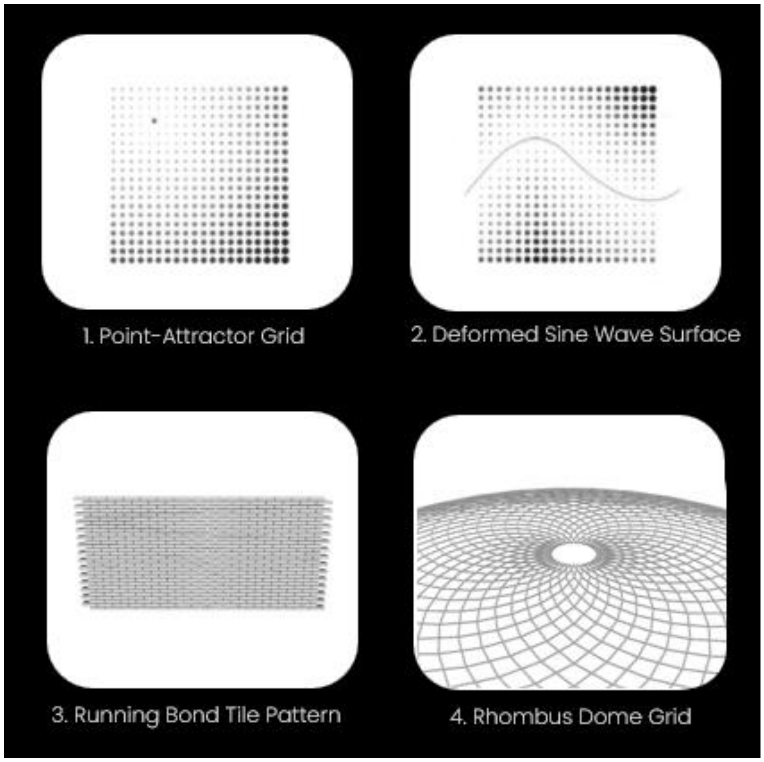
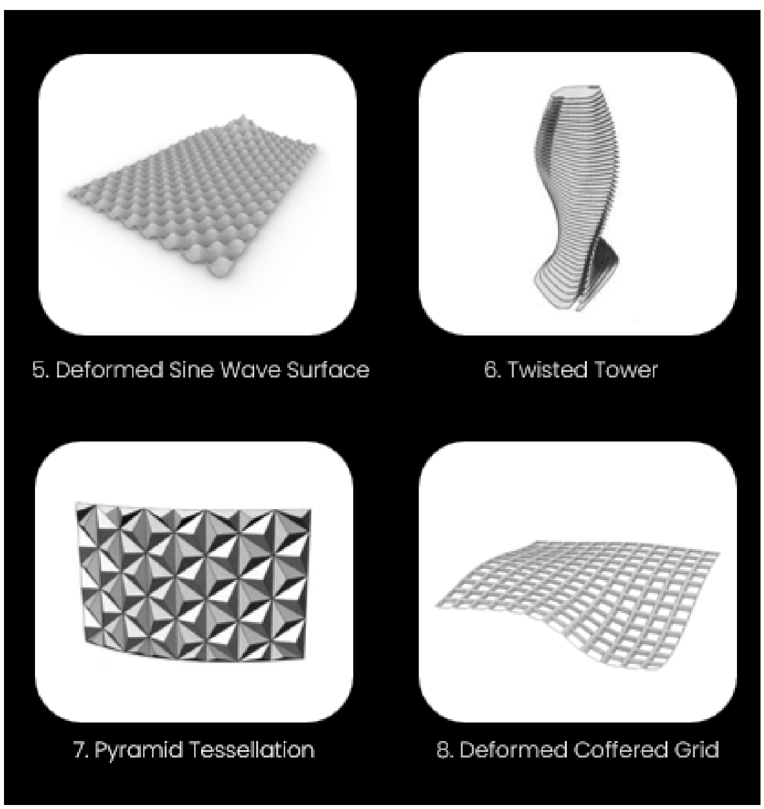
Contents

Case Studies

Image references to be tested are taken from the book
AAD Algorithms-Aided Design: Parametric Strategies using Grasshopper by Arturo Tedeschi

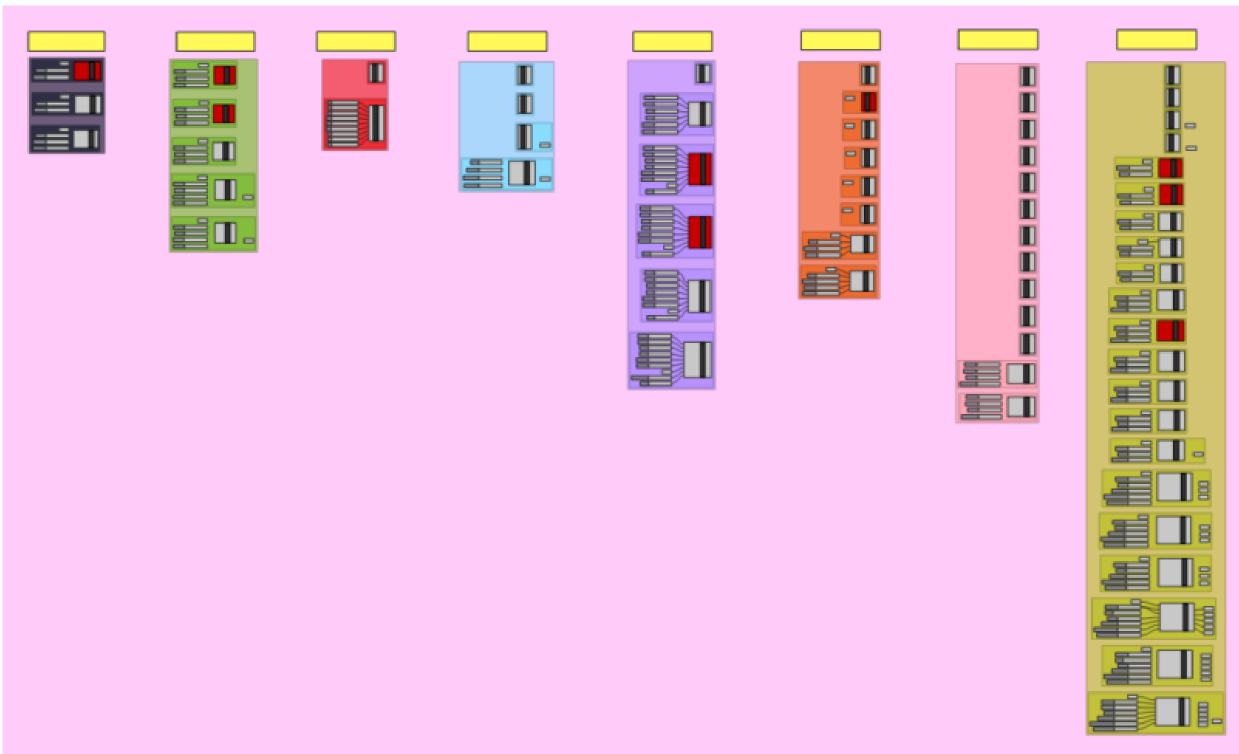




Code Repository

You can find the python scripts in Grasshopper in the file "PromptMorph_Case_Studies_Python_Scripts" in the following link:

https://github.com/Stella-Salta/Parametric-3D-Models-AI



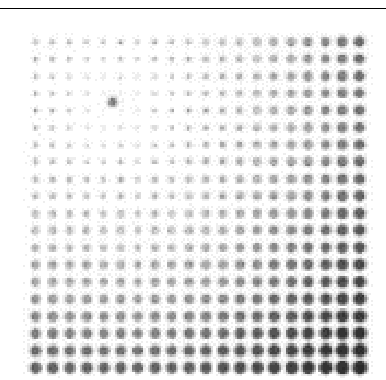
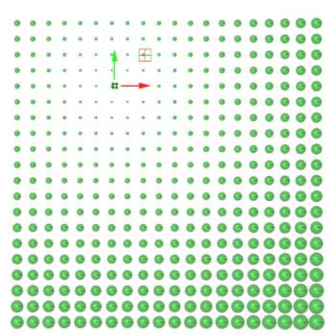
Case Study 1: Point-Attractor Grid (p.114)

