GENERATIVE ALGORITHMS

using GRASSHOPPER

ZUBIN KHABAZI

GENERATIVE ALGORITHMS

using GRASSHOPPER

ZUBIN KHABAZI

To see the latest updates visit my website or for enquiries contact me at:

www.MORPHOGENESISM.com

zubin.khabazi@googlemail.com

Introduction

Have you ever played with LEGO Mindstorms NXT robotic set? Associative modelling is something like that! While it seems that everything tends to be Algorithmic and Parametric why not architecture?

During my Emergent Technologies and Design (EmTech) master course at the Architectural Association (AA), I decided to share my experience in realm of Generative Algorithms and Parametric-Associative Modelling with Grasshopper as I found it a powerful platform for design in this way.

In this second edition, as I changed the name 'Algorithmic Modelling' to 'Generative Algorithms', I tried to update some of the experiments and subjects due to the changes happening to the work-in-progress project of Grasshopper. I hope this tutorial helps you to understand Generative Algorithms and delicate Grasshopper as well. I would try to keep updating whenever needed but consider that most of experiments and examples were established by previous versions of plug-in, so if you faced some differences it might be because of that.

Although I still believe that the book needs editorial review, since this is a non-profit, non-commercial product, please forgive me about that. I am very pleased that since publishing this book, I have found great friends worldwide, so feel free to contact me for any queries and technical issues.

Enjoy and Good luck!

Acknowledgements

First of all I would like to thank Bob McNeel for his support in Grasshopper3D and David Rutten for his inspiration and support as well. I also like to thank AA/EmTech directors and tutors Mike Weinstock, Michael Hensel and also Achim Menges who established my parametric and computational concepts. Many thanks to Stylianos Dritsas (AA/KPF) and Dr.Toni Kotnik (AA/ETH) for their computation, scripting and advance geometry courses.

I am extremely grateful to the students, architects and designers who contacted me and shared their knowledge and let me know short comes and errors of the work.

Zubin M Khabazi

March 2010

Contents

Chapter_1_Generative Algorithms

1_1_ Generative Algorithms

If we look at architecture as an object, represented in space, we always deal with geometry and a bit of math to understand and design this object. In the History of architecture, different architectural styles have presented multiple types of geometry and logic of articulation and each period has found a way to deal with its geometrical problems and questions. Since computers have started to help architects, simulate space and geometrical articulations, it became an integral tool in the design process. Computational Geometry became an interesting subject to study and combination of programming algorithms with geometry, yielded algorithmic geometries known as Generative Algorithms. Although 3D softwares helped to simulate almost any space visualized, it is the Generative Algorithm notion that brings the current possibilities of design, like 'parametric design' in the realm of architecture.

Architects started to use free form curves and surfaces to design and investigate spaces beyond the limitations of conventional geometries of the "Euclidian space". It was combination of Architecture and Digital that brought 'Blobs' on the table and pushed it further. Although the progress of the computation is extremely fast, architecture has been tried to keep track with this digital fast pace progress.

Contemporary architecture after the age of "Blob" seems to be more precise about these subjects. Architectural design is being affected by potentials of algorithmic computational geometries with multiple hierarchies and high level of complexity. Designing and modelling free-form surfaces and curves as building elements which are associated with different components and have multiple patterns is not an easy job to do with traditional methods. This is the power of algorithms and scripts which are forward pushing the limits. It is obvious that even to think about a complex geometry, we need appropriate tools, especially softwares, which are capable of simulating these geometries and controlling their properties. As a result, architects feel interested to use Swarms or Cellular Automata or Genetic Algorithms to generate algorithmic designs and go beyond the current pallet of available forms and spaces. The horizon is a full catalogue of complexity and multiplicity that combines creativity and ambition together.

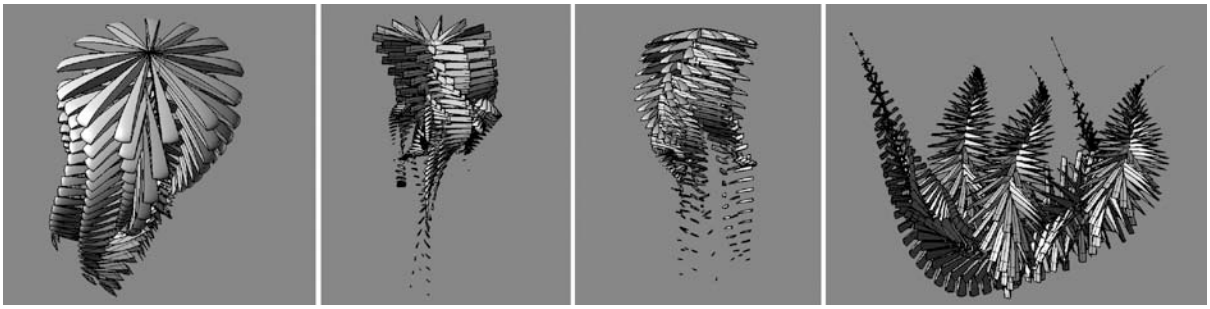


Fig.1.1. Parametric Modelling for Evolutionary Computation and Genetic Algorithm, Zubin Mohamad khabazi, Emergence Seminar, AA, conducted by Michael Weinstock, fall 2008.

A step even forward, now embedding the properties of material systems in design algorithms seems to be more possible in this parametric notion. Looking at material effects and their responses to the hosting environment in the design phase, now the inherent potentials of the components and systems should be applied to the parametric models of design. Not only these generative algorithms deal with form generation, but also there is a great potential to embed the logic of material systems in them.

"The underlying logic of the parametric design can be instrumentalised here as an alternative design method, one in which the geometric rigour of parametric modelling can be deployed first to integrate manufacturing constraints, assembly logics and material characteristics in the definition of simple components, and then to proliferate the components into larger systems and assemblies. This approach employs the exploration of parametric variables to understand the behaviour of such a system and then uses this understanding to strategise the system's response to environmental conditions and external forces" (Hensel, Menges, 2008).

To work with complex objects, a design process usually starts from a very simple first level and then other layers are added; complex forms are comprised of different hierarchies, each associated with its own logic and details. These levels are also interconnected and their members affect each other and in that sense this method called 'Associative'.

Generally speaking, Associative Modelling relates to a method in which elements of design being built gradually in multiple hierarchies and at each level, some parameters of these elements being extracted to be the generator for other elements in the next level and this goes on, step by step to produce the whole geometry. So basically the end point of one curve could be the center point of another circle and any change in the curve would change the circle accordingly. Basically this method of design deals with the huge amount of data and calculations and happens through the flow of algorithms.

The point is that all these geometries are easily adjustable after the process. Designer always has access to the elements of design product from start point up to details. Actually, since the design product is the result of an algorithm, inputs of the algorithm could be changed and the result would also be updated accordingly. It is now possible to digitally sketch a model and generate hundreds of variations of project by adjusting very basic geometrical parameters. It is also viable to embed the properties of material systems, fabrication constraints and assembly logics in parameters. It is also possible to respond to the environment and be associative in larger sense. *"… Parametric design enables the recognition of patterns of geometric behaviour and related performative capacities and tendencies of the system. In continued feedback with the external environment, these behavioural tendencies can then inform the ontogenetic development of one specific system through the parametric differentiation of its sub-locations" (Hensel, Menges, 2008).*

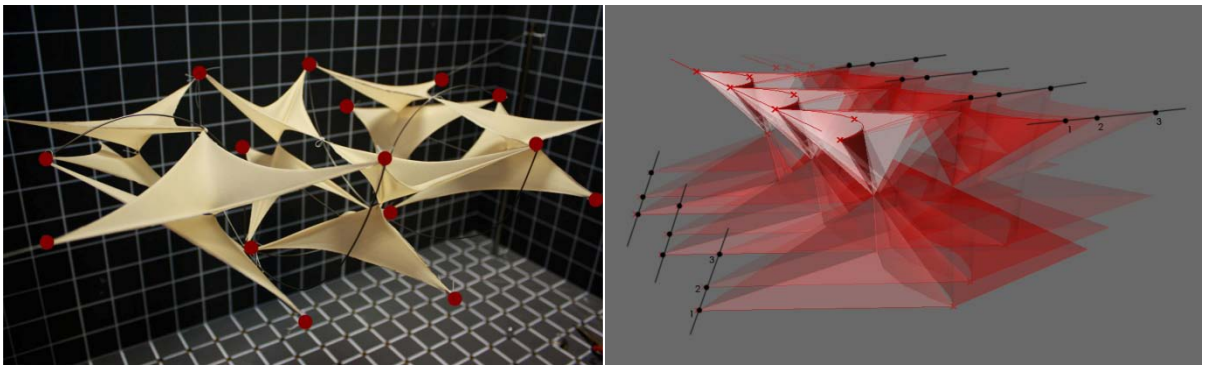


Fig.1.2. A. form-finding in membranes and minimal surfaces, physical model, B. membrane's movement modelled with Grasshopper, Zubin Mohamad Khabazi, EmTech Core-Studio, AA, Conducted by Michael Hensel and Achim Menges, fall 2008.

Grasshopper is a platform in Rhino to deal with these Generative Algorithms and Associative modelling techniques. The following chapters are designed in order to combine geometrical subjects with algorithms to address some design issues in architecture in an 'Algorithmic' method. The idea is to broaden subjects of geometry and use more commands and examples are designed to do so.

Chapter_2_The very Beginning

2_1_Method

This new edition of previously 'Algorithmic Modelling' and now '**Generative Algorithms**' is prepared due to my worldwide Grasshopper friends' questions and correspondence and also changes which were happened in Plug-in. Since Grasshopper is a work-in-progress project and improves and changes rapidly, it seems necessary to upgrade the book in this moment (and I am not totally sure by the time you receive it, another upgrade is needed or not!). You should consider that most of the experiments have been done by previous versions of the plug-in but I tried to update them wherever needed and I am sure that if you faced any difference, you can find your way through.

The main concept of the book is to focus on some geometrical and architectural problems and projects and to develop the understanding of Generative Algorithms, parametric modelling, based on design experiments instead of describing pure math or geometry. To do so, in most cases I assumed that you already know the basic understanding of ingredients of discussions and I would not go through the definition of the 'degree of a curve' although I will touch some, whenever necessary.

Grasshopper is fast growing and becoming a suitable platform for architects to design. More than a tool or software, it presents a way of thinking for design issues, a 'method' called Parametric or Associative these days. This method is developing by users all around the world as a practical example of distributed intelligence. Since these developments of the methods happen constantly and there are always upgrades to the software, and also interesting discussions, I would recommend checking the Grasshopper web page occasionally. By the way here in this chapter I would briefly discuss general issues of workplace and basics of what we should know in advance.

http://www.grasshopper3d.com/

2_2_Basics of Grasshopper

2_2_1_Interface, Workplace

Beside the other usual Windows menus, there are two important parts in the Grasshopper interface: Component Panels and Canvas. Component Panels provide all elements we need for our design and Canvas is the work place, where we put our Components and set up our algorithms. You can click on any object from Panels and click again on Canvas to bring it to work place or you can drag it on to the work place. Other parts of the interface are easy to explore and you will be familiar with them through using them later on. More information about this subject is also available at:

http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPluginInterfaceExplained.html

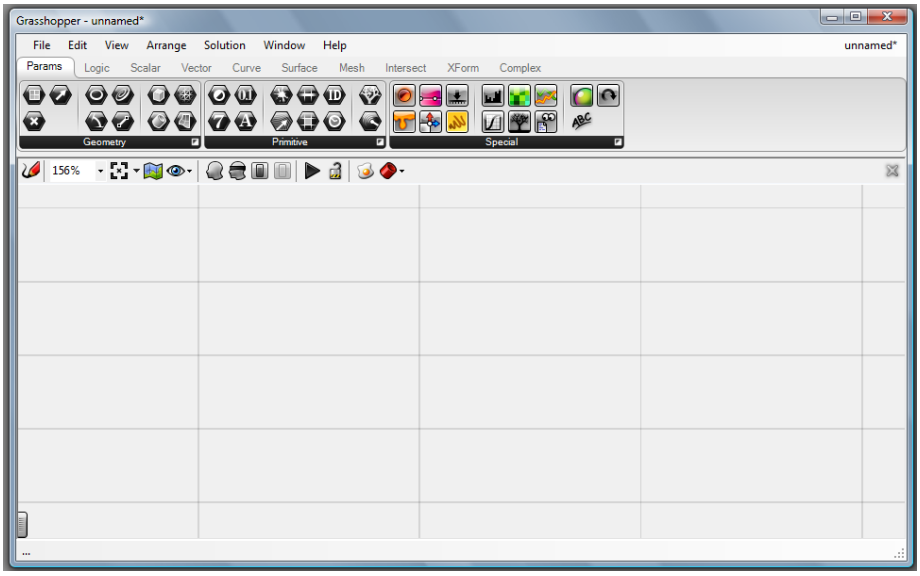
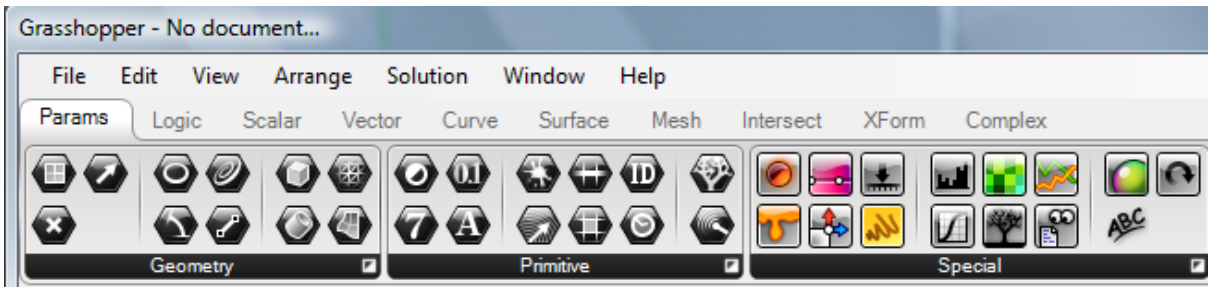


Fig.2.1. Grasshopper Component Tabs/Panels and Canvas

2_2_2_Components

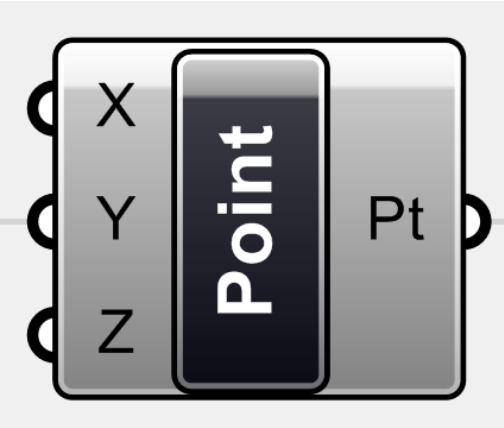
There are different types of objects in Grasshopper panels or components menu which we use to design stuff. You can find them under ten different tabs called: Params, Logic, Scalar, Vector, Curve, Surface, Mesh, Intersect, XForm and Complex.



Each tab has multiple panels and different objects, and commands are sorted between these panels. There are objects in these panels to draw geometries like lines and circles and there are also lots of commands to move, rescale, divide, deform and ... these geometries.

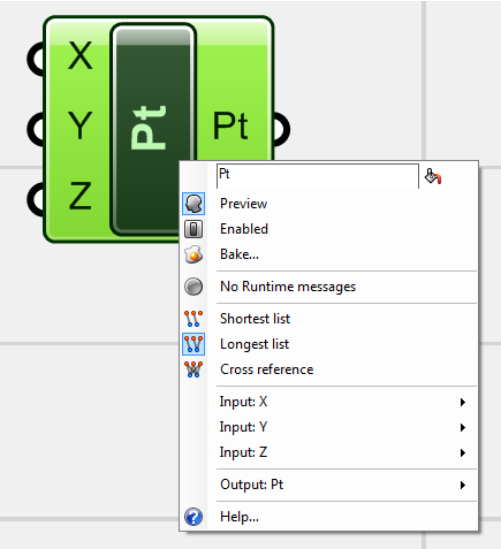
So some objects draw stuff and generate data, some of them manipulate an already existing geometry or data. Parameters are objects that represent data, like a point or line. You can draw them by relevant Parameters or you can define them manually from drawn objects of Rhino workplace. Components are objects that do actions like move, orientate, and decompose. We usually need to provide relevant data for them to work.

As I told, each of them has an object in Panels which you can bring to canvas to use. In this manual I used the term component to talk about any objects from the component panels to make life easier! and I always used <> to address them clearly in the text, like **<Point>**.



<Point> component

If you right-click on a component, a menu will pop-up that contains some basic aspects of the component. This menu called **"Context pop-up menu"**.



"Context pop-up menu"

From now on, you need to find relevant components from panels and set up connections between these components in order to generate your design algorithm and see the result in Rhino workplace. If Scripting is a coded and abstract version of algorithms, here in Grasshopper, the canvas represents a visual version of algorithms like Flowcharts which is more sensible and flexible in designer's hand.

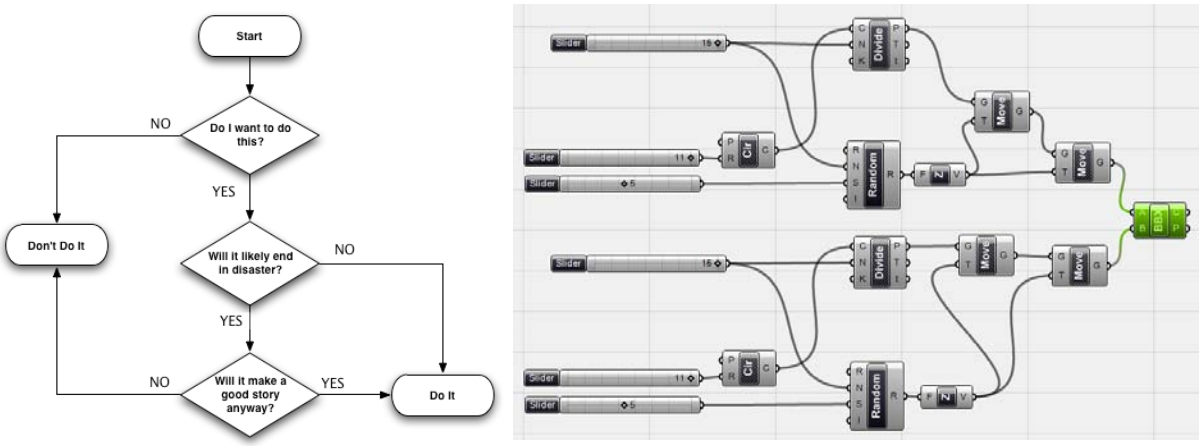


Fig.2.2. Flowchart vs. Grasshopper Algorithm

Defining External Geometries

Most of the time we start our design projects by introducing drawn objects from Rhino workplace to the Grasshopper canvas. It could be a point, a curve, a surface up to multiple complex objects. It means we can use our manually created objects or even script generated objects from Rhino in Grasshopper as external sources. Since any Geometry in Grasshopper needs a component in canvas to work with, we have to define our external geometries in canvas by relevant components. For this purpose we can look at the Params tab under Geometry panel. There is a list of different types of geometries that you can use to define your external object from Rhino workplace.

After bringing the proper geometry component to the canvas, define a Rhino object by right-click on the component (context menu) and use "set one ... / set multiple … " to assign abject to the component. Here you need to select your geometry from Rhino workplace. By introducing an object/multiple objects to a component it becomes Grasshopper object which we can use it for any design purpose.

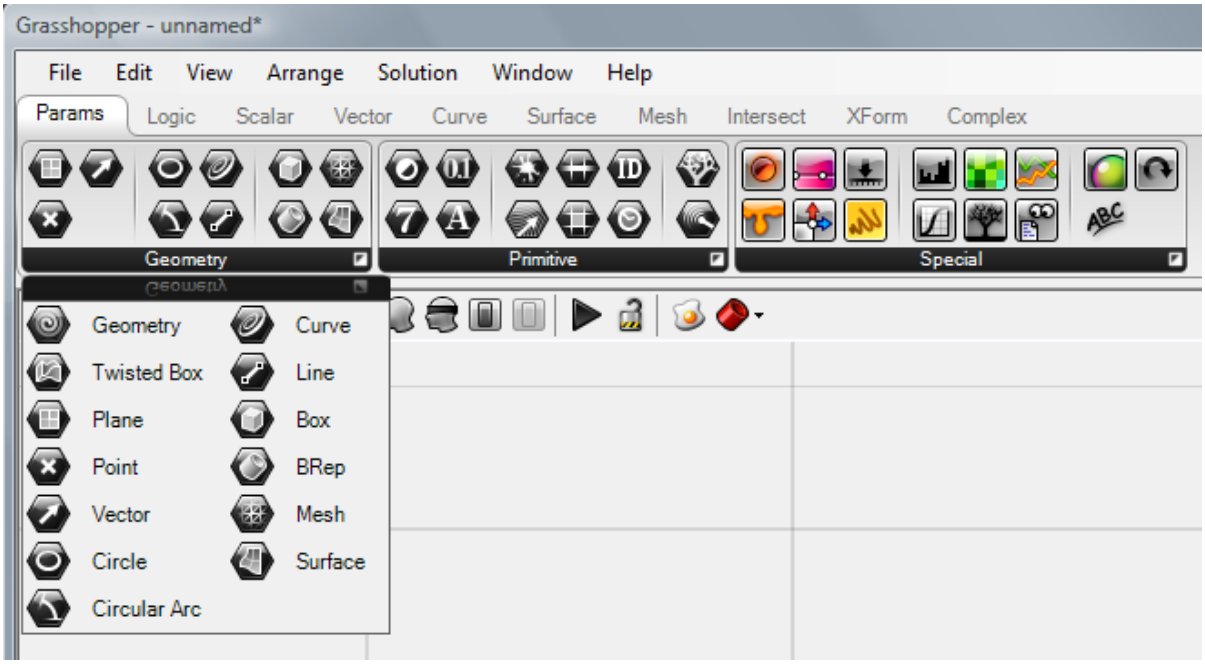


Fig.2.3. Different geometry types in the Params > Geometry panel

Let's have a simple example.

We have three points in Rhino viewport and we want to draw a triangle by these points in Grasshopper. First we need to introduce these points in Grasshopper. We need three <point> components from Params > Geometry > Point and for each we should go to their context menu (right click) and select 'set one point' and then select the point from Rhino viewport (Fig.2.4).

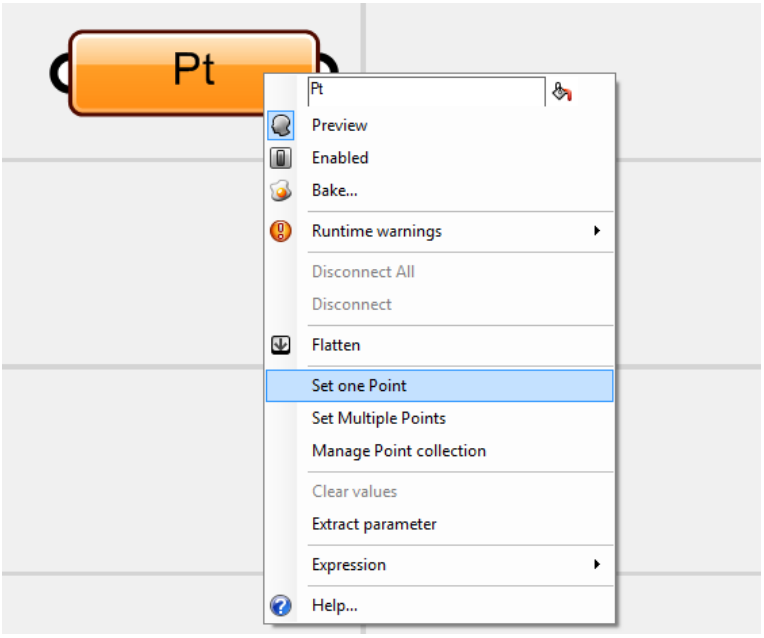


Fig.2.4. Set point from Rhino in Grasshopper component

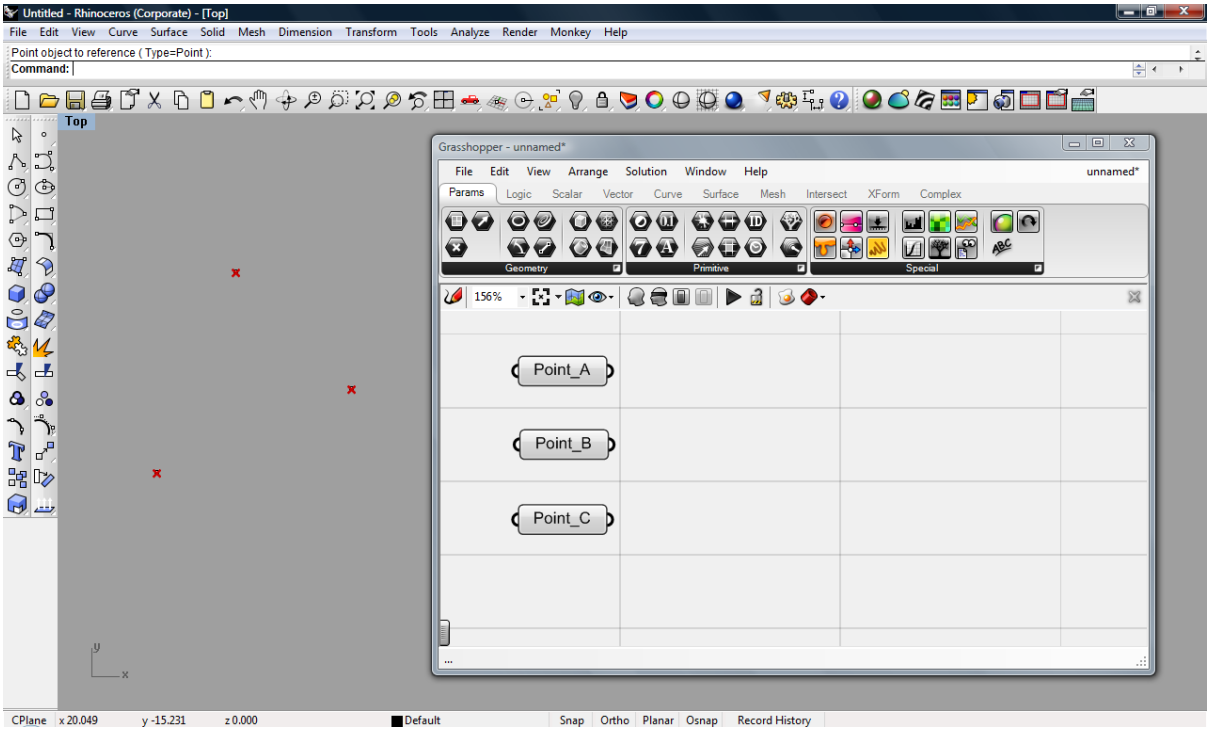


Fig.2.5. Grasshopper canvas and three points defined which turned to red crosses (x) in Rhino workplace. I renamed components to point A/B/C by the first option of their context menu to recognize them easier in canvas.

Components Connectivity

There are so many different actions that we can perform by components. Generally a component takes some data from one/multiple source and gives the result back. We need to connect the component which includes the input data to the processing component and connect the result to the other components that need this result and so on.

Going back to the example, now if you go to the Curve tab of components, in the Primitive panel you will see a <line> component. Drag it to the canvas. Then connect <point A> to the 'A' port of the <line> and <point B> to the 'B' port (to connect components, just click on the semi-circle at the right side of <point> and drag it up to the other semi-circle on the target (A/B input port of the <line>). You can see that Rhino draws a line between these points.

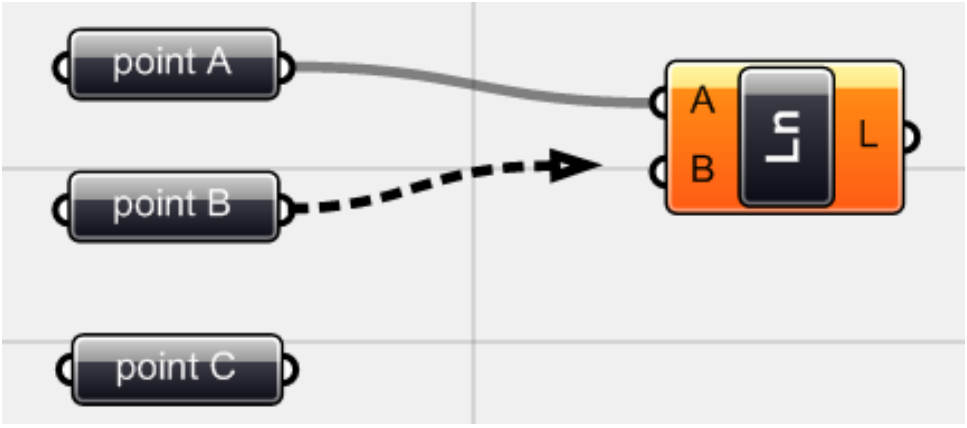


Fig.2.6. Connecting <point> components to a <line> component by dragging from output of the <point B> to the input of <line>.

Now add another <line> component for <point B> and <point C>. Do it again for <point C> and <point A> with the third <line> component. Yes! There is a triangle in Rhino viewport.

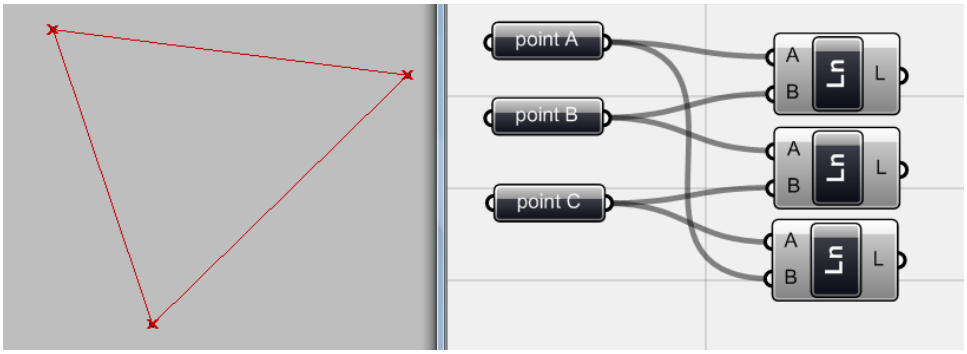


Fig.2.7. <line> components draw lines between <point> components. As you see any component could be used more than once as the source of information for other actions.

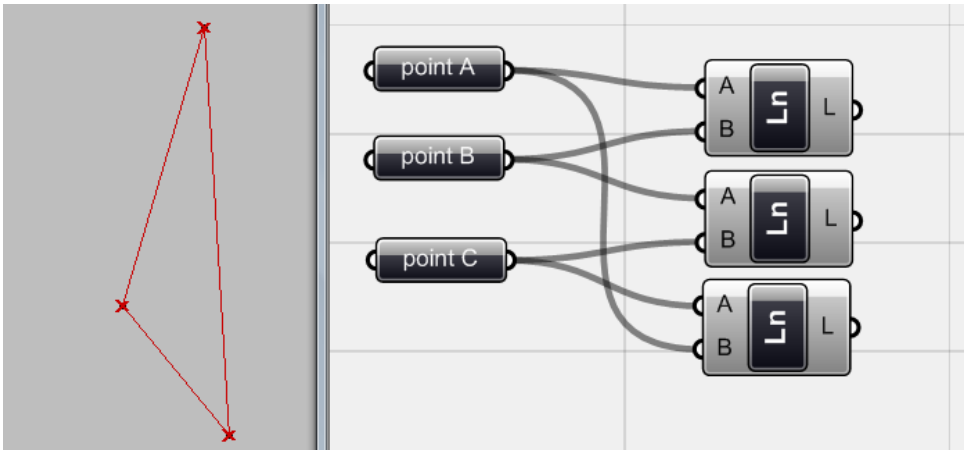


Fig.2.8. Now if you change the position of points manually in Rhino viewport, position of points in Grasshopper (X ones) and resultant triangle will change accordingly but lines between points (triangle) would remain.

As you can see in this very first example, associative technique made it possible to manipulate points and still have triangle between these points without further need to adjust them. So the idea is to prepare objects (feeding algorithm/input), set up relations between objects and add other manipulations to them (algorithm's function) and generate the design (algorithm output). We will do more by this concept to develop our understandings about Algorithms.

Input / Output

As mentioned before, any component in Grasshopper has input and output which means it processes the given data and gives the processed data back. Inputs are in left part of components and outputs at right. Data comes from any source attached to the input section of the component and output of the component is the result of that specific function.

There are some features on this subject that you can learn more at:
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryVolatileDataInheritance.html

You have to know that what sort of input you need for any specific function and what you get after that. We will talk more about different sort of data we need to provide for each component later on. Here I propose you to hold your mouse or "hover" your mouse over any input/output port of components. A tool-tip will pop-up and you will see the name, sort of data you need to provide for the component, is any predefined data there or not, and even what is it for.

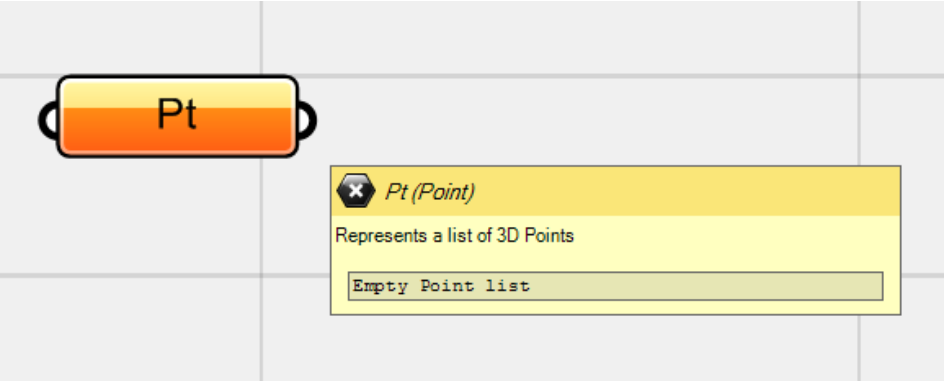


Fig.2.9. Pop-up tool-tip comes up if you hold your mouse over input/output port of the component.

Multiple connections

Sometimes you need to feed a component by more than one source of data. Imagine in the above example you want to draw two lines from <point A> to <point B> and <point C>. You can use two different <line> components or you can use one <line> component and attach both point B and C as the second point of the <line> component. To do this, you need to **hold Shift key** when you want to connect the second source of data to a component, otherwise Grasshopper would substitute it. When you hold shift, the arrow of the mouse appear in a green circle with a tiny (+) icon while normally it is gray. You can also use **Ctrl key to disconnect** a component from another one (normally you can disconnect a component from another one using context menu). In this case the circle around the mouse appears in red with a tiny (-) icon.

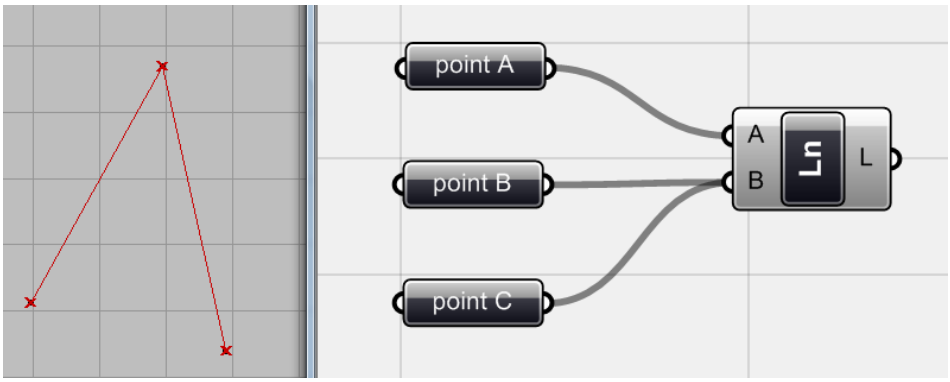


*Fig.2.10. Multiple connections for one component by holding **shift** key*

Colour Coding

There is a colour coding system inside Grasshopper which shows components working status.

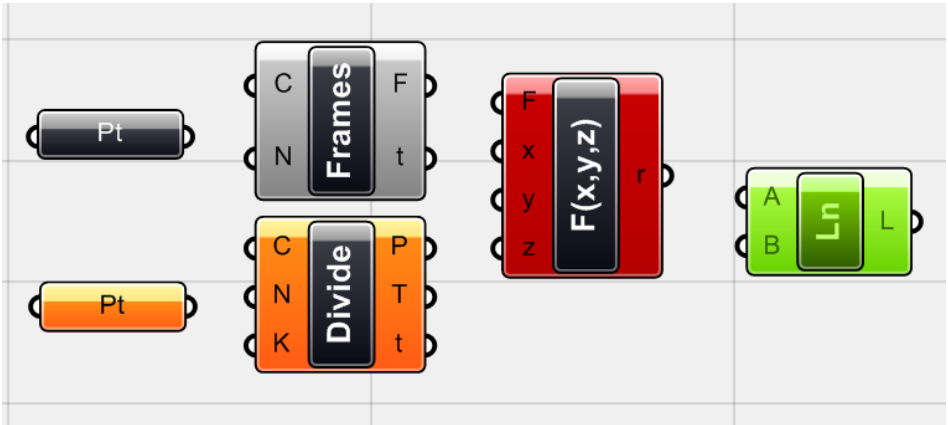


Fig.2.11. Colour Coding.

Any grey component means there is no problem and the data defined correctly/the component works correctly. The orange shows warning and it means there is at least one problem that should be solved but the component still works. The red component means error and the component does not work in this situation. The source of error should be found and solved in order to make component work properly. You can find the first help about the source of error in component's context menu (context menu > Runtime warning/error) and then search the input data to find the reason of error. The green colour means this component selected. The geometry which is associated with this component also turns into green in Rhino viewport (otherwise all Grasshopper geometries are predefined to red).

Preview

All components that produce objects in Rhino have 'Preview' option in their menu. We can use it to hide or unhide geometries in workplace. Any unchecked preview (Hidden output) make the component name becomes hatched. We usually use preview option to hide undesired geometries like base points and lines in complex models to avoid distraction. This option in complex models helps to process data faster, so please hide your base geometries when you do not need them to be seen.

2_2_3_Data matching

For many Grasshopper components it is always possible to provide a list of data instead of just one input. So in essence you can provide a list of points and feed a <line> component by this list and draw more lines instead of one. It is possible to draw hundreds of objects just by one component if we provide information needed.

Look at this example:

I have two different point sets, each with seven points. I used two <point> components and I used 'set multiple points' to introduce all upper points in one component and all lower ones in another component as well. As you see, by connecting these two sets of points to a <line> component, seven lines being generated between them. So we can generate more than one object with each component (Fig.2.12)

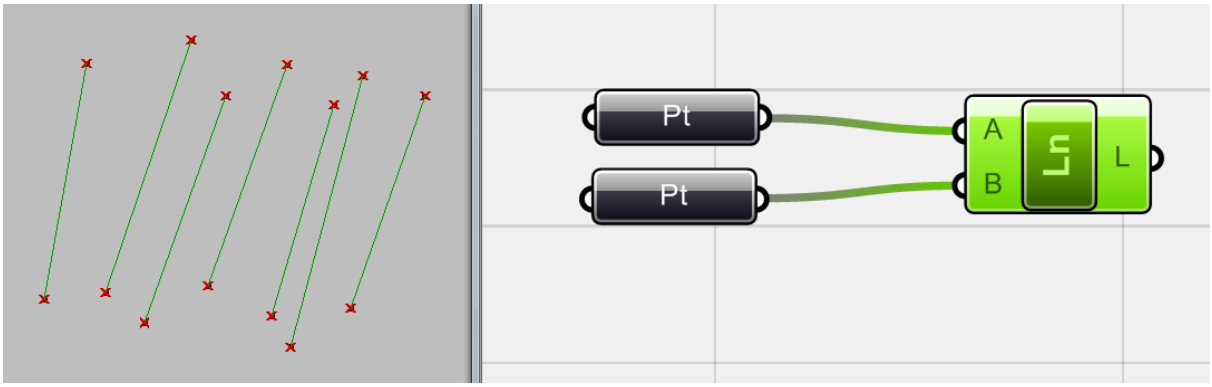


Fig.2.12. Multiple point sets and generating lines by them.

But what would happen if the number of points was not the same in two point (data) sets?

In the example below I have 7 points in top row and 10 points in the bottom. Here we need a concept in data management in Grasshopper called 'Data matching'. If you have a look at the context menu of the component you see there are three options called:

Shortest list

Longest list

Cross reference

Look at the difference in Figure 2.13.

It is clear that the shortest list uses the shortest data set to make lines, and the longest list uses the longest data set while uses an item of the shorter list more than once. The cross reference option connects any possible two points from lists together. It is very memory consuming option and sometimes it takes a while for the scene to upgrade changes.

Since the figures are clear, I am not going to describe more. For more information go to the following link:

http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryDataStreamMatchingAlgorithms.html

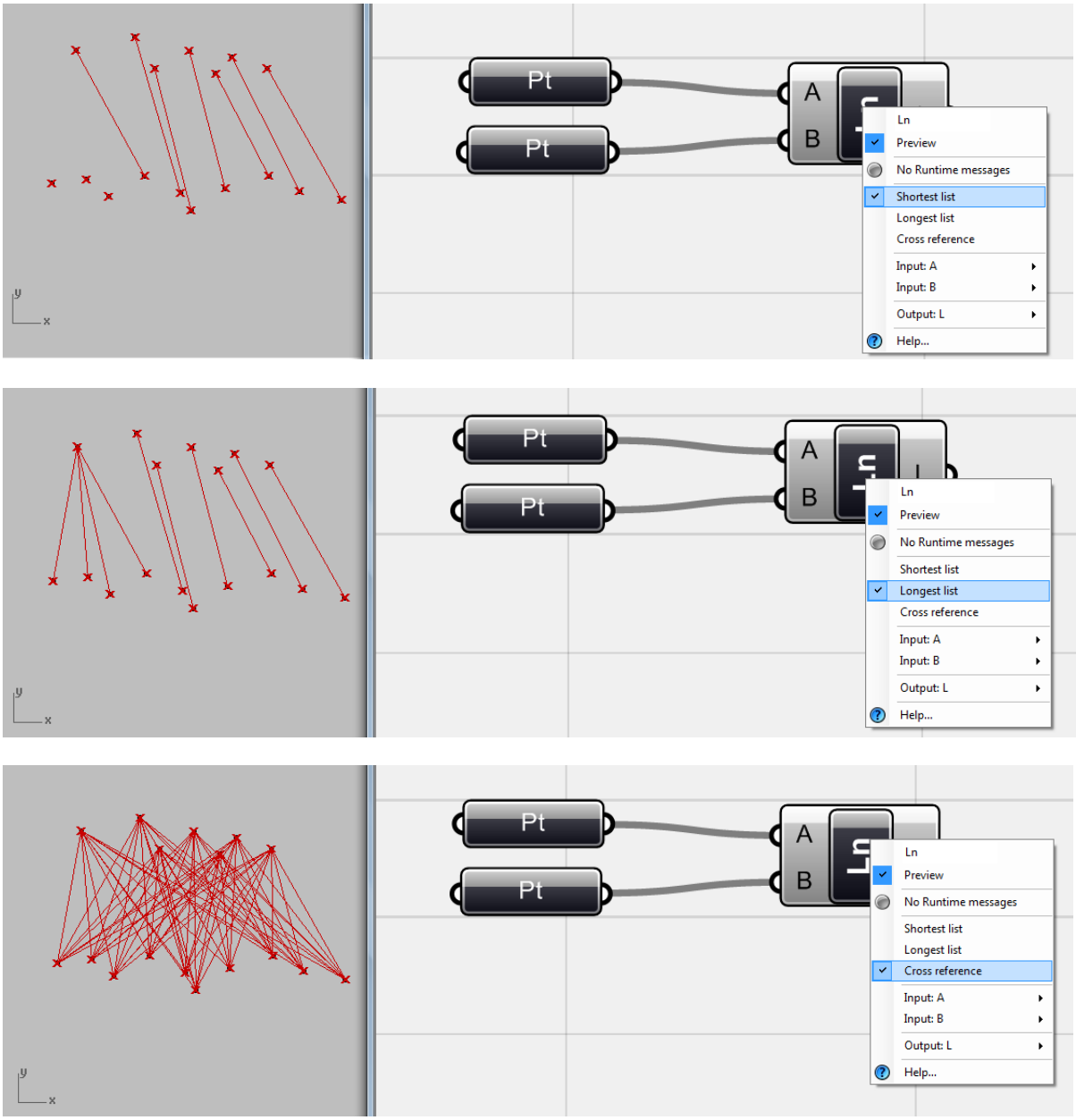


Fig.2.13. Data matching A: shortest list, B: longest list and C: cross reference

2_2_4_Component's Help (Context pop-up menu)

As it is not useful to introduce all components and you will better find them and learn how to use them gradually in experiments, I recommend you to play around, pick some components, go to the components context menu (right-click) and read their Help which is always useful to see how this component works and what sort of data it needs and what sort of output it provides. There are other useful features in this context menu that we will discuss about them later.

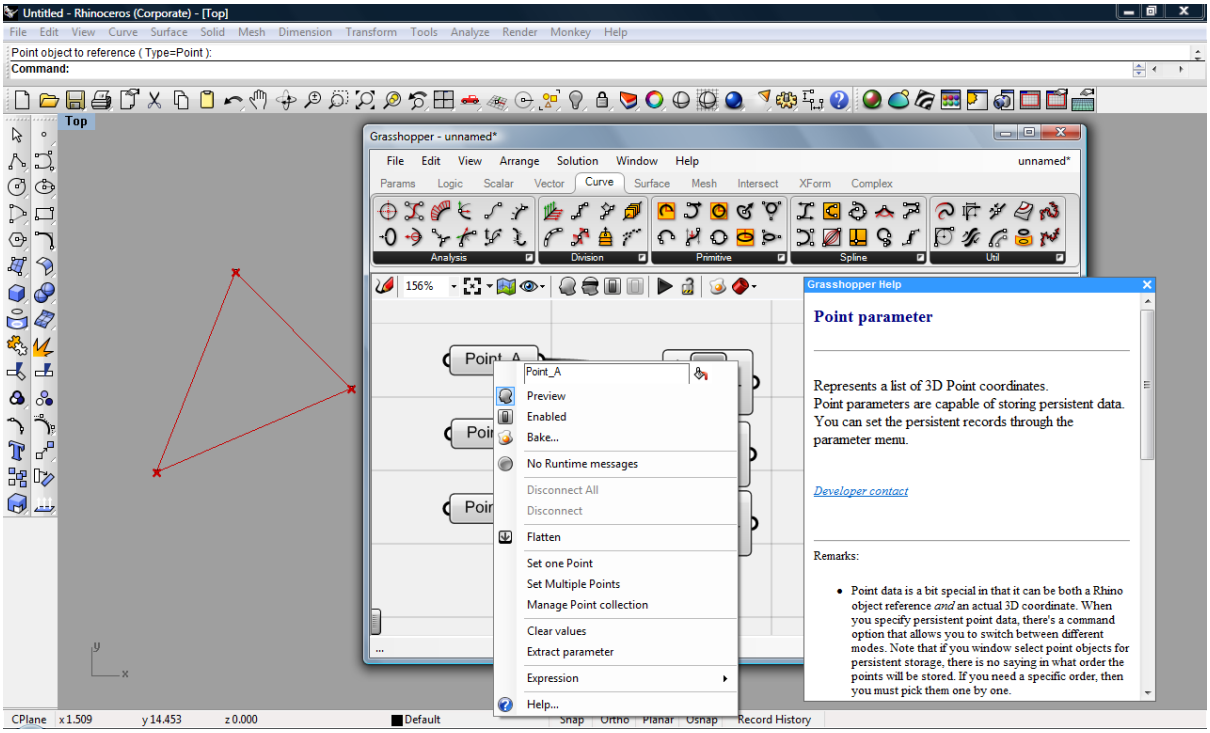


Fig.2.14. Context pop-up menu and Help part of the component

2_2_5_Type-In Component Search / Add

If you know the name of component you want to use, or if you want to search it faster than shuffling components' tabs and panels, you can **double-click on the canvas** and **type-in** the name of the component to bring it to the canvas. For those who used to work with keyboard entries, this would be a good trick!

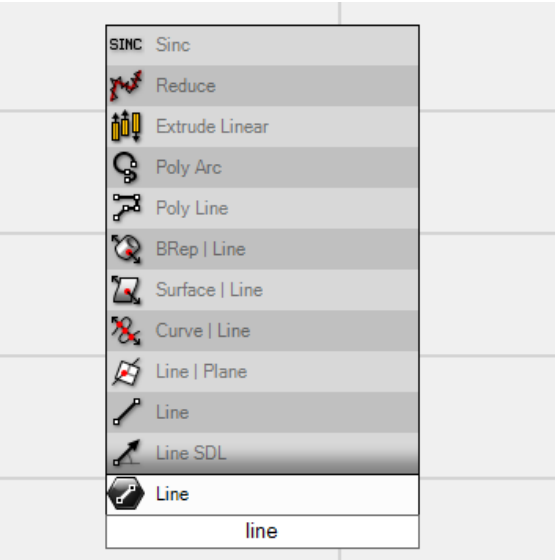


Fig.2.15. Searching for <line> component in the component-pop-up menu by double clicking on the canvas and typing the name of it. The component will be brought to the canvas.

Chapter_3_Data Sets and Math

Chapter_3_Data Sets and Math

Although in 3D softwares we can select our geometries from menus and draw them explicitly by clicking without thinking of the mathematical aspects behind, in order to work with Generative Algorithms, as the name sounds, we need to think a bit about data and math to make inputs of algorithm and generate multiple objects. Since we do not want to draw everything manually, we need some sources of data as the basic ingredients to make this generation possible and feed the algorithm to work more than once and result in more than one object.

The way in which algorithm works, the Workflow, is simple. It includes input of data, processing the data and output. This process happens in the whole algorithm or if we look closer, at each part of it. So instead of conventional method of drawing every object, we provide information, this information will process by the algorithm and the resultant geometry will be generated. As I said, for example, instead of copying an object by clicking 100 times in screen, we can tell the algorithm, copy an item for '100 times' in 'X positive direction' with the space of '3' between them. To do that you need to define the '100' as number of copying, and 'X Positive' direction and '3' as the space in between and the algorithm performs the job for you automatically.

All we are doing in geometry has a little bit of math behind. We can use these simple math functions in our algorithms with numbers and objects, to generate infinite geometrical combinations. It starts with numbers and numerical sets of data.

Let's have a look; it is easier than what it sounds!

3_1_Numerical Data Sets

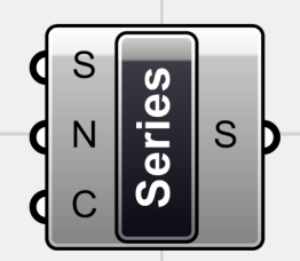
All math and algorithms start with numbers. Numbers are hidden codes of the universe. To begin, first of all we should have a quick look at numerical components to see how we can generate different numerical data sets in Grasshopper and then the way we can use them to design stuff.

One numerical value

The most useful number generator is <Number slider> component (Params > Special > Number slider) that generates one number which is adjustable manually. It could be integer, real, odd, even and with limited lower and upper values. You can set them all by 'Edit' part of the context menu.

For setting one fixed numeric value you can go to the Params > Primitive > Integer / Number to set one integer/real value through context menu of <Int>/<Num>.

Series of numbers



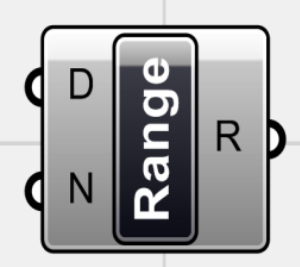
We can produce a list of discrete numbers by <series> component (Logic > Sets > Series). This component produces a list of numbers which we can adjust the first number, step size of the numbers, and the number of values.

0, 1, 2, 3, … , 100

0, 2, 4, 6, … , 100

10, 20, 30, 40, … , 1000000

Range of numbers



We can divide a numerical range between a low and high value by evenly spaced numbers and produce a range of numbers. We need to define an interval to set the lower and upper limit and also the number of steps between them (Logic > Sets > Range).

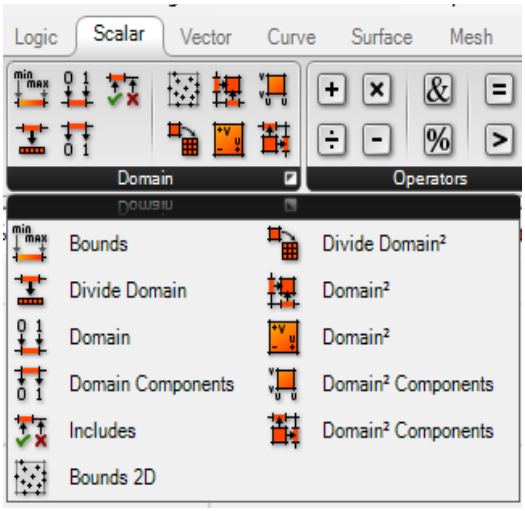
Any numeric interval (i.e. from 1 to 10) could be divided into infinite parts:

1, 2, 3, … , 10

1, 2.5, 5, … , 10

1, 5, 10

Domains (Intervals)



Domains ('Intervals' in previous versions) provide a range of all real numbers between a lower and upper limit. There are one dimensional and two dimensional domains that I will talk about them later. We can define a fixed domain by using Params > Primitive > Domain/Domain^2 component or we can go to the Scalar > Domain which provides a set of components to work with them in more flexible ways.

Domains by themselves do not provide numbers. They are just extremes, with upper and lower limits. As you know there are infinite real numbers between any two real numbers. We use different functions to divide them and use division factors as the numerical values.

To see the difference and usage lets go for some examples.

3_2_On Points and Point Grids

Points are among the basic elements for geometries and Generative Algorithms. As points mark a specific position in the space they can be start points of curves, centre of circles, origin of planes and so many other roles. In Grasshopper we can generate points in several ways:

- We can simply pick a point/bunch of points from the scene and introduce them to our workplace by <point> component (Params > Geometry > point) and use them for any purpose (These points could be adjusted and moved manually later on in Rhino scene and affect the whole project. Examples on chapter_2).

- We can introduce points by <point xyz> component (vector > point > point xyz) and feed the coordinates of the points by numbers. Or we can feed it by different datasets, based on our needs.

- We can make point grids by <grid hexagonal> and <grid rectangular> components.

- We can extract points from other geometries in many different ways like endpoints, midpoints, etc.

- Sometimes we can use planes (origins) and vectors (tips) as points to start other geometries and vice versa.

You have seen the very first example of making points in chapter_2 but let's have a look at how we can produce points and point sets by <series>, <range> and <number slider> components and other numerical data providers.

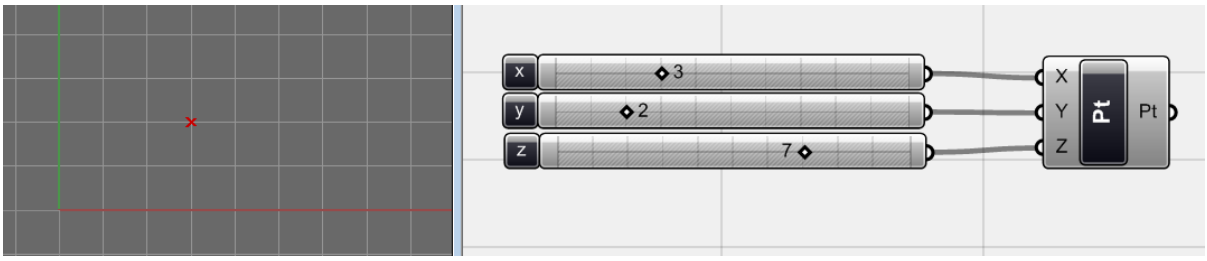


Fig.3.1. feeding a <point xyz> or <pt> component by three <number slider> to generate a point by manually feeding the X,Y and Z coordinates.

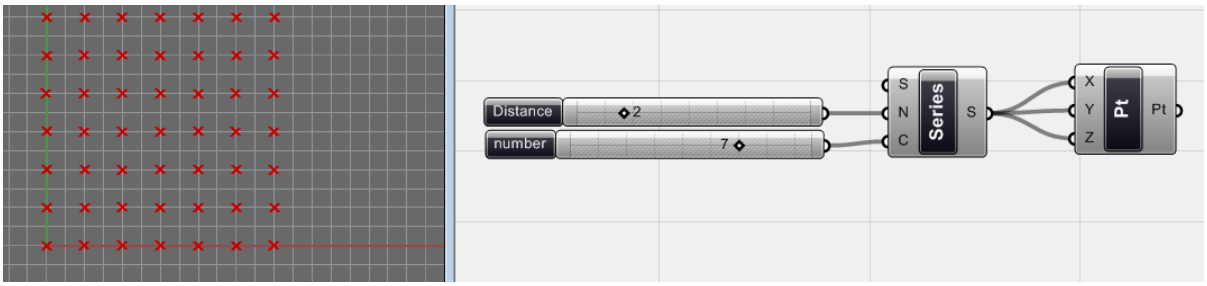


Fig.3.2. Generating a grid of points by <series> and <pt> components while the first <number slider> controls the distance between points (step size) and the second one controls the number of points in grid by controlling the number of values in <series> component (The data match of the <pt> set into cross reference to make a grid of points but you can try all data matching options).

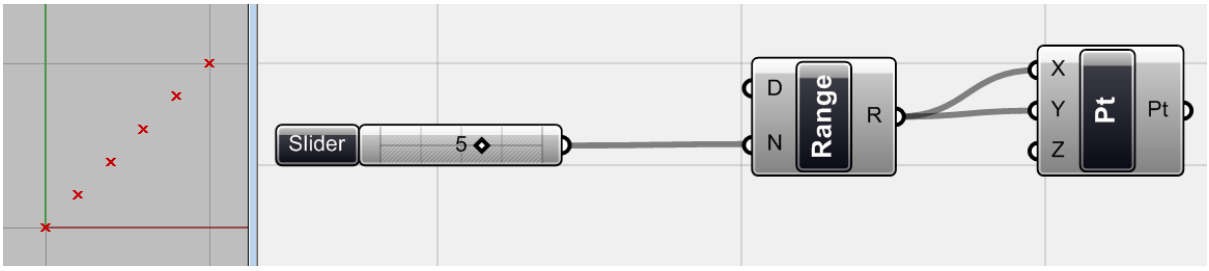


Fig.3.3. Dividing a numerical range from 0 to 1 by a manually controllable number (5) and feeding a <pt> component with 'Longest list' data match by these numbers. Here we divided the range by 5 so we have 6 points and all points drawn between the origin point(0,0) and point(1, 1) on the Rhino workplace (you can change the lower and upper limit of the <range> to change the coordinates of the points. To do so you need to right-click on the D part of the component (domain) and change the domain. There are other ways to work with intervals and change them which we will discuss later).

Since our first experiments sound easy, let's go further, but you can have your own investigations of these components and provide different point grids with different positions and distances.

3_3_Other Numerical Sets

Random Data Sets

I am thinking of making a randomly distributed set of points for further design issues. All I need is a set of random numbers instead of <series> to feed <pt> component. So I pick a <random> component from Logic > sets. A <random> component provides a list of random numbers and we can control the number of values and domain of them. But the <random> component produces one set of random numbers and I don't want to have same numbers for all X,Y and Z coordinates. To avoid same values, I need different random numbers for each. I need to provide three lists of random numbers either by three <random> components with different seeds (by feeding <random> component's (S) port with different numbers, to generate different random values otherwise all <random> components would generate same values) or to shuffle the current list of numbers.

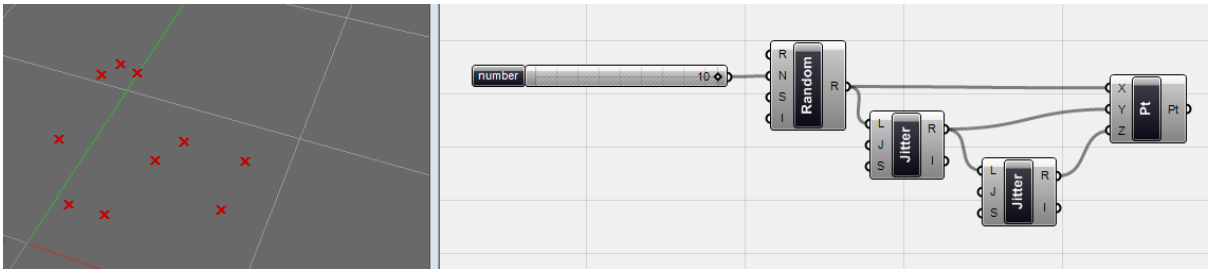


Fig.3.4. Generating a random point set. The <random> component produces 10 random numbers which is controlled by <number slider> and then this list is shuffled by <jitter> component (Logic > Sets > Jitter) for Y coordinates of the points once, and again for Z coordinates, otherwise you could see some sort of pattern inside your grid (attach the <random> to X, Y and Z of the <pt> without <jitter> and check it!). The data match set to longest list.

In figure 3.4 all points are distributed in the space between 0 and 1 of the coordinate system for each direction. To change the distribution area of the points we should change the numerical domain in which <random> component produces numbers. This is possible by manually setting the "domain of random numeric range" on Rhino command line if you right-click on (R) port (random numbers domain) of component or by defining the domain intervals adjustable by sliders. (Fig.3.5)

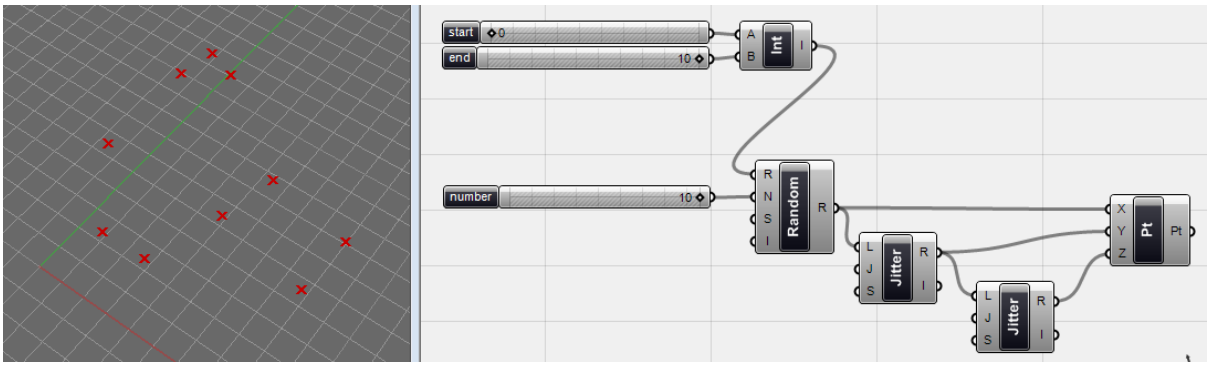
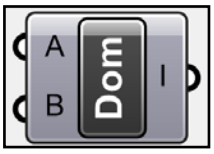


Fig.3.5. Setting up a domain by an <interval> component (Note: from now on please use Scalar > Domain > Domain in new version of Grasshopper instead of <interval>) to change the distribution area of points (look at the density of scene's grid in comparison with Fig.3.4).



Fibonacci series

What about making a point grid with non-evenly spaced, and increasing values? Let's have a look at available components. We need series of numbers which grow rapidly and under Logic tab and Sets panel we can see a <Fibonacci> component.

A Fibonacci is a series of numbers with two first defined numbers (0 and 1) and the next number is the sum of two previous numbers.

N(0)=0, N(1)=1, N(2)=1, N(3)=2, N(4)=3, N(5)=5, … , N(i)=N(i-2)+N(i-1)

Here are some of the numbers of the series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

As you see numbers grow rapidly.

Here I used <Fibonacci> series (Logic > Sets > Fibonacci) to produce incremental numbers to feed a <pt> component with them.

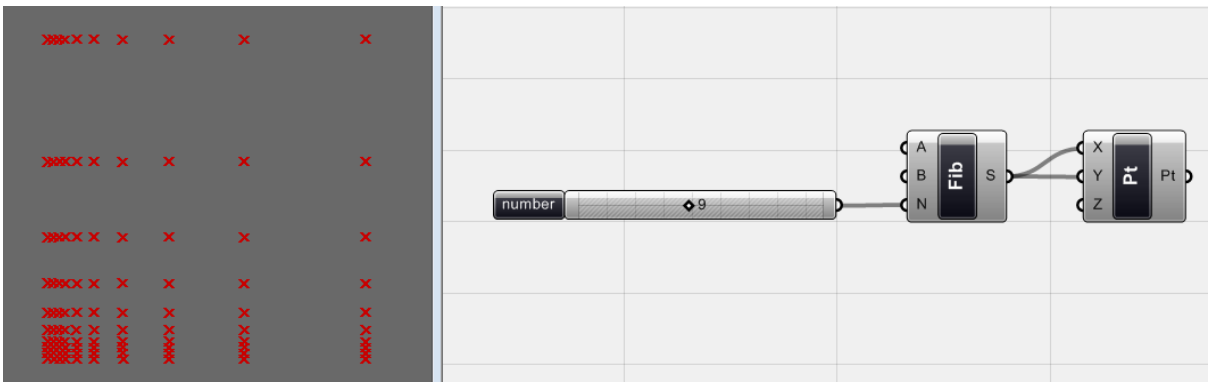


Fig.3.6. Using <Fibonacci> series to produce increasing distances (non-evenly spaced series of numbers) to generate points. The number of points could be controlled with a <number slider>.

3_4_Functions

Predefined components in Grasshopper might not be always your best way of designing stuff. You might need to generate your own data sets or at least manipulate the data of existing components. To do so, you need to use math functions and change the power, distance, … of numbers. Functions are components which are capable of performing math functions in Grasshopper. There are functions with different variables (Logic > script). You need to feed a function with relevant data (not always numeric but also Boolean, String) and it performs a user defined function on the input data. To define the function you can right-click on the (F) part of the component and type it or go to the Expression Editor. Expression Editor has so many predefined functions and a library of math functions for help.

Pay attention to the name of variables you use in your expression and the associated data you match to the function component!

Math functions

As mentioned before, using a predefined component is not always what we aimed for, but in order to get the desired result we can use mathematical functions to change the data sets and feed them to generate geometries.

A simple example is the mathematical function of a circle that is X=Sin(t) and Y=Cos(t) while (t) is a range of numbers from 0 to 2Pi. I produce it by a <range> of numbers which starts from 0 to 1 with N numbers in between, times 2Pi by <function> component. This would result a range of numbers from 0 to 2pi that makes a complete circle in radian.

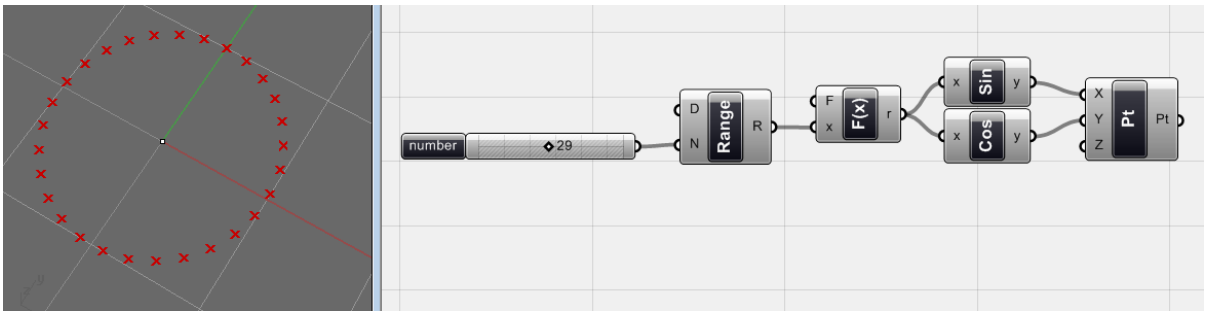


*Fig.3.7. Parametric circle by mathematical functions. You have <Sin> and <Cos> functions in the Scalar > Trig. (F(x) = x * 2Pi).*

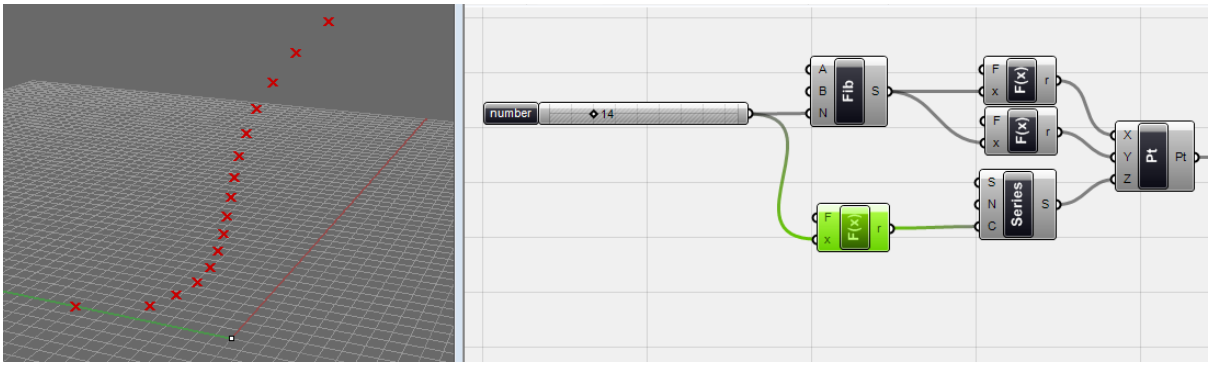


Fig.3.8. More experiments. Series of points which are defined by <Fibonacci> series and simple mathematical functions (x: F(x)=x/100, y: F(x)=x/10). The selected green F(x) is a function to add 2 to the value of <number slider> (F(x)=x+2) in order to make the values of <series> equal to the Fibonacci numbers (Fibonacci has two extra first values). The aim is to show you that we can simply manipulate these data sets and generate different geometries accordingly.

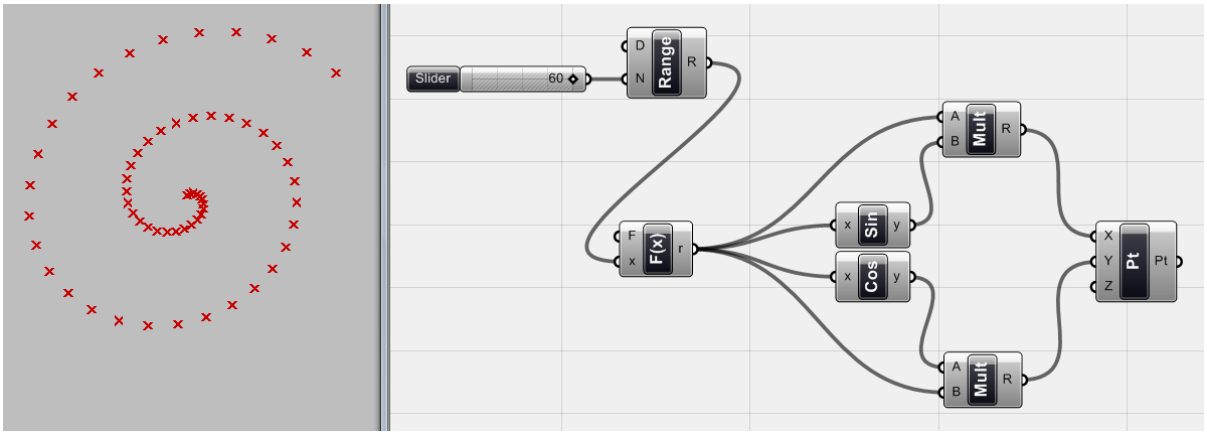


Fig.3.9. A <range> of numbers from 0 to 2 which times by 2Pi with <Function> that makes it a numerical range from 0 to 4Pi. This range divided into 60 parts. The result feeds the <pt> component by the following math function:

*X=t * Sin(t), Y=t * Cos(t)*

*You know <sin> and <cos> components. To apply t*sin/cos I used <multiplication> component from Scalar>Operators. There you could find components for simple math operations as well.*

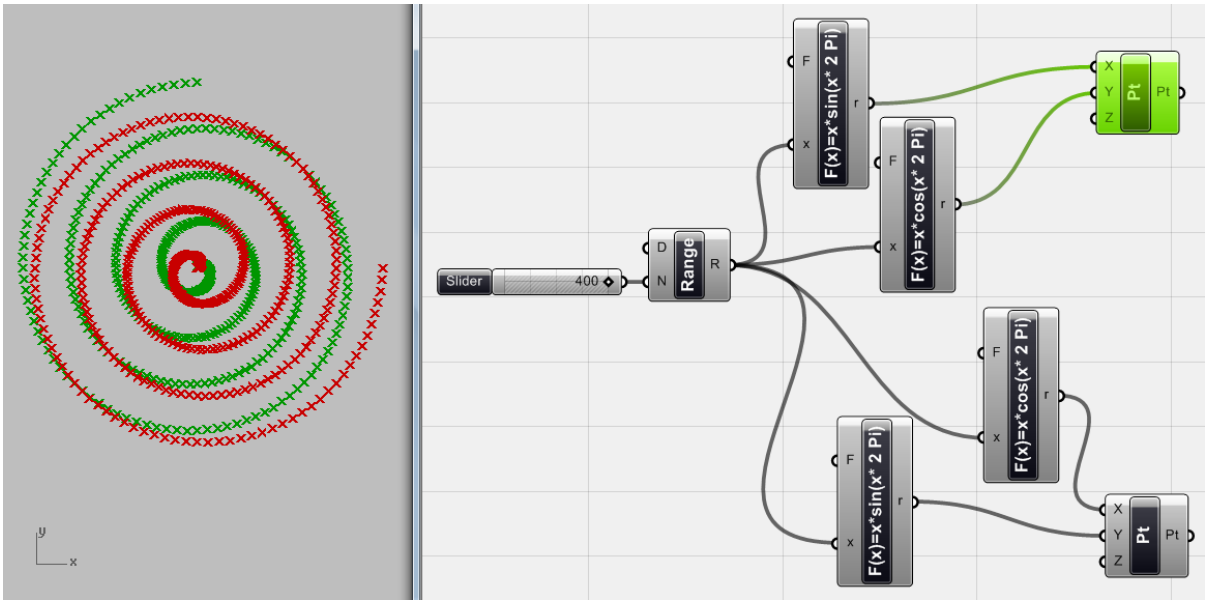


Fig.3.10. A complex one! Inter tangent spirals from two inverted spiral point sets. <range> interval is from 0 to 4 which is divided into 400 points and then multiplied by functions:

*First <pt>: X: F(x) = x * Sin(x*2 Pi), Y: F(x) = x * Cos(x * 2 Pi)*

Second <pt> has the same functions but inverted.

