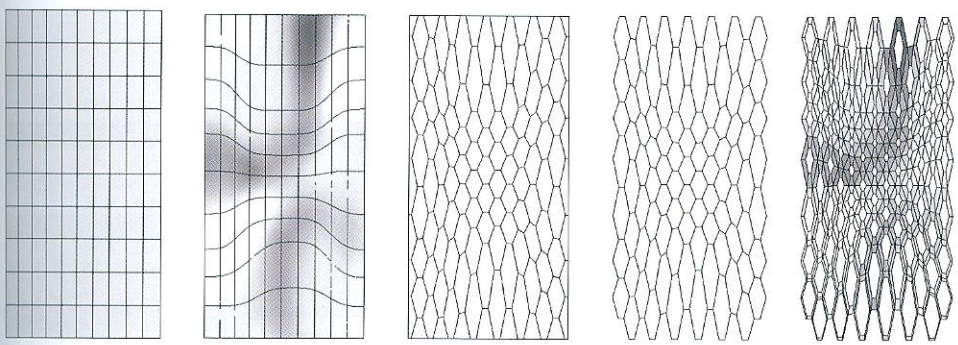
5_skins

advanced data management

"Complexity that works is built up out of modules that work perfectly, layered one over the other".

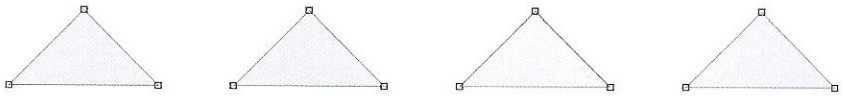
Kevin Kelly

Tridimensional skins are defined by managing the structure of algorithmic data; in Grasshopper, data is managed according to the hierarchical structure of the Data Tree.

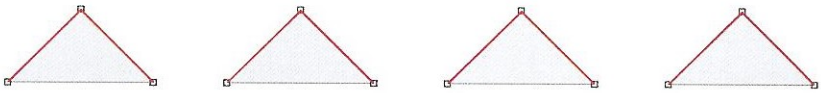


The Data Tree enables complex relationships among objects to be formed. Thus far, computational logics have been straightforward and discussing data structure was unnecessary. This part of the text will elaborate on the basic concepts of using data structure to control form.

The following example will introduce the concept of how data structure controls output. To output a polyline curve that connects twelve vertices, an algorithm is defined using four triangular surfaces set from Rhino. The surfaces are deconstructed to extract each Brep's vertices using the *Deconstruct Brep* component, then the *Polyline* component is used to draw a polyline connecting the vertices.

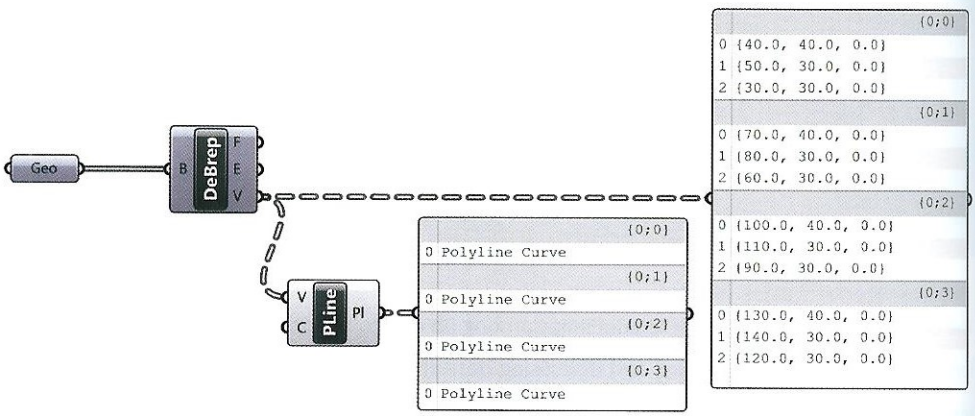


The result differs from the desired final output. The defined logic did not create a single polyline through the twelve points. Instead as expected, the data structure defined four polylines each with three vertices.

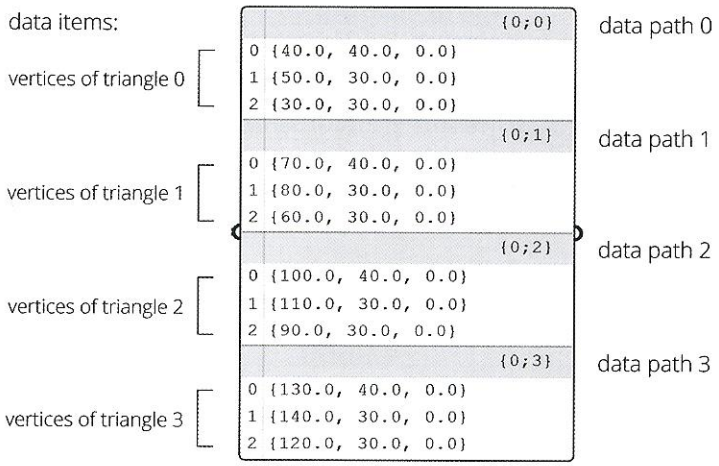


To understand the algorithms hierarchical data output structure, two *Panel* components can be connected to the (V) and (Pl) outputs of *Deconstruct Brep* and *Polyline* components respectively. Two observations can be made:

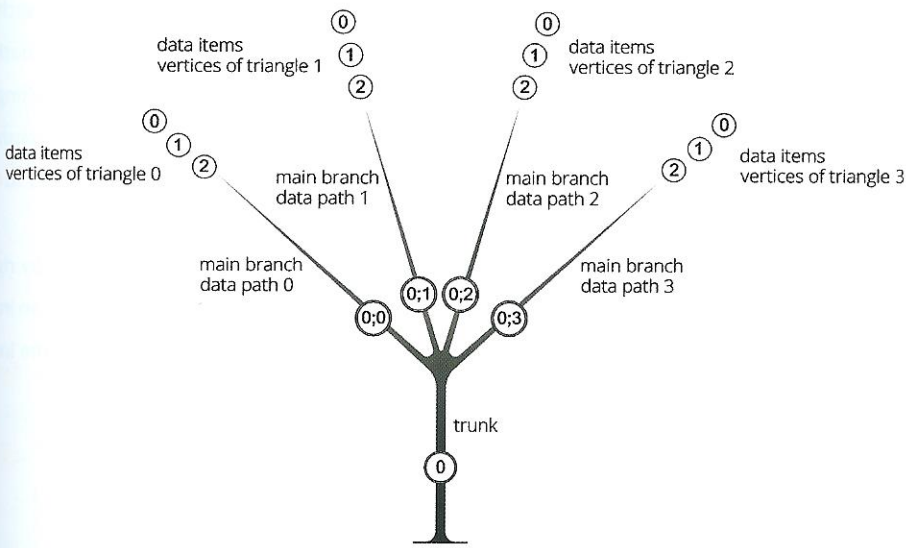
1. The outgoing wire generated by the *Deconstruct Brep* component is *dashed*;
2. Data within the lists are split into four subsets.



Grasshopper stores data according to a **Parent-Child** logic, which creates a subset for each parent ***data path***. In the previous definition a subset was generated for each triangular *parent* surface, and an item was created for each *child* vertex. Meaning, the vertices of the triangle 0 are hosted within the *data path* {0;0}, the vertices of the triangle 1 are hosted within the *data path* {0;1}, etc.



The hierarchy of data structuring can be graphically visualized using a **tree-chart**, as shown below. For this reason, subsets of the initial input trunk are referred to as **branches**.



As already pointed out, the Data Tree is "invisible" when components do not work simultaneously on multiple data. In those cases the Tree is made by just one *branch* or by the only *trunk.*

Data Trees follow two fundamental rules:

- **Branches are watertight subsets**, meaning connections between data hosted in different *branches* cannot be established. **Any operation performed on a Data Tree will affect the data stored in every branch**. For this reason, the *Polyline* component cannot create a single polyline through the entire set of vertices, but four different polylines through the vertices belonging to each relative triangular surface.
- **Data Tree can be manipulated** to generate specific results. For instance, to create a polyline through the set of twelve points the Data Tree structure is required to be manipulated via a set of specific components.

5.1 Manipulating the *Data Tree*

Components used to manipulate the structure of *Data Trees* include: *Flatten Tree*, *Unflatten Tree*, *Graft Tree*, and *Flip Matrix* (Sets > Tree).

5.1.1 *Flatten Tree*

The component ***Flatten Tree*** (Sets > Tree) simplifies a Data Tree by removing all branching information and storing all data inside the trunk {0}. Wires previously graphically displayed as dashed are converted to continuous wires after the data is flattened. Flattening is achieved by connecting an output to the T-input of the *Flatten Tree* component. For instance, if the V-output of the *Deconstruct Brep* component is connected to *Flatten* (T-input) and the T-output of *Flatten* is connected to the *Polyline* component (V-input), the desired result will be achieved.

Every component provides the possibility to internally *Flatten* incoming or outgoing data, by right-clicking on the specific input/output and selecting *Flatten* from the contextual menu. When an input or output data stream is being *Flattened* a downwards arrow symbol will appear next to the input or output.

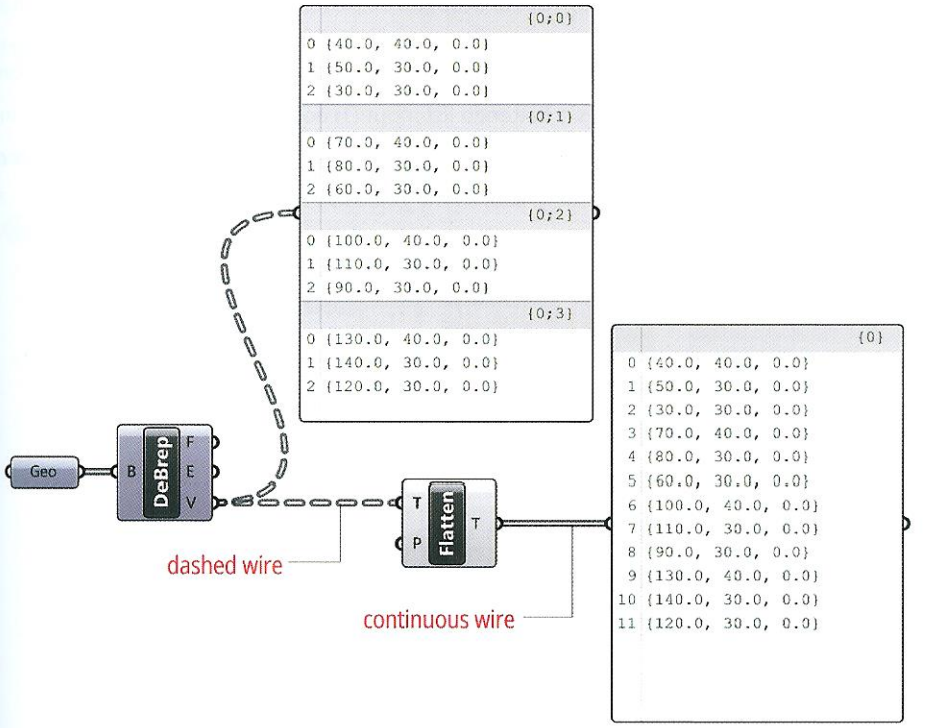
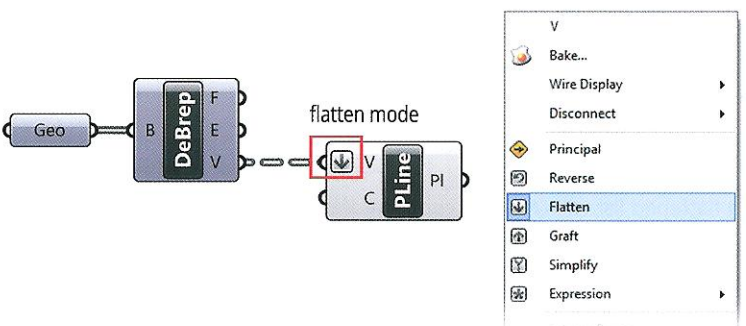


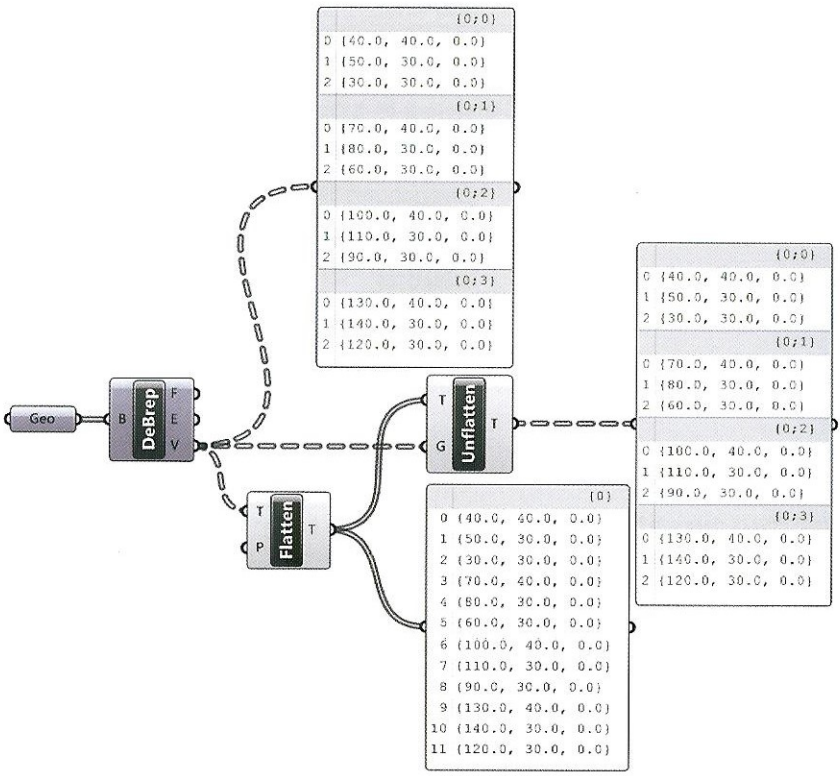
FIGURE 5.1

A "structured" flow of data can be identified by a dashed wire, while a flattened flow by a continuous line.