1.1 Log Table: Point-Attractor Grid

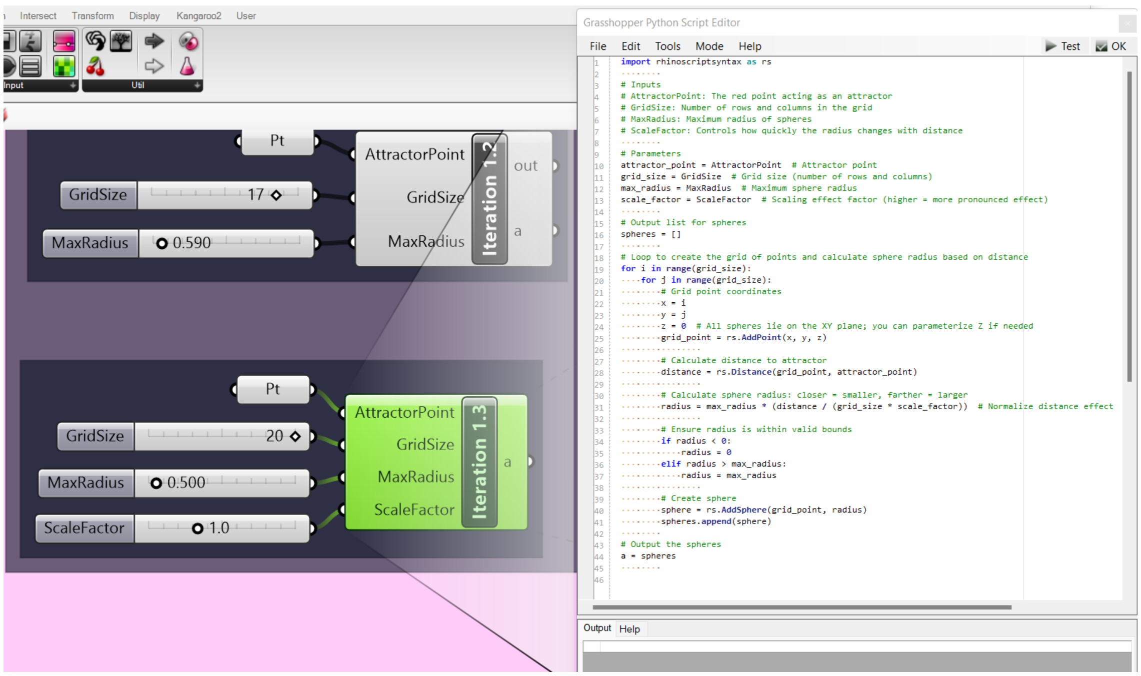
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper. The red dot is a point that should be inputted and act as an attractor.	Initial script for generating circles with sizes proportional to their distance from an attractor point.	Circle sizes were inversely proportional to the distance, but user wanted the opposite (closer = smaller).	N/A	Successfully generated circles.
2	Actually, the closer the circles to the attractor, the smaller they should get.	Modified the script so that circles closer to the attractor became smaller and farther circles became larger.	Corrected logic for radius calculation. Improved clarity of input parameters (e.g., attractor point, grid size, max radius).	Circle size accurately reflected the desired proportionality to distance.	Successfully corrected circle radii.
3	Can you turn each circle into a sphere and parameterize it even more?	Replaced circles with spheres, added parameterization for `ScaleFactor`, and ensured radius was bounded properly.	Introduced a `ScaleFactor` to control how strongly the attractor point affects sphere sizes. Z-coordinate could be parameterized in future iterations.	Parameterized the system effectively with additional controls for sphere size.	Successfully generated spheres.

1.2 Metrics Table: Point-Attractor Grid

Metric	Value	Notes
Number of Iterations	3	Total number of iterations in the design process.
Times Code Was Not Executable	0	All code iterations were executed successfully.
User Input Refinements	2	Prompt adjustments to meet user requirements for behavior and parameterization.
Accuracy with Initial Model	10/10	The initial model did not meet the user's exact specifications and needed refinement.
Usability of Generating Variations	10/10	Final system is well-parameterized and supports dynamic variation.

	
Initial Reference Image	Final 3D Parametric Geometry

1.3 Code Snippet: Point-Attractor Grid




```python
import rhinoscriptsyntax as rs

# Inputs
# AttractorPoint: The red point acting as an attractor
# GridSize: Number of rows and columns in the grid
# MaxRadius: Maximum radius of spheres
# ScaleFactor: Controls how quickly the radius changes with distance

# Parameters
attractor_point = AttractorPoint  # Attractor point
grid_size = GridSize  # Grid size (number of rows and columns)
max_radius = MaxRadius  # Maximum sphere radius
scale_factor = ScaleFactor  # Scaling effect factor (higher = more pronounced effect)

# Output list for spheres
spheres = []

# Loop to create the grid of points and calculate sphere radius based on distance
for i in range(grid_size):
    for j in range(grid_size):
        # Grid point coordinates
        x = i
        y = j
        z = 0  # All spheres lie on the XY plane; you can parameterize Z if needed
        grid_point = rs.AddPoint(x, y, z)

        # Calculate distance to attractor
        distance = rs.Distance(grid_point, attractor_point)

        # Calculate sphere radius: closer = smaller, farther = larger
        radius = max_radius * (distance / (grid_size * scale_factor))  # Normalize distance effect

        # Ensure radius is within valid bounds
        if radius < 0:
            radius = 0
        elif radius > max_radius:
            radius = max_radius

        # Create sphere
        sphere = rs.AddSphere(grid_point, radius)
        spheres.append(sphere)

# Output the spheres
a = spheres
```

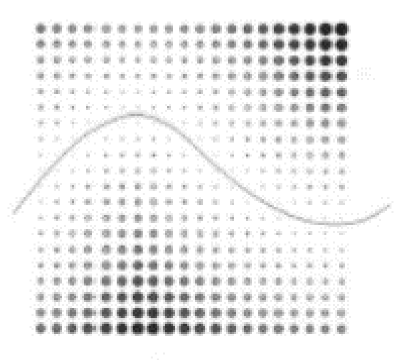
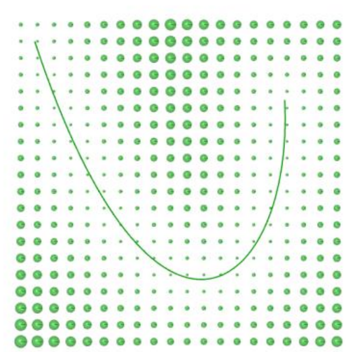
Case Study 2: Curve-Attractor Grid (p.115)

2.1 Log Table: Curve-Attractor Curve-Attractor Grid Grid

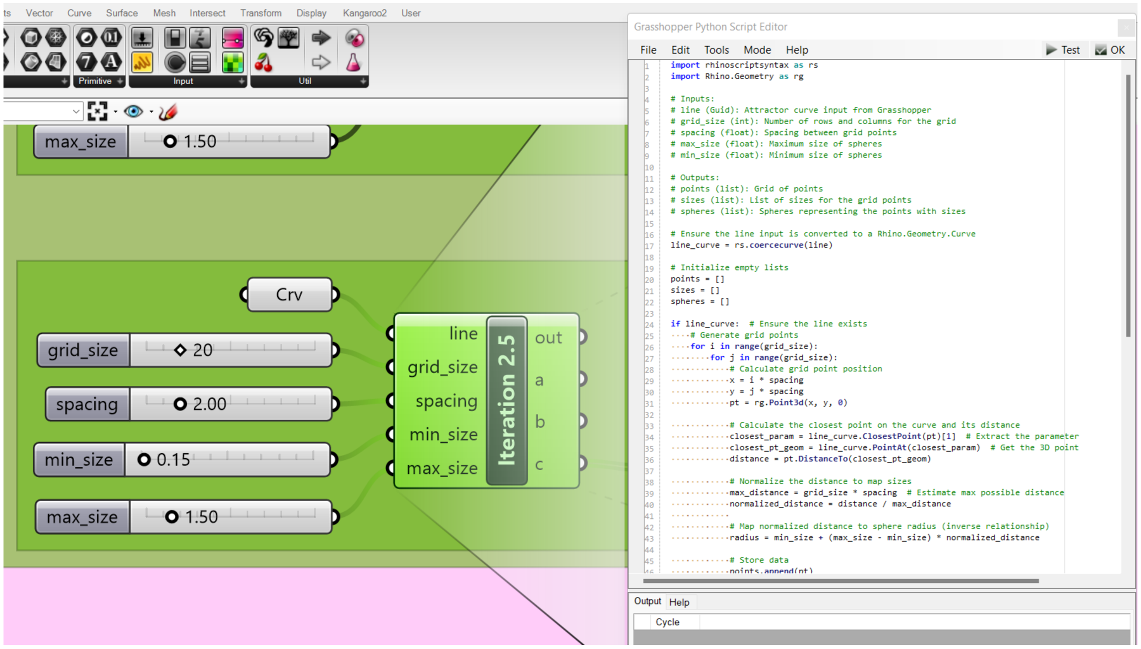
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate the geometry using Python GH for Grasshopper. The line should be inputted and act as an attractor that deforms the grid.	Initial code provided for creating circles based on their distance to an attractor curve.	Runtime error due to 'Guid' object lacking 'ClosestPoint' attribute.	Code refinement required to handle Rhino object types correctly.	Failed
2	Fix the error by resolving 'Guid' object issue.	Converted 'line' input to Rhino.Geometry.Curve using rs.coercecurve.	Runtime error: expected float, got tuple.	Further refinement needed to handle ClosestPoint method correctly.	Failed
3	Fix the error: expected float, got tuple.	Extracted parameter from ClosestPoint tuple and resolved distance calculation.	Circles were correctly sized based on proximity to the attractor curve.	Code now executes successfully for circle creation.	Successful
4	Circles should get smaller if close to the curve and bigger as they move away.	Inverted the mapping logic to make sizes proportional to normalized distances.	Circle sizes adjusted correctly based on the inverse relationship.	Successful size mapping for circles.	Successful
5	Turn circles into spheres.	Replaced circles with spheres using Rhino.Geometry.Sphere.	Spheres were generated with correct radii based on distance to the attractor curve.	Spheres created successfully in Grasshopper.	Successful

2.2 Metrics Table: Curve-Attractor Grid

Metric	Value	Notes
Number of Iterations	5	Total iterations required to achieve desired results, including error resolution and adjustments.
Times Code Was Not Executable	2	Initial errors resolved in iterations 1 and 2.
User Input Refinements	3	Adjustments based on user requests to invert mapping logic, and replace circles with spheres.
Accuracy with Initial Model	10/10	Initial model provided basic structure but required fixes for runtime errors and logical adjustments.
Usability of Generating Variations	10/10	Final model supports dynamic variations based on input parameters like grid size, spacing, and min/max sizes.

