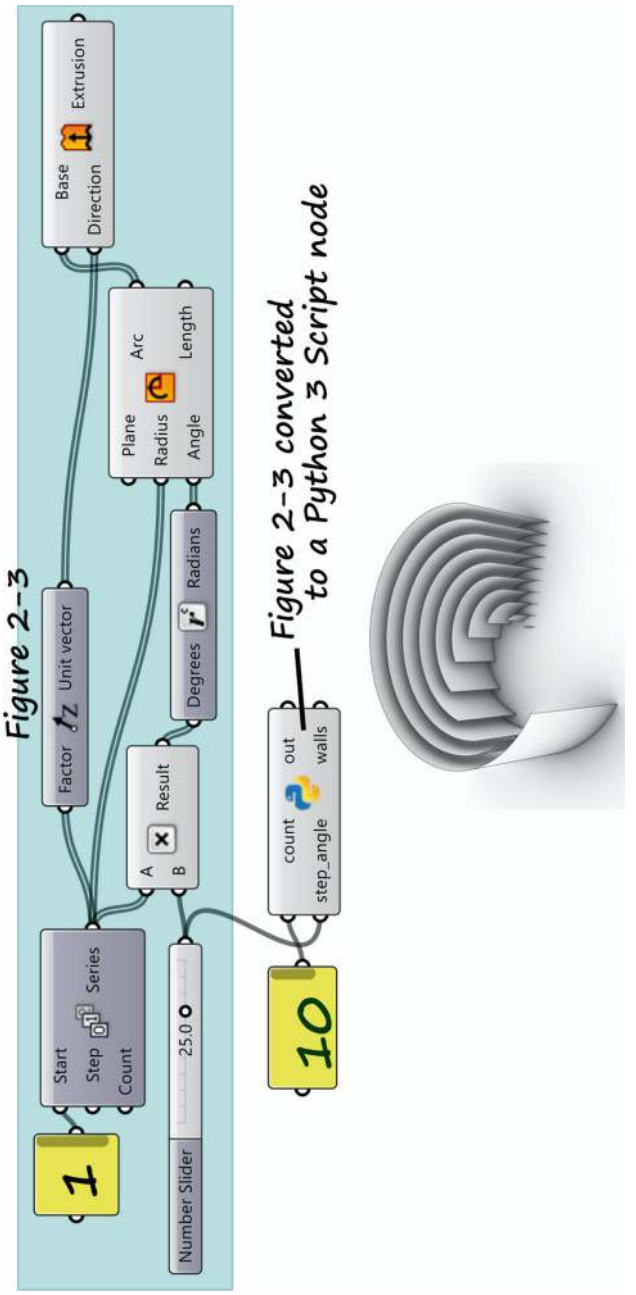


Figure 13-23. *The node graph (top) from Figure 2-3 which generates a series of nested arc walls, with arc angle increasing at fixed increments. The graph is converted to a Python 3 Script node with two inputs—count (how many walls to create) and step_angle (arc angle increment). See* `/Ch13/BuildingRhinoCommand/Figure2-3_series.gh`

The script node has two inputs—*count* (number of walls to create) corresponds to the Count input of the Series node in the original graph, whereas *step_angle* is the arc angle increment in degrees. We will reuse the Tooltip texts for the inputs/output here for the Prompt... of their contextual inputs/output counterparts later in this section.

Table 13-3. *Type Hints, Access Hints, and Tooltips for the Python 3 Script node in Figure 13-23*

Name	Input or Output	Type Hints	Access Hint	Tooltip
count	input	int	Item Access	Number of walls to create.
step_angle	input	float	Item Access	Angle increment (for the arc defining the wall's curvature) between neighboring walls.
walls	output	Surface	N/A	Nested curved walls as a collection of surfaces.

Listing 13-1 shows the code inside the Python 3 Script node in Figure 13-23. We start by importing `ghpythonlib.components` as the shorthand `ghcps` via which to call GH components as Python functions.

The five lines of code after the import statement translate the nodes in Figure 2-3 to Python function calls. Line 1 replaces the Series node with Start = 1 and Step = 1. Instead of using the default value of 10 for Count, we replace it with the input `count` to the Python 3 Script node so we can vary the number of walls.

Line 2 multiplies the output of Series (a list of `count` numbers) with the input `step_angle` and converts the results from degrees to radians—this produces a list of angles, one per wall, for how wide each wall's arc will span.

Line 3 creates an XY plane centering at (0, 0, 0), which is to be used by Line 4 as the Plane input for Arc.

Line 4 calls Curve ➤ Primitive ➤ Arc as a function to create a list of arcs (curves) based on the list of angles derived by Line 2. We use the output of Series (Line 1) as the Radius input for Arc so the arcs will have increasing radii as well as angles. The XY plane centering at (0, 0, 0) is used as the Plane input so all the arcs will lay on the XY plane centering at the global origin.

Line 5 calls Surface ➤ Freeform ➤ Extrude to extrude the arcs along Z+ based on the Series output (Line 1) so the walls have increasing heights. Note that the Direction input is computed as Unit Z with the Series output (Line 1) as its Factor.

Listing 13-1. Python code inside the Python 3 Script node in Figure 13-23, which is converted from the graph in Figure 2-3 (see `/Ch13/BuildingRhinoCommand/Figure2-3_series.gh`)

```
import ghpythonlib.components as ghcps

radii = ghcps.Series(1, 1, count)
angle = ghcps.Radians(ghcps.Multiplication(step_angle, radii))
xy_plane = ghcps.XYPlane(ghcps.ConstructPoint(0,0,0))
arc, _ = ghcps.Arc(xy_plane, radii, angle)
walls = ghcps.Extrude(arc, ghcps.UnitZ(radii))
```

Note You can find both a reproduction of Figure 2-3 and its Python 3 Script node counterpart (as shown in Figure 13-23) in the file `/Ch13/BuildingRhinoCommand/Figure2-3_series.gh` in this chapter's downloaded source files.

Adding Contextual Inputs and Output(s)

With the node graph converted to Python 3 Script, we can now use it to create a Rhino command. Since we want the command to be interactive and prompt the user for number of walls to create (`count`) and angle increment between arcs (`step_angle`), we'll insert contextual inputs and output in Figure 13-23 while retaining the original inputs as default values.

As shown in Figure 13-24, we'll add the contextual counterpart for each of the Python 3 Script node's input/output, so when the GH definition is used to create a Rhino command, ScriptEditor is able to infer what the command should prompt the user to enter interactively.

We use Params ➤ Util ➤ Get Integer for `count` as it controls the number of walls to create and should be a whole number. We'll right-click Get Integer to rename it *count* and edit its *Prompt...* by copying and pasting the Tooltip string from the *count* input of the Python 3 Script node. We also set *Minimum...* to 1 and *Maximum...* to 500 to give it a range of 1 to 500.

We use Get Number for `step_angle` since it controls the angle increment for the wall arcs and therefore could be a decimal number. Since `step_angle` is in degrees, we set *Minimum...* to 1 and *Maximum...* to 360. We'll also copy its Tooltip from the Python 3 Script node to its Get Number counterpart's *Prompt...*.

Lastly, we'll link the Python 3 Script node's `walls` output to Context Bake. We right-click Context Bake to rename it and copy the Tooltip from its Python 3 Script counterpart over to its *Prompt...*.

The contextual inputs and output along with their parameter values are summarized in Table 13-4.

Note You can find the GH definition containing Figure 13-24 in `/Ch13/BuildingRhinoCommand/Figure2-3_series_contexual.gh` in this chapter's downloaded source files.



Figure 13-24. *Adding contextual counterparts to the Python 3 Script node's inputs/output so ScriptEditor can infer what the interactive prompts for the Rhino command should be (see `/Ch13/BuildingRhinoCommand/Figure2-3_series_contexual.gh`)*

Table 13-4. *Parameter values for the contextual inputs/output in Figure 13-24*

# in Figure 13-24	Type	Name	Minimum...	Maximum...	Prompt...
1	Get Integer	count	1	500	Number of walls to create.
2	Get Number	step_angle	1	360	Angle increment (for the arc defining the wall's curvature) between neighboring walls.
3	Context Bake	walls	N/A	N/A	Nested curved walls as a collection of surfaces.