5.1.2 *Unflatten Tree*

A flattened Data Tree can be restructured into branches using a guide data structure. The component ***Unflatten Tree*** (Sets > Tree) converts a flattened list input (T) according to a guide Data Tree input (G), outputting data structured according to the guide Data Tree. The component *Unflatten Tree* only operates if the flattened list has the same data as the guide list.

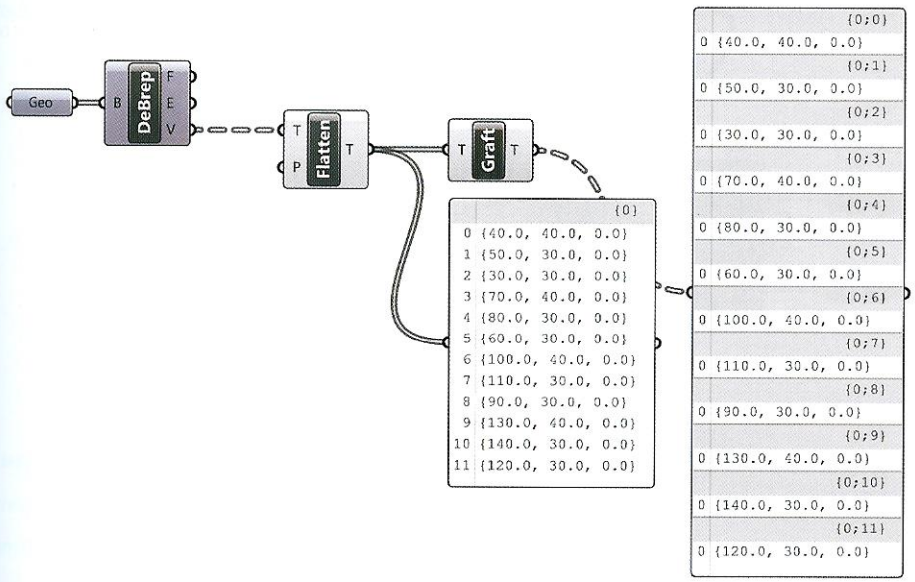


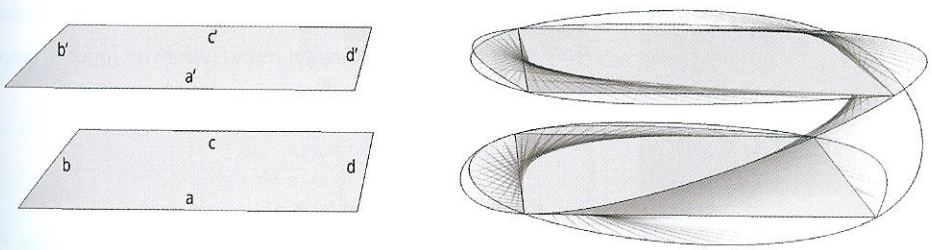
5.1.3 *Graft Tree*

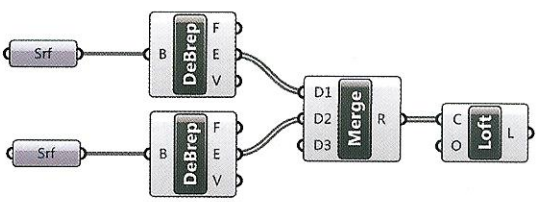
The component ***Graft Tree*** (Sets > Tree) creates a branch for every item within an arbitrary list. Accordingly, a flattened list with N items connected to the T-input of the *Graft Tree* component will return a new list with N branches, one branch for each item.

Graft Tree can be used to match disconnected sets of corresponding objects. For example, two surfaces set from Rhino are connected to the component *Deconstruct Brep* to access each surfaces edges. To output four lofted surfaces through the corresponding edges (a-a', b-b', c-c', d-d'), the two

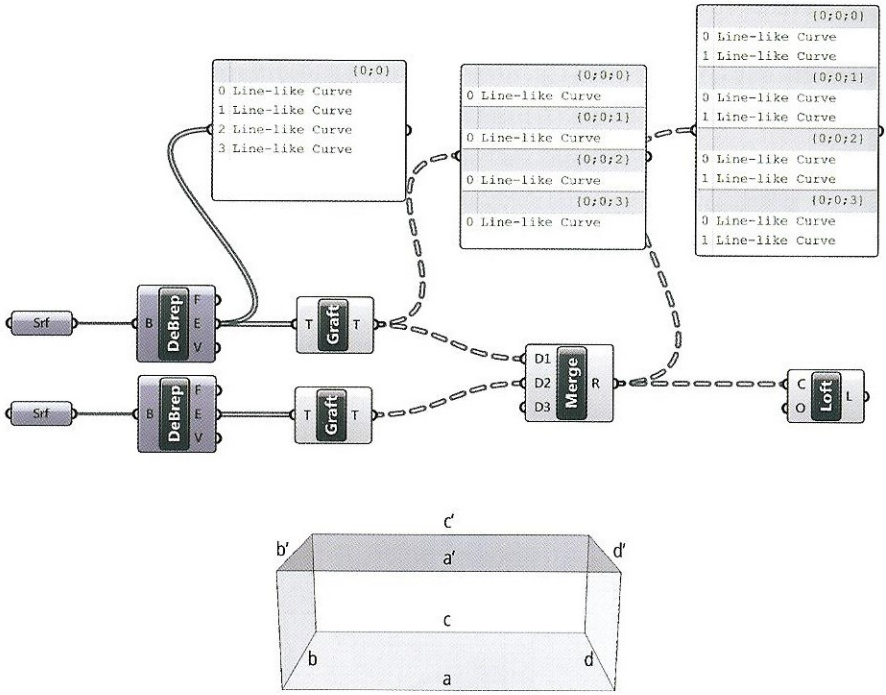
data flows are required to be grafted, then merged. If the two data flows are *merged* and not *grafted*, *Lofting* will be performed through the entire set of curves according to the order: a'-b'-c'-d'-a-b-c-d.







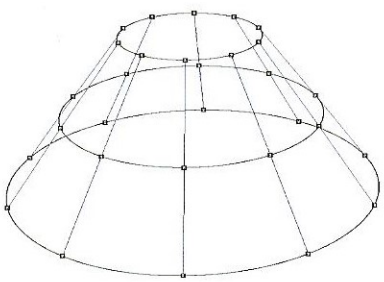
If the data is *grafted* before merging, a branch will be formed for each corresponding edge. If the appropriately structured data is connected to the loft component the desired results will be achieved.



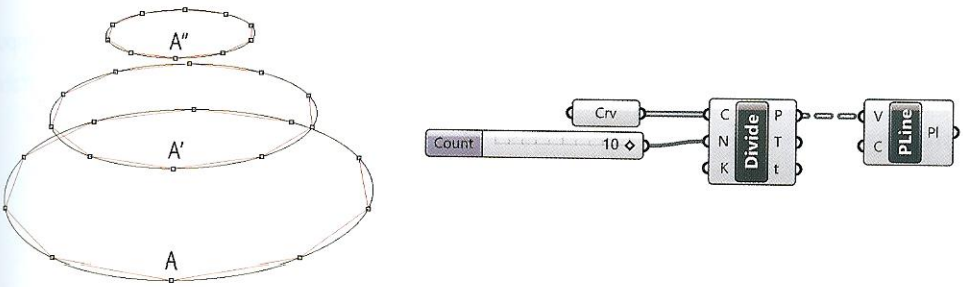
Every component provides the possibility to internally *Graft* incoming or outgoing data by right-clicking on the specific input/output and selecting *Graft* from the contextual menu. When an input or output data stream is being *grafted* an upwards arrow symbol will appear next to the input or output.

5.1.4 *Flip Matrix*

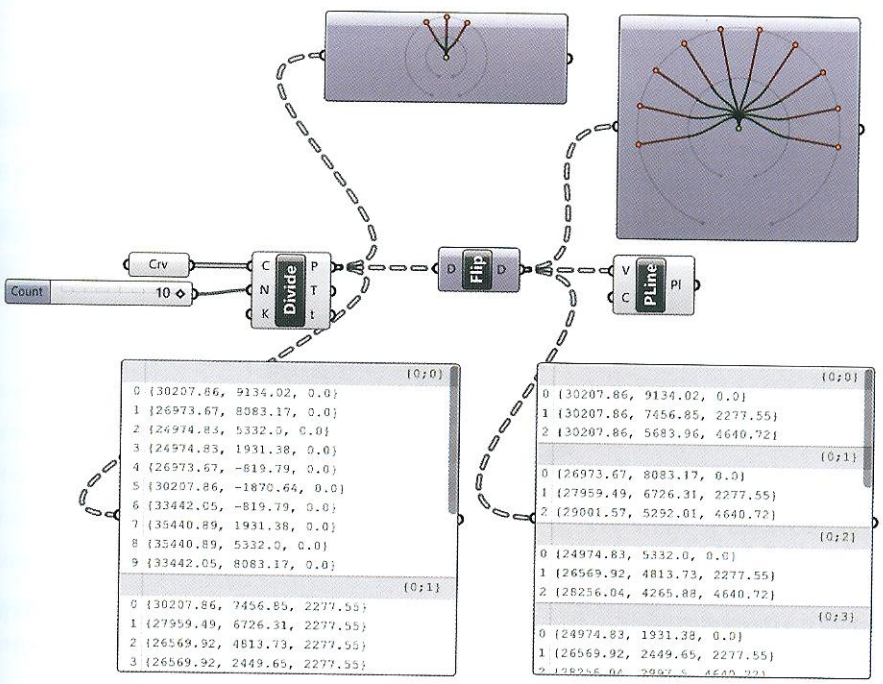
The component ***Flip Matrix*** (Sets > Tree) swaps rows and columns in the *matrix-like* Data Tree. Put another way, the component *Flip* exchanges items with branches and branches with items.



For example, three circles set from Rhino are divided into ten parts using the component *Divide Curve*. The output points (P) are connected to the *PolyLine* component V-input generating three polylines that connect all points of each divided curve respectively. To generate the desired lines through corresponding points (A-A'-A") the data tree must be flipped.



Initially the *Divide Curve's* Data Tree has: **3 branches** corresponding to the 3 circles with *10 items* corresponding to the **10 points**. The flipped matrix is composed of **10 branches** and **3 points** for each branch. The Data Tree can be visualized adding (and double-clicking) *Param Viewer* (Params > Util).