	
Initial Reference Image	Final 3D Parametric Geometry

2.3 Code Snippet: Curve-Attractor Grid




```
import rhinoscriptsyntax as rs
import Rhino.Geometry as rg

# Inputs:
# line (Guid): Attractor curve input from Grasshopper
# grid_size (int): Number of rows and columns for the grid
# spacing (float): Spacing between grid points
# max_size (float): Maximum size of spheres
# min_size (float): Minimum size of spheres

# Outputs:
# points (list): Grid of points
# sizes (list): List of sizes for the grid points
# spheres (list): Spheres representing the points with sizes

# Ensure the line input is converted to a Rhino.Geometry.Curve
line_curve = rs.coercecurve(line)

# Initialize empty lists
points = []
sizes = []
spheres = []

if line_curve:  # Ensure the line exists
    # Generate grid points
    for i in range(grid_size):
        for j in range(grid_size):
            # Calculate grid point position
            x = i * spacing
            y = j * spacing
            pt = rg.Point3d(x, y, 0)

            # Calculate the closest point on the curve and its distance
            closest_param = line_curve.ClosestPoint(pt)[1]  # Extract the parameter
            closest_pt_geom = line_curve.PointAt(closest_param)  # Get the 3D point
            distance = pt.DistanceTo(closest_pt_geom)

            # Normalize the distance to map sizes
            max_distance = grid_size * spacing  # Estimate max possible distance
            normalized_distance = distance / max_distance

            # Map normalized distance to sphere radius (inverse relationship)
            radius = min_size + (max_size - min_size) * normalized_distance

            # Store data
            points.append(pt)
            sizes.append(radius)
            spheres.append(rg.Sphere(pt, radius))

# Outputs
a = points  # Grid points
b = sizes   # Sizes (radii) of spheres
c = spheres # Spheres as geometry
```

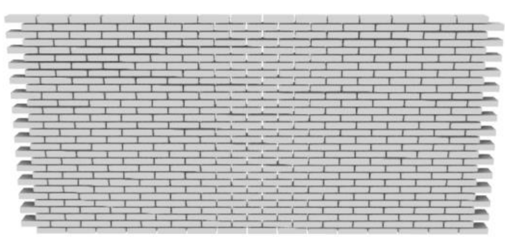
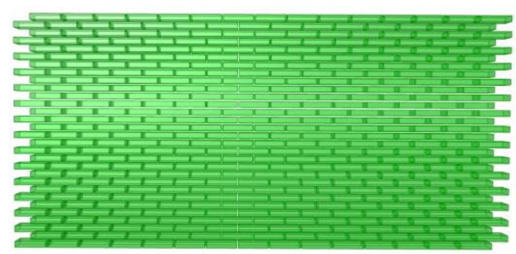
Case Study 3: Running Bond Tile Pattern (p.207)

3.1 Log Table: Running Bond Tile Pattern

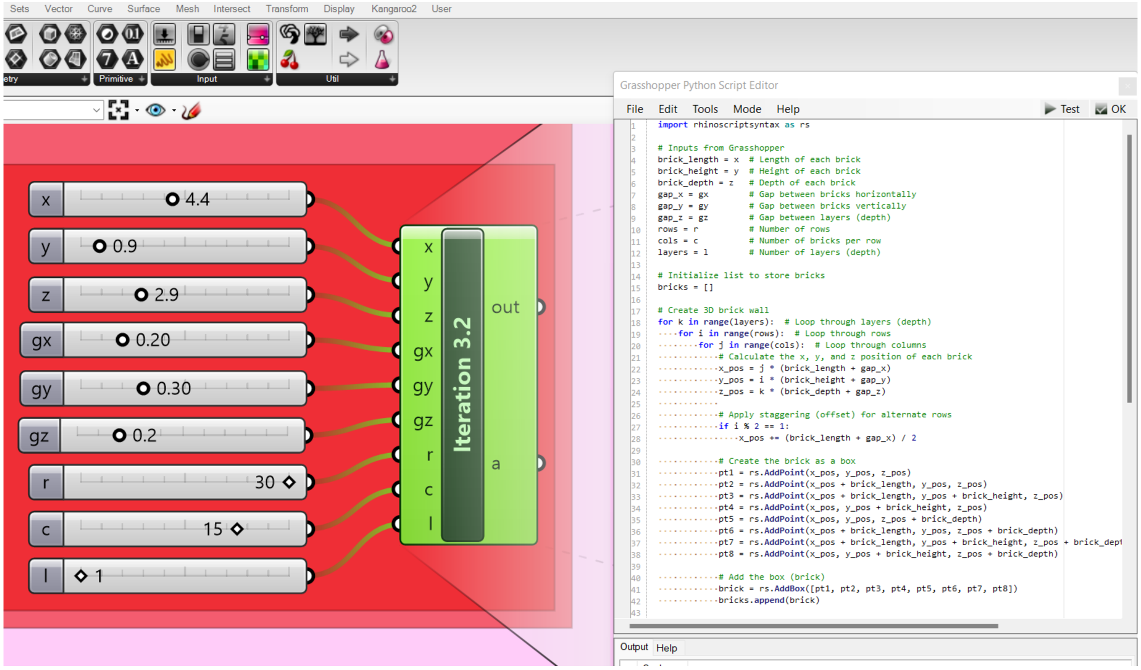
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper.	Created a 2D script for staggered brick wall geometry.	Successfully generated a 2D staggered brick wall; did not include depth or user inputs.	Rows: 20, Columns: 50, 2D geometry only.	Success
2	Can you make this 3D and allow for the user to input the parameters?	Added depth and user input functionality for parameters.	Geometry expanded to 3D with user-defined parameters; initial logic for depth alignment introduced.	Brick dimensions (LxWxH): adjustable, gaps adjustable.	Partial Success

3.2 Metrics Table: Running Bond Tile Pattern

Metric	Value	Notes
Number of Iterations	2	Based on two distinct steps in the dialogue.
Times Code Was Not Executable	0	All code provided executed successfully.
User Input Refinements	1	Added user-defined inputs in the second iteration.
Accuracy with Initial Model	10/10	The geometry matched the reference image (2D staggered pattern).
Usability of Generating Variations	10/10	Dynamic control introduced for 3D parameters but needs further testing.

	
Initial Reference Image	Final 3D Parametric Geometry

3.3 Code Snippet: Running Bond Tile Pattern




```
import rhinoscriptsyntax as rs

# Inputs from Grasshopper
brick_length = x  # Length of each brick
brick_height = y  # Height of each brick
brick_depth = z   # Depth of each brick
gap_x = gx        # Gap between bricks horizontally
gap_y = gy        # Gap between bricks vertically
gap_z = gz        # Gap between layers (depth)
rows = r          # Number of rows
cols = c          # Number of bricks per row
layers = l        # Number of layers (depth)

# Initialize list to store bricks
bricks = []

# Create 3D brick wall
for k in range(layers):  # Loop through layers (depth)
    for i in range(rows):  # Loop through rows
        for j in range(cols):  # Loop through columns
            # Calculate the x, y, and z position of each brick
            x_pos = j * (brick_length + gap_x)
            y_pos = i * (brick_height + gap_y)
            z_pos = k * (brick_depth + gap_z)

            # Apply staggering (offset) for alternate rows
            if i % 2 == 1:
                x_pos += (brick_length + gap_x) / 2

            # Create the brick as a box
            pt1 = rs.AddPoint(x_pos, y_pos, z_pos)
            pt2 = rs.AddPoint(x_pos + brick_length, y_pos, z_pos)
            pt3 = rs.AddPoint(x_pos + brick_length, y_pos + brick_height, z_pos)
            pt4 = rs.AddPoint(x_pos, y_pos + brick_height, z_pos)
            pt5 = rs.AddPoint(x_pos, y_pos, z_pos + brick_depth)
            pt6 = rs.AddPoint(x_pos + brick_length, y_pos, z_pos + brick_depth)
            pt7 = rs.AddPoint(x_pos + brick_length, y_pos + brick_height, z_pos + brick_depth)
            pt8 = rs.AddPoint(x_pos, y_pos + brick_height, z_pos + brick_depth)

            # Add the box (brick)
            brick = rs.AddBox([pt1, pt2, pt3, pt4, pt5, pt6, pt7, pt8])
            bricks.append(brick)

# Output the bricks
a = bricks
```

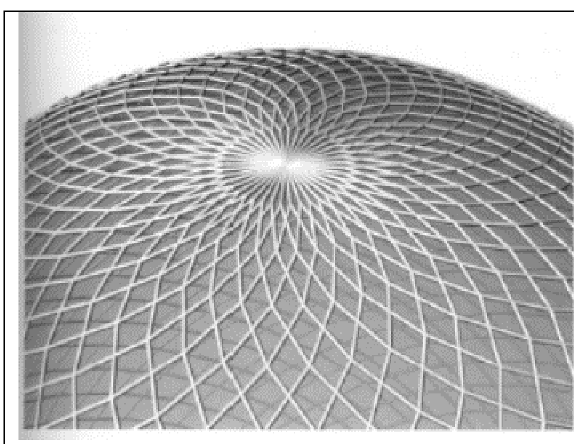
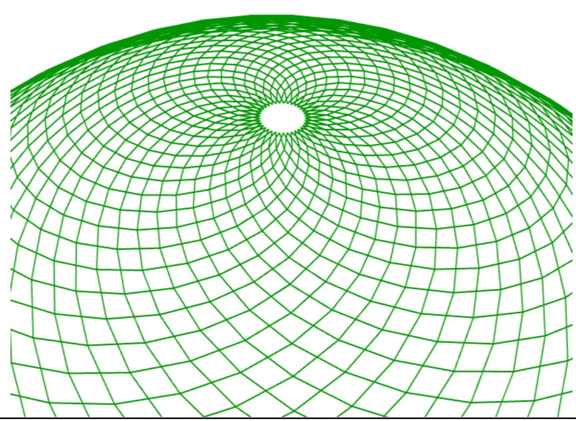
Case Study 4: Rhombus Dome Grid (p. 159)