Linking the GH Definition Under Commands/ in a ScriptEditor Project

We'll now create a Rhino command from the GH definition in Figure 13-24. In ScriptEditor, create a new project (File ➤ Create Project) named CurvedWalls, and use the provided `/Ch13/BuildingRhinoCommand/icon_walls.svg` and `icon_walls_dark.svg` as the light and dark theme icons, respectively. Right-click Commands/ ➤ Link Command Script (Grasshopper) and select the GH definition in Figure 13-24 (`/Ch13/BuildingRhinoCommand/Figure2-3_series_contexual.gh`). Click the newly added item under Commands/, click the Edit icon button to its right (which looks like a pencil), and in the Edit Command window that opens, enter CWalls as the Name and a Description.

To build the command, click "Publish active project" and follow the same steps for building Grasshopper plugins as described in Figure 13-10 in the section "Creating Installers and Packages." The build will yield the following list of files:

- src: Folder containing Visual Studio solution and CSharp files for the command
- CurvedWalls.rhp: Installer for the command
- CurvedWalls.rui: File containing the interface for the command, in this case, a Toolbar with the command's button
- CurvedWalls.rui.bak: Backup for the *.rui file
- curvedwalls-0.1.0-beta-rh8-win.yak: Yak package for the command
- icon.png: Exported PNG file of the light theme icon
- manifest.yml: Metadata for the command, in YAML source format

---

Note A sample project as described in this section `/Ch13/BuildingRhinoCommand/CurvedWalls.rhproj` is provided in this chapter's downloaded files.

Installing the Rhino Command You've Just Built

To install the command, drag and drop the `*.rhp` file to the Rhino window. To install the `*.rui`, go to Window ➤ Toolbars and Macros...; in the window that appears (titled "All RUIs"), click Open RUI and select the `*.rui` file.

Verifying the Rhino Command

After installing the `*.rhp` and the `*.rui`, you should see a floating Toolbar containing the button for the CWalls command as shown in Figure 13-25.



Figure 13-25. *Toolbar containing a button for the CWalls command*

Try clicking the button to invoke the command. You'll see the message

```
Command: _CWalls
```

appear in the log area above the Rhino command line, as shown in Figure 13-26 (note that it may take a few seconds for the command to load). Rhino will then prompt you with

```
Number of walls to create <10> (a=10):
```

Notice this is the *Prompt...* that we entered for the *count* contextual input (Get Integer) in Figure 13-24. Since we've set up a default value of 10 in the original GH definition, the prompt shows `<10> (a=10)` to tell you that you can hit enter without entering a number to use the value of 10. We'll type in 20 here to see if it gets applied correctly:

```
Number of walls to create <10> (a=10): 20
```

The next prompt asks you to enter the angle increment in degrees. This is the *Prompt...* that we customized for the *step_angle* contextual input (Get Number) in Figure 13-24. In Figure 13-24, the default value we used is 25.0. We'll type in 15 here to test that you can override the default correctly:

```
Angle increment (for the arc defining the wall's curvature) between
neighboring walls <25> ( a=25 Undo ): 15
```

If all is well, you should see the set of 20 curved surfaces appear in the viewport, as shown in Figure 13-26.

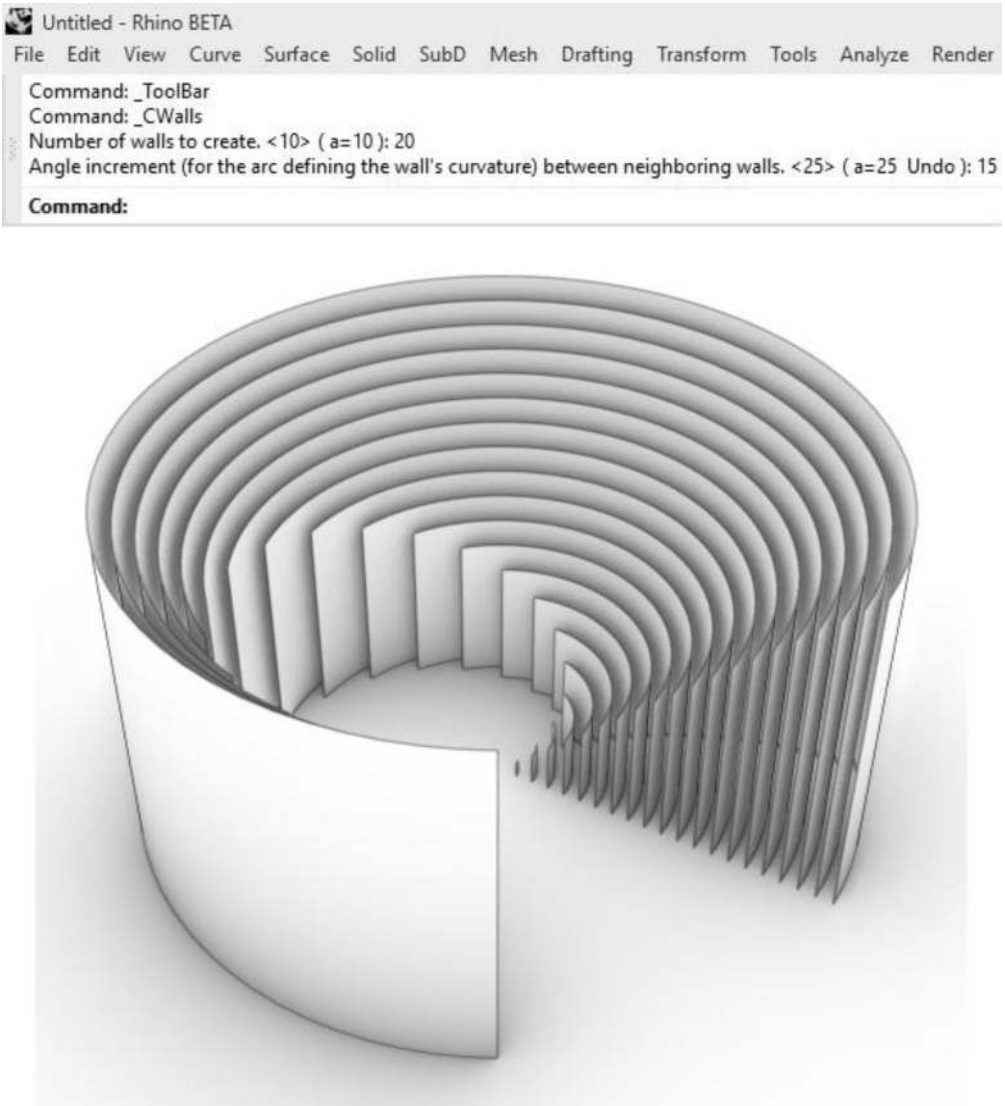


Figure 13-26. *You can run the CWalls command by clicking its button in the Toolbar. Entering "Number of walls to create" (count) = 20 and "Angle increment" (step_angle) = 15 produces the set of surfaces shown*

You should also try running the command by typing CWalls in the Rhino command line. Hitting enter on both prompts, thus using the default values for both count = 10 and step_angle = 25, will produce the set of ten surfaces with arcs 25 degrees apart shown in Figure 13-27.