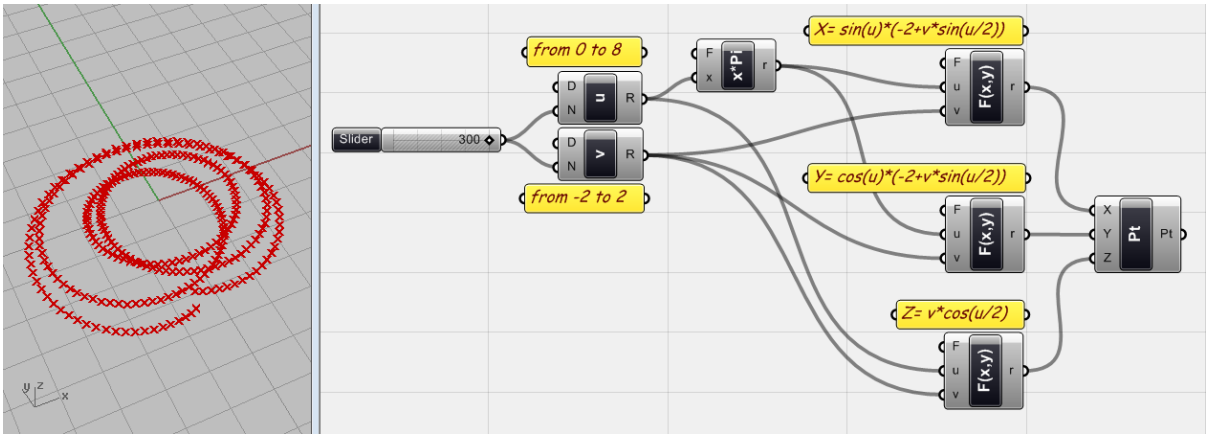


Fig.3.11. Moebius by point sets. <u> and <v> are <range> components which renamed. The numeric domain of each one presented in the scene. The math function to generate Moebius is:

X= Sin(u)(-2+v*sin(u/2))*

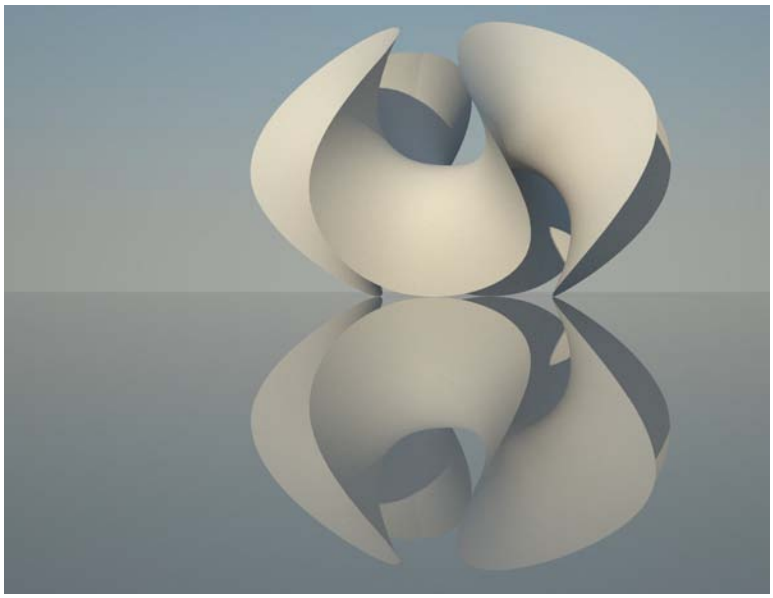
Y= Cos(u)(-2+v*sin(u/2))*

*Z= v*Cos(u/2)*

(While u=0 to 8Pi and v=-2 to 2 which created by function components and all feed <pt> component.)

Playing around math functions could be endless. You can find so many mathematical resources to match your data sets with them. The important point is that you can manipulate the original data sets and generate different numerical values and feed other components by them.

So as you see by simple sets of numerical data we can start to generate geometries and this is how algorithms work. From now on, we need to build up our knowledge based on various geometrical concepts in algorithmic method to deal with problems and design issues, like this very beautiful, Enneper surface, (by Rhino's Math function plug-in):



3_5_Boolean Data types

Data is not limited to Numbers. There are other data types which are useful for different purposes in programming and algorithms. Since we are dealing with algorithms, we should know that the progress of an algorithm is not always linear. Sometimes we want to decide whether to do something or not. Programmers call it conditional statements. We want to see if a statement meets certain criteria or not. The response of a conditional 'question' is a simple yes or no. in algorithms we use Boolean data to represent these responses. Boolean data types represent only True (yes) or False (no) values. If the statement meets the criteria, the response is **True**, otherwise **False**. As you will see later, this data type is very useful in different cases when you want to decide about something, select some objects by certain criteria, sort objects, etc.

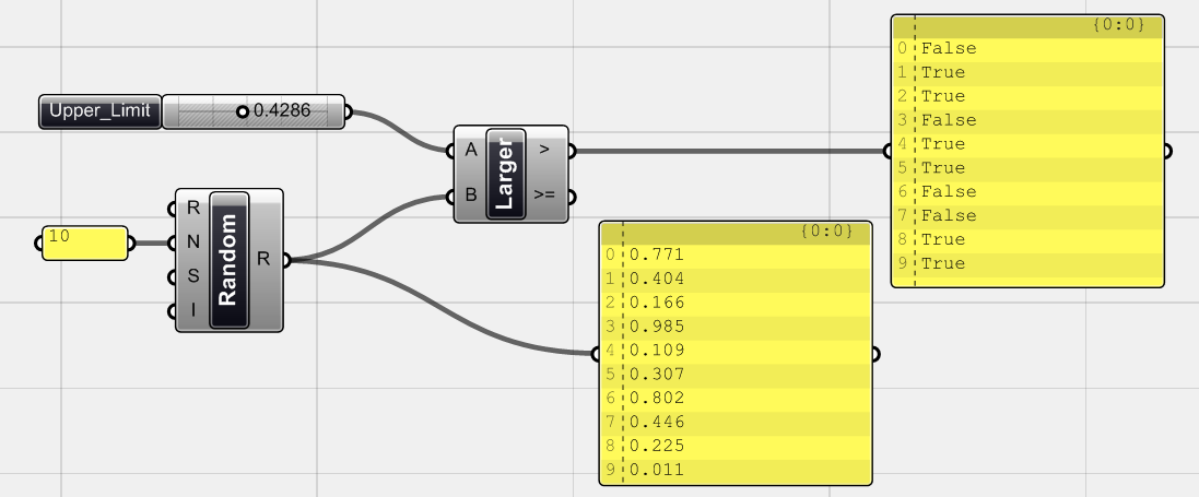


Fig.3.12. Here I generated ten <random> values and by a <Larger> component (Scalar>Operators) I want to see if these numbers are less than a certain <Upper_limit> (any value by a <number slider>) or not. As you see whenever numbers meet the criteria (means it is smaller than the <Upper_limit>), the <Larger> passes 'True' as a result, otherwise 'False'. Here I used <Panel> components form Params > Special to show the contents of the <Random> and the result of the <Larger> component.

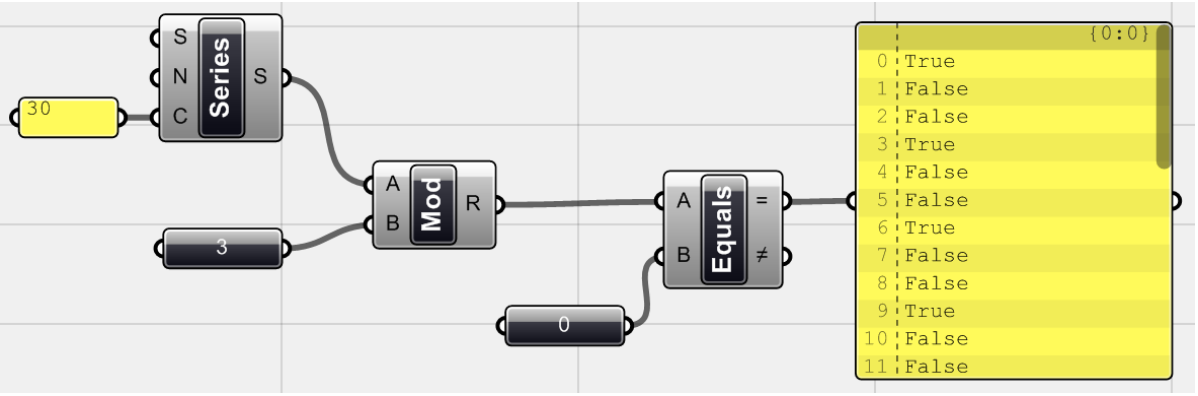


Fig.3.13. For the next step, I generated 30 values with a <series> component and I used a <Modulus> component (Scalar > Operators > Modulus) to find the remainder of the division of the numeric values by <3> and I passed the result to an <Equals> to see if these remainders = 0 or not. As you see the result is another <panel> of True/False values.

So as you can see in these examples, there are different possibilities to check criteria by numeric values and get Boolean values as result. But sometimes, we want to see if the situation meets different criteria, and we want to decide based on the result of them. For example based on the above experiments, we want to see whether a value is smaller than a certain upper_limit and at the same time it is dividable by 3. To know the result, we need to operate on the result of both functions which means we need to operate on Boolean values. If you check, under the Logic tab and in Boolean panel there are various components that work with Boolean data type.

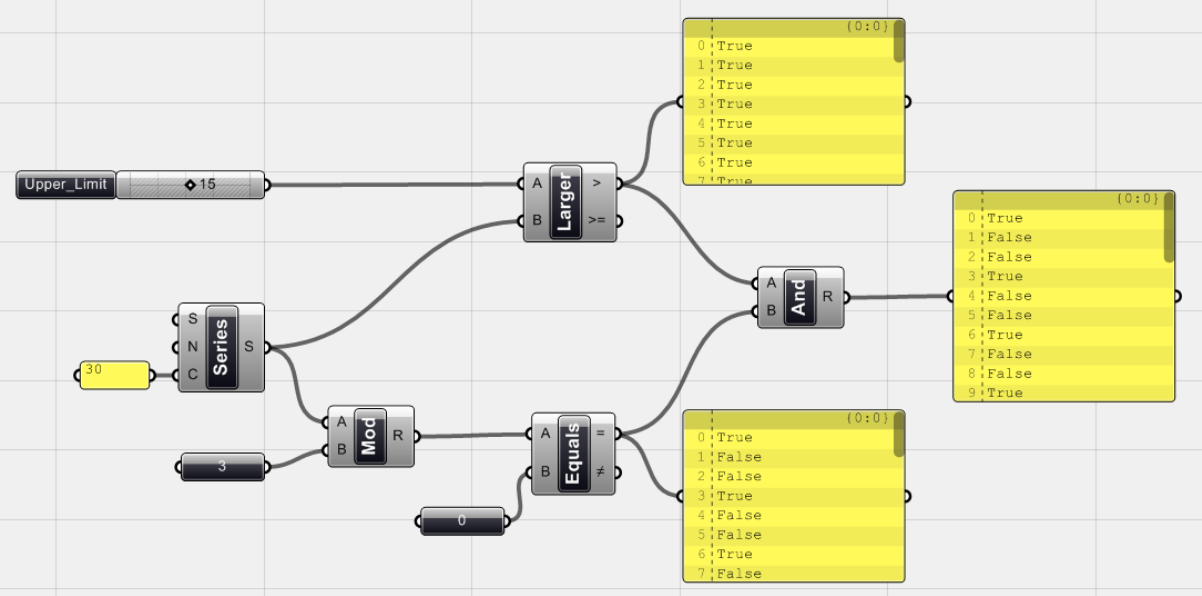


Fig.3.14. Here I combined both concepts. I used a <Gate And> component (Logic > Boolean > Gate And) and I attached both <function>s to perform Boolean conjunction on them. The result is True when both input Boolean values are True, otherwise it would be False. As you see, those numerical values which are smaller than the <Upper_limit> and dividable by 3 are meeting the criteria and pass True at the end.

There are multiple Boolean operators on Boolean panel of the Logic tab that you can use and combine many of them to create your criteria, make decisions and build up your design based on these decisions. We will discuss how to use these Boolean values later.

3_6_Cull Lists

There are many reasons that we might want to select some of the items from a given data set and do not apply a function to all of them. To do this, we either need to select some of the specific items from a list or omit other items. There are different ways to achieve this but let's start with omitting or culling lists of data.

Up to now there are three <cull> components to cull a list of data in Grasshopper. While <cull Nth> omit every N item of a given list of data, <cull pattern> takes a pattern of Boolean values

(True/False) and cull a list of data, based on this pattern, means any item of the list that associates with True value in Boolean list passes and those that associate with False, omit from the list. <Cull index> culls a list of data by index numbers.

If the number of values in the data list and Boolean list are the same, each item of the data list being evaluated by the same item in the Boolean list. But you can define a simple pattern of Boolean values (like False/False/True/True which is predefined in the component) and <cull> component would repeat the same pattern for all items of the data list.

For better understanding, here I want to introduce some of the ways we can select our desired geometries (in this case points) out of a predefined data set.

Distance example

I am thinking of selecting some points from a point set based on their distance to another point (reference point). Both point set and the reference point are defined by <point> component. First of all what I need is a <distance> component (Vector > Point > Distance) that measures the distance between points and the reference and as a result it provides a list of numbers (distances). I compared these distances by a user defined number (<number slider>) with a <F2> component (Logic > Script > F2 / function with two variable). This comparison generates Boolean values as output (True/False) to show whether the value is smaller (True) or bigger (False) than the upper limit F=x>y (this is the same as <Larger> component). I am going to use these Boolean values to feed the <Cull pattern> component.

As mentioned before, <Cull pattern> component takes a list of generic data and a list of Boolean data and omits those members of the generic list of data who associate with 'False' value of the Boolean list. So in this case the output of the <Call pattern> component is a set of points that associate with True values which means they are closer than the specified number shown on the <number slider>, to the reference point, because the X>Y function always pass True for the smaller values of Y which means smaller distances (y=Distance). To show them better I just connected them to the reference point by a simple <line>.

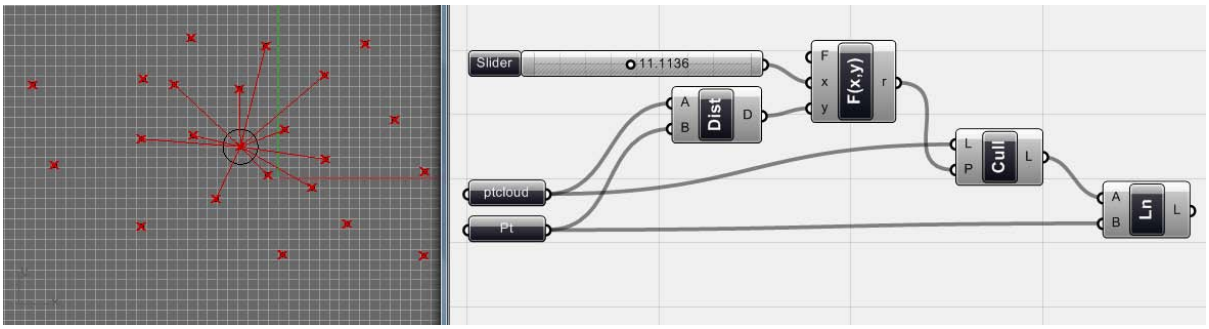


Fig.3.15. Selection of points from a point set by their distance to a reference point, using <Cull pattern> component.

Topography example

Having tested the first distance logic, I am thinking of selecting some points which are associated with contour lines on a topography model, based on their height.

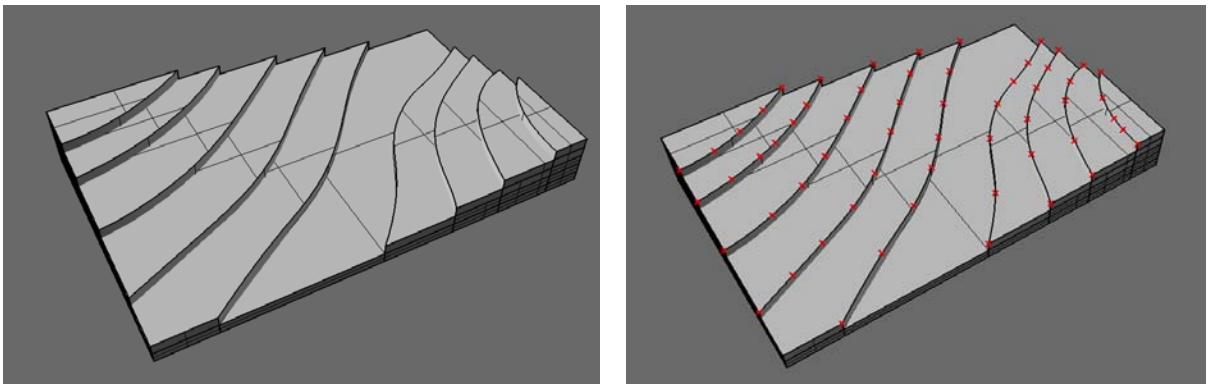


Fig.3.16. Topography with points associated with contour lines.

What I have is a point set which is defined by a <point> component (named topography). I need the height of the points and with the same logic as distance example, I can select my desired points. Here I used a <Decompose> component (Vector > Point > Decompose) to get the Z coordinates (heights) of these points. Point <Decompose> gives me the X,Y and Z coordinates of each point of its input. I compared these values with a given number (<number slider>) with a <Larger> component to produce a list of associative Boolean values. The <Cull pattern> component passes points who associated with the True values which means selected points are higher than the user defined height value.

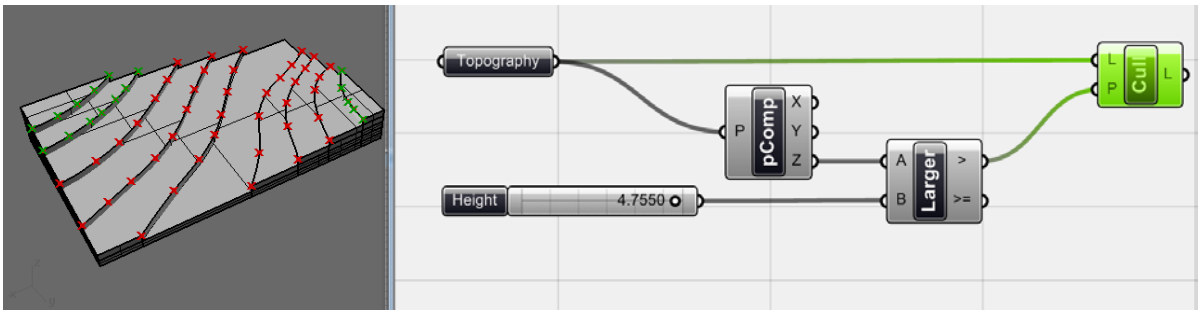


Fig.3.17. Selected points are higher than 4.7550 unit! (A user defined value). These points are now ready to plant your Pine trees!

3_7_ Data Lists

It is almost clear for you now that one of the basics of the algorithmic modelling is data lists. Data lists might be any sort of data like numbers, points, geometries and so on. Looking at the Logic tab, under the List panel there are multiple components that manipulate data lists. We can extract one item from a data list by its index number, we can extract part of a list by the lower and upper index numbers and so on. These list management items help us to gain a desired data list for our design purposes. Look at some examples:

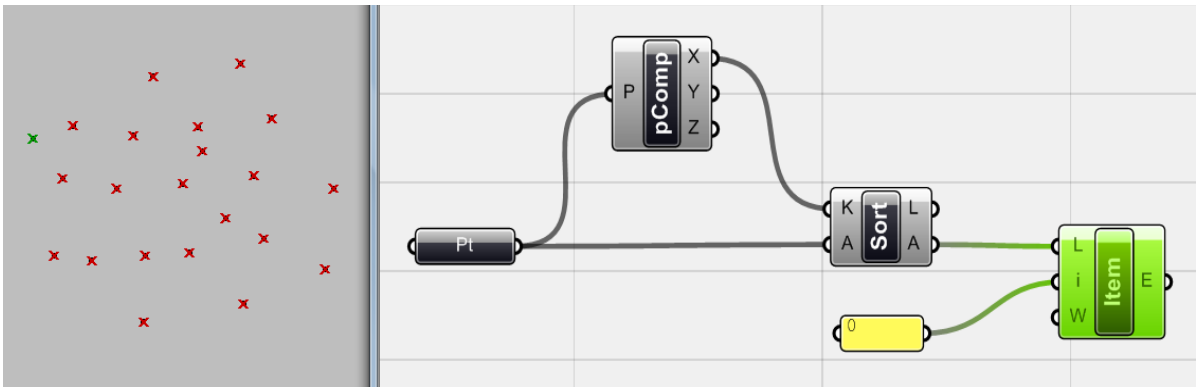


Fig.3.18. Here there is a list of points. I want to select the point with lowest X coordinate. As I said before, a <point decompose> component gives us the coordinates of points. What I need to do is to find the minimum X value of all X values of points. To achieve that I need to sort all these X coordinates to find the minimum. This is what <Sort List> will do for me. Basically <sort> component, sorts a list (or multiple lists) of data based on a numeric data list as sortable keys, so when it sorts numbers of key, the associated data will sort as well. So here I sorted all points with their X coordinates as Key data. What I need is to select the first item of this list. To do this, I need an <Item> component which extracts an item form a list by its index number. Since the first item (index 0) has the minimum X value, I extracted index 0 from the list and the output of the component would be the point with the minimum X value in the point set.

Lets go for more examples:

Triangles

Let's develop our experiments with data management. Imagine we have a network of points and we want to draw lines to make triangles with a pattern like figure 3.19. This concept is useful in mesh generation, panelization and relevant issues but for this time it is important to be able to generate this basic concept.

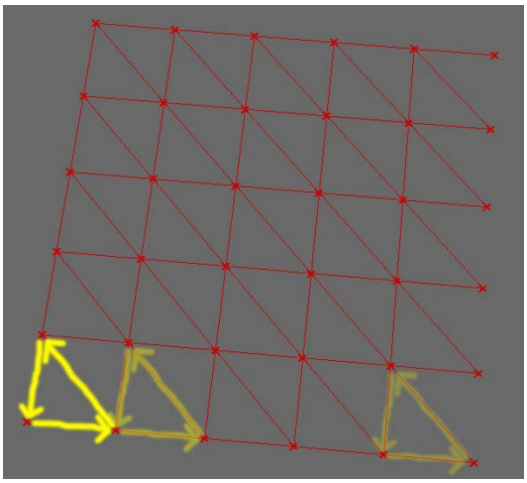


Fig.3.19. Generating triangles by a network of points.

The first step is to create a grid of points by <series> and <pt> components. The next step is to find the proper points to draw lines in between. Each time we need a line starts from a point and ends at the next point on the same row but next column, then another line goes from there, to the back column but at the next row and final line goes back from there to the start point. To do this, it seems better to make three different lists of points, one for all 'first points', one for all 'second points' and another for all 'third points' and then draw lines between them.

I can use the original points as the list for all 'start pints'. The first, 'second point' is the second point in the point set and then the list goes on, one by one. So to select the 'second points' I just shifted the original list by <Shift list> component (Logic > List > Shift list) by shift offset=1 which means the second item of the list (index 1) becomes the first item (index 0) and the rest of the list would be the same. This new list is the list of 'Second points'.

'Third points' of triangles are in the same column as the 'first points, but in next row. In terms of index numbers, if the grid has N columns, the first point in the second row has the index = index of the first point (0) + N

In a grid with 6 columns, the index of the first point of the second row is 6. So here I shifted the original list of points again by shift offset = the number of columns, to get the first point of the next row (the shift offset comes from the <number slider> which is the number of columns) to find all third points.

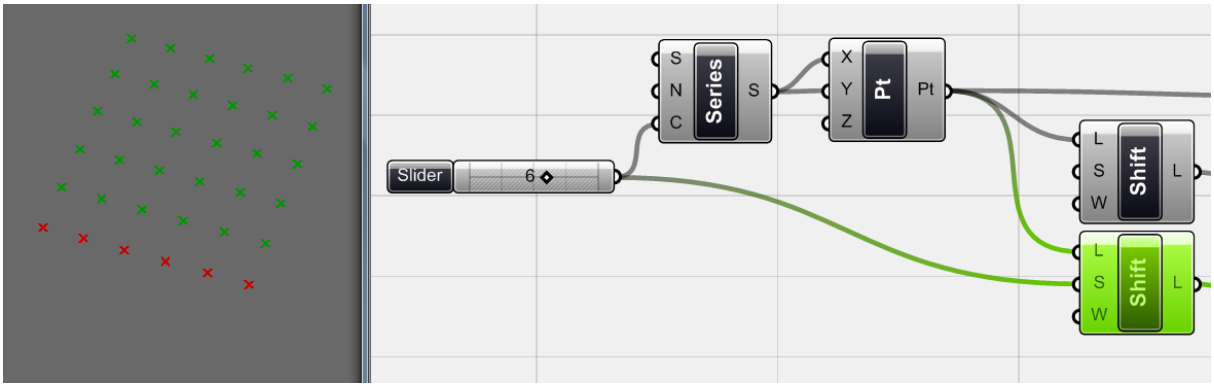


Fig.3.20. Selected item is the shifted points by the shift offset value equal to the number of columns which produces all 'third points' of triangles.

To complete the task I need to manipulate these lists a bit more so concentrate again:

1. First of all, in the list of 'First points', points in the last column never could be first points of triangles, so I need to omit them from the list of 'First points'.

2. Points on the first column also, never could be 'second points', so I need to omit them from the list of 'second points'.

3. The same for 'third points', where points in the last column never could be third point as well.

If you combine all these three parts and imagine and draw them you realize that in all three lists, points in the last column could not be used.

So basically I need to omit last column from each list of data. That's why I attached all points' lists, each to one <Cull Nth> component. <Cull Nth> omits every N number of a data list and N is cull frequency (Fig 3.20). In this case all data lists culled by the number of columns. That's because if for example we have 6 columns, it omits every 6th item of the list, means the item in the last column. And the result is a new list with omitted last column.

So I just connected the <number slider> which defines the number of columns to each <Cull Nth> component as frequency.

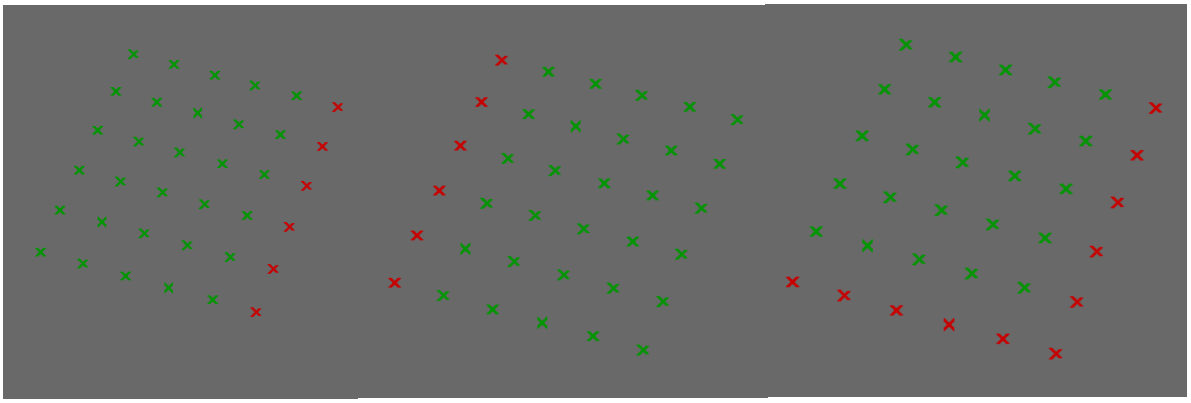


Fig.3.21. Using <Cull Nth> to omit the last column of the first, second and third point lists.

The last step is to feed three <line> components to connect first points to the second, then second points to the third and finally third points to the first again.

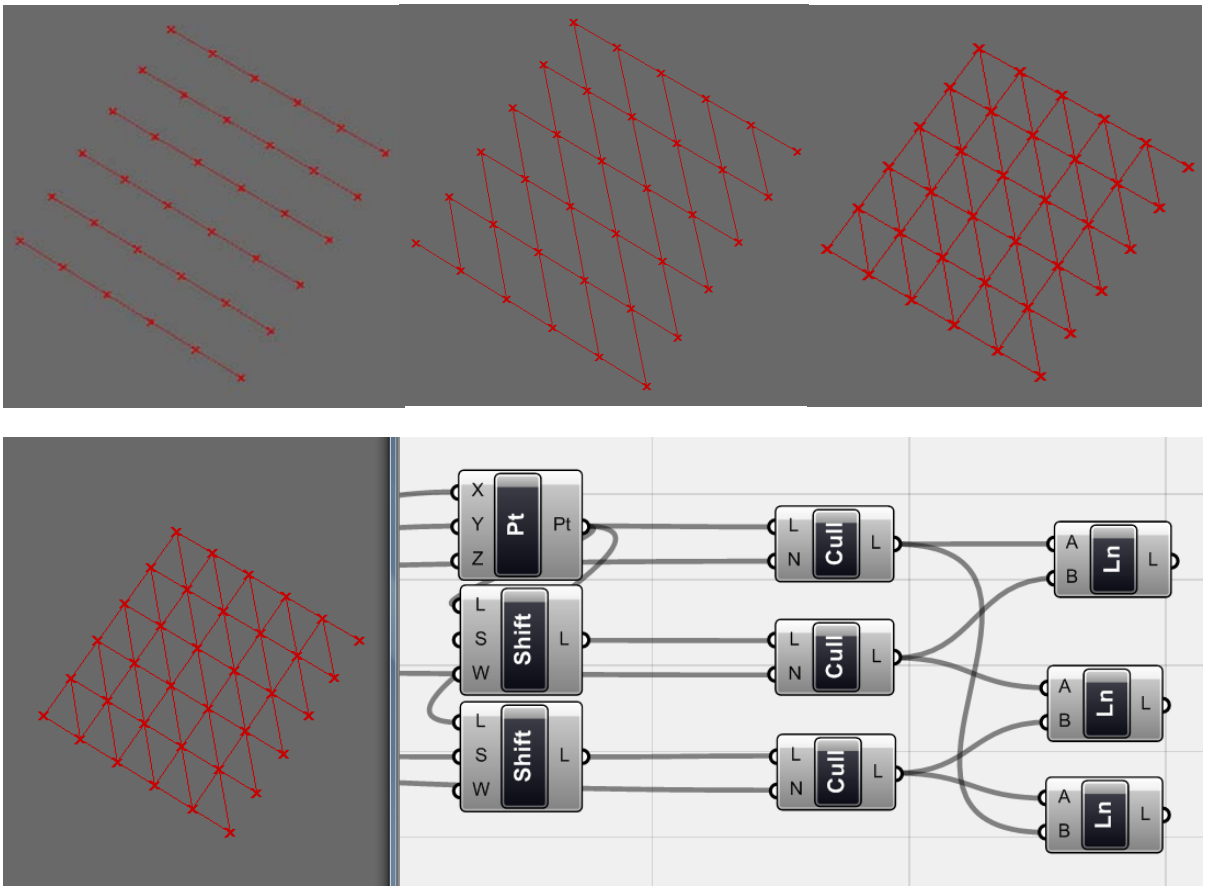


Fig.3.22. Making lines by connecting culled lists of points to the <Line> component. Don't forget that data matching for the <Pt> component set to Cross Reference and for <Line> components set to Longest List.

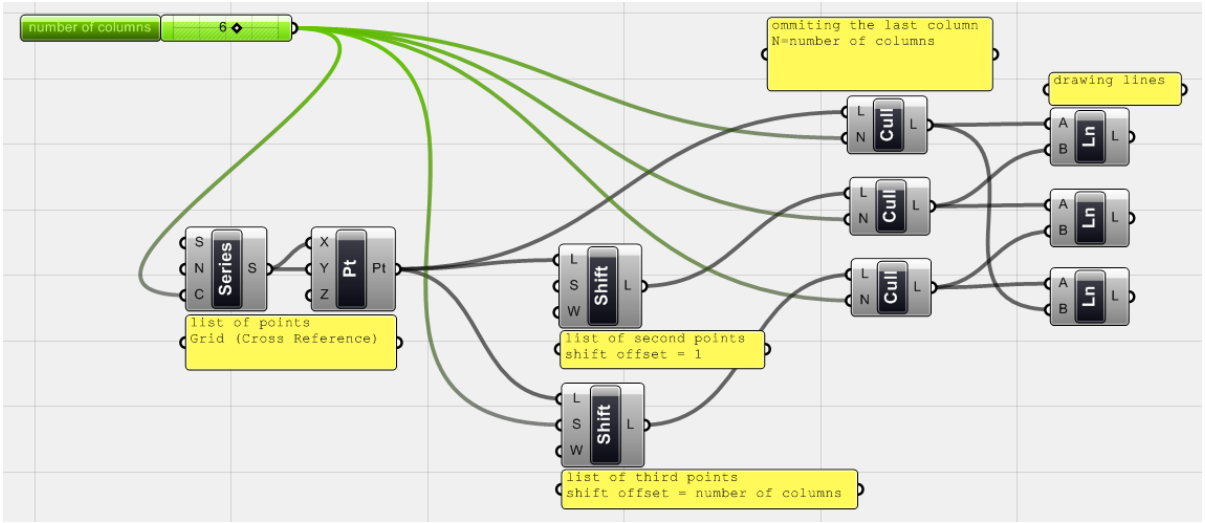


Fig.3.23. Now by changing the <number slider> you can have different grids of points which produce these triangles accordingly.

Although there are still some problems with our design and we know that we should not start any triangle from the points of the last row (and we should omit them from the list of 'first points'), but the concept is clear…… so let's go further. We will come back to this idea while talking about mesh geometries and then I will try to refine it. The main idea is to see how data should be generated and managed. Let's develop our understanding through more experiments.

3_8_On Planar Geometrical Patterns

Geometrical Patterns are among the possible design issues with Generative Algorithms and in Grasshopper. We have the potential to design a motif and then proliferate it as a pattern which could be used as a base of other design products. In case of designing patterns we should have a conceptual look at our design/model and extract the logic that produces the whole shape while being repeated. So by drawing the basic geometry we can copy it to produce the pattern as large as we need (Fig.3.22).

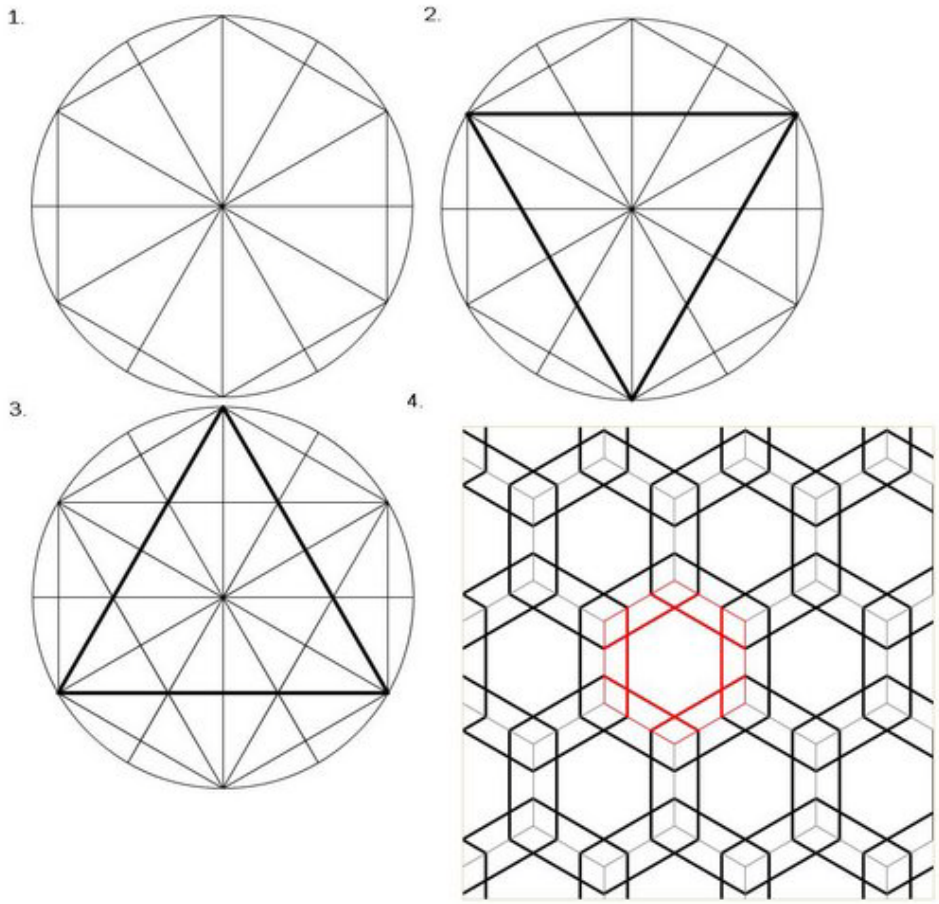


Fig.3.24. Extracting the concept of a pattern by simple geometries.

I still insist on working on this subject by data sets and simple mathematical functions instead of other useful components just to see how these simple operations and numerical data sets have the great potential to generate shapes, even classic geometries.

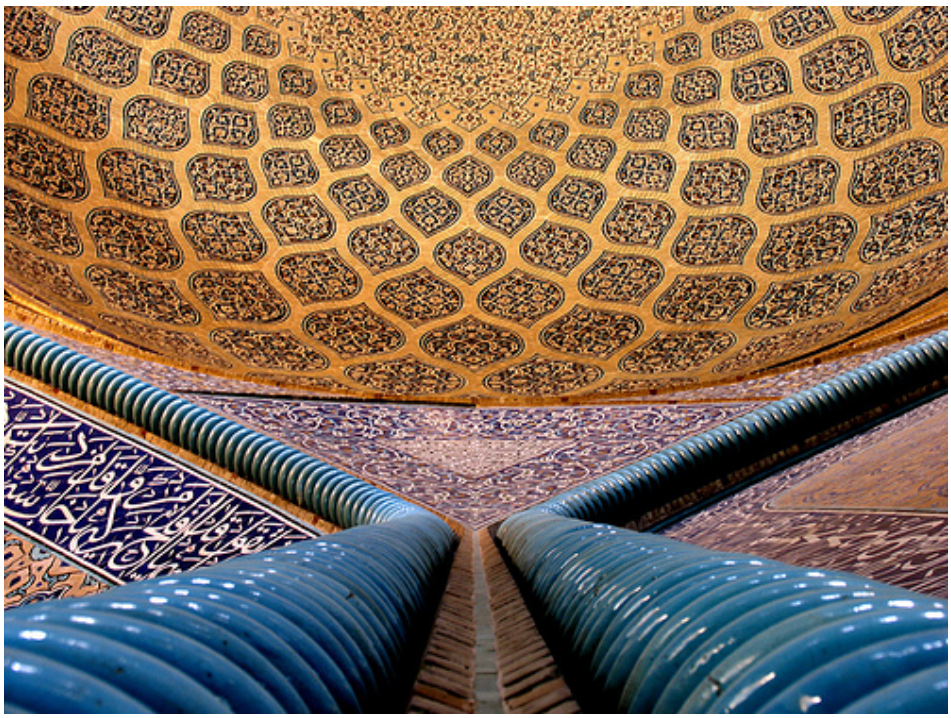


Fig.3.25. Complex geometries of Iran's Sheikh Lotfollah Mosque's tile work comprises of simple patterns which created by mathematical-geometrical calculations.

Simple Linear Pattern

Here I decided to design a pattern with some basic points and lines and my aim is to use simple concepts like Figure 3.26.

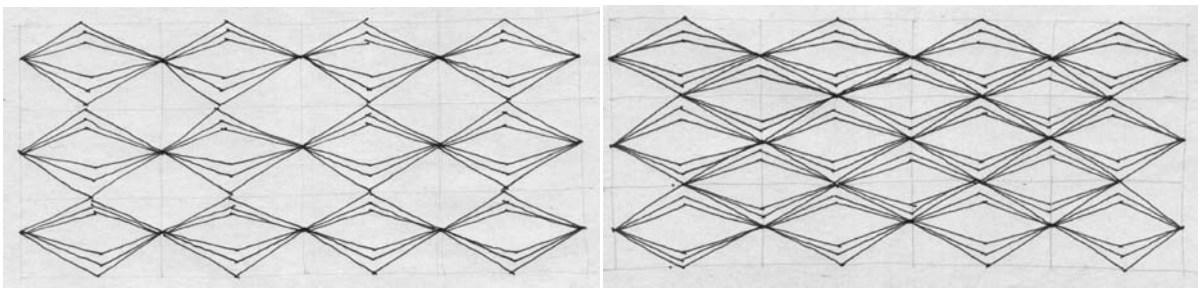


Fig.3.26. Basic concepts to generate patterns.

First of all I want to generate some basic points as base geometries and then draw my lines between them. I started my definition by a <series> which makes it possible to control the number of values (here number of points) and the step size (here distance between points). By this <series> I generated a set of points with only X entries (Y and Z =0).

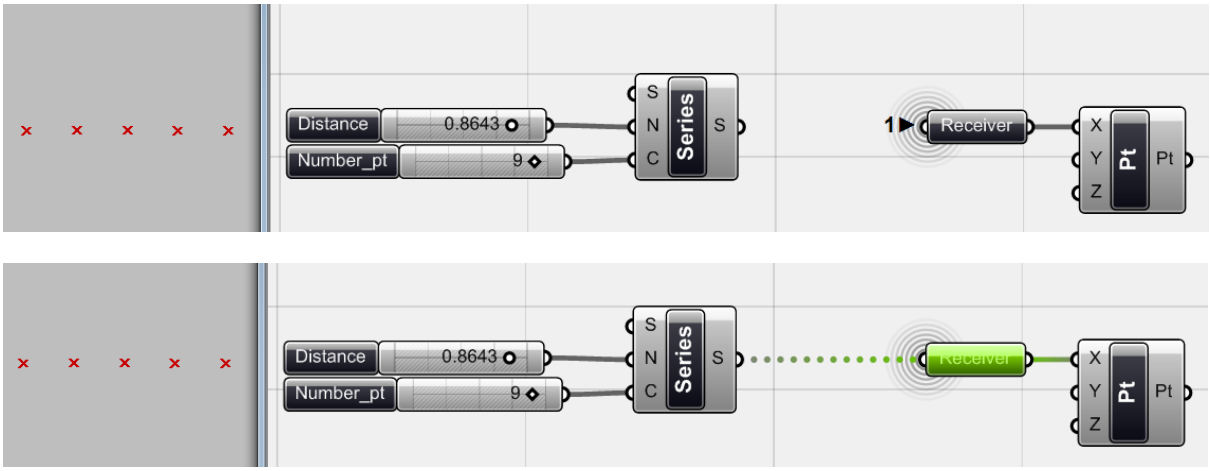


Fig.3.27. Here I generated my first set of points with <series> and <pt> components. The new trick is a <Receiver> component from Params > Special > Receiver. This component takes data from one component and passes it to another one while removes wires from the canvas. So in complex projects, when we want to use one source of data for many other components, it helps us to clear the canvas and don't be distracted by so many long wires. Here as you see in the second picture, the <Receiver> component, receives its data from <series>.

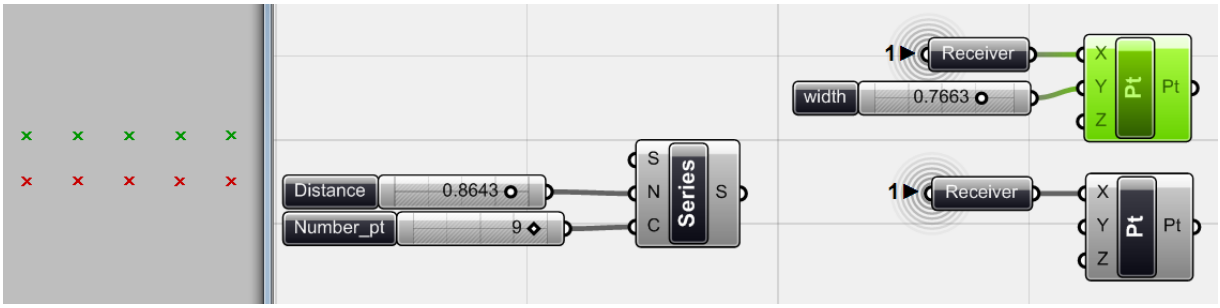


Fig.3.28. To create a "zig-zag" form of connections I need two rows of points as base geometries. I used another <Receiver> to get data from <series> and with another <pt> I generated the second row of points here with Y values come from a <number slider>.

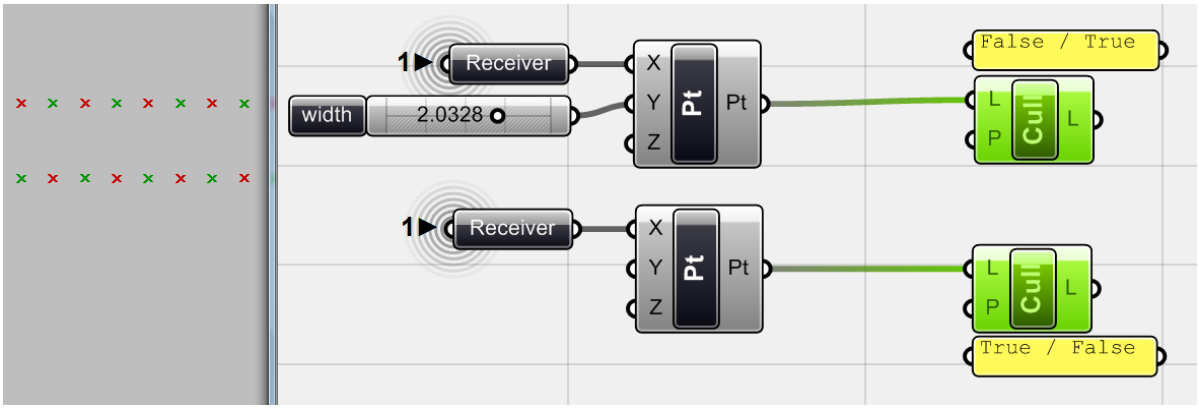


Fig.3.29. In the next step, I have to omit some points from each list to provide basic points for zig-zag pattern. Here I omit those points with <cull pattern>, one with True/False and another one with False/True Boolean pattern.

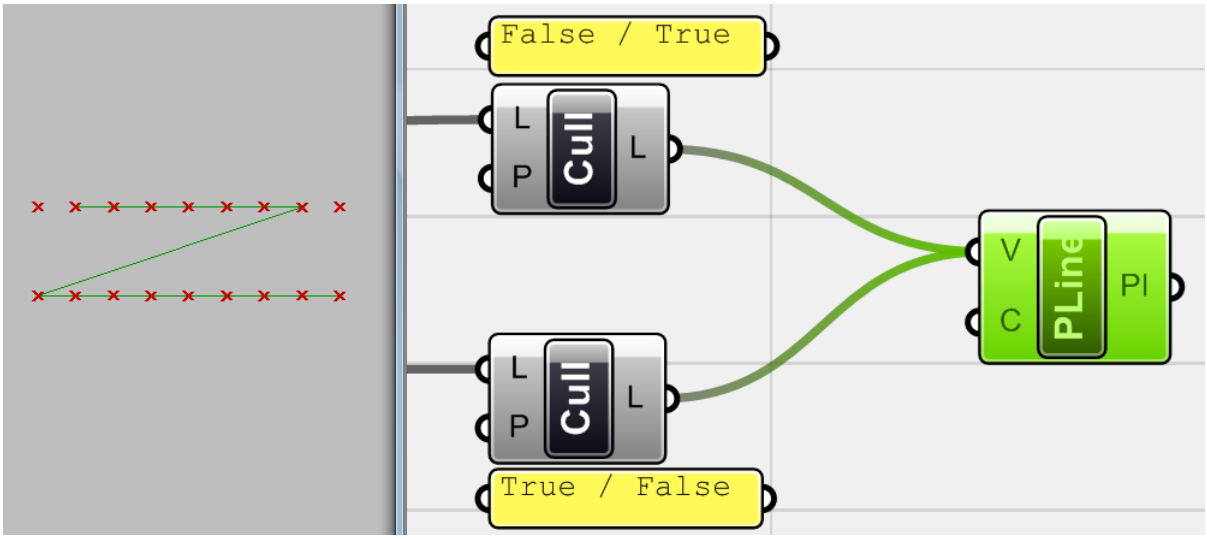


Fig 3.30. Now if you connect both <Cull> components to a <Poly line> component from Curve > Spline which draws lines by multiple vertices instead of two points, you see that a Z shape line would be the result. This is because points are not sorted and they need to be sorted in a list like this: 1st_pt of 1st row, 1st _pt of 2nd row, 2nd_pt of 1st row, 2nd_pt of 2nd row, …

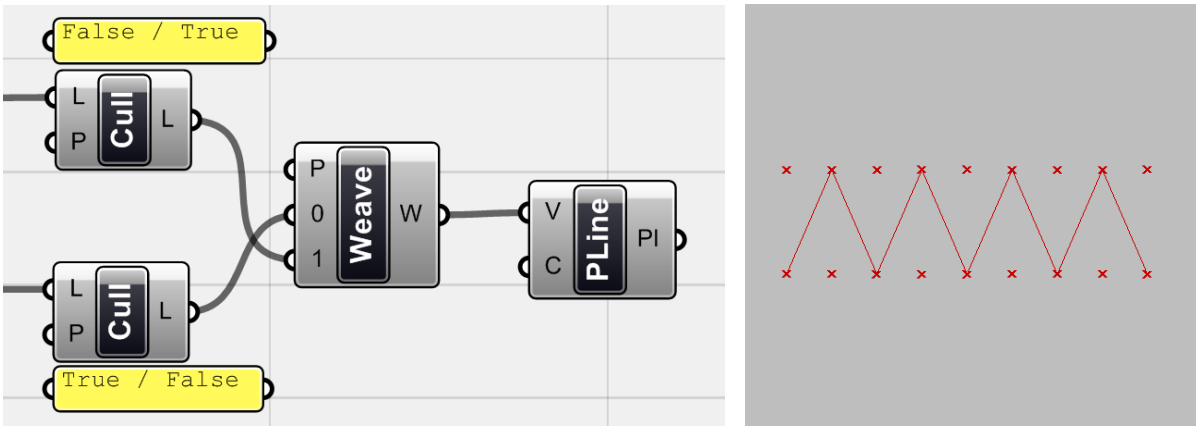


Fig.3.31. The component that sorts points in a way which I described is <Weave> (Logic > List). It takes data from multiple resources and sorts them based on a pattern which should be defined in its P input (like always read the component's help to see detailed information). The result is a list of sorted data and when you connect it to a <Pline> you see that the first zig-zag line is generated.

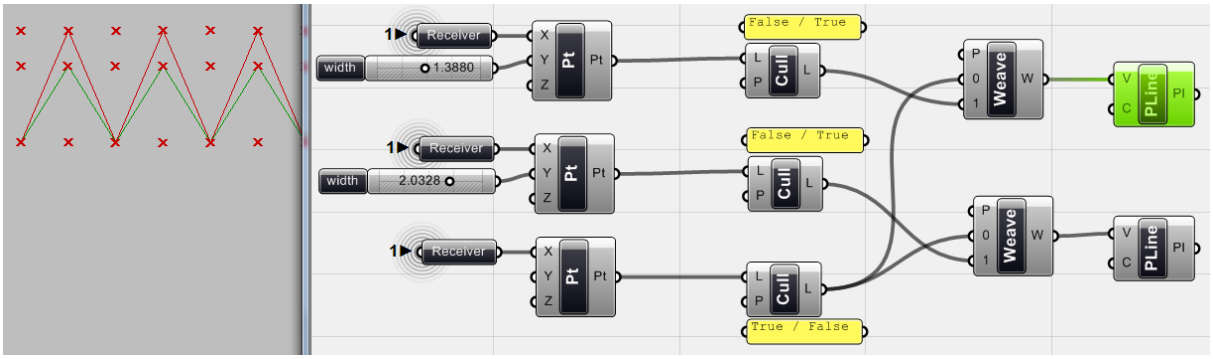


Fig.3.32. With the same concept, I generated the third row of points, and with another <weave> and <Pline> components, I drew second zig-zag line of the pattern.

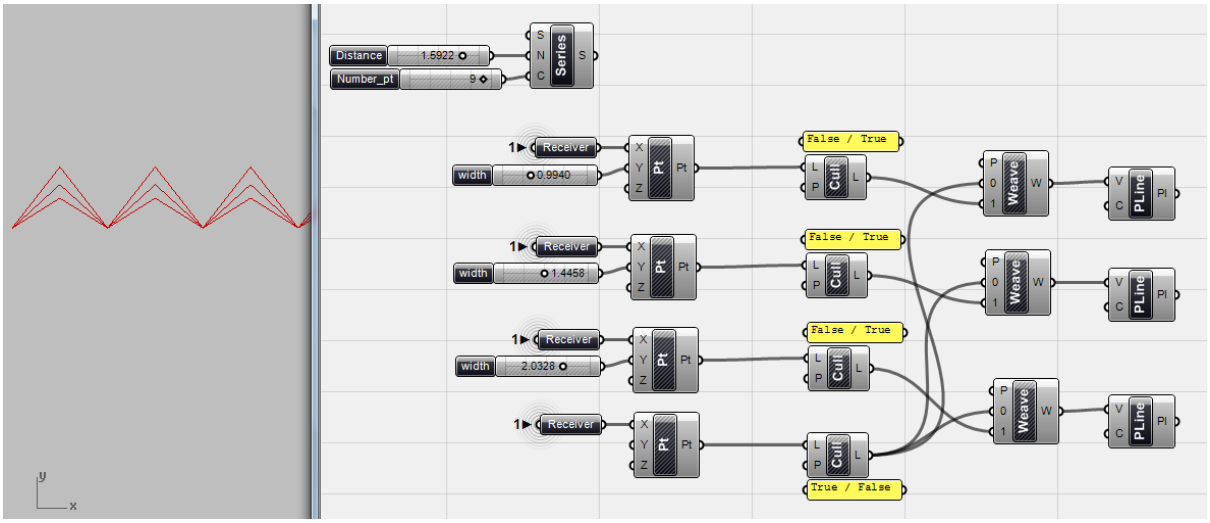


Fig.3.33. Although there are shorter ways to generate these lines, here again I used the same concept for points and pline of the third row. I unchecked the Preview option of <Pt>, <Cull> and <Weave> components (in their context menu) to hide all points and see Plines alone.

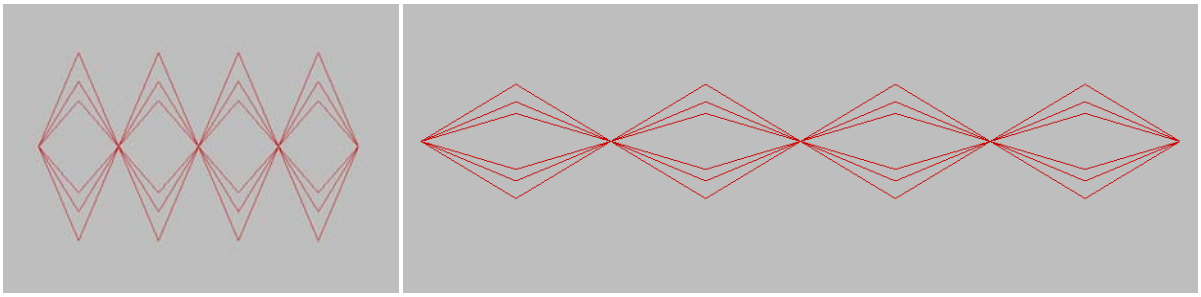


Fig.3.34. If you copy all process again and in this time convert Y values of <pt> components to negative (using same <number slider>s with a function of f(x) = -x), you would have a mirrored set of Plines. Now manipulating distances, you could have patterns in different shapes and scales.

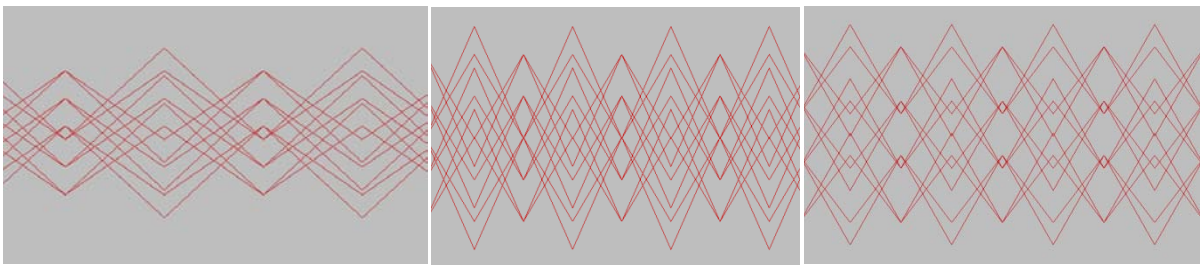


Fig.3.35. you could change the way you generate your base points or cull your lists of data. The result could be different patterns of intersecting lines which is simple, but could be the generative geometry to produce complex models.

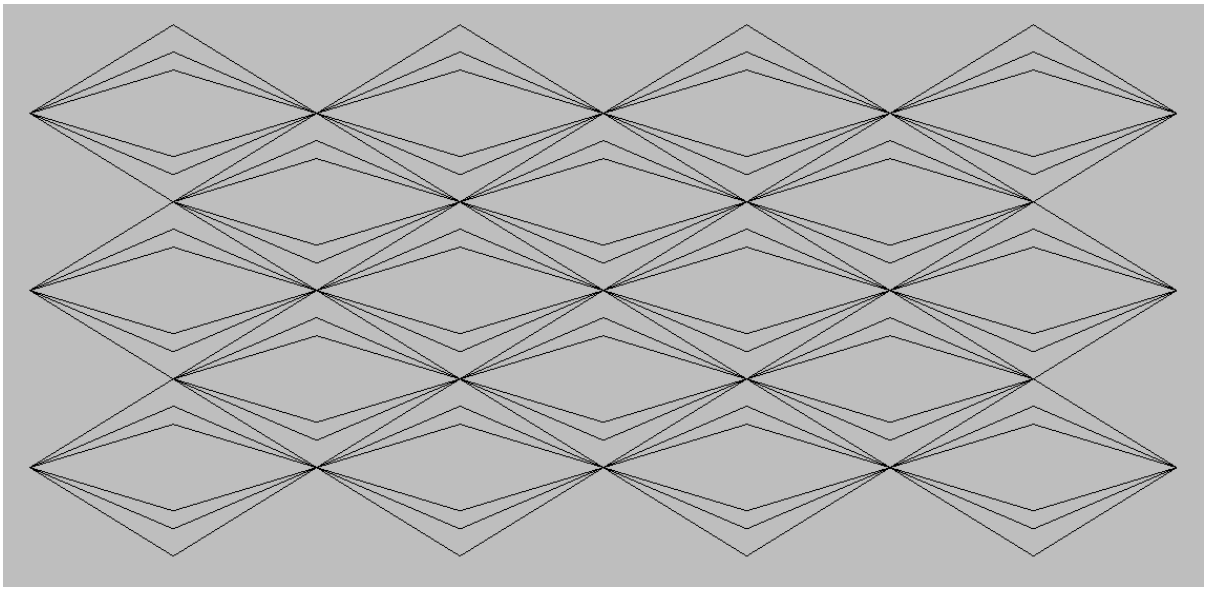


Fig.3.36. This is the first result of the design. The motif is repeated simply and the result could be used in any desired way which depends on your purpose.

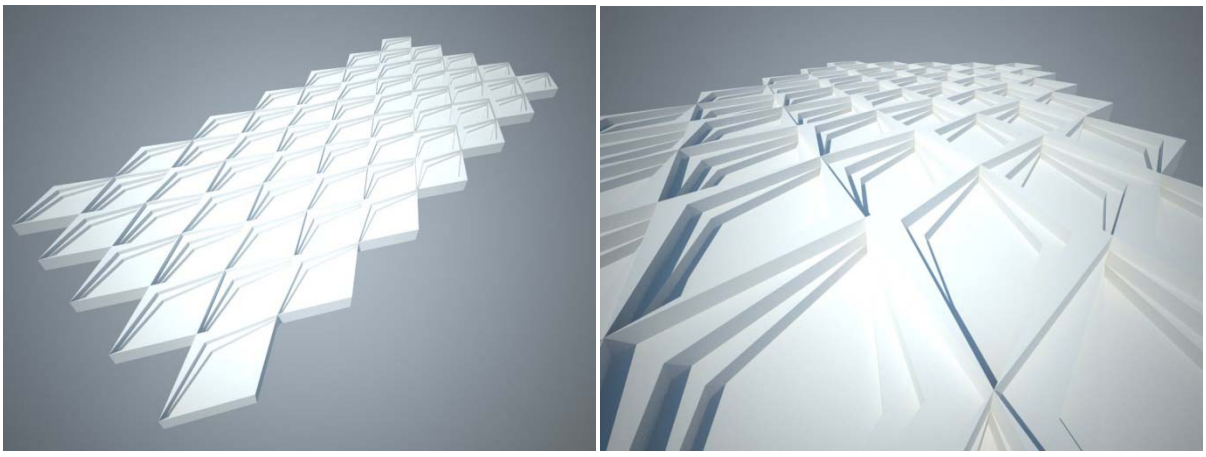


Fig.3.37. And this is just one of the examples among the hundreds of possibilities to use these basic patterns to develop a design. Later on you have the potential to differentiate the basic pattern and get manipulated design outcomes.

Circular patterns

There are endless possibilities to create motives and patterns in this associative modelling method. Figure 3.38 shows another motif which is drawn based on circular geometry rather than linear one. Since there are multiple curves which all have the same logic, I will describe one part of the algorithm and keep the rest for you.

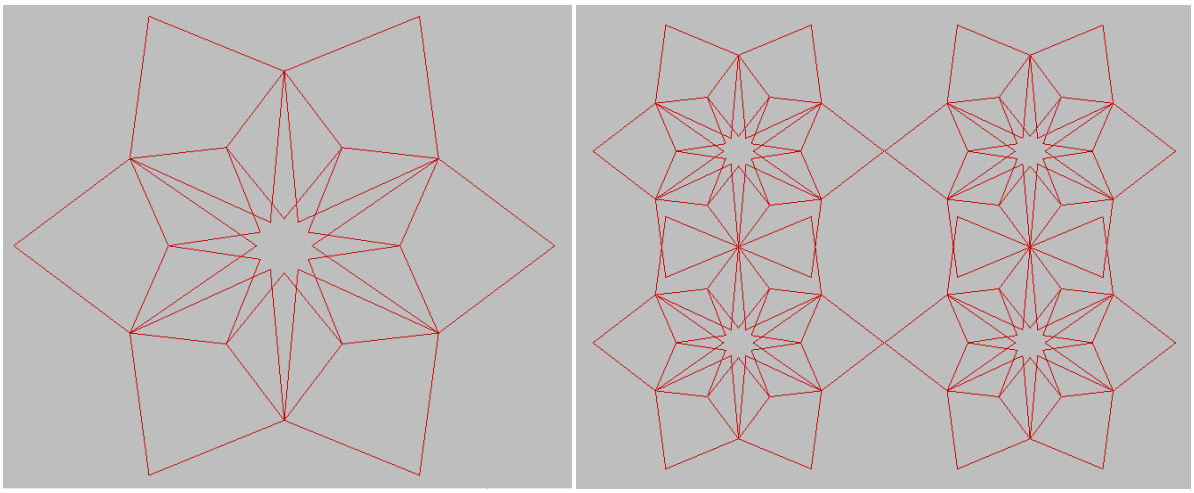


Fig.3.38. Circular geometrical patterns.

The start point of this pattern is a data set which produces a bunch of points along a circle, like the example we did before. This data set could be rescaled from the centre to provide more and more circles around the same centre. I will cull these sets of points with the same way as the last example. Then I will generate a repetitive 'zig-zag' pattern out of these rescaled-circular points to connect them to each other, make a star shape curve. Overlapping of these stars could make one part of the motif.

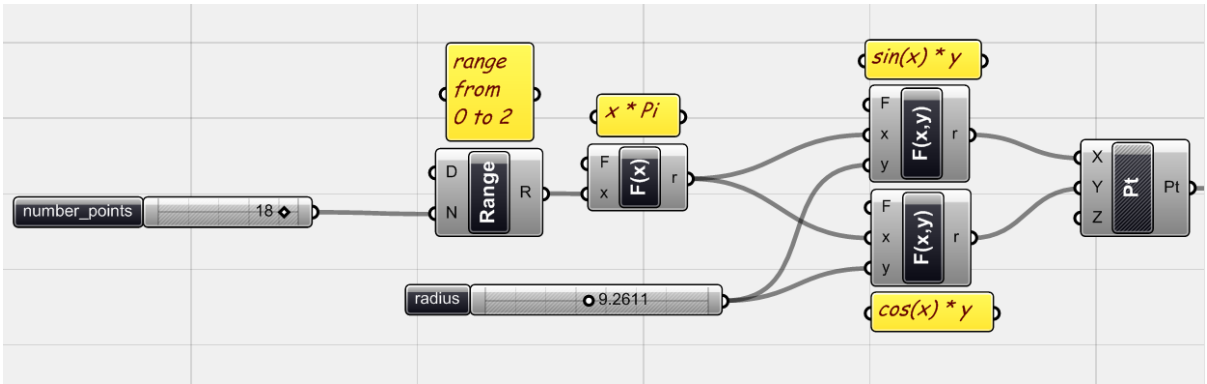


Fig.3.39. Providing a range of 0 to 2Pi and by using Sin/Cos functions, making the first set of points in a circular geometry. I used a function with two variables to multiply the result of Sin/Cos by another <number slider> to change the radius of the circle.

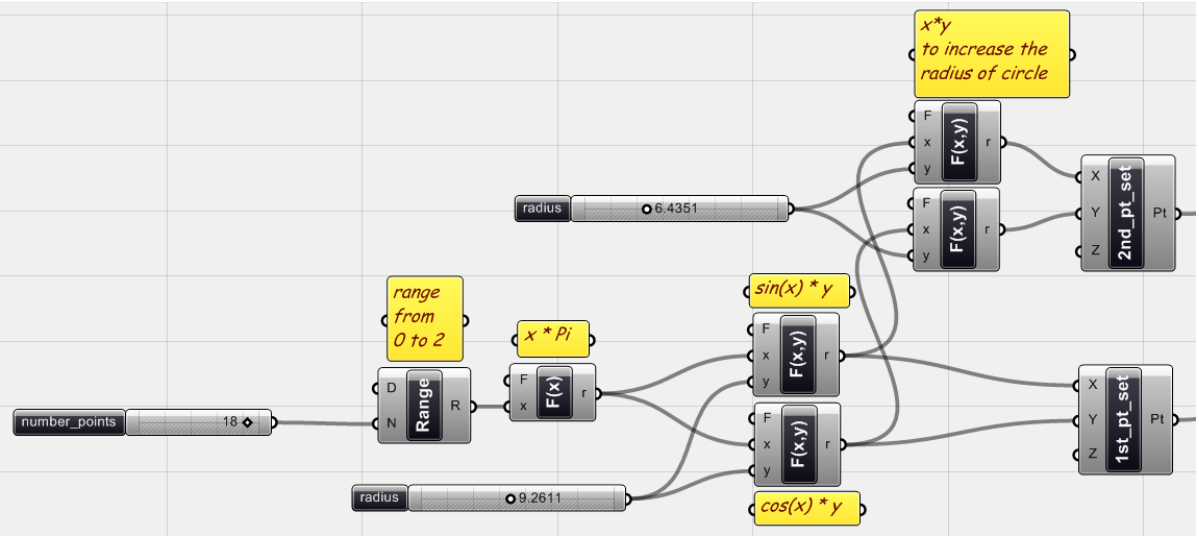


Fig.3.40. Increasing the result of Sin/Cos functions while multiplying by a <number slider>, making the second set of points with bigger radius. As you see the result of this section is two point sets. I renamed <pt> components.

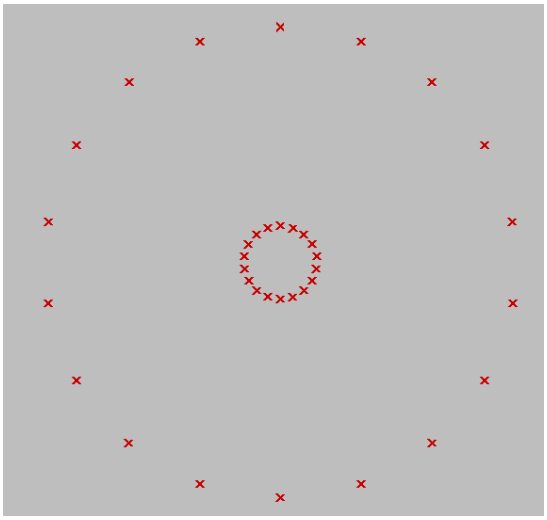


Fig.3.41. First and second circles of points.

In order to cull points, we can simply use the <Cull pattern> for the points and use True/False like the last example. But how we can sort the list of points after all? It is possible again to use <weave> component. But here I want to use another concept of sorting which I think would be useful later. I want to sort points based on their index number in the set.

First of all I need to generate index numbers. Because I produced points by a <range> component with real numbers, here I need a <series> component to provide integers as indices of the points in the list. The N parameter of the <range> defines the number of steps or divisions, so the <range> component produces N+1 numbers. That's why I need a <series> with N+1 values to be the index of the points.

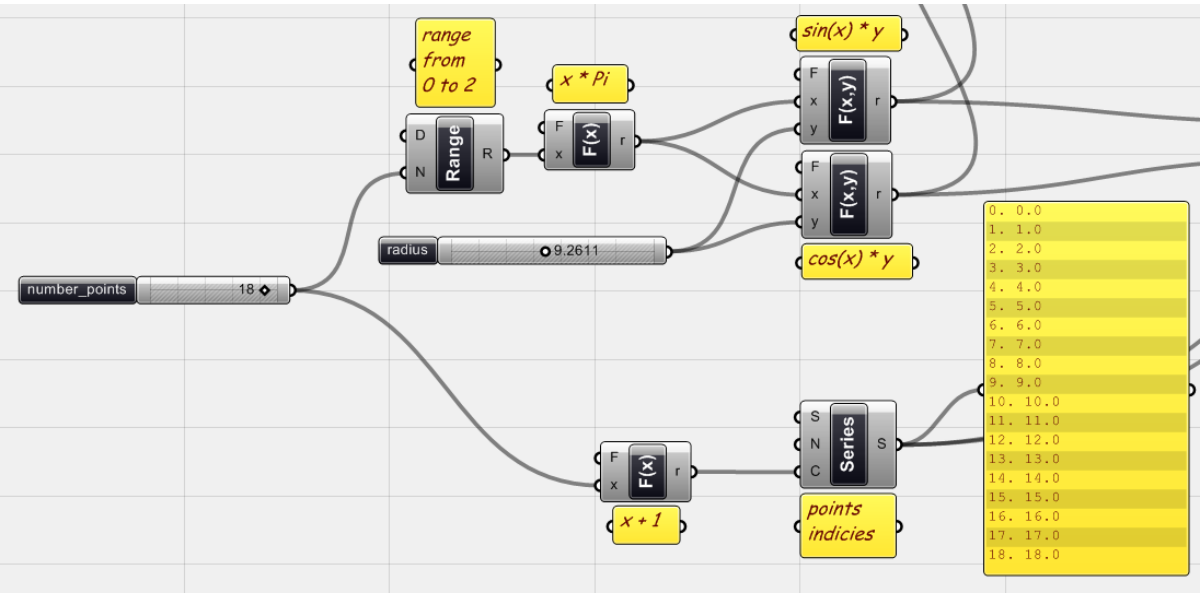


Fig.3.42. Generating index number of the points (a list of integers starts from 0)

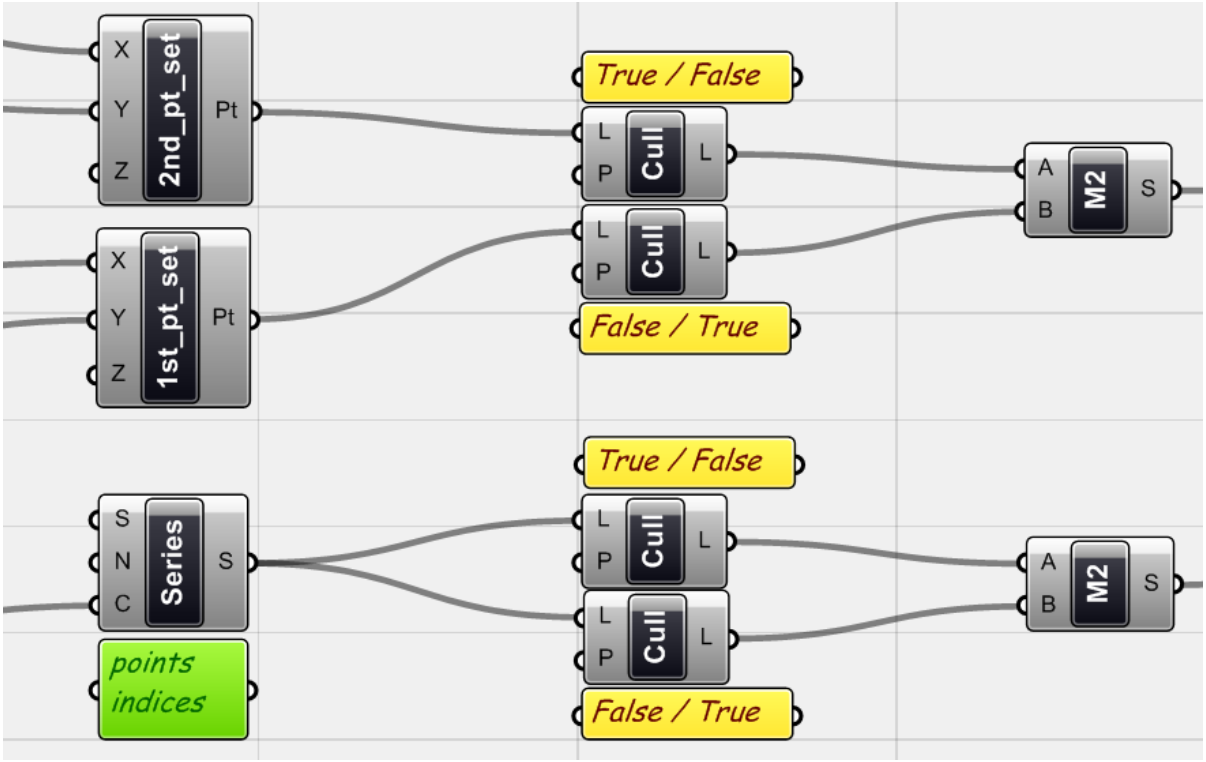


Fig.3.43. Now I need to cull points and indices both the same as previous example. Then I used <Merge> component (Logic > Tree) to generate one list of data from both <cull> lists. I did it for both points and indices. Although the result of the merging for <series> would be again numbers of the whole data set, the order of them is not the same and would be similar to the points. Now by sorting the indices as sortable keys we can sort the associated points as well.