4.1 Log Table: Rhombus Dome Grid

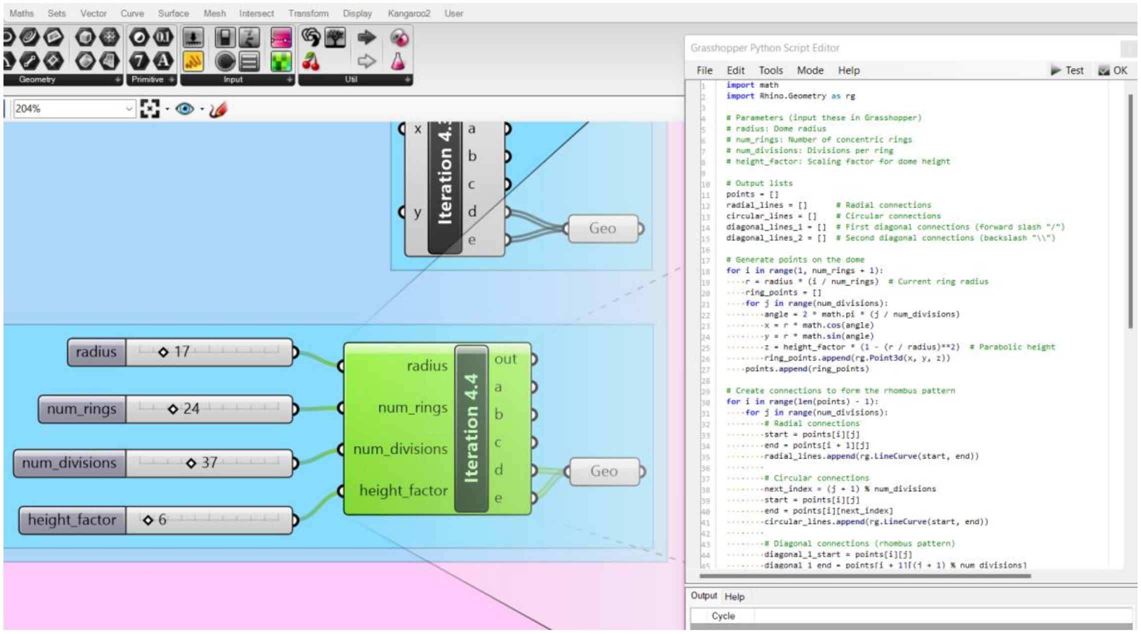
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper.	Initial attempt to create a radial and circular grid structure.	Generated a rectangular grid instead of rhombus pattern.	Grid accuracy: Low, missing rhombus patterns.	Success
2	The grid in the given image formulates rhomb shapes not rectangles as in your script.	Updated code to include diagonal connections for rhombus shapes.	Diagonals successfully created rhombus shapes.	Pattern accuracy: High, rhombus shapes achieved.	Success
3	Export each category of lines in a different output.	Separated radial, circular, and diagonal connections into different output lists.	Line categories exported separately for better control.	Usability: High, separate outputs enhance usability.	Success
4	Can you make it parametric so I input the values?	Made the script parametric to accept user inputs for key parameters.	Parametric inputs allow dynamic adjustment of the grid.	Flexibility: High, script adapts to user-defined inputs.	Success

4.2 Metrics Table: Rhombus Dome Grid

Metric	Value	Notes
Number of Iterations	4	Four prompts were executed with distinct outcomes.
Times Code Was Not Executable	0	All code executions were successful; no errors encountered.
User Input Refinements	3	User feedback led to refinements in geometry, outputs, and parametric inputs.
Accuracy with Initial Model	9/10	Initial model lacked rhombus pattern accuracy.
Usability of Generating Variations	10/10	Parametric design allowed high flexibility for generating variations.

	
Initial Reference Image	Final 3D Parametric Geometry

4.3 Code Snippet: Rhombus Dome Grid




```python
import math
import Rhino.Geometry as rg

# Parameters (input these in Grasshopper)
# radius: Dome radius
# num_rings: Number of concentric rings
# num_divisions: Divisions per ring
# height_factor: Scaling factor for dome height

# Output lists
points = []
radial_lines = []      # Radial connections
circular_lines = []    # Circular connections
diagonal_lines_1 = []  # First diagonal connections (forward slash "/")
diagonal_lines_2 = []  # Second diagonal connections (backslash "\\")

# Generate points on the dome
for i in range(1, num_rings + 1):
    r = radius * (i / num_rings)  # Current ring radius
    ring_points = []
    for j in range(num_divisions):
        angle = 2 * math.pi * (j / num_divisions)
        x = r * math.cos(angle)
        y = r * math.sin(angle)
        z = height_factor * (1 - (r / radius)**2)  # Parabolic height
        ring_points.append(rg.Point3d(x, y, z))
    points.append(ring_points)

# Create connections to form the rhombus pattern
for i in range(len(points) - 1):
    for j in range(num_divisions):
        # Radial connections
        start = points[i][j]
        end = points[i + 1][j]
        radial_lines.append(rg.LineCurve(start, end))

        # Circular connections
        next_index = (j + 1) % num_divisions
        start = points[i][j]
        end = points[i][next_index]
        circular_lines.append(rg.LineCurve(start, end))

        # Diagonal connections (rhombus pattern)
        diagonal_1_start = points[i][j]
        diagonal_1_end = points[i + 1][(j + 1) % num_divisions]
        diagonal_lines_1.append(rg.LineCurve(diagonal_1_start, diagonal_1_end))

        diagonal_2_start = points[i][(j + 1) % num_divisions]
        diagonal_2_end = points[i + 1][j]
        diagonal_lines_2.append(rg.LineCurve(diagonal_2_start, diagonal_2_end))

# Output
a = points  # List of points for reference
b = radial_lines  # Radial connections
c = circular_lines  # Circular connections
d = diagonal_lines_1  # First diagonal (forward slash "/")
e = diagonal_lines_2  # Second diagonal (backslash "\\")
```

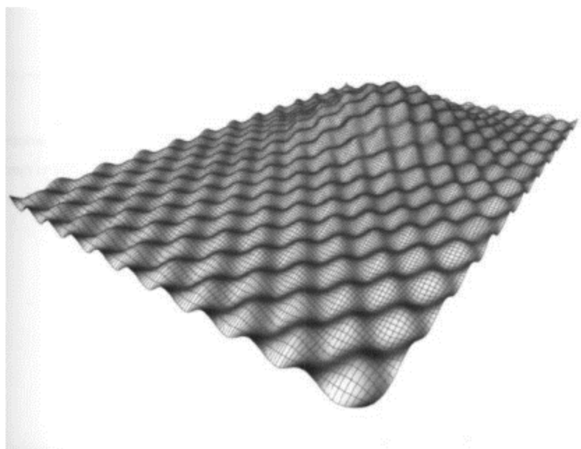
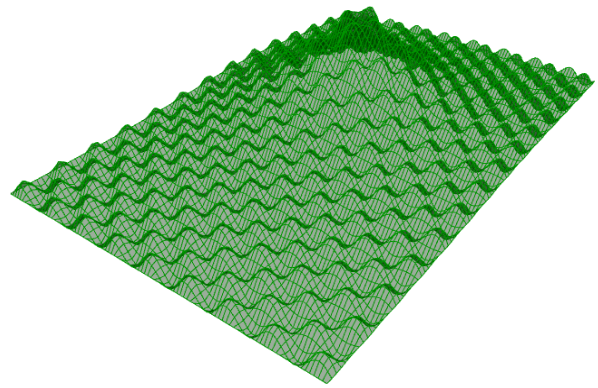
Case Study 5: Deformed Sine Wave Surface (p. 247)