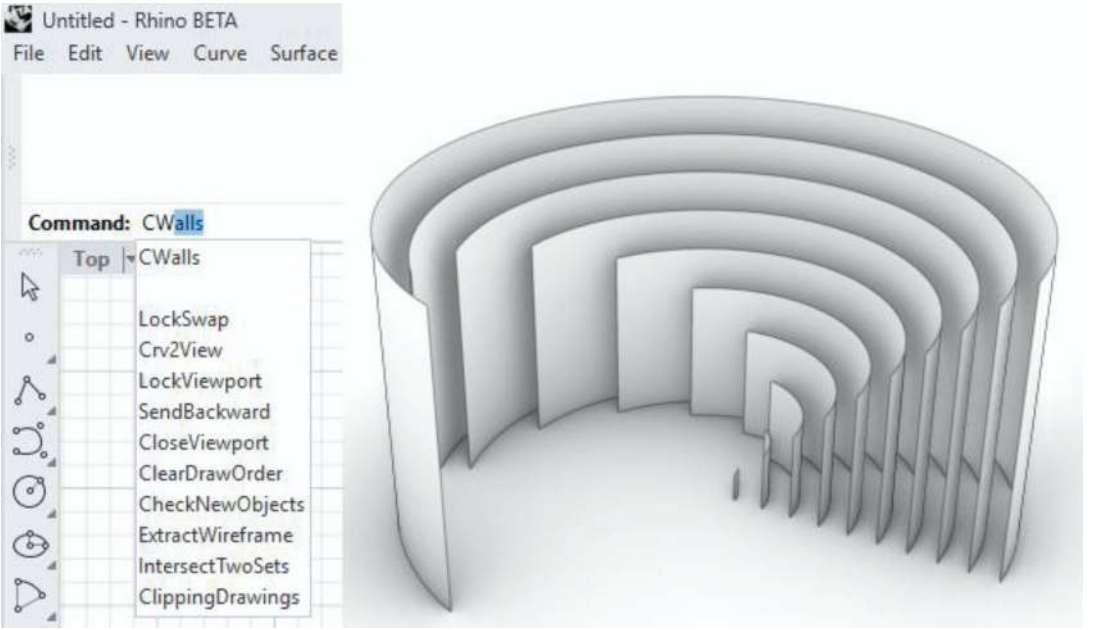


Figure 13-27. *You can also run the CWalls command by entering "CWalls" in the Rhino command line. Using default values for both count and step_angle (by hitting enter for both prompts without typing in a value) produces the set of ten walls shown with arcs at 25 degrees apart*

Publishing the Rhino Command

The process of publishing a Rhino command on the Yak server is identical to how you would publish a Grasshopper plugin. Please refer to the section "Pushing to the Rhino Package Server (Yak)."

Remote Control Panel

Instead of building a plugin or command, sometimes you just need a way to easily share a GH definition with others while exposing parametric controls in an intuitive way. Grasshopper has a feature called Remote Control Panel that lets you do just that—you can select which sliders to appear in a panel that ducks in the Rhino window. After that, you can minimize the GH editor and do a show-and-tell in the Rhino viewport by dragging the sliders in the Remote Control Panel.

The sliders (or other widgets) in the Remote Control Panel and their counterparts in the GH definition act like two-way proxies—changing the slider in the Remote Control Panel changes the slider in the GH definition and vice versa.

The Rhino viewport displays any previews based on the settings of the GH definition. For example, if the option "Only draw preview geometry for selected objects" is set in the GH definition, the components you want to preview must be selected in the GH definition for them to show up in the Rhino viewport, whether or not the Remote Control Panel is used.

Adding Input Widgets to the Remote Control Panel

To see how this works, we'll create a Remote Control Panel for the curved walls definition from Figure 13-23. To ensure that the Remote Control Panel will show up in Rhino, click the "+" button as shown in Figure 13-28 and check the "Grasshopper" box. After that, you should see the Grasshopper panel appear, which should look like Figure 13-29—this is where the Remote Control Panel will show up.

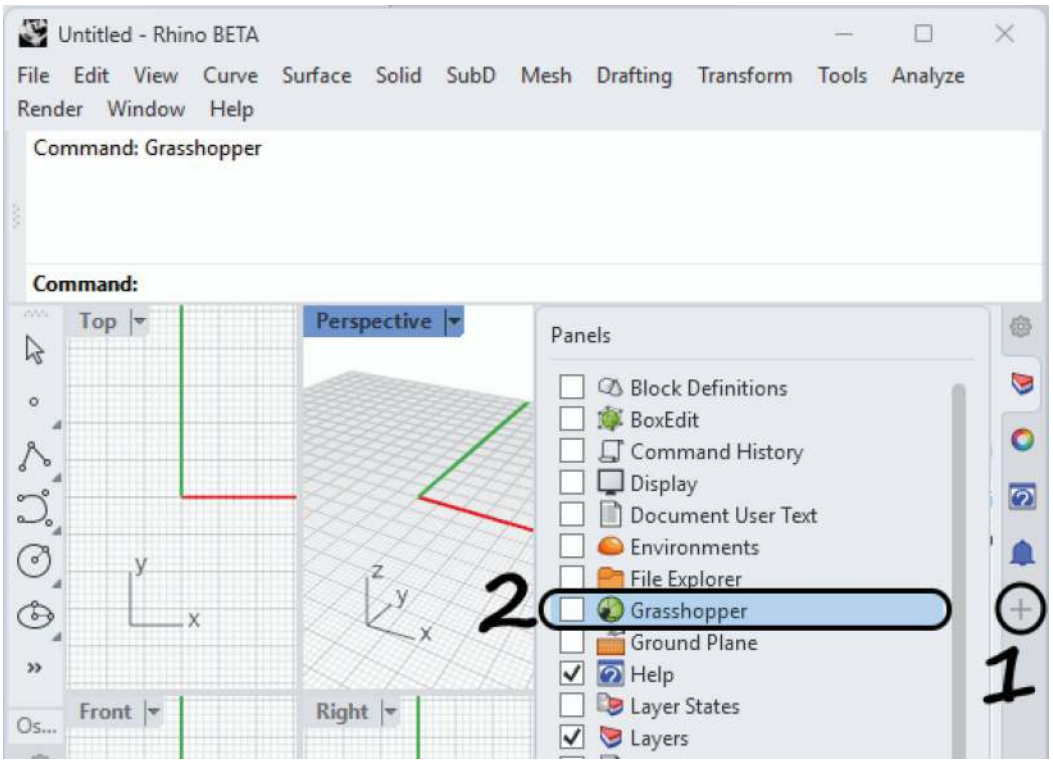


Figure 13-28. *To ensure the Remote Control Panel shows up, click the "+" button (1) at the right side of the Rhino window and check the "Grasshopper" box (2)*

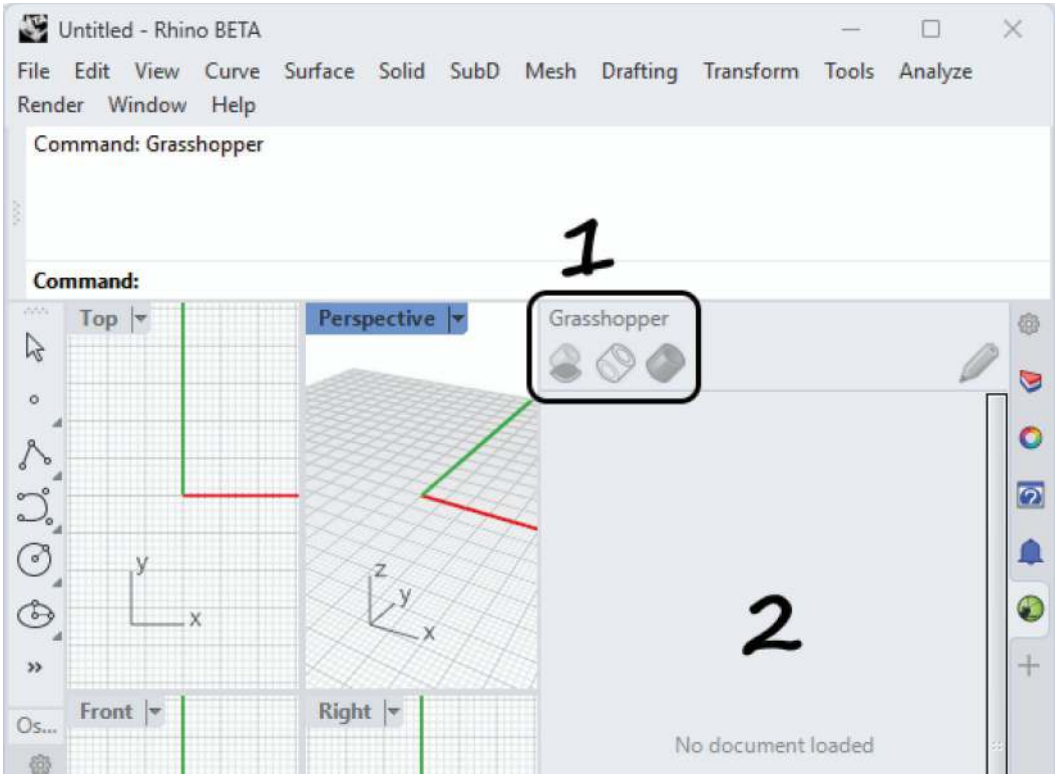


Figure 13-29. *The Grasshopper panel (1), where the Remote Control Panel will appear (2)*

Figure 3-30 shows the curved walls definition slightly modified, with the Panel linked to the Count input of Series replaced by a Number Slider (a). *Most input widget types under Params ➤ Input can be added to the Remote Control Panel.* We will add both the slider controlling how many walls to create (a) and the slider for adjusting the step angle (b) to the Remote Control Panel by right-clicking each slider ➤ Publish to Remote Panel. Afterward, the Remote Control Panel will look like Figure 13-31, with the name of each slider taking after its counterpart in the GH definition.