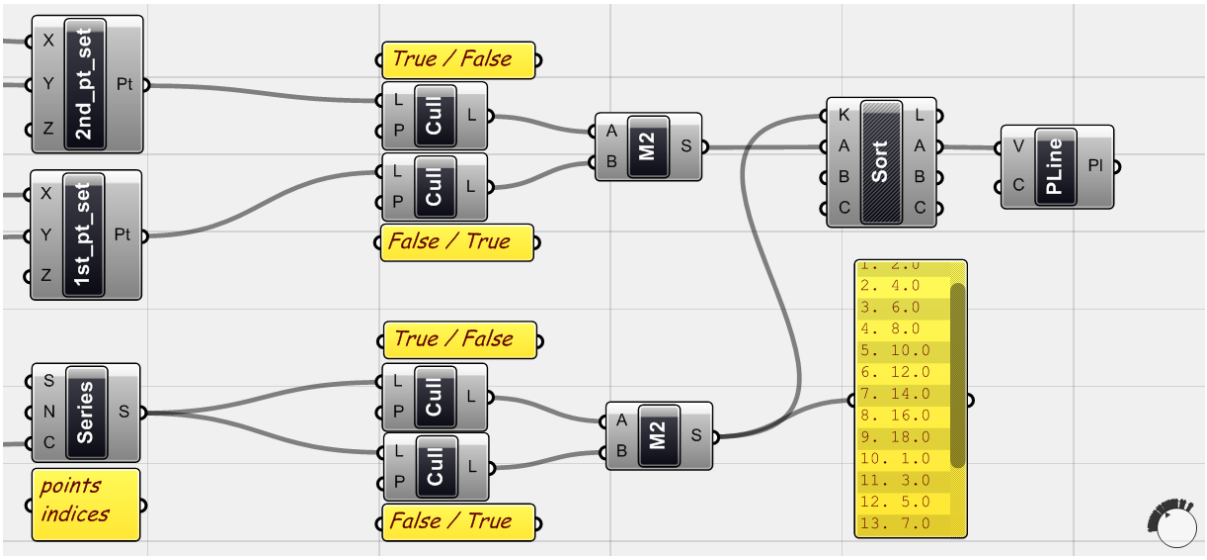


Fig.3.44. Points are sorted with a <sort> component while the sortable key is their indices. A Poly line is drawn by sorted points.

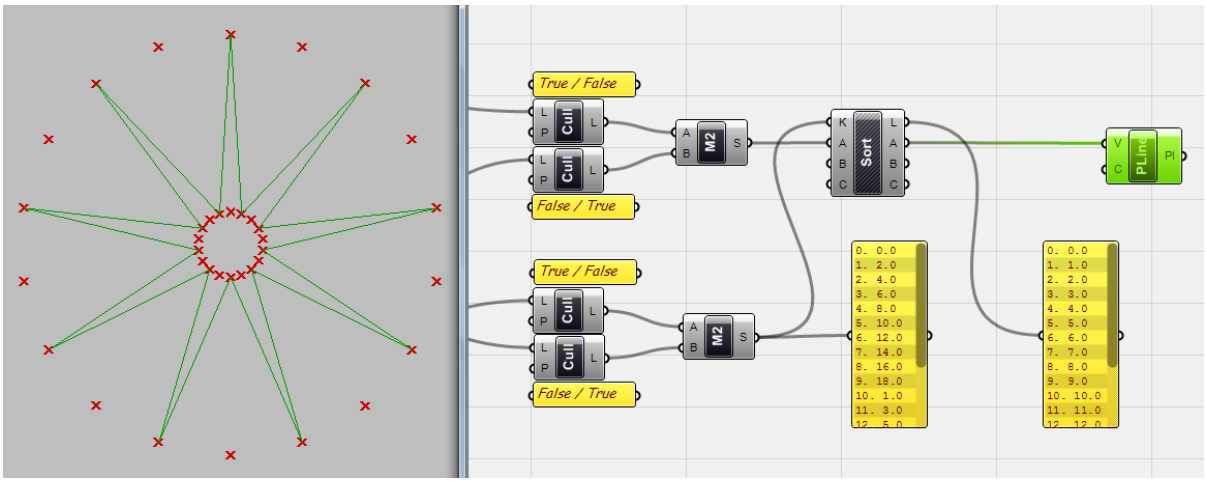


Fig.3.45. Indecies before and after sorting, and associated-sorted points which generated a star-shape poly line.

The same logic could be used to create more complex geometries by simply generating other point sets, culling them and connecting them together to produce desired patterns finally. The trick is to choose the best group of points and the way you connect them to other sets.

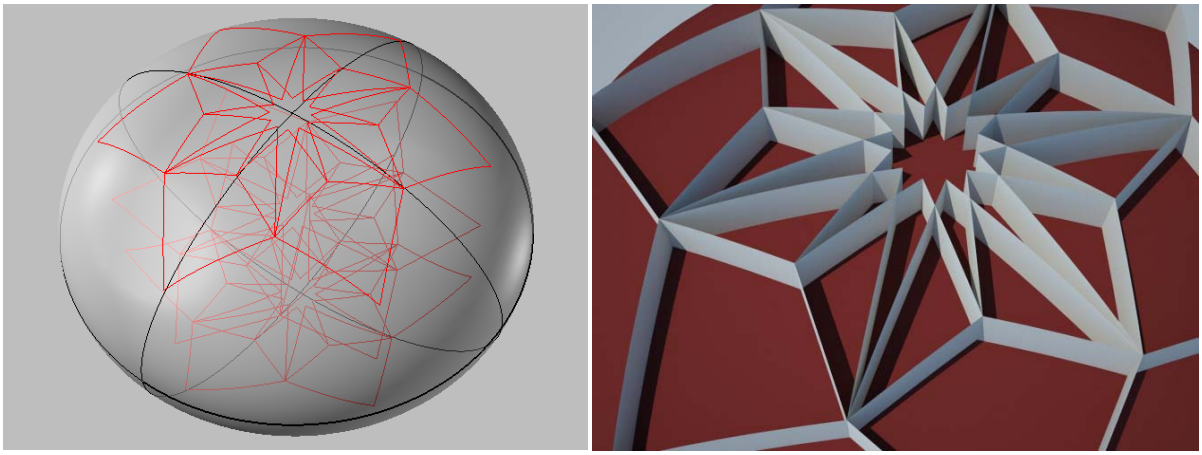


Fig.3.46. You can think about other possibilities of the patterns and linear geometries and their applications like projecting them to other geometries.

Although I insisted to generate all previous models by data sets and simple mathematical functions, we will see other simple components that make it possible to decrease the whole process or change the way we need to provide data. We will discuss them together.

Fig.3.47. Final model.

Chapter_4_Transformations

Chapter_4_Transformations

Transformations are essential operations in modelling and generating geometries. They enable us to get variations from the initial simple objects. Transformations help us to re-scale and orientate our objects, move, copy, mirror them, or may result in accumulation of objects. There are different types of transformations but to classify them, we can divide them to main branches, and the first division is linear and spatial transformations. Linear transformation performs on 2D space while spatial transformation deals with the 3D space and all possible object positioning.

In other sense we can classify transformations by status of the initial object; transformations like translation, rotation, and reflection keep the original shape but scale and shear change the original state of the object. There are also non-linear transformations. In addition to translation, rotation and reflection we have different types of shear and non-uniform scale transformations in 3D space, also spiral and helical transformations and projections which make more variations in 3D space.

In order to transform objects, conceptually we need to move and orientate objects (or part of objects like vertices or cage corners) in the space and to do this, we need to use vectors and planes as basics of these mathematical/geometrical operations. We are not going to discuss basics of geometry and their mathematical logic here, but first let's have a look at vectors and planes because we need them to work with.

Fig.4.1. Transformation is a great potential to generate complex forms from individuals. Nature has some great examples of transformation in its creatures.

4_1_Vectors and Planes

Vector is a mathematical/geometrical object that has magnitude (or length) and direction and sense. It starts from a point, goes towards another point with certain length and specific direction. Vectors have wide usage in different fields of science and in geometry and transformations as well.

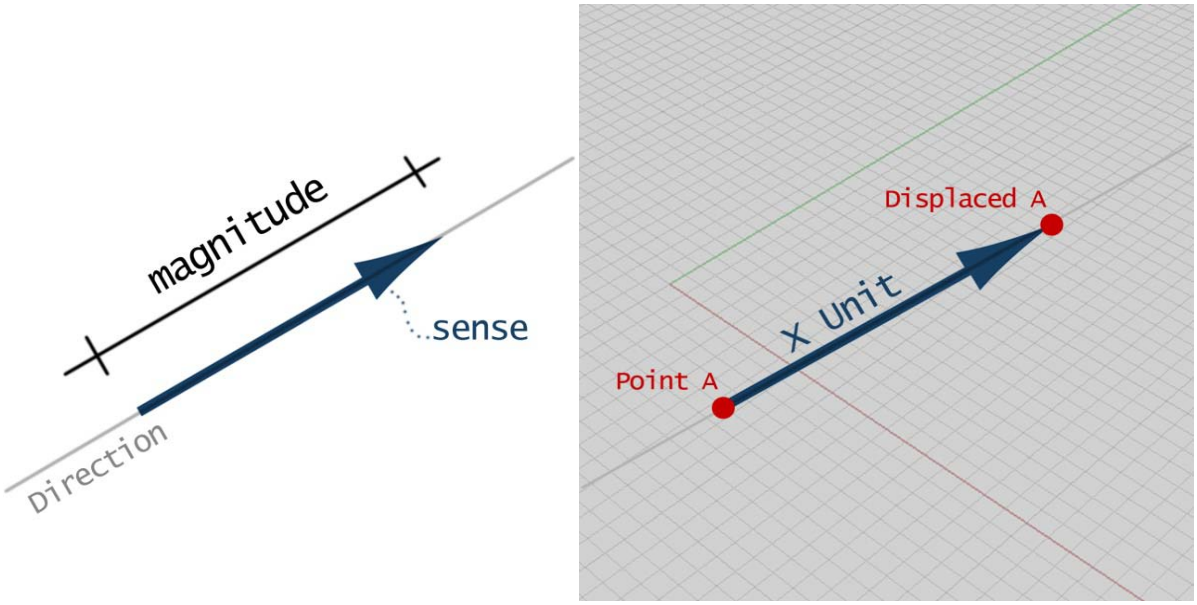


Fig.4.2. A: Basic elements of a Vector, B: point displacement with a vector.

Simply if we have a point and a vector, this vector can displace the point with the distance of vector's magnitude and towards its direction to create a new position for it. We use this simple concept to generate, move, scale and orientate geometries in our associative method.

Planes are another useful set of geometries that we can describe them as infinite flat surfaces which has an origin point. Construction planes in Rhino are these types of planes. We can use these planes to put our geometries on them and do some transformations based on their orientation and origin. For example in 3D space, we cannot orientate an abject on a vector! and we need two vectors to create a plane to be able to put geometry on it.

Vectors have direction and magnitude while planes have orientation and origin. So they are two different types of constructs that can help us to create, modify, transform and articulate our models in space.

Grasshopper has some of the basic vectors and planes as predefined components. These are including X, Y and Z unit vectors and XY, XZ, and YZ planes. There are couple of other components to produce and modify them which we will talk about them in our experiments. So let's jump into design experiments and start with some of the simple usage of vectors and go step by step forward.

4_2_On Curves and Linear Geometries

As we have experimented with points which are 0-Dimension geometries, now we can start to think about curves as 1-Dimensional objects. Like points, curves could be the base for construction of so many different objects. We can extrude a curve along another one and make a surface, we can connect different curves together and make surfaces and solids, we can distribute any object along a curve with specific intervals and so many other ways to use a curve as a base geometry to generate other objects.

Displacements

We generated many point grids in chapter 3 by <series> and <pt> components. But there is a component called <Grid rectangular> (Vector > Point > Grid rectangular) which produces a grid of points. We can control the number of points in X and Y direction and the distance between points (equal in both directions) in this component.

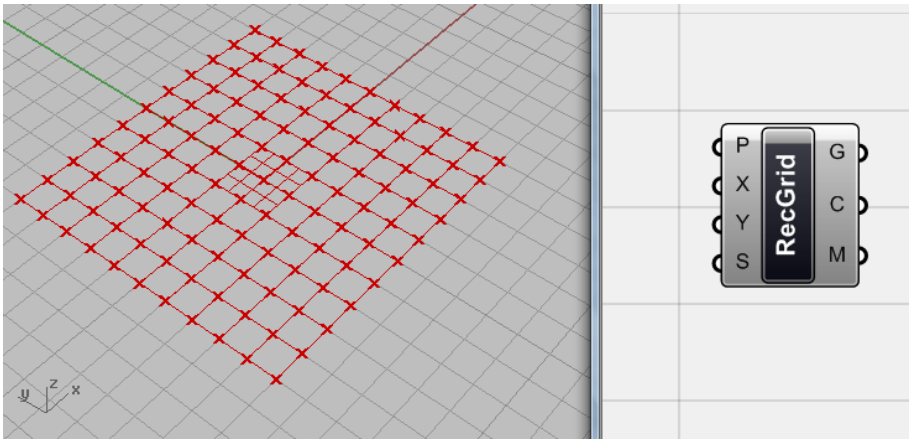


Fig.4.3. A simple <Grid Rectangular> component with its predefined values.

You can change the size of grid by a <number slider> as its distance input (S). You can also change the orientation of the points. To do this, you need a plane to stick your grid on it. Here, I introduced an <XY plane> component (Vector > Constants > XY plane) which is a predefined plane in the orientation of X and Y axis and I displaced it in Z direction by a <Z unit> component (Vector > Constants > Z unit) which is a vector along Z axis with the length (magnitude) of one. I can change the height of this displacement by the size of vector through a <number slider> that I connected to the input of the <Z unit> component; changing the position of the <XY plane> along the Z axis would change the position of the grid.

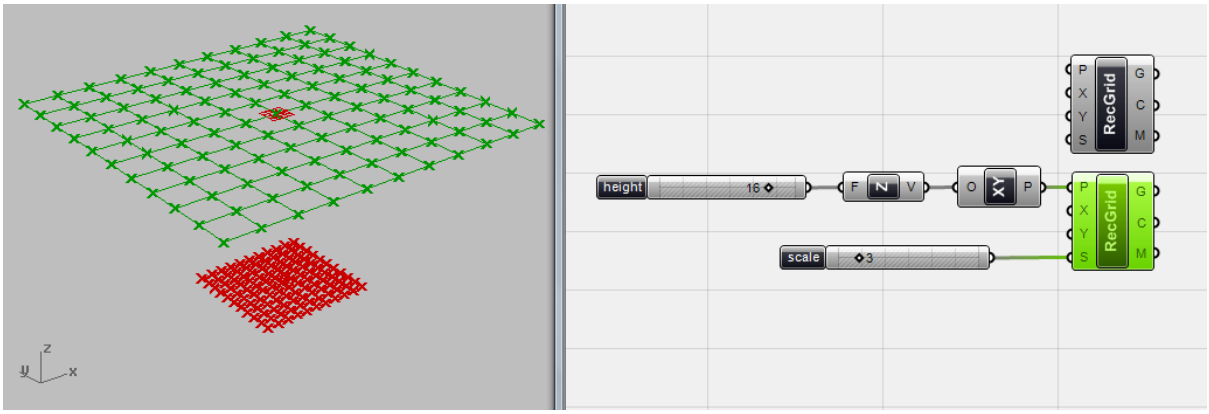


Fig.4.4. Manipulated Grid (selected in green) with one <number slider> for scale of the grid (distance between points) and another with a <z unit> and <XY plane> to change the Z coordinates of the grid's points.

Now if you look at the output of the <grid rectangular> you can see that we have access to the whole points as well as grid cells and cell centers. For this experiment, I am looking for a bunch of lines that start from the grid cells' centre points and spread out of it to the space, using these grids only. I can simply connect points from two <grid> component's M part to a <line> component, and generate bunch of lines. Then changing the size of second grid would result on the direction of all lines. Here the problem is the length of lines which in this case would be different from each other, but I want to draw lines with the same length, and I need another strategy and that's why I am going to use a <line SDL> component.

A <line SDL> component draws a line by Start point(S), Direction (D), and Length (L). So the length of lines is controllable. Exactly what I need; I have the start points (cell's midpoint), and length of my lines (whatever I like). What about the direction? I wanted to use the second grid-cell midpoints as the second point of lines so the direction of my lines is in the direction of the connecting lines of grid-cell mid points. To define these directions here I need some vectors instead of lines. That's why I am going to make a set of vectors by these two point sets to create directions for my <line SDL> component.

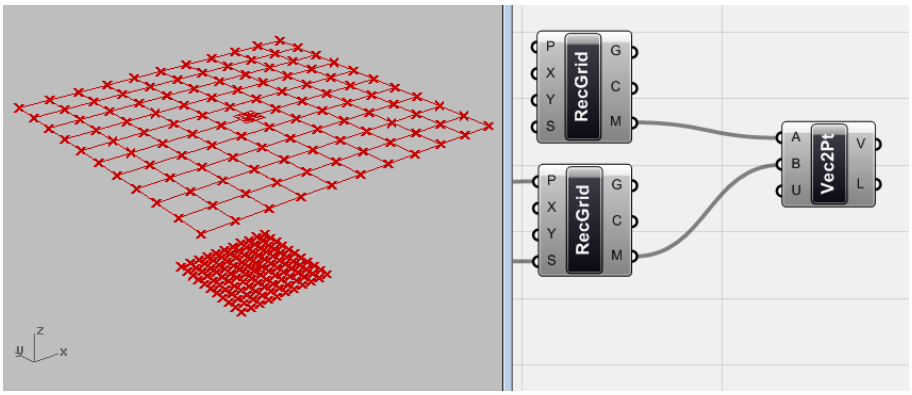


Fig.4.5. Creating vectors from the cells' midpoints of the first grid toward the cells' midpoints of the second grid by <vector 2pt> component (Vector > Vector > vector 2pt). This component creates vectors by start and end point of vectors.

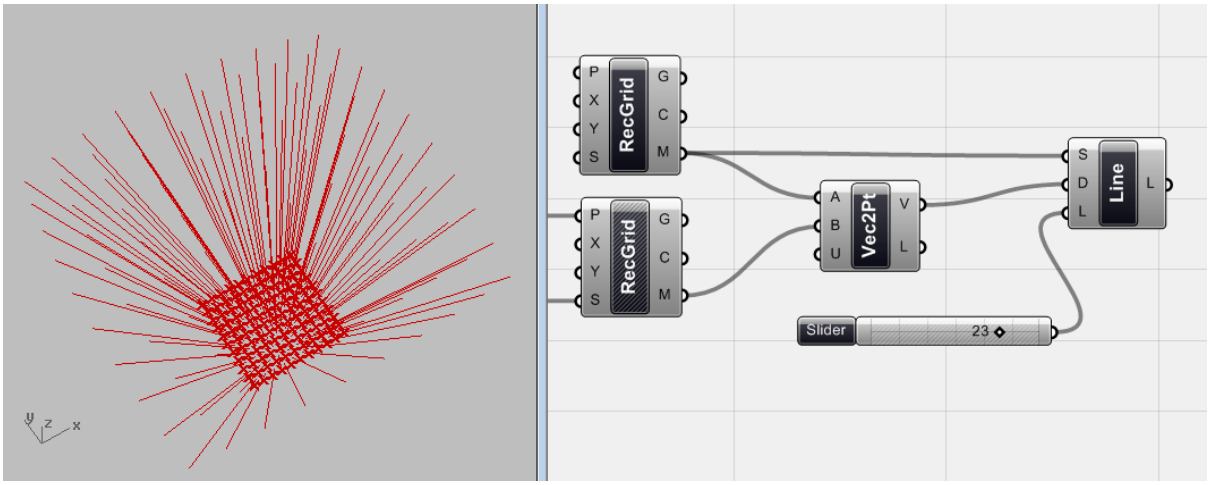


Fig.4.6. The <line SDL> component generates bunch of lines from the grid cell midpoints that spread out into space because of the bigger size of the second grid. I can change the length of lines by <number slider> and I can change their direction by changing the size of second grid.

For the next step, I want to add a polygon at the end of each line and extrude it towards the start point of the line to see the generative potentials of these curve components. To generate polygons I have to add some planes at the end point of my lines as base planes to be able to create polygons.

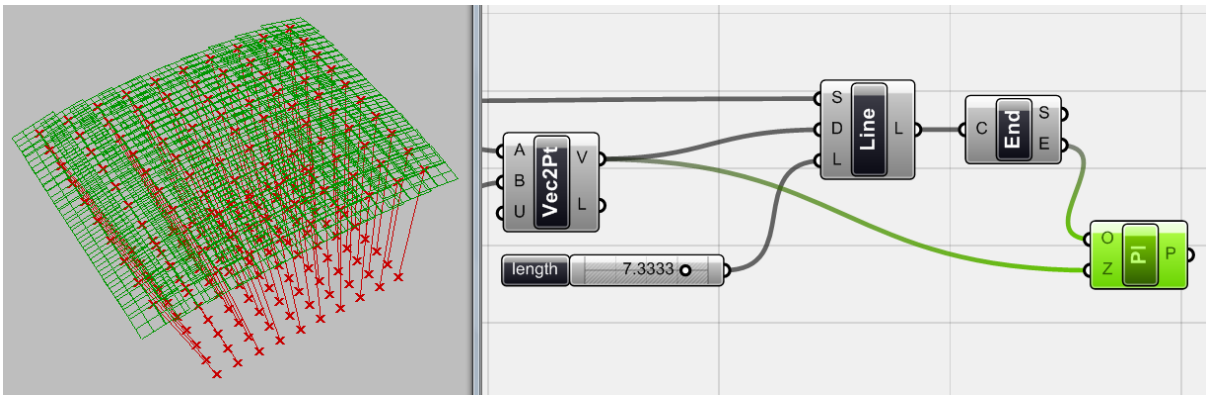


Fig.4.7. By using an <end points> component (Curve > Analysis) and using these 'end points' as 'origin points' for a set of planes I could generate my base planes. Here I used <Plane Normal> component (Vector> Plane) which produces a plane by an origin point (lines' end point) and a Z direction vector for the plane (a normal vector which is perpendicular to the plane). Here I used same vectors of the line direction as normal vectors for planes.

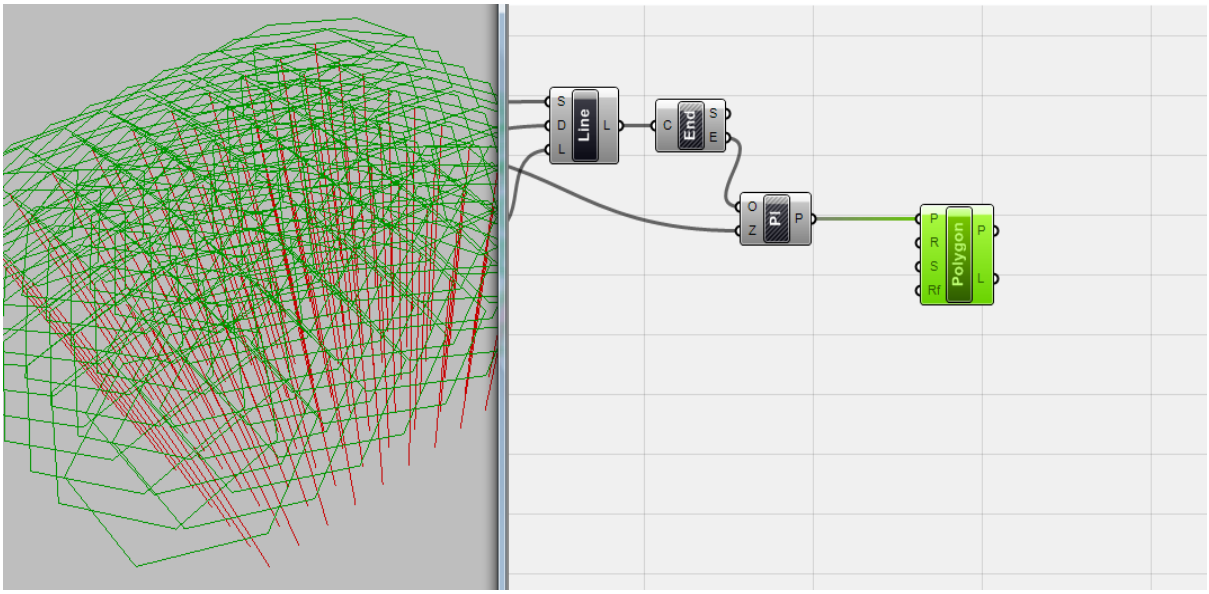


Fig.4.8. Adding a <Polygon> component and Using generated planes as base planes for polygons, we would have a set of polygons at the end of each line and perpendicular to it. As you can see, these polygons have same size but I want to apply a system of size differentiation to them to have a smooth shape change at the end.

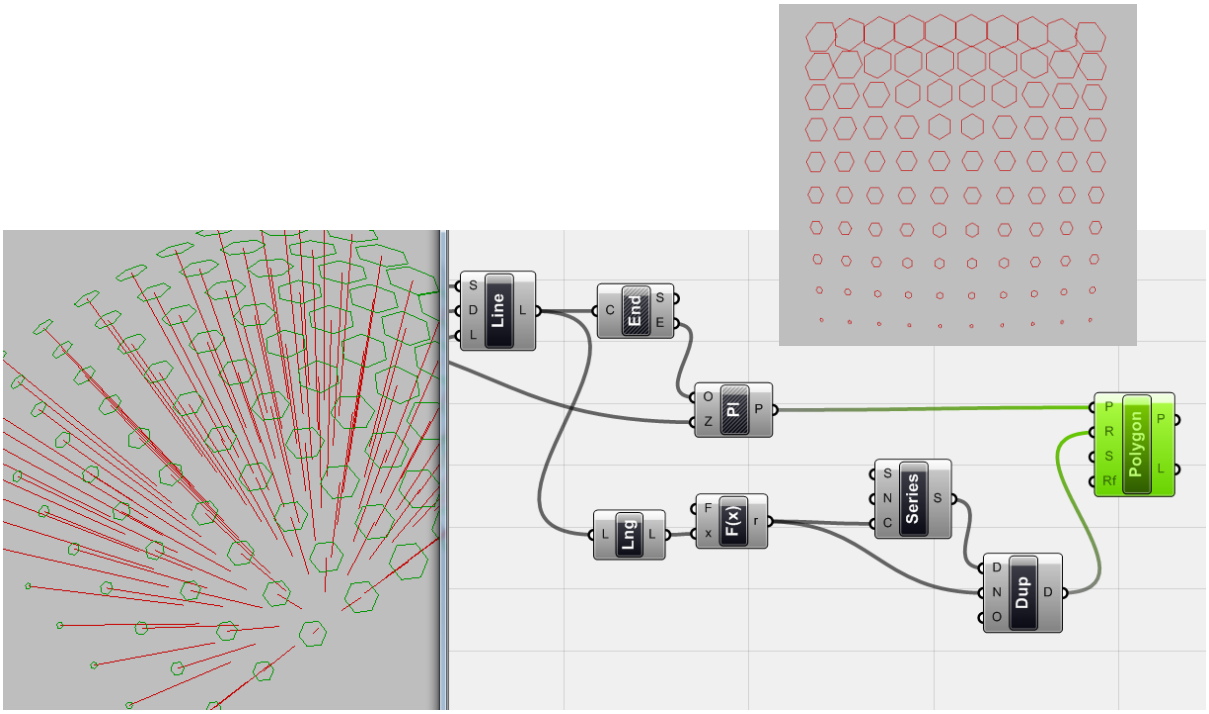


Fig.4.9. With a <List Length> component I get the number of my lines and the next <function> component which is the square root of the input (F(x)=Sqrt(x)), calculates the number of lines at each row. I used a <series> component with the start point and step size = 0.1 wile the number of values are coming from the number of rows. So I generated a list of gradually growing numbers equal to the number of polygons at each row. To be able to use these values for all polygons, I duplicated these data list with the amount of columns (here equal to the number of rows) and attached it to the Radios input of polygons. As you can see in the model, at each row, the size of polygons gradually changed and this pattern repeated up to the last one.

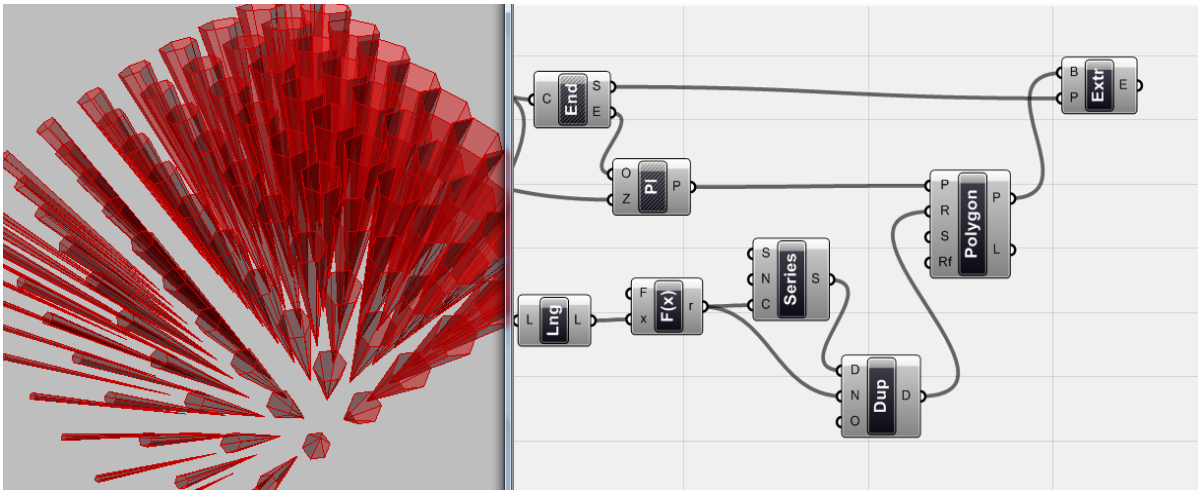
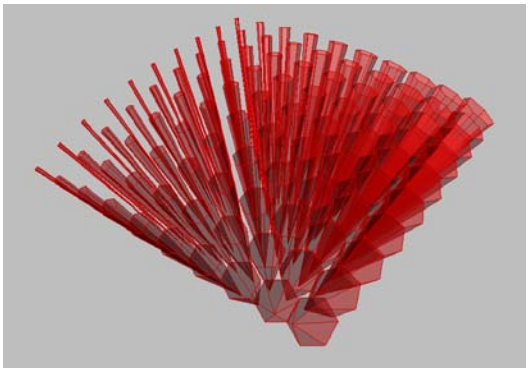


Fig4.10. In the last step, I used an <Extrude Point> component (Surface>Freeform) and I attached lines' start points as the points towards which I wanted my polygons to extrude.



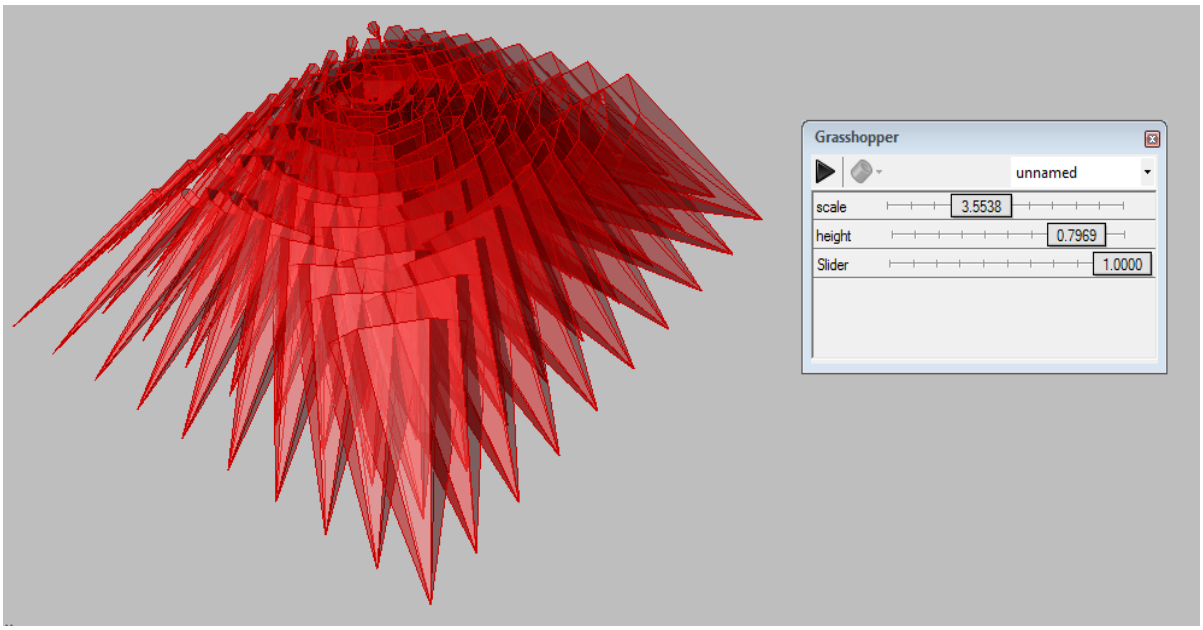


Fig4.11. Now by using 'Remote Control Panel' from View menu, you can simply change the values of number sliders for different options and check the overall look of the model and select the best one. Don't forget to uncheck the Preview option of unnecessary objects.

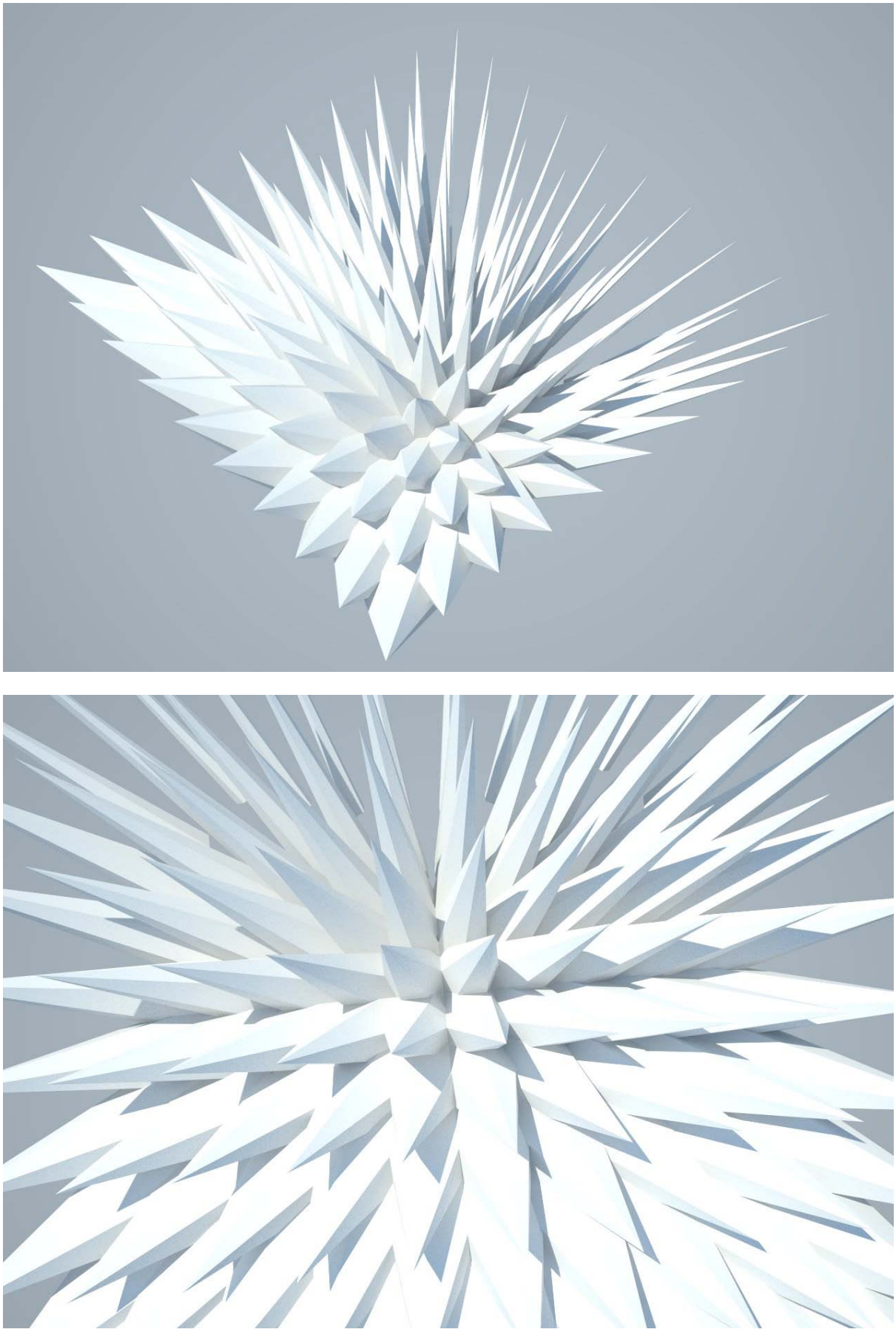


Fig.4.12. Final model

4_3_Combined Experiment: Swiss Re

Today it is very common to design the concept of towers with associative modelling methods. It allows designers to generate differentiated models, simple and fast. There are so many potentials to vary the design product and find the best concept quite quickly. Here I decided to model a tower and I think the "Swiss Re" tower from 'Foster and partners' seems sophisticated enough for modelling experiments.

First have a look at the project:

Fig.4.13. Swiss Re HQ, 30 St Mary Axe, London, UK, 1997-2004, Photos from Foster and Partners website, http://www.fosterandpartners.com.

Let me tell you the concept. I am going to draw a circle as outline of the tower and copy it to make some of the floors in which façade changes its curvature. Then I will rescale these floors to match the shape, and then I will make the skin of the tower by them. Finally for the façade's structural elements I will add up section polygons and make these elements with them. To do this process, I am going to assume the size and portions and I will deal with the model with very simple geometries to make the process simple.

Let's start with floors. I know that the Swiss Re's floors are circles that have some V-shaped cuts around them, but I just used a simple circle to make the outline of the tower. I want to copy these floors in certain heights which make it possible to play around proportions of the tower visually. As I said before, these points are located in positions of curvature change in façade.

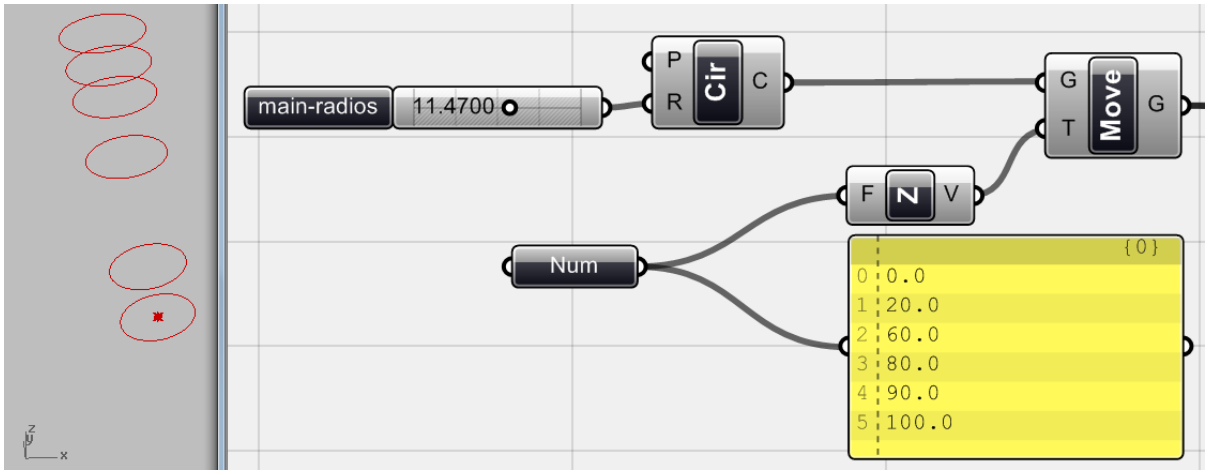


Fig.4.14. A <circle> component with <number slider> as radios is the outline of the tower. This circle copied by <move> component along Z direction by a <Z unit> vector component for 6 times above itself. These numbers are provided by 'set multiple numbers' manually and they are assumptions about the distance of different parts of the tower (based on the size of the base circle).

Although I generated these basic circles, all the same, but we know that all floors does not have same size, so we need to rescale them; If we look at the section of the tower we will see that from the circle which is grounded on earth, they first become bigger up to certain height, look constant on the middle parts and then become smaller and smaller up to the top point of the tower. So I need to rescale these sample floors, which means I have to provide a list of scale factors. Here again I am going to use another assumption about the scale factors of these sample floors. You could change these numbers to see if your project looks like the original design, more or less.

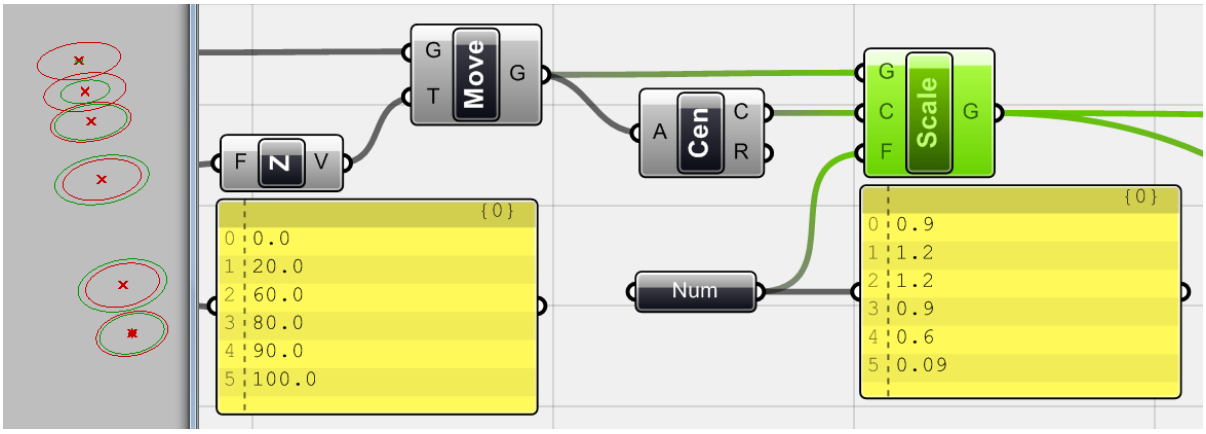


Fig.4.15. I need a <scale> component (XForm > Affine > Scale) to rescale my sample floors. The <scale> component needs the geometry to rescale, centre for scaling and the factor of scaling. So I need to feed the geometry part of it by our floors or circles which is the output of the <move> component. The predefined centre of scaling is the origin point, but if I scale floors by the origin as the centre, they would displace in space because their height also rescales. I need the centre of rescaling at the same level at each floor. It should be for each one and exactly at the centre of floor.

That's why I used a <Centre> component (Curve > Analysis > Centre) which gives me the centre of circles. By connecting it to the <scale> you can see that all circles would rescale in their level without displacement.

Again I have to say that factors of scale are assumptions about the scale factors in different height samples that I made before. These values could be changed to see which combination best fits the overall view. They all set in one <number> component.

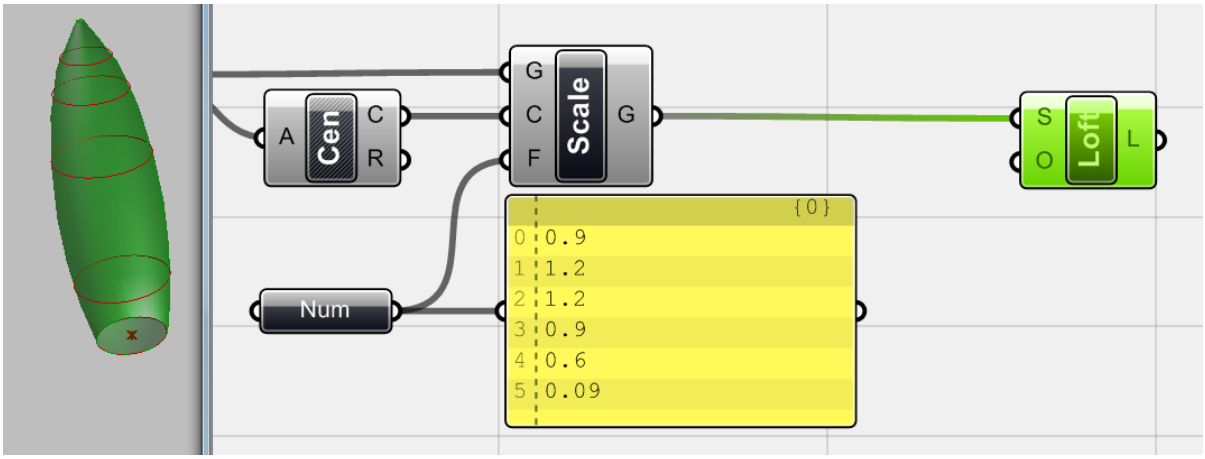


Fig.4.16. Now if I loft all these sample floors by a <loft> component (surface > freeform > loft) the first image of the tower appears. Little by little I should uncheck the preview option of the previously generated points and curves to clean up the scene.

Ok! Let's go for façade elements.

The facade's structural elements are helical shapes that have cross section like two connected triangles, but again to make it simple, I just model the visible part of it which is almost like a triangle in plan. I need these sections to 'loft' them to create their volume.

I want to generate these triangle sections on the façade. To do that, first I need to find the position of these triangles on the façade. I think if I generate a curve on the façade surface and divide it, it would be an acceptable place to posit all triangles before any transformation.

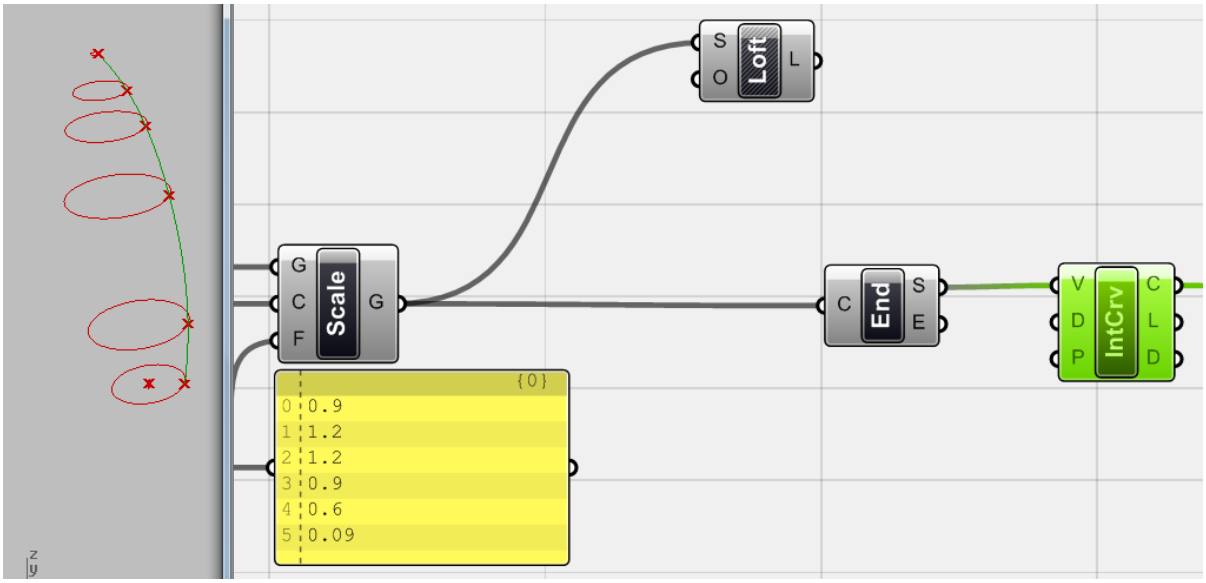


Fig.4.17. I used an <end points> component to get the start/end points of my sample floors. By attaching these points as the vertices to an <interpolate> component (curve > spline > interpolate) I would have a curve which is positioned on the façade.

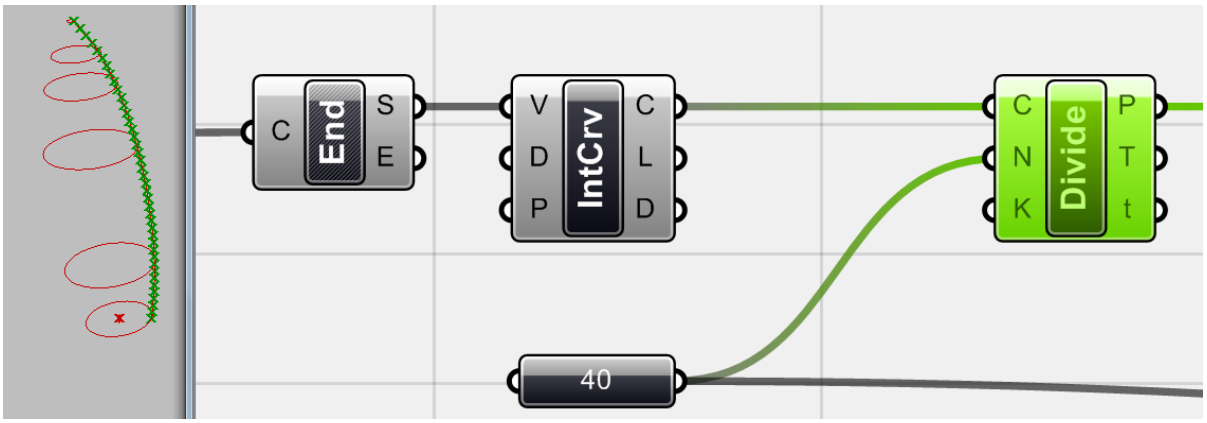


Fig.4.18. Here I divided the <interpolate> curve into 40 parts. The number of divisions helps the smoothness of the element when we want to adjust it on the façade.

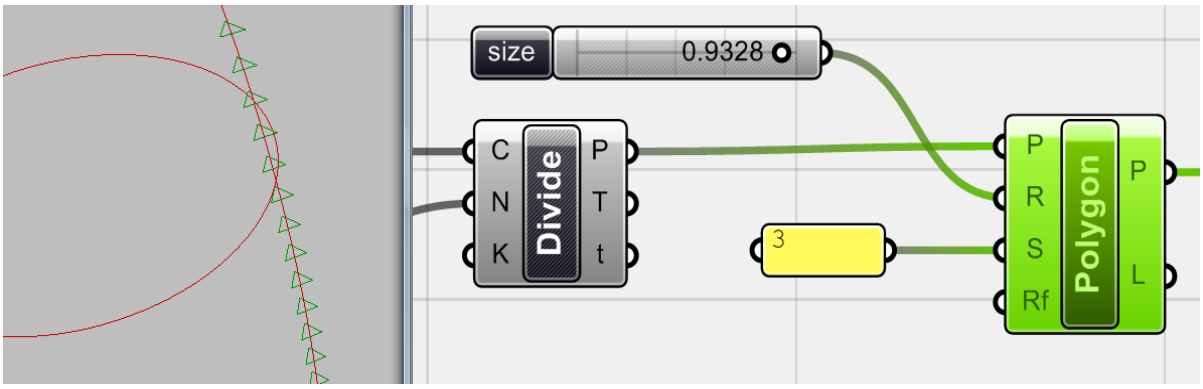


Fig.4.19. Now division points become base points to generate <polygon> on the façade. I set 'sides' to 3 to generate triangles and size of the elements, 'R' part, is controllable by <number slider>.

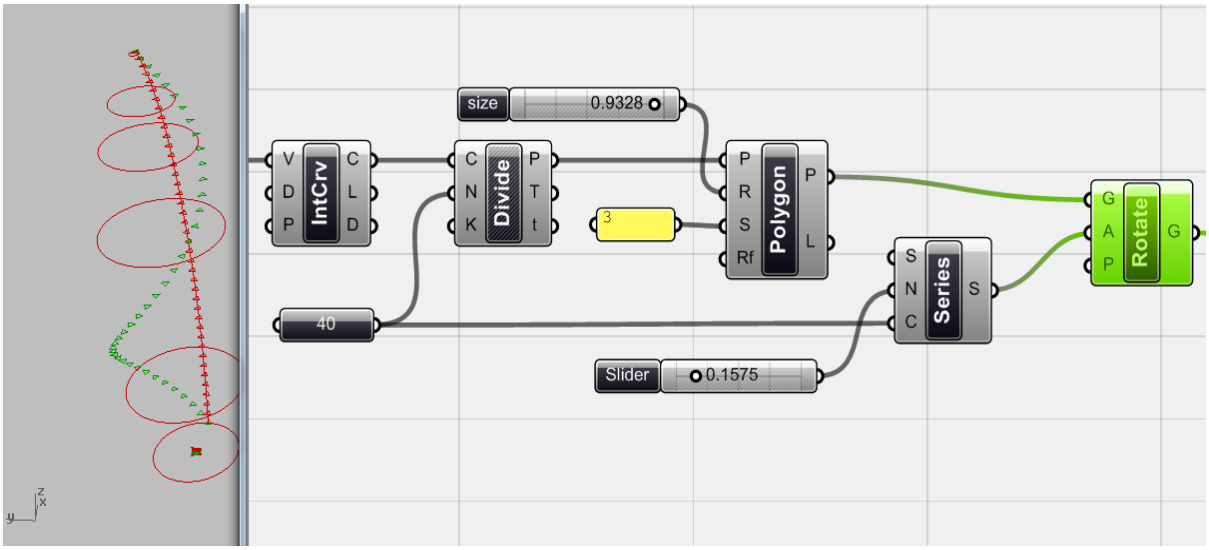


Fig.4.20. Façade structural elements are spiral and turn around the skin, up to the top point of tower. To achieve this, I have to rotate all triangle sections gradually. I want to use <Rotate> component and for that, I need to provide angles of rotation. As I said, angles of rotation should be a list of numbers growing slowly. The <series> component here generates our angles of rotation and it has as many items as the <divide> component (points-triangles) has. So as the result, all section triangles rotate around the façade.

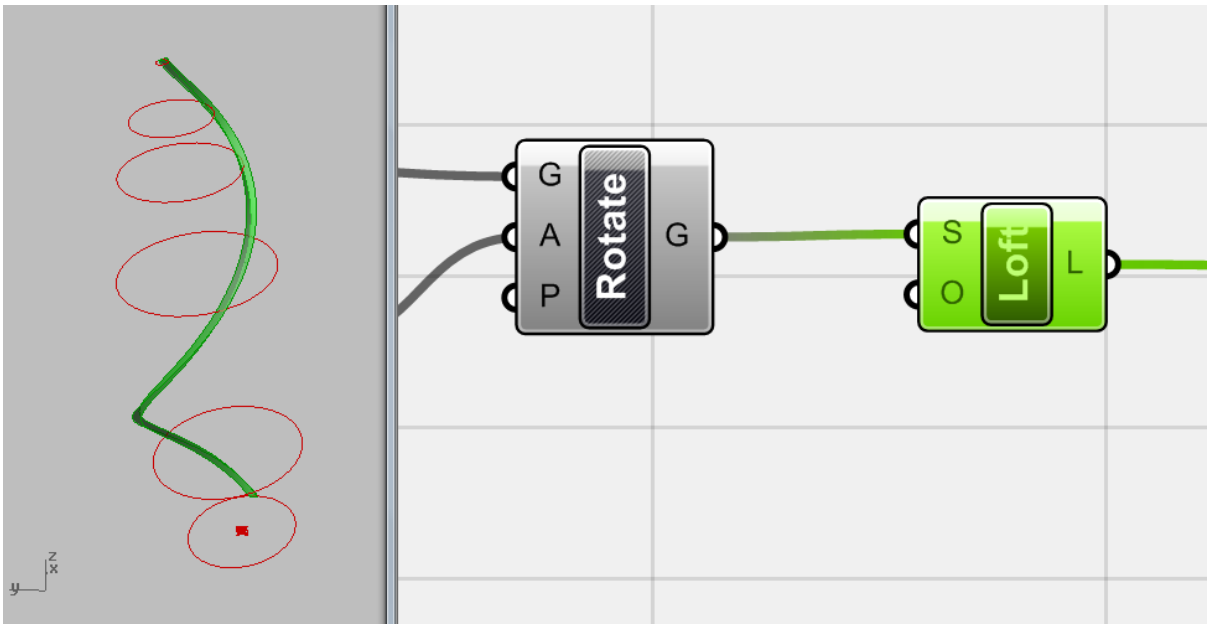


Fig.4.21. Now if I <loft> all section triangles, you see a single façade element appears. Degree of rotation and size of element is controllable so we need to match it to the façade in its best look.

Domains

As I mentioned before, Domains (or intervals) are numeric ranges. They are real numbers from lower limit to upper limit. Since I said 'real numbers' it means we have infinite numbers in between which means we need different types of usage for these numerical domains. As we experimented before, we can divide a numerical range and get divisions as evenly distributed numbers between two extremes.

Here I want to distribute façade elements all around the base circle. To do that, I need an interval to cover the whole base circle.

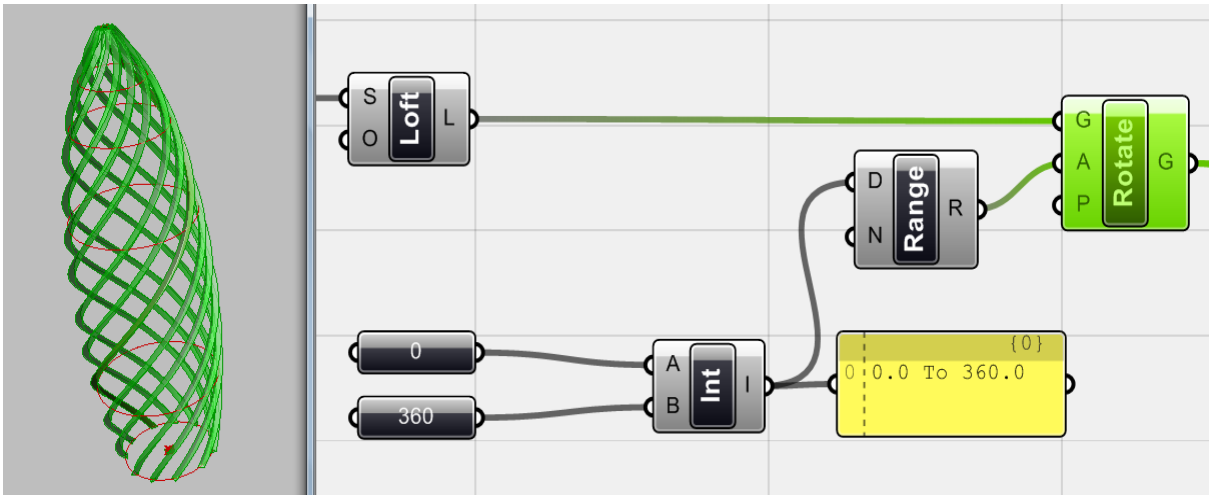


Fig.4.22. An <interval> component (as mentioned before, in new versions you can use Scalar > domain > domain) used to define numerical range from 0 to 360. This numerical range divided by a <range> component into 10 parts and the result is used as angle factors for a <rotate> component. So as it is shown in the image, the façade elements are distributed all around the base circle.

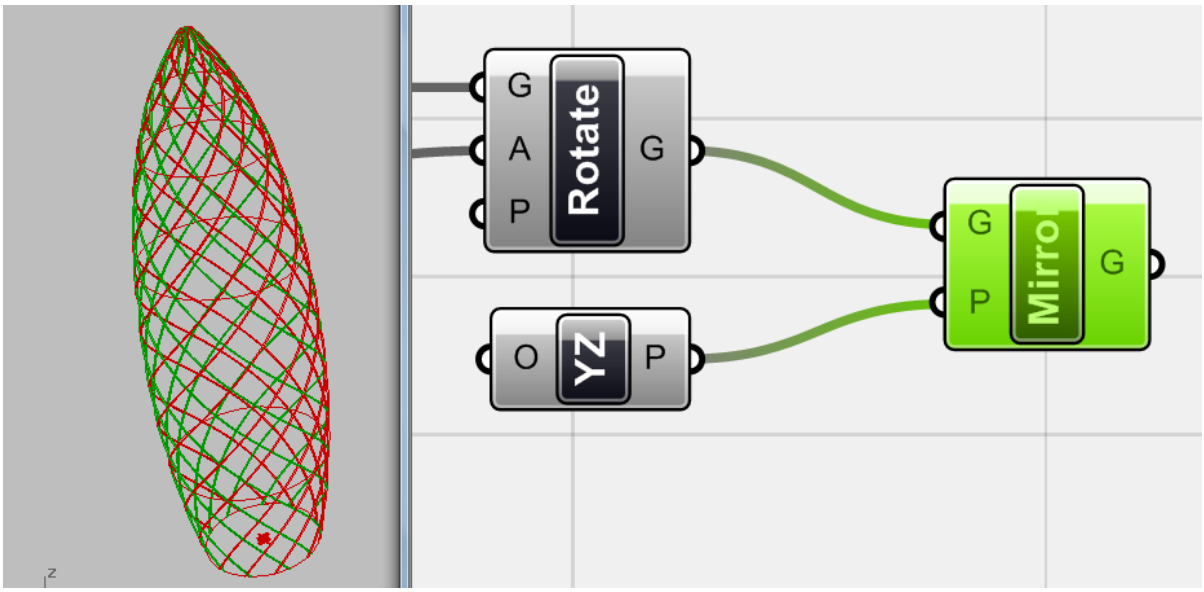


Fig.4.23. Now if I <mirror> (XForm > Euclidian > Mirror) the rotated geometries by <YZ plane> (Vector > Constants > YZ plane) I would have the façade elements in a mirrored helical shape. So at the end I have a lattice shape geometry around the tower.

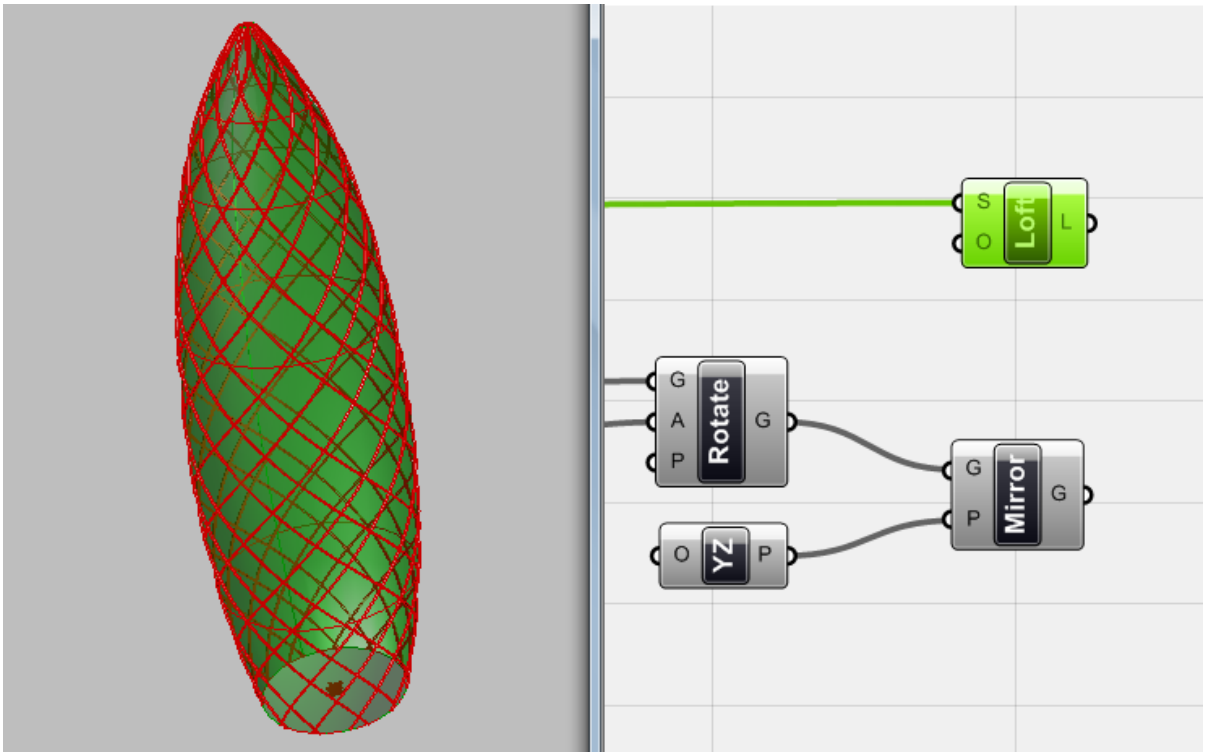


Fig.4.24. Preview the façade again, we have a rough representation of the 'Swiss Re' with an associative technique.

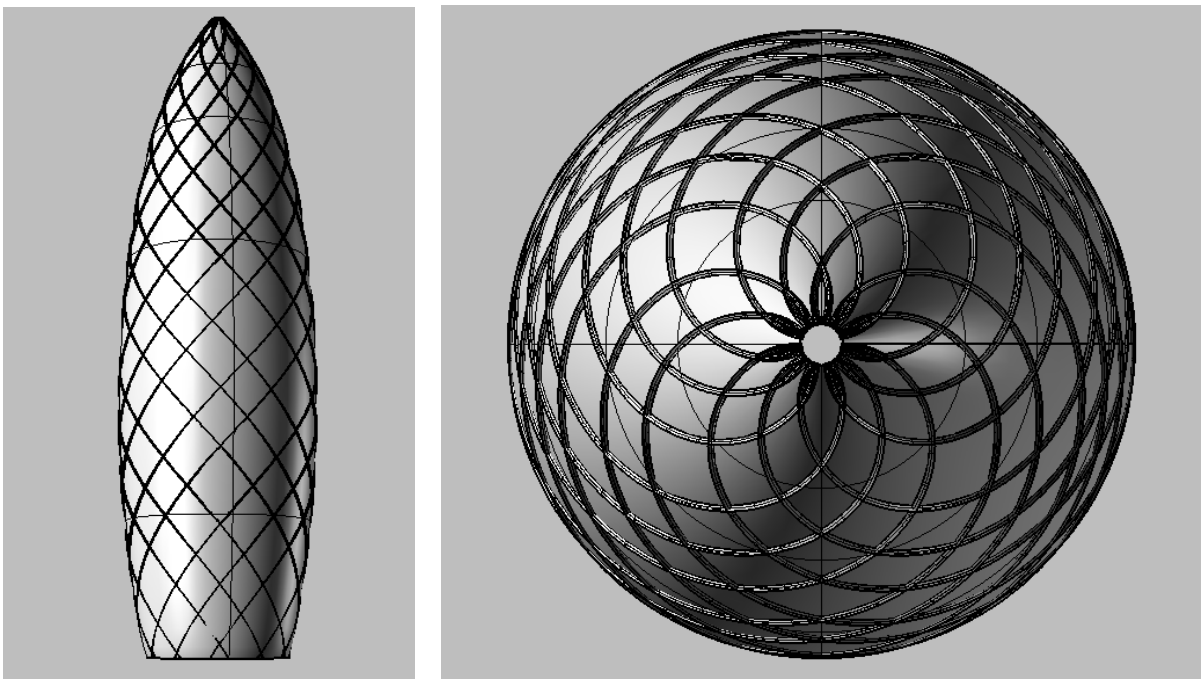
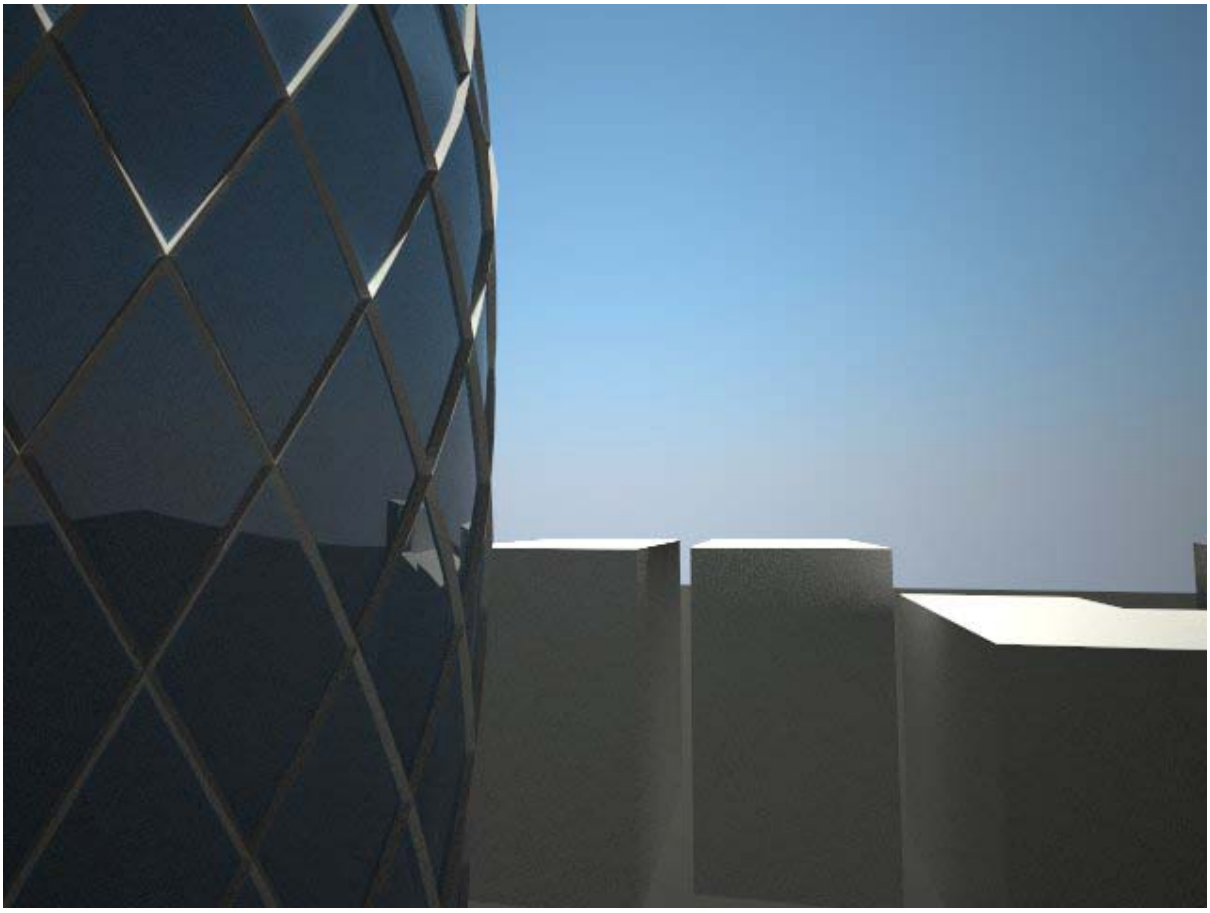


Fig.4.25. To generate the geometry in Rhino, Select those components which created the desired geometry in the scene, and select 'Bake Selected Objects' from canvas toolbar or component context menu.



4.26. Final model. Although it is not exactly the same as the original one, but for a sketch model in a short time, it would work.

Fig.4.27. Between main structural elements, there are other smaller scale structures and I think you can model them by yourself. Photo by the author.

4_4_On Attractors

"Attractor is a set of states of a dynamic physical system towards which that system tends to evolve, regardless of the starting conditions of the system. A **point attractor** is an attractor consisting of a single state. For example, a marble rolling in a smooth, rounded bowl will always come to rest at the lowest point, in the bottom center of the bowl; the final state of position and motionlessness is a point attractor." (Dictionary.com/Science Dictionary)

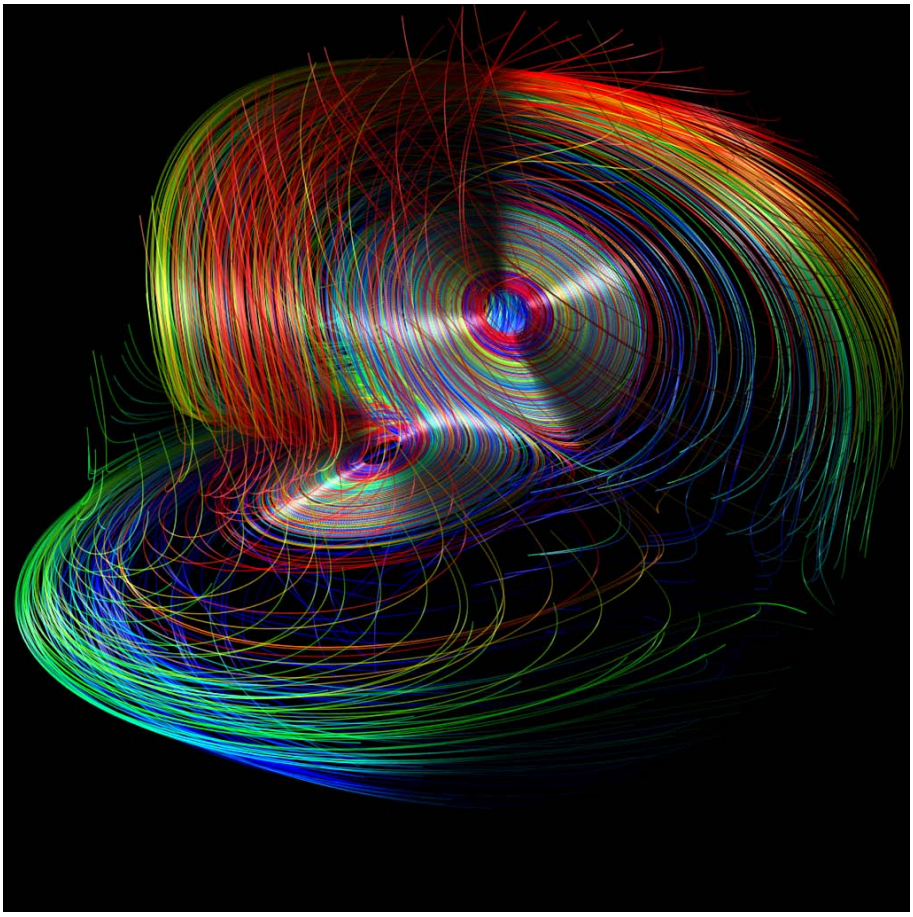


Fig.4.28. Strange Attractor (Illustration from: http://www.cs.swan.ac.uk/~cstony/research/star/)

In case of design and geometry, attractors are elements (usually points but could be curves or any other geometry) that affect other geometries in the space, change their behaviour and make them displace, re-orientate, rescale, etc. They can articulate the space around themselves and introduce fields of actions with specific radios of power. Attractors have different applications in parametric design, since they have the potential to change the whole objects of design constantly. Defining a field, attractors could also affect the multiple agent systems in multiple actions. The way they could affect the product and the power of attractors are all adjustable. We go through the concept of attractors in different occasions so let's have some very simple experiments first.

Point Attractors

I have a grid of points and I want to generate a set of polygons on them. I also have a point that I named it <attractor_1> and I draw a <circle> around it, just to realize it better. I want this <attractor_1> affects all my <polygon>s on its field of action. It means that based on the distance between each <polygon> and the <atractor_1>, and in domain of the <attractor_1>, each <polygon> responds to the attractor by change in its size.

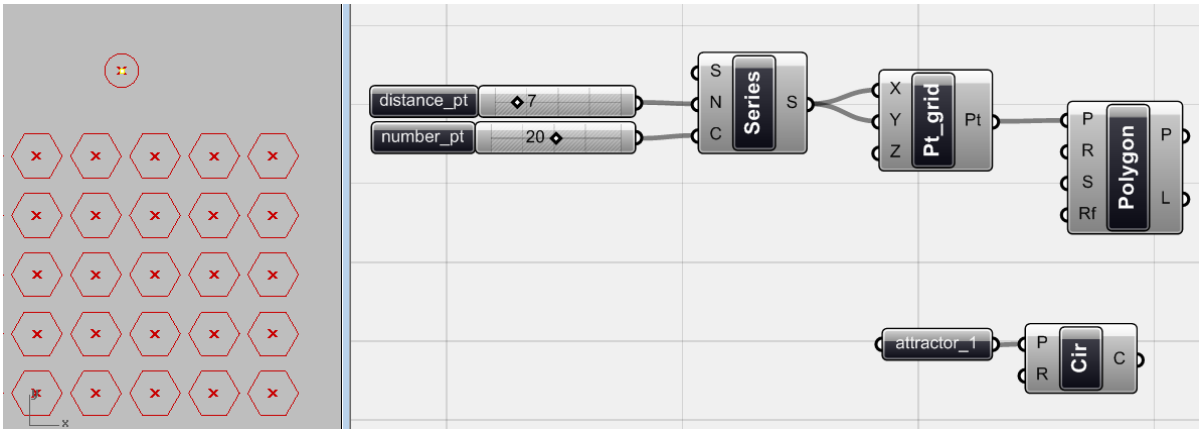


Fig.4.29. Base <point_grid> and <polygon>s and the <attractor_1>.