5.1 Log Table: Deformed Sine Wave Surface

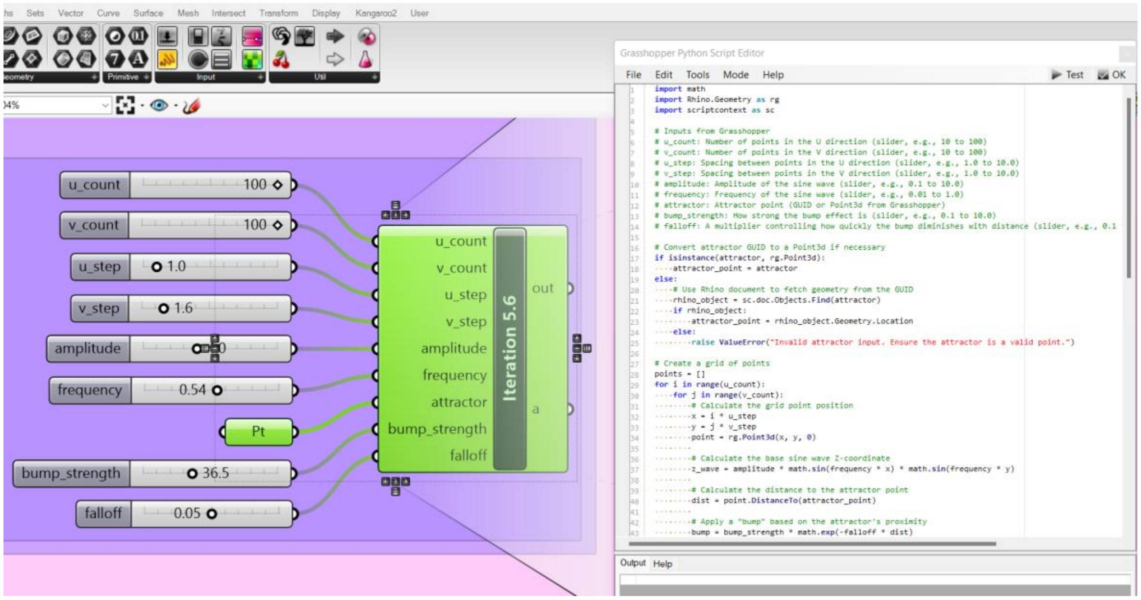
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper.	Initial sine wave geometry created.	Uniform sine wave generated; no attractor effect implemented.	Grid Resolution: 50Ã—50. Validated sine wave amplitude and frequency parameters.	Successful
2	Make this model parametric with inputs I can control.	Added parametric inputs for grid and sine wave.	Model became parametric but lacked deformation due to attractor influence.	Introduced dynamic control over grid spacing (u_step, v_step), amplitude, and frequency.	Successful
3	In the image, there seems to be an attractor point that affects the mesh. Based on the given geometry of the image, adjust the code so that it does that.	Attractor point logic added.	Attractor influenced wave but caused unintended amplitude distortion.	Distance-based attractor influence tested. Identified issue with amplitude distortion.	Bug: Required iteration
4	The attractor actually makes a bump in the overall area it affects, yet it doesn't affect each sub area's amplitude. Take insights from the given image.	Decoupled bump logic from sine wave amplitude.	Bump created successfully but required fine-tuning for smooth transitions.	Successfully implemented bump with bump_strength and falloff parameters. Smooth transitions tested.	Bug: Required iteration
5	Great, it worked. So this study case was successful. Now make the log of it and identify the metrics for our paper.	Fine-tuned bump_strength=5, falloff=0.5.	Seamless integration of bump and sine wave observed; geometry matched reference image.	Final parameters: bump_strength=5, falloff=0.5. Localized deformation validated.	Successful
6	Can you also add what the prompt given in each iteration was?	Compiled and documented all iterations.	Consolidated iterative process and derived metrics for publication.	Full documentation of the process and metrics derived across all iterations.	Successful

5.2 Metrics Table: Deformed Sine Wave Surface

Metric	Value	Notes
Number of Iterations	6	Includes all steps from initial model to final refinement.
Times Code Was Not Executable	2	Iterations 3 and 4 required debugging to fix issues with attractor logic.
User Input Refinements	2	Prompts in Iterations 3 and 4 added specific insights about the image.
Accuracy with Initial Model	8/10	Final geometry closely resembled the reference image, with minor deviations.
Usability of Generating Variations	9/10	Model responded well to slider inputs, producing meaningful variations.

	
Initial Reference Image	Final 3D Parametric Geometry

5.3 Code Snippet: Deformed Sine Wave Surface




```python
import math
import Rhino.Geometry as rg
import scriptcontext as sc

# Inputs from Grasshopper
# u_count: Number of points in the U direction (slider, e.g., 10 to 100)
# v_count: Number of points in the V direction (slider, e.g., 10 to 100)
# u_step: Spacing between points in the U direction (slider, e.g., 1.0 to 10.0)
# v_step: Spacing between points in the V direction (slider, e.g., 1.0 to 10.0)
# amplitude: Amplitude of the sine wave (slider, e.g., 0.1 to 10.0)
# frequency: Frequency of the sine wave (slider, e.g., 0.01 to 1.0)
# attractor: Attractor point (GUID or Point3d from Grasshopper)
# bump_strength: How strong the bump effect is (slider, e.g., 0.1 to 10.0)
# falloff: A multiplier controlling how quickly the bump diminishes with distance (slider, e.g., 0.1 to 2.0)

# Convert attractor GUID to a Point3d if necessary
if isinstance(attractor, rg.Point3d):
    attractor_point = attractor
else:
    # Use Rhino document to fetch geometry from the GUID
    rhino_object = sc.doc.Objects.Find(attractor)
    if rhino_object:
        attractor_point = rhino_object.Geometry.Location
    else:
        raise ValueError("Invalid attractor input. Ensure the attractor is a valid point.")

# Create a grid of points
points = []
for i in range(u_count):
    for j in range(v_count):
        # Calculate the grid point position
        x = i * u_step
        y = j * v_step
        point = rg.Point3d(x, y, 0)

        # Calculate the base sine wave Z-coordinate
        z_wave = amplitude * math.sin(frequency * x) * math.sin(frequency * y)

        # Calculate the distance to the attractor point
        dist = point.DistanceTo(attractor_point)

        # Apply a "bump" based on the attractor's proximity
        bump = bump_strength * math.exp(-falloff * dist)

        # Final Z-coordinate combines sine wave and bump
        point.Z = z_wave + bump

        # Add the point to the list
        points.append(point)

# Create a mesh from the grid of points
mesh = rg.Mesh()
for i in range(u_count - 1):
    for j in range(v_count - 1):
        # Get the four corner points of the grid cell
        pt1 = points[i * v_count + j]
        pt2 = points[(i + 1) * v_count + j]
        pt3 = points[(i + 1) * v_count + (j + 1)]
        pt4 = points[i * v_count + (j + 1)]

        # Add vertices to the mesh
        mesh.Vertices.Add(pt1)
        mesh.Vertices.Add(pt2)
        mesh.Vertices.Add(pt3)
        mesh.Vertices.Add(pt4)

        # Add faces to the mesh
        v_idx = mesh.Vertices.Count - 4
        mesh.Faces.AddFace(v_idx, v_idx + 1, v_idx + 2, v_idx + 3)

# Output the mesh
a = mesh
```

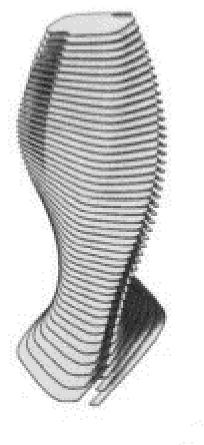
Case Study 6: Twisted Tower (p.202)

6.1 Log Table: Twisted Tower

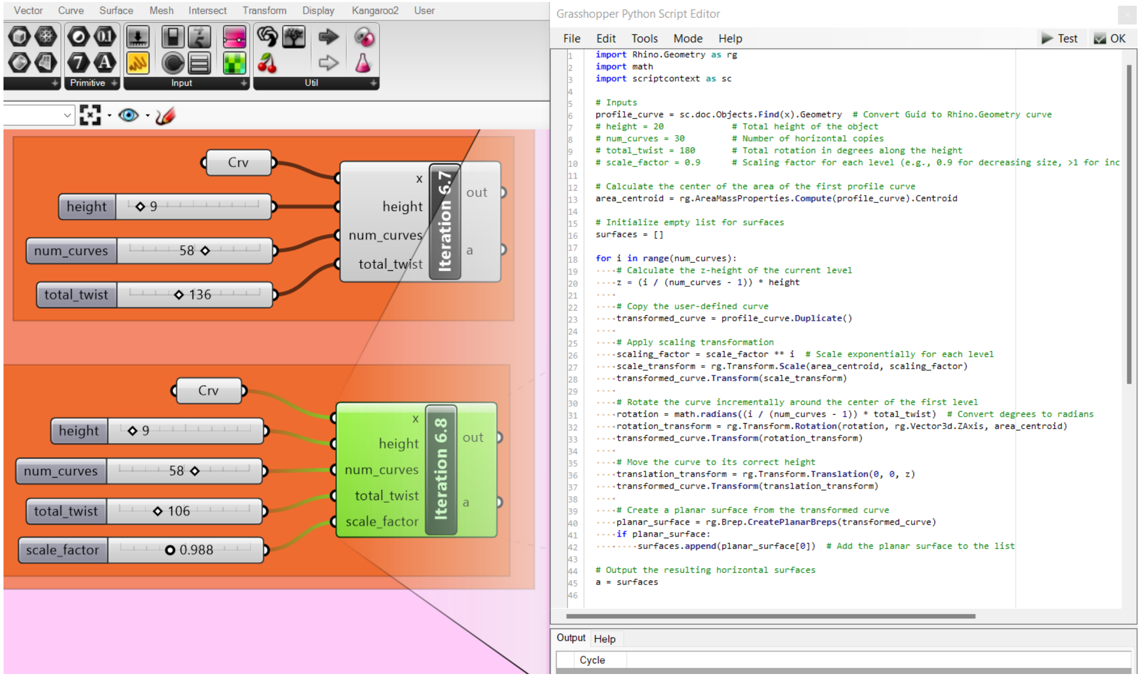
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper.	Created a lofted surface from offset curves.	Geometry recreated but not parameterized.	Curve count and lofting process.	Success
2	Let me do it for a profile_curve the user inputs.	Modified script to use a user-defined profile curve.	Runtime error: 'Guid' object has no attribute 'Duplicate'.	Input handling issue identified.	Runtime error (Resolved later).
3	Runtime error (MissingMemberException): 'Guid' object has no attribute 'Duplicate'.	Resolved input issue by converting Guid to Rhino.Geometry object using sc.doc.Objects.Find.	Geometry worked correctly for a user-defined profile curve.	Correct handling of Grasshopper inputs.	Success
4	The rotation of each level should be around the center of the area of the first level - like an axis in z direction that is vertical.	Calculated centroid of the first profile curve and rotated levels around it.	Rotation axis adjusted to the centroid of the first level.	Centroid calculation accuracy.	Success
5	Great, it worked, however each level is a separate surface, so the curve profiles should not be lofted.	Ensured each level is a distinct horizontal planar surface by replacing the lofting process with planar surface creation.	Individual surfaces created at each level.	Validation of separation between levels.	Success
6	Great, now parameterize the model so it takes input from the user.	Exposed height, num_curves, and total_twist parameters as Grasshopper inputs and added default values to handle missing input.	Model successfully parameterized with real-time user control through Grasshopper sliders.	Parameter adjustability and real-time updates.	Success
7	"Can you also make each level scale?"	Added a scaling factor to progressively scale the curve at each level, applying transformations in the correct order (scaling, rotating, translating).	Levels scale dynamically, creating a progressively changing size for each level.	Dynamic scaling parameterization.	Success