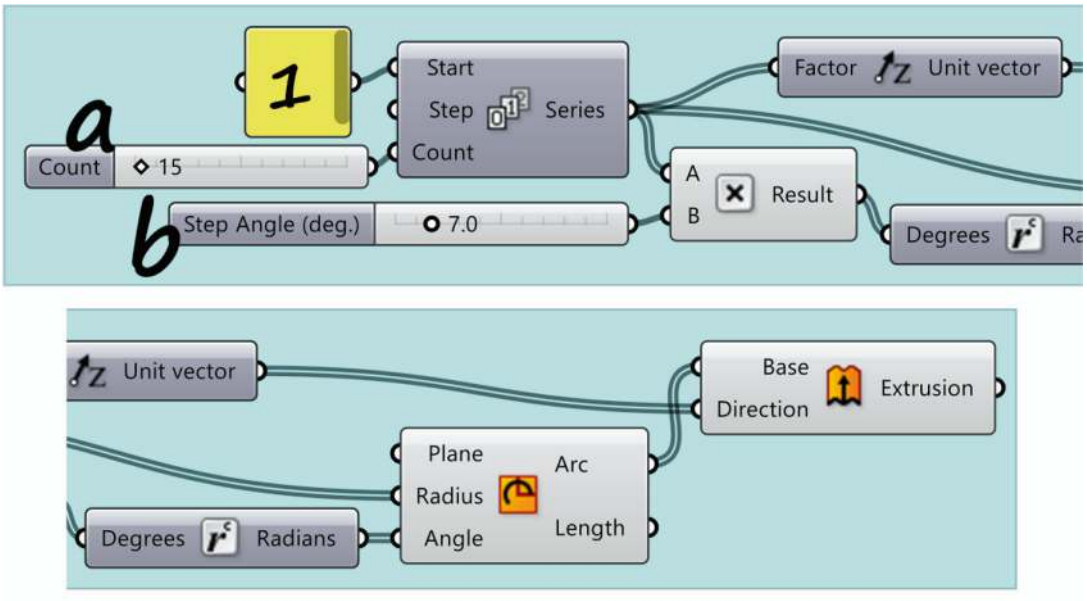


Figure 13-30. *The curved walls node graph from Figure 13-23, with the Panel linked to the Count input of Series replaced by a Number Slider (a). We add both the slider for controlling how many walls to create (a) and for adjusting angle increment between walls (b) to the Remote Control Panel by right-clicking each slider ➤ Publish to Remote Panel*

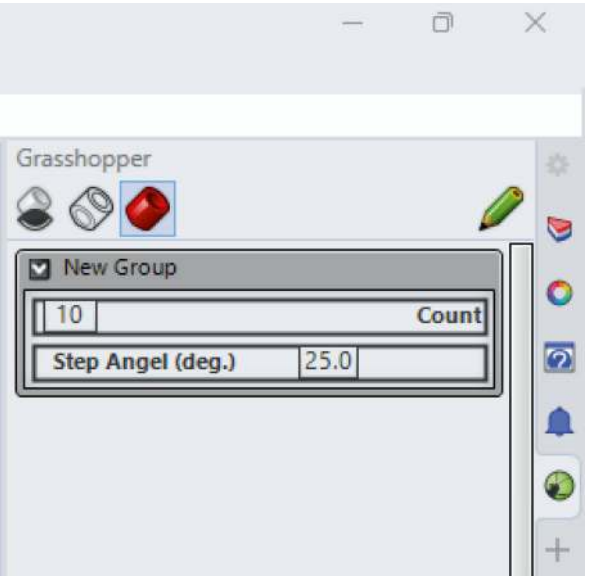


Figure 13-31. *Remote Control Panel after the two Number Sliders (a and b in Figure 13-30) have been published to it. You can see that the name of each slider in the panel takes after the name of its corresponding slider in Figure 13-30*

Customizing the Remote Control Panel

We can customize the Remote Control Panel with color, labels, and dividers. To enable editing, click the "pencil icon" at the upper-right corner of the panel ((1) in Figure 13-32).

Changing the Title and Color

Once editing is enabled, right-clicking somewhere along the top border of the panel will open the dialog shown in (2) of Figure 13-32, which allows you to enter a new title and color for the panel.

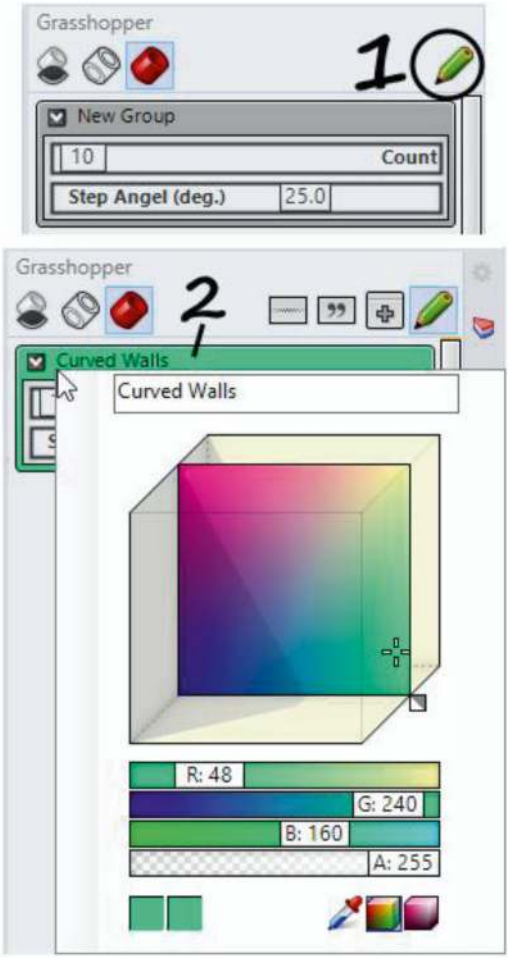


Figure 13-32. *(1) Click the "pencil icon" to enable editing of the Remote Control Panel, and (2) right-click along the top border of the panel to open the dialog for editing the panel's name and color*

Adding Labels

You can add text labels to use as descriptions for the sliders or the panel itself. As shown in Figure 13-33, to create a label, enable editing, click the button at the top that looks like a double quote, type in a message, and hit enter. After the label is created, you can drag and drop it anywhere within the panel to organize it.