The algorithm is so simple. Based on the <distance> between <attractor_1> and the <Pt-grid>, I want to affect the radius of the <polygon>, so the 'relation' between attractor and polygons define by their distance.

I need a <distance> component to measure the distance between <attractor_1> and the polygon's center or <pt_grid>. Because this number might become too big, I need to <divide> (Scalar > Operators > Division) this distance by a given number from <number slider> to reduce the power of the <attractor_1> as much as I want.

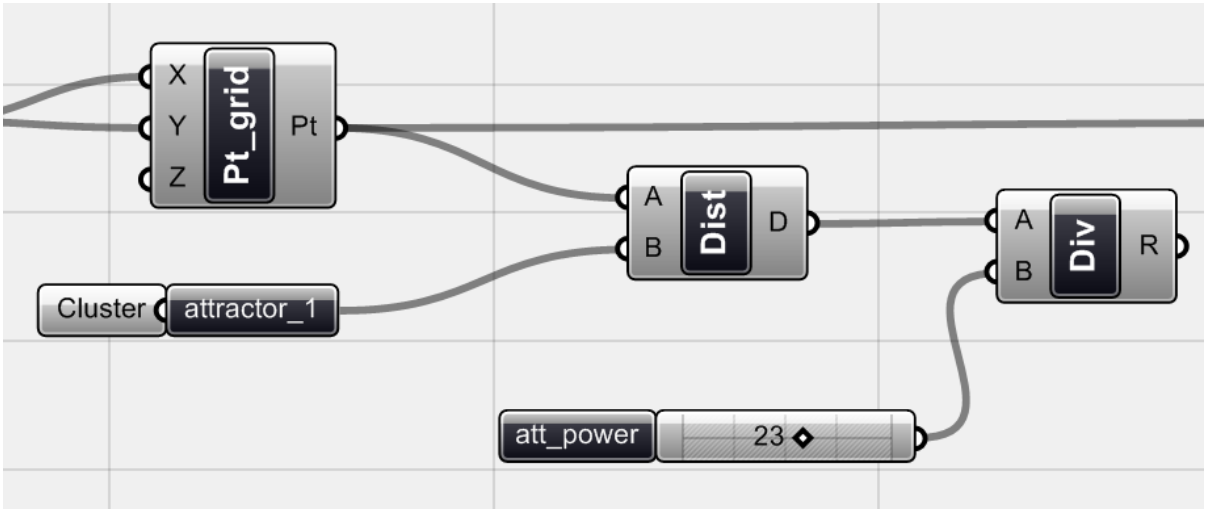


Fig.4.30. <Distance> divided by a number to control the 'power' of the <attractor_1>. Although I made a Cluster by <attractor_1> and its <circle> it seems that new versions of Grasshopper do not like clusters anymore so please use a simple <Pt> as <Attractor_1>.

Now if you connect this component to the Radios (R) part of the <polygon> you can see that the scale of polygons increases when they go farther than the <attractor_1>. Although this could be good for the first time, we need to control the maximum radius of the polygons, otherwise if they go farther and farther, they become too big, intersecting each other densely (it also happens if the power of the attractor is too high). So I have to control the maximum radios value of the polygons manually.

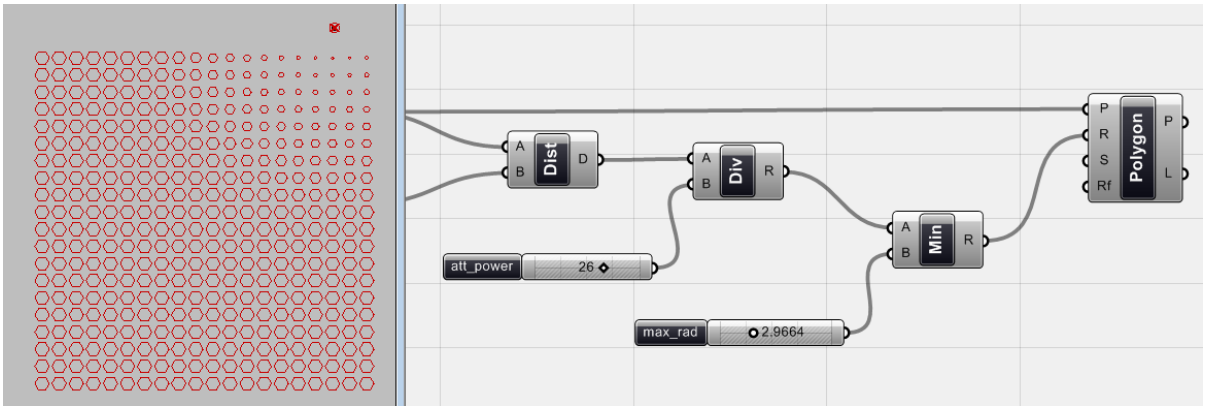


Fig.4.31. By using a <minimum> component (Scalar > Util > Minimum) and a user defined number, I am telling the algorithm to choose the value from the component, if it is smaller than the number that I define as the maximum radios by <number slider>. As you can see in the image, those polygons that going to be bigger than the <max_rad> remain constant, and we can literally say they are not in the power field of attractor.

Now if you change the position of the <attractor_1> in the Rhino workplace manually, you can see that all polygons get their radius according to the <attractor_1> position.

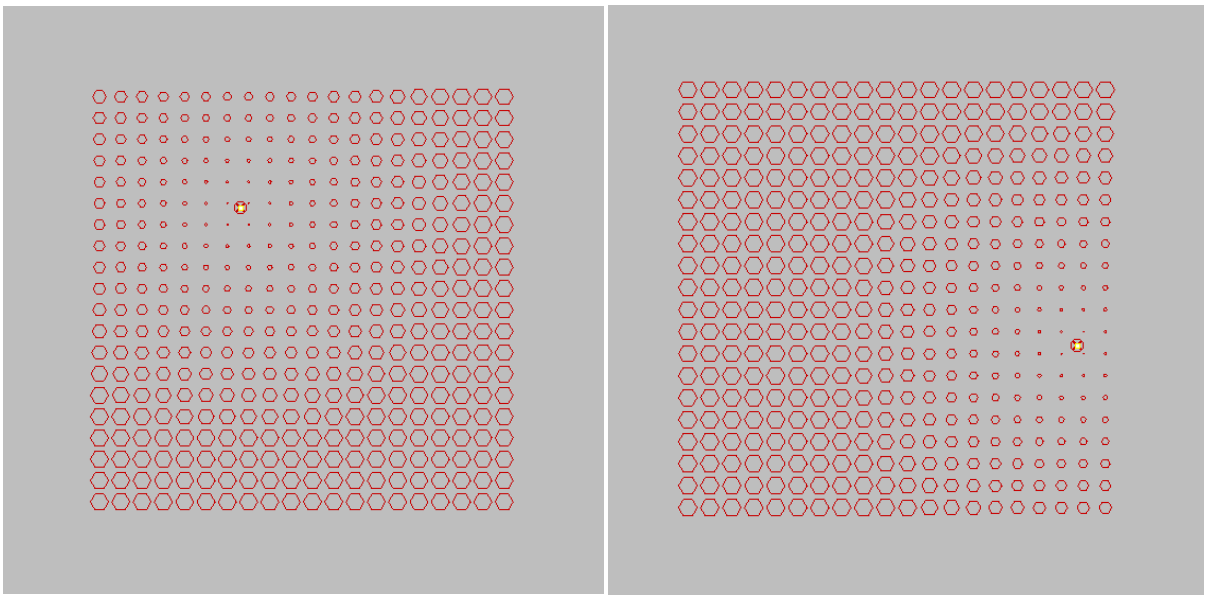


Fig.4.32. Effect of the <attractor_1> on all polygons. Displacement of the attractor, affects all polygons accordingly.

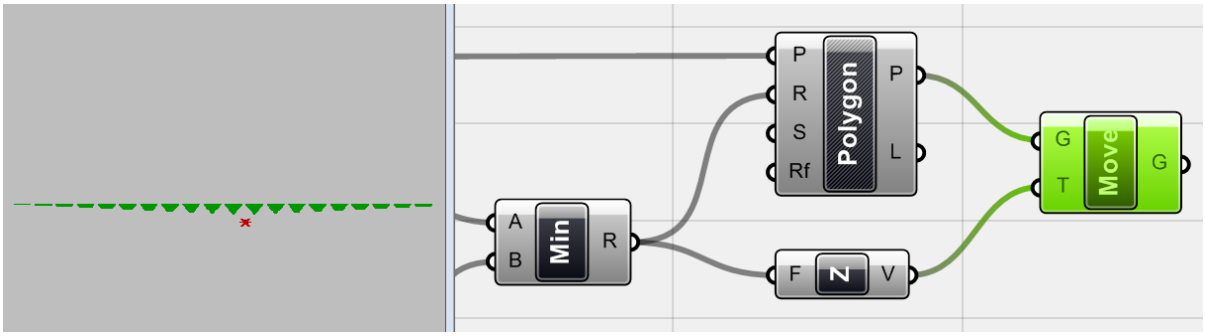


Fig.4.33. With the same concept, I can displace polygons in Z direction based on the numbers coming from the <Min> component or changing it by mathematical functions, if necessary. So the usage of attractors is not limited to size only!

Simple !!!!!!! I can do any other function on these polygons like rotate, change colour, etc. But let's think what would happen if I had two attractors in the field. I make another cluster which means another point in Rhino associated with a <point> and <circle> in Grasshopper.

It seems that the first part of the algorithm is the same. Again I need to measure the distance between this <attractor_2> and the polygons' center or <pt_grid> and then divide it by the same <number slider> as <att_power> to control the power of attractor.

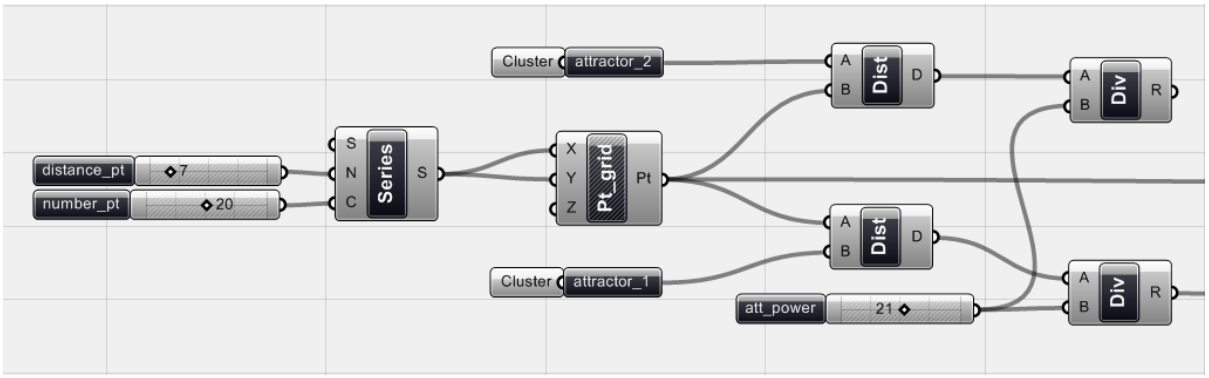
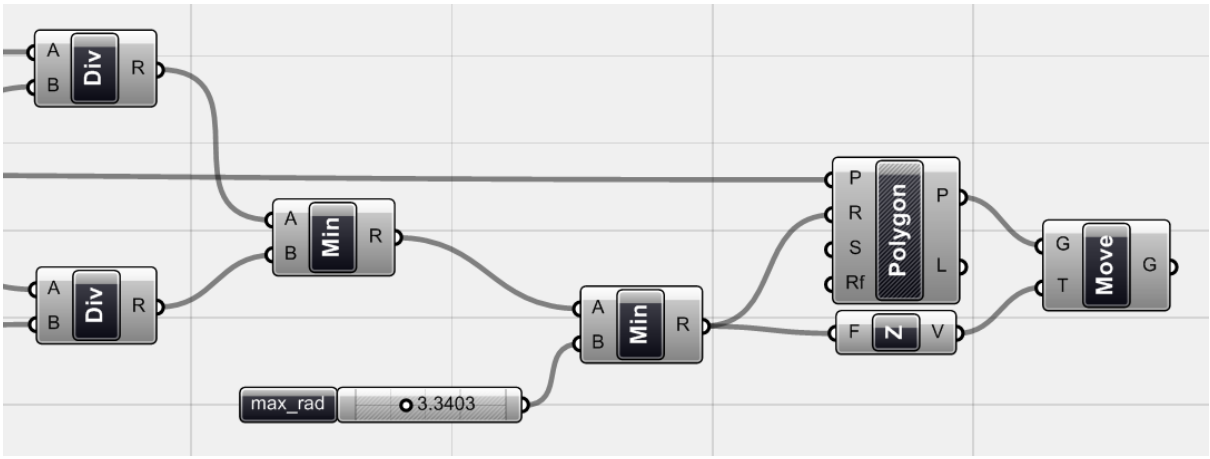


Fig.4.34. Introducing second <attractor_2> and applying the same algorithm to it.

Now we have two different data lists that include the distance from the polygon to each attractor. Since the closer attractor would affect the polygon more, I should find which one is closer and use that one, as the source of action. I will use a <min> component to find which distance is minimum or which point is closer.



4.35. Finding the closer attractor. After finding the closer one by <min> component, the rest of the process would be the same. Now all <polygon>s are being affected by two attractors.

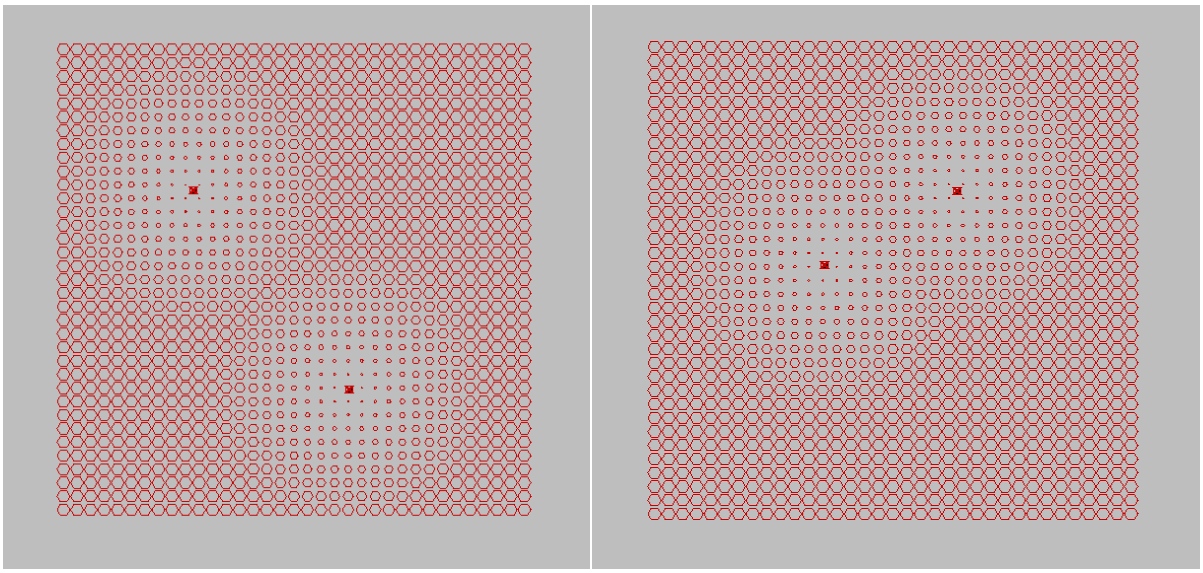


Fig.4.36. Again you can change the position of the attractors and see how all polygons reacting accordingly.

We can add more and more attractors. The concept is to find the attractor which is closer for each polygon and apply the predefined effect. This concept is useful to deal with design issues with huge amount of small scale elements.

Curve Attractors: Wall project

Let's complete this discussion with another example but this time by Curve attractors because in so many cases you need to articulate your field of objects with linear attractors instead of points.

My aim here is to design a porous wall for an interior space to have a multiple framed view to the other side. This piece of work could be cut from sheet materials. In my design space, I have a plane sheet (wall), two curves and bunch of randomly distributed points as base points of cutting shapes. I decided to generate some rectangles by these points, cut them out of a sheet, to make this porous

wall. I also want to organize my rectangles by this two given curves so at the end, my rectangles are not just some scattered rectangles, but randomly distributed in accordance to these curves which have a level of organisation in macro scale and controlled randomness in micro scale.

What I need is to generate this bunch of random points and displace them towards the curves, literally based on the amount of power that they receive from them. I also decided to displace points towards both curves, so I do not need to select closer one. Then I want to generate my rectangles over these points and finally I will define the size of these rectangles in relation to their distance to the attractors.

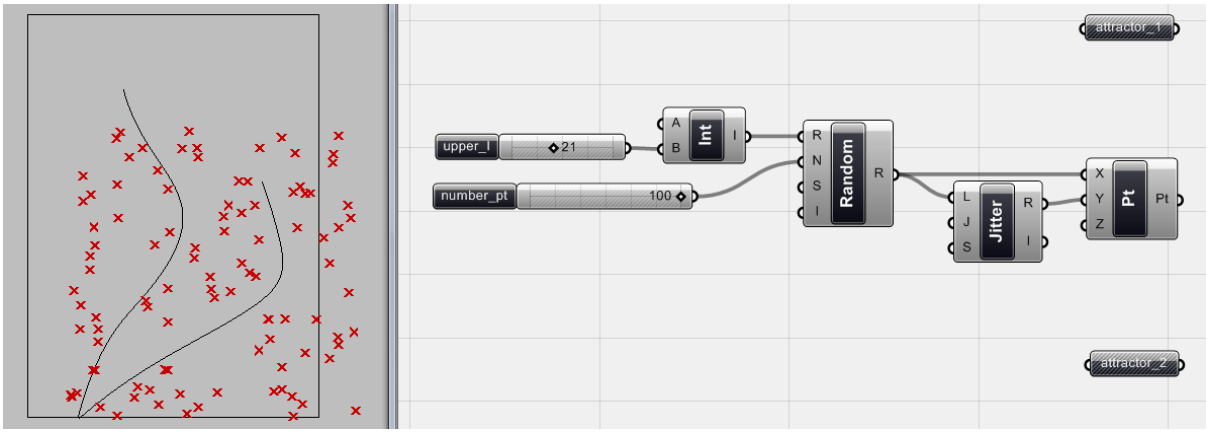


Fig.4.37. Generating a list of randomly distributed <point>s and introducing attractors by two <curve> component (Params > Geometry > Curve) in the space of a sheet. I used an <interval> component to define the numeric interval between 0 (defined manually) and <number slider> for the range of random points (you should use <Domain> component in new versions as mentioned before). I would rename the <Pt> to the <Rnd_Pt_Grid>.

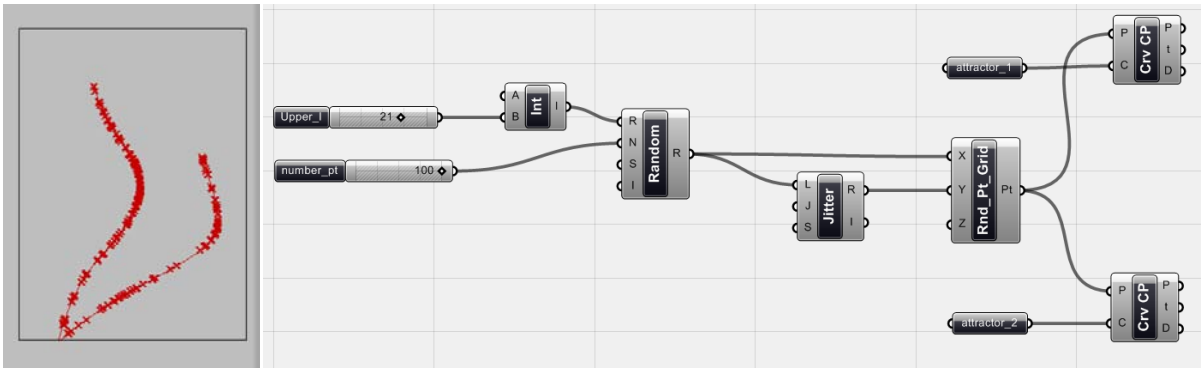


Fig.4.38. When the attractor is a point, you can simply displace your geometry towards it. But when the attractor is a curve, you need to find a relative point on curve and displace your geometry towards that specific point. And this point must be unique for each geometry, because there should be one to one relation between attractor and any geometry in the field. If we imagine an attractor like a magnet, it should pull the geometry from its closest point to the object. So basically what I first need is to find the closest point of <Rnd_pt_grid> on both attractors. These points are the closest points on attractors for each member of the <Rnd_Pt_Grid> separately. I used <Curve CP> component (Curve > Analysis > Curve CP) which gives me the closest point of curve to my <Rnd_Pt_Grid>.

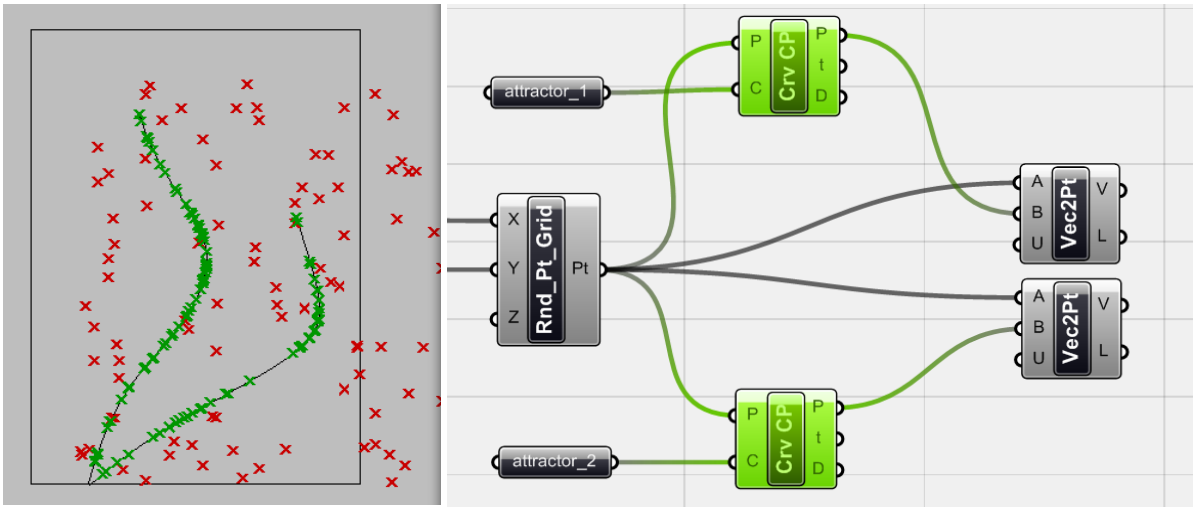


Fig.4.39. In order to displace points towards the attractors, I need to define a vector for each point in <Rnd_Pt_Grid>, from point to its closest point on the attractors. Since I have the start and end point of the vector I use a <vector 2Pt> component. The second point of the vector (B port of the component) is the closest point on the curve.

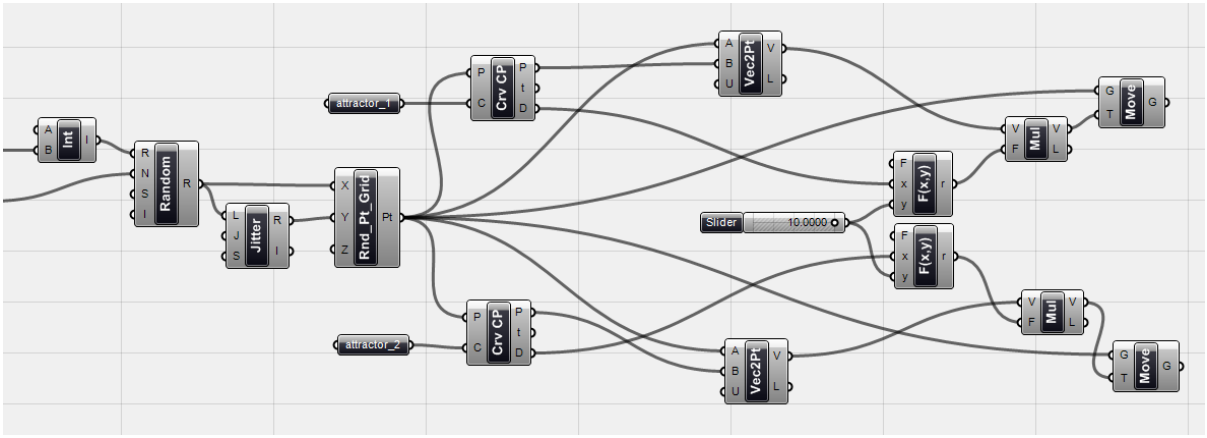


Fig.4.40. Now I connected all my <Rnd_Pt_Grid> to two <move> components to displace them towards the attractors. But if I use the vector which I created in the last step, it displaces all points on curves and that's not what I want. I want to displace the points in relation to their distance to the attractor curves. If you look at the <Curve CP> component it has an output which gives us the distance between each point and the relevant closest point on curve. Good! We do not need to measure the distance by another component. I just used a <Function> component and I attached the distance as X and a <number slider> to Y to divide the X/Log(Y) to control the factor of displacement (Log function change the linear relation between distance and the resulting factor).

Now I need to change the size of my vectors based on these newly created factors. Here I need a component to change the size of a vector and that's why I used a <multiply> component (Vector > Vector > Multiply) which does that for me, so I attached the <vector 2P> as base vectors and I changed their size by the size factors, and I attached the resulting vectors to the <move> components which displace the <Rnd_Pt_Grid> in relation to their distance to the attractors, and towards them.

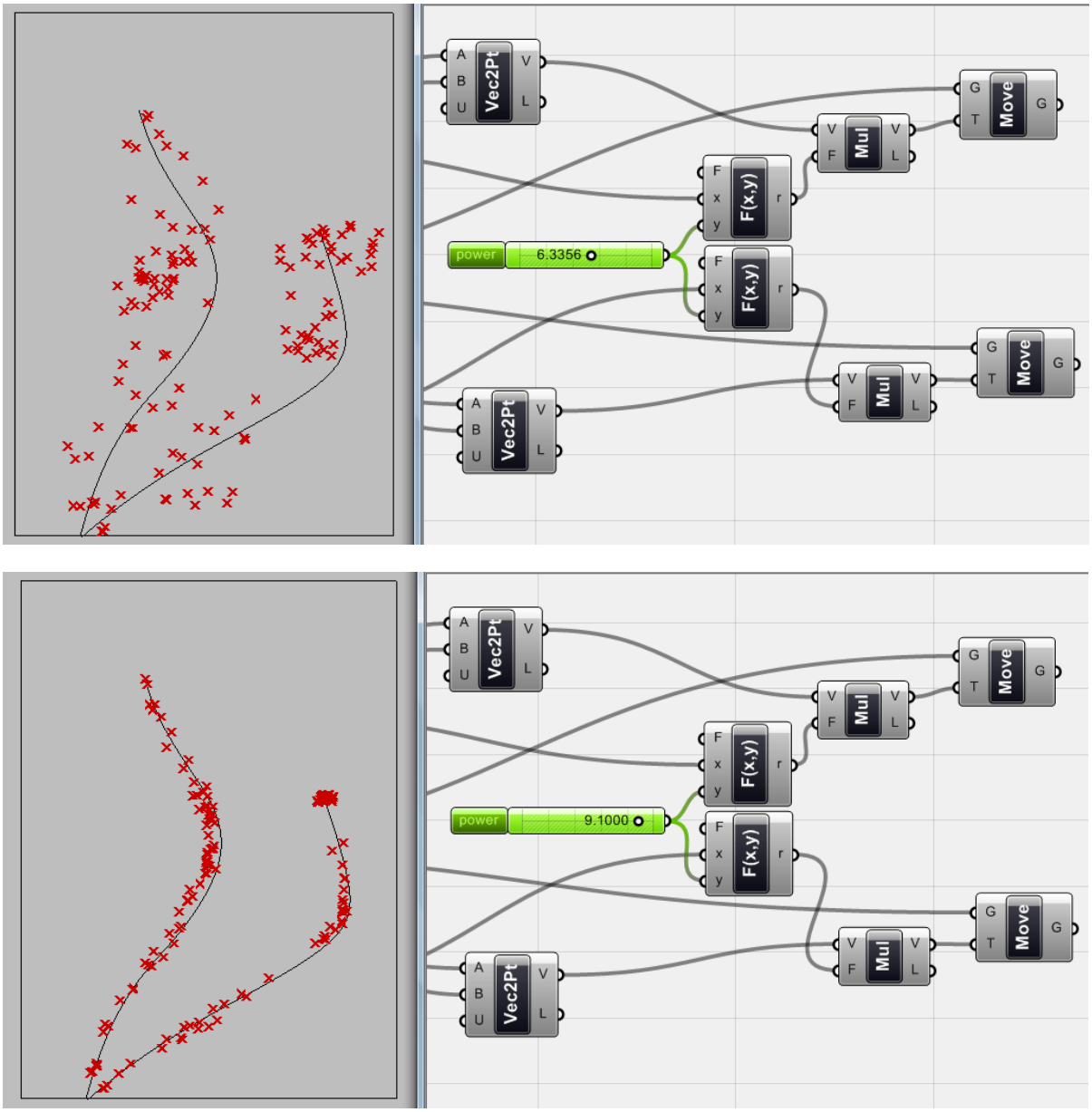


Fig.4.41. The <number slider> changes the power with which attractors displace objects towards themselves.

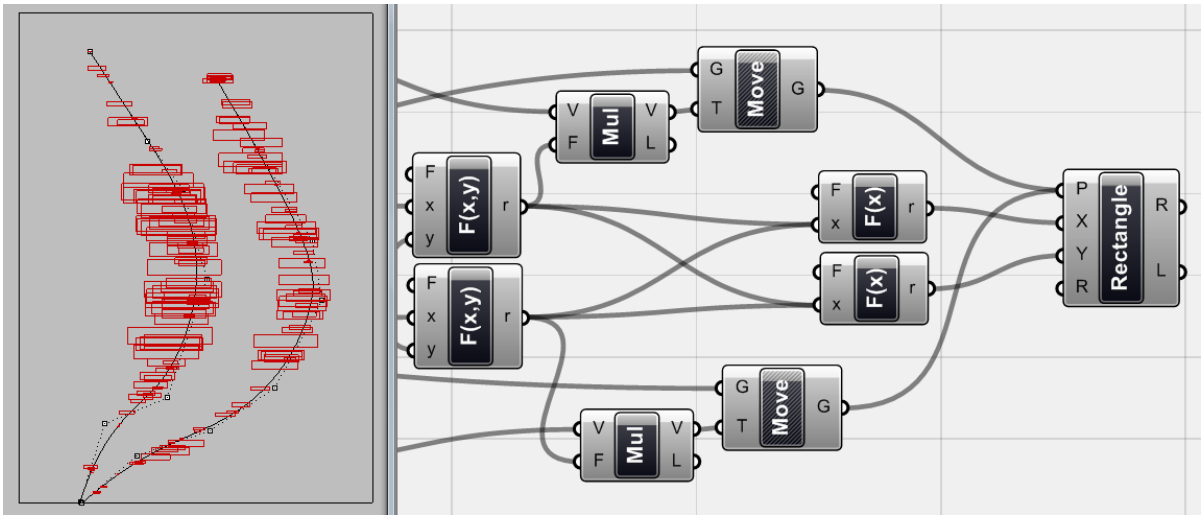


Fig.4.42. The next step is the generation of rectangles. I used a <rectangle> component (curve>primitive) and I attached the <move>d or displaced points to it as base points. But as I told you, I want to change the size of the <rectangle>s based on their distances to each <attractor> as well. So I used the same numerical values which I used for vector magnitude and I changed them by two functions. I divided these factors by 5 for the X value of the rectangles and I divided them by 25 for their Y value. As you can see, rectangles have different dimensions based on their original distance from the attractor but they all have same ratio because of the above division factors. You can change these division factors (5, 25) to anything you want or use sliders to change them gradually to see which ratio is more delicate for your taste.

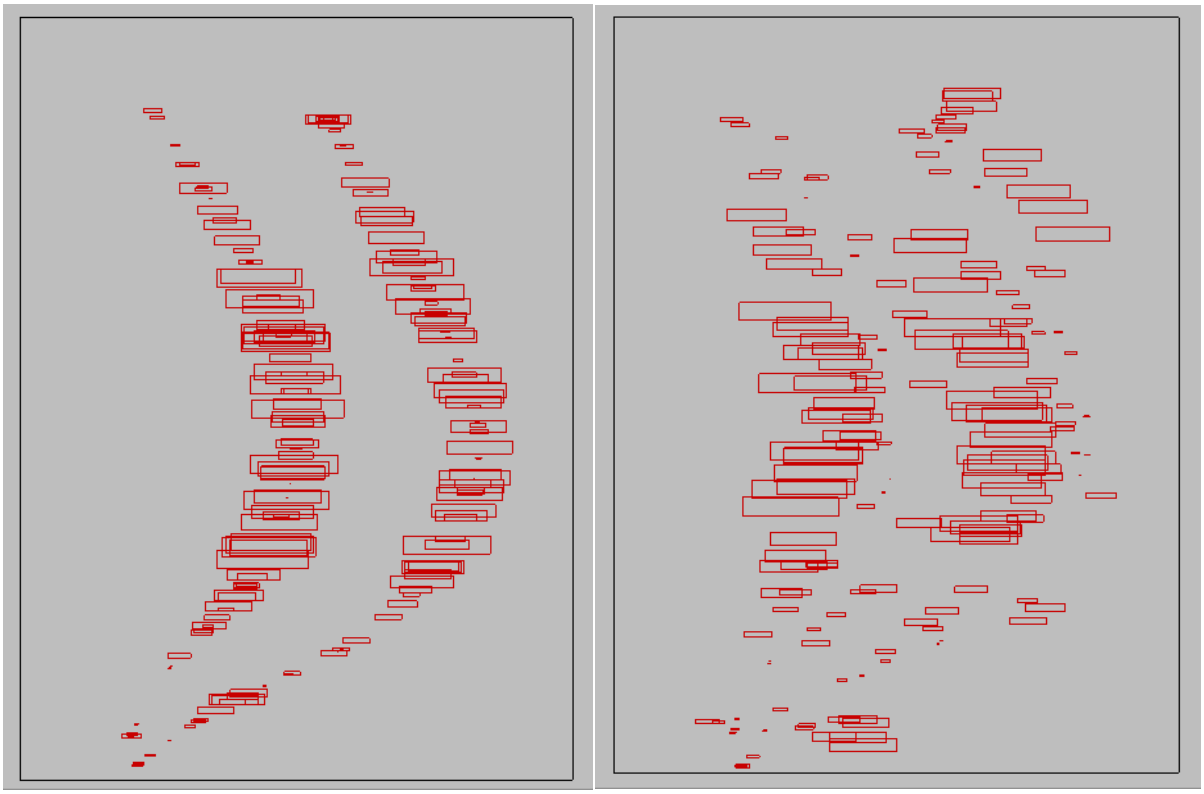


Fig.4.43. Manipulating the variables would result in different models and I can choose the best one for my design purpose.

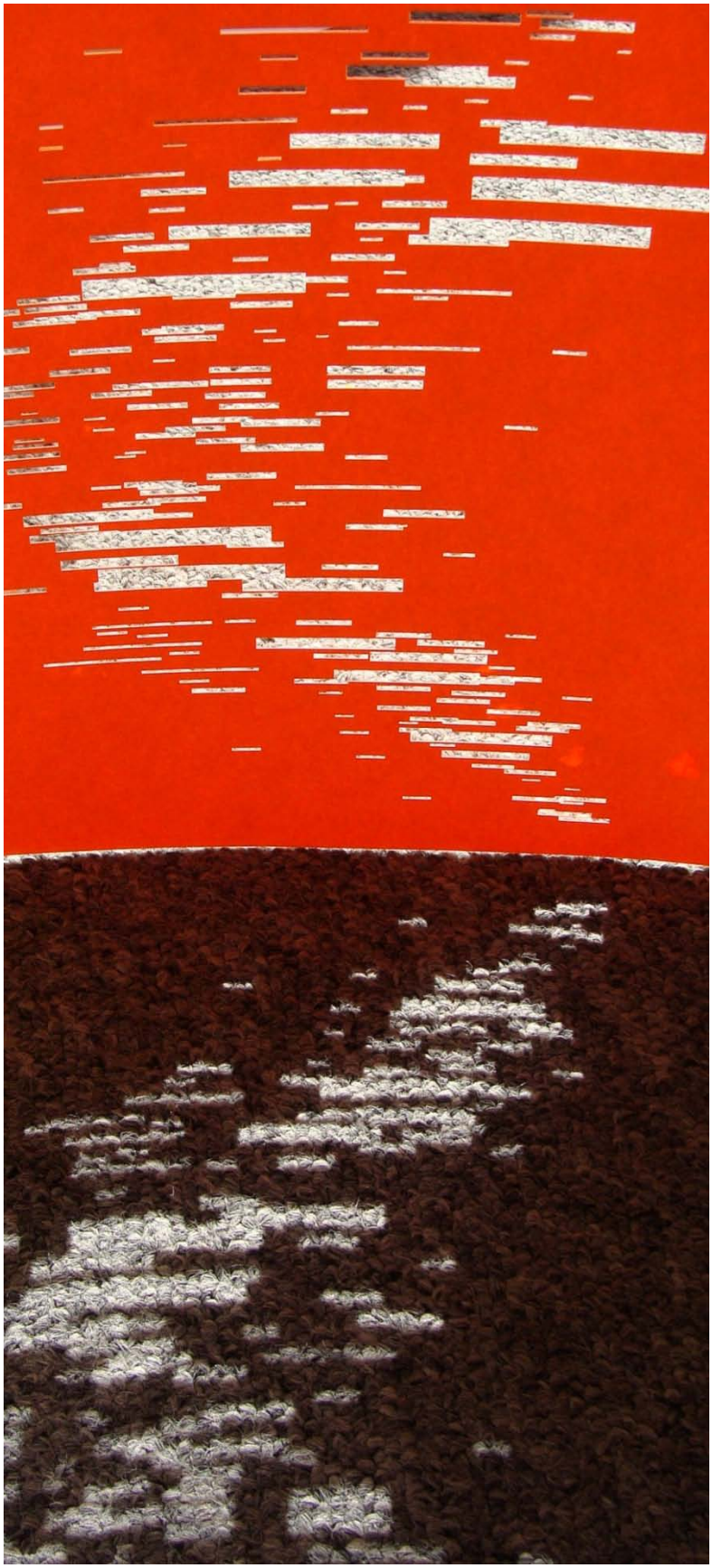


Fig.4.44. Model of the final design product as a porous wall system. Different shadow effects which can be considered as a factor to control the size of openings.

Chapter_5_Parametric Space

Chapter_ 5_Parametric Space

Our survey in Geometry observes objects in space; Digital representation of forms and tectonics; different articulation of elements and multiple processes of form generations; from classical ideas of symmetry and pattern up to NURBS and Meshes.

We are dealing with objects. These objects could be boxes, spheres, cones, curves, surfaces or any articulation of them. In terms of their presence in the space they generally divided into points as 0-dimensional, curves as 1-dimensional, surfaces as 2-dimensional and solids as 3-dimensional objects.

We formulate the space by coordinate systems to identify some basic properties like position, direction and measurement. The Cartesian coordinate system is a 3 dimensional space which has an Origin point O=(0,0,0) and three axis intersecting at this point which make the X, Y and Z directions. But we should consider that this 3D coordinate system also includes two-dimensional (flat space (x, y)) and one-dimension (linear space (x)) systems as well. What we know as parametric design, deals with these spaces. We need to go through these spaces to design 'parametric' with freeform curves and surfaces. While parametric design shifts between these spaces, we need to understand them as parametric spaces.

5_1_One Dimensional (1D) Parametric Space

The X axis is an infinite line which has some numbers associated with different positions on it. Simply x=0 means the origin and x=2.35 a point on the positive direction of the X axis which is 2.35 unit away from the origin. This simple, one dimensional coordinate system could be parameterised on any curve in the space. So basically not only the World X axis has some real numbers associated with different positions on it, but also any curve in the space has the potential to be parameterized by a series of real numbers that show different positions on the curve. So in our 1D parameter space when we talk about a point, it could be described by a real number which is associated with a specific point on a curve we are dealing with.

It is important to know that since we are not working on the world X axis any more, any curve has its own parameter space and these parameters does not exactly match the universal measurement systems. Any curve in Grasshopper has a parameter space starts from zero and ends in a positive real number (Fig.5.1).

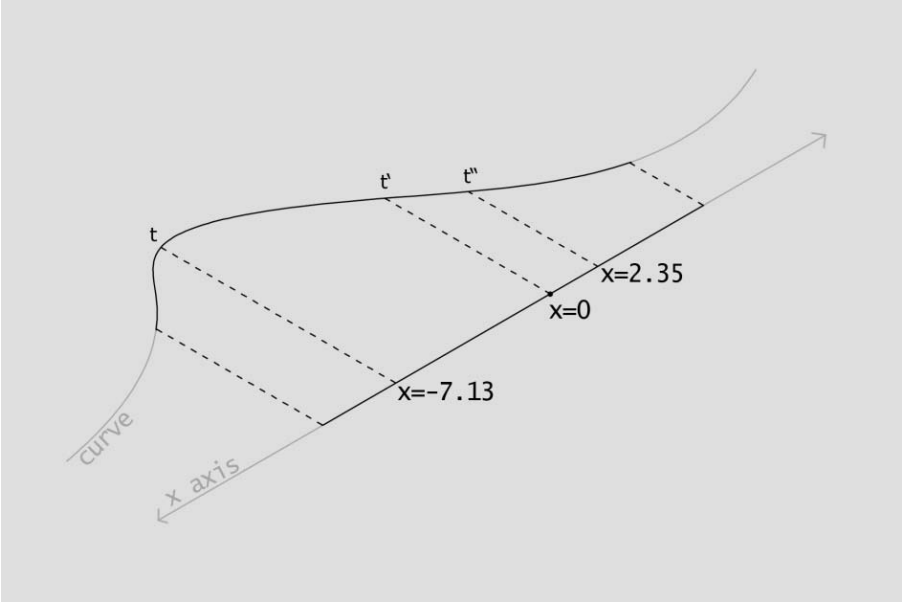


Fig.5.1. 1D-parameter space of a curve. Any 't' value is a real number associated with a position on the curve.

So talking about a curve and working and referencing some specific points on it, we do not need to deal always with points in 3D space with p=(X,Y,Z) but we can recall a point on a curve by p=t as a specific parameter on it. And it is obvious that we can always convert this parameter space to a point in the world coordinate system. (Fig.5.2)

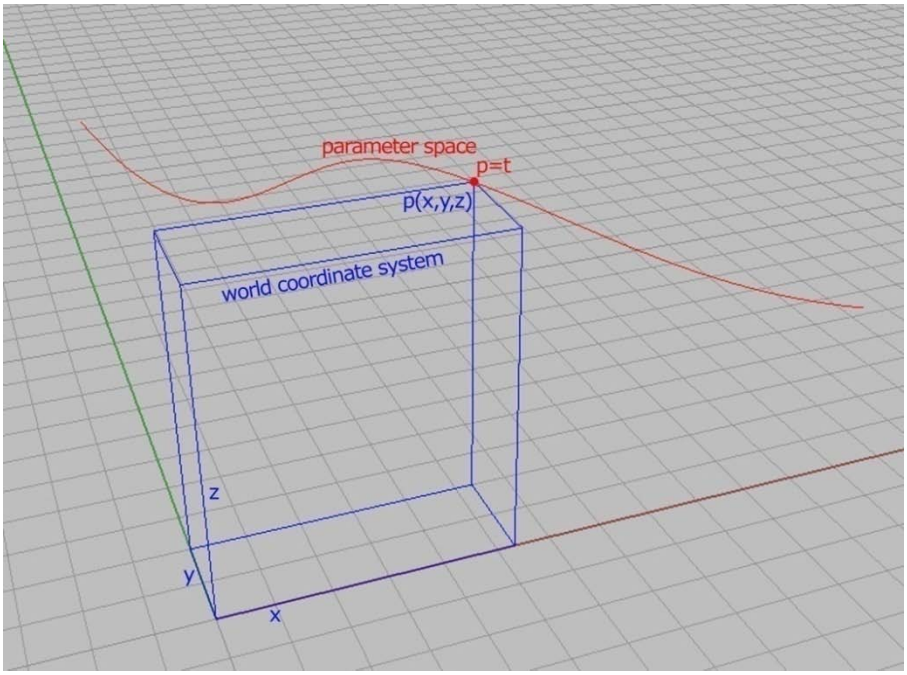


Fig.5.2. 1D-parmeter space and conversion in 3D coordinate system.

5_2_Two Dimensional (2D) Parametric Space

Two axis, X and Y of the World coordinate system deals with the points on an infinite flat surface in that, each point on this space is associated with a pair of numbers p=(X,Y). Quite the same as 1D space, here we can imagine that all values of 2D space could be traced not only on World's coordinate flat surface, but also on any surface in space. So basically we can parameterize a coordinate system on a curved surface in space, and call different points of it by a pair of numbers here known as UV space, in which any point P on the surface is P=(U,V). Again we do not need to work with 3 values of P=(X,Y,Z) as 3D space to find the point and instead, we can work with the UV "parameters" of the surface. (Fig.5.3)

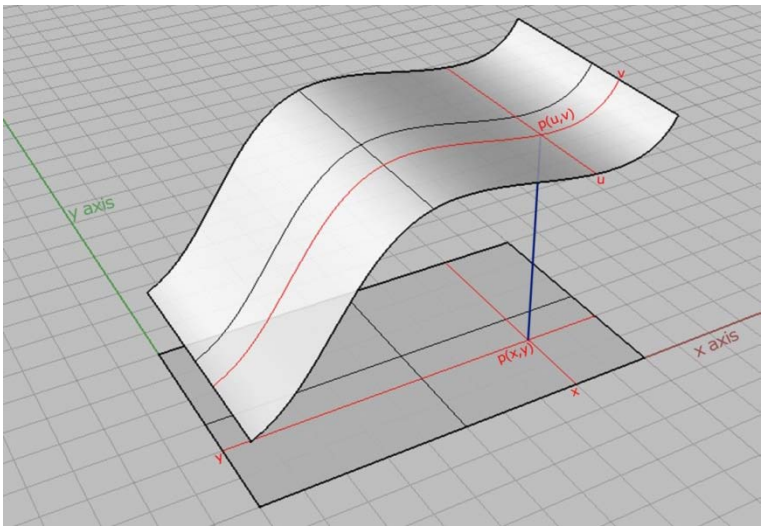


Fig.5.3. UV and 2D parameter space.

These "Parameters" are specific for each surface by itself and they are not generic data like the World coordinate system, and that's why we call them parametric! Again we have access to the 3D equivalent coordinate of any point on the surface. (Fig.5.4)

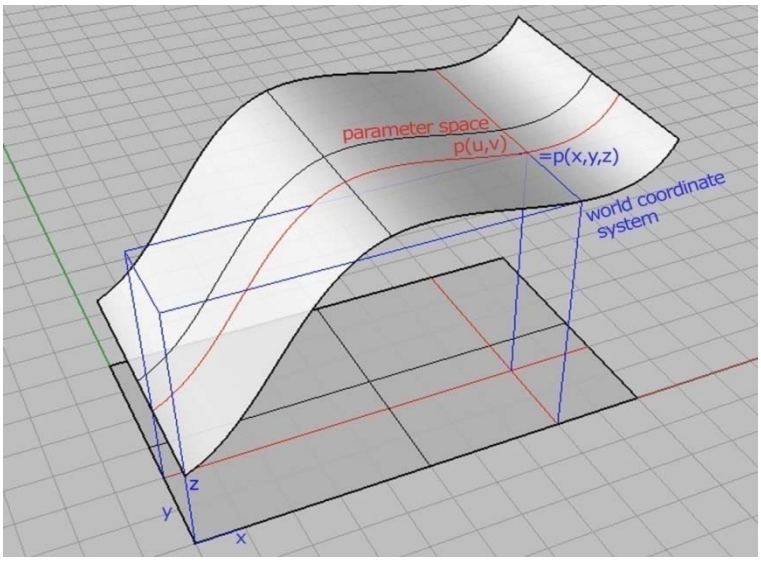


Fig.5.4. Equivalent of the point P=(U,V) on the world coordinate system p=(X,Y,Z).

5_3_Transition between spaces

It is a crucial part in parametric thinking of design to know exactly which coordinate system or parameter space we need to work with, in order to design. Working with free form curves and surfaces, we need to provide data for parameter space but we always need to go back and forth for the world coordinate system to provide data for other geometry creations or transformations. It is more complicated in scripting, but since Grasshopper has a visual interface rather than code, you would simply identify which sort of data you need to provide for your design purpose.

Here note that it is not always a parameter or a value in a coordinate system that we need in order to call geometries in Generative Algorithms and Grasshopper, sometimes we need just an index number to do it. If we are working with bunch of points, lines or whatever, and they have been generated as a group of objects, like point clouds, since each object associated with a natural number that shows the position of it in a list of all objects, we just need to call the number of the object as its 'index' instead of any coordinate system. The index numbering like array variables in programming is a 0-based counting system (Fig.5.5).

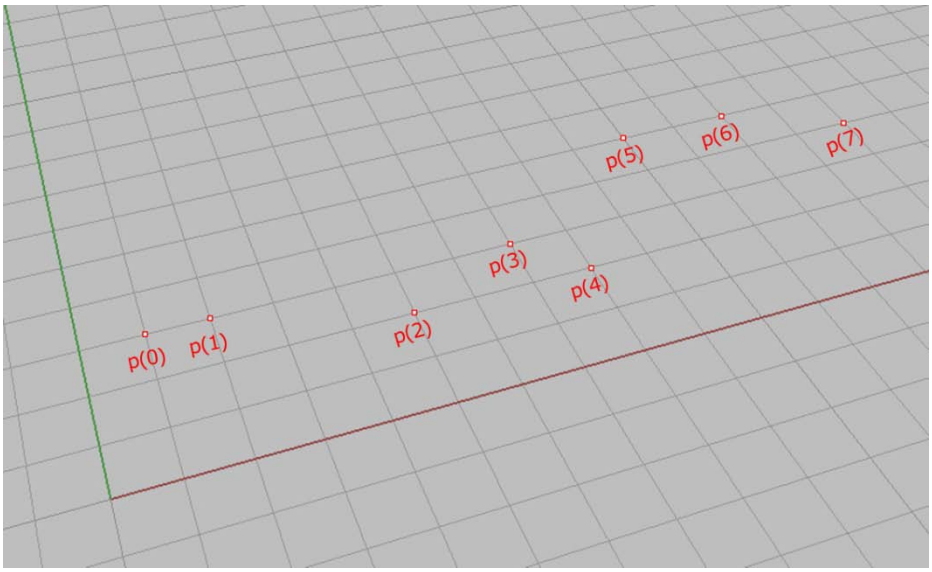


*Fig.5.5. Index number in a group of objects is a simple way to call one. This is a **0-based counting system** which means numbers start from 0.*

So as mentioned before, in Associative modelling we generate our geometries step by step as some related objects and for this reason we go into the parameter space of each object and extract specific information of it and use it as the base data for the next steps. This could be started from a simple field of points as basic generators and ends up at tiny details of the resultant model, in different hierarchies.

5_4_Basic Parametric Components

5_4_1_Curve Evaluation

The <evaluate> component is the function that finds the point on a curve or surface, based on the parameter we feed. The <evaluate curve> component (Curve > Analysis > Evaluate curve) takes a curve and a parameter (a number) and gives back a point on curve on that parameter.

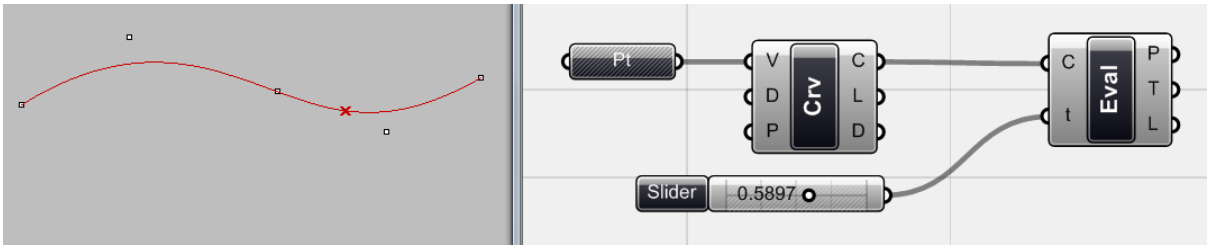


Fig.5.6. The evaluated point on <curve> on specific parameter which comes from the <number slider>.

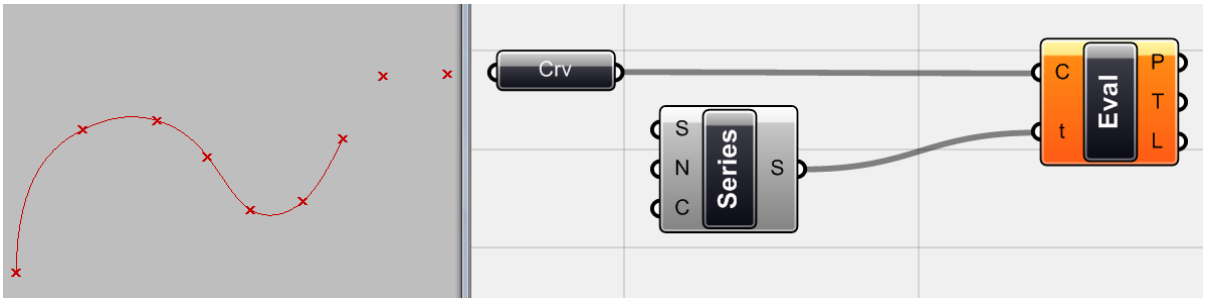


Fig.5.7. We can use any <curve> that drawn in Rhino or in Grasshopper to <evaluate>. And you see that we can use <series> of numbers as parameters to <evaluate> instead of one parameter. In the above example, because some numbers of the <series> component are bigger than the domain of the curve, you see that <Evaluate> component gives me warning (becomes orange) and that points are located on the imaginary continuation of the curve.

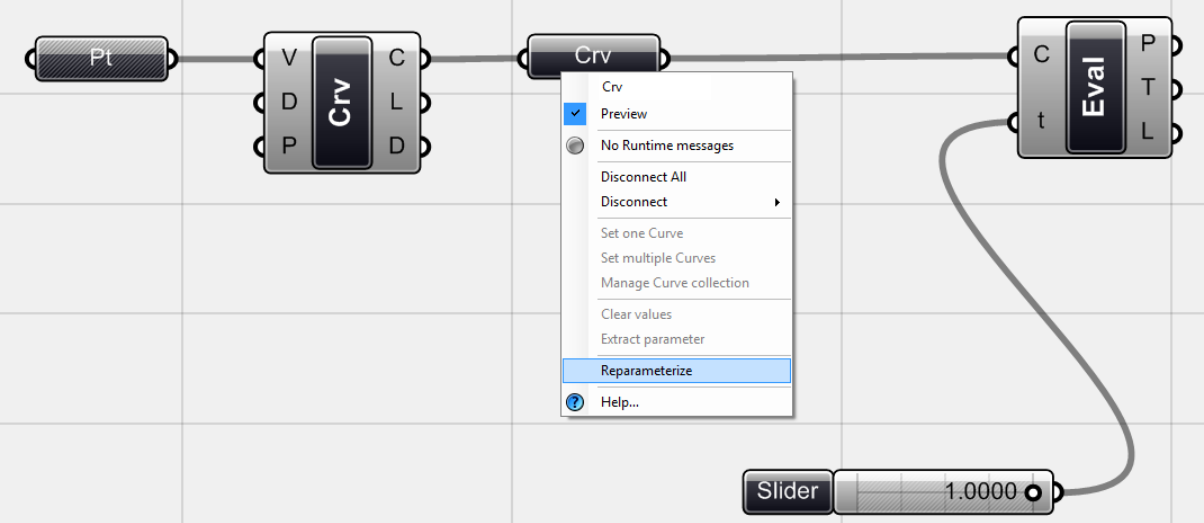


Fig.5.8. Although the 'D' output of the <curve> component gives us the domain of curve (minimum and maximum parameters of the curve), alternatively we can feed an external <curve> component from Param > Geometry and in its context menu, check the ***Reparameterize*** *option. It sets the domain of the curve from 0 to 1. So basically I can track all <curve> long by a <number slider> or any numerical set between 0 and 1 and not be worry that parameters might go beyond the numerical domain of the curve.*

There are other useful components for parameter space on curves in Curves > Analysis / Division that we would talk about them later.

5_4_2_Surface Evaluation

While for evaluating a curve we need a number as parameter (because curve has a 1D-space) for surfaces we need a pair of numbers as parameters (U, V), with them, we can evaluate a specific point on a surface. We use <evaluate surface> component (Surface > Analysis > Analysis) to evaluate a point on a surface on specific parameter.

We can simply use <point> components to evaluate a surface by using it as UV input of the <Evaluate surface> (it ignores Z dimension) and you can track your points on the surface just by X and Y parts of the <point> as U and V parameters.

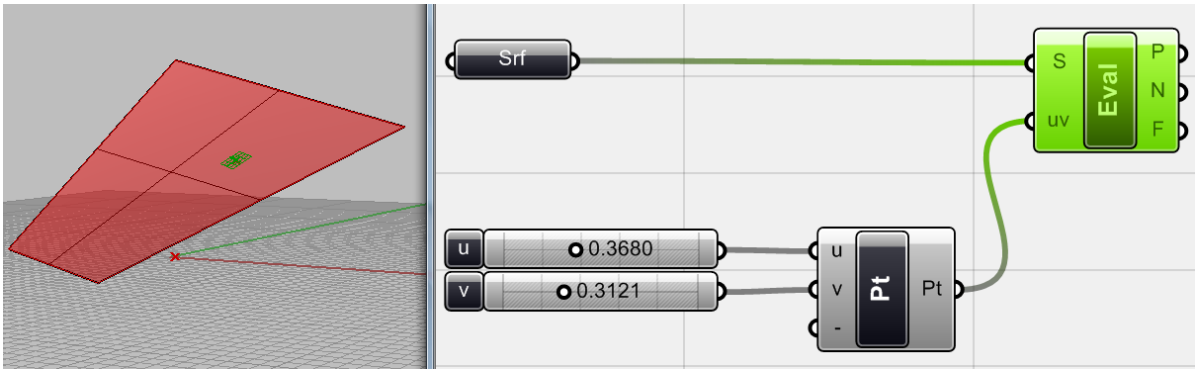


Fig.5.9. A point <Evaluate>d on the <surface> based on the U,V parameters coming from the <number slider> with a <point> component that make them a pair of Numbers. Again like curves you can check the 'Reparameterize' on the context menu of the <surface> and set the domain of the surface 0 to 1 in both U and V directions. Change the U and V by <number slider> and see how this <evaluate>d point moves on the surface (I renamed the X,Y,Z inputs of the component to U,V,- manually).

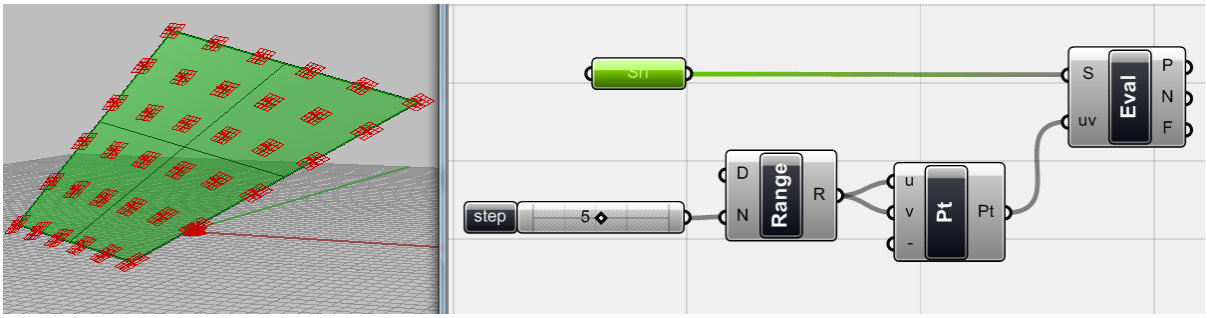


Fig.5.10. Since we need <points> to evaluate a <surface> as you see we can use any method that we used to generate points to evaluate on <surface> and our options are not limited just to a pair of parameters coming from <number slider>, and we can track a surface with so many different ways.

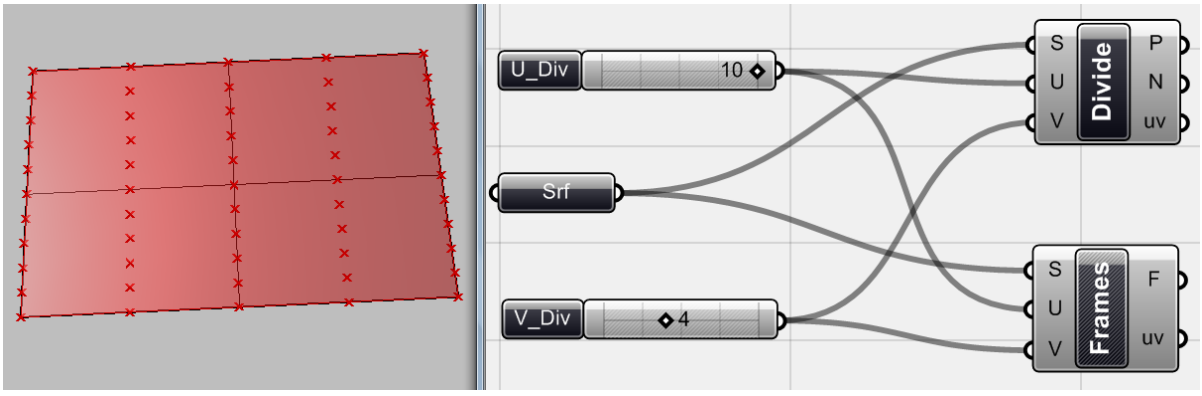


Fig.5.11. To divide a surface (like the above example) in certain rows and columns we can use <Divide surface> or if we need some planes across certain rows and columns of a surface we can use <surface frame> both from Surface set under Util paenl.

5_4_3_Curve and Surface Closest Point

We always don't have the parameter to look for points, some times we have the point and we want to know its parameter for further uses. This is when finding closest point comes to play. <Curve CP> and <surface CP> components (curve/surface closest point) are two components that help us to do that.

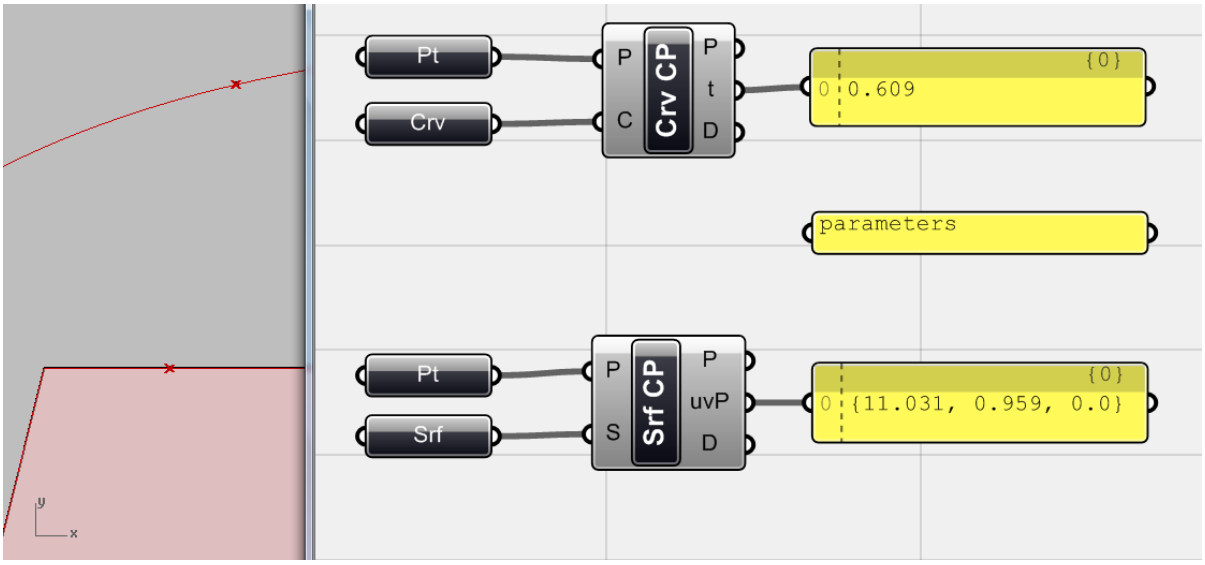


Fig.5.12. <Curve Cp> and <Surface CP> help us to find the parameter of a point on a curve or surface. There are other components that you need to feed them with these parameters.

5_5_On Object Proliferation in Parametric Space

For so many design reasons, designers use surfaces to proliferate some other geometries on them. Surfaces are flexible, continues two dimensional objects that represent acceptable bases for this purpose. There are multiple methods to deal with surfaces like Penalisation, but here I am going to start with one of the simplest one and we will discuss about some other methods later.

We have a free-form surface and a simple geometry like a box. The question is, how we could proliferate this box over the surface, in order to have a differentiated surface i.e. as an envelope, in that we have control of the macro scale (surface) and micro scale (box) of the design separately, but in an associative way.

The method is like this: In order to accomplish the task, we should divide the surface into desired parts and generate our boxes on these specific locations on the surface and re-adjust them if we want to have local manipulation of these objects.

Generating the desired locations on the surface is easy. We can divide surface or we can generate some points based on any numerical data set that we want.

About the local manipulation of proliferated geometries, again we need some numerical data sets which could be used for transformations like rotation, local displacement, resize, adjustment, etc.

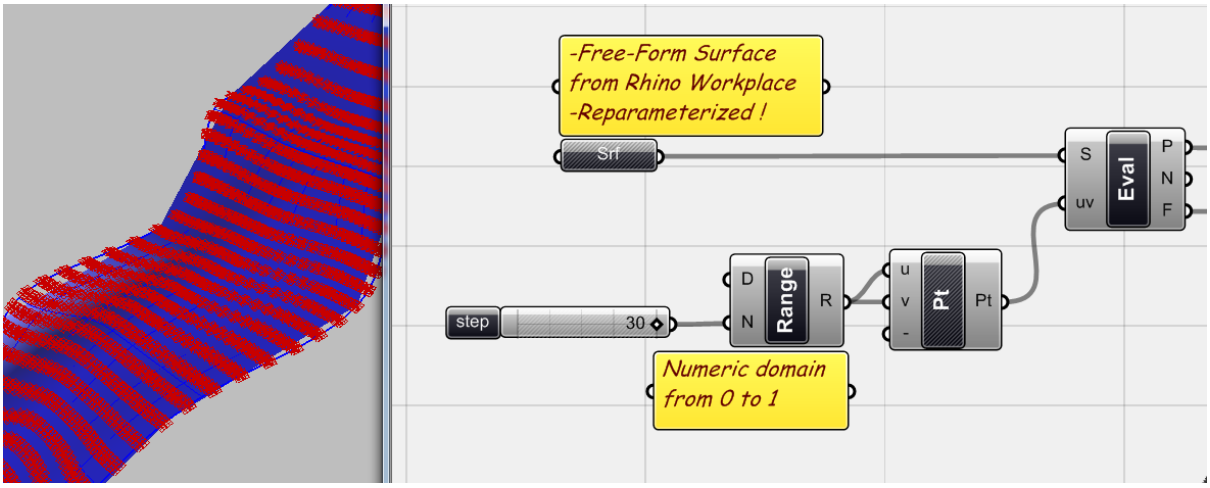


Fig.5.13. A free-form, reparameterized, <surface> being <evaluate>d by a numeric <range> from 0 to 1, divided by 30 steps by <number slider> in both U and V direction. (Here you can use <divide surface> but I used the <point> component to remind you all point-generation techniques from chapter two are possible options to insert into these experiment).

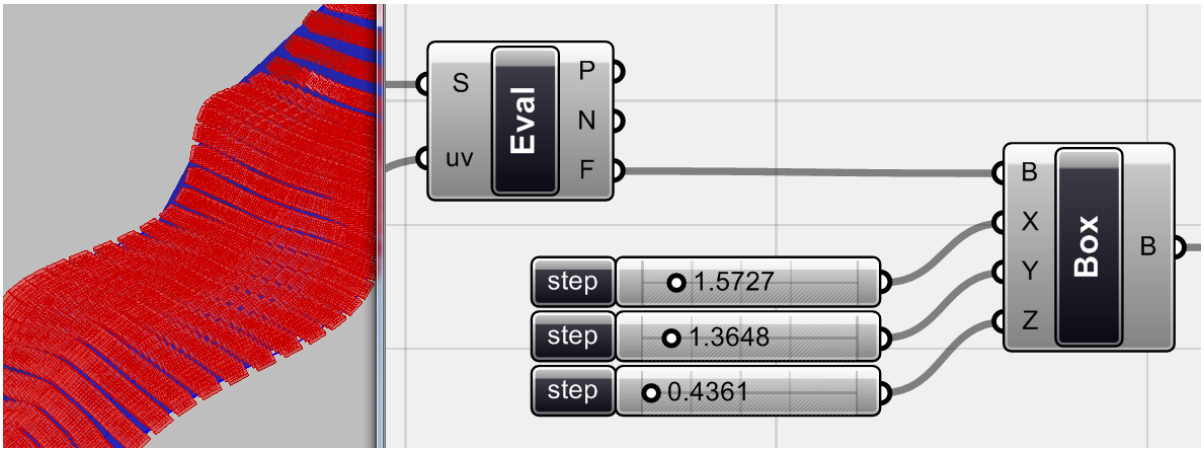


Fig.5.14. As you see the <evaluate> component gives 'normal' and 'plane' of any evaluated points on the surface. I used these planes or frames to generate series of <box>es on them while their sizes are being controlled by <number slider>s. the <box> component (surface>primitive> center box) needs center of the box and its length in X,Y and Z directions

In order to manipulate boxes locally, I just decided to rotate them. I want to set the rotation axis parallel to the U direction of the surface and based on the situation of this simple surface I am going to choose the XZ plane as the base plane for their rotation (Fig.5.15).

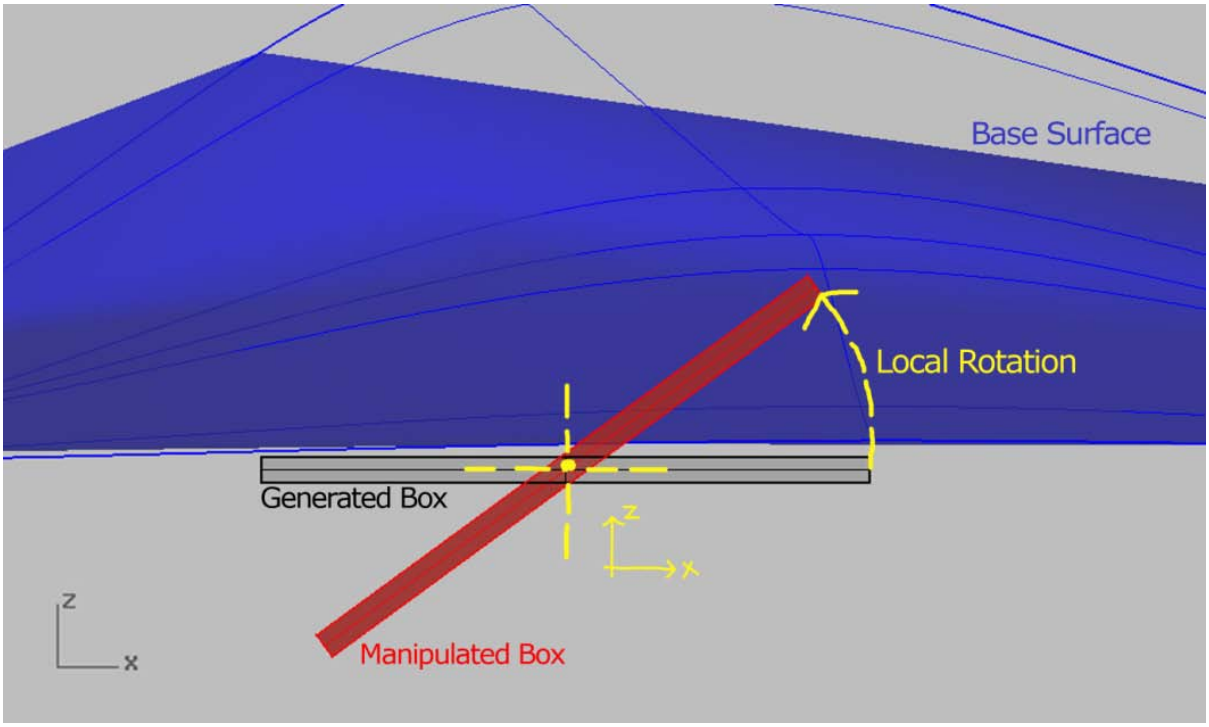


Fig.5.15. Local rotation of box.

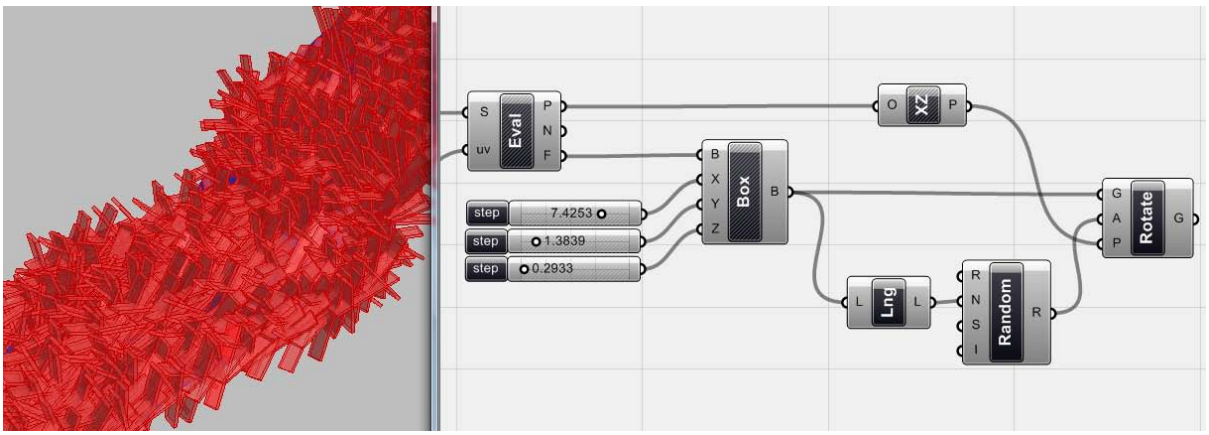


Fig.5.16. The <rotate> component needs three inputs. First is the geometry which means <box>es. The second item is rotation angle. I want to rotate them by random values (you can rotate them gradually or any other way) so I want to generate a set of <random> numbers and I set the Number of random values as much as boxes. So I just used a <list length> component to realize how many <box>es I have and attached it to the 'N' input of the <random> and attached the random values as angles of rotation to the <rotate> component. Finally to define the plane of axis, I generated <XZ plane>s on any point that I <evaluate>d on the <surface> and I attached it to the <rotate> component.