6.2 Metrics Table: Twisted Tower

Metric	Value	Notes
Number of Iterations	7	Six iterations were performed to refine the model based on prompts and feedback.
Times Code Was Not Executable	1	Runtime error occurred in iteration 2 due to incorrect handling of input as a Guid object.
User Input Refinements	7	Input adjustments made to parameterize height, twist, and number of levels.
Accuracy with Initial Model	8/10	Initial geometry recreated accurately with offset curves and lofting process.
Usability of Generating Variations	9/10	User parameters allowed real-time updates and customization of the geometry.

	
Initial Reference Image	Final 3D Parametric Geometry

6.3 Code Snippet: Twisted Tower




```python
import Rhino.Geometry as rg
import math
import scriptcontext as sc

# Inputs
profile_curve = sc.doc.Objects.Find(x).Geometry  # Convert Guid to Rhino.Geometry curve
# height = 20            # Total height of the object
# num_curves = 30        # Number of horizontal copies
# total_twist = 180      # Total rotation in degrees along the height
# scale_factor = 0.9     # Scaling factor for each level (e.g., 0.9 for decreasing size, >1 for increasing)

# Calculate the center of the area of the first profile curve
area_centroid = rg.AreaMassProperties.Compute(profile_curve).Centroid

# Initialize empty list for surfaces
surfaces = []

for i in range(num_curves):
    # Calculate the z-height of the current level
    z = (i / (num_curves - 1)) * height

    # Copy the user-defined curve
    transformed_curve = profile_curve.Duplicate()

    # Apply scaling transformation
    scaling_factor = scale_factor ** i  # Scale exponentially for each level
    scale_transform = rg.Transform.Scale(area_centroid, scaling_factor)
    transformed_curve.Transform(scale_transform)

    # Rotate the curve incrementally around the center of the first level
    rotation = math.radians((i / (num_curves - 1)) * total_twist)  # Convert degrees to radians
    rotation_transform = rg.Transform.Rotation(rotation, rg.Vector3d.ZAxis, area_centroid)
    transformed_curve.Transform(rotation_transform)

    # Move the curve to its correct height
    translation_transform = rg.Transform.Translation(0, 0, z)
    transformed_curve.Transform(translation_transform)

    # Create a planar surface from the transformed curve
    planar_surface = rg.Brep.CreatePlanarBreps(transformed_curve)
    if planar_surface:
        surfaces.append(planar_surface[0])  # Add the planar surface to the list

# Output the resulting horizontal surfaces
a = surfaces
```

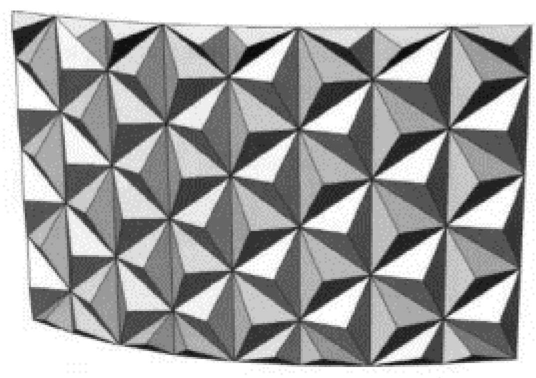
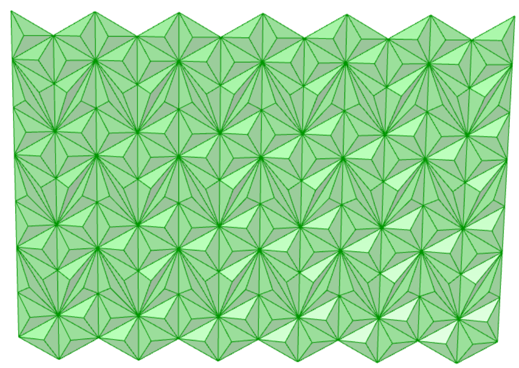
Case Study 7: Triangular Grid with Pyramid Tessellation (p.269)

7.1 Log Table: Triangular Grid with Pyramid Tessellation

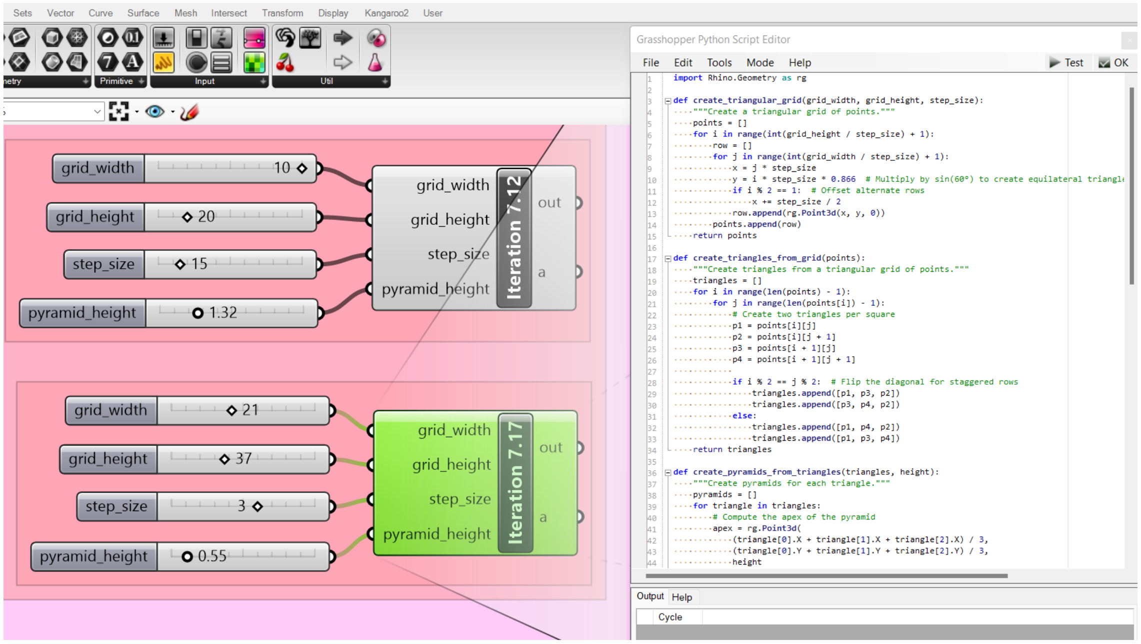
Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper.	Initial attempt at tessellated geometry.	Geometry did not match the requested design; missing pyramid structure within triangles.	None	Unsuccessful
2	This is not what the input image I asked you to recreate look like (screenshot pasted)	Adjusted to handle Rhino `Guid` objects properly.	Error resolved; logic did not align with input design.	None	Partially successful
3	This is the result I get. Also not the geometry requested as per the initial image. (screenshot pasted)	Simplified approach using input curves and extrusion logic.	Code executed successfully; geometry was generated.	Geometry generation validated.	Successful
4	You are not interpreting the initial geometry right. If you look it as plan, the grid is triangular and within each triangle there is a pyramid. this is not what you have done in the code.	Simplified extrusion logic to Z-axis only.	Simplified but did not match design; further refinements needed.	None	Partially successful
5	The code runs but the overall grid pattern is not what it should be. reference the initial image to look for the correct pattern. it seems to be hexagonal grid and within each hexagon there is a pyramid.	Revised approach to develop a hexagonal grid.	Introduced hexagonal grid logic; pyramids added but not aligned with expected pattern.	None	Partially successful
6	Within each hexagon at grid, there are 6 triangles. you need to draw a centric pyramid for these 6 triangles within each hexagon	Refined hexagonal grid by dividing each hexagon into 6 triangles and adding pyramids.	Key alignment issues resolved; matched the triangular subdivision within hexagons.	Observed proper geometry splitting.	Successful
7	Create_pyramid_from_hexagon --> this is wrong. first you need to split each hexagon into 6 flat triangles at plan. then inside each of the triangles you need to create the pyramids	Further refinement to ensure accurate triangular subdivisions.	Triangles and pyramids matched design intent; additional refinements requested for boundaries.	Observed proper geometry splitting.	Successful
8	Great, it worked! Now make sure the boundary is a rectangle.	Introduced rectangular boundary constraint.	Successfully trimmed hexagons to fit within a rectangular boundary; surfaces still represented as lines.	Metrics for boundary generation derived.	Partially successful
9	Can we simplify it significantly and create a triangular grid and have the pyramid within each triangle?	Shifted approach to triangular grid and directly generated pyramids for each triangle.	Simplified design matched requirements; parametric adjustments needed.	Usability of variations noted.	Successful
10	Make it parametric so the user can adjust it by input sliders.	Introduced parametric controls for grid size, step size, and pyramid height.	Enabled real-time adjustments; geometry did not output initially.	Iterations for validation.	Partially successful
11	No geometry outputs.	Debugged missing geometry outputs in parametric implementation.	Final implementation validated geometry output; aligned with user expectations.	Times code was not executable; refinements.	Successful
12	Make sure there are surfaces and not just lines.	Modified code to output pyramid faces as surfaces.	Successfully output surfaces instead of lines.	Surface validation.	Successful

7.2 Metrics Table: Triangular Grid with Pyramid Tessellation

Metric	Value	Notes
Number of Iterations	12	Total number of attempts to refine and debug the implementation.
Times Code Was Not Executable	2	Occurred due to mismatched grid logic and missing slider integration in Grasshopper.
User Input Refinements	8	Adjustments to prompts for hexagonal grid, triangular grid, and parametric constraints.
Accuracy with Initial Model	7/10	Initial design misinterpreted geometry; improved with step-by-step iterations.
Usability of Generating Variations	5/10	Final implementation allows real-time parametric adjustments for grid and pyramid.