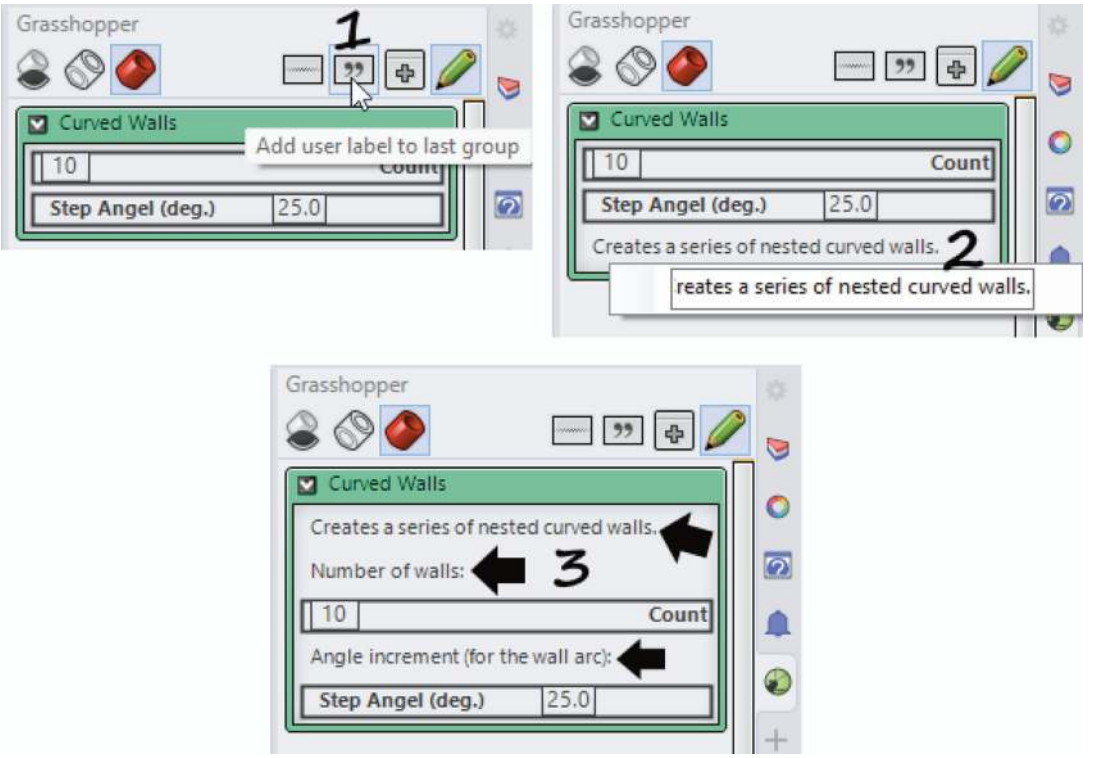


Figure 13-33. *To add a label, click the (1) button that looks like a double quote, (2) type in the message, and hit enter. After the label is created, you can drag and drop it anywhere within the panel to organize it. (3) shows three labels, one for each slider and a description for the panel at the top*

Adding Dividers

You can also add dividers to group items within the Remote Control Panel. After enabling editing, click the button with a horizontal bar as shown in Figure 13-34 to create a new divider. After its creation, you can drag and drop the divider anywhere within the panel to organize it.

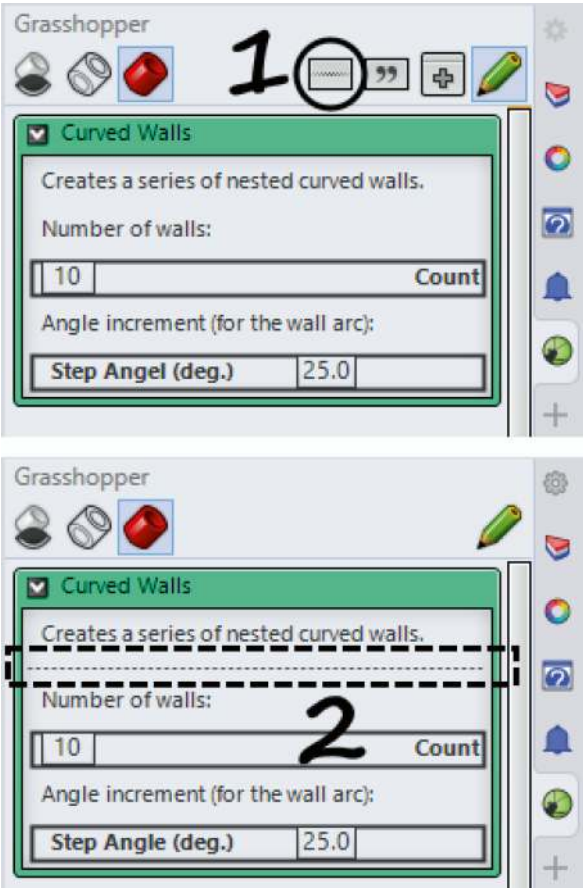


Figure 13-34. *You can add dividers to group items in the Remote Control Panel. After enabling editing, click the (1) button with a horizontal bar to create a new divider, which will look like (2). You can drag and drop a divider anywhere within the panel to organize it*

Note You can find the GH definition with the complete Remote Control Panel as described in this section (Figure 13-34) in `/Ch13/RemoteControlPanel/Figure2-3_series_RC_panel.gh` in this chapter's downloaded source files.

Juggling Multiple Grasshopper Definitions with Remote Control Panels

You can have multiple Grasshopper definitions open at the same time, each with its own Remote Control Panel. Whichever GH definition is currently selected, therefore displayed on the GH editor canvas, will have its Remote Control Panel shown in the Rhino window.

To select a different definition, click the file name at the upper-right corner of the GH editor to display the list of currently open definitions, as shown in Figure 13-35, and click on a definition to select it.

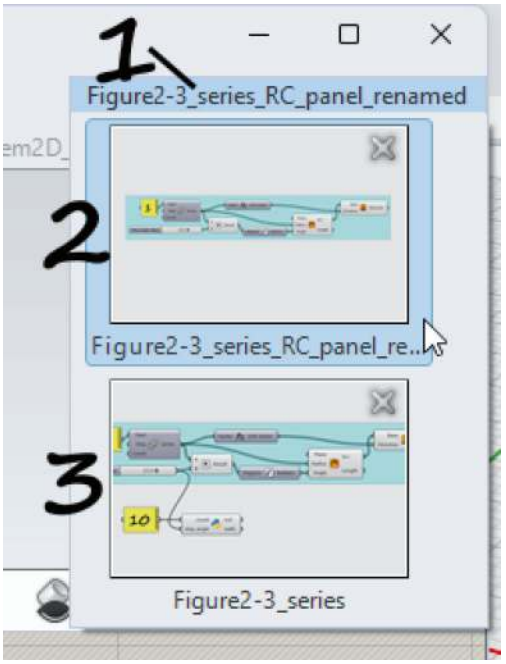


Figure 13-35. *Switching between multiple open GH definitions. (1) Click on the file name at the upper right corner of the GH editor to bring up the list of currently open files (2 and 3). Click to select a definition to display on the canvas as well as open its Remote Control Panel in the Rhino window*

Summary

In this chapter, you learned how to build Grasshopper plugins and Rhino commands from GH definitions containing Python 3 Script nodes, using ScriptEditor—Rhino's plugin/command building tool since Rhino 8. In addition, you learned how to share GH definitions with others without creating plugins by exposing parametric controls with Remote Control Panels.

The first part of the chapter showed you how to build a GH plugin in two ways. The first method creates one component per Python 3 Script node in the source GH definition. The second method combines all script nodes into a single component based on contextual inputs/output(s) you add to the source GH definition.

The second part of the chapter guided you through building and publishing your plugins to Yak, the Rhino Package Server. Along the way, you learned the anatomy of a Yak package and how to push to the Yak test server for practice (a server that is wiped nightly). You also found out how to distribute the same Yak package on Food4Rhino, either by directly uploading the file or linking a package already on the Yak server.

In the third part of the chapter, you learned how to create interactive Rhino commands that prompt users for inputs while providing optional default values. You also discovered what constitutes a command package and how to publish it using the same process as GH plugins.

Finally, we wrapped up the chapter by discussing how to share GH definitions without creating a plugin or command, by adding Remote Control Panels to expose parametric controls in an intuitive manner. You learned how to edit the panel color, organize items by dragging and dropping, add descriptive labels, and group items using dividers.

References

Lupiani, I. (2025). *Procedural Content Generation for Games: Automate 3D Asset and Environment Creation with Blender Python and Geometry Nodes*. Apress.

Prusinkiewicz, P., Lindenmayer, A., Hanan, J. S., Fracchia F. D., Fowler, D., de Boer, M. J. M., Mercer, L. (1990). *The Algorithmic Beauty of Plants*. Springer-Verlag, New York.