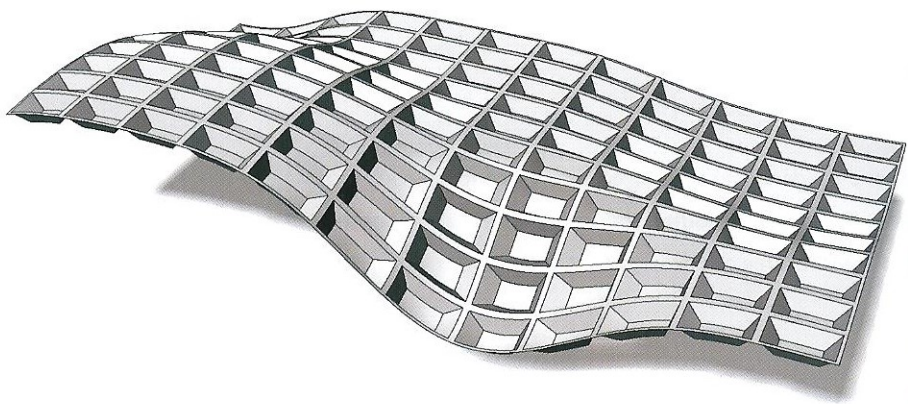


5.2 Skins

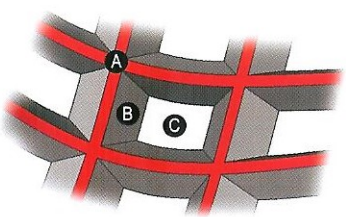
The following exercises introduce methods to create **skins** and **patterns** on an input NURBS surface by manipulating Data Tree structure.

5.2.1 Rectangular based pattern

The first example will create a tridimensional pattern from the rectangular subdivision of an input surface. The final output will be a set of *deformed pyramidal frustums*[11] or, in the case of a flat surface, *pyramidal frustums*.



The tridimensional pattern is created by joining three different surfaces: surface A, surface B and surface C, as shown below.

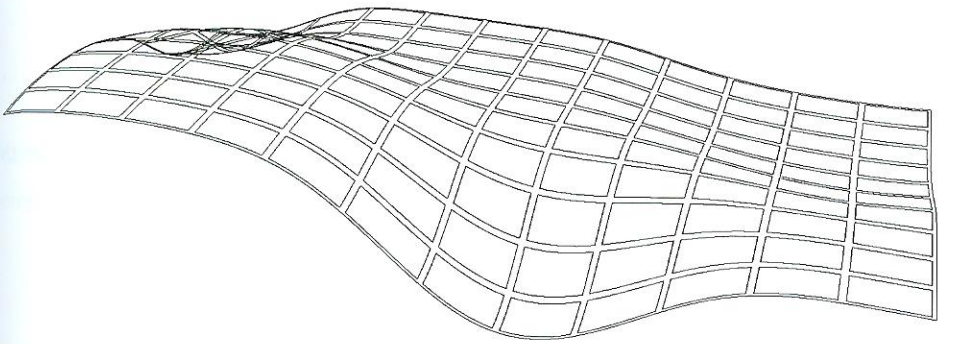


NOTE 11

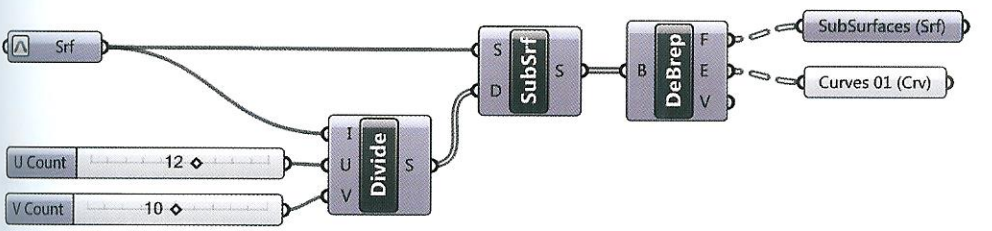
A *prismatoid* is a polyhedron where all vertices lie in two parallel planes. A *prismoid* is a *prismatoid* where parallel planes host the same number of vertices and lateral faces are trapezoids. An example of *prismoid* is the *pyramidal frustum*.

SURFACES A

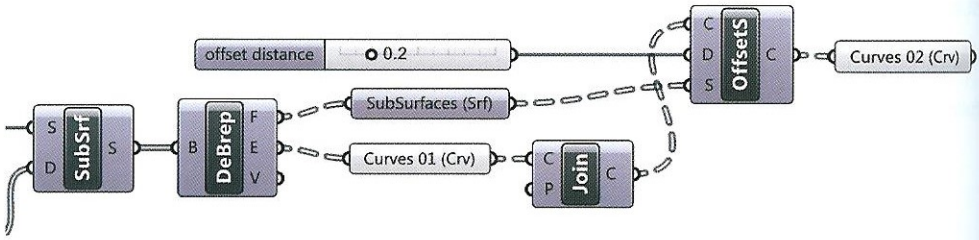
The surfaces denoted as "A" frame the *pyramidal frustums;* the frame's edges are displayed in the following image.



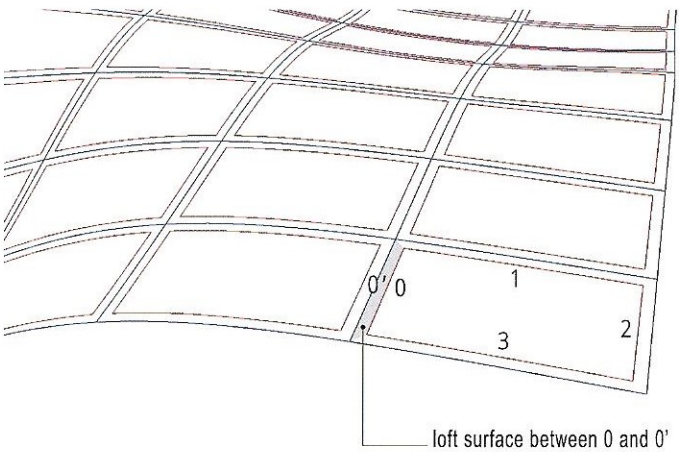
The first step in the definition is to set and reparameterize a surface from Rhino using the *Surface* "container" component. Then the *Isotrim-SubSrf* output is deconstructed using the *Deconstruct Brep* component to extract the defined faces and edges. To easily identify the *Deconstruct Brep* output (F) and (E) two box components are defined and renamed as *SubSurfaces* and *Curves 01* respectively.



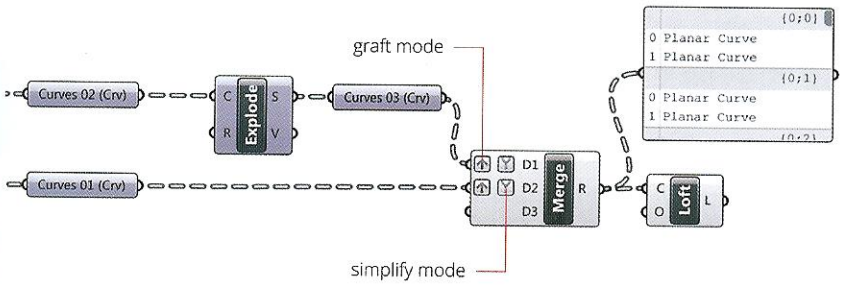
The four edges of each sub-surface are joined, forming a polyline which is offset coincident to the set surface using the component ***Offset on Srf*** (Curve > Util). The C-input of the *Offset on Srf* component requires curves to offset, the D-input the **offset distance**, and the S-input the surface or surfaces on which to perform the offset. In this instance, the sub-surfaces returned by the *Deconstruct Brep* component are the surfaces on which to perform the offset. The C-output, stored in a container component, is the collection of offset curves defined for each sub-surface.



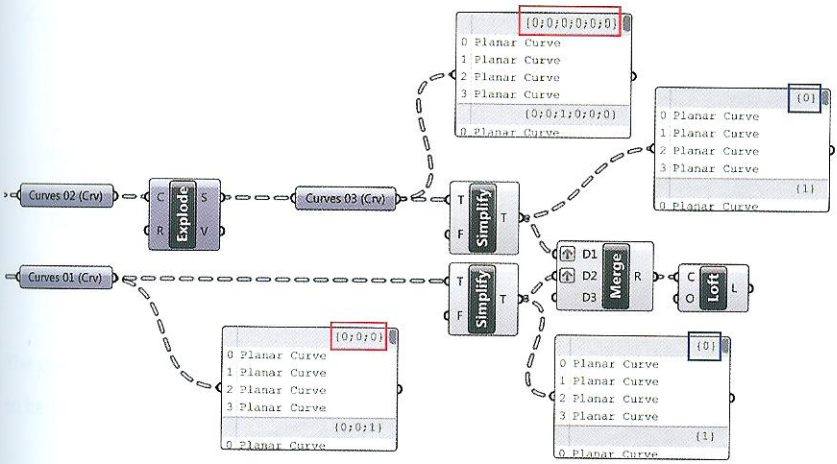
The following image depicts the *red* offset curves and the *blue* sub-surface edges. To create the frame surfaces a **loft** operation is performed between corresponding curves (e.g. 0 to 0').



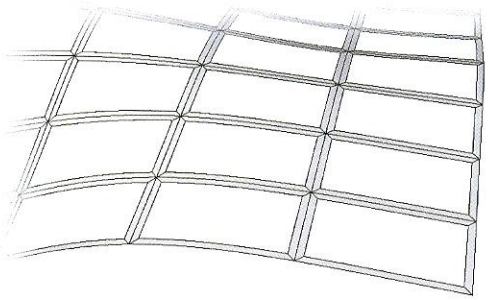
Since the output (C) of the *Offset on Srf* component stored as *Curves 02* are joined polylines, the curves must be exploded to define a loft operation between corresponding curves. The component *Explode Curve* (Curve > Util) explodes the polyline to define four edges. The output (S) of *Explode Curve* component is stored in the container component *Curves 03. To* define the lofting sequence, *Curves 01* and *Curves 03* are merged using the *Merge* component. **As explained in 5.1.3**, the inputs (D1) and (D2) are required to be **grafted** to correlate the data within the two corresponding sets. The data is also required to be simplified. Meaning, the data tree structure is reduced by removing overlapping branches. Data can be simplified using the component *Simplify Tree* (Sets > Tree) or internally by right-clicking on a input/output and specifying *Simplify* from the context menu.



The following image, visualizes the change in data structure resulting from the *Simplify Tree* component.



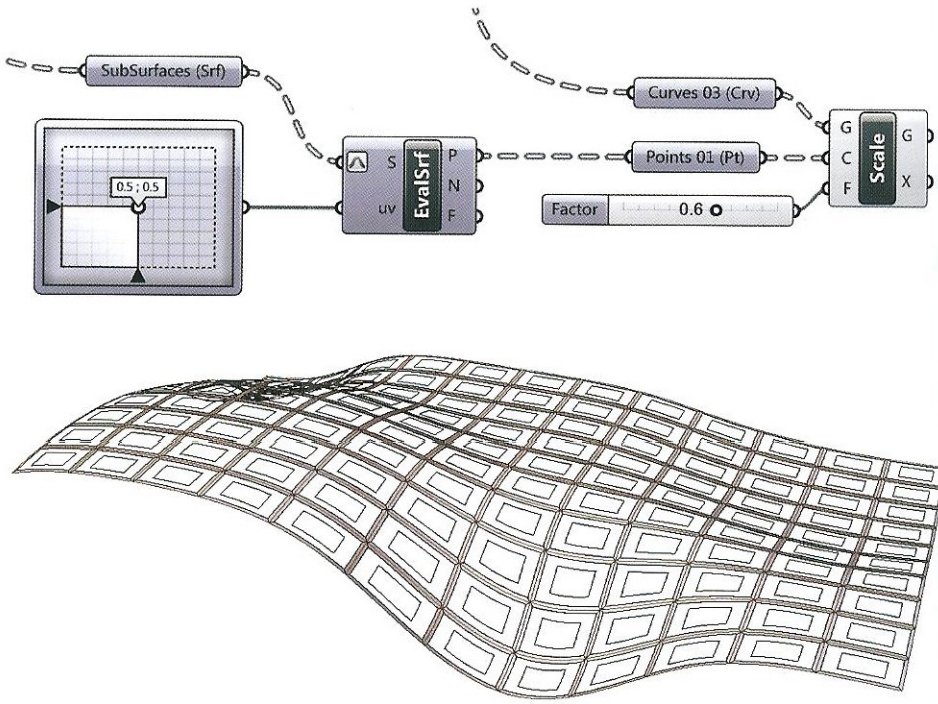
The properly formatted data is lofted returning the set of surfaces "A". If the offset distance input (D) of the *Offset Srf* component is modified the frames width will parametrically change.