Don't forget to uncheck the Preview of the previously generated objects to enhance the performance of the process.

Fig.5.17. Final geometry.

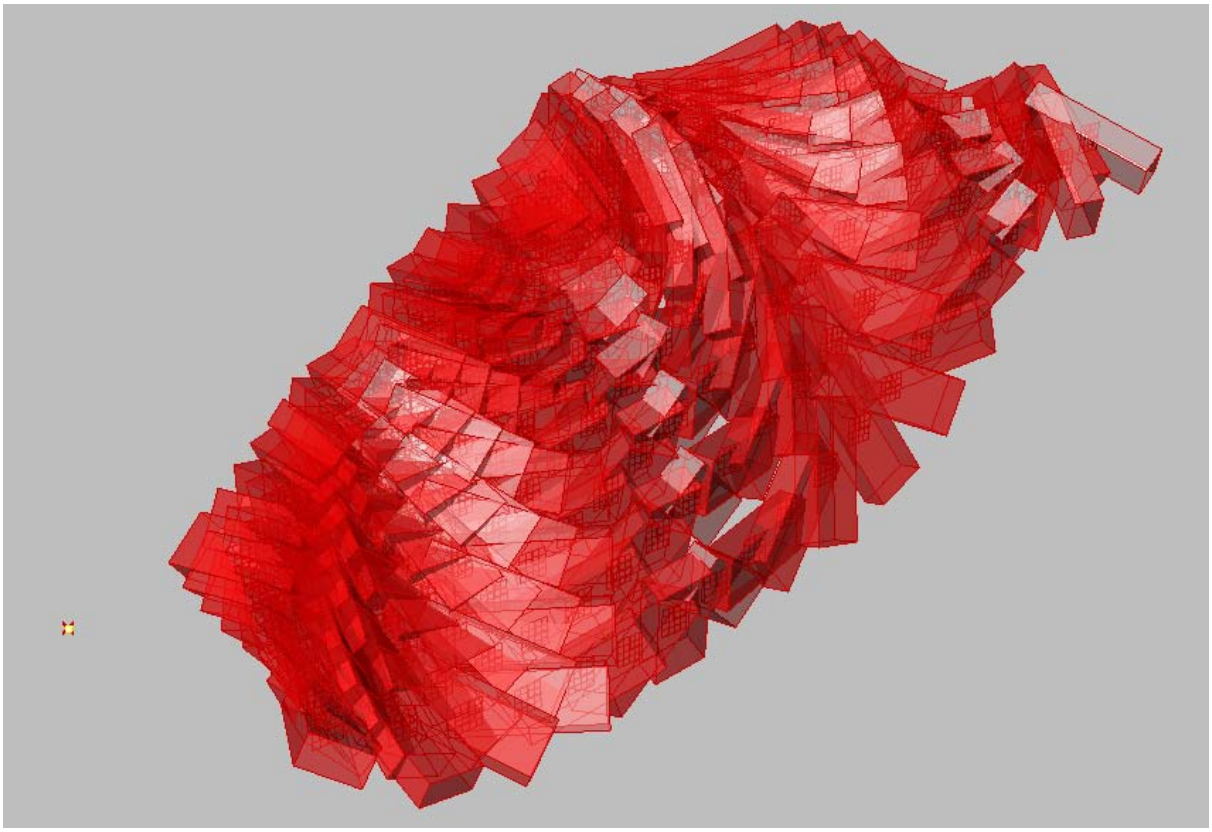


Fig.5.18. Try to combine different concepts in your projects. Here instead of random values for rotation of boxes, I used a point attractor and set its distance from each box as the rotation factor and as you see, new results are shown in the experiment. These are techniques for local manipulation of the boxes, but you know that you could apply changes to the global scale as well.

Non-uniform use of evaluation

During a project this idea came to my mind that why should I always use the uniform distribution of points over a surface and add components to it? Can I set some criteria and evaluate my surface based on that and select specific positions on the surface? Or since we use U,V parameter space and incremental data sets (or incremental loops in scripting) are we always limited to a rectangular division on surfaces?

There are couple of questions regarding the parametric tracking of a surface but here I am going to deal with a simple example to show how in specific situations we can use some of the U,V parameters of a surface and not a uniform rectangular grid over it.

Columns Example

I have two Free-form surfaces as covers for a space and I am thinking of creating a social open space in between. I want to add some columns between these surfaces but because they are free-form surfaces and I don't want to make a grid of columns, I decided to limit the column's length and add as many places as possible in certain positions with height limit. I want to add two inverted and intersected cones as columns in this space, just to make a simple shape.

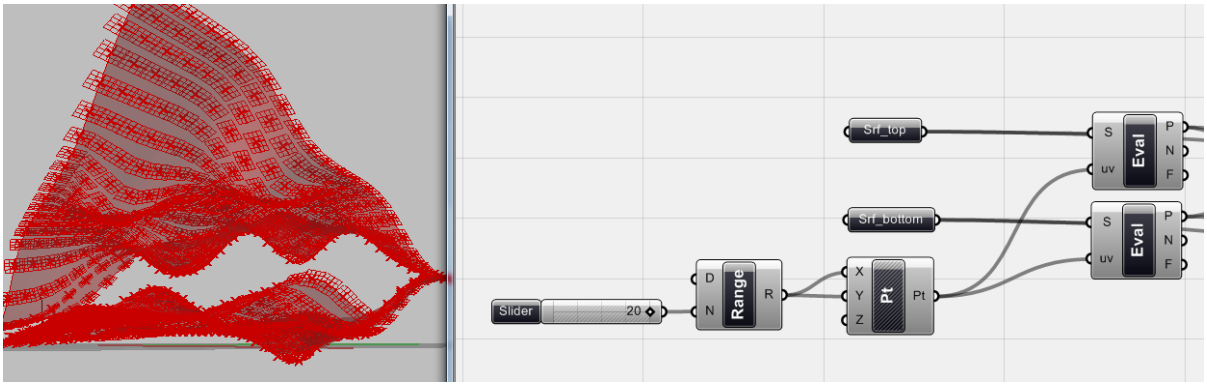


Fig.5.19. I introduced two general surfaces to Grasshopper by <srf_top> and <srf_bottom> as space covers and I Reparameterized them. I also generated a numerical <range> between 0 and 1, divided by <number slider>, and by using a <point> component I <evaluate>d these surfaces at that <points>.

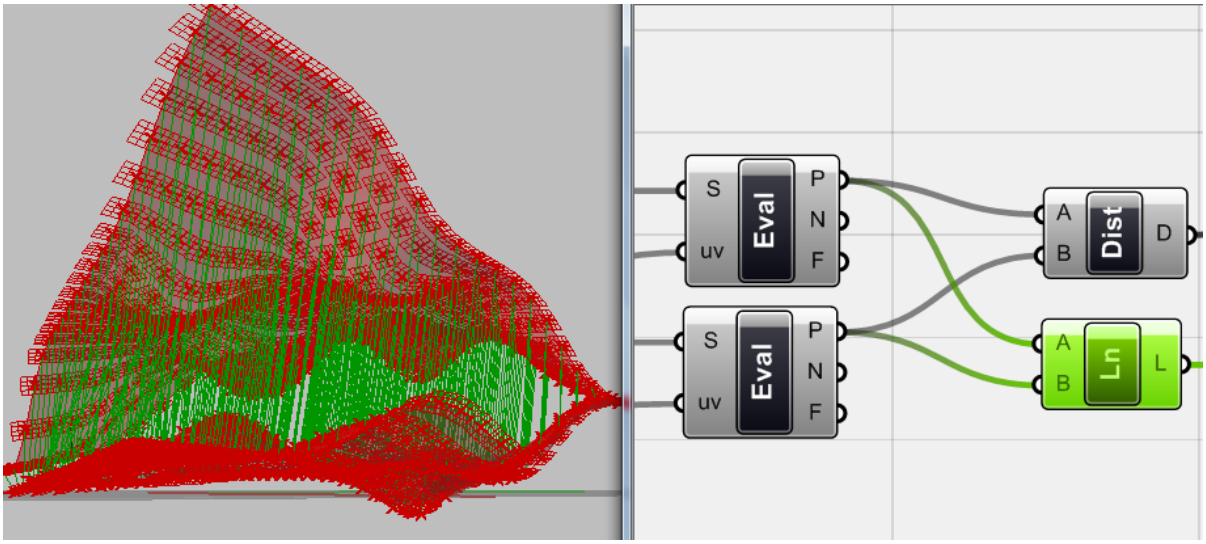


Fig.5.20. Next, I generated bunch of <line>s between all these points, but I also measured the distance between any pair of points.

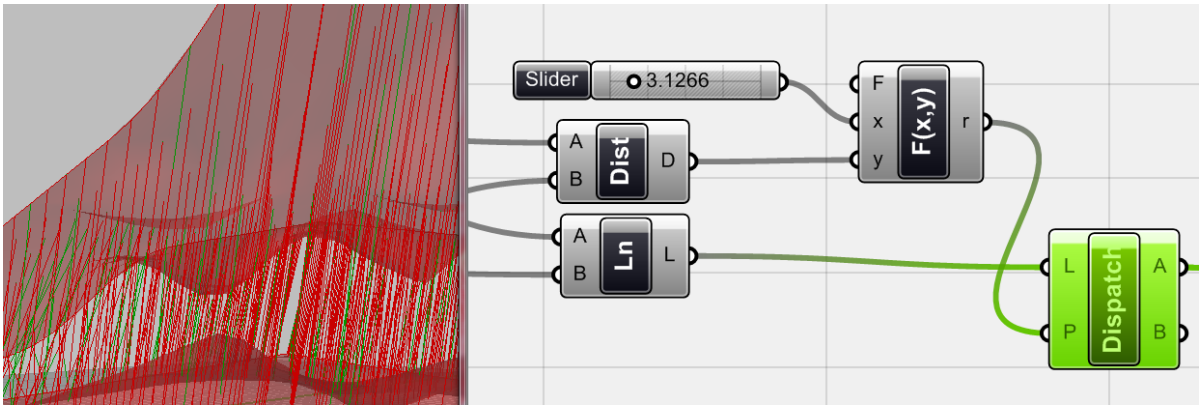


Fig.5.21. Now I need to extract my desired lines from the list. Here I used a <dispatch> component (Logic >Llist > Dispatch) to select my lines from the list. A <dispatch> component needs Boolean data which is associated with the data from the list, to send those who associated with True to the A output and those associated with False, to the B output. The Boolean data comes from a simple comparison function. In this <function> I compared the line length with a given number as maximum length of line (x>y, x=<number slider>, y=<distance>). Any line length less than the <number slider> creates a True value by the function and passes it through the <dispatch> component to the A output. So if I use lines coming out the <dispatch> A output I am sure that they are all smaller than the certain length, so they are my columns.

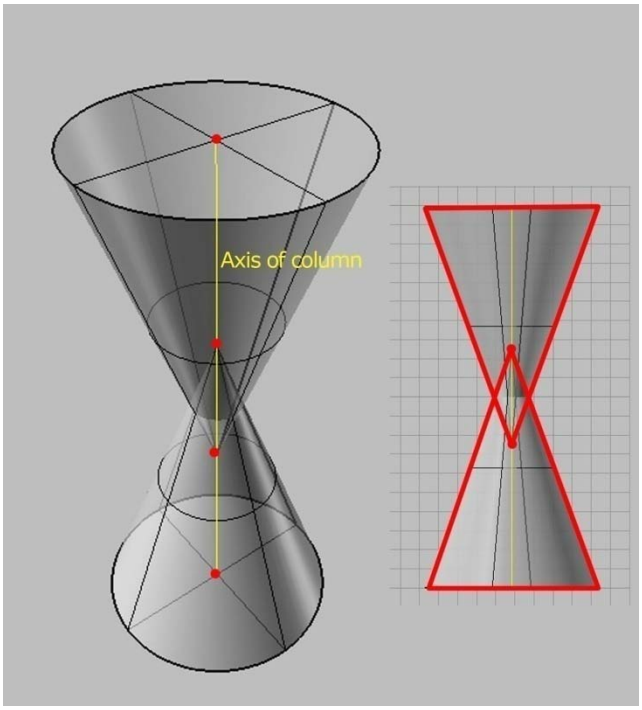


Fig.5.22. The geometry of columns is two inverted cones which are intersecting at their tips. Here because I have the axis of the column, I want to draw two circles at the end points of the axis and then extrude them to the points on the curve which make this intersection possible.

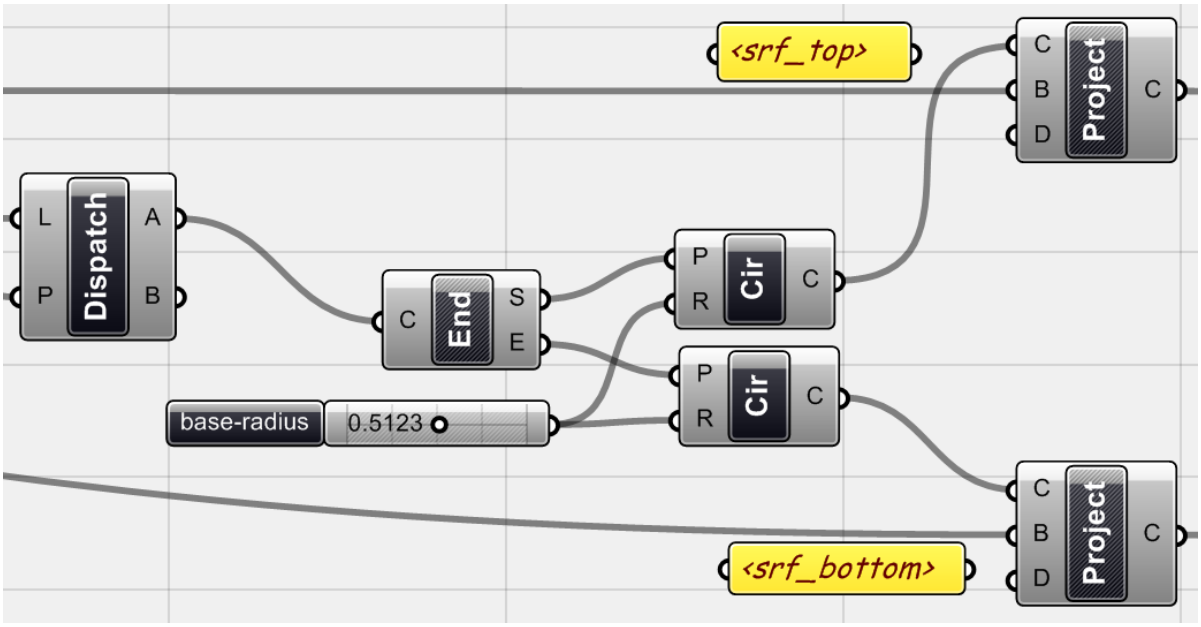


Fig.5.23. By using an <end points> component I can get both ends of columns. So I attached these points as base points to make <circle>s with given radios. You already know that these circles are flat but our surfaces are not flat. So I need to <project> my circles on main surfaces to find their adjusted shape. I used a <project> component (Curve > Util > Project) for this reason. B part of the <project>s connected to the top and bottom surface.

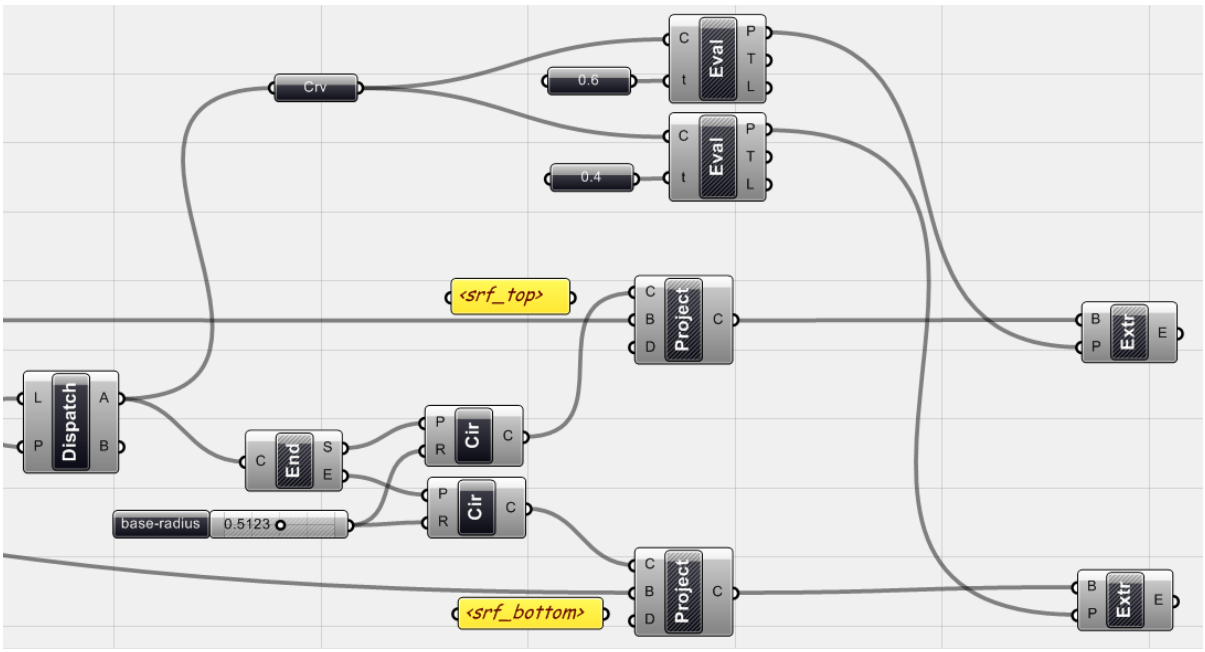


Fig.5.24. The final step is to extrude these projected circles towards the specified points on column's axis (Fig.5.23). I used <extrude point> component and then I attached <project>ed circles as base curves. For the extrusion point, I attached all columns' axis to a <curve> component and I 'Reparameterized' them, then I <evaluate>d them in two specific parameter of 0.6 for top cones and 0.4 for bottom cones.

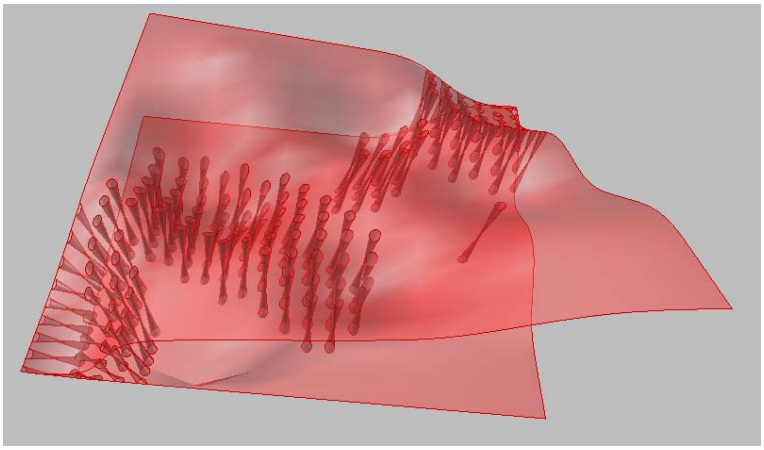


Fig.5.25. Although in this example, again I used the grid based tracking of a surface, I used additional criteria to choose some of points and not all of them uniformly.

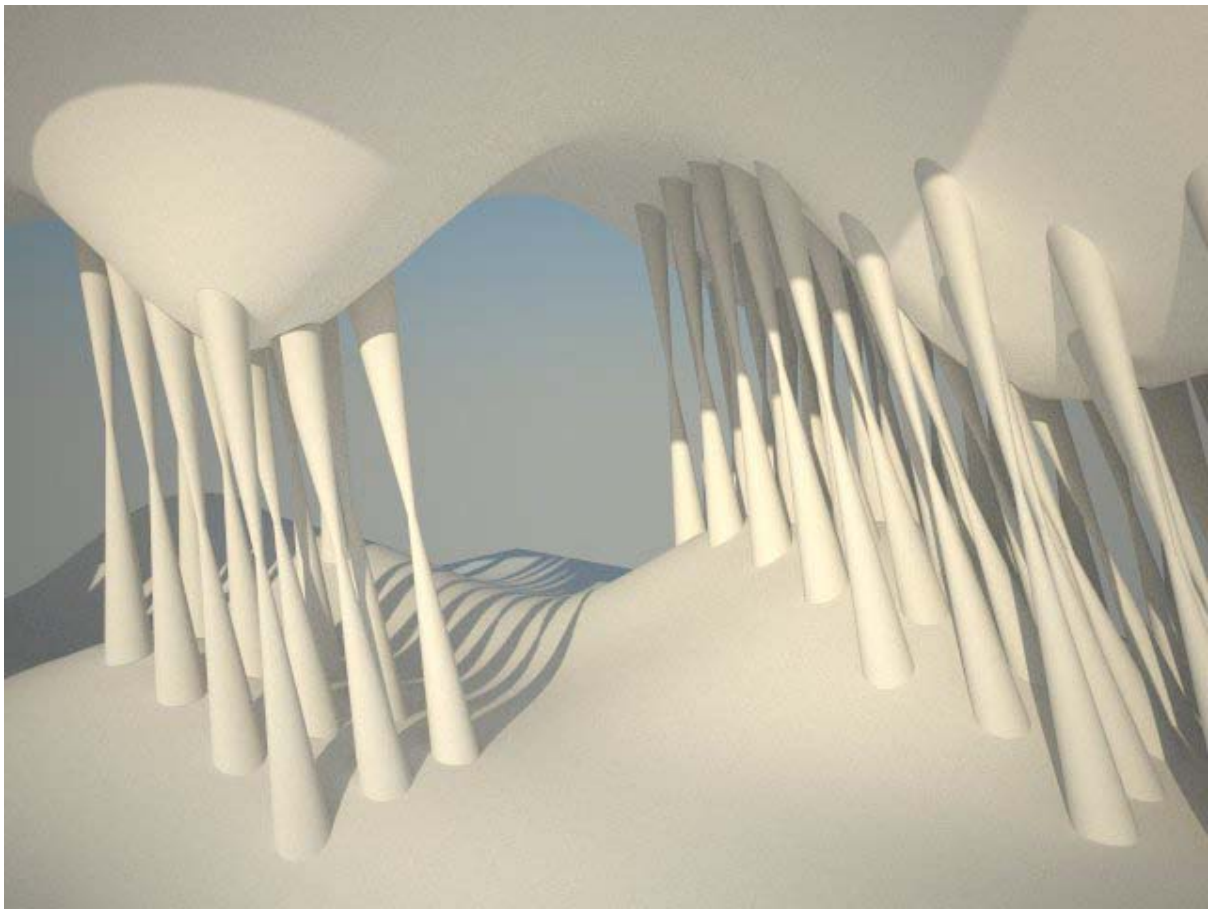


Fig.5.26. Final model.

The idea of using Parameter space of curves and surfaces to proliferate objects on them has so many options and methods. Don't stick to one of them and try to explore more. We will see a bit more.

5_6_On Data Trees

Talking about parametric space and using related components, now it is time to open up a new subject about data management in Grasshopper called 'Data Tree', which little by little you might need it when working with complex models, especially in parametric space of curves and surfaces.

One of the great potentials of generative algorithms is that they enabled us to design and manage hundreds of objects together associatively. Working with huge amount of objects, sometimes we need to apply commands to all of them and sometimes we need to extract one item and apply a command to it. So we need to have access to our objects and manage our data (objects) in different ways.

Imagine we have 5 curves in our design space and we divided them into 10 parts. Now we want to extract all second points of these curves and connect them together with a new interpolated curve.

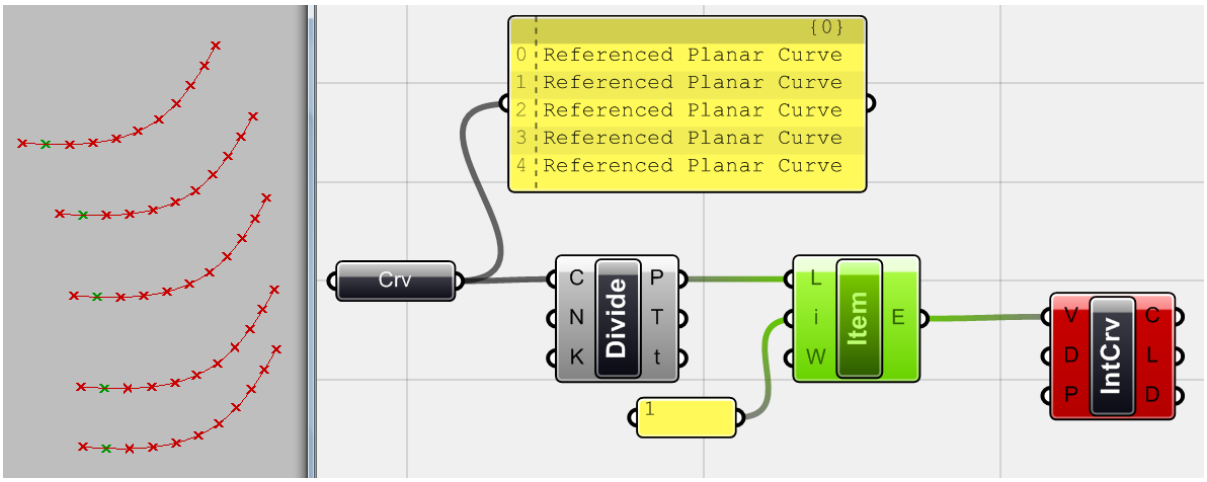


Fig.5.27. A <curve> component with 5 curves, all are <Divide>d by 10. If you select index number 1 with <item> component from division points, you see that all second points of curves are being selected, not just the second point of the first line. Great!. But if you attach these points to an <interpolate> to draw a curve, you see the <interpolate> shows error and does not draw anything!

Here to understand the problem lets introduce a useful component and observe the situation. This component is <Param Viewer> from Params>Special. Let's compare the result:

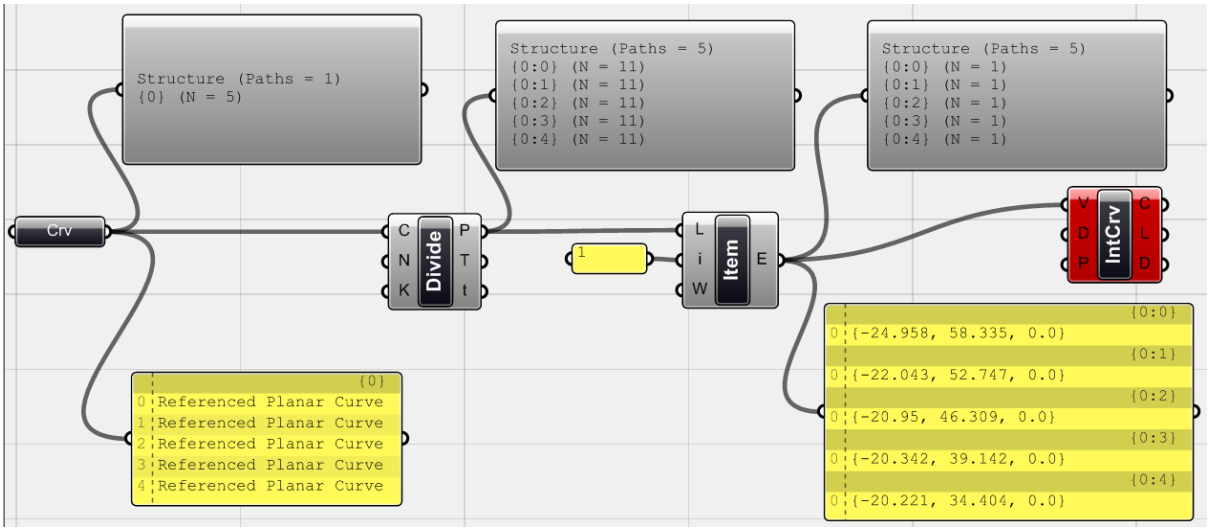


Fig.5.28. The <Param Viewer> component shows some other data information inside components which is the reason of the error in <interpolate>.

What you see in <Param Viewer> is the concept of Data Tree in Grasshopper. As you can see in the Figure.5.28 the <curve> component has 5 items but when these curves <divide>d and generated some points for each curve, points has been sorted into different data lists called Branches. This means that the result of the <divide> component is not just one list of data comprised of 55 points, but now we have five lists of data each has 11 points inside. So the main data of the 'Tree' has been divided into 'Branches' in order to facilitate further usage and easier access to them in our design. That's why when we select <item> index 1, it selects items with index 1 in each list.

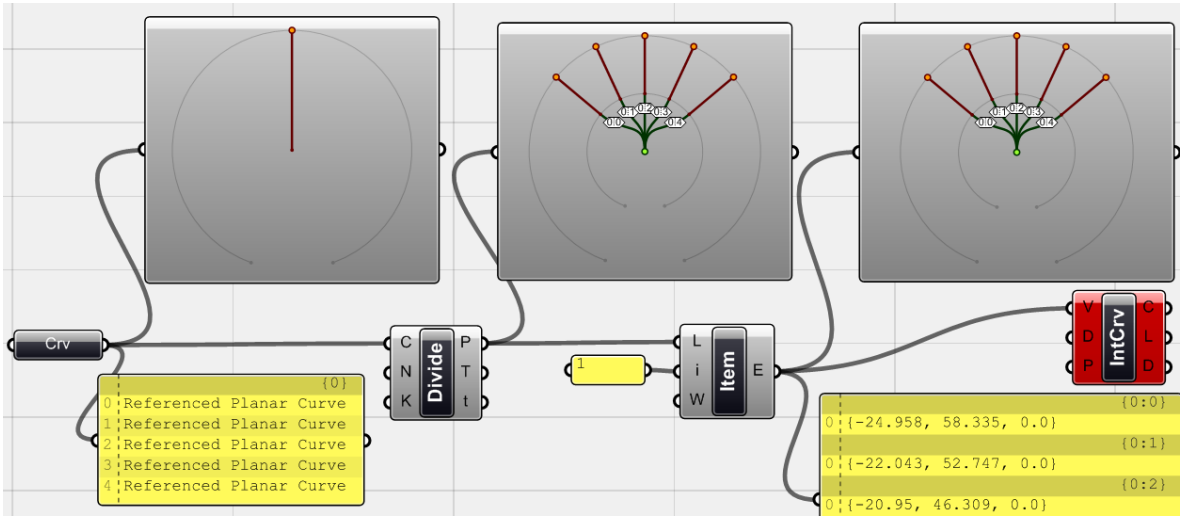


Fig.5.29. <Param Viewer> with 'Draw Tree' option checked in their context menu to show the difference between data branches in each component.

Now, why the <interpolate> component cannot draw a curve? What is the problem? Let's have a closer look at the information we gathered from our situation:

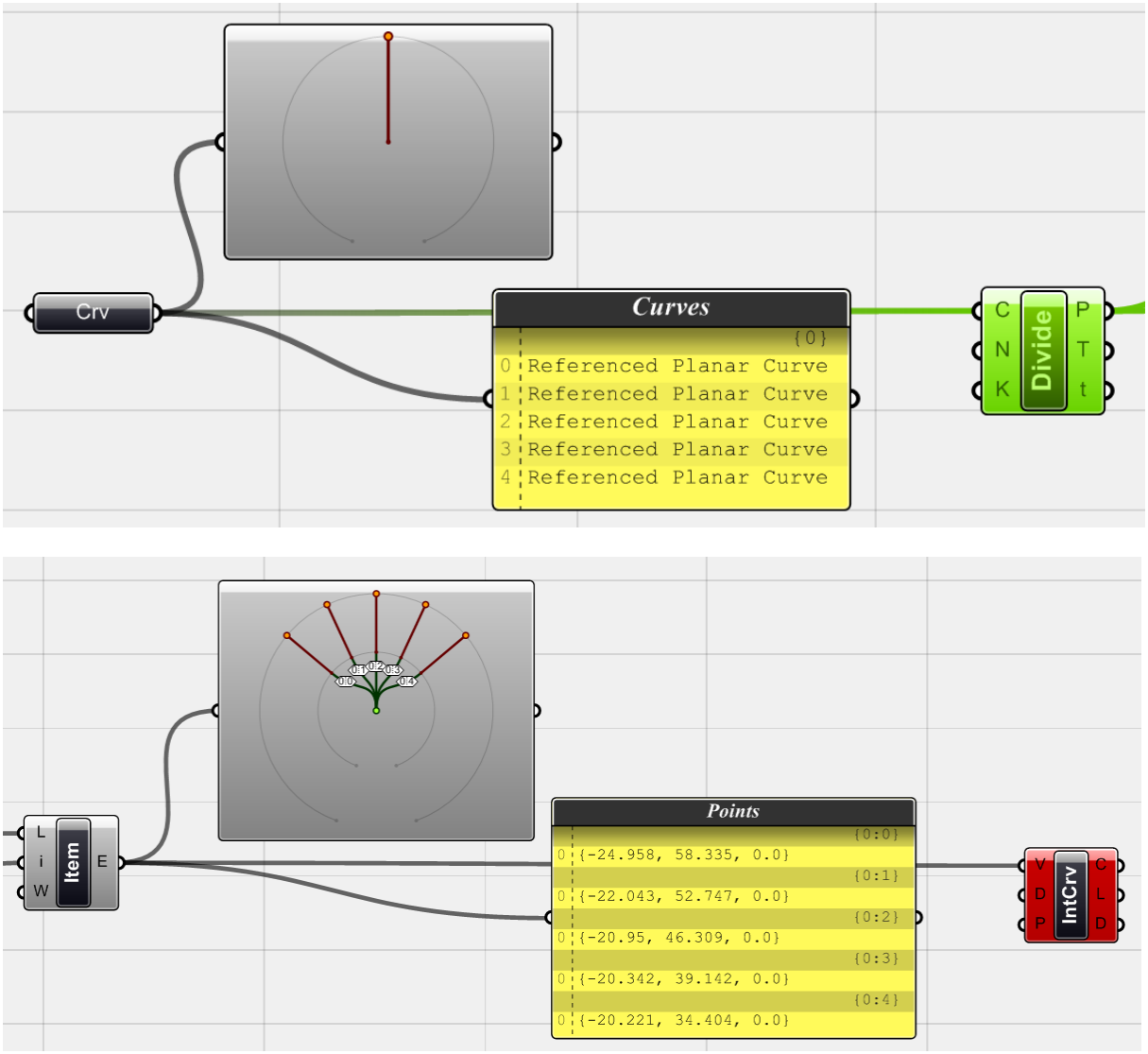
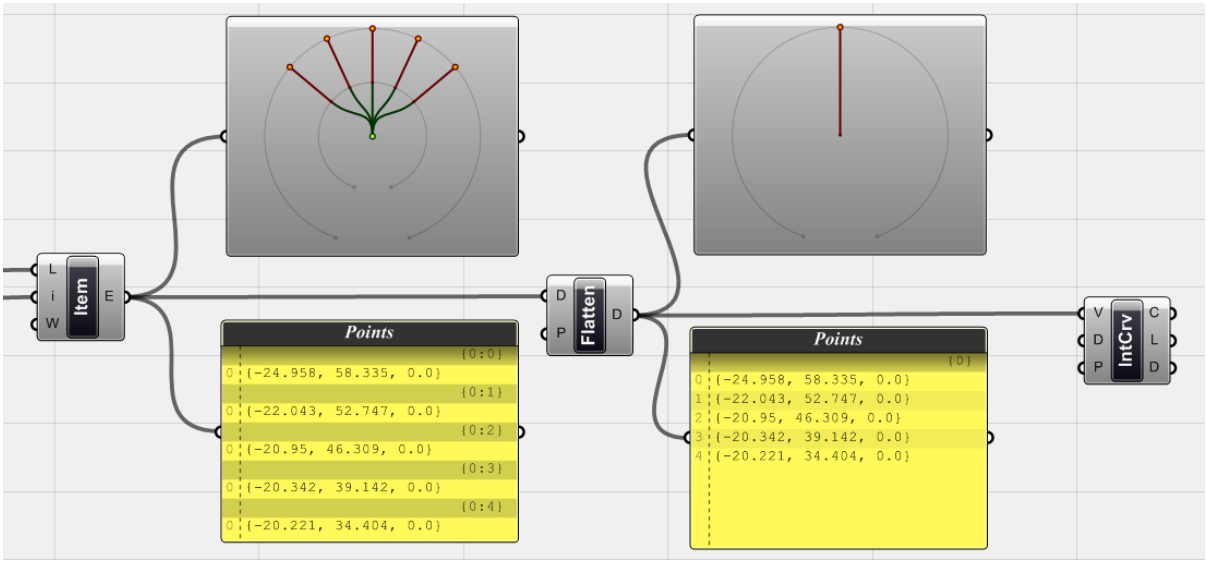


Fig.5.30. <Param Viewer> and <Panel> of A:curves and B:selected index 1 points.

In the first image of the Figure.5.30 the <Param viewer> of the <curve> component shows one main branch of data. If you look at its <Panel>, you see there is a list of curves under the title of { 0 }. Here in Grasshopper { } is the symbol for data tree and shows the branch that the object is positioned in. So all curves in the first image under the { 0 } are situated in the main branch. If you go back to the Figure.5.28 you see that for the <curve>'s <param viewer>, it says (Paths = 1) which means we have only one branch of data and in this branch { 0 } (N = 5) we have 5 items.

But in the second image of the Figure.5.30 we can see that the data in the <item> component listed under five different branches: { 0:0 }, { 0:1}, … { 0:4 } and there is one point at each list. If you again check the Figure.5.28 you see that the third <param viewer> has 5 branches (paths = 5) and each branch of data has one item (N = 1 for all branches). This means that the data has been divided into different branches and when transferred into <interpolate> component, they are separate from each other and <interpolate> component cannot draw polyline by five separated points.

How can we solve this problem?



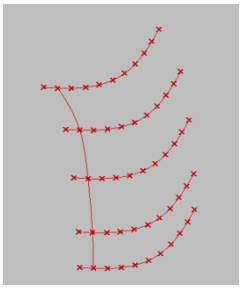


Fig.5.31. To solve this problem I used a <Flatten> component (Logic>Tree) which as it sounds, converts data from multiple branches to one branch which is visible in <param viewer>. As you can see in the second <panel>, now we have five points under the branch of { 0 } and <interpolate> component can draw a polyline by these five points.

To summarize and get the concept, we should understand that while working with multiple objects in different levels, data has hierarchical structure in branches and each branch of data has its own path as an index number which shows the unique address of that branch(i.e. { 0:1}). It is important to know that working with list management components is affected by this concept and these components work on each branch as a separate list. We might need to generate branches of data by ourselves or converge branched data into one branch, or other types of data management that I try to use them in later experiments.

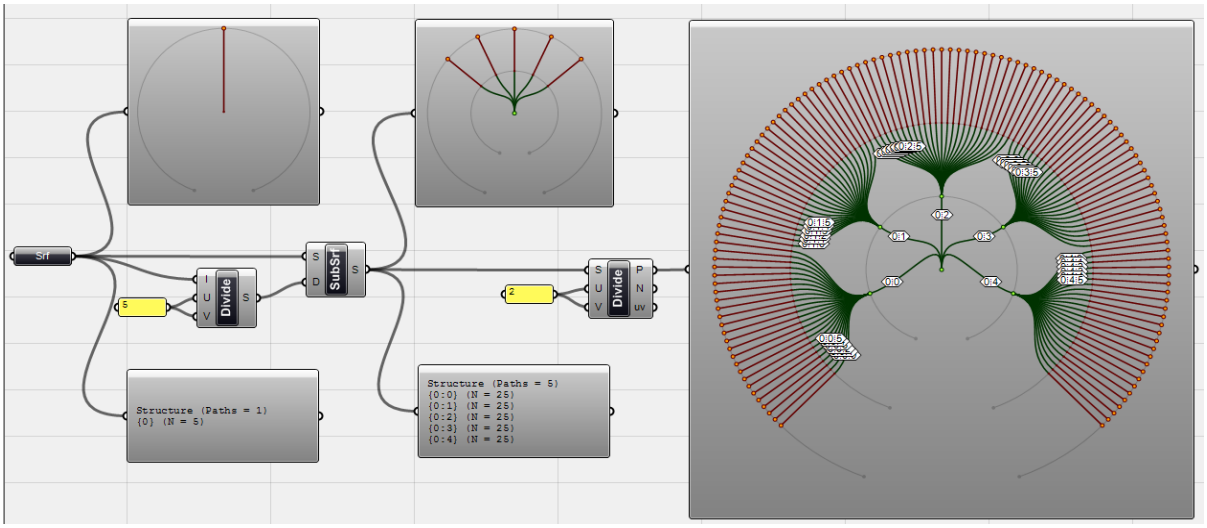


Fig.5.32. Data Branches on a 'DATA Tree'.

Let's have another example to wrap up the subject:

I want to design a porous surface like what I sketched in Figure.5.33 based on one given surface. I am going to describe the process a bit fast to see the final bit.

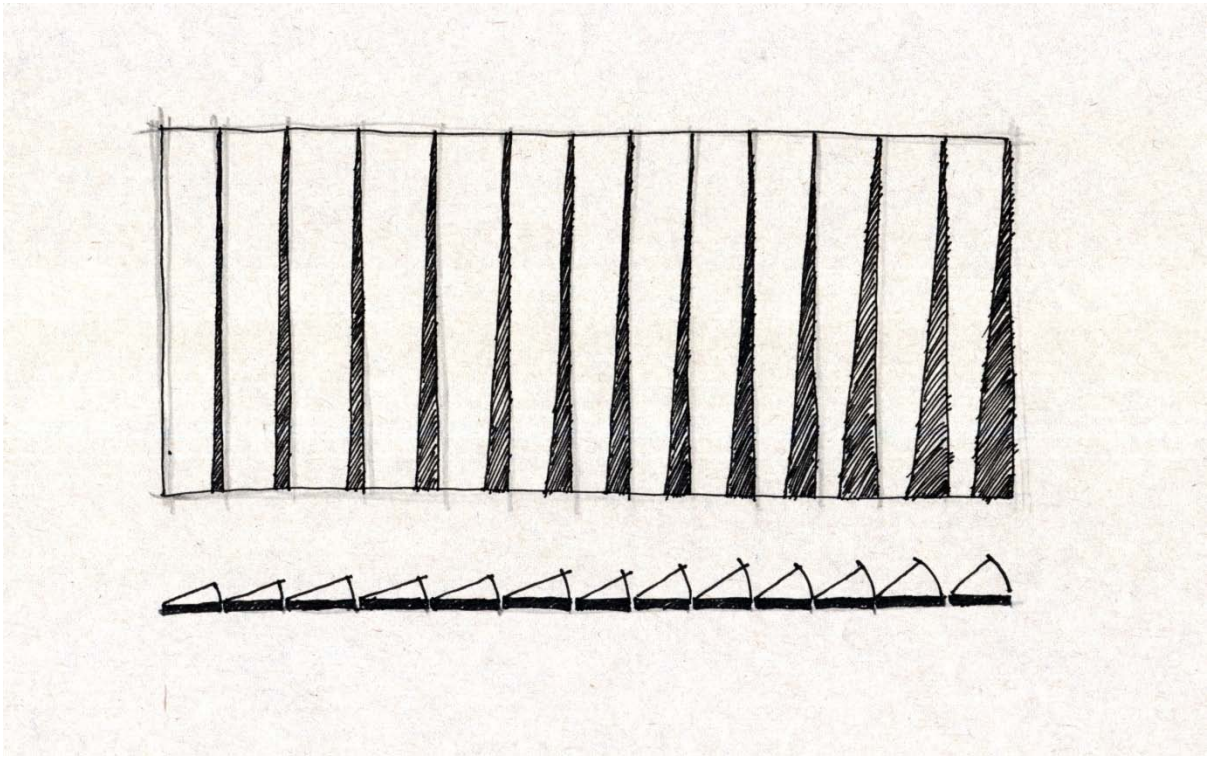


Fig.5.33. Sketch of the desired surface.

To design this porous surface, I want to generate couple of small lines in top and bottom edges of a surface to loft them together. This would generate some small surfaces which all together make the general look of my porous surface. I should take care of direction of my small lines to be able to control the gradual differentiation of surfaces or let's say the general porosity.

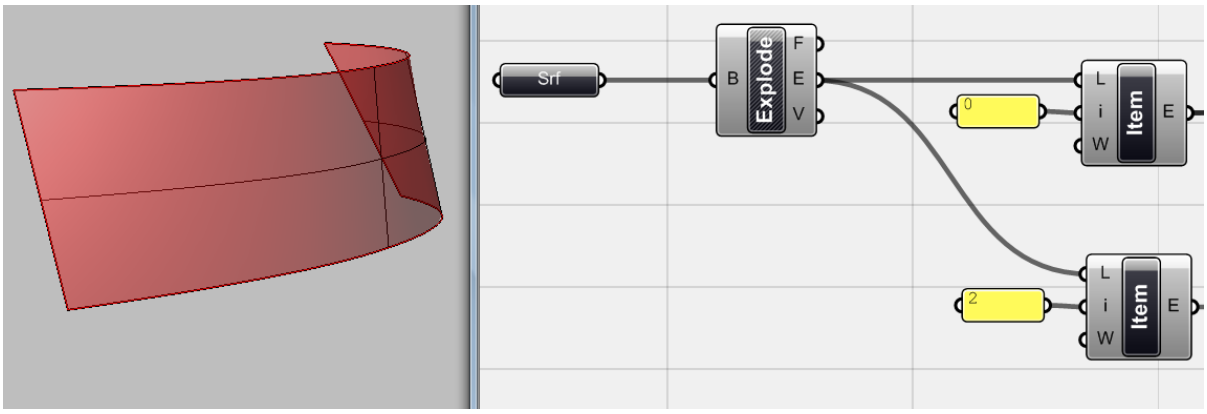


Fig.5.34. I introduced a generic <surface> into canvas and I exploded it by <BRep Components> (Surface>Analysis) to have access to its edges. Then I selected the bottom and top edge with <item> by index 0 and 2.

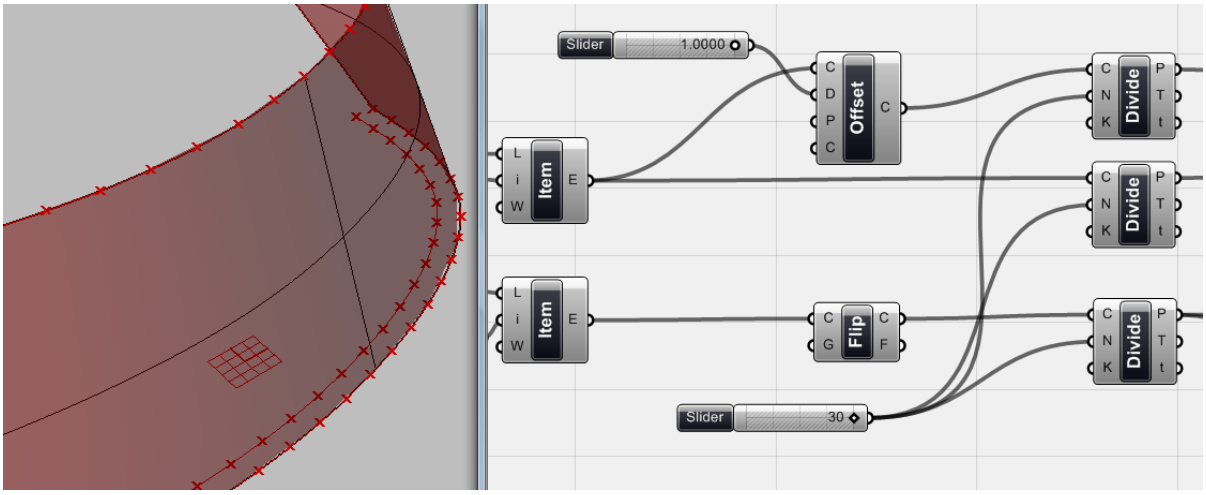


Fig.5.35. I used the bottom edge to <offset> (Curve>Util), and I also changed the direction of the top edge with <Flip> (Curve>Util) because I know that the bottom and top edge curves are not in the same direction. I used <divide> component to divide these edge curves and have multiple points. (We can use one <divide> component but I don't like to make it complex now).

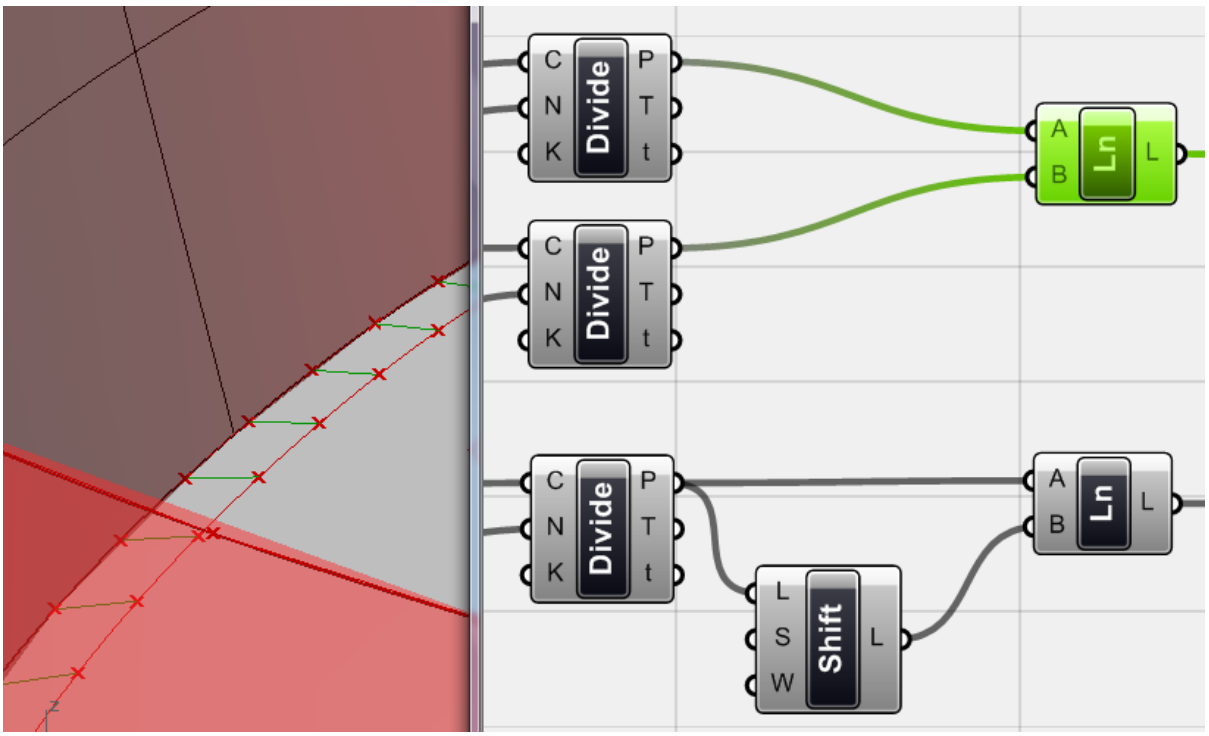


Fig.5.36. A <line> component is used to connect all bottom division points to all bottom-offset division points. Another <line> is used to connect all top-edge division points to their next points in the list (shift offset = 1).

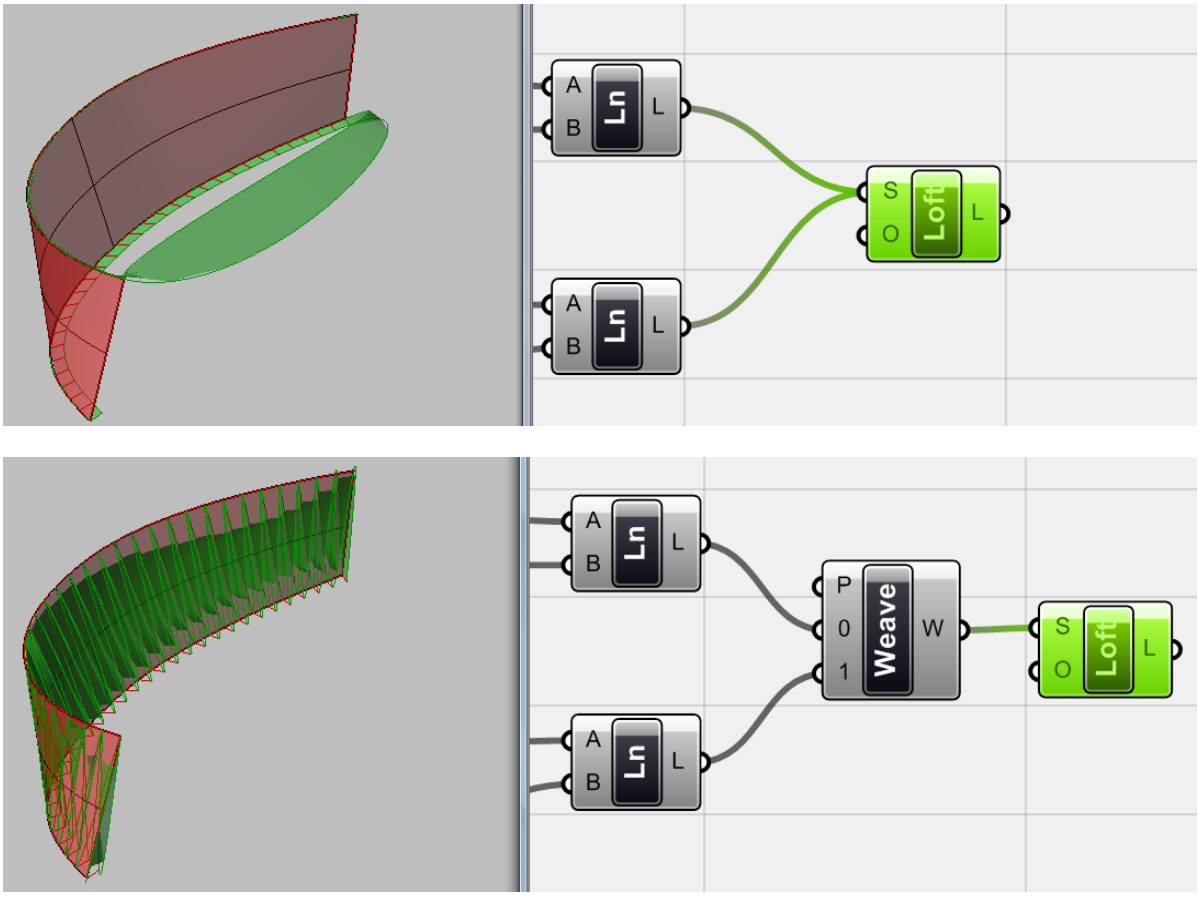


Fig.5.37. now if I use these <line> components to <loft>, you see that a surface being generated which is not my design purpose, even using <weave> component does not help in this situation.

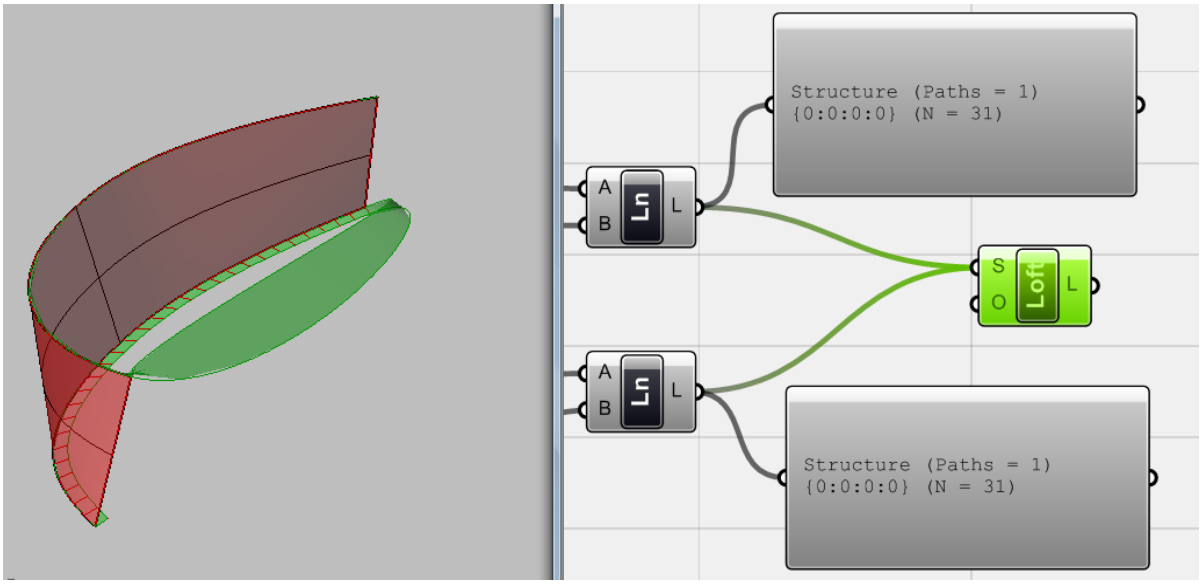


Fig.5.38. looking at <param viewer> of <line> components, we can see that both components have only one branch of data and when lofted, it lofts all of them together and not as separated pairs of lines. But here we want our lines to be in different data lists to be treated as single data and when lofted, they become pairs of lines.

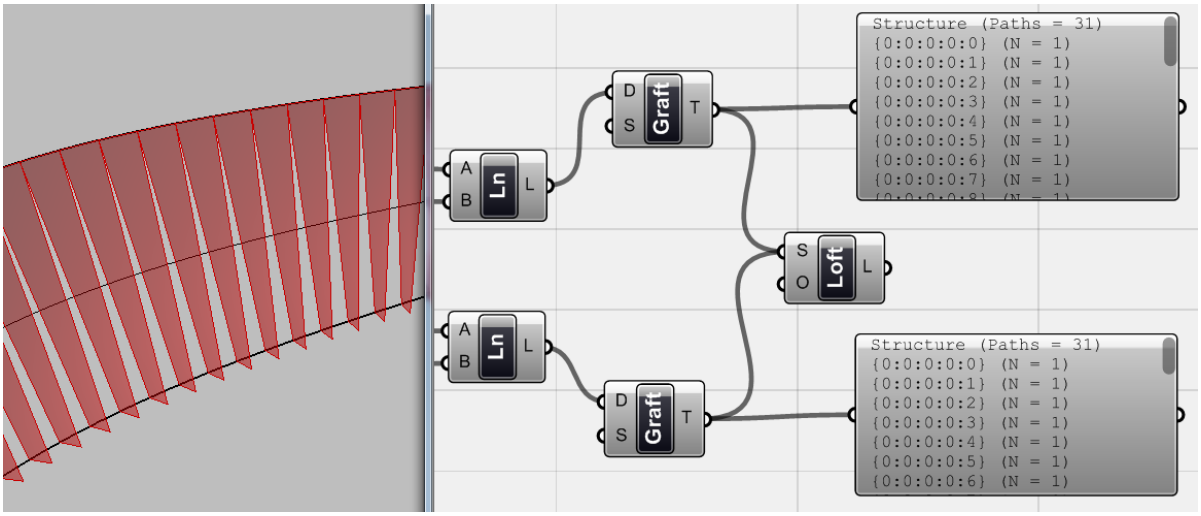


Fig.5.39. To solve this problem here I used a useful component which is <Graft> (Logic>Tree>Graft Tree). As you can see, as a result of this component, data lists in both <line> components have been divided into branches, one branch for each item in the list. Now <loft> component lofts each line from the first data list to an associated line in the second data list. And we can see that the resultant geometry is a porous surface as I sketched.

Here, opposite to the first example, we have to sort our data in different branches in order to get result in multiple geometries otherwise it was only one continuous surface.

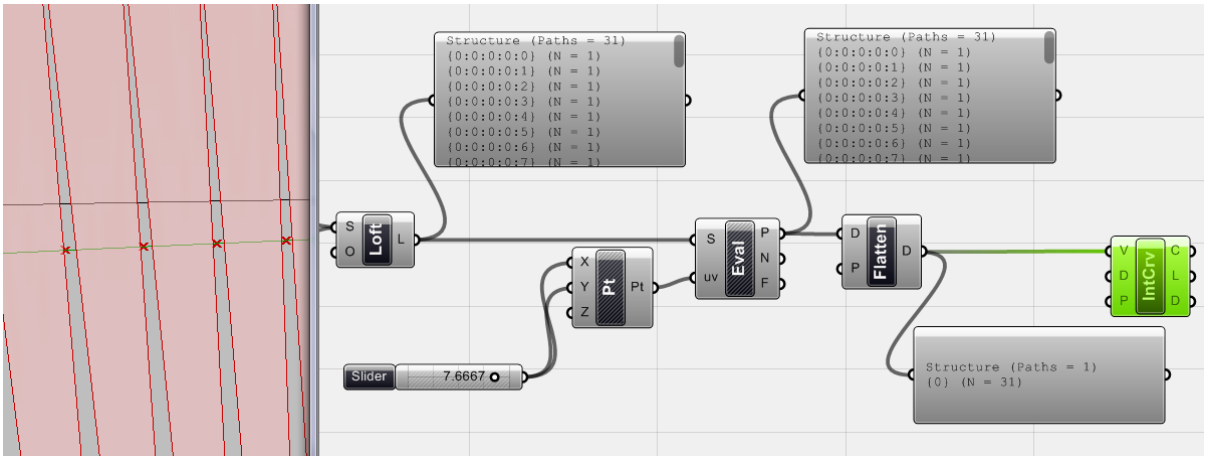


Fig.5.40. Here I should remind you that if you again want to draw an interpolate curve on specific points which you evaluated on these surfaces, since all these surfaces are in different branches of data and the resultant evaluated points would be in different branches of data as well, in order to draw a curve, you need to <Flatten> the data list again, make one branch of data for points and pass it to the interpolate to draw your curve.

The point is to realize that in different components and design situations we have to provide data in branches or in one branch and there are couple of components in Logic>Tree that help us to do so.

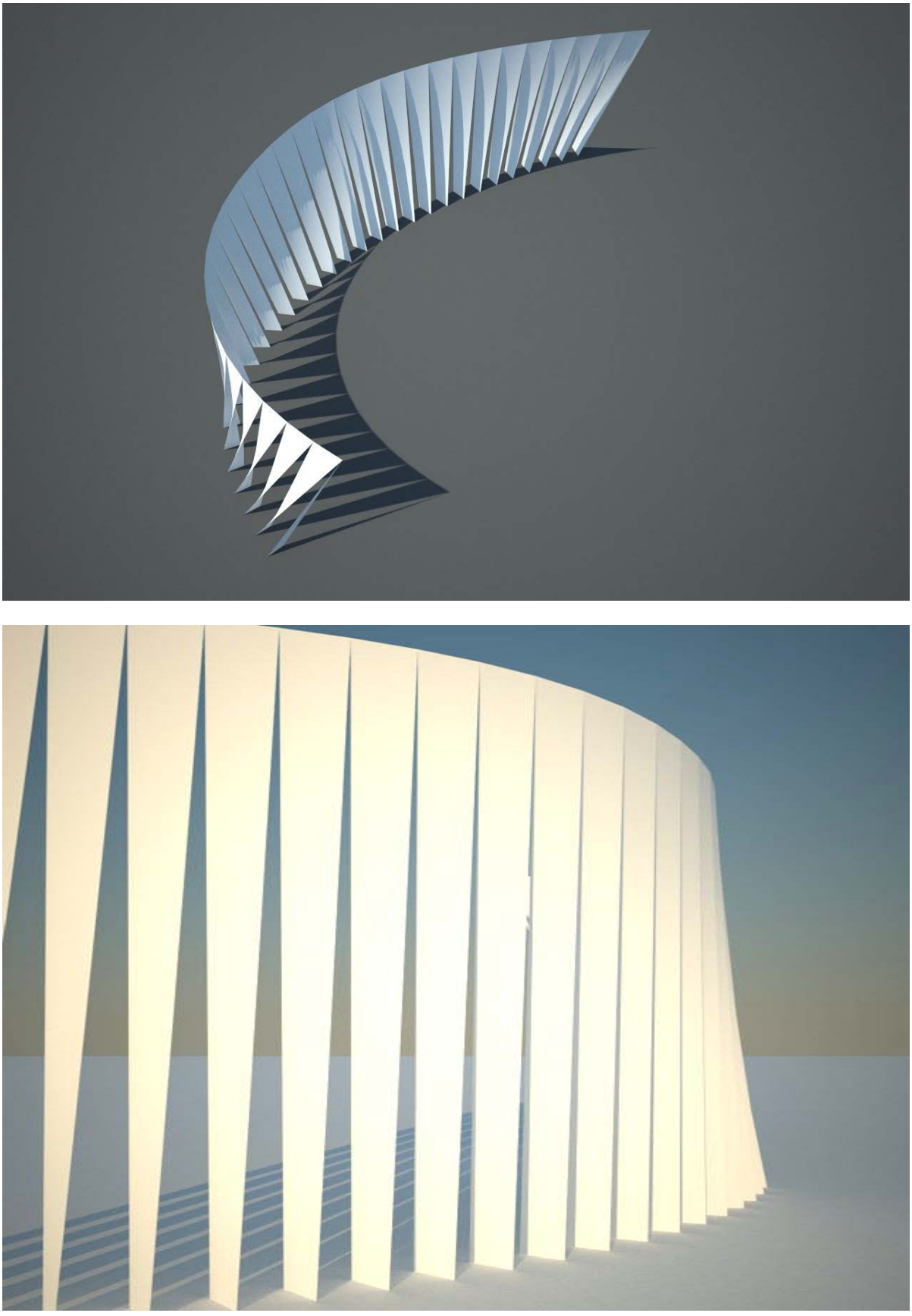


Fig.5.41. Final model of the porous surface.

Chapter_6_ Deformations and Morphing

6_1_Deformations and Morphing

Geometry is not always about pure objects. We need to change the portions and general conditions of volumes and other geometrical products in order to design objects. Deformation and Morphing are some of the tools to do that.

Deformation and Morphing are among the powerful functions in the realm of free-form design. By deformations we can twist, shear, bend, … geometries and by Morphing we can deform geometries from one boundary condition to another.

Let's have a look at a simple deformation. If we have an object like a sphere, we know that there is a bounding-box (cage) around it and manipulation of this bounding-box could deform the whole geometry.

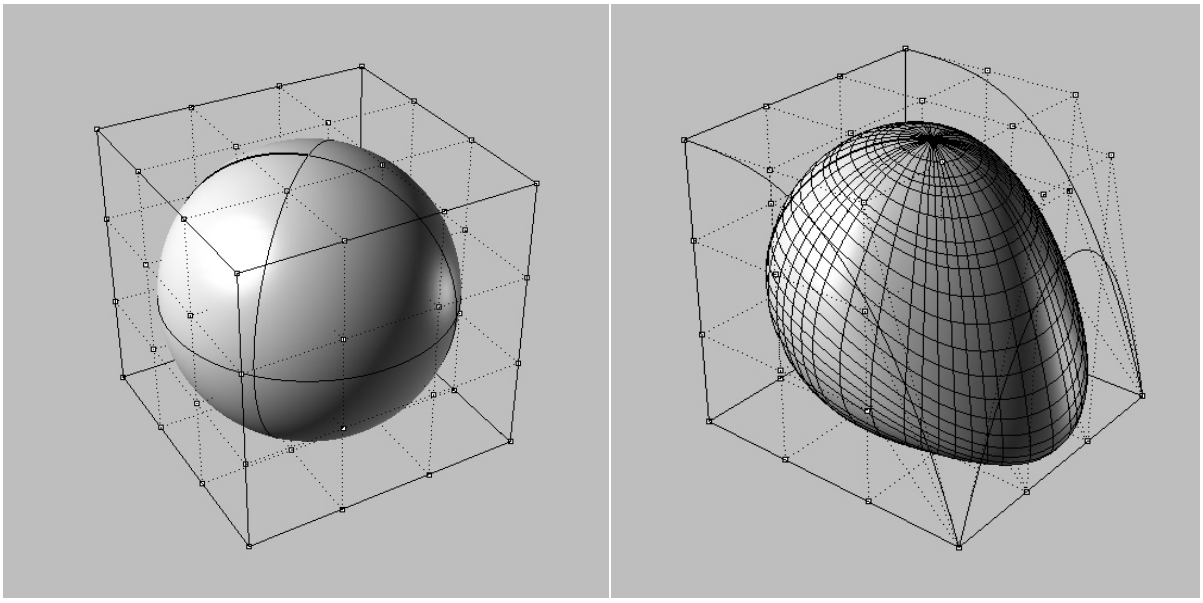


Fig.6.1. Deformation of an object by its Bounding-box (cage).

Based on angel, movement, etc. we might call it shear or bend or free deformation. For any deformation function, we might need the whole bounding-box, or just one of its sides as a plane or even one of the points to deform. If you check different deformation components in Grasshopper you can easily find the base geometrical constructs to perform deformations.

Morphing in animation means transition from one picture to another smoothly or seamlessly. Here in 3D space it means deformation from one state or boundary condition to another. Morphing components in Grasshopper work in the same fashion. Here for example <Box morph> component (XForm>Morph) deforms an object from a reference box (Bounding Box) to a target box, or <Surface morph> component works with a surface as a base, on that you can deform your geometry, on the specific domains of the surface and height of the object.

The first one which is <Box Morph> and the next one is <Surface Morph> both from XForm tab under the Morph panel, and there are couple of additional components there, that could possibly

provide data for these components in our designs. Since we have couple of commands that deform a box, if we use these deformed boxes as target boxes then we can deform any geometry in Grasshopper by combination with Box Morph component.

As you see in Figure 6.2 we have an object which is introduced to Grasshopper by a <Geometry> component. This object has a bounding-box around it which I draw here just to visualize the situation. I also draw another box by manually feeding values.

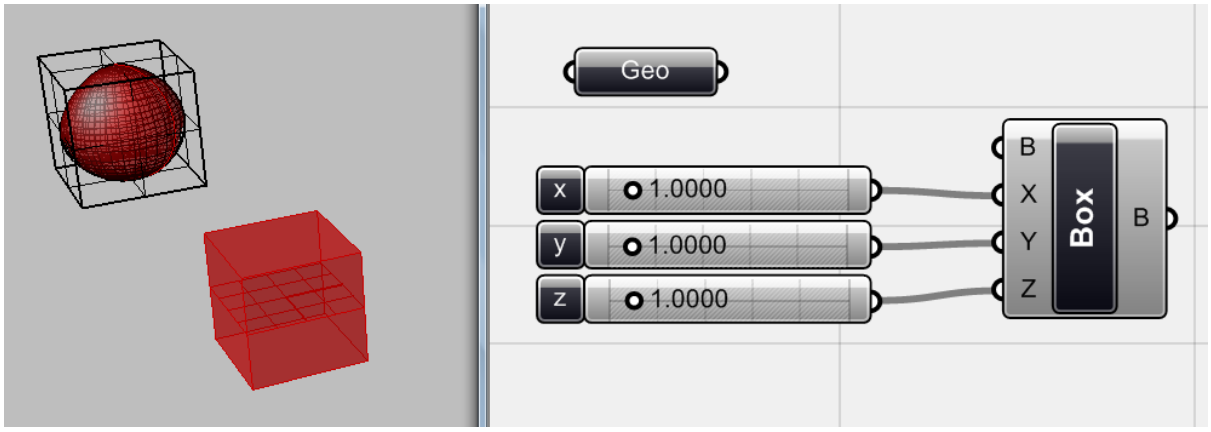


Fig.6.2. Initial object (sphere) and manually fed box.

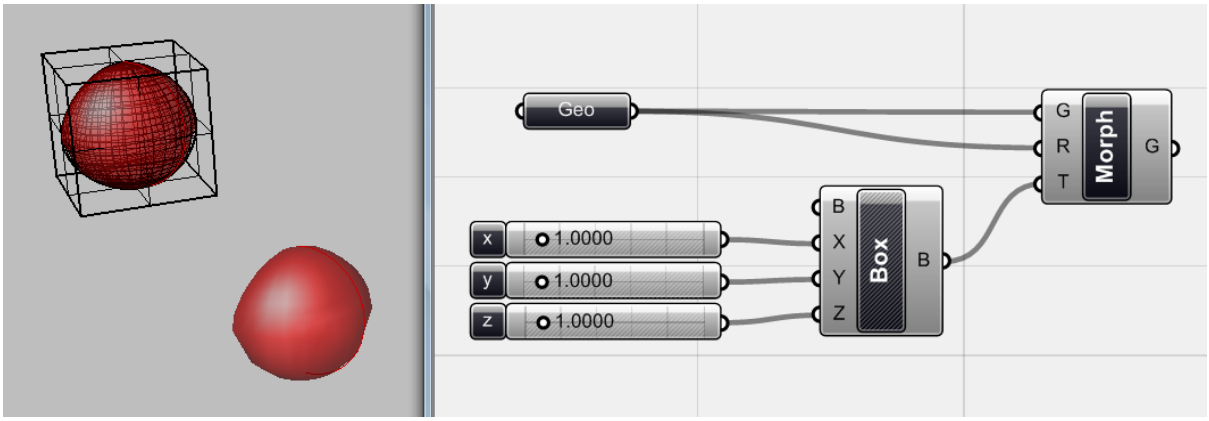


Fig.6.3. A <Box morph> component (XForm > Morph > Box morph) deforms an object from a reference box to a target box. Because I have only one geometry I attached it as the geometry and also as bounding box or reference box to the component (if there are different geometries or in other cases, you can use <Bounding box> component (Surface > Primitive > Bounding box) as well). I unchecked the preview of the <Box> component to see the morphed geometry inside it better.

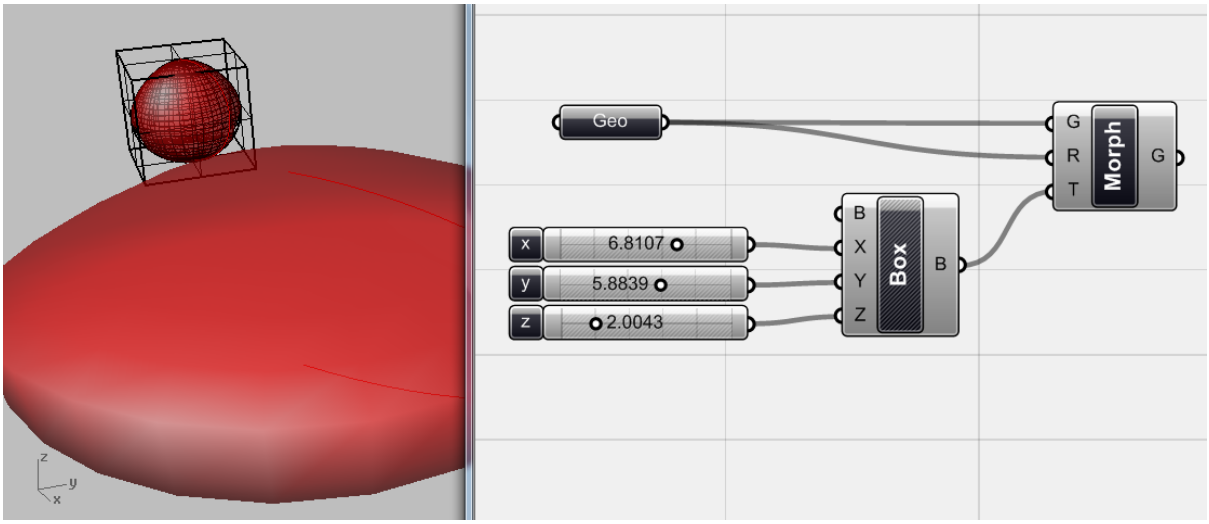


Fig.6.4. Now if you simply change the size of the target box you can see that the morphed geometry would change accordingly.