Index

A

Anonymous variable
  Python, 240
Area
  component for finding area and center point, 144, 261, 276, 283, 353, 410, 510
Array
  Hexagonal (component), 107
  Linear Array, component for arranging a list of geometry evenly spaced along a straight line, 101, 353
  polar array, polar grid, 122
    To Polar (node), 130, 133
    Polar Array (component), 123
  Rectangular Array (component), 110, 150, 166, 173, 181
Attractor, 99

B

Boolean
  Boolean Toggle (component), 38, 375
  a true or false value, 38, 46
  via Button (component), 38
  via Panel (component), 38
Boundary Surfaces
  component for creating surface with given curve, 268
  component for creating surface with given curve as border, 106, 114, 194, 231, 238, 356, 375, 378, 417, 510
Bounding Box
  component for finding the minimal box that will enclose the entire geometry (right-click, "Per Object" finds a bounding box for each object in the list), 60, 111, 140, 181, 391
Brep
  Area, component for finding area and center point, 261
  Brep | Brep, component for finding intersection curves between two breps, 288–290
  Brep Edges (component), 159–161, 170, 179–181, 192, 214, 276, 287
  Brep Join (component), 269
  Deconstruct Brep, component for isolating the faces, edges, and vertices of a Brep, 269, 272, 311, 319, 323, 391
  Mesh Brep, component for converting brep to mesh, 191, 372, 394, 410
  Voronoi 3D (component), 261, 304

C

Canvas, 6
  backdrop in the GH editor on which you compose a GH definition, 4
Component
  color
    error (red), 15
    warning (orange), 15

© Isabel Lupiani 2026
I. Lupiani, *Parametric and Generative Design with Rhino Python and Grasshopper*,
https://doi.org/10.1007/979-8-8688-2724-2

Component (*cont.*)
custom name, 17
Display (GH editor top menu bar)
Draw Full Names, 17
Draw Icons, 17
Enabled (right-click, option), 15
grouping, 92
input
default value, 24
Extract parameter (right-click, option), 26
Internalize data (right-click, option), 26
Set One or Set Multiple (right-click, option), creating data in GH, 27
Set One or Set Multiple (right-click, option), referencing from Rhino, 24
Wire Display (right-click, option, Default, Faint, or Hidden), 41, 114
locating, 9
node, an operation or a data container in GH
locating components, 5
organization
align, 90–92
distribute, 90–92
output, 13, 15, 18–22, 29, 46
Preview (right-click, option), 15
selection
box select, 92
Shift-click (one by one), 92
Construct Mesh
component for creating a mesh from existing lists of vertices and faces, 399
Construct Point
component for creating a point from X, Y, and Z, 39, 76, 244, 375, 386, 398, 510
Contextual input
"Get data type" components under Params ➤ Util that you add to a GH definition for ScriptEditor to infer as inputs when combining multiple scripts., 543
Contextual output
Params ➤ Util ➤ Context Print and Context Bake, which are components you add to a GH definition for ScriptEditor to infer as outputs when combining multiple scripts, 543
Convex hull
Convex Hull 3D, component for finding the 3D convex hull of a given (set of) geometry (-tries), 473
the minimal convex polygon fully enclosing a given (set of) geometry (-tries), 473
Culling
Cull Nth (component), 56
Cull Pattern (component), 56
Curve
Arc (component), 53
Area, component for finding area and center point, 135, 276, 283, 417, 510
Area (node), finding the area or centroid point of, 261
Boundary Surfaces, component for creating surface with given curve as border, 106, 194, 231, 238, 247, 268, 356, 375, 378, 417, 510

Circle CNR, component for creating a circle from center, normal, and radius, 208
Closed, component that tells which curves are closed with a list of Boolean (True/False) values, 356
Curve Closest Point, component for finding the closest point on a curve to a given point, 142
Divide Curve (component), 230, 276, 393
End Points, component for finding the end points of a curve, line, or polyline, 159, 170, 179, 214, 230, 236, 238, 276
Evaluate Curve (component), 160, 170, 172, 179, 214, 218, 219, 276, 312
Evaluate Length (component), 237
Explode, component for extracting the points constituting a curve, 471
Extrude (component), 57, 208, 219, 231, 268
Extrude, component for growing geometry along a given direction, 512
Interpolate (component), 141, 198, 208, 221, 244
Join Curves (component), 160, 166, 180, 393, 405
Length, component for measuring curve length, 417, 509
Loft (component), 198, 200, 206, 214
Maelstrom, component that swirls curves, 277
Maelstrom (node), swirls curves, 179
Mirror (component), 160, 166, 173, 180, 277
Move (component), 108, 151, 160, 180, 273
Pipe (component), 252
Pipe, component for giving curves volume, 221
Polygon (component), 101, 150, 231, 236
Polygon Center (component), 106, 111, 159, 179
PolyLine (component), 206, 230, 276
Project, 252
Project Curve (component), 273
Rectangle (component), 110, 159, 170, 179, 211, 246, 247, 251, 268
Rotate (component), 155, 233
Scale, component for scaling geometry uniformly, 106, 136, 214
Scale NU, component for scaling geometry non-uniformly (a different amount along each dimension), 111, 150
Scale NU (node), non-uniform scaling, 150
Scale (node), uniform scaling, 214, 509
Shatter (component), 237
Cylinder
component for creating a cylinder as an untrimmed surface, 414

D

Data
data conversion, 47
data matching, 47
data tree, 47
filtering
Stream Filter (component), 107, 112, 121, 136
Interpolate Data (component), 140

Data conversion
how GH converts data when compatible but non-identical types are connected, 48
numbers to Boolean, 48
plane
from planar geometry, 49
XY plane from point, 49
rounding numbers, 48
Data matching
Cross Reference (component)
Diagonal, 66
Holistic, 66, 198, 199
how GH deals with inputs of mismatched sizes, 61
Longest List (component), 61
Flip, 65
Interpolate, 65
Repeat First, 65
Repeat Last, 62, 65
Wrap, 65
Shortest List (component)
Interpolate, 65
Trim End, 65
Trim Start, 65
Data tree
branch
data path, path, 70
via Tree Branch (component), 72
flattening
Flatten (right-click input/output of a component, option), 74
Flatten Tree (component), 74
Flip Matrix, component for swapping rows and columns, 72, 391
grafting
Graft (right-click input/output of a component, option), 76
Graft Tree (component), 78
reversing
Reverse (right-click input/output of a component, option), 81
simplifying
Simplify (right-click input/output of a component, option), 70
Simplify Tree (component), 72
Tree Statistics (component), 79
a type of hierarchical data structure in GH, 41, 70
Deconstruct
component for separating a point or vector's X,Y,Z coordinates, 104, 238, 351, 393, 398, 405
Deconstruct Box
component for finding the X, Y, and Z domains of a box, 103, 351, 414
Deconstruct Brep
component for isolating the faces, edges, and vertices of a Brep, 269, 272, 311, 319, 323, 391
Deconstruct Face
component for extracting (up to 4) indices of the corner vertices that frame a mesh face, 323, 324, 342
Deconstruct Mesh
component for extracting the vertices and faces of a mesh, 323, 341, 398
Definition (script, node graph)
a GH program, 4, 5
version, 8
Dispatch
component for dividing a collection into two based on a given pattern, 356, 471, 512

Domain
Bounds, component for finding the span of a domain, 111
Construct Domain (component), 31, 211, 230, 236, 268, 385
creating using Panel (component), 31
Deconstruct Domain, component for finding the start and end values of a domain, 104, 112, 140, 351
a 1D or 2D range of values, 32
Remap Numbers, component for mapping a number from one domain to another, 111, 386, 398
Dual
mesh (the dual of a mesh), 284

E

Emitter, 99
Expansion
L-System, 426
Rewriting (*see* Rewriting)
Expression
component for entering a math formula, 198, 410
Extrude
component for growing geometry along a given direction, 54, 231, 268, 417, 511, 512
Extrude (component), 117, 208, 217, 219

F

Face Boundaries
component for finding edges framing each mesh face as a curve, 283