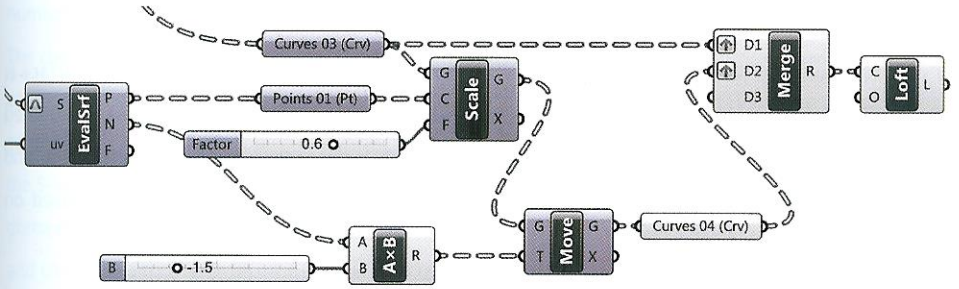
SURFACES B

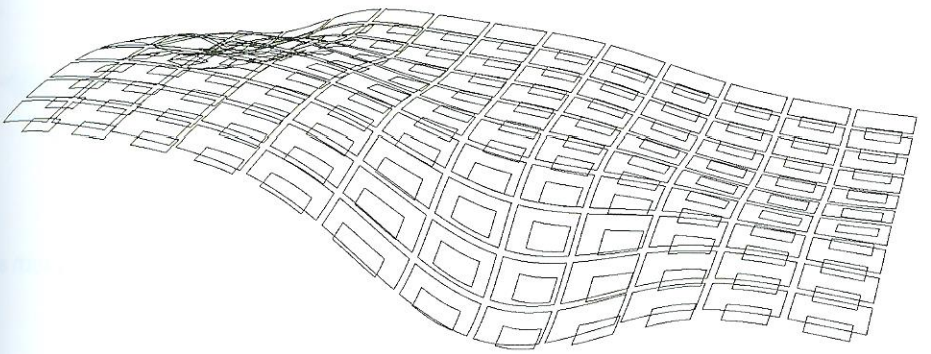
The surfaces denoted as "B" form the lateral faces of the *pyramidal frustums*. The *pyramidal frustums* faces are defined by a loft between data set *Curves 03* and an additional data set *Curves 04*. The data set *Curves 04* is defined by scaling and moving *Curves 03*.

To define the data set *Curves 04*, the mid point of each sub-surface is calculated using the component *Evaluate Surface*, with the input slot (S) set to reparameterize, and the input (uv) defined by an *MD Slider* set to (0.5;0.5). The *Evaluate Surface* component output (P) calculates a set of points that are collected by the container component *Points 01. The* data set *Curves 03*, is scaled using the component *Scale* by an input factor (F) specified by a *Number Slider*. The result is illustrated in the following picture.

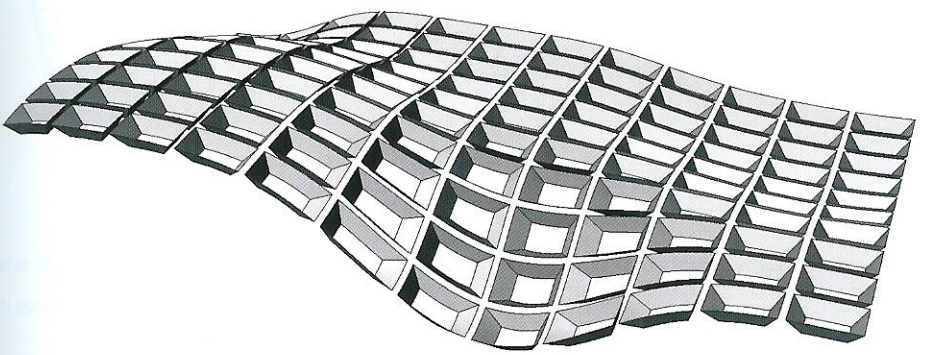


Then the scaled *Curves 03* are translated according to the surface normals, defined by the output (N) of *Evaluate Surface* component. Scalar multiplication is performed to define the translation. The translated curves are stored in the box component *Curves 04*.



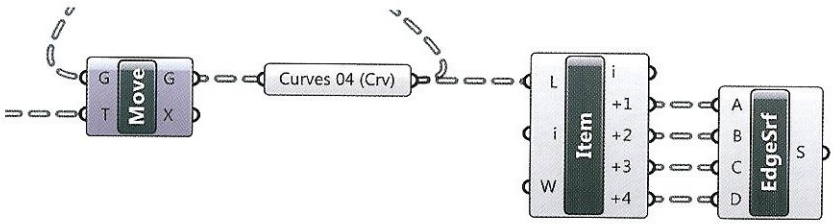


The *pyramidal frustums* faces are defined by lofting *Curves 03* and *Curves 04*. The curves are required to be *Grafted* but not simplified.

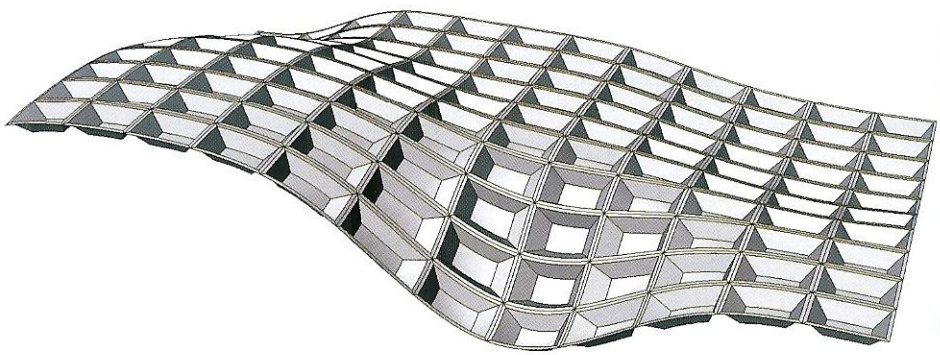


SURFACES C

The surfaces denoted as "C" are the *pyramidal frustums* caps defined by *Curves 04.* The surface is defined by the component *Edge Surface* (Surface > Freeform). The four defining edges are extracted from the data set *Curves 04* using the *List Item* component, by zooming-in on the component and adding four output parameters as shown below (alternative to the standard method based on adding four *List Item* components).



The desired output can be manipulated parametrically by changing the input parameters, such as: the offset distance, the scale factor, the translation factor or even the initial surface.

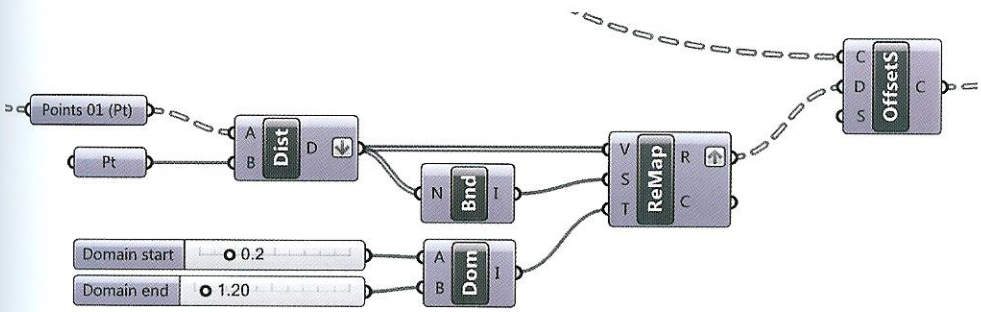


Input parameters are arbitrary data that can be defined manually or alternatively linked to other parameters. For example, an attractor point can be used to vary the offset distance that defines the frames width.

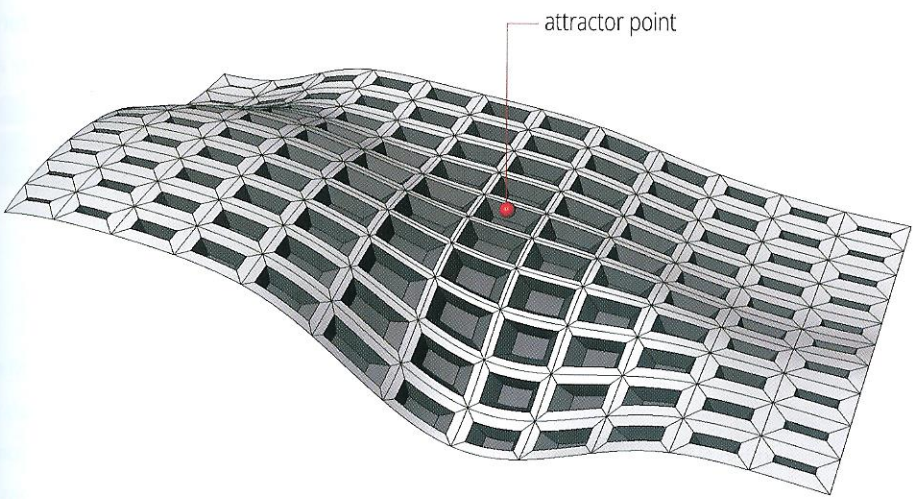
To create an attractor point, a point is set from Rhino using a *Point* container component. The distance is measured between the centers of the sub-surfaces, *Points 01,* and the set point. Since the calculated distances will be greater than the frame's width, the numbers are remapped such that the range of

numbers is within a defined domain; in this case the range is set between 0.2 and 1.2 units.

The distance output (D) is flattened, then connected to the N-input of the *Bounds* (Maths > Domain) component which calculates the source domain for the data set. The input (V) of the *Remap Numbers* component (Maths > Domain) is connected to the flattened output (D) of the *Distance* component, the S-input of *Remap Numbers* is connected to the I-output of *Bounds*, and the T-input to the target domain (I-output of *Construct Domain*). The output (R) of the *Remap Numbers* component is grafted and connected to the D-input of *Offset Srf* component, replacing the *Number Slider* previously set.

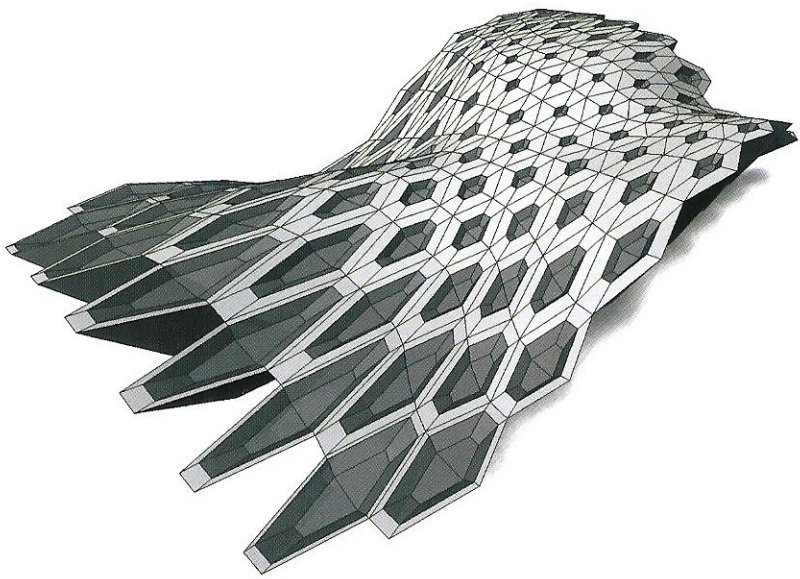


The resulting responsive frame is controlled by an external point. As the distance between the attractor point and the sub-surfaces decreases the frames width will also decrease.

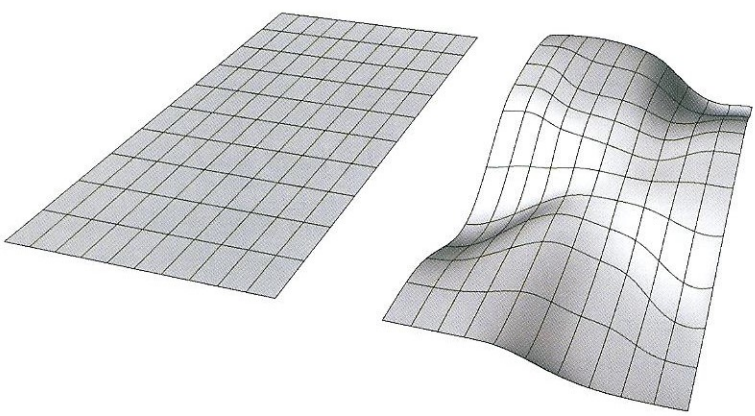


5.2.2 Hexagonal based pattern

The second example will create a tridimensional pattern on a surface by means of a hexagonal subdivision. The procedure is similar to the first example, however there are several pattern based differences.