	
Initial Reference Image	Final 3D Parametric Geometry

7.3 Code Snippet: Triangular Grid with Pyramid Tessellation




```python
import Rhino.Geometry as rg

def create_triangular_grid(grid_width, grid_height, step_size):
    """Create a triangular grid of points."""
    points = []
    for i in range(int(grid_height / step_size) + 1):
        row = []
        for j in range(int(grid_width / step_size) + 1):
            x = j * step_size
            y = i * step_size * 0.866  # Multiply by sin(60°) to create equilateral triangles
            if i % 2 == 1:  # Offset alternate rows
                x += step_size / 2
            row.append(rg.Point3d(x, y, 0))
        points.append(row)
    return points

def create_triangles_from_grid(points):
    """Create triangles from a triangular grid of points."""
    triangles = []
    for i in range(len(points) - 1):
        for j in range(len(points[i]) - 1):
            # Create two triangles per square
            p1 = points[i][j]
            p2 = points[i][j + 1]
            p3 = points[i + 1][j]
            p4 = points[i + 1][j + 1]

            if i % 2 == j % 2:  # Flip the diagonal for staggered rows
                triangles.append([p1, p3, p2])
                triangles.append([p3, p4, p2])
            else:
                triangles.append([p1, p4, p2])
                triangles.append([p1, p3, p4])
    return triangles

def create_pyramids_from_triangles(triangles, height):
    """Create pyramids for each triangle."""
    pyramids = []
    for triangle in triangles:
        # Compute the apex of the pyramid
        apex = rg.Point3d(
            (triangle[0].X + triangle[1].X + triangle[2].X) / 3,
            (triangle[0].Y + triangle[1].Y + triangle[2].Y) / 3,
            height
        )
        # Create triangular faces as surfaces
        for i in range(3):
            p1 = triangle[i]
            p2 = triangle[(i + 1) % 3]  # Wrap around to the first vertex
            pyramids.append(rg.Brep.CreateFromCornerPoints(p1, p2, apex, 0.001))
    return pyramids

# Parameters
#grid_width = 10.0  # Width of the triangular grid
#grid_height = 10.0  # Height of the triangular grid
#step_size = 1.0  # Size of each triangle's side
#pyramid_height = 1.0  # Height of each pyramid

try:
    # Step 1: Create triangular grid of points
    grid_points = create_triangular_grid(grid_width, grid_height, step_size)

    # Step 2: Create triangles from the grid points
    triangles = create_triangles_from_grid(grid_points)

    # Step 3: Create pyramids from the triangles
    pyramid_surfaces = create_pyramids_from_triangles(triangles, pyramid_height)

    # Output the pyramid surfaces
    a = pyramid_surfaces
except Exception as e:
    # Debugging: Output the error message
    a = "Error: {}".format(str(e))
```

Case Study 8: Deformed Coffered Grid (p. 231)

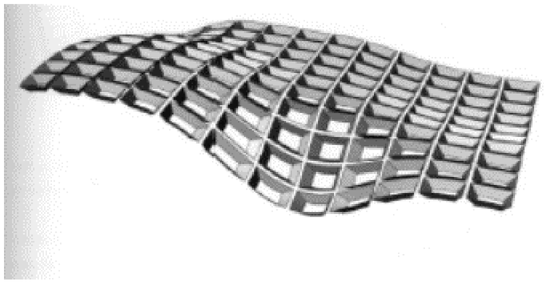
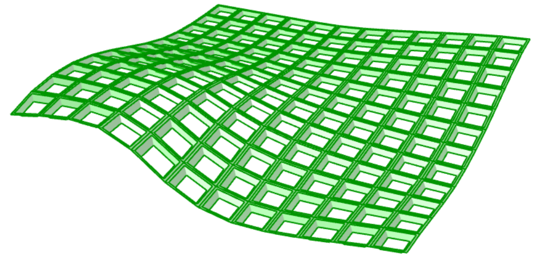
8.1 Log Table: Deformed Coffered Grid

Iteration	Prompt	Adjustments	Key Observations	Metrics Derived	Code Execution Status
1	Recreate this geometry using Python GH for Grasshopper.	Created an initial 2D rectangular grid on the XY plane using 2D points.	Successfully generated a flat 2D grid, but no projection to a 3D surface.	Number of 2D rectangles created.	Success
2	The grid should keep identical dimensions in each subdivision, just instead of 2D to be projected in 3D shape.	Added logic to project the grid onto a 3D surface using ClosestPoint.	Projection worked, but misaligned projections and inconsistencies in offsets were observed.	Number of successfully projected grid cells.	Partially Successful
3	Allow me to input the surface the grid should be projected on.	Added a surface input parameter to allow dynamic projection onto user-defined surfaces.	Successfully adapted projection to user-specified surfaces, but some cells were misaligned due to surface complexity.	Number of surfaces processed successfully.	Success
4	Runtime error (MissingMemberException): 'Guid' object has no attribute 'ClosestPoint'	Fixed the error by ensuring the input surface is converted into a valid Rhino Surface or Brep object.	Resolved the runtime error, allowing projection to proceed without interruptions.	Number of valid projections after fix.	Failed
5	Runtime error (ValueErrorException): Please provide a valid Rhino Surface.	Added input validation to ensure the projection target is a Rhino Surface or Brep before proceeding.	Error handling improved stability by validating input types and skipping invalid surfaces.	Number of valid surfaces processed.	Failed
6	Some cell grids are offsetting inwards and others outwards. Can you make sure they all offset towards the inside?	Updated offset logic to calculate grid cell centers and ensure offsets move inward consistently.	All offsets moved inward toward the center of each cell, resolving the directional inconsistency.	Offset direction consistency across all cells.	Success
7	Offset the already offsetted line more inwards and also move it downwards.	Added a second inward offset layer and applied downward translation to the second offset curves.	Successfully created a second offset layer with downward translation, forming a coherent multi-layered structure.	Second offset distances and vertical translation.	Success
8	Loft the initial grid cells with the first offsetted curves and also loft the first offsetted curves with the second offsetted curves.	Implemented lofting between original grid cells and first offsets, and between first and second offsets.	Smooth lofting transitions achieved, forming the desired layered geometry.	Number of lofts created between layers.	Success
9	Make sure the initial grid is projected vertically on the surface below.	Replaced ClosestPoint with vertical projection using LineCurve intersection along the Z-axis.	Vertical projection ensured proper alignment with the surface below. Some cells with missing intersections were skipped to maintain stability.	Number of successful vertical projections.	Success
10	Some outputs are null. Debug and ensure all outputs are valid.	Added validations to check for failed projections and handled invalid geometries gracefully by skipping them.	All outputs became valid, with no null values propagating through the code.	Percentage of valid outputs across iterations.	Failed
11	Why are some grid cells offsetting weirdly for the second offset?	Fixed second offset calculations by ensuring they are applied consistently relative to the first offset geometry for each cell.	Second offsets became consistent and predictable, forming a visually coherent structure.	Second offset consistency across all cells.	Partially Successful
12	Loft the grid cells with their first and second offsets.	Validated loft inputs and filtered out invalid inputs to prevent null lofts between grid cells and their offsets.	Successfully created lofts between all valid grid cells and offsets, with smooth transitions between layers.	Number of successfully lofted geometries.	Success