G

Geographic Information System (GIS), 479
building footprint, 483
Import SHP, component for reading polyline curves from shapefile, 493, 508
shapefile, 483, 490, 493–494, 497–502
Geometry
Brep, 240, 241, 391, 394
curve, 22, 47, 101, 123, 126, 172, 181, 194, 252, 277, 394, 474
line, 160, 180, 252
mesh, 260, 279, 281, 283, 300, 324, 342, 352, 367, 369, 394, 474
Move (node), 108, 115
polyline, 56, 230, 236, 437, 446
Scale NU (node), non-uniform scaling, 111, 150
Scale (node), uniform scaling, 214, 260, 299, 372, 509
surface, 28, 39, 47, 123, 151, 252, 260, 375, 376, 378, 379, 394
transform, 42, 47, 58, 101, 121, 123, 173, 194, 252, 387
GH, *see* Grasshopper (GH)
*.gha
file extension of GH plugin installer, 538
GIS, *see* Geographic Information System (GIS)
Graph Mapper
Bezier (right-click, Graph types), 51
component for generating points sampled from a curve, 51, 244, 245
Sine (right-click, Graph types), 244

Grasshopper (GH)
Rhino's visual scripting language, 4
Grasshopper Editor
canvas, 4
compass, 8
Markov widget, 8
search box, 12
Grid
Hexagonal (component), 273
Linear Array, component for arranging a list of geometry evenly spaced along a straight line, 101
Polar array (component), 123
polar array, polar grid, 122
Polar Array (component), 130
polar grid, polar array, 122
Rectangular (RecGrid, component), 310
Rectangular Array (component), 110, 150, 166, 173, 181
Triangular (TriGrid, component), 311, 336, 339

H

Horizontal Frames
component for creating evenly spaced planes along a polyline, 209
Plane, 209

I, J, K

Identity matrix, 440
Image
Image Sample, component for sampling image pixel intensity, 384, 385
Import Image, component for converting image to mesh with embedded color info, 397, 398
SVG
Inkscape, 524
Image Sampler, 386
Inkscape, 524
Input (to GH definition)
MD Slider, component for creating a 2D slider, 111

L

Length
component for measuring curve length, 417, 509
Line
Line, component for creating a line segment between two given points, 312
Line SDL (component), 230, 236, 375
List
Cull Index (component), 230
Dispatch, component for dividing a collection into two based on a given pattern, 356, 471, 512
List Item (component), 159, 172, 272, 311, 391
Mass Addition, component for summing a list of numbers, 236, 237, 352, 509
Partition List (component), 57, 199
Random (component), 78, 236, 238
Range (component), 51, 198, 231, 244
Range, component for creating evenly spaced tick marks between a start and an end value, 385
Series (component), 53, 76, 199, 204

Sub List (component), 57
Weave, component for combining two lists based on a pattern, 512
Loft (component), 5, 198, 200, 206, 214
Loft Options (component), 393, 395
L-System, 425
axiom, 426
rewriting (expansion), 426
rewriting rules
predecessor, 426
successor, 426
LunchBox (GH plugin)
Möbius Surface (component), 191
Panels (tab)
Quad Panels (component), 192
Skewed Quads (component), 266, 268
Staggered Quad Panels (component), 192
Triangle Panels (component), 192
Triangular Panels A (component), 192

M

Maelstrom
component that swirls curves, 274, 275, 277–278
Mass Addition
component for summing a list of numbers, 236, 237, 352, 509
Math
Clamp, component for limiting numbers to a range, 106
comparison
Smaller Than (component), 512
Division, 236
Expression, component for entering a math formula, 198, 410
Mass Addition, component for summing a list of numbers, 236, 352, 509
Mesh
Area, component for finding area and center point, 353, 410
Area (node), finding the area or centroid point of, 261
Bounding Box (component), 351
Construct Mesh, component for creating a mesh from existing lists of vertices and faces, 399
Convex Hull 3D, node for finding the minimal convex 3D polygon that will enclose a piece of geometry, 473
Deconstruct Box, component for finding the X, Y, and Z domains of a box, 351
Deconstruct Face, component for extracting (up to 4) indices of the corner vertices that frame a mesh face, 323
Deconstruct Mesh, component for extracting the vertices and faces of a mesh, 323, 398
dual (the dual of a mesh), 283
Face Boundaries, component for finding edges framing each mesh face as a curve, 283
Mesh Brep, component for converting brep to mesh, 191, 372, 394, 410
Mesh Difference, component for finding the solid difference between two (sets of) meshes, 372, 475

Mesh (*cont.*)
Mesh Join, component for combining multiple meshes into one, 394, 407, 410, 411, 414, 476
Mesh | Plane, component for finding the intersection between a mesh and a (list of) plane(s), 356
Quad Remesh, component for turning faces into quads and reducing poly count, 260, 279, 281, 299, 323, 340, 369
Quad Remesh Settings, 260
Quad Remesh Settings, node for specifying parameters for Quad Remesh, 279, 281, 299, 323, 340, 369
reducing poly count (*see* Quad Remesh)
ShrinkWrap, component for generating a mesh that fully encloses a given geometry with an optional offset, 474
SubD from Mesh, component for converting a mesh to SubD, 299
TriRemesh, component for converting mesh faces to equilateral triangles, 281, 283, 341
Unify Mesh, component that ensures the normals of a mesh are consistent, 369, 394, 407, 410
Voronoi 3D (component), 261, 286, 288, 292, 295, 297, 300
Mesh Brep
component for converting brep to mesh, 191, 372, 394, 410
Mesh Difference
component for finding the solid difference between two (sets of) meshes, 372, 475
Mesh Join
component for combining multiple meshes into one, 394, 407, 410, 411, 414, 476
Möbius strip, 190
MultiPipe
component for turning curves into SubD, 150, 152, 153, 161, 173, 269, 273, 279, 281, 283, 287, 289, 470

N, O

Node
polar array, polar grid
To Polar (node), 130, 133
Node graph, *see* Definition
Normalize, *see* Unitize
Normalized length
cumulative, 236
Normal vector
Evaluate Surface, component for finding the normal vector of a surface, 358
a vector that points away from and is perpendicular to a surface, 357
Number Slider
decimal, 33, 35
integer, 33, 35
precision
custom, 33, 35
default, 33–35
property editor, 35
range
custom, 35
default, 33

P

Panel
- as Boolean input, 38
- component for inputting or outputting text, 10, 21, 22, 29
- as domain for input, 31
- as a list of numbers for input, 30
- as pattern input, 38
- as Point (component), 39
- as a single number input, 29
- as sticky note, 94

Perp Frames
- component for creating evenly spaced planes centering on a polyline, 206
- Plane, 206

Pipe
- component for giving curves volume, 57, 221, 252, 253

Plane
- Horizontal Frames, component for creating evenly spaced planes along a polyline, 209
- Perp Frames (component), 206

Point
- Construct Point, component for creating a point from X, Y, and Z, 39, 76, 244, 375, 386, 398, 510
- Deconstruct, component for separating a point or vector's X,Y,Z coordinates, 111, 238, 353, 393, 398, 405
- Distance, component for finding distance between points, 106, 111
- Interpolate (component), 198, 208, 221, 244
- a location in 2D or 3D, 72, 78, 85, 114, 139, 144, 146, 147, 246, 249, 251, 258, 261, 264, 286, 393, 477
- Move (component), 276
- Populate 3D (component), 251, 261, 286, 292, 300
- Project (component), 252
- via Panel (component), 39

Polyline
- MultiPipe, component for turning curves into SubD, 470
- Rectangle (component), 110
- Rectangle (node), 159, 170, 179

PolyLine
- component for creating a polyline by linking a sequence of points, 231, 236
- Perp Frames, component for creating evenly spaced planes centering on a polyline, 206
- Rectangle (component), 246, 247, 251
- Rectangle (node), 211, 268
- Scale, component for scaling geometry uniformly, 509

Preview
- component, 15
- lasso-select, 19
- selected, 19
- via Panel (component), 21, 70
- via Param Viewer (component), 70
  - Draw Tree, 70

Pseudo random, 238

Python
- anonymous variable, 240
- matplotlib, module for making plots, 500
- numpy, module for handling matrices and doing calculations with them
  - random, 506