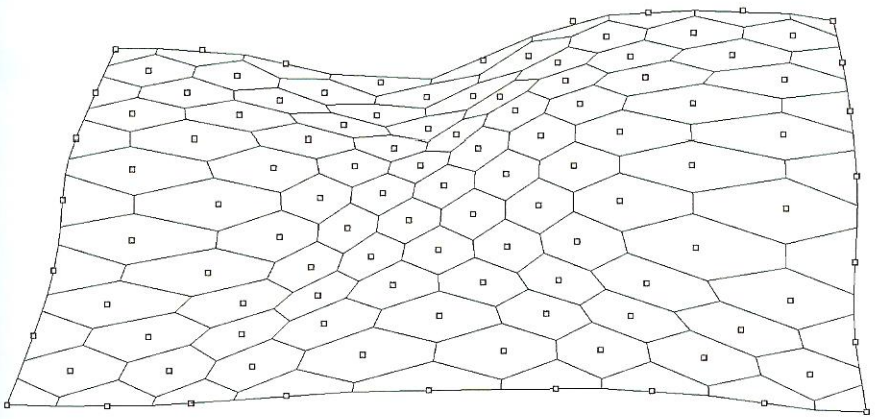


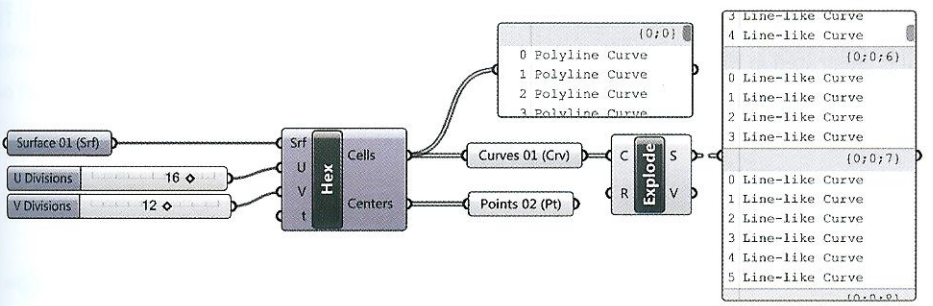
The first difference is that initial surface has an uneven grid of isocurves. The uneven set of isocurves will result in a non-regular hexagonal grid, with smaller hexagons around the surfaces center and larger hexagons near the surfaces edges.



The hexagonal grid is created using the **Lunch Box** plug-in (see 3.7.3) and, in particular, the component *Hexagon Cells* (LunchBox > Panels). The component requires a target surface and a number of subdivisions in U and V directions to operate.

The *Hexagon Cells* component's output (Cells-output) returns single hexagonal cells, each defined as a hexagonal closed polyline, while the Centers-output returns the centers of each closed polyline. Since the grid is applied to a rectangular surface the border-cells are not hexagonal; instead the border cells are comprised of four or five sided polygons.



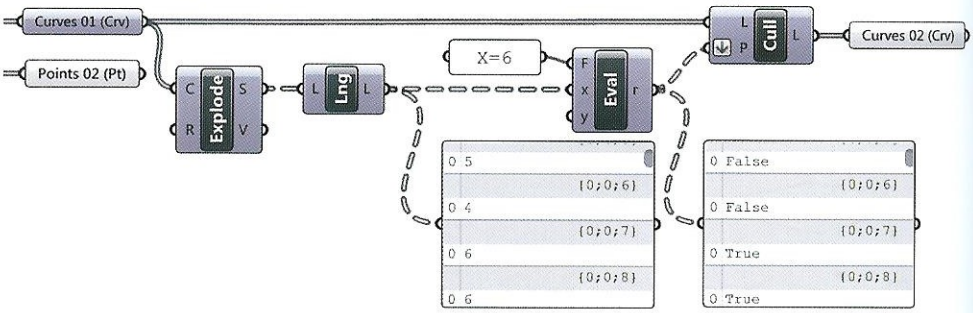


The first step in the definition is to cull the non-hexagonal cells (i.e. cells with less than six sides) from the data set defined by the *Hexagon Cells* output stored in the container component *Curves 01*. To remove the cells with less than six sides, a conditional Boolean statement is defined.

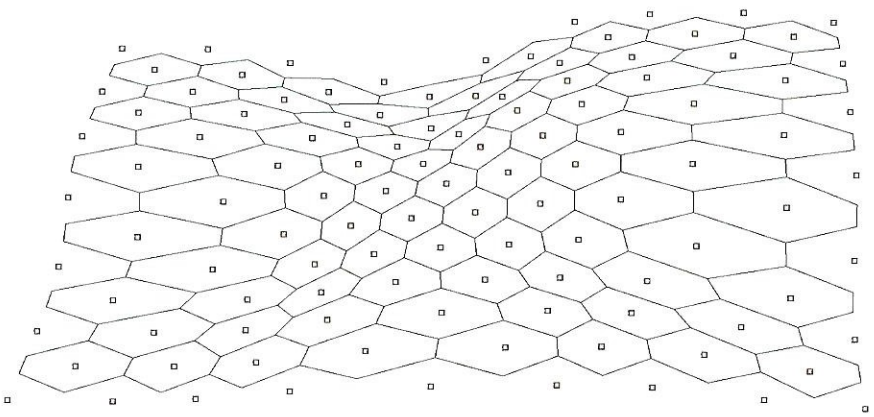
To count the number of sides comprising each cell, the cells are exploded using the component *Explode Curve*. The list length is calculated for each branch of the Data Tree using the *List Length* component. The resulting lengths: four, five or six, are tested against the condition X = 6 using the

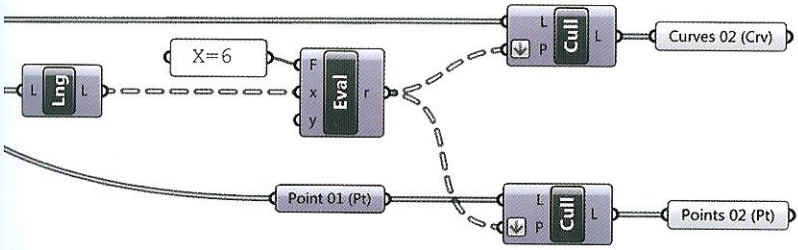
Evaluate component. The resulting output (r) of the *Evaluate* component will be a list of Boolean statements which define the P-input of the *Cull Pattern* component. The culling list input (L), or list to cull, is set as *Curves 01*.

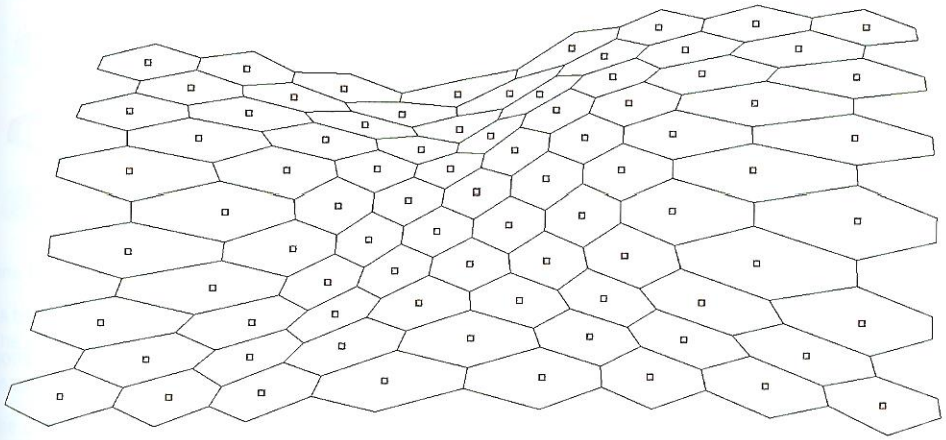
The data contained in the *Curves 01* container component has no branches, only a trunk; this is graphically visualized by the continuous wire. While the output of the *Evaluate* component is a Data Tree with multiple braches, graphically visualized by the dashed wire. In order to match the data structure, the P-input of *Cull Pattern* component is required to be *flattened*. The output of *Cull Pattern* component, containing only hexagonal cells, is collected by the container component *Curves 02*.



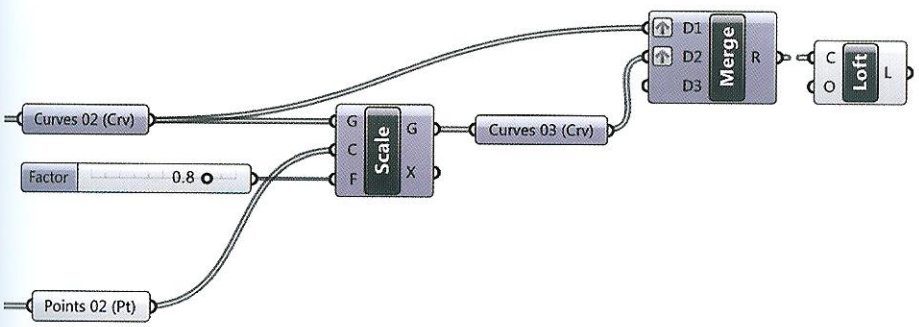
Similarly, the centers of the non-hexagonal cells are culled by another *Cull Pattern* component; with L-input connected to the Centers-output of the *Hexagon Cell* component collected as *Points 01*. The result is illustrated in the following images.

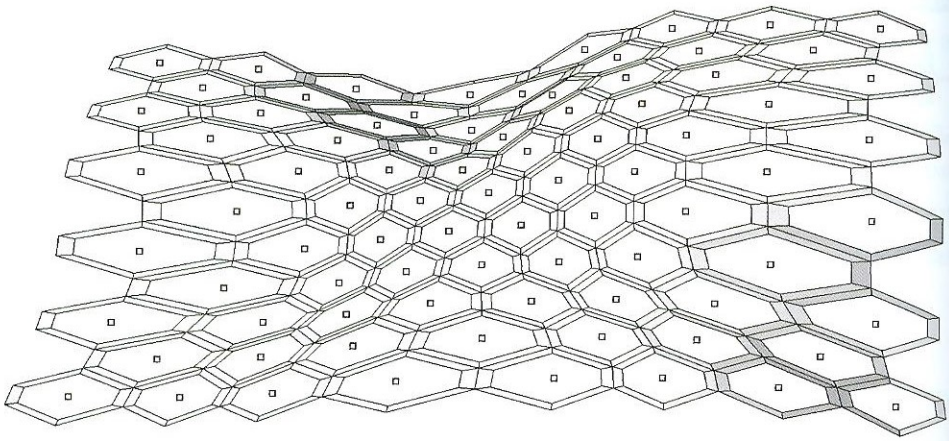




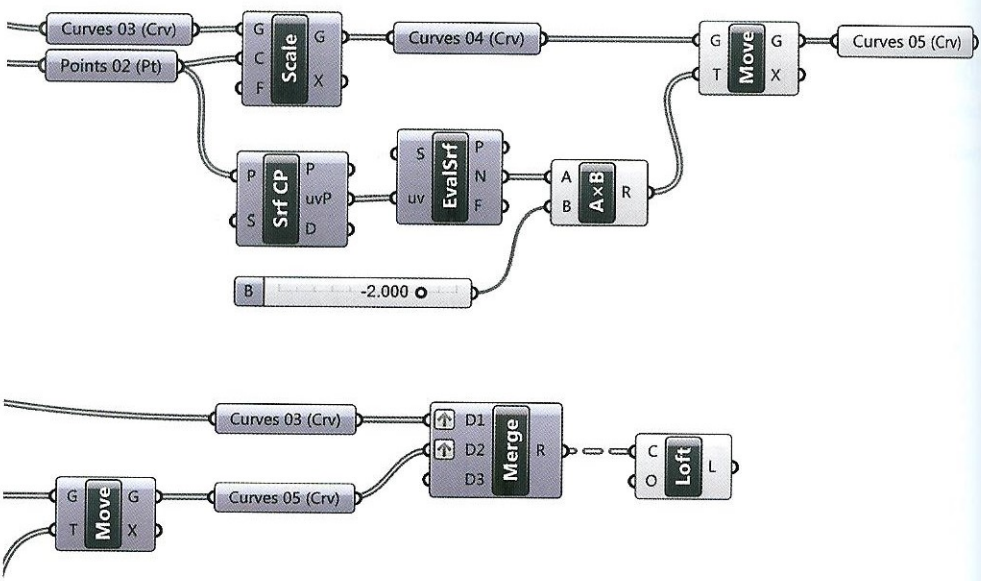


The remaining procedure is similar to the first example. The hexagonal frame is achieved by lofting between *Curves 02* and *Curves 03. Curves 03* are obtained by scaling *Curves 02.* Also in this case the two flows are grafted before merging.