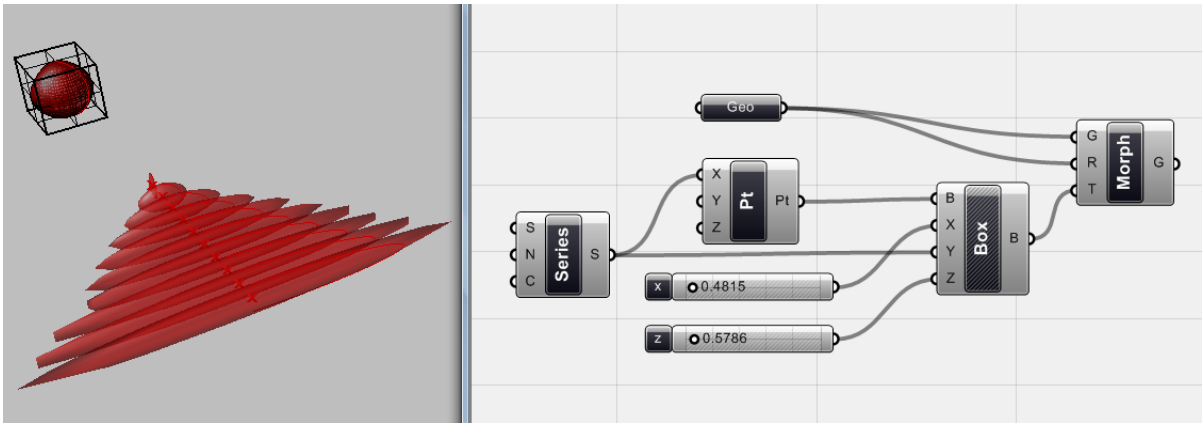


Fig.6.5. Here you see that instead of one box, if I produce bunch of boxes, I can start to morph my object more and more. As you see, differentiated boxes by the <series> component in their Y dimension, caused differentiation in morphed objects as well.

6_2_On Panelization

One of the most common applications of morphing functions is Panelization. The idea of panelization comes from the division of a free-form surface geometry into small parts and pieces especially for fabrication issues. Although free-form surfaces are widely being used in car industry, it is not an easy job for architecture to deal with them in large scales. Benefit of panelization is to divide a surface into small parts, called components which are easier to fabricate and transport and also more controllable in terms of precision in final product.

It is also possible sometimes to divide a curve surface into small flat parts and then get the overall curvature by accumulation of the flat geometries which could be then fabricated from sheet materials. There are multiple issues regarding the size, curvature, adjustment, etc. that we try to discuss some of them.

Let's start with a simple surface and a component as a module to panelize this surface.

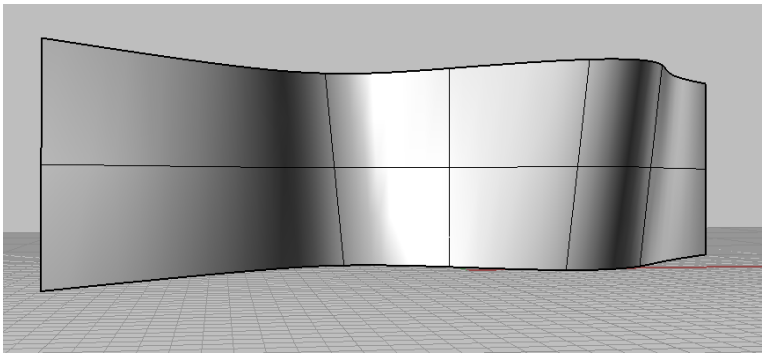


Fig.6.6. A Generic double-curved surface for panelization.

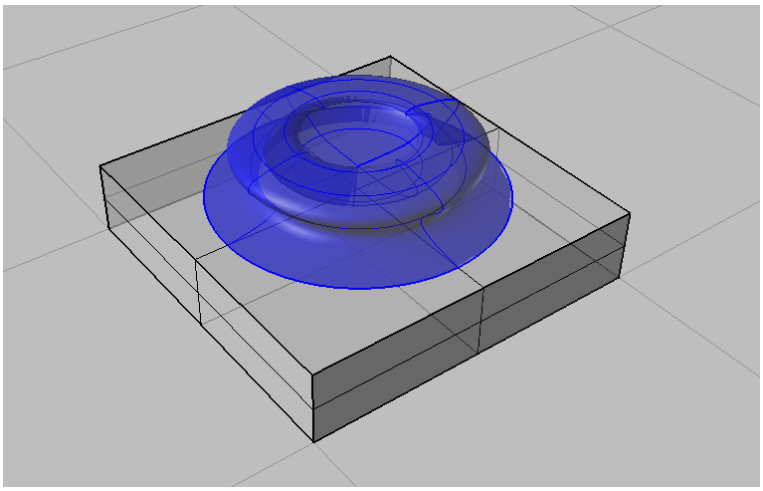


Fig.6.7. Component that I want to proliferate on the surface……. Not special, just as an example!!!

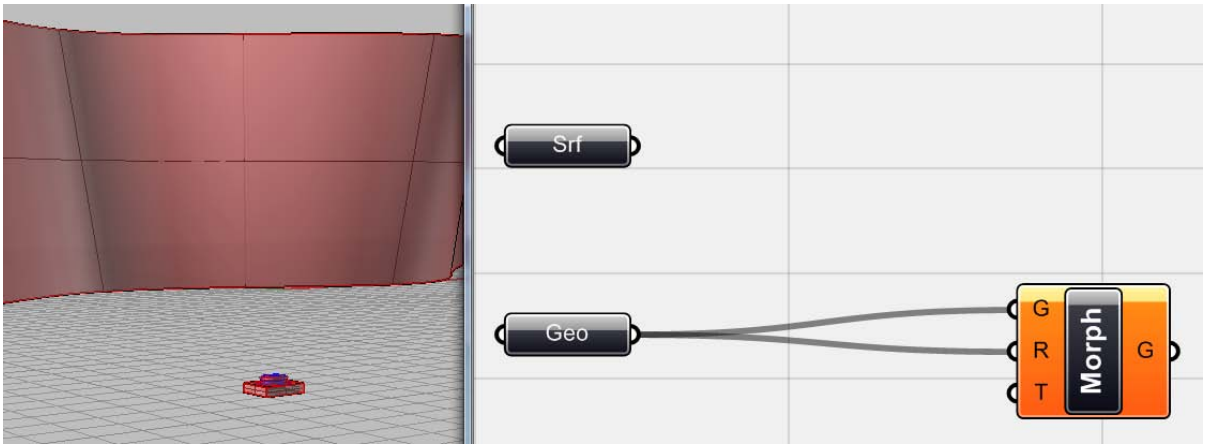


Fig.6.8. First of all, I need to introduce the surface and module as Grasshopper components. Based on the possible components in the Grasshopper, the idea is to generate couple of boxes on the surface and use these boxes as target boxes and morph the module into them. So I introduced a <box morph> and I used the module as geometry and as bounding-box. Now I need to generate target boxes to morph the component into them.

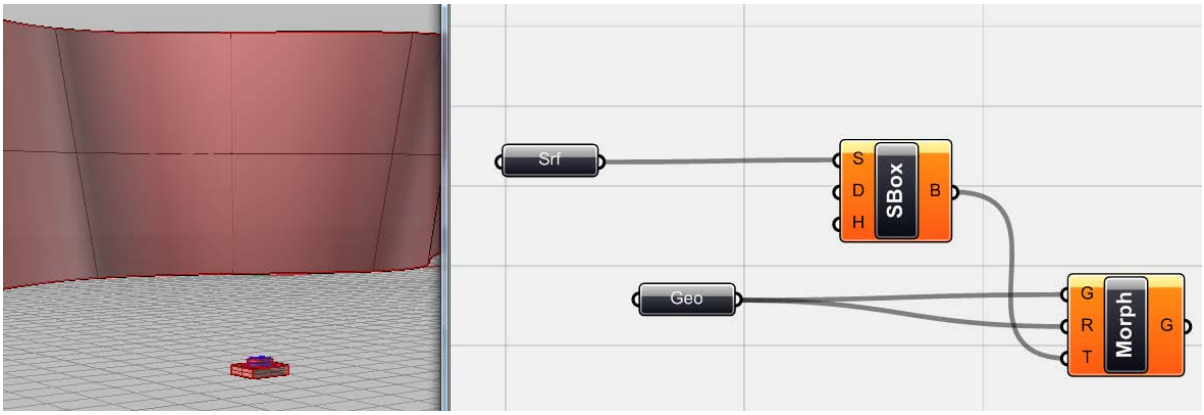


Fig.6.9. The component that I need to make target boxes is <surface box> (XForm > Morph > Surface box). This component generates multiple boxes over a surface based on the intervals on the surface domain and height of the box. So I just attached the surface to it and the result would be target boxes for the <box morph> component. Here I need to define the domain interval of the boxes, or actually divide the numeric interval of the surface in its U and V direction to generate boxes.

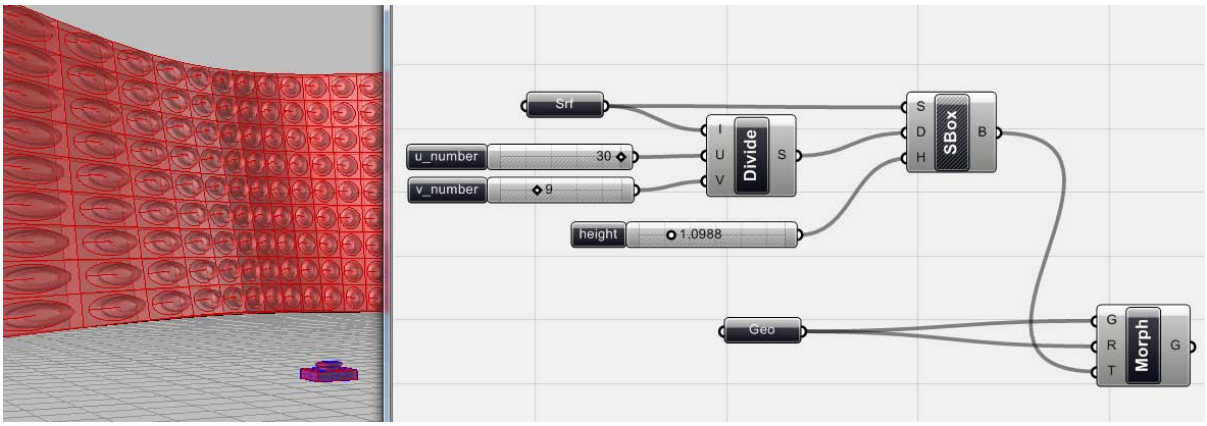


Fig.6.10. In order to divide the surface domain, I used <divide interval2> which tells the <surface box> that how many divisions in U and V directions I need. Another <number slider> defines the height of target boxes which means height of morphed components.

So basically the idea is simple. We produce a module (a component) and we design our general surface. Then we generate certain amount of boxes over this surface (as target boxes) and then we morph the module into these boxes. After all we can change the number of elements in both U and V direction and also change the module which updates automatically on the surface.

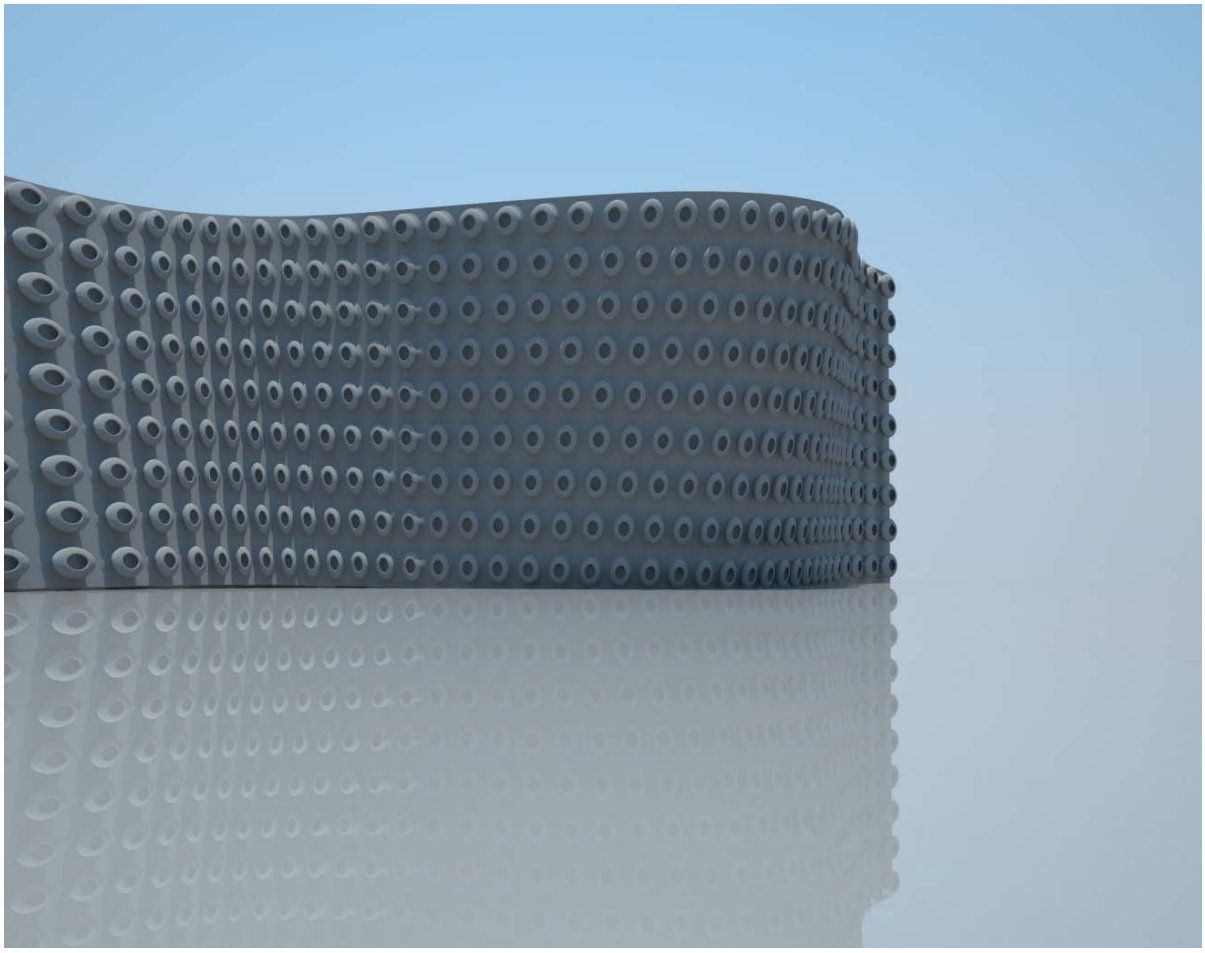


Fig.6.11. Final surface made up of our base modules

6_3_Micro Level Manipulations

Although it is great to proliferate a module over a surface, it still seems a very generic way of design. We know that we can change the number of modules, or change the module by itself, but the result is a generic surface and we don't have local control of our system.

Now I am thinking of making a component-based system that we could apply more local control and avoid designing generic surfaces which are not responding to any local, micro-scale criteria.

In order to introduce the concept, let's start with a simple example and proceed towards a more practical one. We used the idea of attractors to apply local manipulations to a group of objects. I am thinking of applying the same method to design a component based system with local manipulations by an attractor. The idea is to change the components size (in this case, their height) based on the effect of a point attractor.

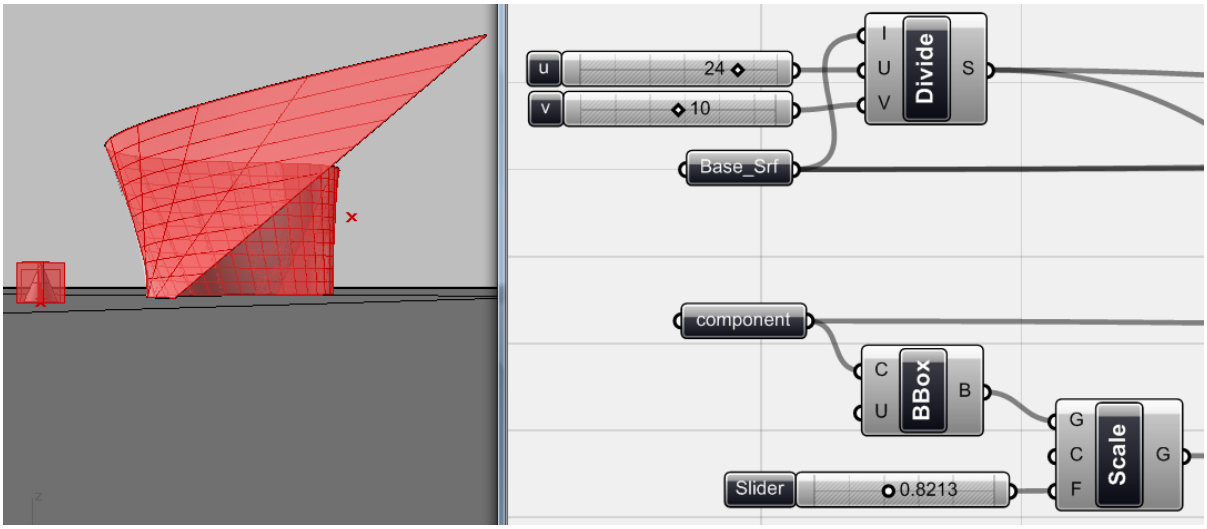


Fig.6.12. Lets look at the ingredients: A double-curved surface introduced as <Base_Srf> and a cone introduced as <component> to the Grasshopper, a <divide interval2> for surface divisions, and a <bounding box> as the reference box of the <component>. Here I used a <scale> component for my bounding box. Later on, if I change the size of the bounding box, I can change the size of all <component>s on the <base_srf> because of the change in reference box.

The <surface box> component has the height input which asks for the height of boxes in the given intervals. The idea is to use relative heights instead of constant one. So instead of one number as height, we can make a relation between the position of each box to the attractor's position and generate different numbers as associative heights.

What I need is to measure the distance between each box and the attractor. The technical problem here is that there is not any box generated yet, so I need a point on surface at the center of each box to measure the distance.

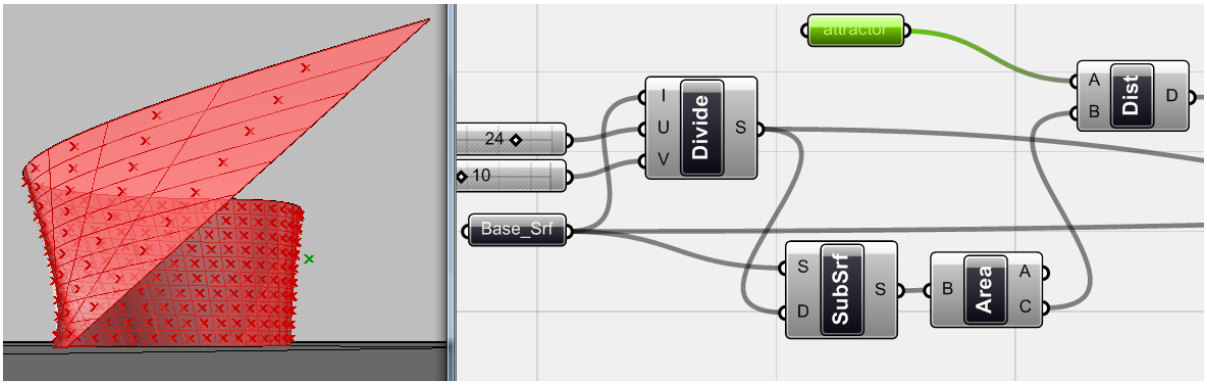


Fig.6.13. Here I used the same <divide interval2> which I want to use for <surface Box> for an <Isotrim> component (Surface > Util > Isotrim). This component divides the surface into sub-surfaces. By these sub-surfaces I can use another component which is <BRep Area> (Surface > Analysis > BRep area) to use the by-product of this component which is 'Area Centroid' for each sub-surface. I measured distances of these points (area centroids) from the <attractor> to use them as reference factors for height of the target boxes in <surface box> component.

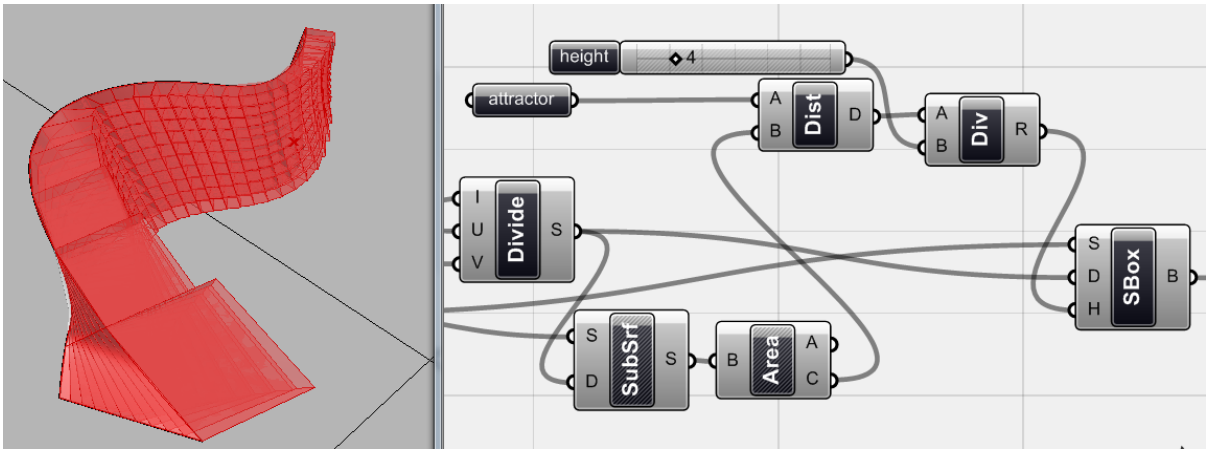


Fig.6.14. Now I divided the measured distances by a given number from <number slider> to control the effect of the attractor and I used the result as 'height' input to generate target boxes with <surface box> component. The surface comes from the <base_srf>, the <divide interval2> used as surface domain and the heights coming from the relation of box positions and the attractor. As you see, the height of boxes differ, based on the position of the <attractor> point.

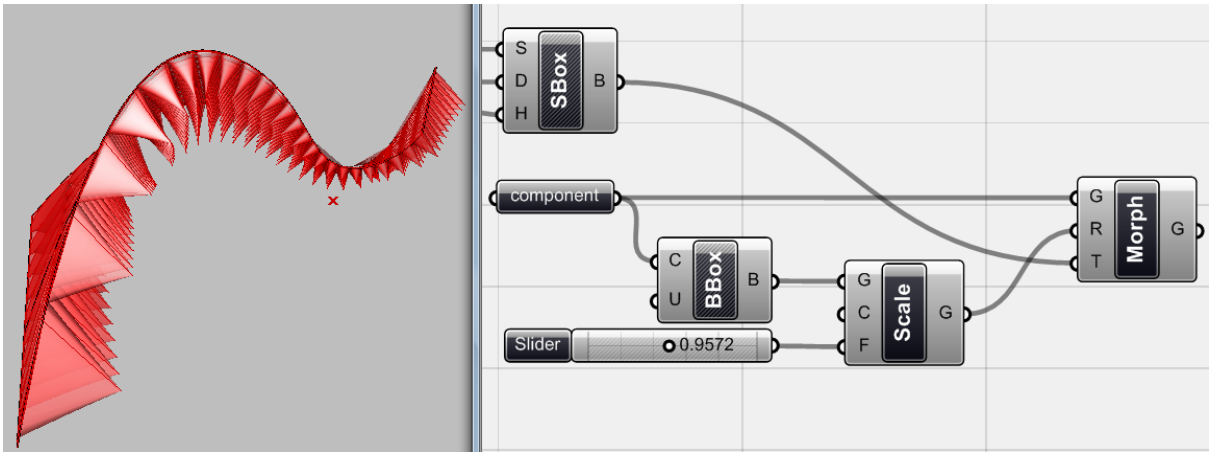


Fig.6.15. The only remaining part, connecting the <component>, <scale>d bounding box and <surface box> to a <morph box> component which proliferates component over the surface. By changing the scale factor, you can change the size of the all components and like always, position of the attractor is also manually controllable.

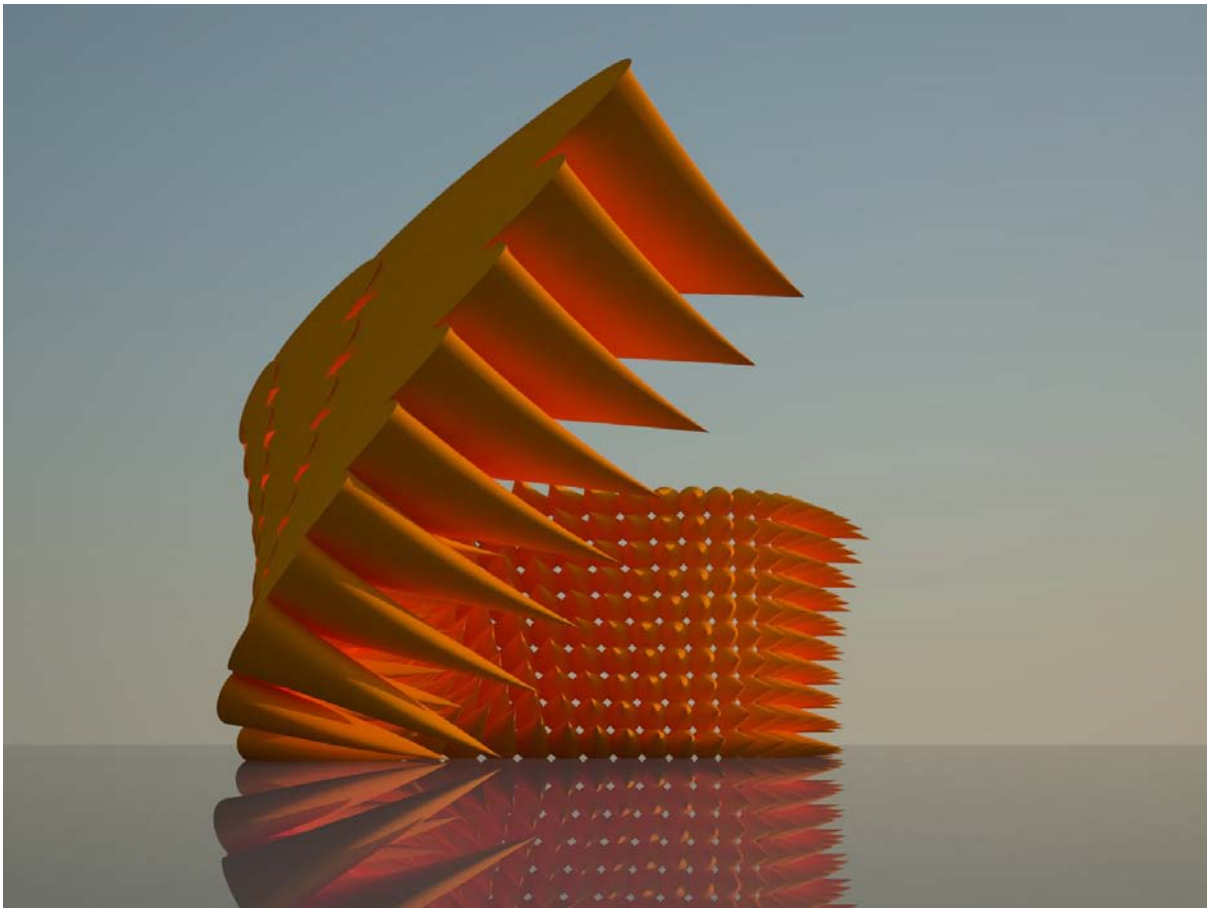


Fig.6.16. Final model.

As you see, the size of components started to accept local manipulations, based on an external property which is a point attractor here. Although the idea is a simple attractor, the result could be interesting and the idea is to show that we could differentiate reference boxes and get differentiated results as well. Now we know that the morphing concept and panelization is not always generic. Having tested the concept, let's go for another practical experiment.

6_4_On Responsive Modulation

The idea for the next design experiment is to modulate a given surface with control over each module which means any module of this system, has to be responsible for some certain criteria. So even more than regional differentiation of the modules, here I want to have a more specific control over my system by given criteria which could be environmental, functional, visual or any other associative behaviour that we want our module be responsible for.

In the next example, in order to make a building's envelope more responsive to the host environment, I wanted the system to be responsive to the sun light. In your experiments it could be wind, rain or internal functions or any other criteria that you are looking for, even combination of them.

Here I have a surface, simply as the envelope of a building which I want to cover with two different types of components. The first one is closed and does not allow penetration of the sun light and the other has opening. These components should be proliferated over my envelope based on the main direction of the sun light at the site. I set a user defined angle to say the algorithm that for the certain degrees of sun light we should have closed components and for the others, open ones.

Grasshopper definition does not have anything new, but it is the concept that allows me to make variations over the envelope instead of making a generic surface. Basically when the surface is free-form and it moves around and has different orientations, it has different angles with the main sun light at each part, so based on the angle differentiation between the surface and sun light, this variation in components happens in the system.

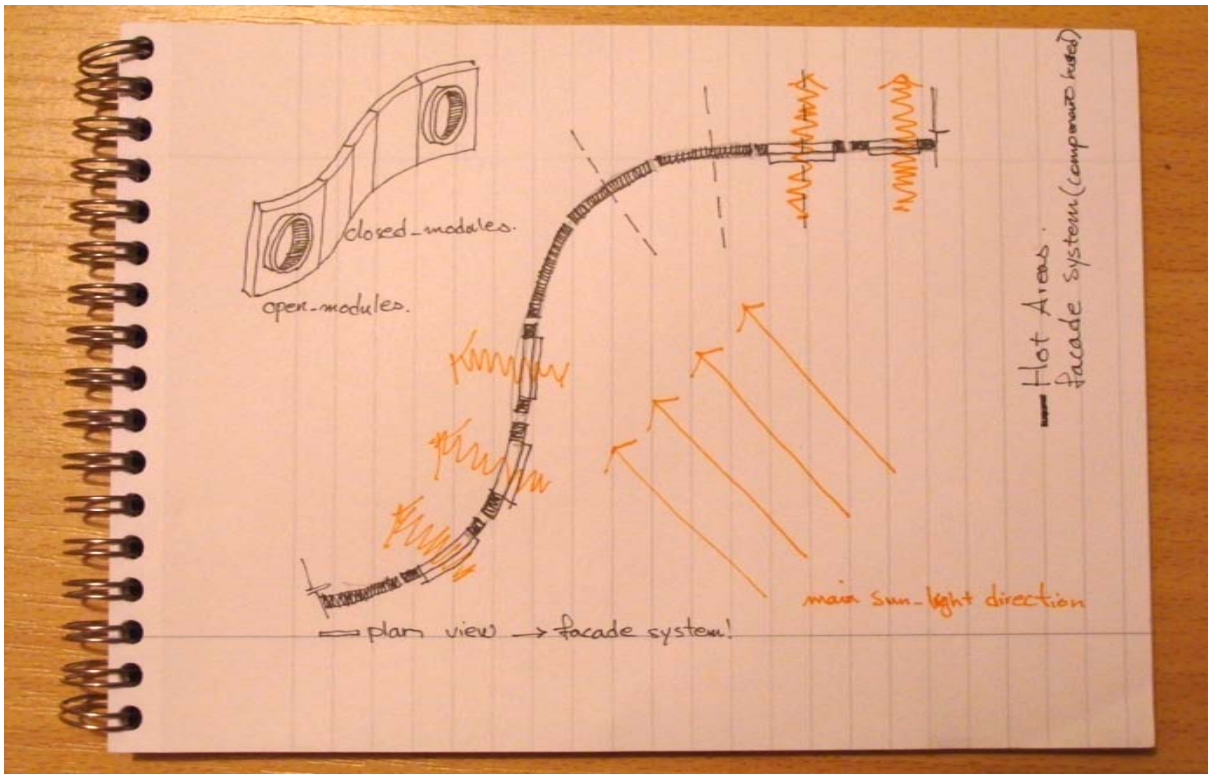


Fig.6.17.First sketches of responsive modules of a façade system.

Ingredients:

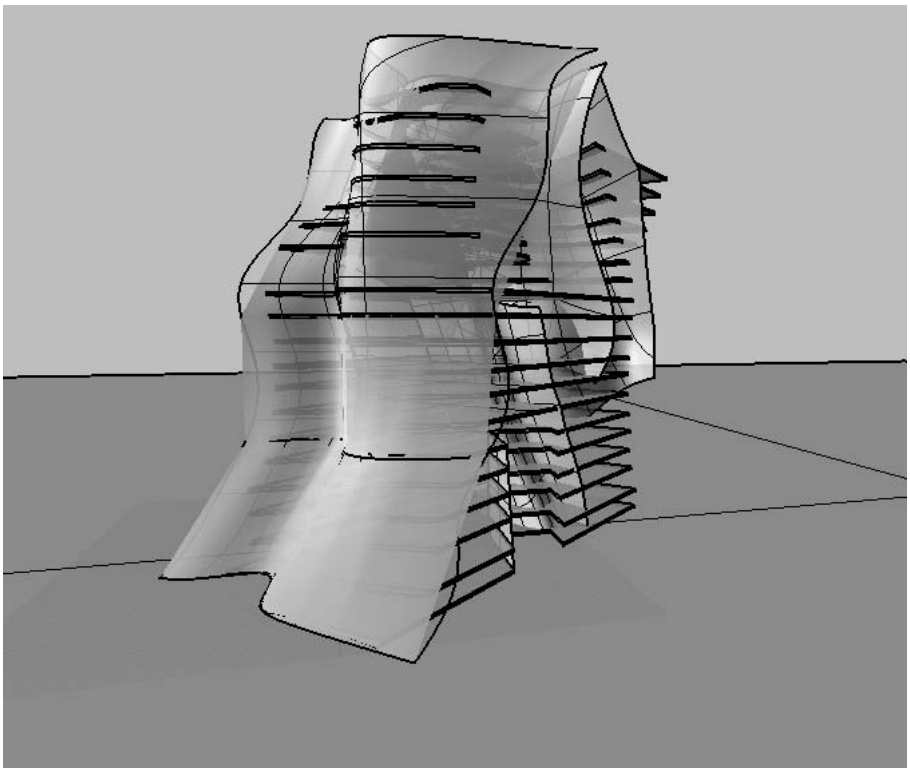


Fig.6.18. External surface of building as envelope which is aimed to panelize.

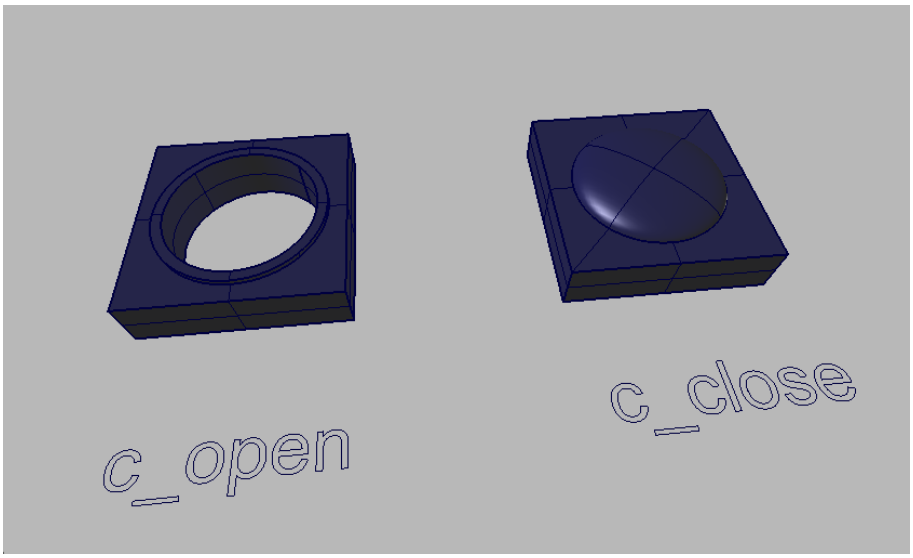


Fig.6.19. Two different types of components for panelization.

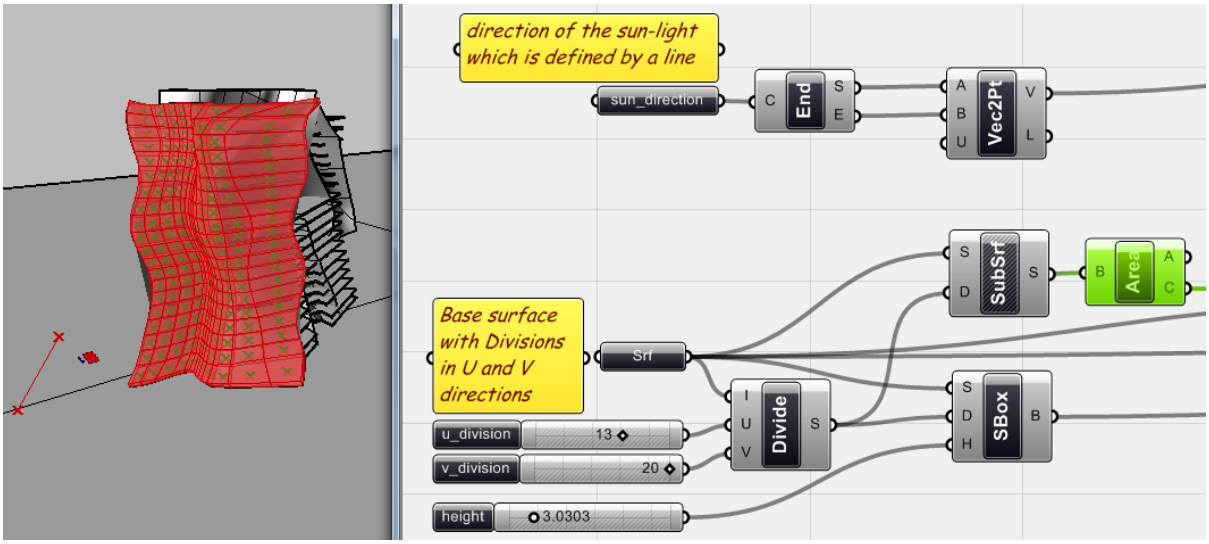


Fig.6.20. The first step is similar to the previous experiments. I introduced <surface> and I used <divide interval2> to divide it in U and V directions and I generated target boxes by <surface box>. I also used <isotrim> with the same intervals as boxes to find the positions of boxes on the surface and I used <BRep area> to find the centroid of this area (which is selected in green). At the same time I used a <curve> component to introduce the main sun-light angle of the site and whit its <end points> I made a <vector 2pt> which specify the direction of the sun light. You can manipulate and change this curve to see the effect of sun light on components in different directions.

As you can see from the first image, there is a surface as envelope which is divided into parts for component generation and there is a sun-light vector. I want to know angle between this vector and the surface at the position of each component. I have to have a unique result for angle calculation, and the best way is to use Normals of surface which are unique at each point. A Normal is a vector which is perpendicular to the surface at a specific point. So I can use that to check the angle for each component.

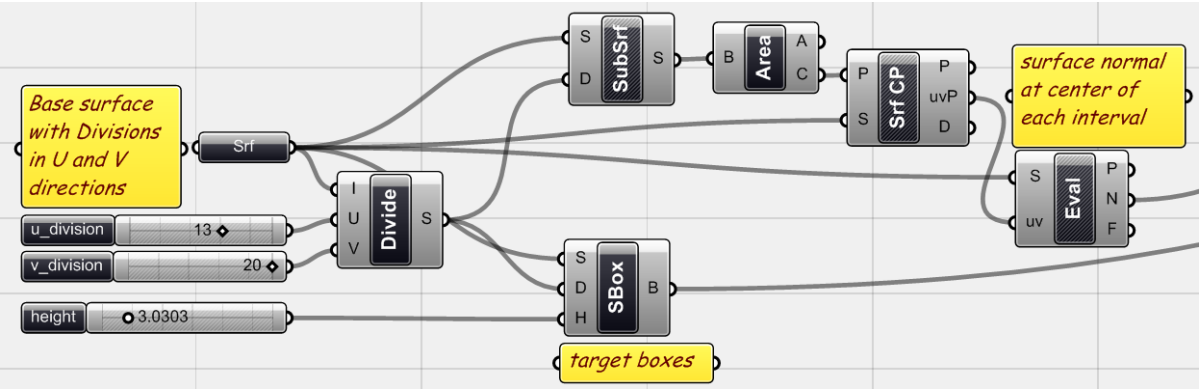


Fig.6.21. in order to evaluate the angle between sun-light and surface, I want to measure this angle between sun light and normals of the surface at the position of each component. So I can decide for each range of angles what sort of component would be suitable. So after generating the center points, I need normals of the surface at those points. That's why I used a <surface CP> to get the UV parameters of points on the surface and use these parameters to <evaluate> the surface at those points to actually get the normals of surface at those points.

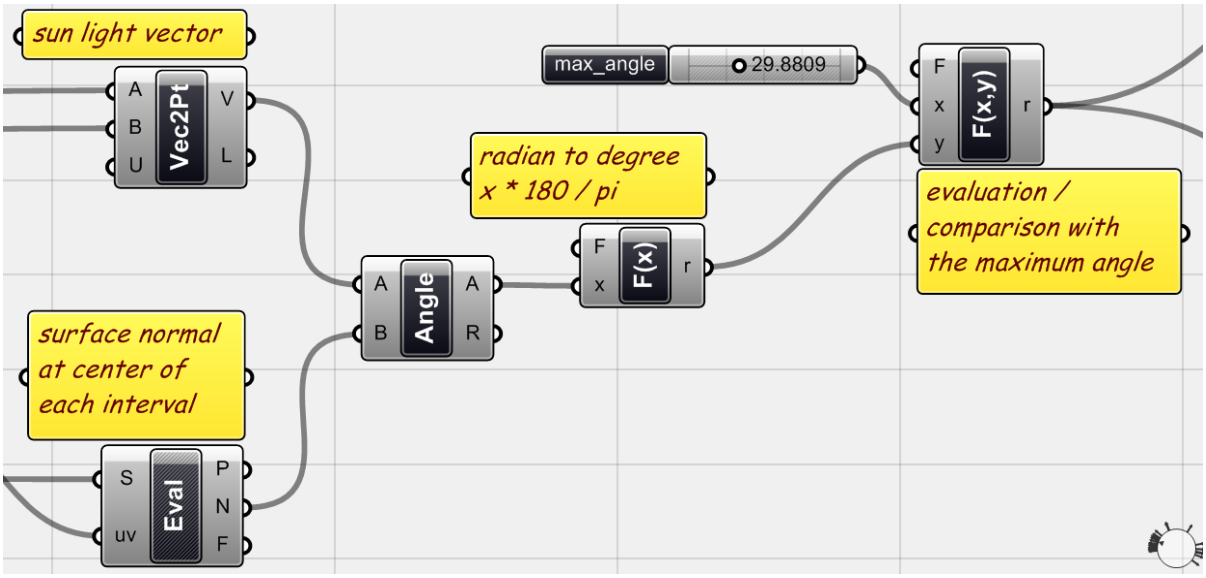


Fig.6.22. Now I used an <angle> component (Vector > Vector > Angle) to evaluate the angle between the sun direction and the façade. Then I converted this angle to degree and I used a <function> to see whether this angle is bigger than the <max_angle> or not. This function (x>y) gives me Boolean data, True for smaller angles and False for bigger angles.

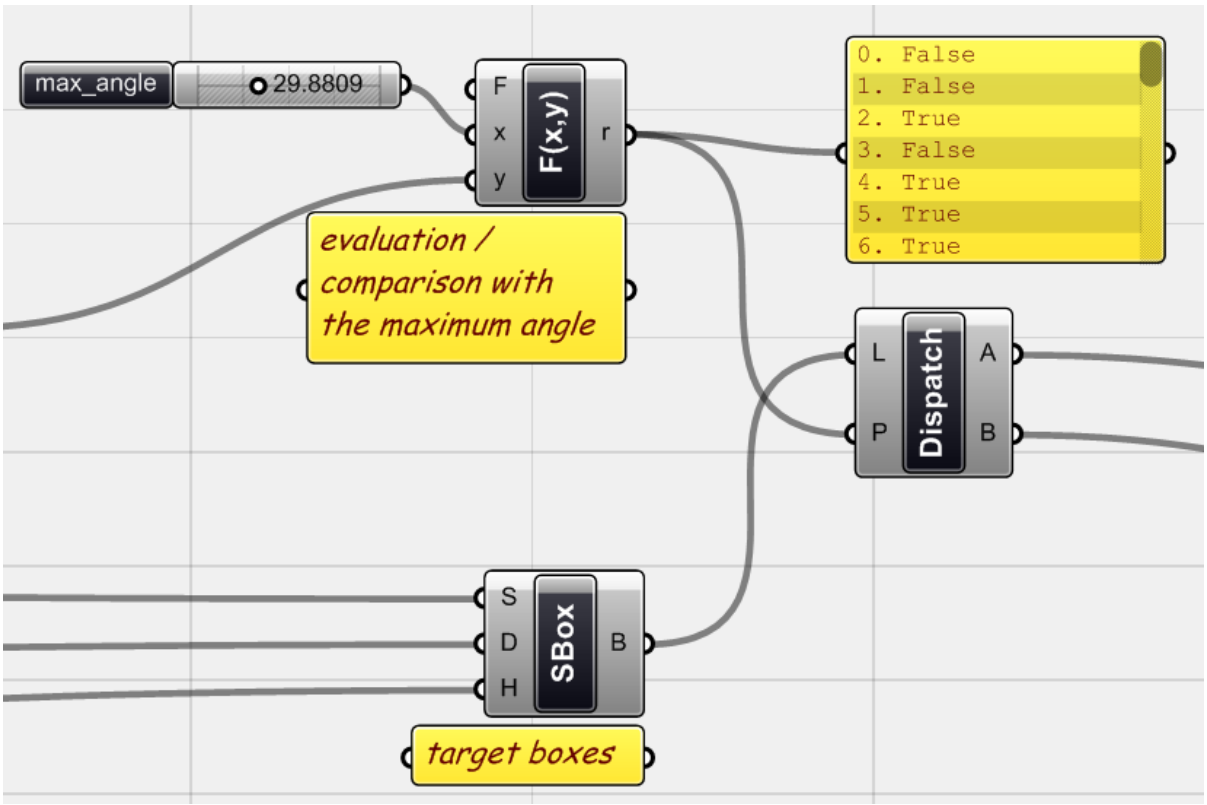


Fig.6.23. Based on the Boolean data comes from the angle comparison, I <dispatch> the data which are target boxes (I have the same amount of target box as the center points and normals so I can use target boxes instead of points). So basically I divided my target boxes in two different groups whose difference is the angle they receive the sun light.

The rest of the algorithm is simple and like what we have done before. I just need to morph my components into the target boxes, here for two different ones.

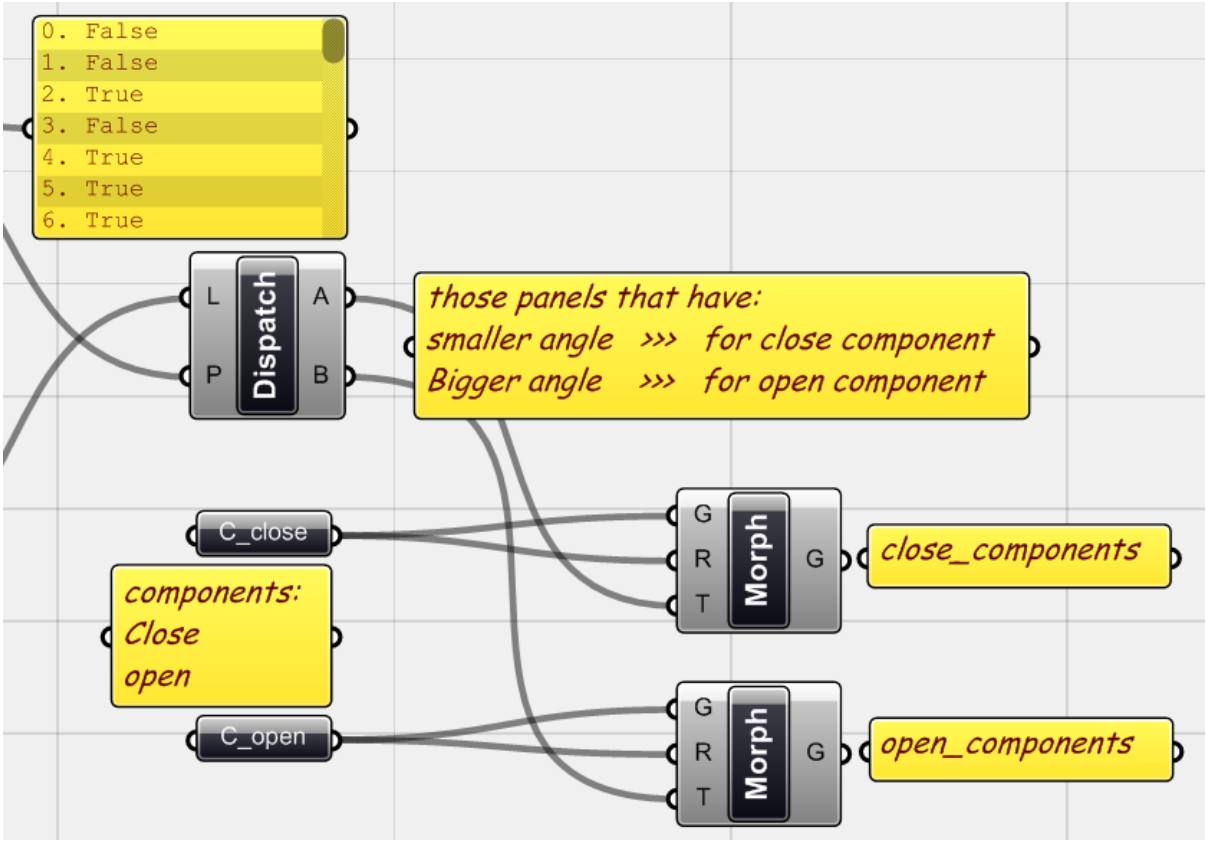
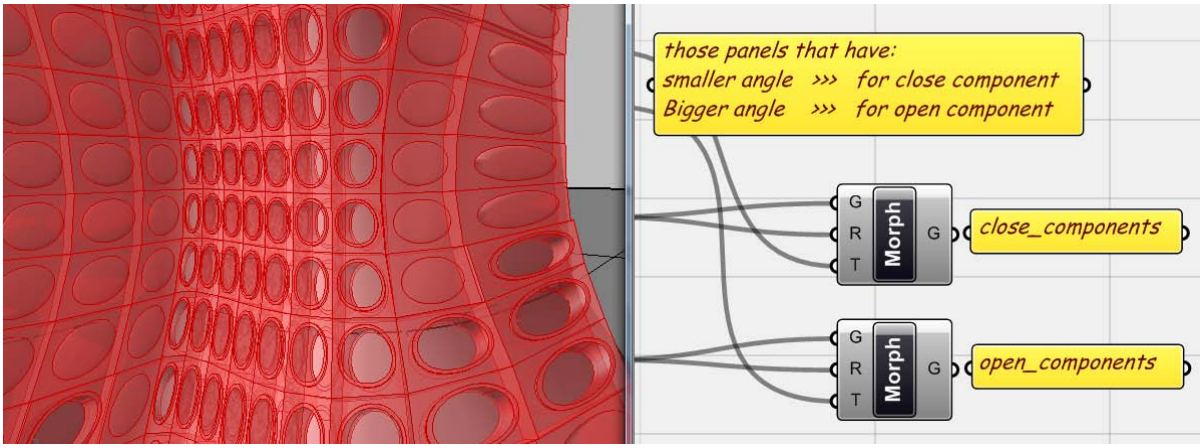


Fig.6.24. Here I introduced two different components as single geometries and I used two <morph box> components each one associated with one part of the <dispatched> data to generate <C_close> or <C_open> components over the façade.



6.25. Now if you look closer, you can see that in different parts of the façade, based on its curvature and direction, different types of components are generated.

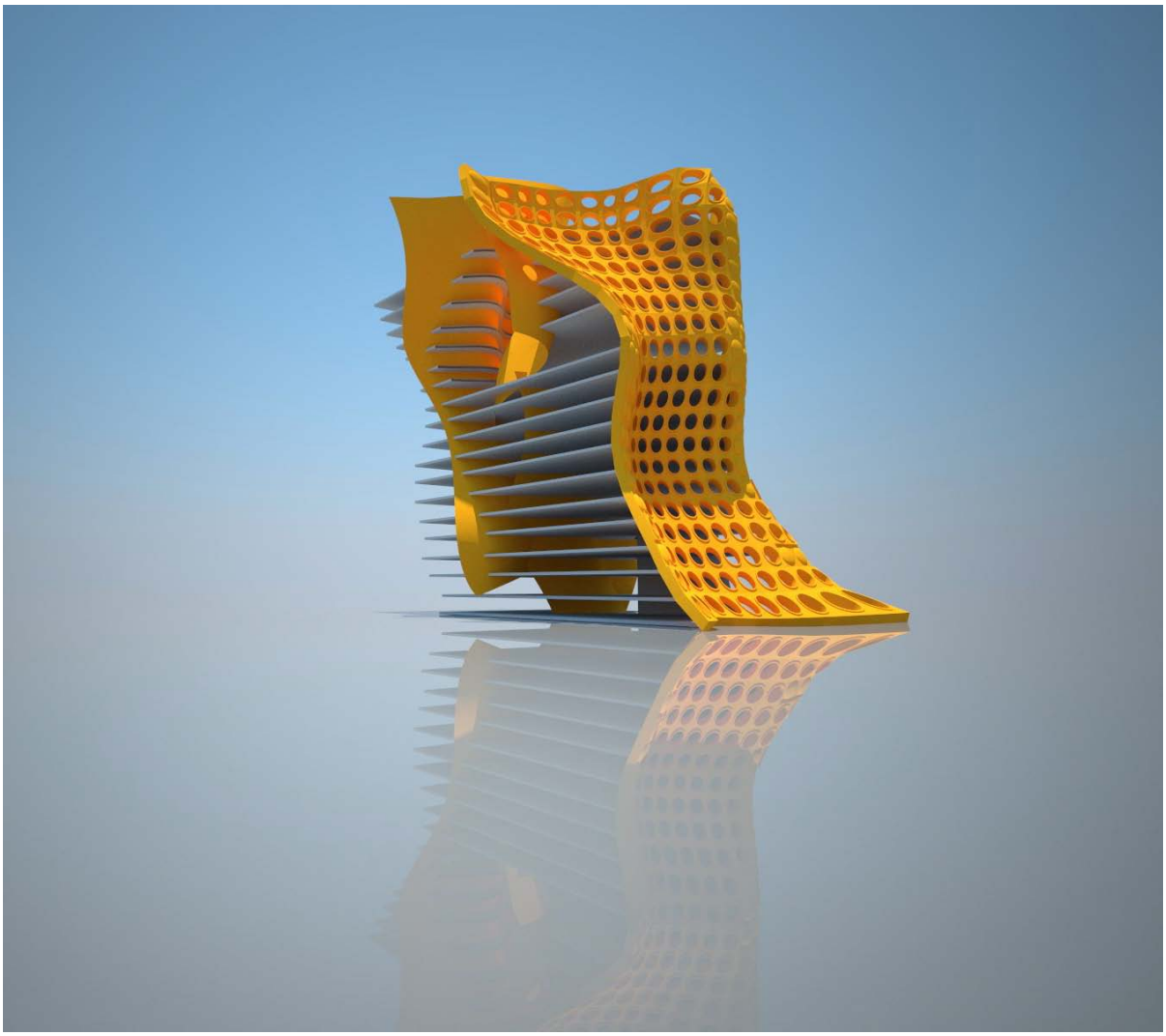


Fig.6.26. Final model. The bifurcation of target boxes (and components) could be more than two in the algorithm. It depends on design and criteria that we use.

We can think about a component based façade, in which some components are closed, and some are open, which open ones have adjustable parts that orientate towards external forces, and even reflect to the internal functions of the building and so on and so forth. You see that the idea is to have micro scale control over the system and avoid generic designs. And it is clear that still we have global (surface by itself) and regional (component by itself) control over the system as well.

Chapter_7_NURBS Surfaces and Meshes

7_1_Parametric NURBS Surfaces

We have had some experiments with surfaces in previous chapters. We used Loft and Pipe to generate some surfaces. We also used free form surfaces and some surface analysis components accordingly. Usually by surfaces, we mean Free-Form NURBS surfaces. In many cases generating surfaces depends on other basic geometries like curves that we provide for our surface geometries or sometimes points. There are multiple surface components in Grasshopper and if you have a little bit of experience working with Rhino you should already know how to generate your surface geometries by them.

Surface geometries seems to be the final products in our design, like facades, walls etc. and that's why we need lots of effort to generate the data like points and curves that we need as the base geometries. Here I decided to design a very simple schematic building just to indicate that the multiple surface components in the Grasshopper have the potential to generate different design products by very simple basic constructs. I know the design process by itself might not be satisfying , here I just want to concentrate on using new components.

Parametric Tower

In the areas of Docklands of Thames in London, close to the Canary Warf, where the London's high rises have the chance to live, there are potentials to build some towers. I assumed that we can propose one together, and this design could be very simple and schematic, here just to test some of the basic ideas of working with free-form surfaces.

Let's have a look at the area.

Fig.7.1. Arial view, Canary Warf, London (image: www.maps.live.com, Microsoft Virtual Earth).

The site that I have chosen to design my project is in the bank of Thames, with a very prestigious view to the river and close to the entrance square of the centeral area of Canary Warf (Westferry Road). I don't want to go through site specifics so let's just have a look at where I am going to propose my tower and continue with geometrical issues.

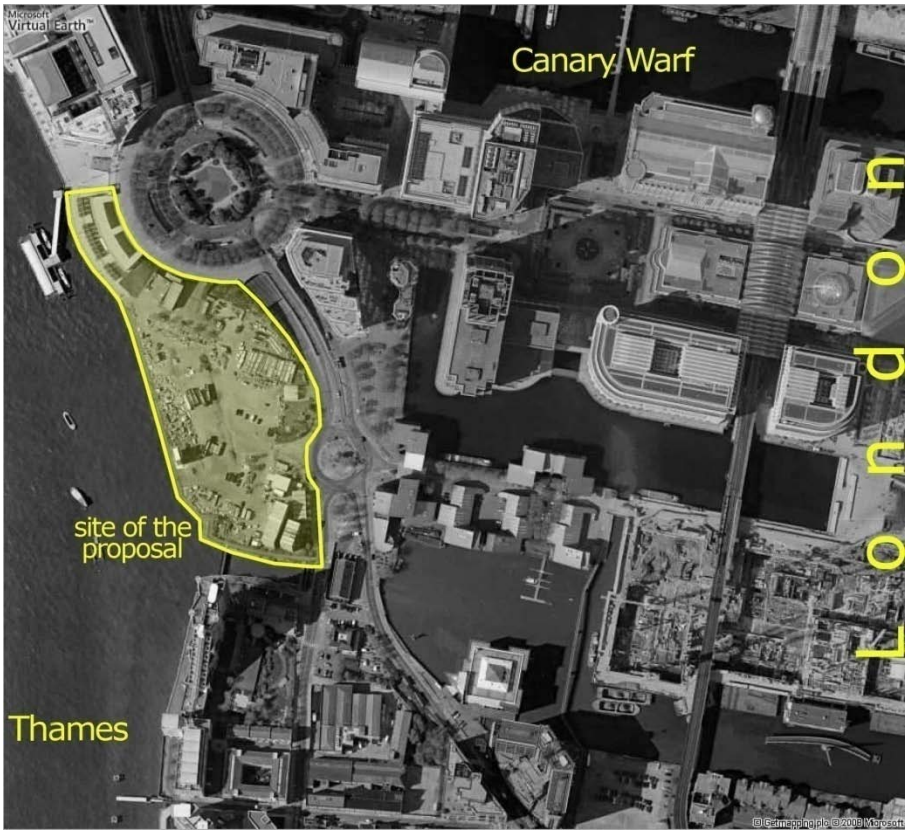


Fig.7.2. Site of the proposed tower.

Manual drawings

There are multiple ways to start this sketch. I can draw the ground floor and copy it above and start to manipulate them and add details. I am sure that you already searched the web and got different ideas about designing a tower associatively with this technique. Here I decided to use some surface components so the technique might not be appropriate but the aim is to expand our experiments.

I have a vague idea in mind. My tower has a general glass surface which is covered by some linear elements in façade, but because I don't like to design a conventional tower, I also want to have some hollow spaces on tower skin, scattered across the façade. These volumes would intersect façade's linear elements so these elements should be cut then. I also want to design a public space close to the river and connected to the tower with the same elements as façade, continuous from tower towards the river bank.

As you see in Figure 7.3 I drew my base curves manually in Rhino. These curves correspond to the site specifics, height limitations, site's shape and borders, etc, etc. The first curve was drawn and then it mirrored for the next corner and again mirrored for the next one and so on. Another two curves drawn as the borders of the public space, they started from the earth level and then went up to be parallel to the tower edges. These curves are experimental. You can draw whatever you like and go for the rest of the process.

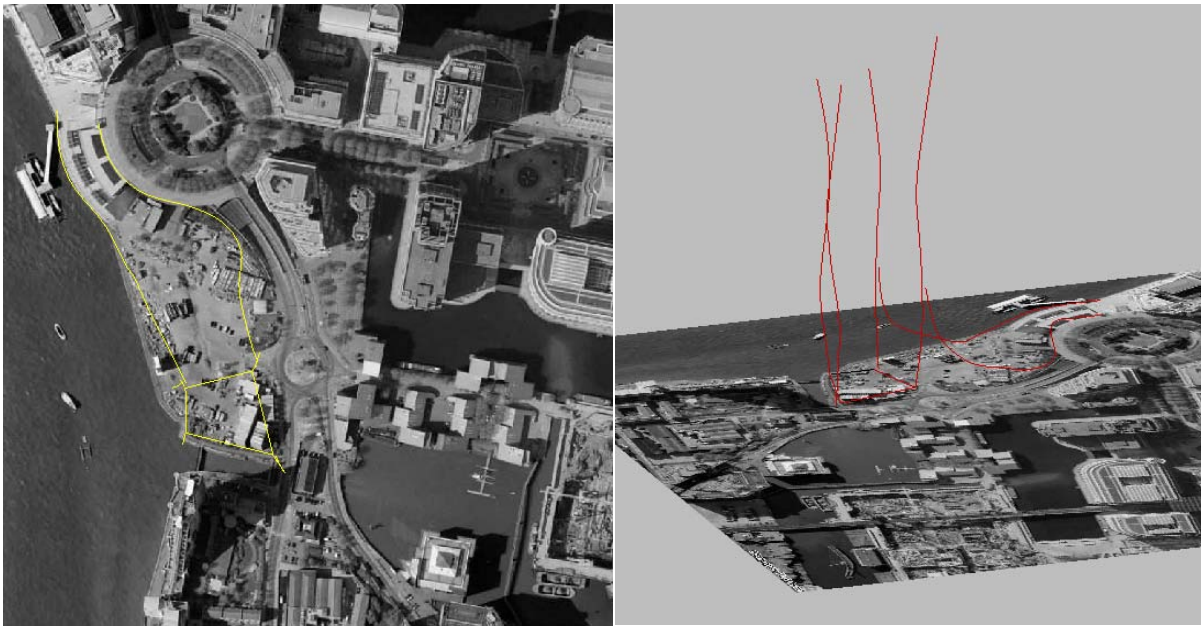


Fig.7.3. Basic lines of the tower's site.

Basic façade elements

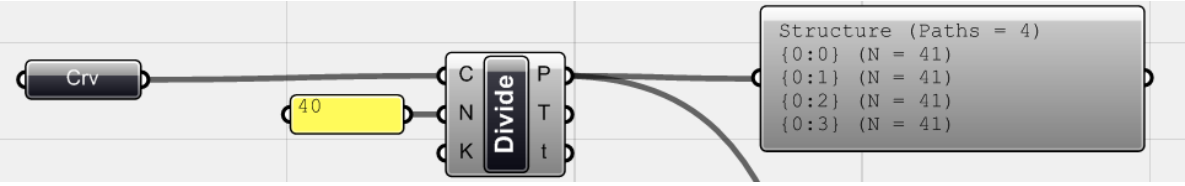


Fig.7.4. For the first step, I imported these four corner curves into Grasshopper by a <curve> component and then I used <divide curve> to divide these curves into 40 parts as floors of the tower. As you see, the resultant division points are sorted in four different data branches.

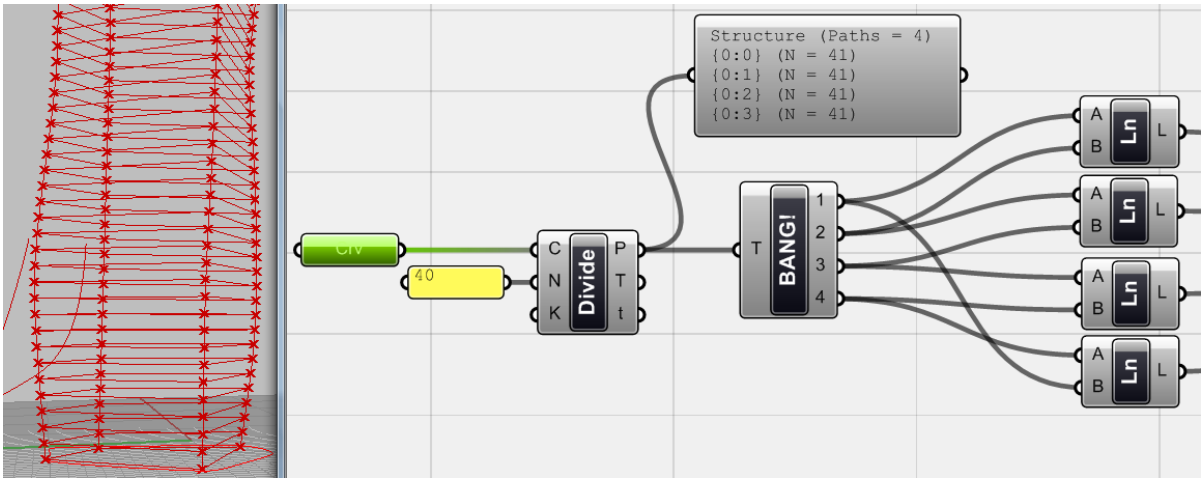


Fig.7.5. Now I have to draw basic lines by these division points on façade to use them for façade elements. Here I want to draw lines from division points of each curve to the same point of the next curve. To do that, I used an <Explode Tree> called <Bang> component (Logic > Tree > Explode Tree) to have access to different branches of data separately. I added <line>s from each branch points to the next ones.

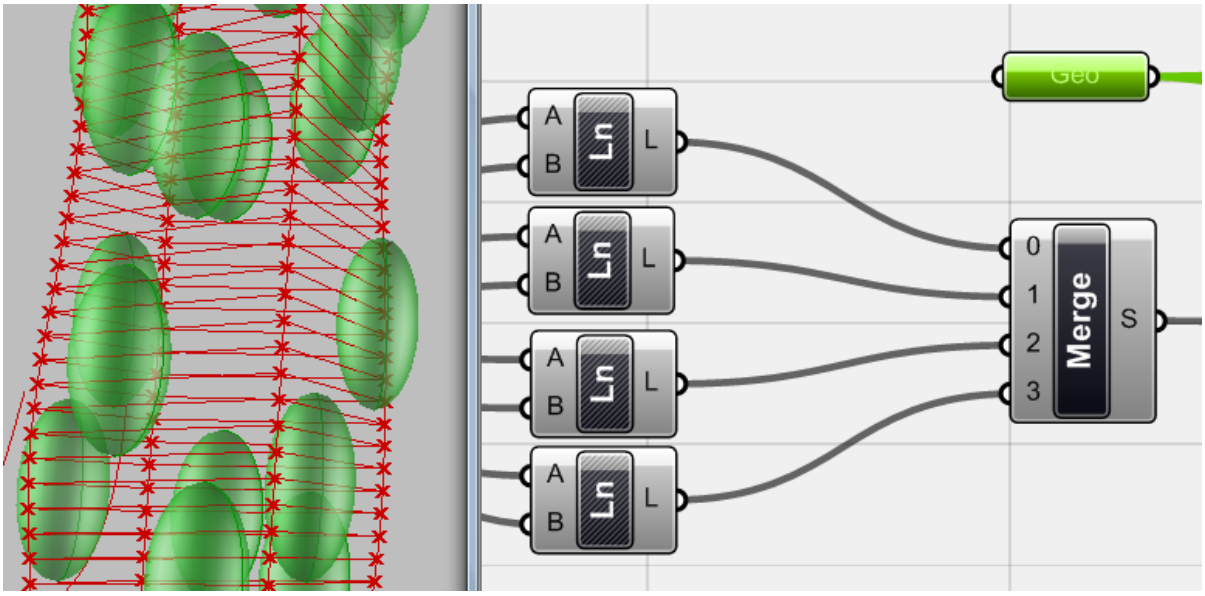


Fig.7.6. In this step I added my hollow spaces for the façade with random distribution. They are ellipsoids who introduced in Grasshopper all together by a <Geometry> component. I also <Merge>d all previously generated lines.

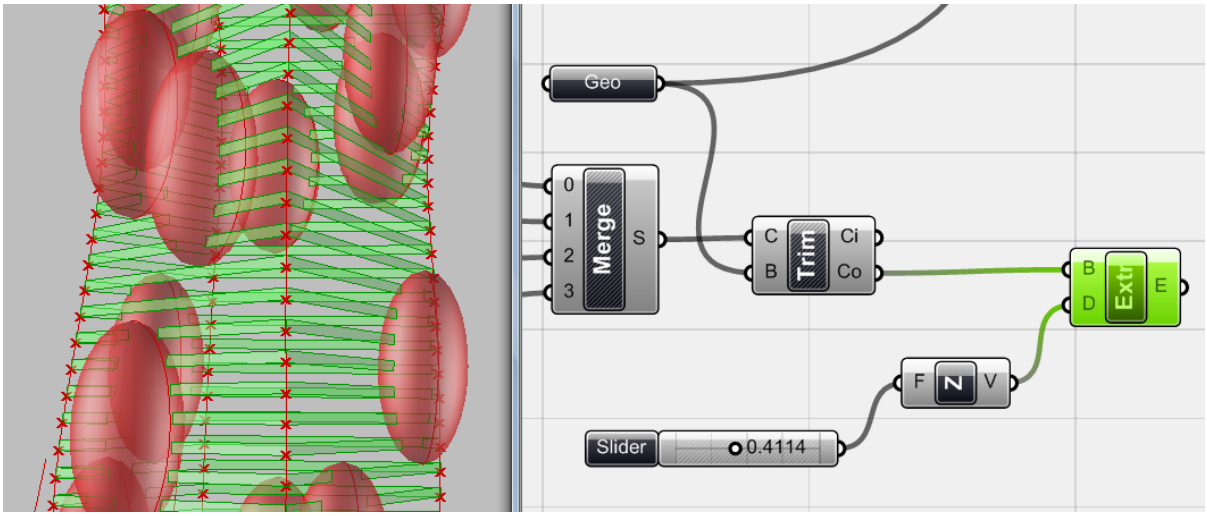


Fig.7.7. In this step, I <trim>ed all <merge>d lines with these <Geo>s. <Trim> component gives me the trimmed part, inside and outside of the trimming area, here I used the outside part. I extruded those parts as the linear façade elements.

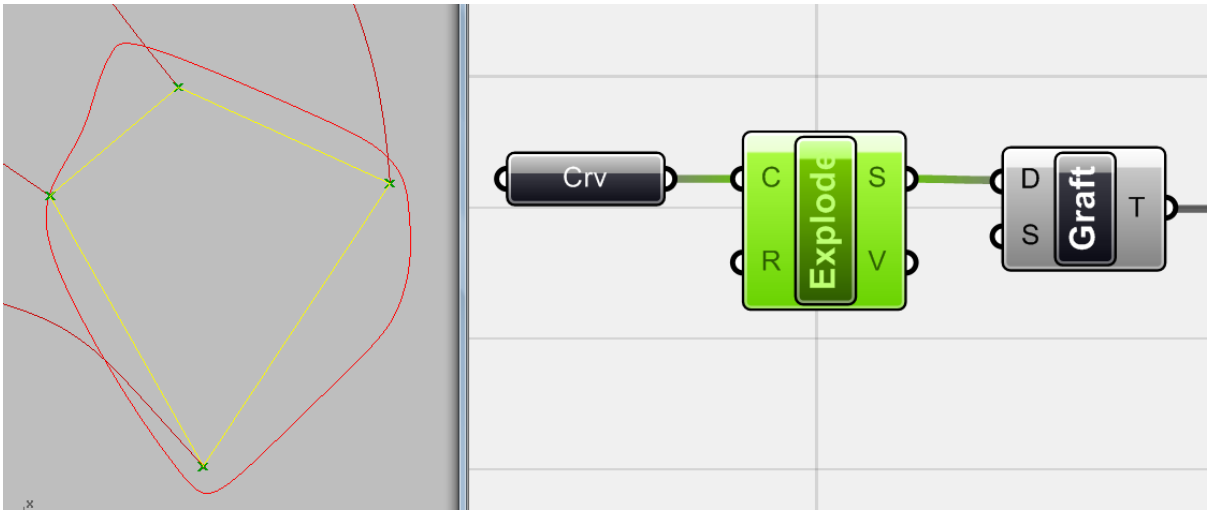


Fig.7.8. There is a closed <curve> that connects four corners of the tower in plan. Here because I need them to generate different surfaces, I <exploded> the curve to get its segments and I also used <graft> to generate one branch for each curve. Since I have planner section curve and two edge curves that define the boundary of the façade on each face, I want to use a <Sweep 2> component to create façade surface by sweep 2 rail.