Python (*cont.*)
pandas, module for reading spreadsheet, 503
a programming language you can use to create Rhino or GH scripts, 81, 189, 242, 309, 383, 437, 479, 519–573
PyPl, 480, 498
pyshp, module for parsing shapefile, 499–500
Python 3 Script (component), 47, 81–87, 89, 96, 200–207, 236, 239, 246, 319, 322, 328, 331, 339, 401, 427, 432–437, 439, 441, 447, 464, 526
Random (Python module)
Randint, function for generating pseudo random whole numbers, 240
Uniform, function for generating pseudo random decimal numbers, 240
stack, 450
Python 3 Script, 126
accessing GH components in Python, 84
Access type
Data Tree, 87
Item Access, 87
List Access, 87
adding inputs/outputs, 85
component for writing your own Python code to be used as part of a GH definition, 47, 200–207, 236, 237, 239–241, 247, 319–322, 328–330, 335, 401, 402, 437, 557, 558
Python, 47, 81–87, 89, 96, 204, 205, 236, 239, 246, 319–322, 339, 401, 432, 526
Type Hints, 87

Q

Quad Remesh
component for turning faces into quads and reducing poly count, 260, 279, 281, 299, 323, 340, 369

R

Random
component for generating a list of pseudo random numbers, 238, 240, 241, 246
pseudo random, 238
Range
component for creating evenly spaced tick marks between a start and an end value, 51, 54, 198, 385
Relay
a Component that acts as a hub for consolidating or redirecting wires, 13, 114
Remap Numbers
component for mapping a number from one domain to another, 111, 386, 398
Remote Control Panel, 565
Repeller, 100
Rewriting (expansion)
L-System, 426

S

Scribble
component for creating a 2D text object that can be placed on the canvas, 94
Script, *see* Definition
ScriptEditor

tool for creating GH plugins or Rhino commands (since Rhino 8), 520
Scripting
Expression, component for entering a math formula, 198, 410
Python, 47, 96, 236, 239, 246, 401, 432, 519–573
Python 3 Script (component), 81, 201, 204, 236, 239, 246, 319, 322, 328, 331, 339, 401, 432, 526
code editor, 82
Terminal, 82
User Object, 87
Sequence
Cull Index (component), 230
Series
component for creating a list of values in fixed intervals from a start value, 53
shapefile, 483
ShrinkWrap
component for generating a mesh that fully encloses a given geometry with an optional offset, 474
Sliceform, 348, 381
Sphere
component for creating a sphere as an untrimmed surface, 254, 262, 268, 272, 289, 295, 411
Splop
component for projecting geometry onto a target surface, 411, 414
SubD
SubD from Mesh, converting a mesh to SubD, 299
a type of high-precision sub-divided surface in Rhino for modeling organic shapes, 150
Surface
Boundary Surfaces, component for creating surface with given curve as border, 106, 268, 356, 375, 378, 417, 510
Boundary Surfaces (node), creating surface with given curve as border, 194, 231, 238, 247
Bounding Box (component), 150, 161, 181, 260, 273, 292, 299, 391, 510
Bounding Box (component, Per Object), 104, 111
Box Properties (component), 242, 260, 273, 292, 299
Brep | Brep, component for finding intersection curves between two breps, 288
Brep Edges (component), 159, 161, 170, 179, 181, 192, 214, 276, 287
Brep Join (component), 269
Center Box (component), 251, 261, 286, 292, 295
Cylinder (component), 256
Cylinder, component for creating a cylinder as an untrimmed surface, 414
Deconstruct Box, component for finding the X, Y, and Z domains of a box, 104, 414
Deconstruct Brep, component for isolating the faces, edges, and vertices of a Brep, 269, 272, 312, 314, 319, 328, 336, 391
Dimensions (component), 151
Divide Surface (component), 254
Evaluate Surface, component for finding the normal vector of a surface, 217, 358

Surface (*cont.*)
Extrude (component), 54, 117, 217
Extrude, component for growing geometry along a given direction, 417
Flow (component), 194
Loft (component), 42, 198, 200, 206, 214
Loft Options (component), 393, 395
Mesh Brep, component for converting brep to mesh, 191
Möbius Surface, 190
Move (component), 115, 121, 510, 512
MultiPipe, component for turning curves into SubD, 152, 161, 173, 269, 273, 279, 281, 283, 287, 289, 463, 466
non-orientable, 190
normal vector, a vector that points away from and is perpendicular to a surface, 357
Pipe (component), 57, 252
Pipe, component for giving curves volume, 221
Project, 252
Rotate (component), 375
Scale (node), 106, 136
Scale, component for scaling geometry uniformly, 260, 299, 372
Scale NU (node), non-uniform scaling, 111, 150
Scale (node), uniform scaling, 214
Solid Difference (component), 268, 272
Solid Intersection (component), 261, 293, 295, 297, 300, 301
Sphere (component), 79, 254, 262, 268, 272, 289, 295
Sphere, component for creating a sphere as an untrimmed surface, 411
Splop, component for projecting geometry onto a target surface, 411, 414
SubD, 150
SubD from Mesh (component), 260, 299
Twist (component), 194, 375
Surface box
Deconstruct Box (node), finding the X, Y, and Z domains of a box, 351
SVG
Inkscape, 524
Symmetry
4-fold Symmetry, 159, 165, 166, 169
n-fold Symmetry, 166

T

Tessellation, 149
Transform
Maelstrom, component that swirls curves, 277
Maelstrom (node), swirls curves, 179
Mirror (component), 160, 166, 173, 180, 277
Move (component), 42, 108, 115, 121, 151, 160, 180, 221, 251, 273, 276, 293, 314, 391, 474, 510, 512
Rotate (component), 44, 155, 233, 375
4-fold Symmetry, 159, 165, 166, 169
n-fold Symmetry, 166
Rotate (node), 155
Rotate 3D (component), 360, 463
Scale, component for scaling geometry uniformly, 106, 136, 260, 299, 372, 417, 509

Scale NU, component for scaling geometry non-uniformly (a different amount along each dimension), 45, 66, 111, 151, 231
Scale NU (node), non-uniform scaling, 150
Scale (node), uniform scaling, 214
TriRemesh
component for converting mesh faces to equilateral triangles, 281, 283, 341
Turtle graphics, 426

U

Unify Mesh
component that ensures the normals of a mesh are consistent, 369, 394, 407, 410
Unitize (Normalize)
making a vector's length 1, 292
Unit vector
a vector with length = 1, 358
User Object
baking
turning GH preview into real geometry in Rhino, 48, 94
an object installed to a copy of GH that behaves like a custom component, 47

V

Vector
Deconstruct, 393
Deconstruct, component for separating a point or vector's X,Y,Z coordinates, 238
Deconstruct (node), separating the X, Y, and Z components, 353
normalize (*see* Unitize)
normal vector, a vector that points away from and is perpendicular to a surface, 357
an object for specifying a direction in 2D (as (X, Y)) or 3D (as (X, Y, Z)), 39, 101, 251
Reverse, component for flipping a vector's direction, 121
unitize, making a vector's length 1, 292
unit vector, a vector with length = 1, 358
Vector 2Pt, component for creating a vector from two points, 120, 292
Vector Display, component for drawing a vector in the viewport, 121
Vector Display Ex, component for drawing a vector in the viewport, 121
View
named view, 110
panning (RMB), 112
zooming in/out (mouse wheel), 112
Viewport, 121
Voronoi diagram, 261
Voxelize, 227, 253

W, X

Weave
component for combining two lists based on a pattern, 512
Weaverbird (GH plugin), 347
Weaverbird's Offset Mesh, component for deriving a scaled (up or down) mesh that is equidistant everywhere along the surface from the original, 323, 341, 369

Weaving
types of weaves
basket, 307
Triaxial, 307
Wire
connection between components, 4, 114
Display (GH editor top menu bar)
Draw Fancy Wires, shows line type, 41
Hidden, 114
line type
double dashed line, 40
double solid line, 40
single solid line, 40
organizing with Relay (component), 13, 114

Y, Z

*.yak
file extension of Rhino package for publishing plugins or commands to the Yak server, 538
Yak
Rhino Package Server, 519