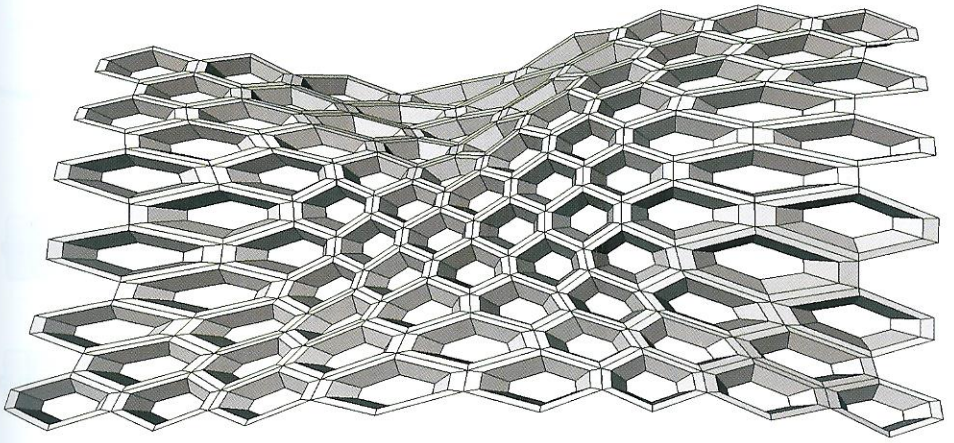




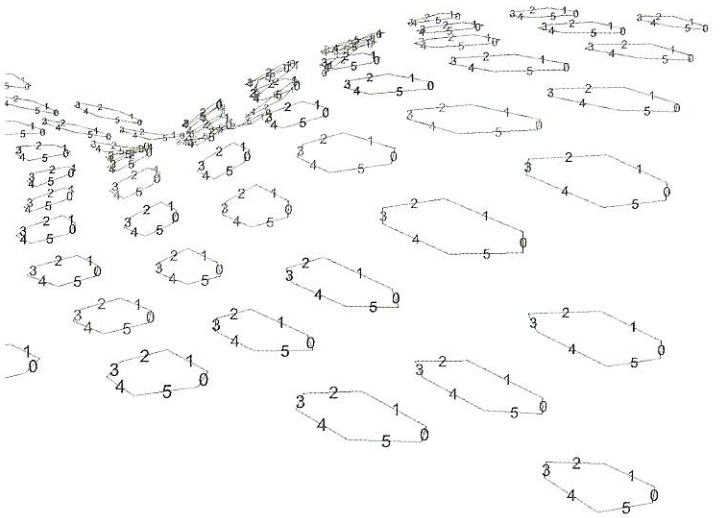
The next step is to generate the lateral faces of the hexagonal frustums; by scaling *Curves 03* and then translating the scaled curves according to the surface normals. The resulting curves, stored as *Curves 05*, are the lower edges of the hexagonal frustums. To create the lateral faces a loft operation is performed between *Curves 03* and *Curves 05*.



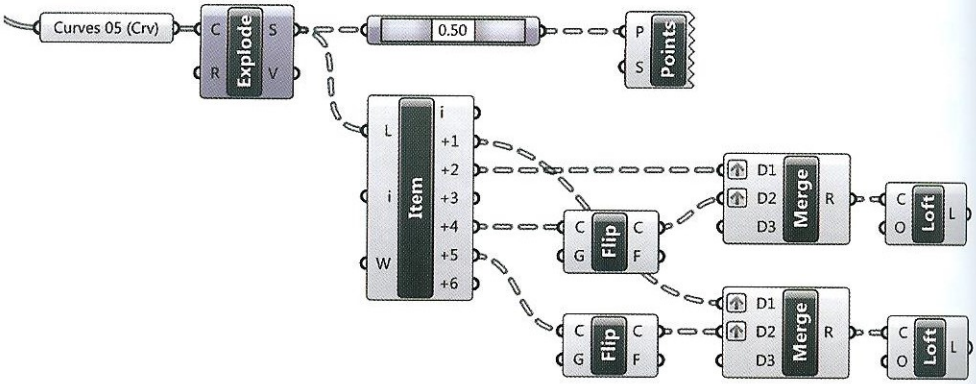
The result is illustrated below.

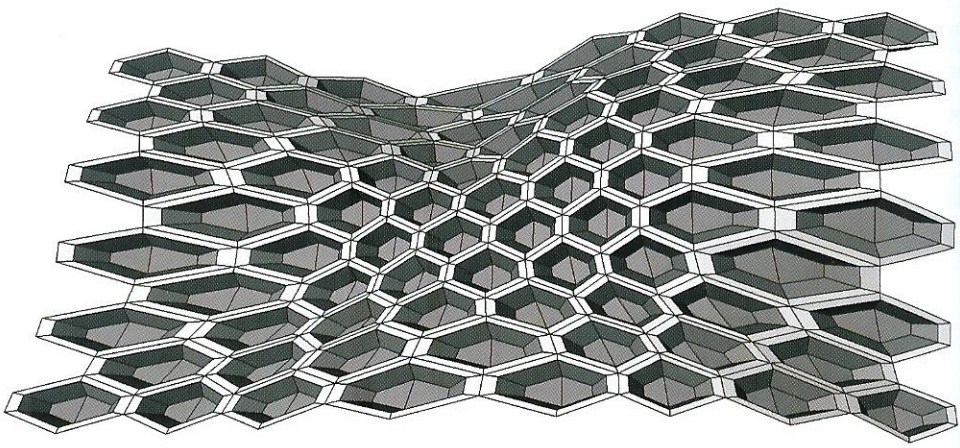


The caps are defined differently from the previous example. The caps have six edges and for this reason they cannot be created as a single surface without using the *Patch* component. To generate the cap a loft operation is performed between curves 2-4 and 1-5 respectively (see image below). Since the hexagon cells have an anticlockwise direction, curve 2 and curve 4 have opposite directions as well as curve 1 and curve 5. If lofted, the resulting surface will be twisted.

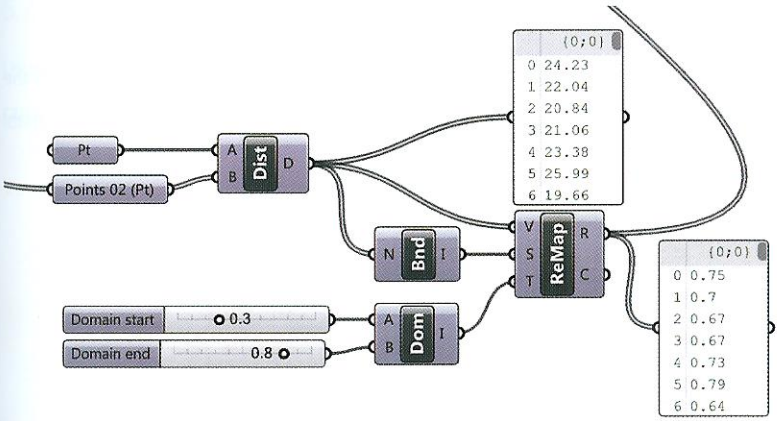


For this reason, curves 1 and 2 must be flipped using the component *Flip Curve*. The following images show the complete sequence and the final geometry.

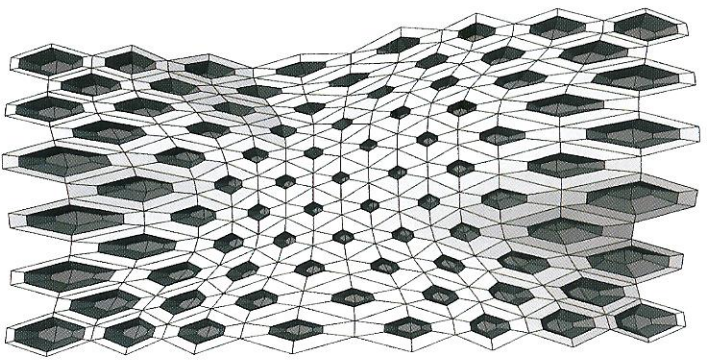




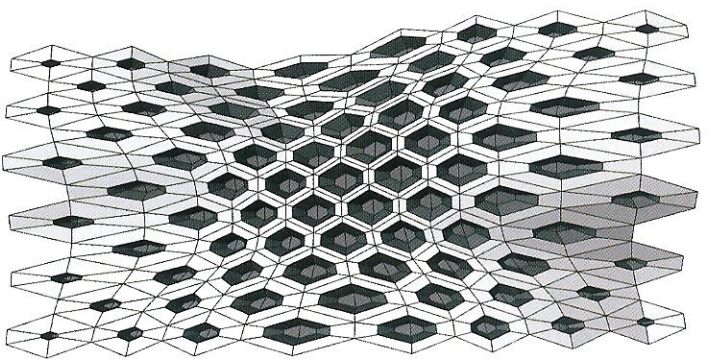
Similar to the previous example, the frames width can be controlled by an attractor point. In this case, the remapped distance values are used as scale factors for the first *Scale* component. The two *Domain* values A and B are the minimum and maximum frame width respectively.



In the following image, A = 0.3 and B = 0.8.



In the following image, A = 0.8 and B = 0.3.



5.2.3 Further Study: Responsive facade

The Masharabiya shading system, based on a traditional Arabic shading lattice-work, is one of the primary features of the competition winning entry for the ADIC headquarters towers in Abu-Dhabi developed by Aedas[12].

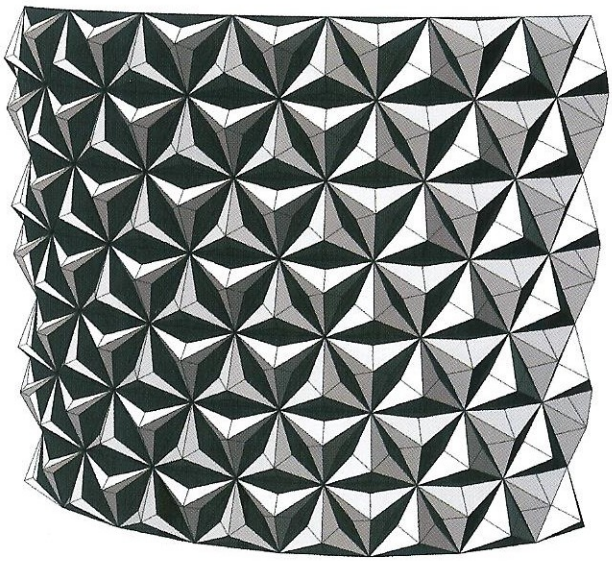


FIGURE 5.2
Abu Dhabi investment Council Headquarters – responsive facade by Aedas.

A simplified algorithm used to define this responsive facade is accesible through Quick Response Code or QR code found on this page.
The QR code can be read by an imaging device such as a smartphone or tablet, and directly links to the Grasshopper definition. The QR code will be used in several parts of this book in order to simplify the understanding of complex algorithms and to visualize the entire construction history of algorithms.

1

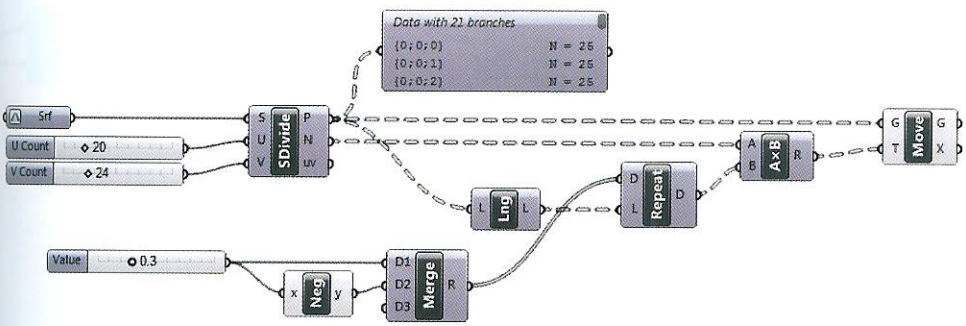
NOTE 12

http://www.aedas.com/Research/ADIC-Responsive-Facade

5.2.4 Weaving

Using strategies developed in previous sections a tridimensional weaving "warp and weft" pattern can be defined on a set surface with a specified thread count in both U and V directions.