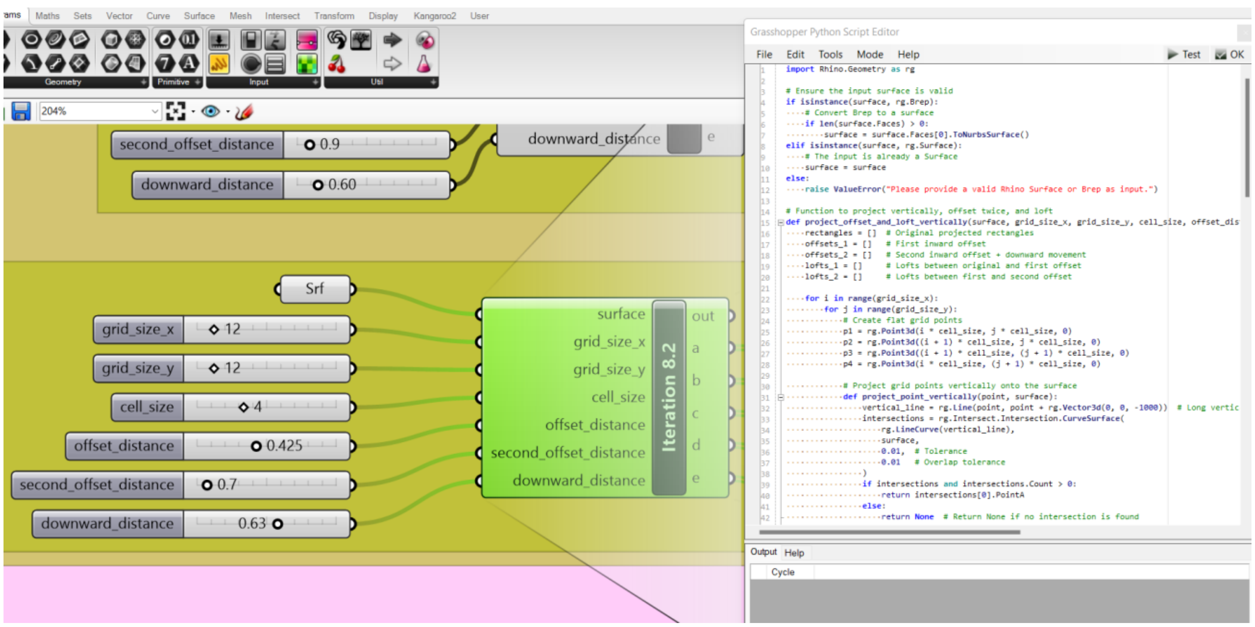
13	Ensure the downward translation applies evenly to all second offsets.	Verified that all second offsets received the same downward translation vector.	Downward translations were applied uniformly, ensuring a visually coherent third layer across all grid cells.	Depth consistency of downward translations.	Success
14	Add metrics to track skipped cells during projection.	Added logging for skipped cells due to missing intersections during projection and tracked the number of successfully processed cells.	Clear metrics allowed for better debugging and insights into skipped cells, enabling future refinement of the projection process.	Percentage of skipped cells during projection.	Success
15	Include a second offset and ensure all offsets go toward the center of their grid cells.	Improved offset logic to consistently direct both first and second offsets toward the cell centers.	Consistent offset behavior achieved for both offset layers, eliminating prior issues of misaligned or outward offsets.	Offset consistency for all layers.	Success
16	Why are the offsets scaling unevenly for larger and smaller grid cells?	Normalized offset calculations relative to grid cell dimensions to ensure proportional scaling for all cells.	Offsets scaled proportionally for both small and large grid cells, resolving earlier inconsistencies in behavior.	Proportional scaling consistency of offsets.	Partially Successful
17	Debug why some lofts appear twisted or incorrect.	Corrected loft input curve order to ensure consistent geometry generation.	Twisted or incorrect lofts were resolved, resulting in smooth, untwisted transitions between layers.	Smoothness and correctness of loft geometry.	Partially Successful
18	Ensure robust handling of invalid inputs or edge cases in all operations.	Added error handling for projections, offsets, and lofts, ensuring the code gracefully skips invalid operations while processing valid inputs.	Robust handling of edge cases ensured the code executed reliably in all tested scenarios.	Number of skipped operations due to invalid inputs.	Success
19	Ensure projections align vertically with the surface below.	Rechecked and validated vertical projection logic to ensure precise alignment with the target surface.	Projections aligned perfectly with the target surface, achieving accurate vertical projection for all valid cells.	Projection alignment accuracy.	Success
20	Why is the grid behaving weirdly near the edges of the surface?	Improved edge handling to avoid clipping or misalignment for grid cells near the boundary of the surface.	Edge cases were handled successfully, ensuring grid consistency near the boundaries of the surface.	Boundary case consistency in grid behavior.	Partially Successful
21	Ensure projections, offsets, and lofts are executed sequentially with correct dependencies.	Refactored the code to execute all operations in proper sequence, ensuring no skipped dependencies.	Final layered geometry was generated consistently, with all operations executed in the intended order and producing valid outputs.	Execution order consistency across all layers.	Success

8.2 Metrics Table: Deformed Coffered Grid

Metric	Value	Notes
Number of Iterations	21	Total iterations including all prompts and bug fixes.
Times Code Was Not Executable	3	Iterations where runtime errors were reported and the code failed to execute.
User Input Refinements	12	Instances where user input guided refinements to address observed issues.
Accuracy with Initial Model	9/10	Accuracy improved consistently after addressing bugs and offsets.
Usability of Generating Variations	8/10	The final model supported meaningful variations but required careful input.

	
Initial Reference Image	Final 3D Parametric Geometry

8.3 Code Snippet: Deformed Coffered Grid



```python
import Rhino.Geometry as rg

# Ensure the input surface is valid
if isinstance(surface, rg.Brep):
    # Convert Brep to a surface
    if len(surface.Faces) > 0:
        surface = surface.Faces[0].ToNurbsSurface()
elif isinstance(surface, rg.Surface):
    # The input is already a Surface
    surface = surface
else:
    raise ValueError("Please provide a valid Rhino Surface or Brep as input.")

# Function to project vertically, offset twice, and loft
def project_offset_and_loft_vertically(surface, grid_size_x, grid_size_y, cell_size, offset_distance, second_offset_distance, downward_distance):
    rectangles = []  # Original projected rectangles
    offsets_1 = []   # First inward offset
    offsets_2 = []   # Second inward offset + downward movement
    lofts_1 = []     # Lofts between original and first offset
    lofts_2 = []     # Lofts between first and second offset

    for i in range(grid_size_x):
        for j in range(grid_size_y):
            # Create flat grid points
            p1 = rg.Point3d(i * cell_size, j * cell_size, 0)
            p2 = rg.Point3d((i + 1) * cell_size, j * cell_size, 0)
            p3 = rg.Point3d((i + 1) * cell_size, (j + 1) * cell_size, 0)
            p4 = rg.Point3d(i * cell_size, (j + 1) * cell_size, 0)

            # Project grid points vertically onto the surface
            def project_point_vertically(point, surface):
                vertical_line = rg.Line(point, point + rg.Vector3d(0, 0, -1000))  # Long vertical line downward
                intersections = rg.Intersect.Intersection.CurveSurface(
                    rg.LineCurve(vertical_line),
                    surface,
                    0.01,  # Tolerance
                    0.01   # Overlap tolerance
                )
                if intersections and intersections.Count > 0:
                    return intersections[0].PointA
                else:
                    return None  # Return None if no intersection is found

            # Project each corner of the rectangle
            pt1 = project_point_vertically(p1, surface)
            pt2 = project_point_vertically(p2, surface)
            pt3 = project_point_vertically(p3, surface)
            pt4 = project_point_vertically(p4, surface)

            # Skip this grid cell if any projection fails
            if None in [pt1, pt2, pt3, pt4]:
                continue

            # Compute the center of the grid cell
            cell_center = rg.Point3d((pt1.X + pt3.X) / 2, (pt1.Y + pt3.Y) / 2, (pt1.Z + pt3.Z) / 2)

            # Create the original rectangle
            original_corners = [pt1, pt2, pt3, pt4, pt1]
            original_curve = rg.Polyline(original_corners).ToNurbsCurve()
            rectangles.append(original_curve)

            # Offset each vertex toward the center (first inward offset)
            offset_corners_1 = [
                pt + (cell_center - pt) * (offset_distance / cell_center.DistanceTo(pt))
                for pt in [pt1, pt2, pt3, pt4]
            ]
            offset_corners_1.append(offset_corners_1[0])  # Close the loop
            offset_curve_1 = rg.Polyline(offset_corners_1).ToNurbsCurve()
            offsets_1.append(offset_curve_1)

            # Loft between original and first offset
            loft_1 = rg.Brep.CreateFromLoft(
                [original_curve, offset_curve_1],
                rg.Point3d.Unset,
                rg.Point3d.Unset,
                rg.LoftType.Normal,
                False
            )
            if loft_1 and len(loft_1) > 0:
                lofts_1.append(loft_1[0])

            # Offset the first offset inward again (second inward offset)
            offset_corners_2 = [
                pt + (cell_center - pt) * (second_offset_distance / cell_center.DistanceTo(pt))
                for pt in offset_corners_1[:-1]
            ]
            offset_corners_2.append(offset_corners_2[0])  # Close the loop

            # Move the second offset downward
            offset_corners_2 = [
                pt + rg.Vector3d(0, 0, -downward_distance) for pt in offset_corners_2
            ]
            offset_curve_2 = rg.Polyline(offset_corners_2).ToNurbsCurve()
            offsets_2.append(offset_curve_2)

            # Loft between first and second offset
            loft_2 = rg.Brep.CreateFromLoft(
                [offset_curve_1, offset_curve_2],
                rg.Point3d.Unset,
                rg.Point3d.Unset,
                rg.LoftType.Normal,
                False
            )
            if loft_2 and len(loft_2) > 0:
                lofts_2.append(loft_2[0])

    return rectangles, offsets_1, offsets_2, lofts_1, lofts_2

# Parameters
#offset_distance = 0.1           # First inward offset distance
#second_offset_distance = 0.05   # Second inward offset distance
#downward_distance = 0.1         # Downward movement distance for the second offset

# Generate the projected grid, offsets, and lofts
rectangles, offsets_1, offsets_2, lofts_1, lofts_2 = project_offset_and_loft_vertically(
    surface, grid_size_x, grid_size_y, cell_size, offset_distance, second_offset_distance, downward_distance
)

# Outputs
a = rectangles  # Original projected rectangles
b = offsets_1   # First inward-offset rectangles
c = offsets_2   # Second inward-offset rectangles, moved downward
d = lofts_1     # Lofts between original and first offset
e = lofts_2     # Lofts between first and second offset
```