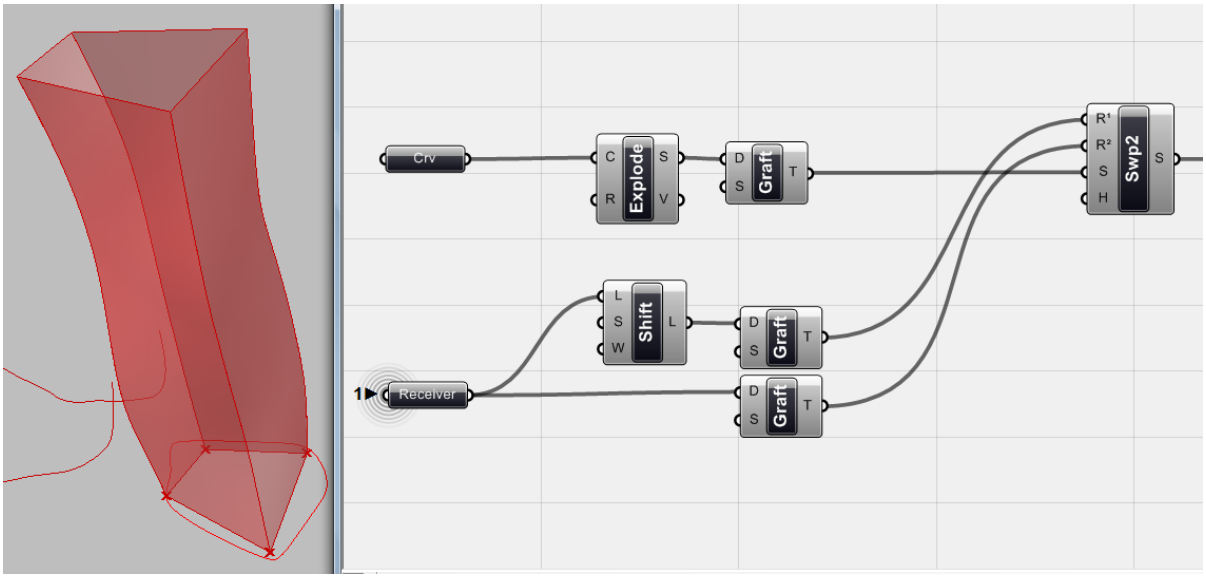


Fig.7.9. I introduced a <sweep 2> component to generate general façade surfaces. I used <graft>ed plan curves as Section curves for sweep command. Rails should be edge curves. The <receiver> component connected to the edge curves. I <graft>ed it once and I also <Shift>ed and <graft>ed it again to generate all first and second rail curves associated with plan Section curves.

!! Note: if you do not get the same result as illustrated, the order of your edge curves (Rails) is not associated with the order of your plan curves (Section curves) and you need to change the order of your edge curves in the list either manually by re-assigning them to the <curve> component by different order or by shifting the list in Grasshopper.

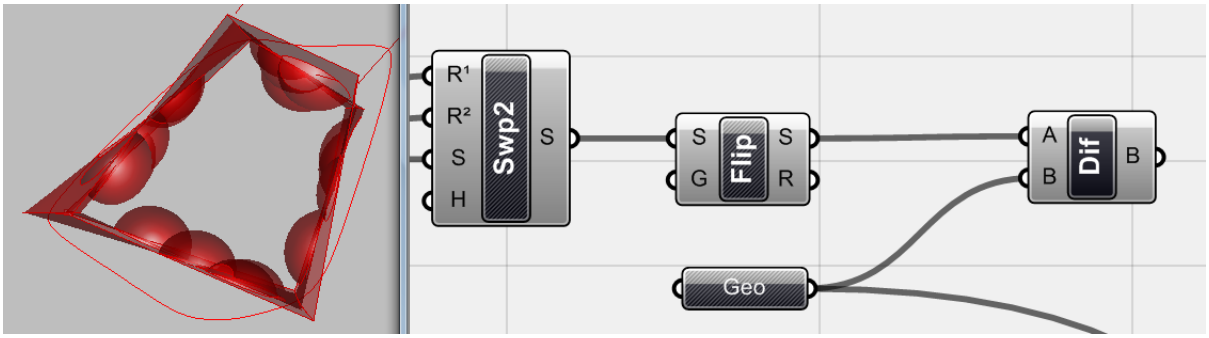


Fig.7.10. In this step I want to subtract hollow space <Geo>s from the previously generated façade surface. Since the normal direction of the surface is important in this command, I <Flip>ed the surface normal direction, and then I used a <Solid Difference> component (Intersect > Boolean > Solid Difference) and the result is the main surface while <Geo>s has been removed.

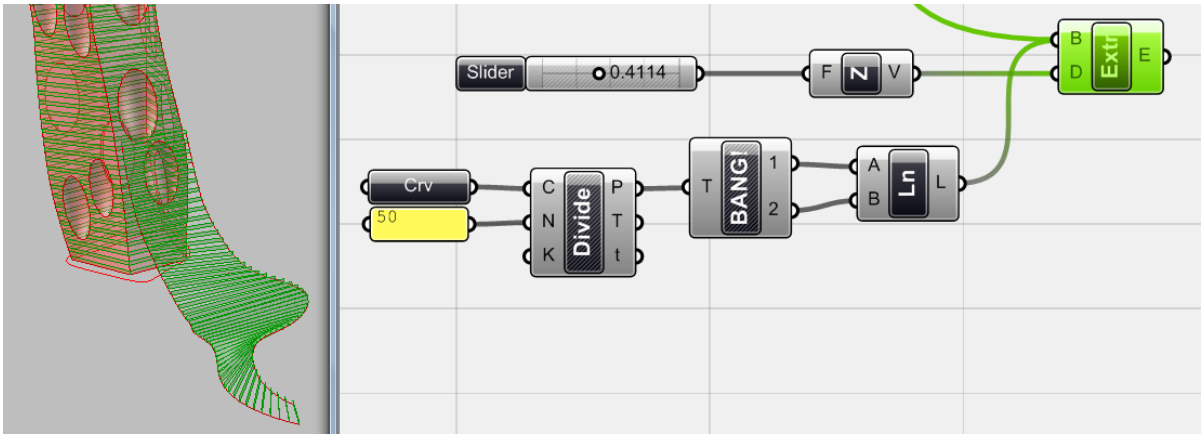


Fig.7.11. With the same method, I introduced both curves of the public space, divided them, exploded them and draw lines between them. I can connect this <line> component to the same <extrude> component as façade elements to generate identical geometries.

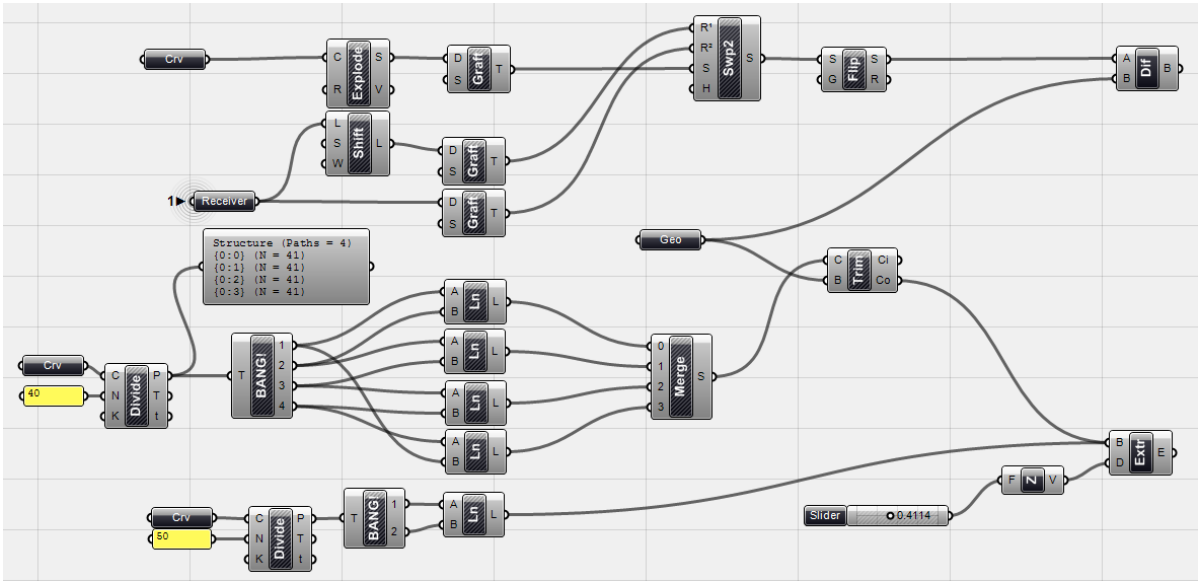


Fig.7.12. After generation of all geometries, bake <Difference> and <Extrude> components.

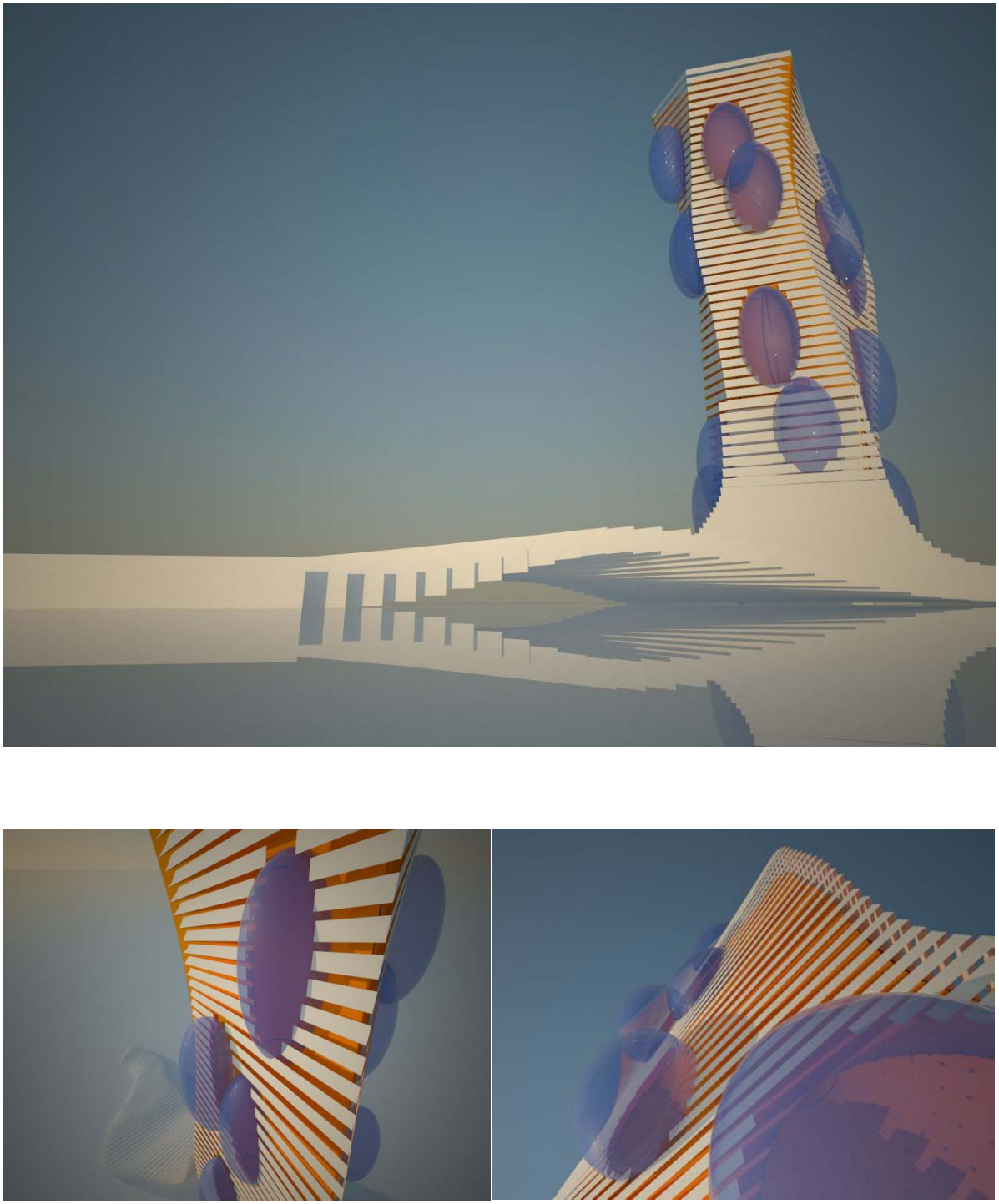


Fig.7.13. Final sketch model.

7_2_Geometry and Topology

Up to now we have used different components and worked with NURBS surfaces. But as mentioned before there are other types of surfaces which are useful in other contexts. It is not always the smooth beauty of NURBS that we aimed for, but we might need more precise control, easier processing or simpler equations. Beside the classical surface types of revolution, ruled or pipes, we have different free form surfaces like Besier or B-Splines. But here I am going to talk a little bit about Meshes which are different types of surfaces.

Meshes are another type of free-form surfaces but made up of small parts (faces) and accumulation of these small parts makes the whole surface. So there is no internal, hidden mathematical function that generates the shape of the surface, but these faces define the shape of the surface all together.

If we look at a mesh, first we see its faces. Faces could be triangle, quadrant or hexagon. By looking closer, we can see a grid of points which make these faces. These points are basic elements of a mesh surface. Any tiny group of these points (for example any three in triangular mesh) make a face with which the whole geometry become surface. These points are connected together by straight lines.

There are two important issues about meshes: position of these points and connection between these points. Position of points related to the geometry of mesh and connectivity of points related to the topology.

Geometry vs. Topology

While Geometry deals with the position of objects in space, Topology deals with their relations. Mathematically speaking, topology is a property of object that transformation and deformation cannot change it. So for instance circle and ellipse are topologically the same and they have only geometrical difference. Have a look at Figure 7.14 As you see there are four points which are connected to each other. In the first image, both A and B have the same topology because they have the same relation between their points (same connection). But they are geometrically different, because of displacement of one point. But in the second image, the geometry of points is the same but their connectivity is different and they are not topologically equivalent.

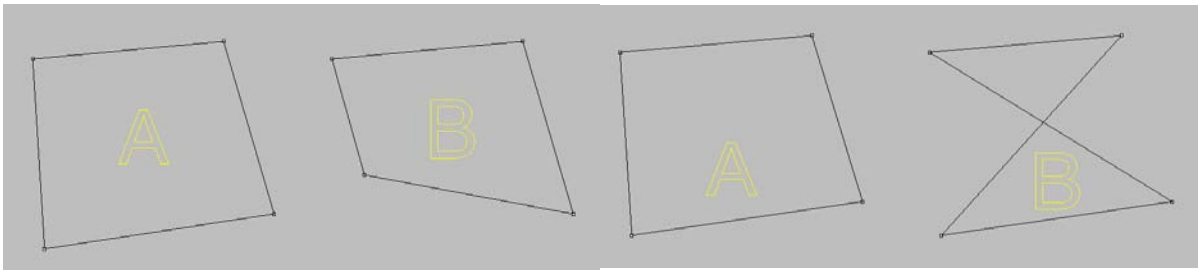


Fig.7.14. Topology and Geometry.

The idea of topology is very important in meshes. Any face in a mesh object has some corner points and these corner points are connected to each other with an order in a same way for all faces of the mesh object. So we can apply any transformation to a mesh object and displace vertices of the mesh in space even non-uniformly, but the connectivity of mesh vertices should be preserved to preserve faces otherwise it collapses.

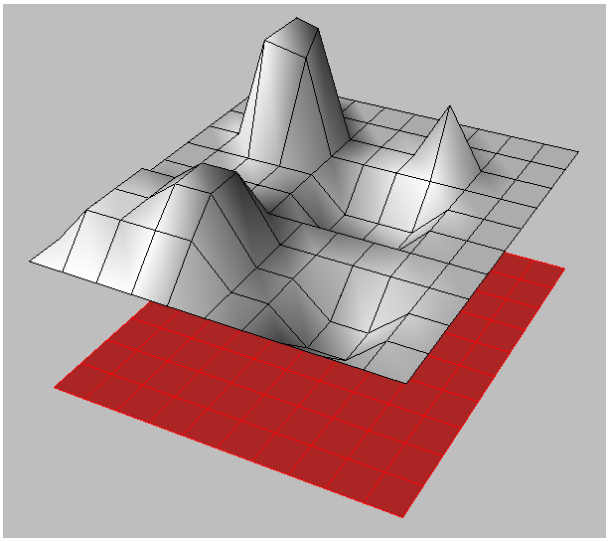


Fig.7.15. Both red and grey surfaces are meshes with the same faces and vertices, in the grey one, vertices are displaced, make another geometrical configuration of mesh, but connectivity of mesh object is not changed and both surfaces are topologically the same.

Knowing the importance of topological aspects of mesh objects, they are powerful geometries while we have bunch of points and we need a surface type to represent them as a continuous space. Different types of algorithms that work with points could be applied to a mesh geometry since we save the topology of the mesh. For instance, using finite element analysis or specific applications like dynamic relaxation, and particle systems, it is easier to work with meshes than other types of surfaces since the function can work with mesh vertices.

Mesh objects are simple to progress and faster to process; they are capable of having holes inside and discontinuity in the geometry. There are also multiple algorithms to refine meshes and make smoother surfaces. Since different faces could have different colours initially, mesh objects are good representations for analysis purposes (by colour) as well.

There are multiple components that deal with mesh objects in ‘mesh’ tab in Grasshopper. Let’s start a mesh from scratch and push the primary limits that we are facing.

7_3_On Meshes

I have a group of points and I want to create a surface by these points. In this example the group of points is simplified in a grid structure. I am thinking of a vertical grid of points that represent the basic parameters of a surface which is being affected by an imaginary wind pressure. I want to displace these points by wind factor (or any force that has a vector) and represent the resultant deformed surface. Basically by changing the wind factor, we can see how the resultant surface changes.

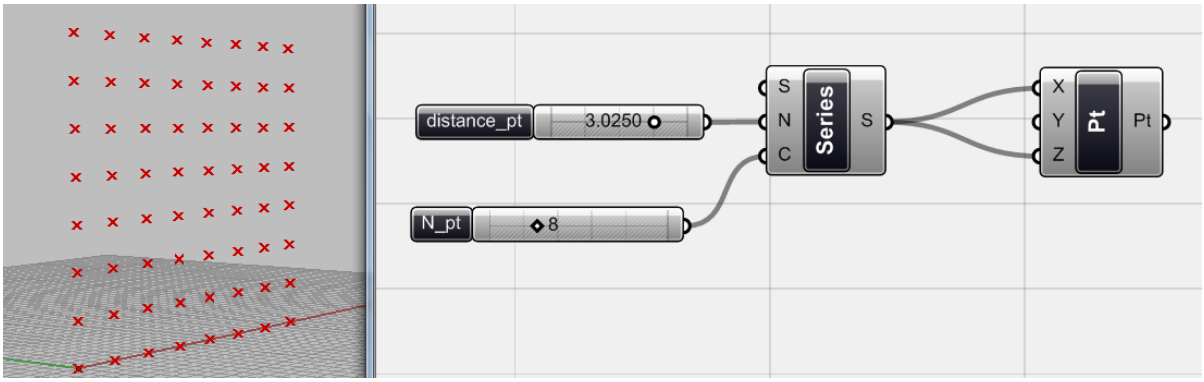


Fig.7.16. The first step is simple. By using a <series> component with controlled number of points <N_pt>, and distance between them <distance_pt> I generated a grid of cross referenced <point>s.

The pressure of the imaginary wind force, affects all points in the grid but I assumed that the force of wind increases when goes up, so the wind pressure becomes higher in bigger Z values of the surfaces. And at the same time, wind force affects the inner points more than the points close to the edges. Points on the edges in the plan section do not move at all (fix points).

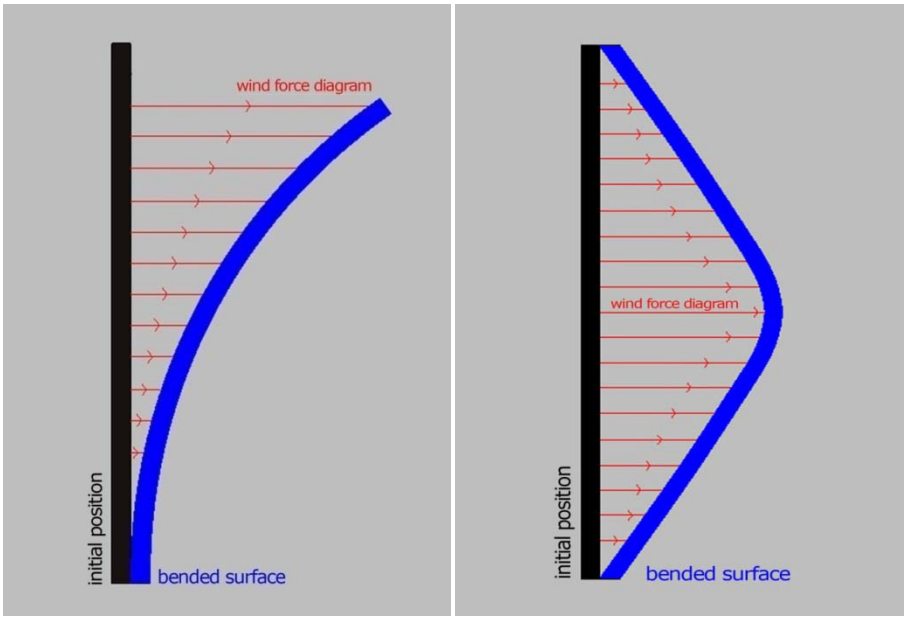


Fig.7.17. Diagram of the wind force affected the surface. A: section; the vertical effect of the force, B: plan; the horizontal effect.

Basically I need two different mechanisms to model these effects, one for section diagram and another for plan. I simplified equations just to mimic the way we want the force affects points. For the first mechanism the concept is a simple mathematical equation; I just used (X^2) while X is the Z value of the point being affected by the force. So for each point I need to extract the Z coordinate of the point.

To make everything simple, I assumed that the force direction is in the Y direction of the world coordinate system. So for each point on the grid, I need to generate a vector in Y direction and I set its force by the number that I receive from the Z coordinate of that point.

For the second diagram we need a bit more of an equation to do. Let's have a look at part one first.

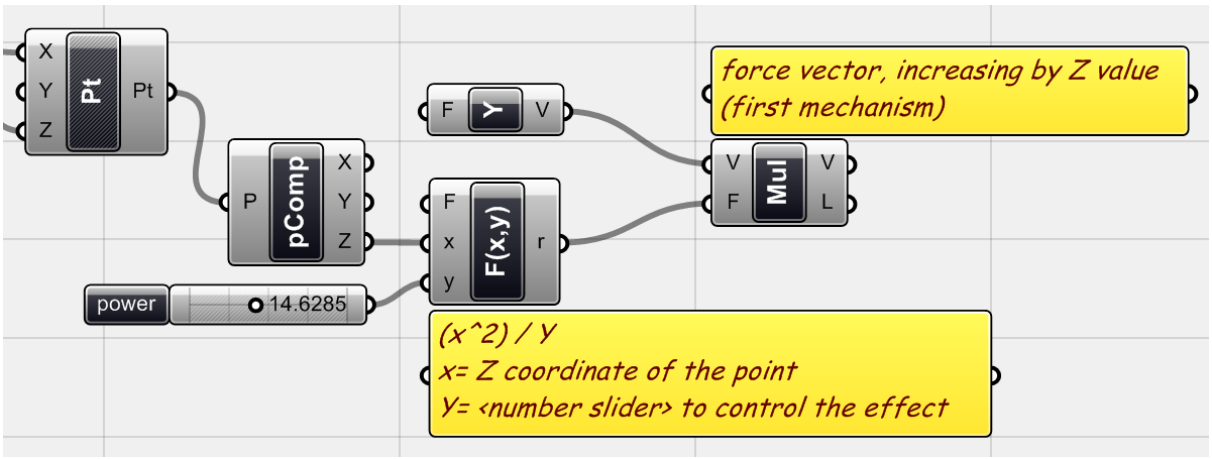


Fig.7.18. The Z coordinates of points extracted by a <decompose> component and then powered by (x^2) and divided by a given <number slider> just to control the general movement. The result is factors to <multiply> the force vector (Vector > Vector > Multiply) which is simply a world <unit Y> vector.

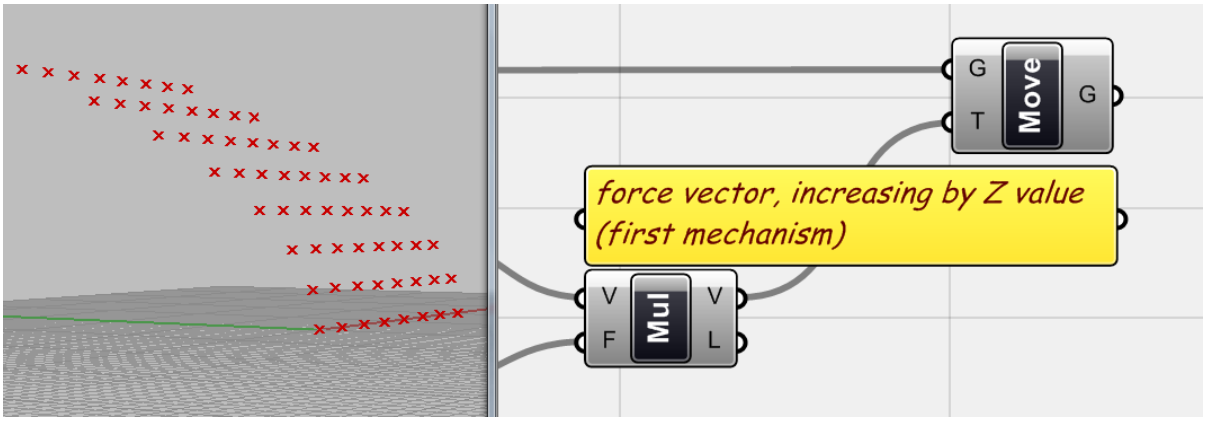


Fig.7.19. If I displace points by these vectors you can see the resultant grid of points that satisfy the first step of this task.

Now if we look at the second part of described forces, as I said, I assumed that in the planner section, points on the edges are fixed and points on the middle displace more than others. Figure 7.20 shows this displacement for each row of the point grid.

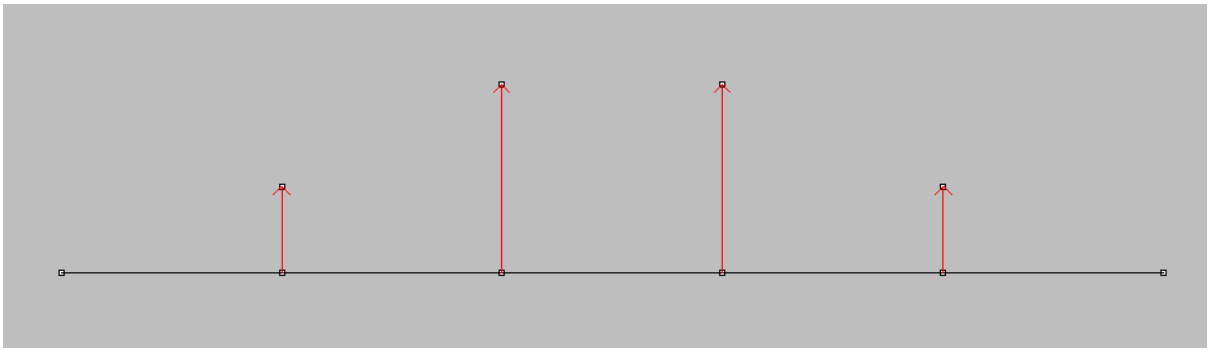


Fig.7.20. Displacement of points in rows (planner view).

Since I have force vectors for each point, I need to control them and set a value again, to make sure that their displacement in the planner section is also met the second criteria. So for each row of the points in the grid, I want to generate a factor to control the force vector's magnitude. Here I assumed that for points in the middle, the force vector's power are maximum that means what they are, and for points on the edges, it become zero means no displacement and for the other points a range in between.

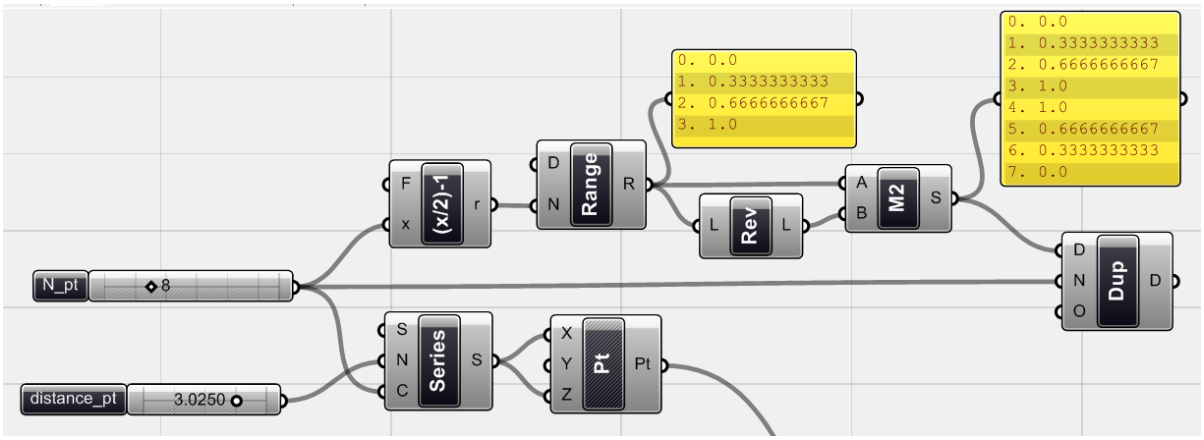


Fig.7.21. For the second mechanism, I need a <range> of numbers between 0 and 1 to apply to each point; 0 for the edge, 1 for the middle. I need a range from 0 to 1 from one edge to the middle and then from 1 to 0 to go from middle to other edge. I need this <range> component generates values as much as the number of points in each row.

I set the <N_pt> to the even numbers, and I divided it by 2, then minus 1 (because the <range> component takes the number of divisions and not number of values). You see the first <panel> shows four numbers from 0 to 1 for the first half of the points. then I <reverse>d the list and I merged these two lists together and as you see in the second <panel> I generated a list from 0 to 1 to 0 and the number of values in the list is the same as number of points in each row.

The final step is to generate these factors for all points in the grid. So I <duplicate>d the points as much as <N_pt> (number of rows and columns are the same). Now I have a factor for all points in the grid based on their positions in their rows.

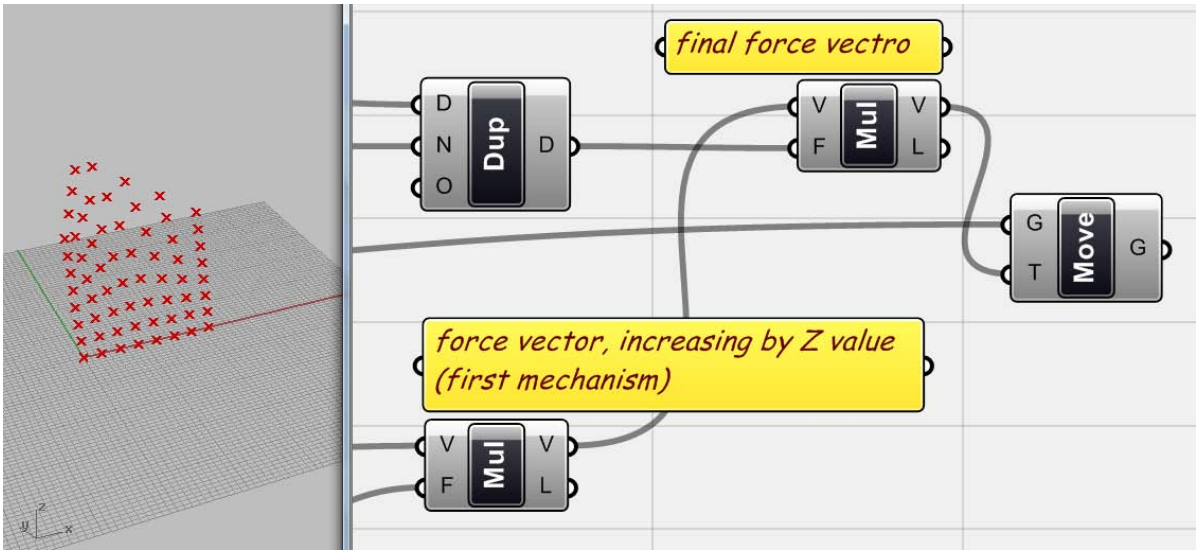


Fig.7.22. Now I need to <multiply> force vectors again by new factors. If I displace points by these new vectors, we can see how two different mechanisms affected the whole point grid.

Actually this part of the example needed a little bit of analytical thinking. In reality, methods like Particle Spring Systems or Finite Element Analysis, use the concept that multiple vectors affecting whole points in the set and points affecting each other as well. So when you apply a force, it affects all points and points affecting each other simultaneously. These processes should be calculated in iterative loops to find the resting position of the whole system. Here I just make a simple example without these effects and I just wanted to show a very simple representation of such a system dealing with multiple forces and I used very simple mathematical equations, which in real subjects are a bit more complicated! The idea is more about the mesh representation of this process, so let's go for mesh generation part.

Mesh

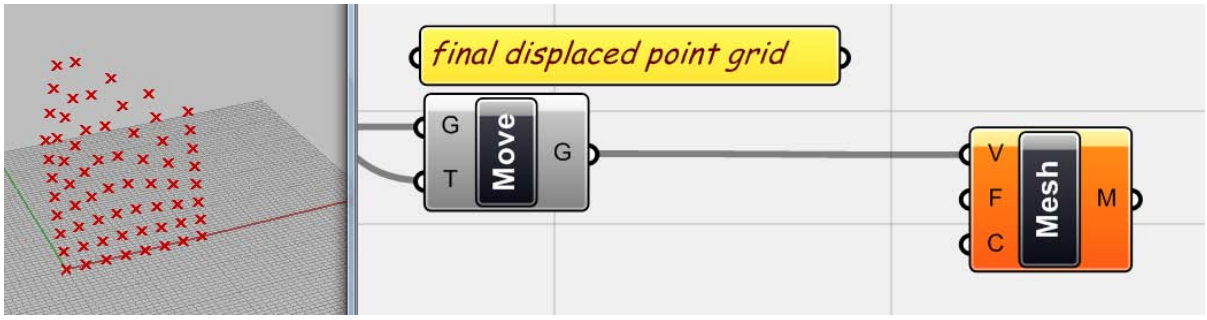


Fig.7.23. Mesh generating. Now if you simply add a <mesh> component (Mesh > Primitive > Mesh) to the canvas and connect displaced points to it as vertices, you will see that nothing happening in the scene. We need to define faces of the mesh geometry to generate it. Faces of mesh are actually series of numbers who just define the way these points are connected together to make the faces of each surface. So here vertices are geometrical part of the mesh but we need the topological definition of the mesh to generate it as well.

Every four corner point of the grid, define a quadrant face for the mesh object. If we look at the point grid, we see that there is an index number for each point in the grid. We know each point by its index number instead of coordinates in order to deal with its topology.

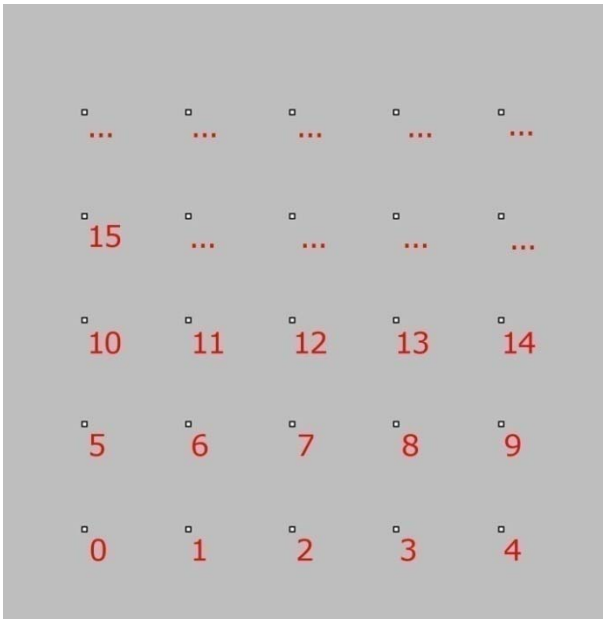


Fig.7.24. Index numbers of points in the grid.

To define mesh faces, we need to call every four corners that we assumed to be a face and put them together and give them to the <mesh> component to be able to make mesh surface.

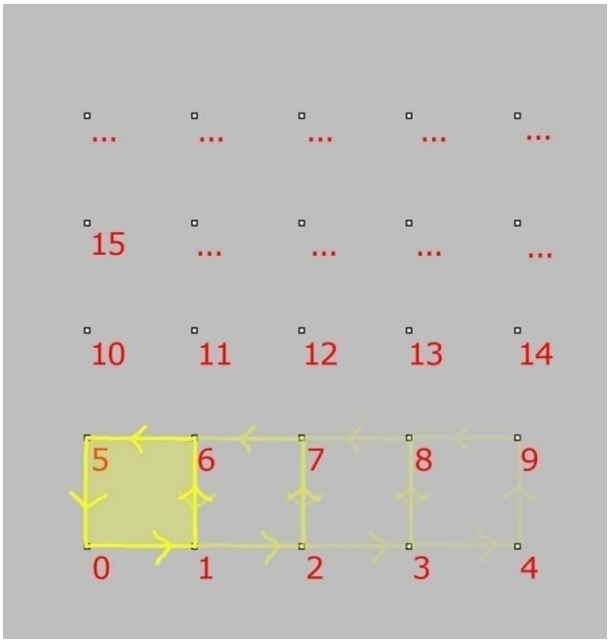


Fig.7.25. In a given point grid, a simple quadrant face defines by an order of points that if you connect them by a curve, you can make a face. This curve starts from a point in the grid, goes to the next point, then goes to the same point of the next row and then goes to the back column point of that row, and by closing this curve, you see the first face of the mesh finds its shape. Here the first face has points [0,1,6,5] in its face definition. The second face has [1,2,7,6] and so on.

To define the whole mesh faces, we should find the relation between these points and then make an algorithm that generates these face matrices for us.

If we look at the face matrix, we see that for any first point, the second point is the next in the grid. So basically for each point (n) in the grid, the next point of the face is (n+1). Simple!

For the next point of the grid, we know that it is always shifts one row, so if we add the number of columns (c) to the point index (n) we should get the point at the next row (n+c). So for instance in the above example we have 5 columns so c=5 and for the point (1) the next point of the mesh face is point (n+c) means point (6). So for each point (n) as the first point, the third point would be (n+1+c). That's it.

For the last point, it is always stated in one column back of the third point. So basically for each point (n+1+c) as the third point, the next point is (n+1+c-1) which means (n+c). So for instance for point (6) as the third point the next point becomes point (5).

All together for any point (n) in the grid, the face that starts from that single point has this points as the ordered list of vertices: [n, n+1, n+1+c, n+c] while (c) is the number of columns in the grid.

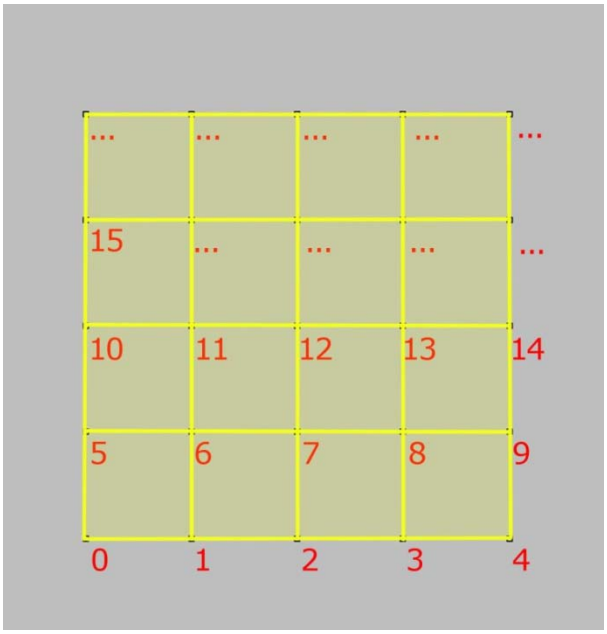


Fig.7.26. After defining all mesh faces, the mesh can be generated.

Looking at the mesh vertices, there is a bit more to deal with. If you remember the 'Triangle' example of chapter 3, there was an issue to select points that could be the first points in the grid. If you look at the grid of points in the above example, you see that points on the last column and last row could not be start points of any face. So beside the fact that we need an algorithm to generate faces of the mesh object, we need a bit of data management to generate the first points of the whole grid and pass these first points to the algorithm and generate mesh faces.

So basically in the list of points, we need to omit the points of the last row and last column and then start to generate face matrices. To generate the list of faces, we need to generate a list of numbers as index of points.

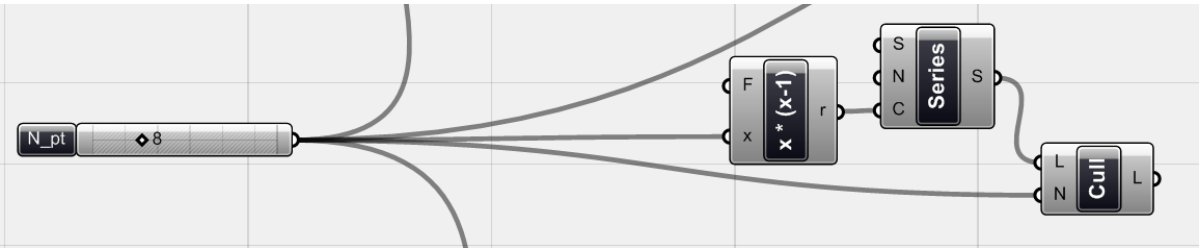


*Fig.7.27. Generating index number of the first points in the grid with a <series> component. The number of values in the series comes from the <N_pt> as the number of columns (same as rows) and by using a function of < x * (x-1)> I want to generate a series of numbers as <columns*(rows-1)> to generate the index for all points in the grid and omit the last row. The next step is to <cull> the index list by the number of columns (<N_pt>) to omit the index of points in the last column as well.*

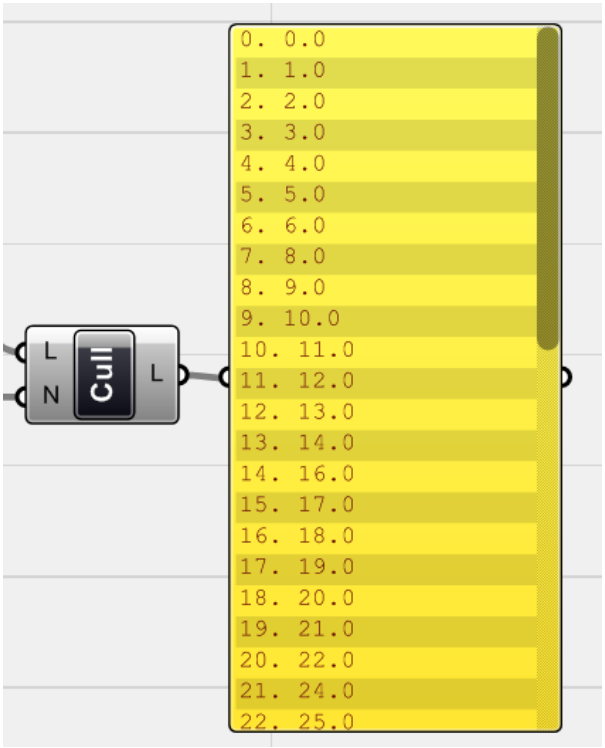


Fig.7.28. Final index number of the possible first points of mesh faces in the grid (with 8 points in each column).

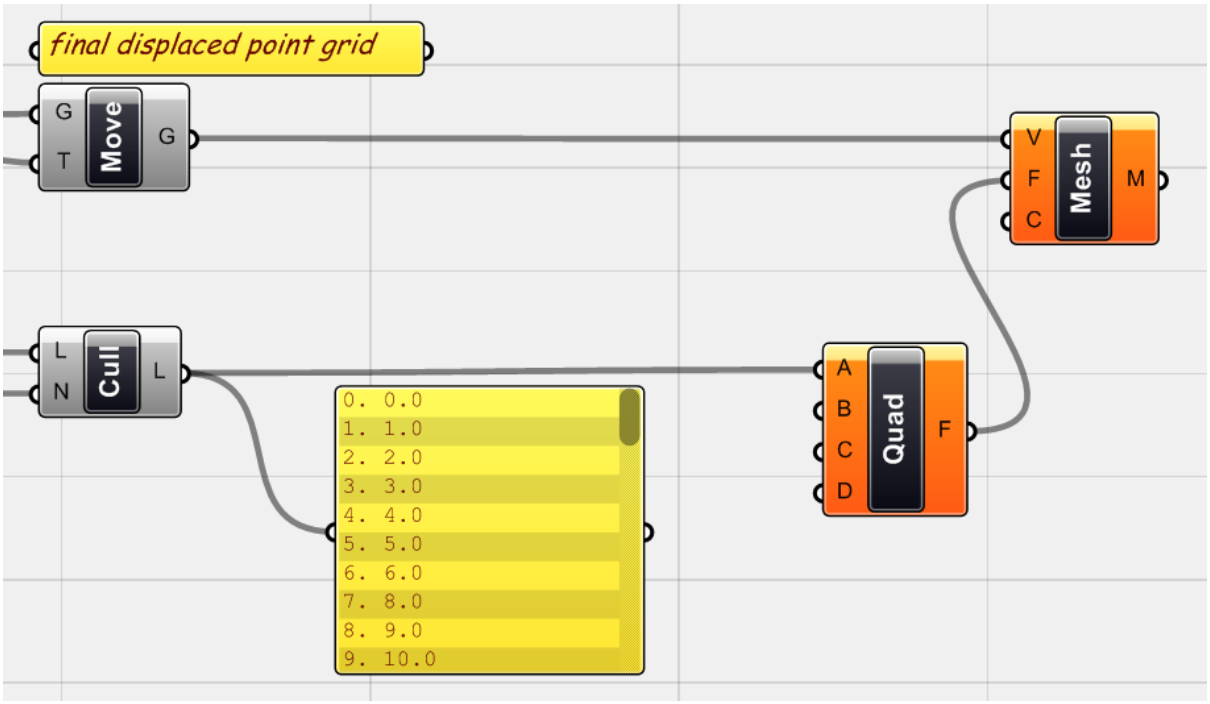


Fig.7.29. A <Mesh quad> component (Mesh > Primitive > Mesh quad) is in charge of generating faces in Grasshopper. I just attached the list of first numbers to the first point of the <quad>.

Now this is time to generate the list of indices for the faces:

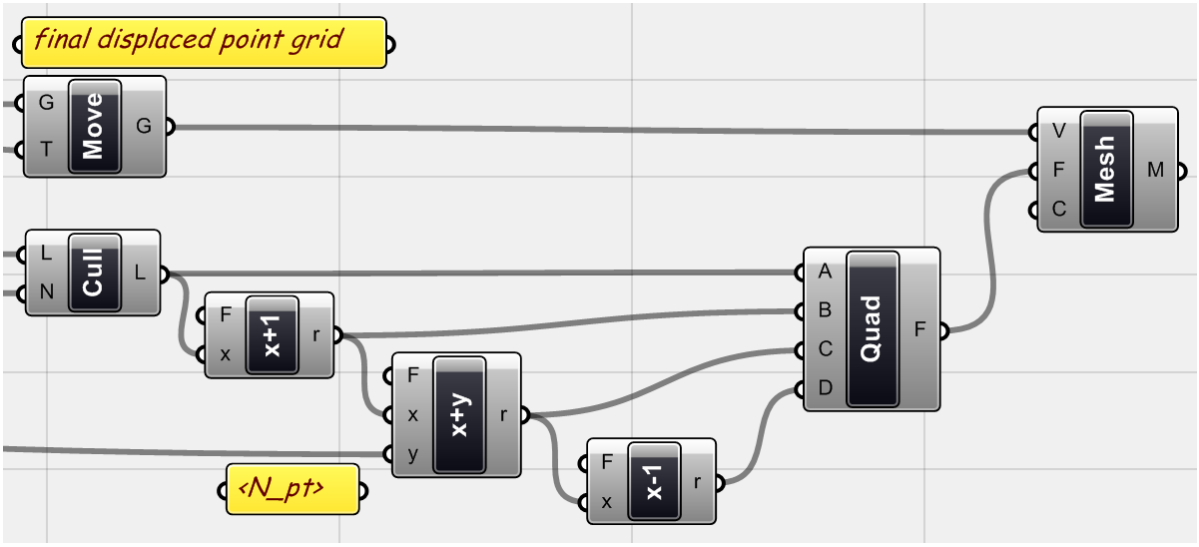


Fig.7.30. While (n) is the index of the first point and (c) is the number of columns, the second point is (n+1), the third point is ((n+1)+c) (the index of second point + number of columns), and the last point is ((n+1+c)-1) (the index of the third point -1).

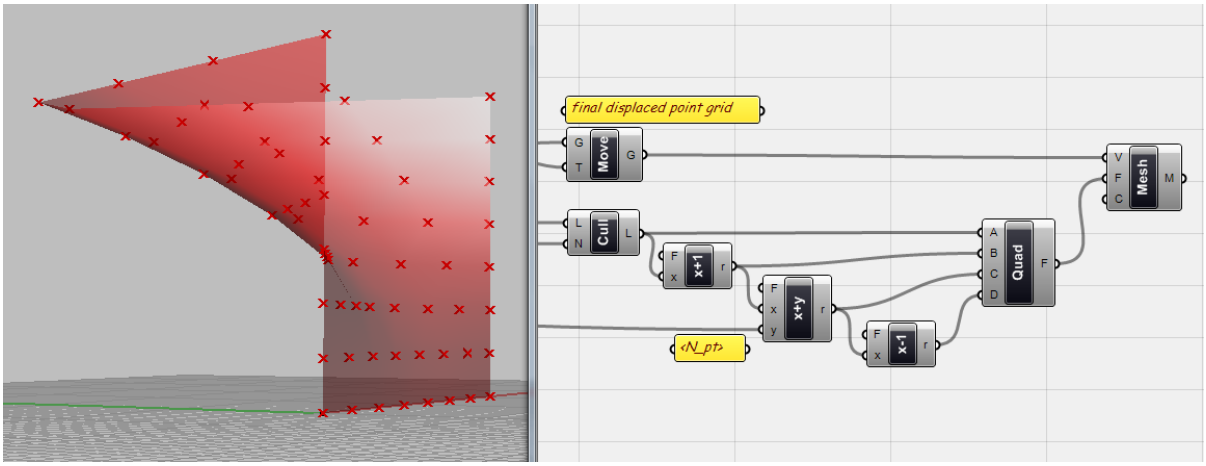


Fig.7.31. The resultant mesh.

7_4_On Colour Analysis

To finish this example, let's have a look at how we can represent our final mesh with colours as a medium for analysis purposes. There are different components in Grasshopper that provide us colour representations and these colours are suitable for our analysis purposes.

Here in this example, again to bring a concept, I simply assumed that at the end, we want to see the amount of deviation of our final surface from the initial position (vertical surface). I want to apply a gradient of colours start from points which remained fix with bottom colour up to points which has the maximum amount of deviation from the vertical position with the higher colour of the gradient.

Simply, to find the amount of deviation, I need to measure the final state of each point to its original state. Then I can use these values to assign colour to the mesh faces base on these distances.

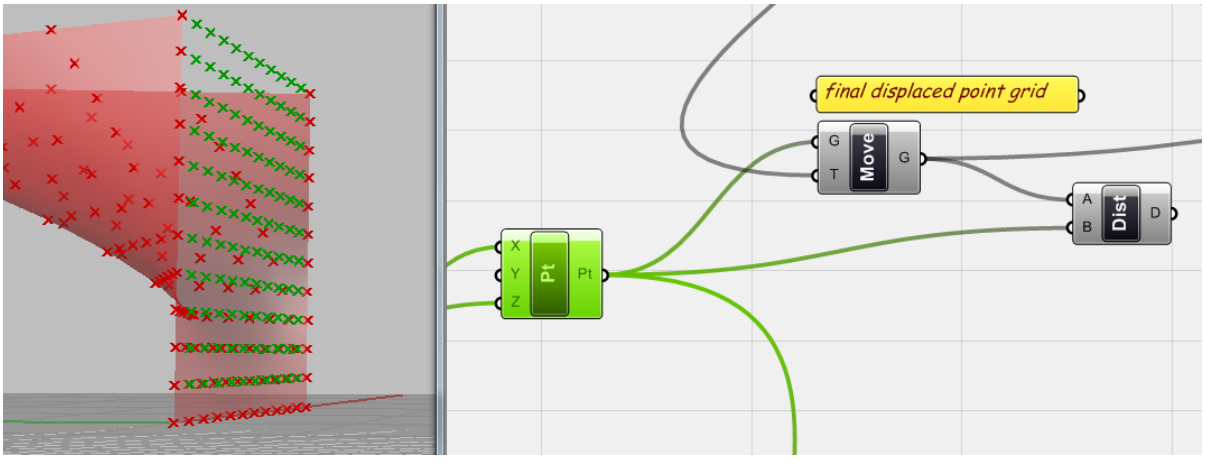


Fig.7.32. If I go back, I have the initial point grid that we generated in the first step and I also have the final displaced point grid that I used to generate the mesh vertices. I can use a <distance> component to measure distance between the initial position of points and their final position to see deviations of points.

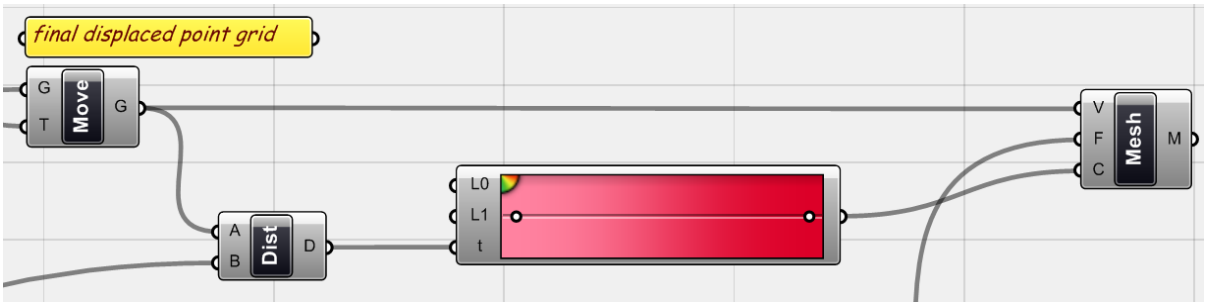


Fig.7.33. For our analysis purpose I want to use a <Gradient> component (Params > Special > Gradient) to assign gradient of colours to the mesh. I attached my <distance> values to the parameter part (t) of the <Gradient> and I attached it to the Colour input of the <mesh> component.

But to complete the process I need to define the lower limit and upper limit of gradient range (L0 and L1). Lower limit is the minimum value in the list and upper limit is maximum value in the list and other values are being divided in the gradient in between. To get the lower and upper limit of the list of deviations I need to sort the data and get the first and last values in that numerical range.

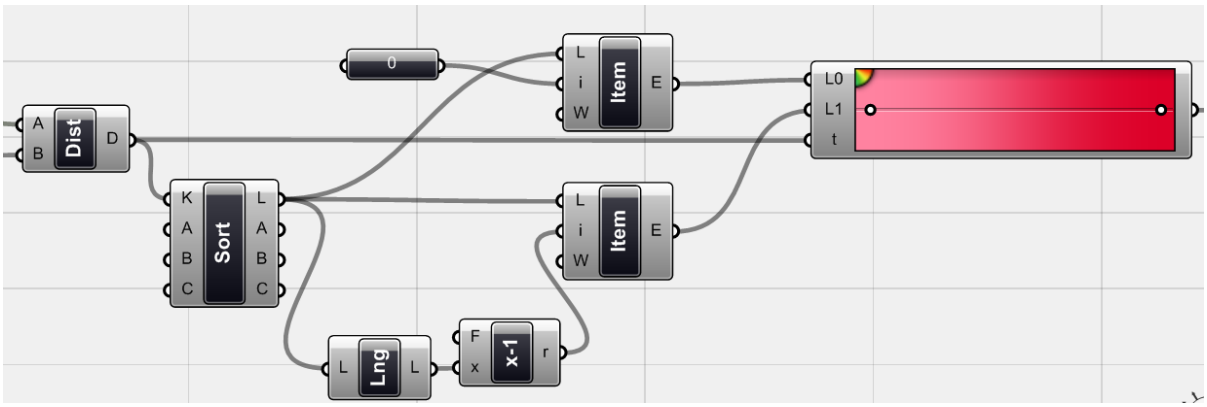


Fig.7.34. By using a <sort> component to sort distances, I get the first item of the data list (index= 0) as lower limit and the last one (index= <list length> - 1) as the upper limit of the data set (deviation values) to connect to the <gradient> component to assign colours based on this range.

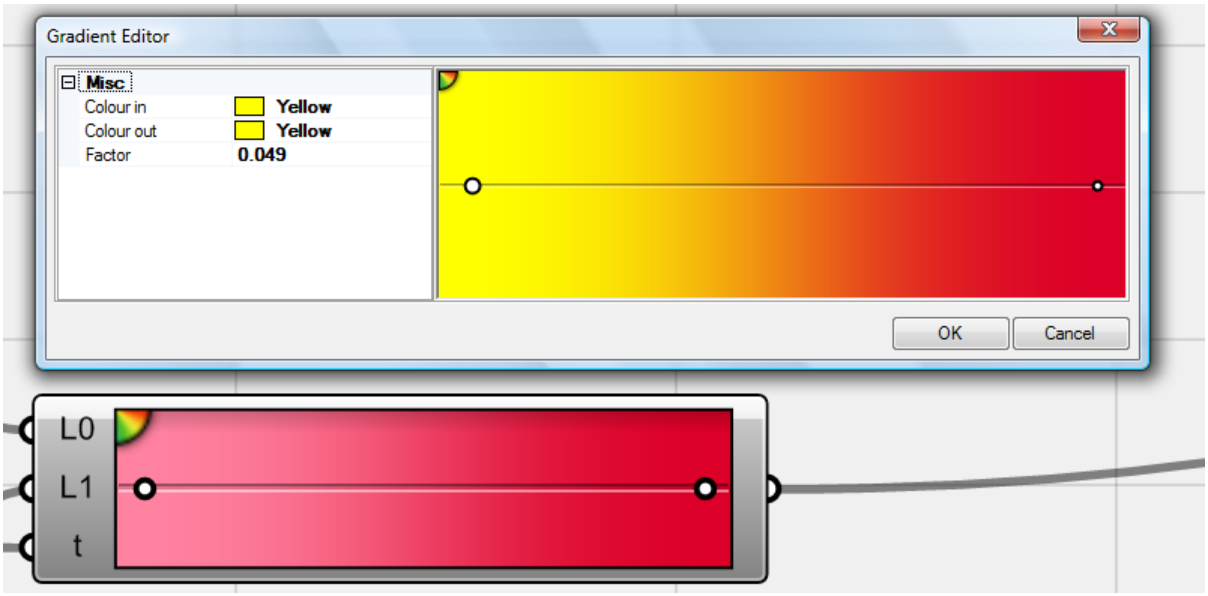


Fig.7.35. By clicking on the small colour icon on the corner of the <gradient> component we can change the colours of the gradient.

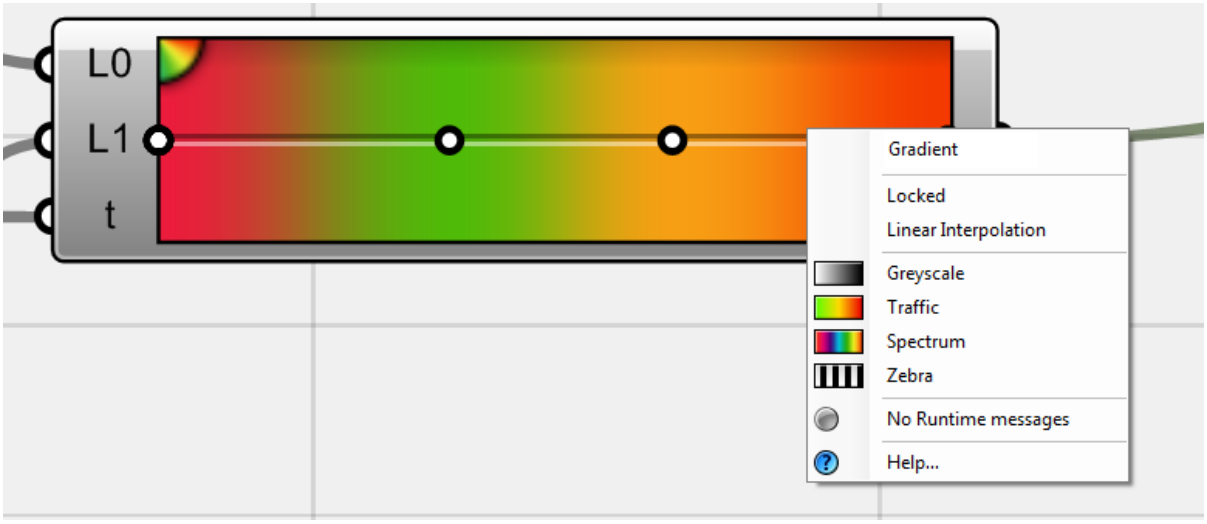


Fig.7.36. Right-click on the component and on the context pop-up menu you have more options to manipulate your resultant object, different types of colour gradients to suit the graphical representation of your analysis purpose.

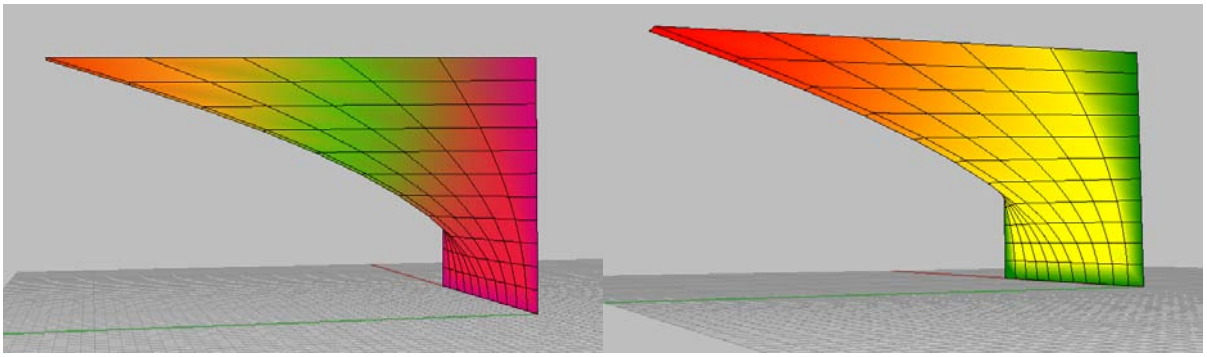


Fig.7.37. Different gradient thresholds.

7_5_Manipulating Mesh objects as a way of Design

Depends on the object and purpose of the modelling, I personally prefer to get my mesh object by manipulating a simple mesh geometry instead of generating a mesh from scratch since defining point set and face matrices are not always simple. By manipulating, I mean we can use a simple mesh object, extract its components and change them and then again make a mesh with varied vertices and faces. So I do not need to generate points as vertices and matrices of faces.

Let's have a look at a simple example.

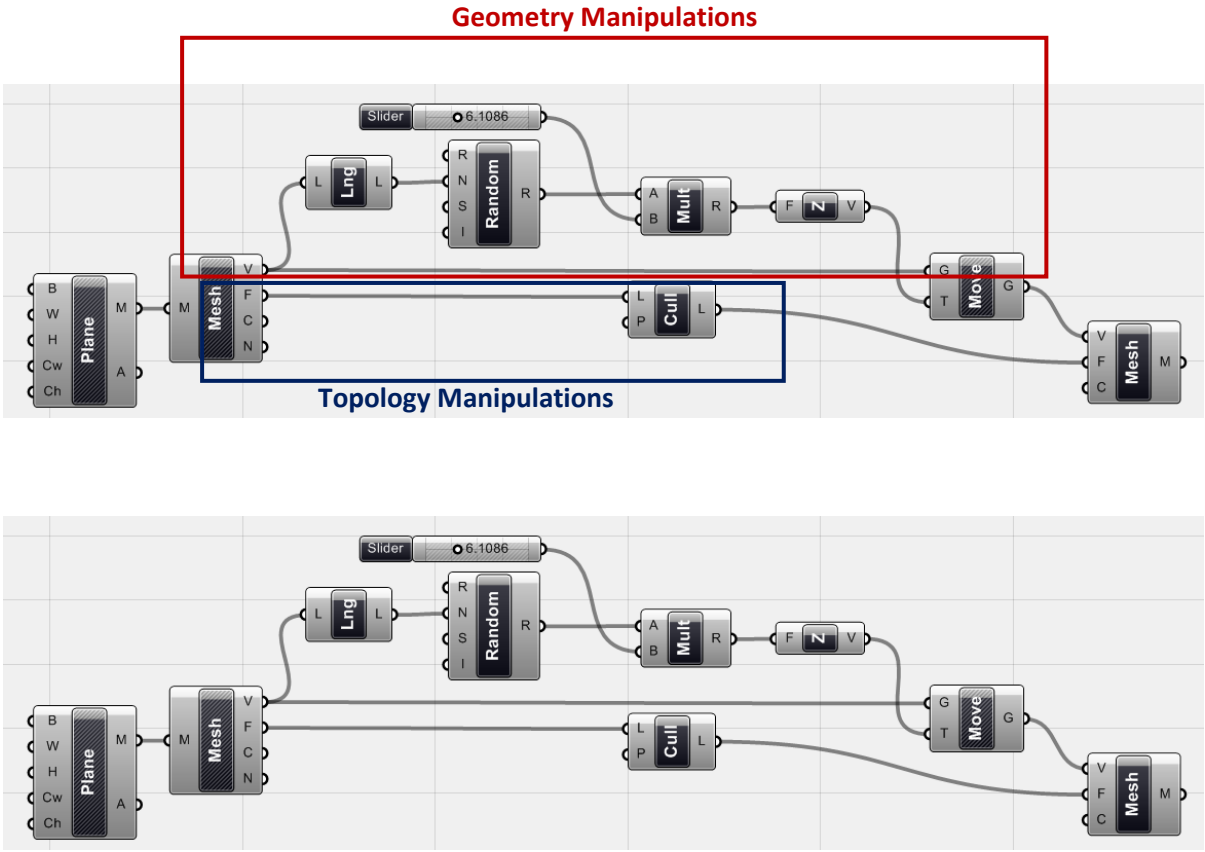


Fig.7.38. In this example, I simply used a <mesh plane> component and I extracted its data by using a <mesh components> to have access to its vertices and faces. Then I displaced vertices along Z direction by random values powered by a <number slider> and again attached them to a <mesh> component to generate another mesh. Here I also used a <cull pattern> component and I omitted some of the faces of original mesh and then I used them as new faces for making another mesh. The resultant mesh has both geometrical and topological difference by its initial mesh and can be used for other design purposes.

This idea of geometrically manipulating the vertices and topologically changing the faces has so many different possibilities that you can use in your design experiments. Since the mesh object has the potential to omit some of its faces and still it remains as a surface, the idea of making porous surfaces could be pursued with different ways.

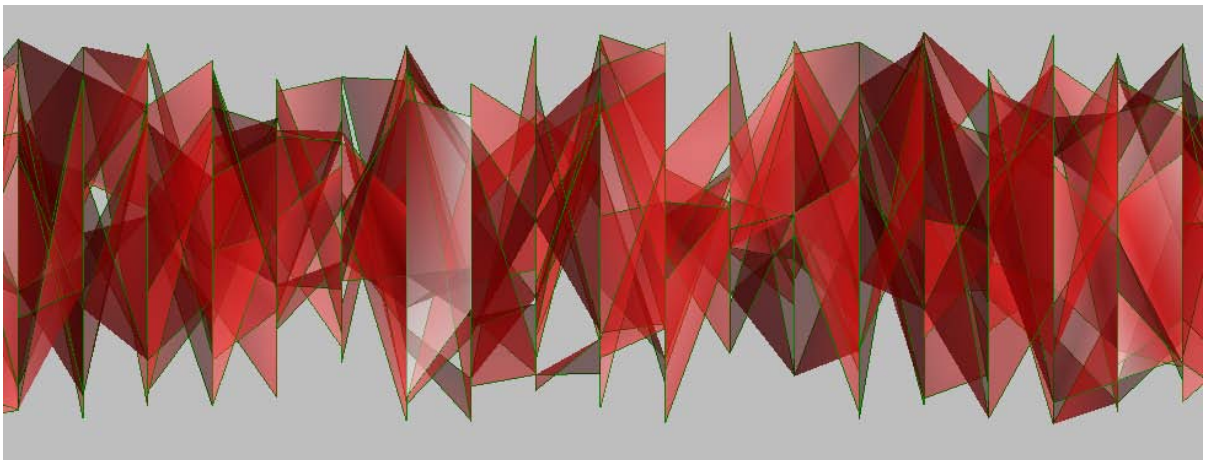


Fig.7.39. Resultant manipulated mesh (just a random case!).

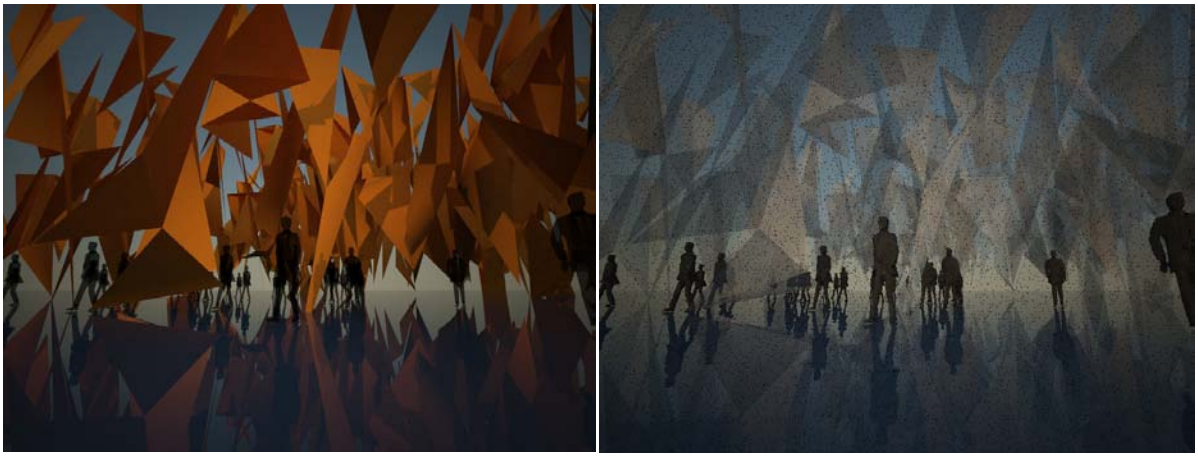


Fig.7.40. This is a sketch of a manipulated mesh!

Triangulation

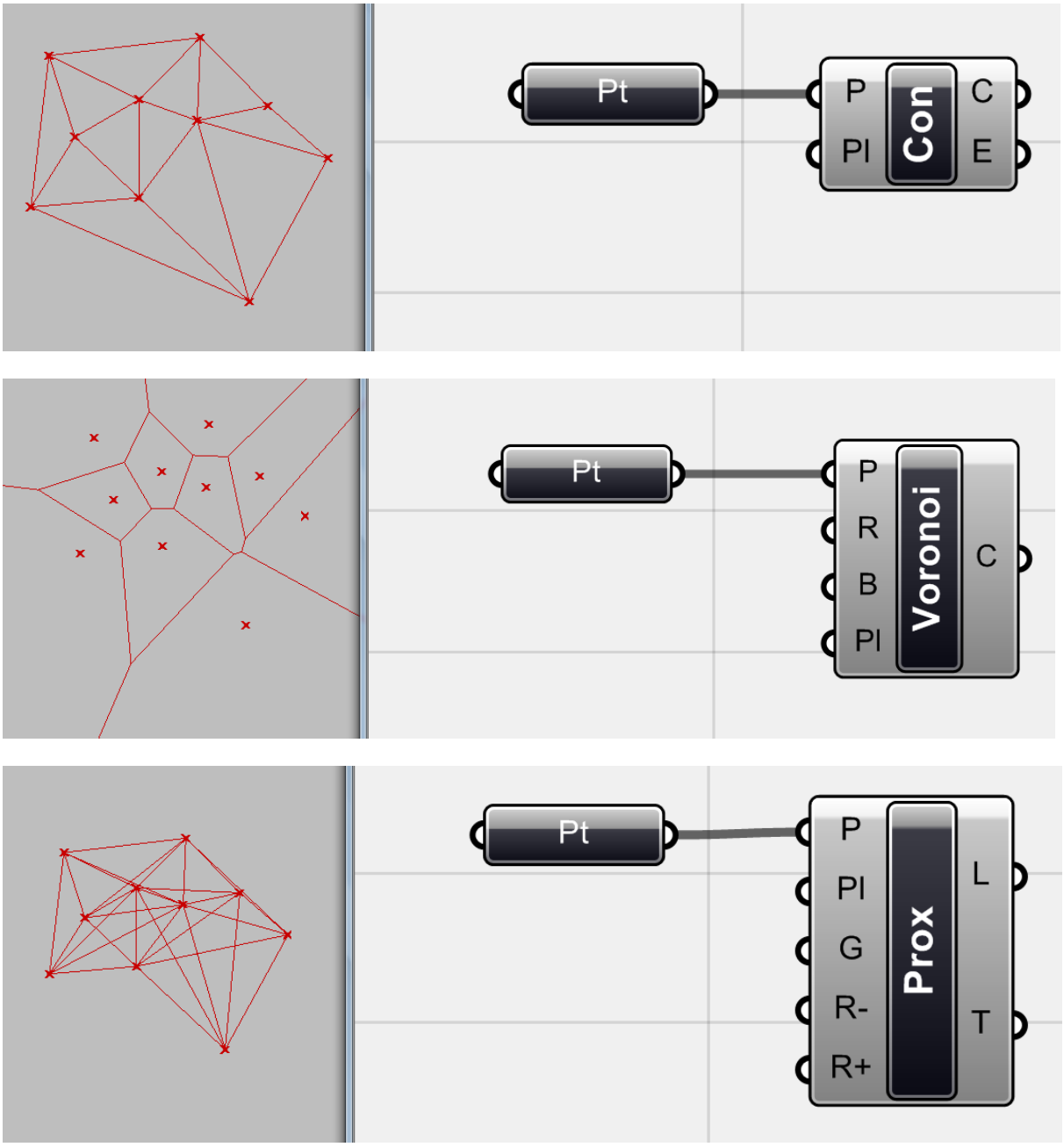


Fig.7.41. Examples of Triangulation components.

There are multiple components under the Triangulation panel in Mesh tab which provides useful algorithms like Delaunay or Voronoi or Convex Hull for design purposes. These internal algorithms could be useful to design complex objects and the most important point about them is that they use point sets to generate their output geometries. They are easy to explore and you can find lots of examples on-line and I don't want to go in-depth to describe them. So go ahead and explore them.

Chapter_8_Fabrication

Chapter_8_Fabrication

Today there is a growing interest on practice with Computer Aided Manufacturing and digital fabrication. Because of changes and new trends in design processes, it seems a crucial move and one of the 'Musts' in the field of design, to shift into the realm of digital fabrication. Any design decision in digital space, should be tested in different scales to show the ability of fabrication and assembly. Since it is obvious that the new design processes and algorithms do not fit into the traditional building techniques, designers now try to use modern technologies for fabrication and they adapt their design products to meet necessities. From the moment that CNC machines started to serve the building industry up to now, a great relation between digital design and physical fabrication have been made and many different technologies and machineries being invented or adjusted to do these types of tasks.