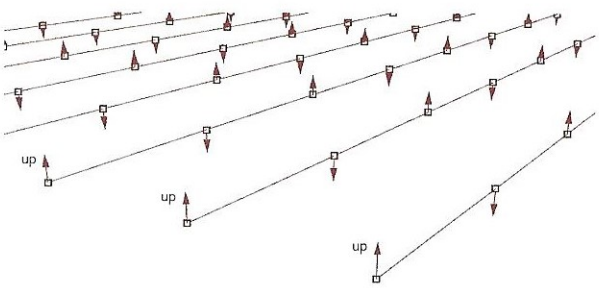
In general, digital weavings are defined by translating a grid of points according to alternating positive and negative surface normal vectors. The first step in the weaving definition, is to generate a grid of points on a set reparameterized surface using the *Divide Surface* component. The defined points are then translated using a recurring list of scalar factors multiplied by surface normal unit vectors.



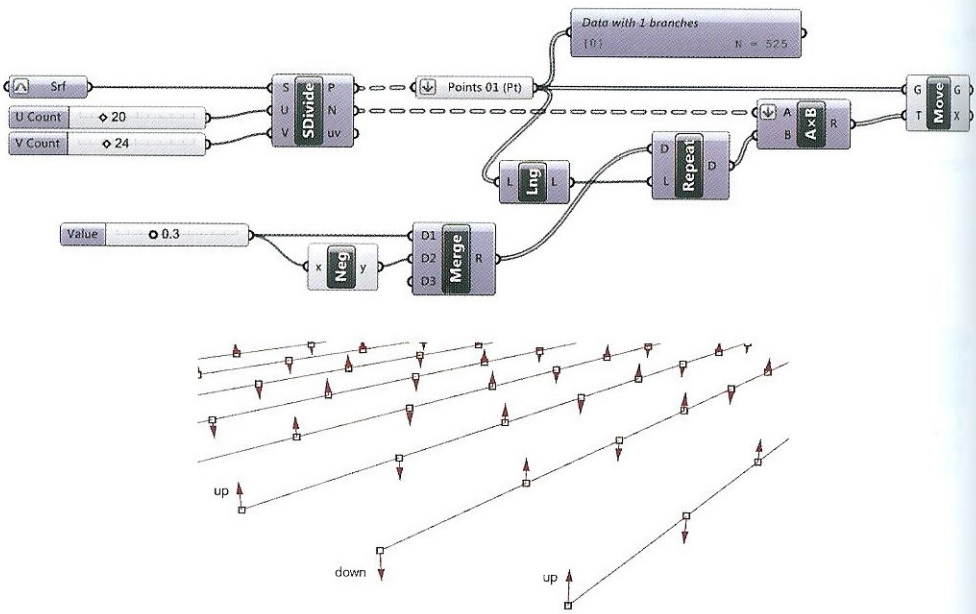
WARP

The output of the *Divide Surface* component is structured with U+1 branches gathering points in *columns* according to the V direction (warp). Due to this structure, the translation repeats similarly

for each column: the start points of each thread are raised, differently from a weaving, where the start points of each thread are alternated (the first raised, the second lowered and so on).

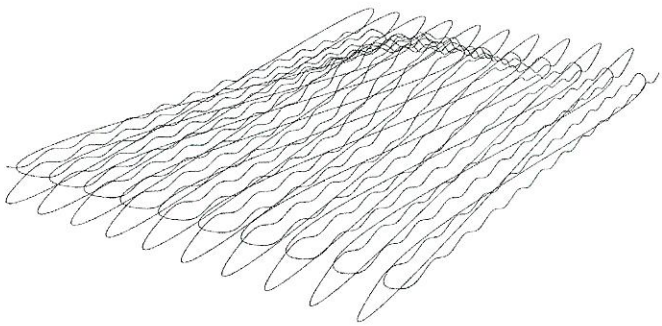


For this reason the P-output of Divide Surface is required to be flattened. The flattened list of points *Points 01*) are then translated using the *Move* component. The translation vectors are defined by multiplying a repetitious scalar list of alternating positive and negative values by a list of normal vectors for each point.

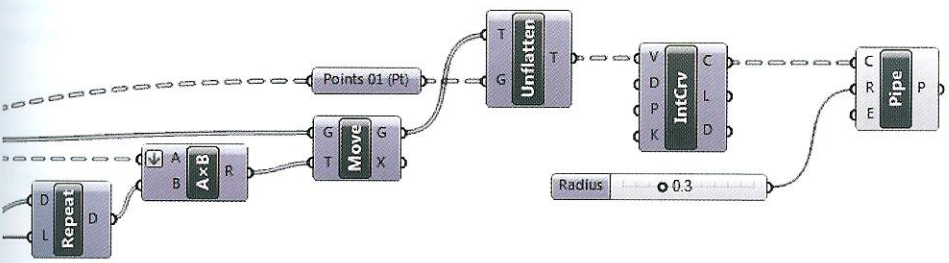


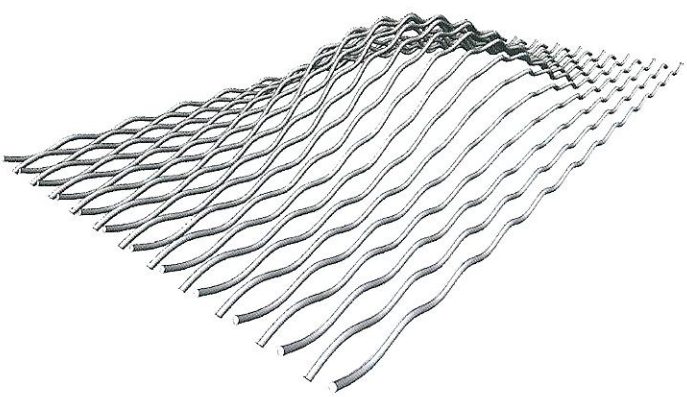
The weaving *warp* is created by interpolating a line through the G-output of the *Move* component using the *Interpolate* component. Since the points data structure has been flattened the *Interpolated*

curve is defined as a single curve through the entire set of points: an undesired result.



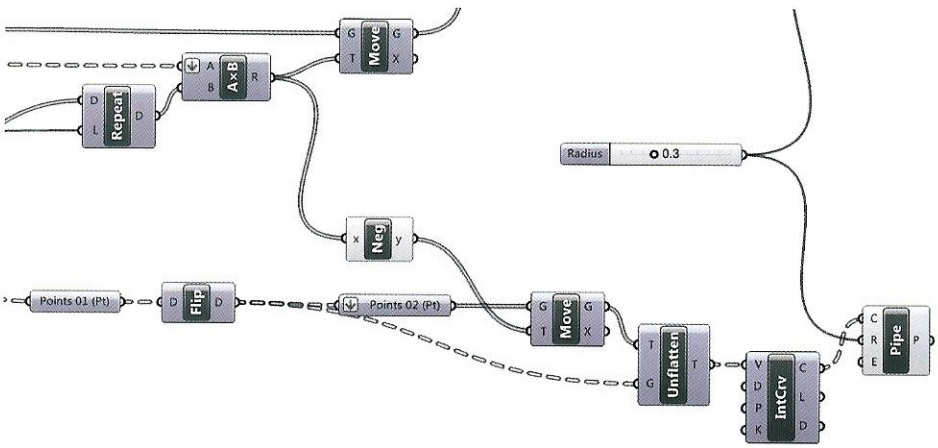
In order to interpolate a curve through each V direction column, the data must be unflattened using the *Unflatten Tree* component. The *Divide Surface* output *Points 01* is used as the guide Data Tree structure input (G-input of *Unflatten Tree*). The resulting data is structured according to the initial Data Tree, each thread is created by connecting the points in each (U+1) column in the V direction.



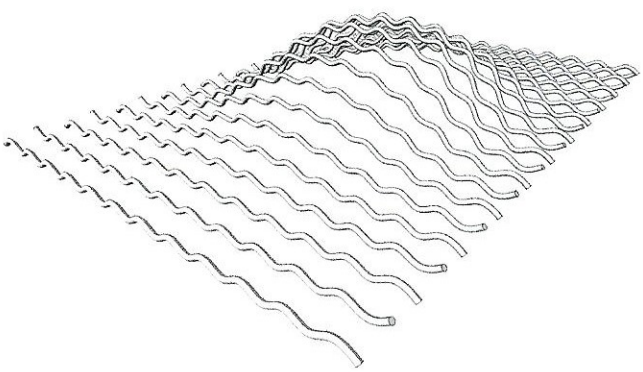


WEFT

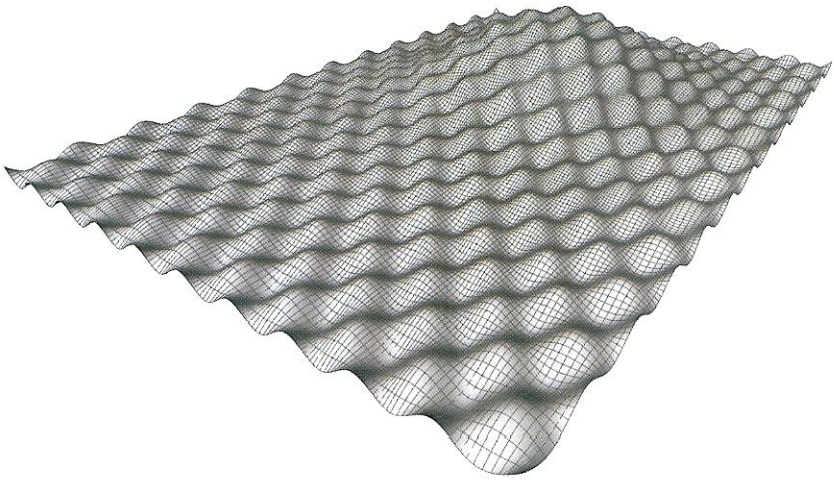
Using a similar logic the *weft* can be defined in the U direction. It is not necessary to repeat the entire algorithm to define the *weft*, instead flip the initial *Points 01* matrix and copy the end of the algorithm as shown below.

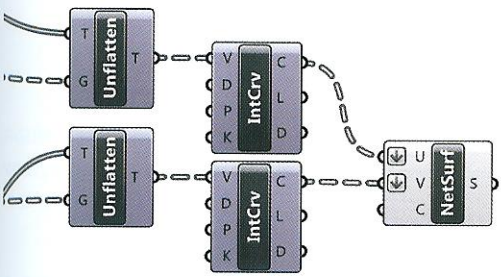


Since the *warp* and *weft* threads should never touch; the translation vectors of the *weft* must have the opposite sense as compared to the translation vectors of the *warp* (*Negative* component).

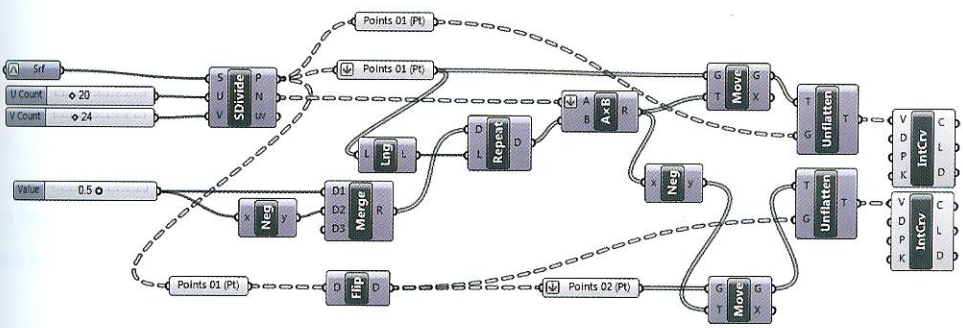


If the *Negative* component is not used to define vectors with opposite sense; the warp and weft interpolated curves will form a curve network. The curve network can be used to create a surface using the *Network Surface* component.



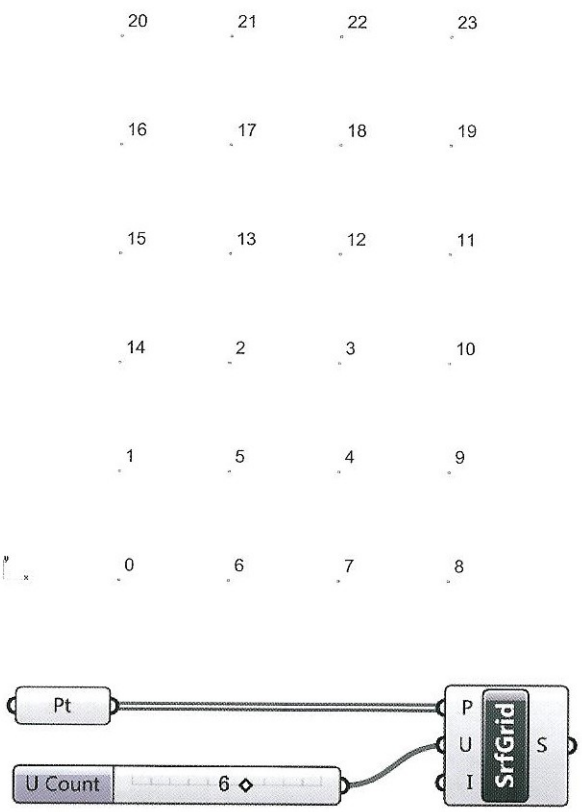


The complete algorithm is illustrated below.



5.3 Sorting strategies using Data Tree

To achieve a desired output, data is required to be structured in a certain way. Data Tree management components structure data to achieve desired results. For example, a series of 24 points randomly distributed in a 3 x 5 grid can be ordered to define a surface using the component *Surface From Points* (Surface > Freeform), by managing data.



In order to get a correct surface using *Surface from Points*, points need to be organized in rows or columns. If points are stored in a random way (as in our case) the resulting surface might look like the following.

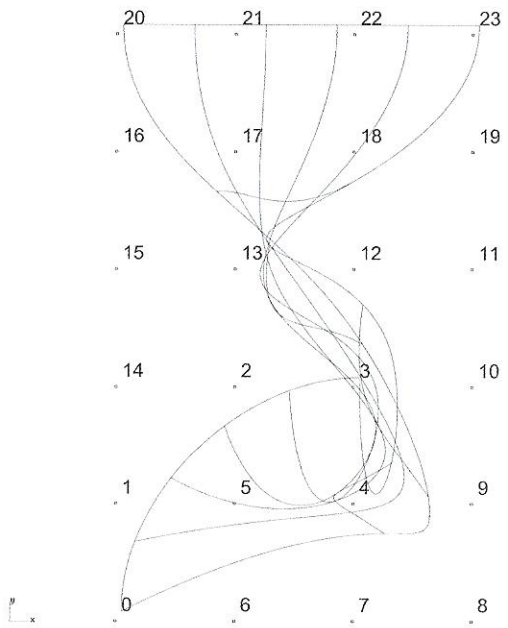


FIGURE 5.3
When points are not organized in rows or columns, *Surface From Points* component could generate incorrect results.

A strategy to reorganize the points into rows and columns is to define the sequence based on each points coordinates. The related algorithm consists of two steps:

1. Sorting the points according to the x-axis, yielding 4 different columns of points;
2. Sorting the points inside every column according to the y-axis.

STEP 1

The component *Sort List* (Sets > List) sort points according to each points position on the x-axis or by the x coordinate. *A-input is the list of points to rearrange, the K-input*, defined by using the component *Deconstruct* (Vector > Point), is the data used to sort.

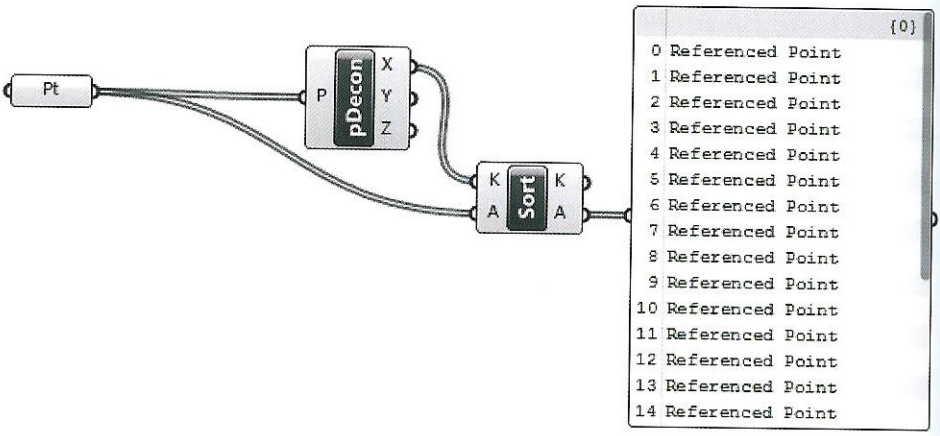
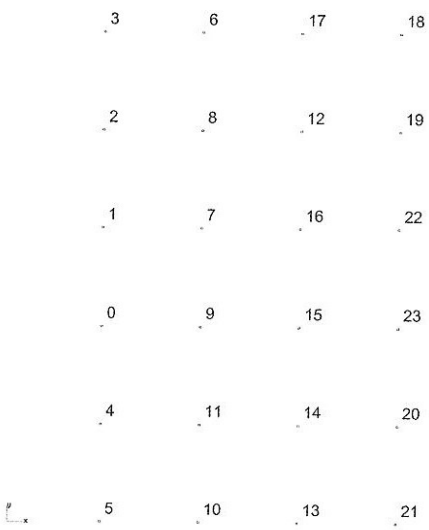


FIGURE 5.4
Sorting points according to the x-axis.

As result, the first set of points (0 - 5) are randomly hosted with respect to their y position within the first column, the second set of points (6 -11) are randomly hosted with respect to their y position within the second column, etc.



The output (A) of the *Sort List* component is a flattened list of points hosted in a trunk. The data is restructured into 4 branches with 6 index items per branch using the component ***Allocate N***.

The component *Allocate N,* part of the **Tree 8** plug-in developed by **Jissi Choi**, is available for free download[13]. The component *Allocation N* splits a flattened list into branches. In the example below 6 items have been allocated for each branch.

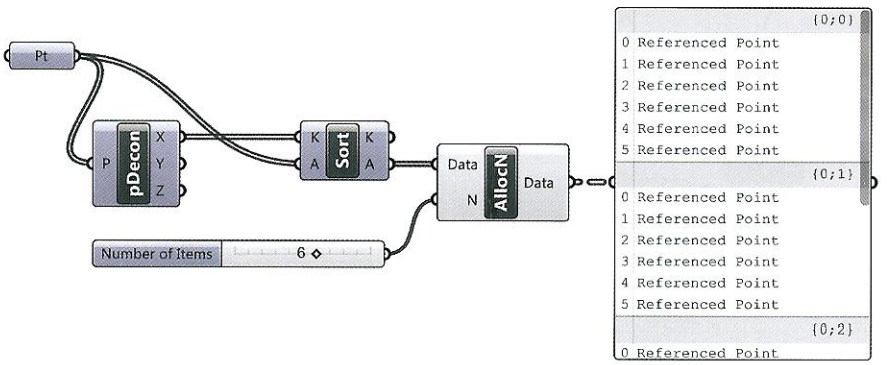
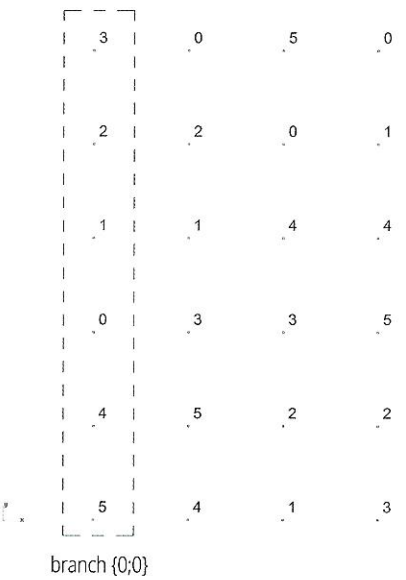


FIGURE 5.5
Splitting data using the *Allocate N* component. Each branch is related to a different column of the points grid.



NOTE 13
Tree8 can be download on Grasshopper's website. The component is a small part of STRAUTO, a parametric structural modeling tool based on Grasshopper, SAP2000, MIDAS. Website: http://tree8.chang-soft.co.kr/

STEP 2

The points returned by the Data-output of the *Allocate N* component is structured into 4 columns of six points randomly organized with respect to the y-axis. The final step is to sort the points according to the y coordinate.

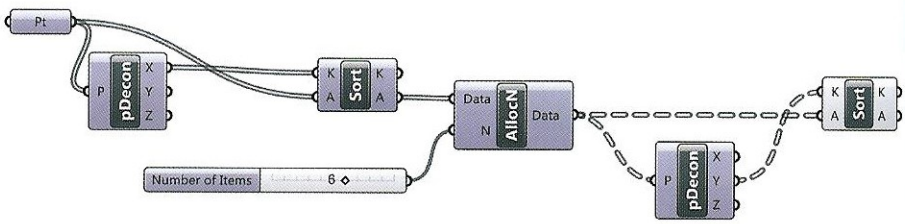
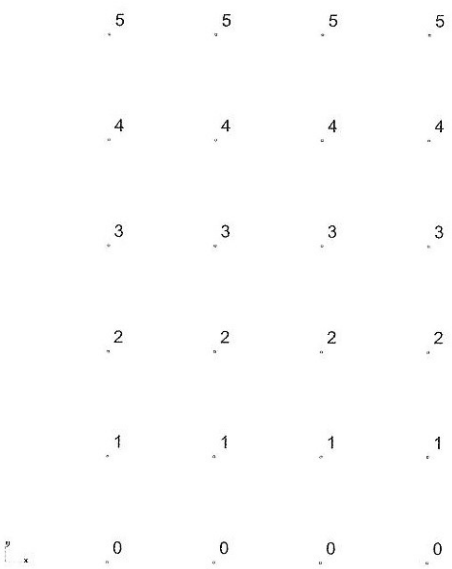
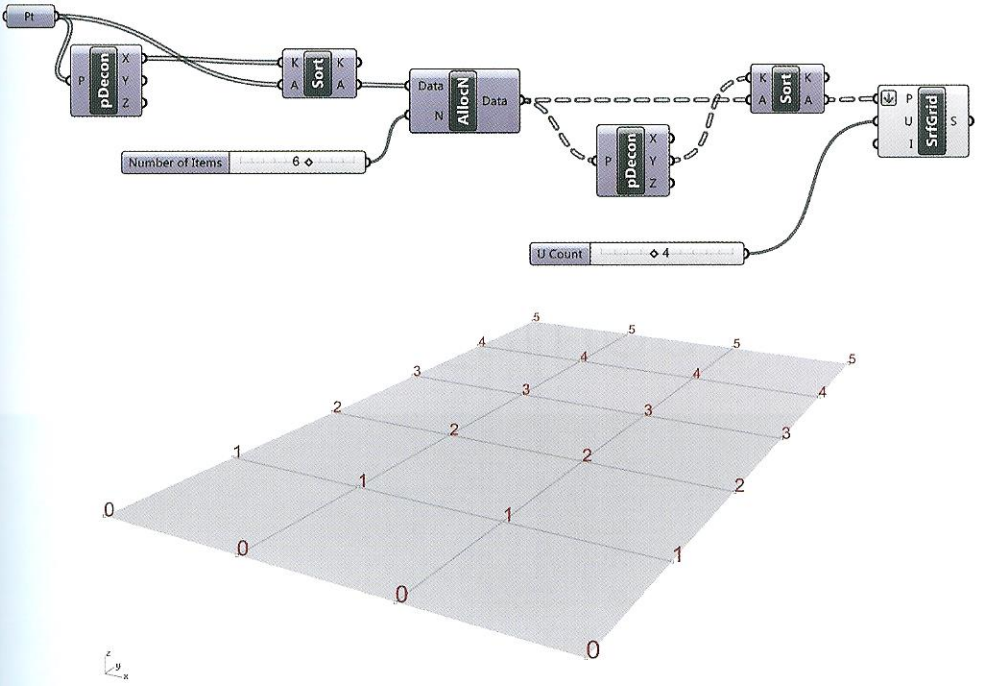


FIGURE 5.6
Points sorted according to the y-axis.



The points organized sequentially with respect to the x and y coordinates can be used to define a surface by means of the component *Surface From Points*. The input (P) of the *Surface From Points* component is set to *flatten* mode.



If we change the point's z coordinate position, the algorithm will return a surface through the adjusted points.