In order to design building elements and fabricate them, we need to have a brief understanding of fabrication processes for different types of materials and know how to prepare our design outputs for them. Based on the object we designed and material we used, assembly logic, transportation, scale, etc. we need to provide suitable data from our design product and get desired output, to feed machineries. If traditional way in realization of a project made by Plans, Sections, Details, etc. today, we need data and code to transfer to machines to fabricate a project.

The point here is that designer should involve in providing required data, because it is highly interconnected with design object. Designer sometimes should use the feedback of the fabrication-data-preparation for the design re-adjustment. Sometimes the design object should be changed in order to fit the limitations of the machinery or assembly.

Up to this point, we already know different potentials of Grasshopper to alter the design, and these design variations could be in favour of fabrication as well as other criteria we did. I just want to open the subject and touch some of the points related to the data-preparation phase, to have a look at possibilities of data extraction from design project for fabrication but we know that the subject is widely open for different techniques, machineries, materials, etc.

8_1_Datasheets

In order to prepare data to realize an object, sometimes we simply need a series of measurements, angels, coordinates and generally numerical data. There are multiple components in Grasshopper to compute measurements, distances, angels, etc. The important point is the correct and precise selection of points that we need to address for any specific purpose. We should be aware of any geometrical complexity that exists in design to choose desired positions for measurement purposes. The next point is to find positions that give us proper data for our fabrication purpose and avoid to generate lots of tables of numbers which could be time consuming in big projects but useless at the end. Finally we need to export data from 3D software to spreadsheets and datasheets and sometimes we need to manipulate this data in a way needed.

Paper Strip Project

The idea and technique of paper strips attracted me for some investigations. To understand the logic of assemblies I started with very simple combinations for first level and I tried to add these simple combinations together as the second level of assembly. It was interesting in the first tries but soon became out of order and the result was not what I assumed. So I tried to be more precise to deal with the complex geometries at the end.

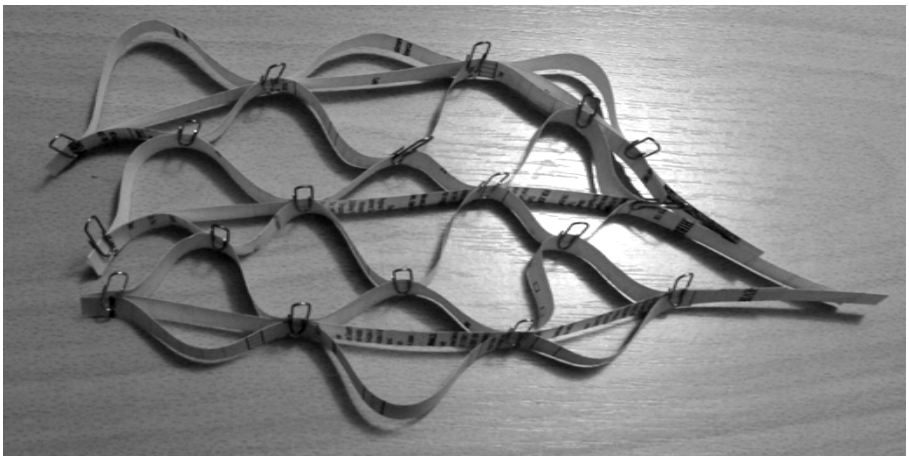


Fig.8.1. Paper Strips, first try.

In the next step I tried to start with a very simple set up and understand the geometrical logic and use it as the base for digital modelling. I assumed that by jumping into digital modelling I would not be able to make physical model and I was sure that I need to test the early steps with paper.

My aim was to use three paper strips and connect them, one in the middle and another two, in both sides of middle one, but with longer length, restricted at their ends to the middle strip. This could be the basic module to repeat and generate bigger assemblies.

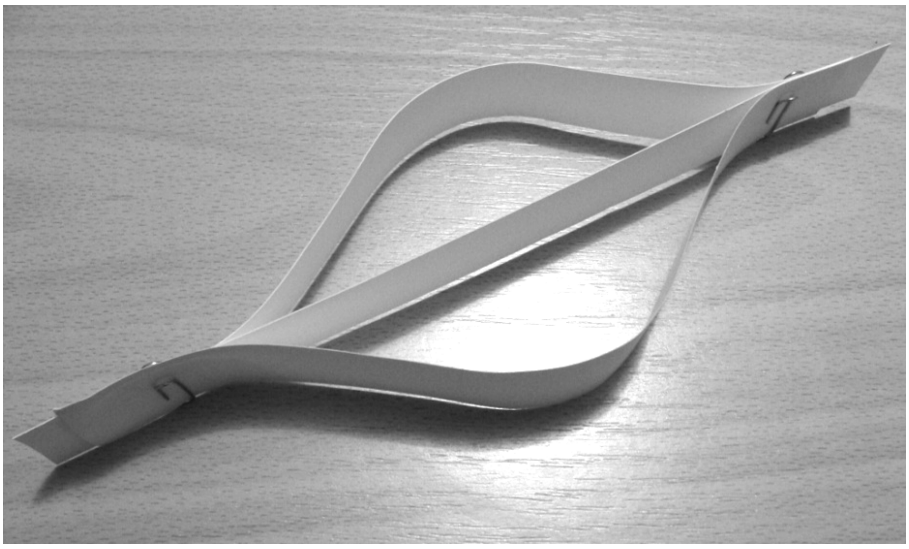


Fig.8.2. simple paper strip combination to understand connections and logic.

Digital modelling

Here I wanted to model the paper strip digitally, after my basic understanding of the physical one. From the start point I need a very simple curve in the middle as the base of my design and I can divide it and by culling these division points (false, true) and moving True ones perpendicular to the middle curve and using all these points (moved ones and false ones) as the vertices for two interpolated curves I can model this paper strips almost the same as what I described.

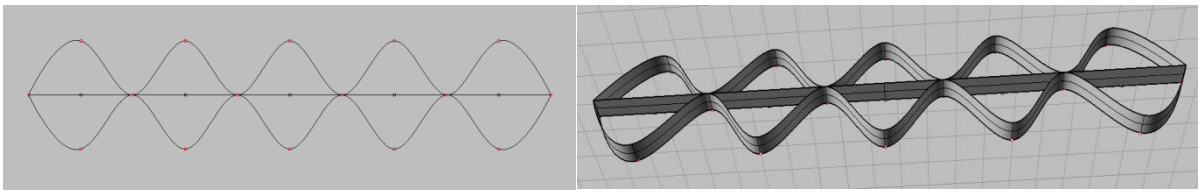


Fig.8.3. First modelling method with middle line and interpolated curves as side strips.

But it seemed so simple and straightforward. So I decided to add a gradual size-differentiation in connection points so it would result in a bit more complex geometry. Now let's move into Grasshopper and continue the discussion by modelling. I will try to describe the definition briefly and go to the data parts.

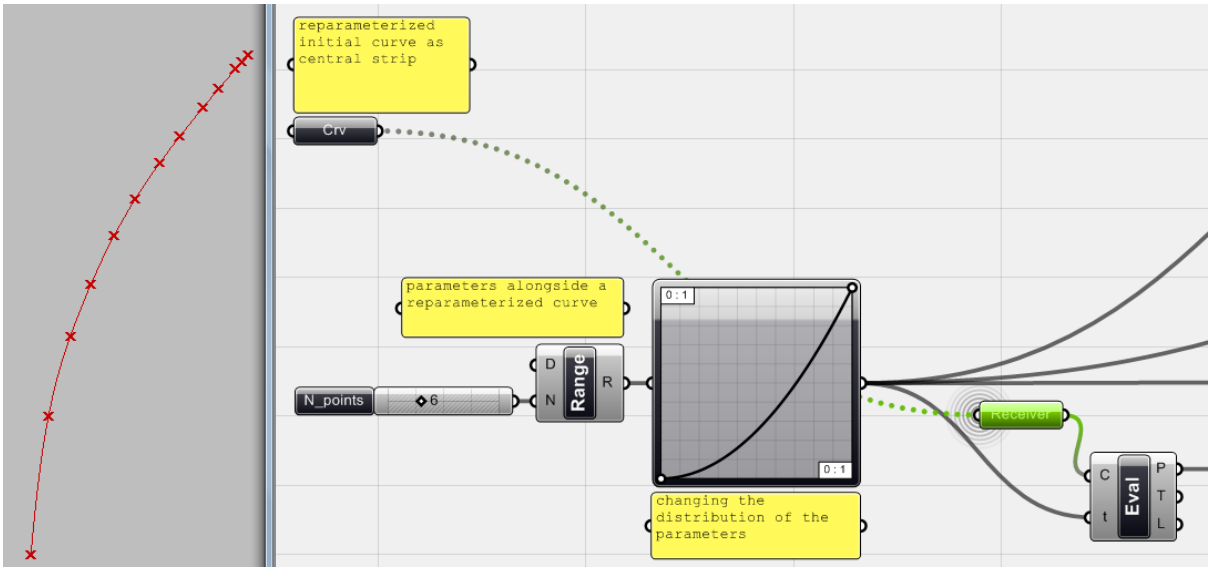


Fig.8.4. The <curve> component is the middle strip which is a simple curve in Rhino. I reparameterized it and I want to evaluate it in decreasing intervals. I used a <range> component and I attached it to a <Graph Mapper> component (Params > Special > Graph Mapper) to generate evaluation parameters. A <Graph mapper> remaps a set of numbers in many different ways by choosing a particular graph type. As you see, I evaluated the curve with this <Graph mapper> with parabola graph type and the resultant points on the curve are clear. You can change the type of graph to change the mapping of numeric range (for further information go to the component help menu). So I <Evaluate>d those parameters on the initial curve (<receiver> connected to the <curve>).

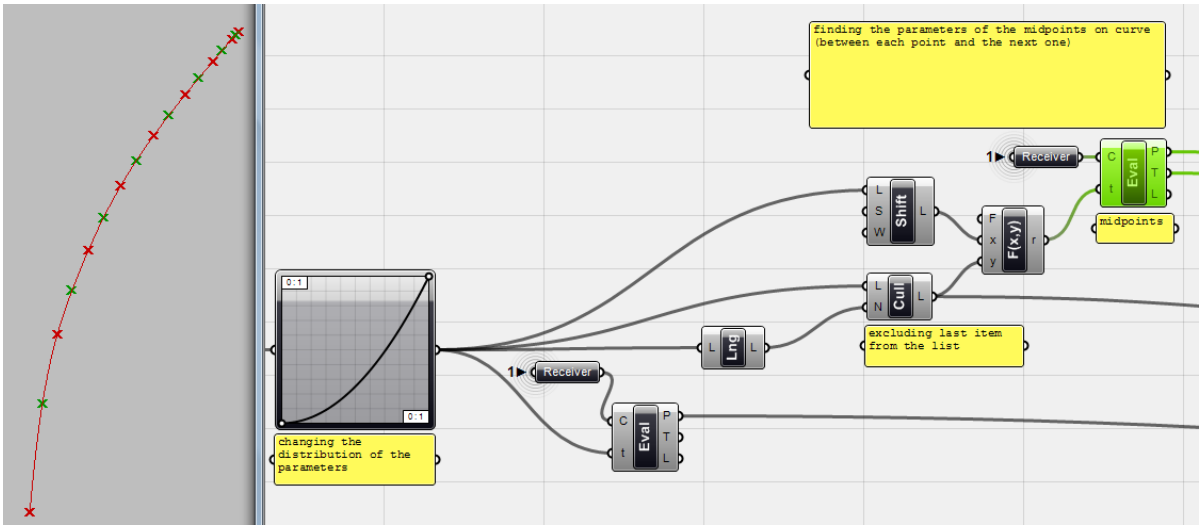


Fig.8.5. After remapping the numerical data and evaluating points, I want to find midpoints for every two points of previous set. Here I have to find the parameters of the curve, between each basic point and the next one, to evaluate. Since I have parameter of every first point, I <shift>ed the data to find next points. I also used <cull> with frequency of <list length> to exclude last item of the main list to have same items as <shift>ed list. The <function> component finds the parameter in between (f(x)=(x+y)/2) and you see the resultant parameters being evaluated (<receiver> connected to the <crv>).

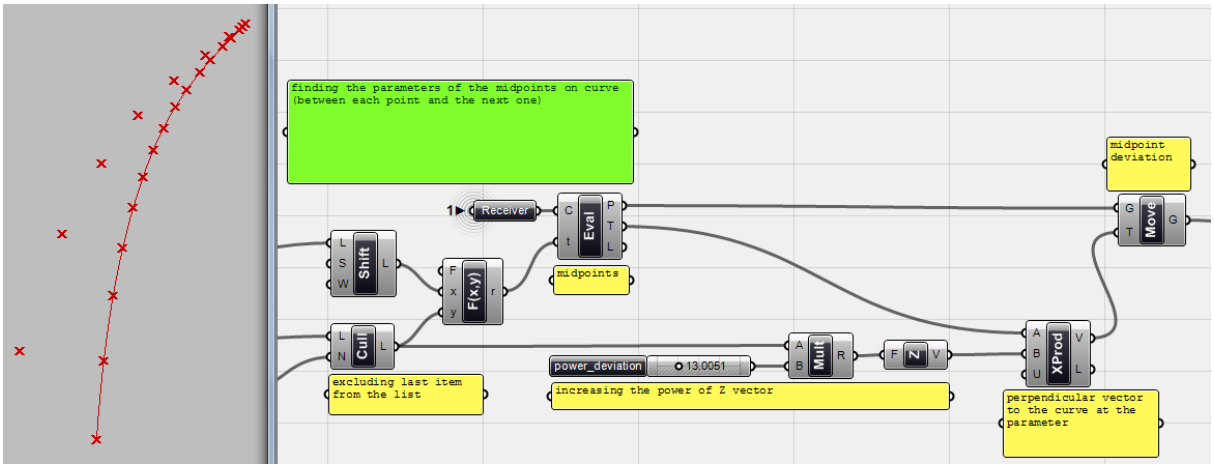


Fig.8.6. Now I want to move midpoints and make deviated vertices of the side strips. These points must move always in a perpendicular direction to the middle curve. So in order to move them, I need vectors, perpendicular to the middle curve at the position of each point. I already have the Tangent vector at each point, by <evaluate> component but I need the perpendicular vector.

*We now that a **Cross product** of two vectors is a vector always perpendicular to both of them (Fig.8.7). For example unit Z vector could be the cross product of unit X and Y vectors. Our middle curve is a planer curve so we know that the Z vector at each point of the curve would be always perpendicular to the curve plane. Tangent vector at each point of curve is situated at the plane of curve as well. So if I find the cross product of Tangent vector and unit Z vector at each point, the result would be a vector perpendicular to the middle curve which is always lay down on the curve's*

plane. I used Tangent of the point from <evaluate> Component and a <unit Z> vector to find the <XProd> of them which I know that is perpendicular to the curve even I manipulate it manually.

Another trick! I used the numbers of <Graph Mapper> as power factors of these Z vectors to have increasing factors for movements of points as well, so the longer the distance between points, the bigger their displacements.

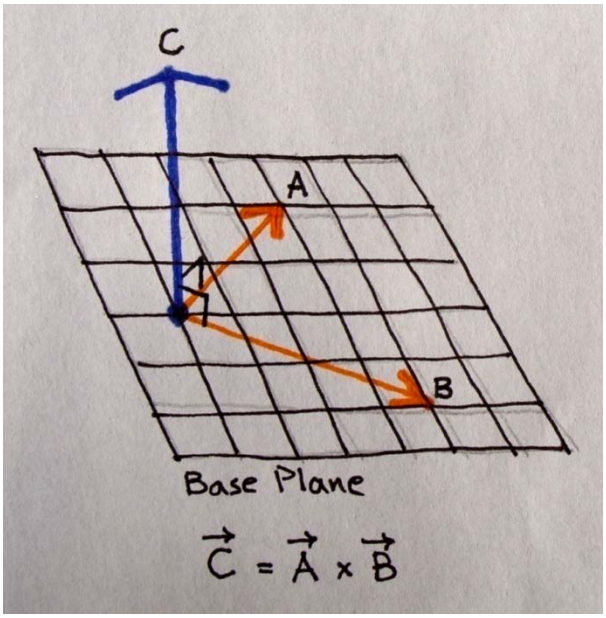


*Fig.8.7. **Vector cross product**. Vector A and B are in base plane. Vector C is the cross product of A and B and it is perpendicular to the base plane so it is also perpendicular to both vectors A and B.*

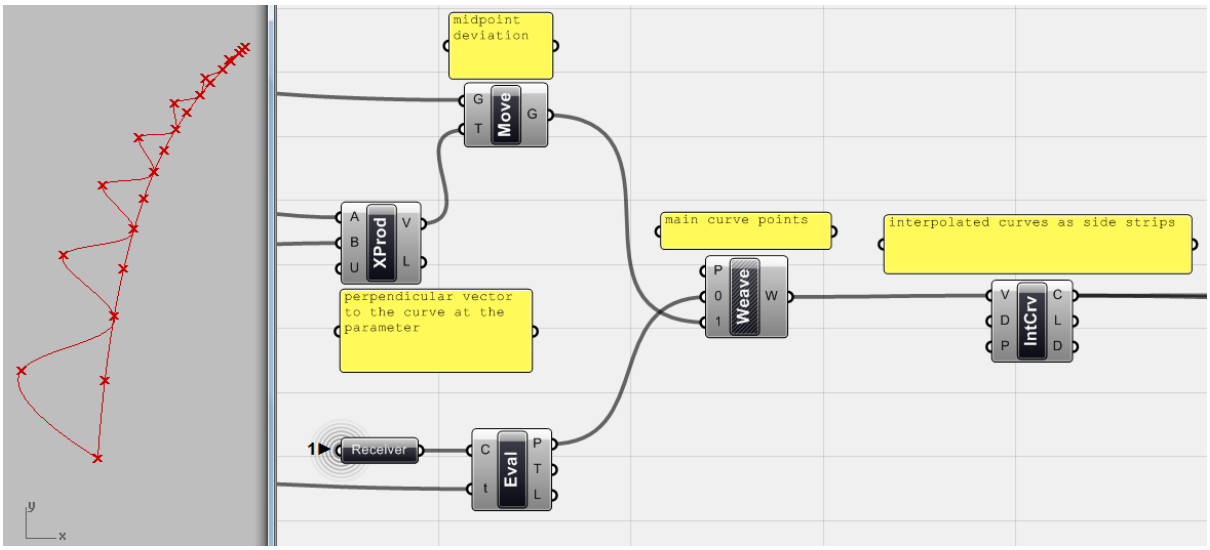


Fig.8.8. Now I have both basic points (first evaluated points) and moved points. I <weave>d them together to have a sorted list of data. Now if I use these points to generate an <interpolate>d curve, you see that the basic curve of the side strip is there.

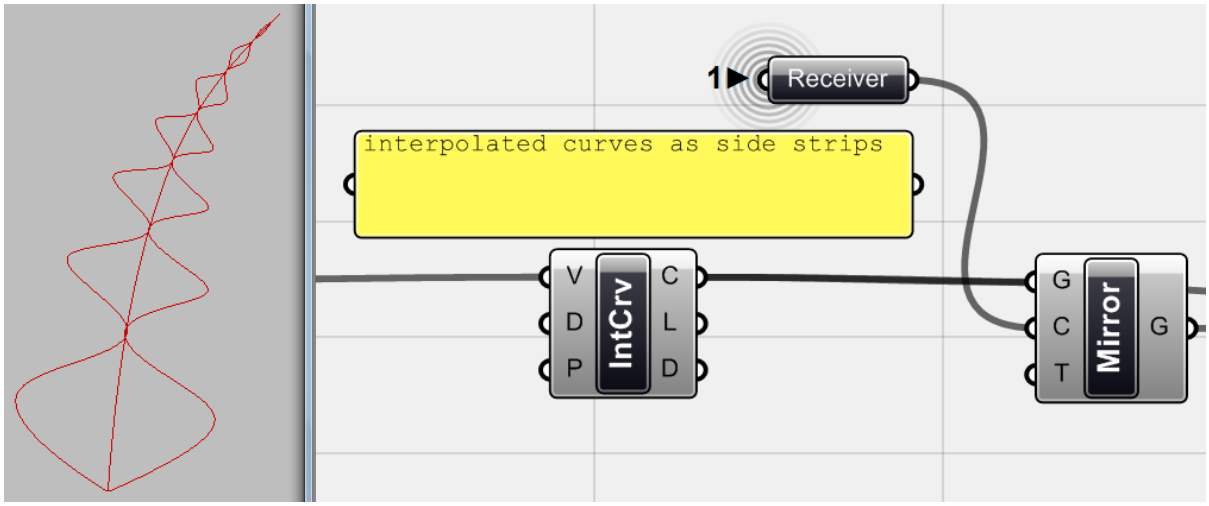


Fig.8.9. Using a <Mirror Curve> component (XForm > Morph > Mirror Curve) I can mirror the <interpolate>d curve by middle <curve> which is connected to the <receiver> so I have both side paper strips with the same concept.

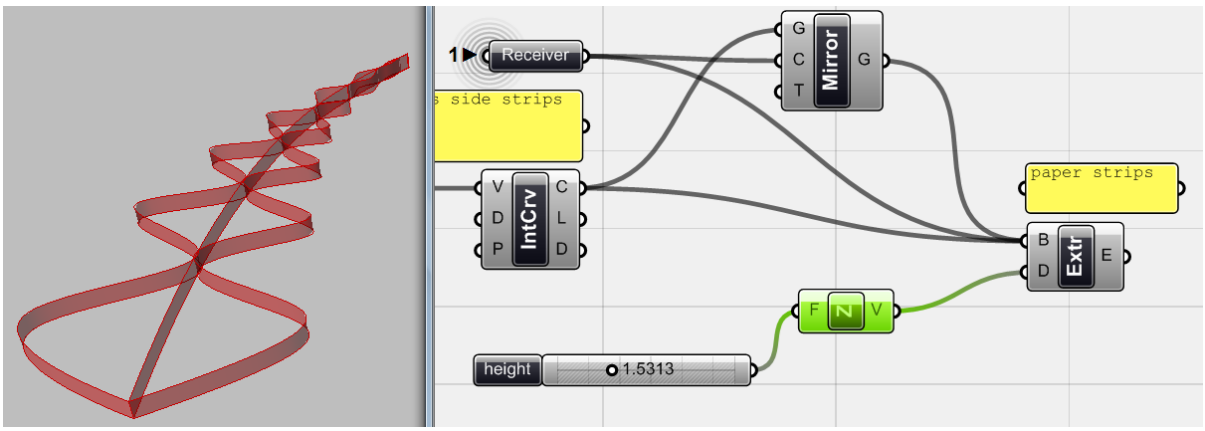


Fig.8.10. Now if I connect middle curve and side curves to an <extrude> component I can see my first paper strip combination with decreasing spaces between connection points.

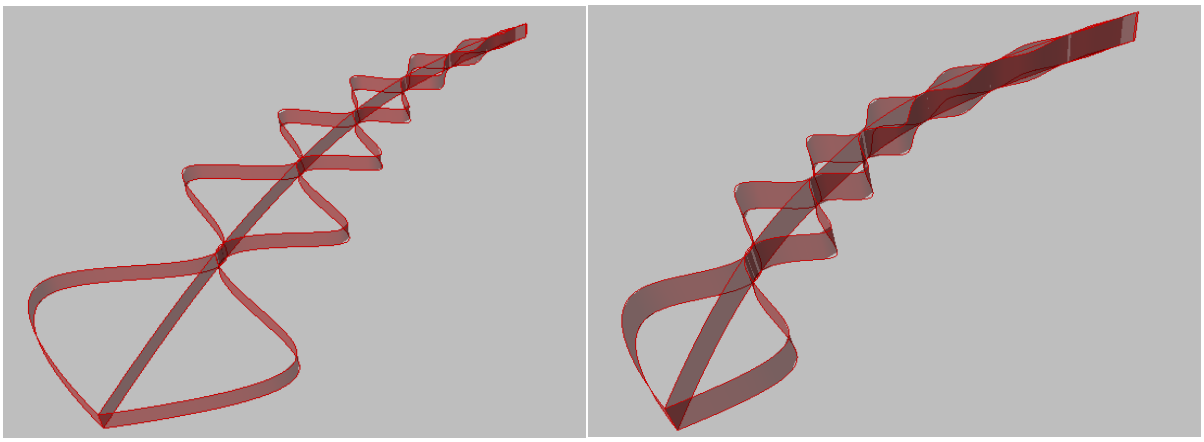


Fig.8.11. I can simply start to manipulate the middle strip and see how Grasshopper updates three paper strips which are connected to each other, or I can change my sliders and check the resultant geometry to select one, which is close to the physical model.

After I found a configuration that I want to create paper strip model with, I need to extract dimensions and measurements to build my model with that data. Although it is easy to model all these strips on paper sheets and cut them with laser cutter but here I like to make the process more general and get the initial needed data, so I am not limited myself to one specific machine and one specific method of manufacturing. You can use this data as generic manufacturing codes!!!!

By doing a simple paper model, I know that I need the position of connection points on strips and it is obvious that these connection points are in different length in left_side_strip, right_side_strip and middle_strip. So if I get the division lengths from Grasshopper I can mark them on strips.

Since strips are curves, the <distance > component does not help me to find measurements. I need distance of points from each other or from the start point of strip but on curve, so when I use it on unfolded paper strip, it gives me the correct position.

To get these lengths I need to find parameters of the connection points on strips (curves) and evaluate their position and the same component would give me the distance of those points from start point of the curve as well.

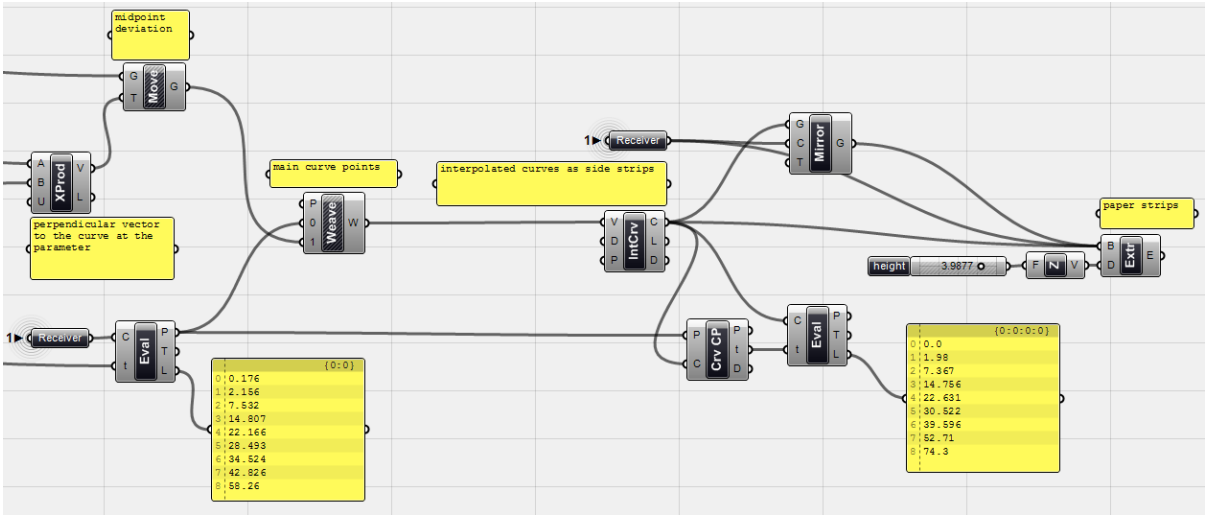


Fig.8.12. As you see I used the first set of evaluated points that I called them main curve points on the middle strip (initial curve). The (L) output of the component gives me distances of points (connection points) from the start points of the strip for middle strip. I also used these points to find their parameter on one side curve. So I used a <curve cp> component to find parameters of points on curve (t). So I used these parameters to evaluate the curve and find their distances from the start point. I would do the same for the next side strip as well.

Make sure that the direction of all curves should be the same and check where is the start point of the curve (the origin of measurements).

Exporting Data

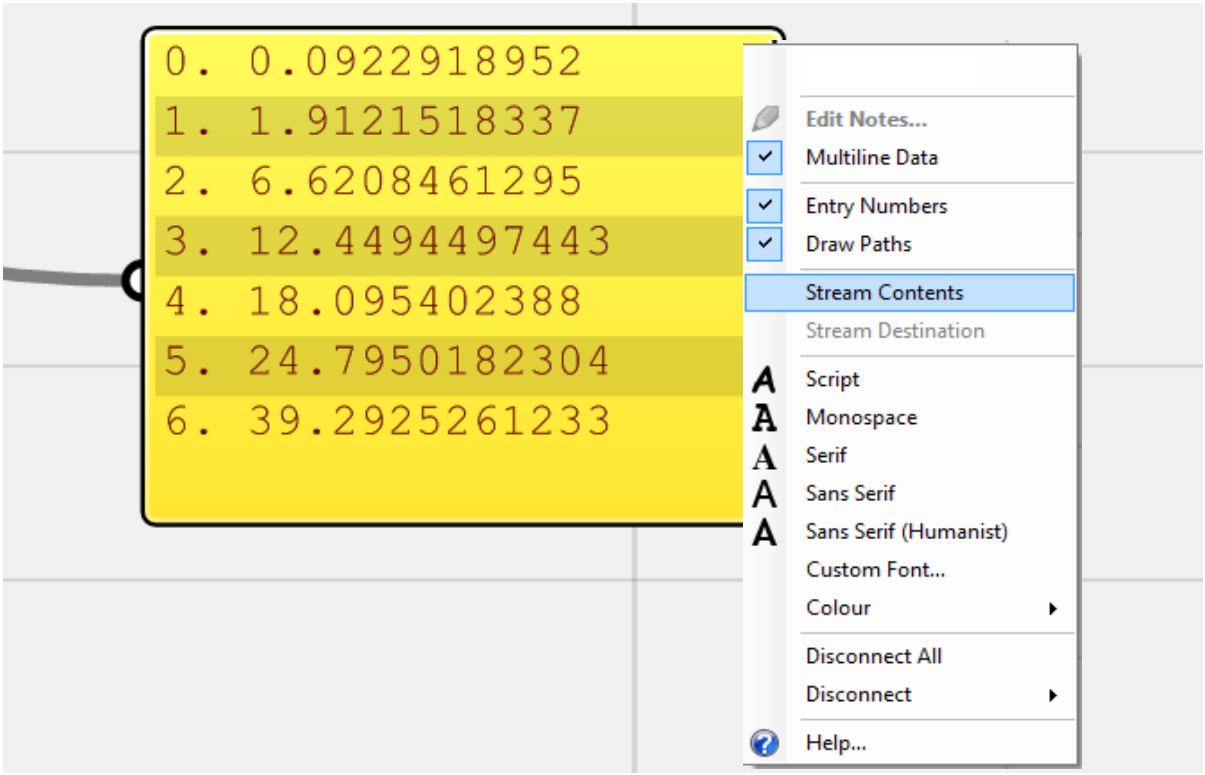


*Fig.8.13. Right-click on the <panel> component and click on the 'Stream Contents'. By this command you would be able to save your data in different formats and use it as a general numeric data. Here I will save it with simple .**txt** format and I want to use it in Microsoft Excel.*

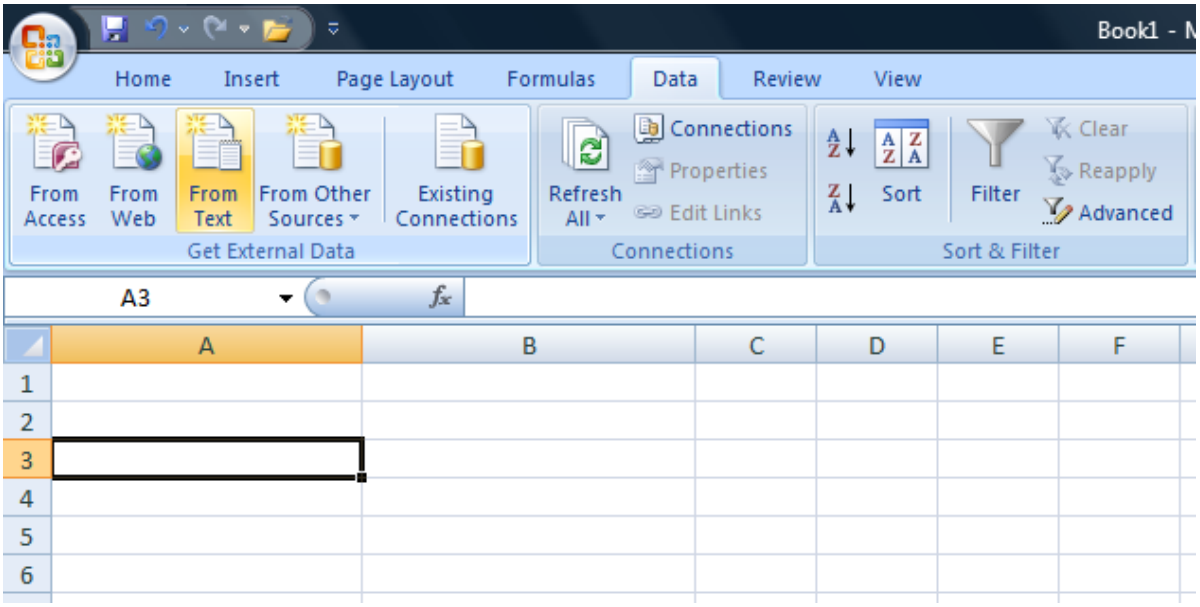


*Fig.8.14. On Excel sheet, simply click on an empty cell and go to the **'Data'** tab and under the **'Get External Data'** select **'From Text'**. Then select the saved txt file from the address you saved your stream contents and follow the simple instructions of excel. These steps allow you to manage your different types of data, how to divide your data in different cells and columns etc.*

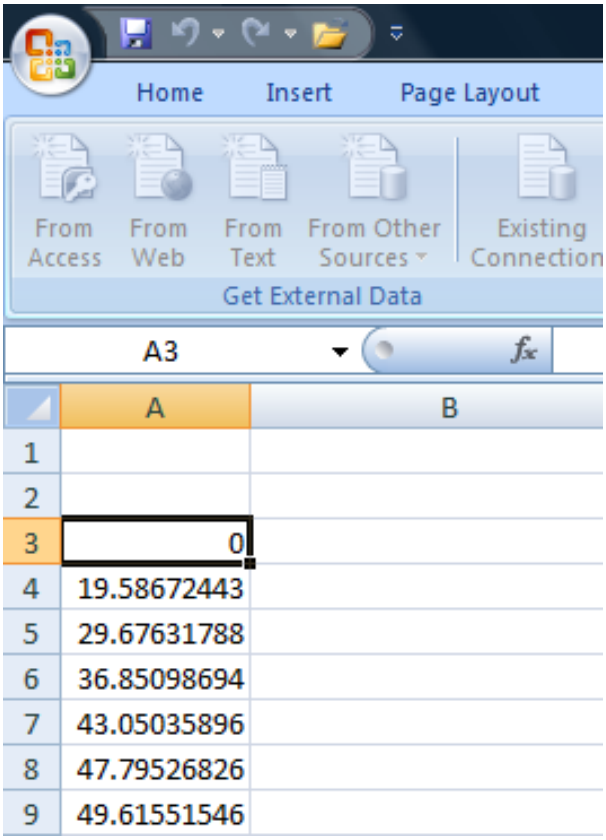


Fig.8.15. Now you see that your data placed on the Excel data sheet. You can do the same for the rest of your strips.

	Connection point distances from start point		
Connection points	Right_strip	Middle_strip	Left_strip
start point	0	0	0
1st connection point	19.58672443	14.49750789	18.71121037
2nd connection point	29.67631788	21.19712374	28.22812292
3rd connection point	36.85098694	26.84307638	34.95597765
4th connection point	43.05035896	32.67167999	41.0266449
5th connection point	47.79526826	37.38037429	45.75826257
end point_(strip length)	49.61551546	39.20023423	47.57834643

Fig.8.16. Table of the distances of connection points alongside the strip.

If you have a list of 3D coordinates of points and you want to export them to Excel, there are different options for that. If you export 3D coordinates with the above method you will see there are lots of unnecessary brackets and commas that you should delete. You can also add columns by clicking in the excel import text dialogue box and separate these brackets and commas from the text in different columns and delete them but again because the size of numbers are not the same, you will find characters in different columns that you could not align separation lines for columns easily.

In such case I simply recommend you to decompose your points to their components and export them separately. It is not a big deal to export three lists of data instead of one.

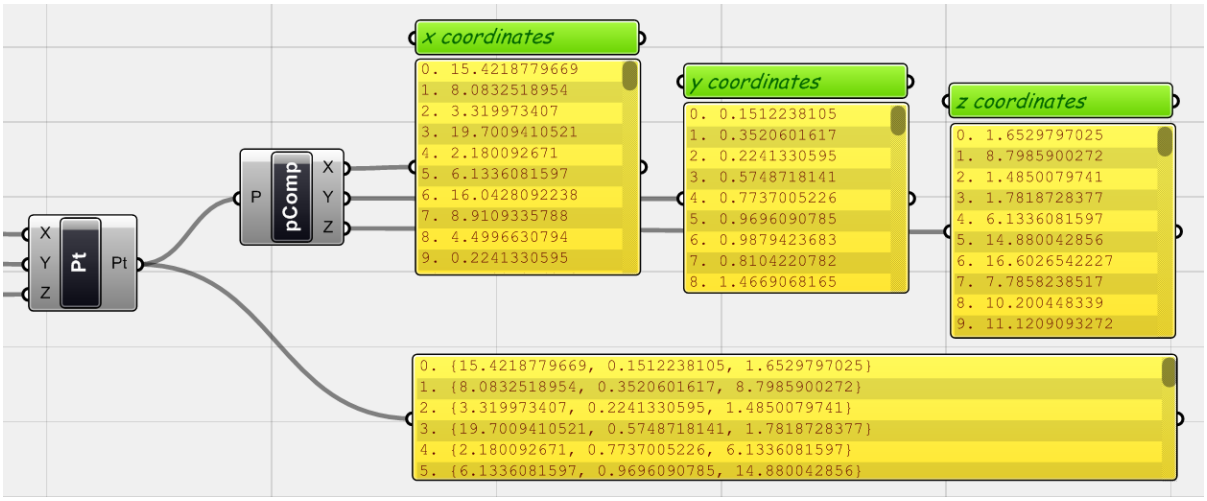


Fig.8.17. Using <decompose> component to get the X, Y and Z coordinates of the points separately to export to a data sheet.

I strongly recommend you to professionally work with Excel and other spreadsheets because they help us in data managements in different ways and situations.

Enough for modelling! I used provided data to mark my paper strips and connect them together and create a simple model. To prove it even to myself, I did all process with hand !!!! to show that fabrication does not necessarily mean laser cutting (but sometimes **HAM**, as Achim Menges (EmTech AA tutor) once used for Hand Aided Manufacturing!!!! For fun). I just spent an hour to cut and mark all strips but the assembly process took a bit longer which should be done by hand anyway.

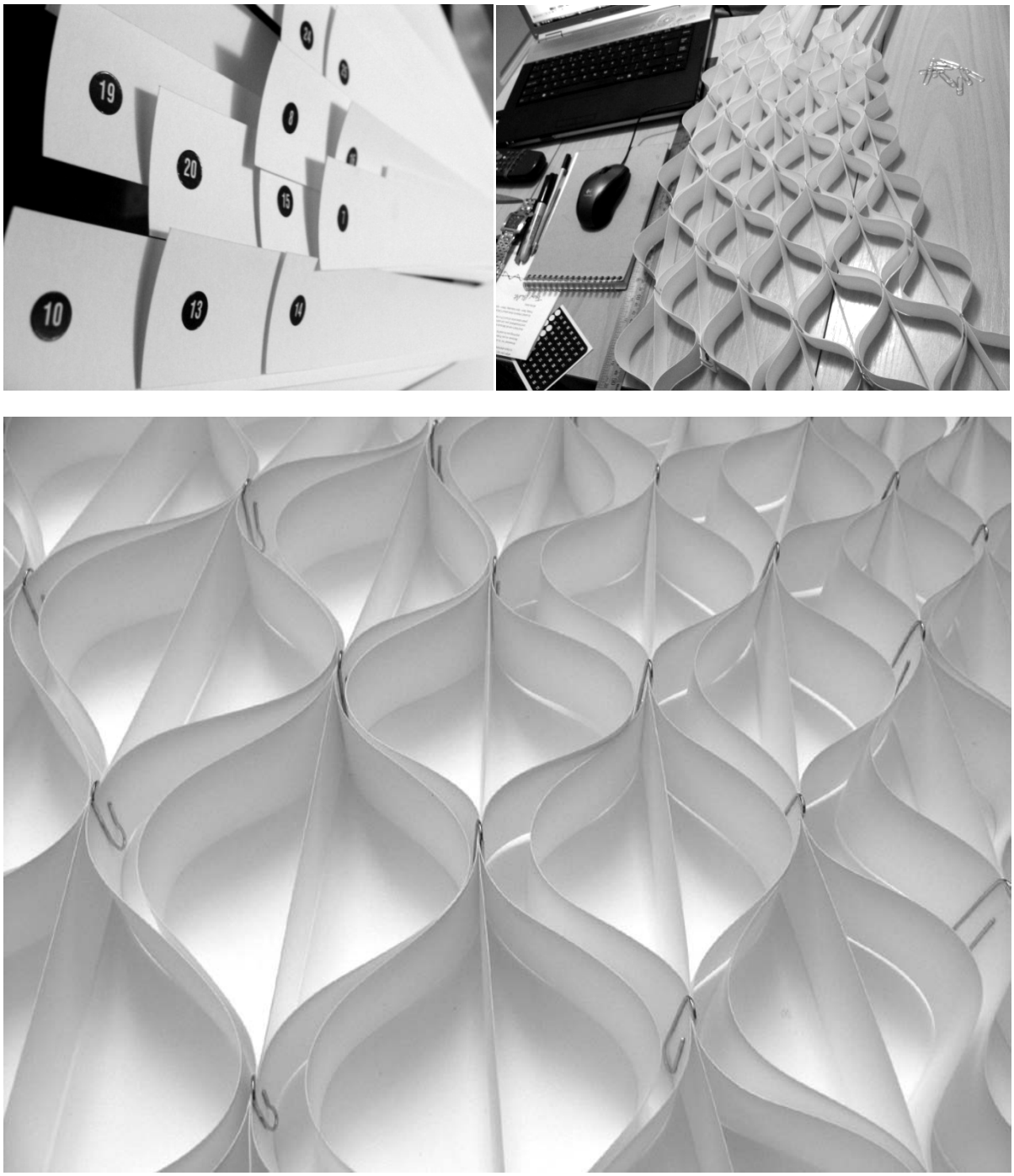


Fig.8.18. Final paper-strip project.

8_2_Laser Cutting and Cutting based Manufacturing

The idea of laser cutting sheet materials is very common these days to fabricate complex geometries. There are different ways that we can use this possibility to fabricate objects. Laser cutter method suits objects that built with developable surfaces or folded ones. One can unfold a digital geometry on a plane and simply cut it out of a sheet and fold the material to build it. It is also suitable to make complex geometries that could be reduced to separate pieces of flat surfaces and one can disassemble the whole model digitally in separate parts, nest them on flat sheets, add the overlapping parts for connection purposes (like gluing) and cut and assemble physically. It is also possible to fabricate double-curved objects by this method. It is well being experimented to find different sections of any 'Blob' shaped object, cut it at least in two directions and assemble these sections together usually with Bridle joints and make rib-cage shaped models.

Since laser cutter is a generic tool, there are various methods, but all together the important point is to find a way, to reduce geometry to flat pieces to cut them from a sheet material, no matter paper or metal, cardboard or wood and finally assemble them together.

Among different ways discussed here, I want to test one of them in Grasshopper and I am sure that you can experiment other methods easily.

Free-Form Surface Fabrication

I decided to fabricate a free-form surface to have some experiments with preparing and nesting pieces of a free-form object to cut and all other issues we need to deal with.

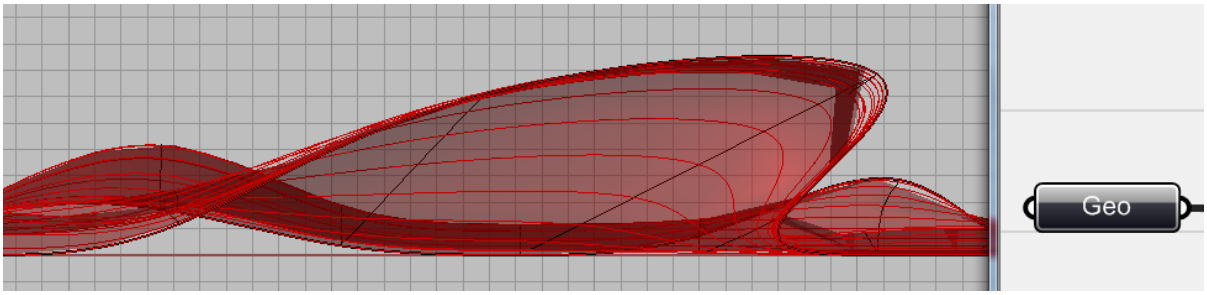


Fig.8.19. Here I have a surface and I introduced this surface to Grasshopper as a <Geometry> component, so you can introduce any geometry that you have designed or use any Grasshopper object that you have generated.

Ribs as Sections

In order to fabricate this generic free-form surface I want to create sections of this surface, nest them on sheets and prepare files to be cut by laser cutter. If the object that you are working on has a certain thickness then you can cut it, but if the model does not have any thickness, you need to add a thickness to cutting parts.

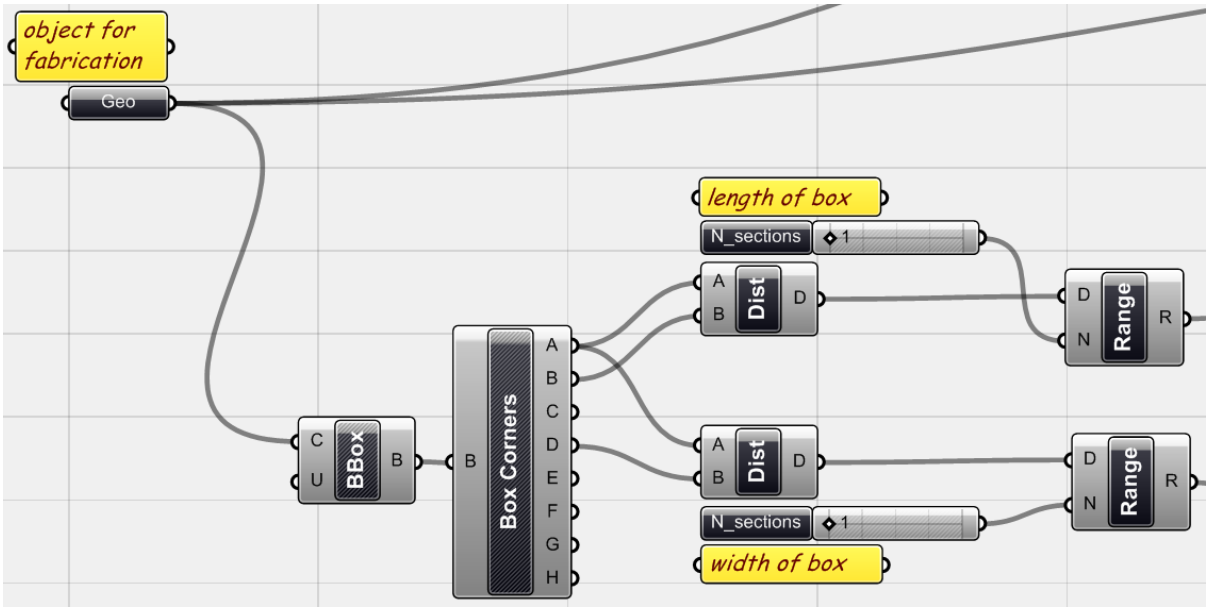


Fig.8.20. in the first step I used a <Bounding Box> component to find the area that I want to work on. I also used a <Box corners> component (Surface > Analysis > Box corners) to find the opposite corners of box and use them as limits of range that I want to generate my ribs alongside the geometry. So by calculating length and width of the box, I used these numbers as domains that I want to divide by a <range> component. Basically by using <number slider> I can simply divide the length and width of the box in desired parts.

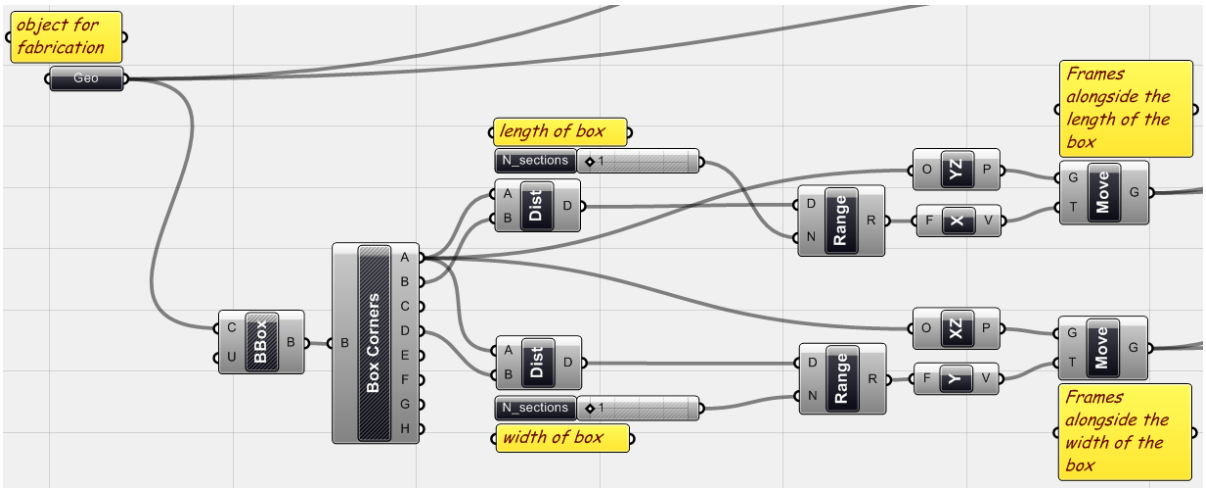


Fig.8.21. My aim is to generate planes alongside the length and width of the box as much as ribs I need. First I generated two planes, one <YZ plane> and one <XZ plane>, first one perpendicular to the length of the box and second one perpendicular to the width. I generated both of them on the first corner of the box by connecting them to the A output of the <box corners>. Now I can generate <Unit X> and <Unit Y> vectors alongside the length and width of the box, and by connecting the <range> components to them, I can make vectors for all division points. Then I can <move> XZ and YZ planes by these vectors and generate series of frames alongside length and width of the object's bounding box.

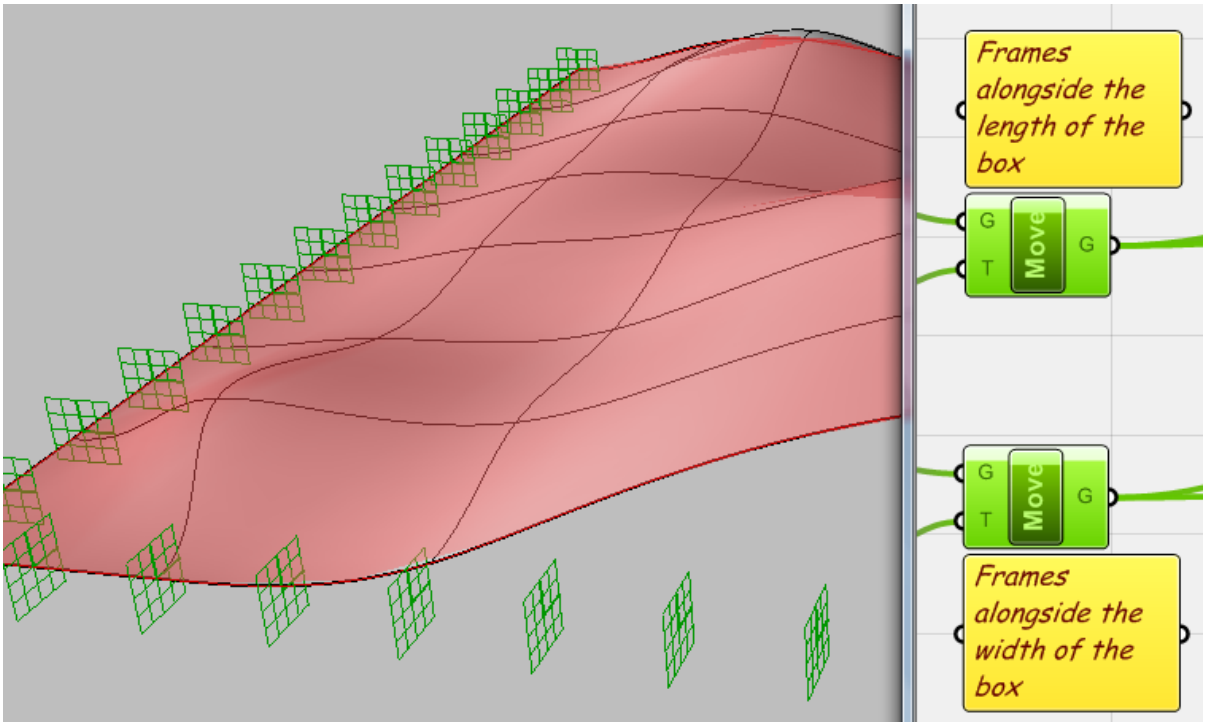


Fig.8.22. Frames generated alongside the length and width of the object's bounding box, perpendicular to the edge.

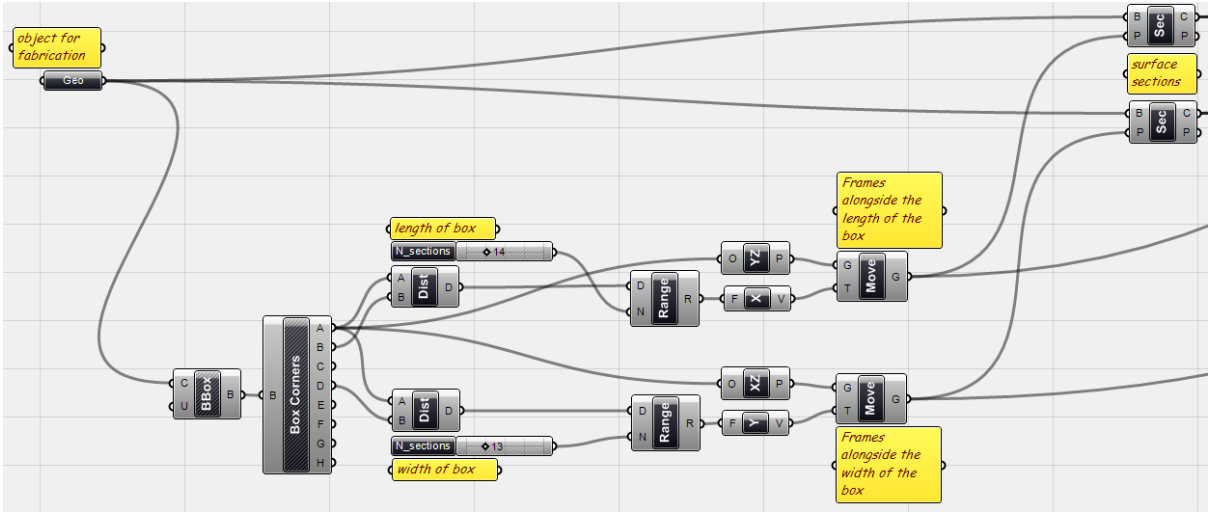


Fig.8.23. Now if I find the intersection of these planes and the surface, I actually generated the ribs and this is the half way to fabricate the surface. Here I used a <BRep | Plane> section component (Intersect > Mathematical > BRep | Plane) to solve this problem. I used the <Geometry> (my initial surface) as BRep and planes of previous step, as planes to feed the section component.

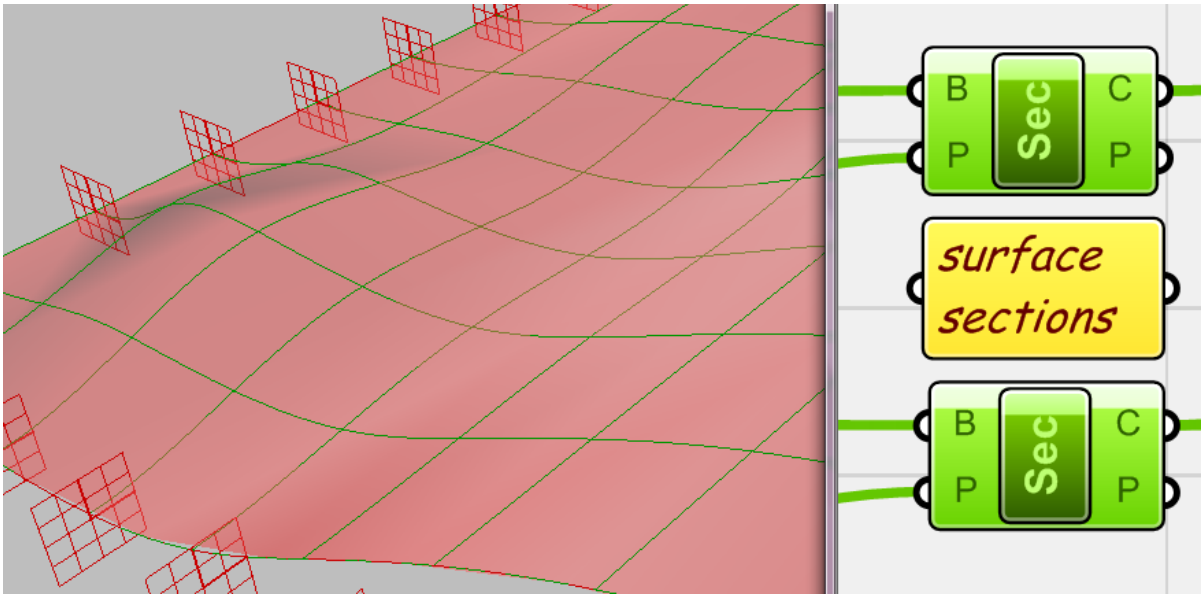


Fig.8.24. Intersections of frames and surface, resulted in series of curves on the surface.

Nesting

The next step is to nest these curve sections on a flat sheet to prepare them for cutting process. Here I drew a rectangle in Rhino with my sheet size. I copied this rectangle to generate multiple sheets overlapping each other and I drew one surface that covers all these rectangles.

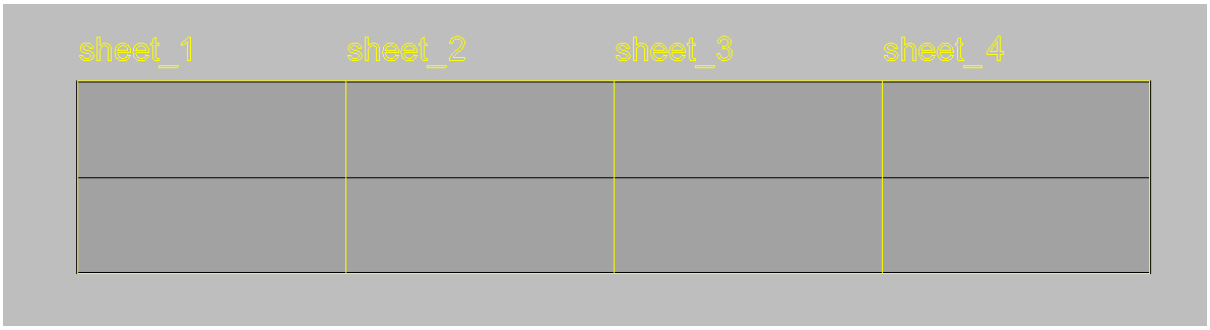


Fig.8.25. Paper sheets and an underlying surface to represent them in Grasshopper.

I am going to use <Orient> component (XForm > Euclidian > Orient) to nest my curves into the surface which represents sheets for cutting purpose. If you look at the <orient> component you see that we need the object's plane as reference plane, and target plane which should be on the sheet. So here I should generate these planes to nest my cutting objects. Since I used planes to intersect the initial surface and generate section curves, I can use them again as reference planes, so I only need to generate target planes.

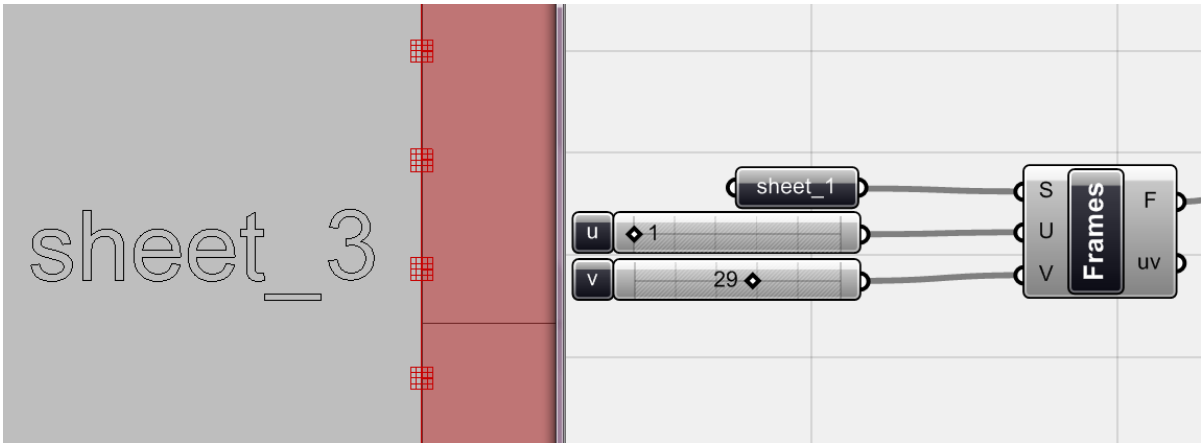


Fig.8.26. I introduced cutting surface to Grasshopper and I used a <surface Frame> component (Surface > Util > Surface frames) to generate series of frames across the surface. We can generate planes, as much as our geometry needs.

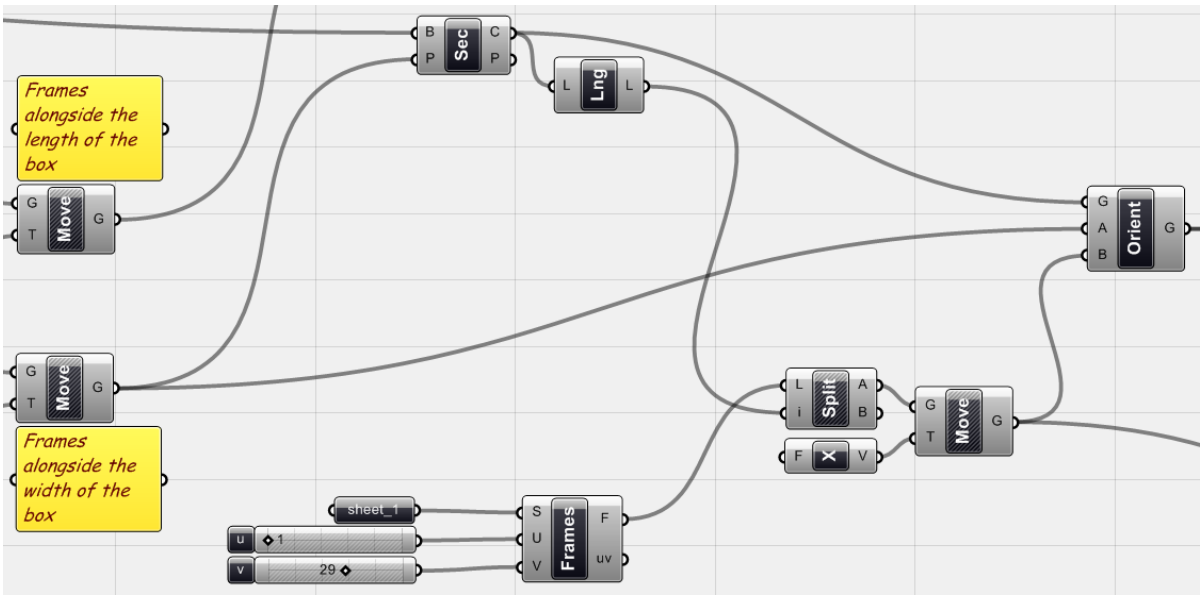


Fig.8.27. Orientation. I connected section curves as base geometries, and planes that I used to generate these sections as reference geometry to the <orient> component. But still a bit of manipulation is needed for the target planes. If you look at the results of <surface frame> component, you see that if you divide U direction by 1, it would generate 2 columns to divide the surface. So I have more planes than needed. That's why I <split>ed the list of target planes by the number that comes from the number of reference curves. So I only use planes as much as curves that I have. Then I moved these planes 1 unit in X direction to avoid overlapping with the sheet's edge. Now I can connect these planes to the <orient> component and you can see that all curves now nested on the cutting sheet.

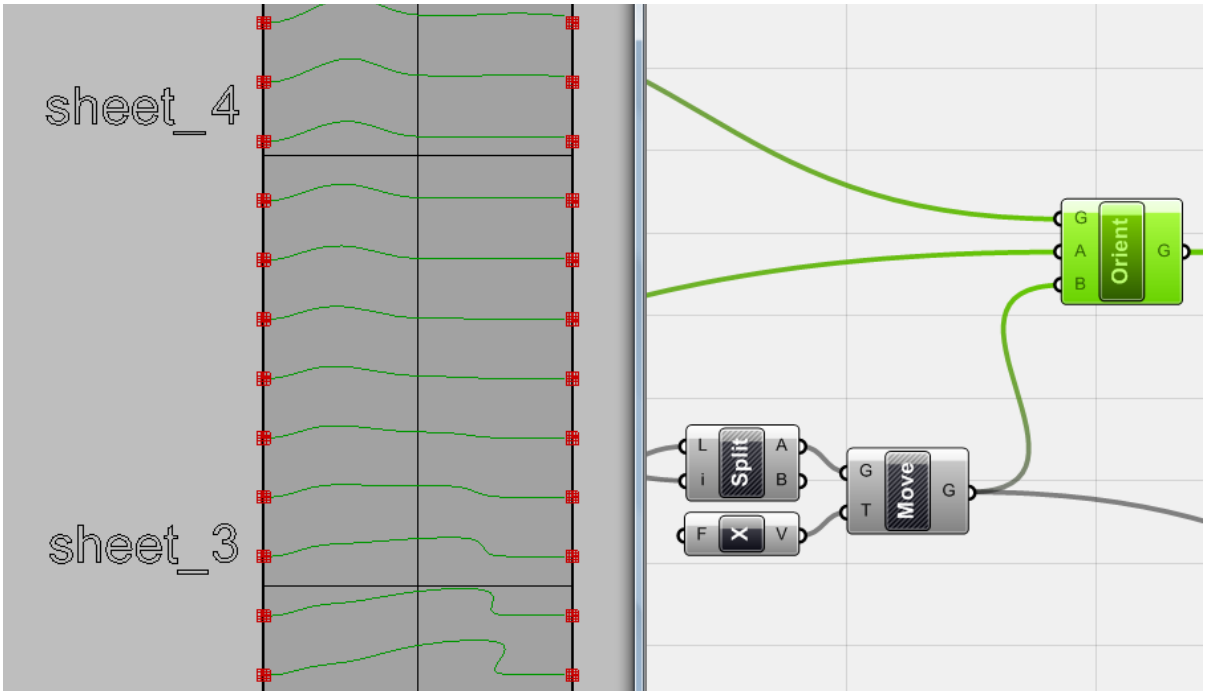


Fig.8.28. nested curves on the cutting sheet.

Generating Ribs

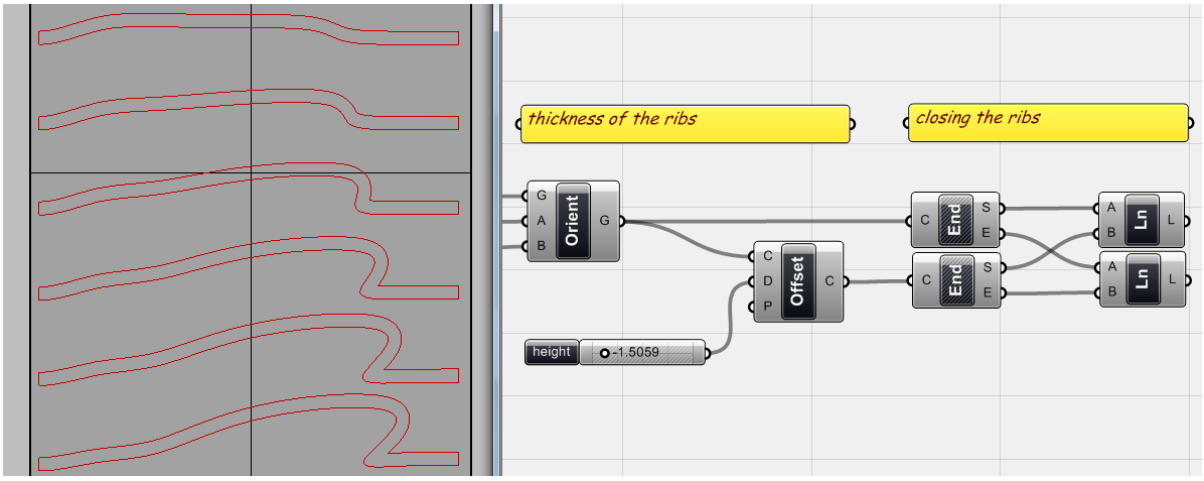


Fig.8.29. After nesting curves into cutting sheet, as I told you, because our object does not have any thickness, in order to cut it, we need to add thickness to it. That's why I <offset> curves with desired height and I also add <line>s to both ends of these curves and their offset ones to close the whole drawing so I would have complete ribs to cut.

Generating Joints (Bridle Joints)

The next step is to generate ribs in other direction and make joints to assemble them after being cut. Although I used the same method of division of the bounding box length to generate planes and then sections, but I can generate planes manually in any desired position as well. So in essence if you do not want to divide both directions and generate sections, you can use other methods of generating planes instead of evenly dividing the edge.

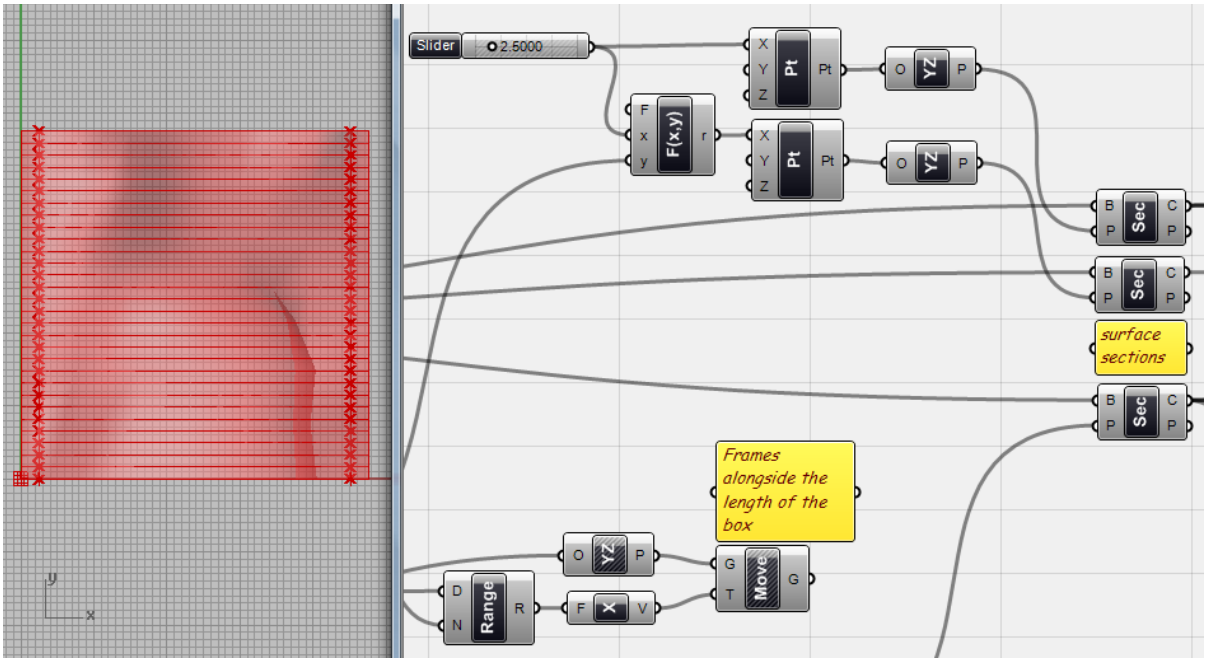


Fig.8.30. As you see here, instead of previously generated planes, I used manually defined planes for sections in the other direction of the surface. One plane generated by X value directly from <number slider> and another plane comes from the mirrored plane on the other side of the surface (surface length (y) – number slider (x)). Section of these two planes and surface is being calculated for the next steps.

Now I can orient these new curves on another sheet to cut, which has the same process as the other one. So let's generate joints for the assembly of parts which is important.

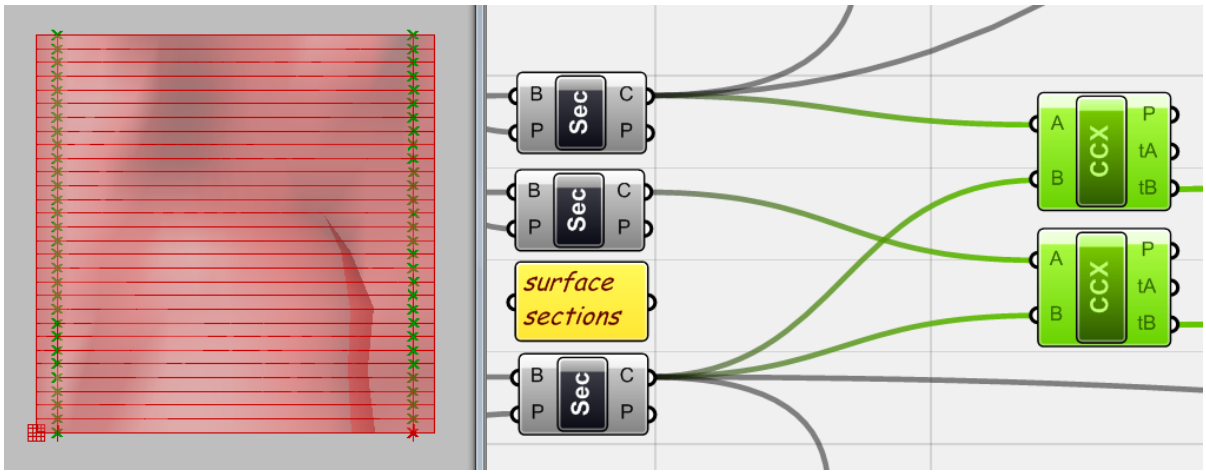


Fig.8.31. since we have curves in two directions, we can find their points of intersection to design joints on that positions. That's why I used <CCX> components (Intersect > Physical > Curve | Curve) to find the intersect position of these curves which means the joint positions (The <CCX> component is in cross reference mode).

I need a bit of drawing to prepare these joints to cut. I am thinking of preparing bridle joints so I need to cut half of each rib on the joint position to be able to join them at the end. First I need to find these intersection positions on nested ribs and then draw lines for cutting.

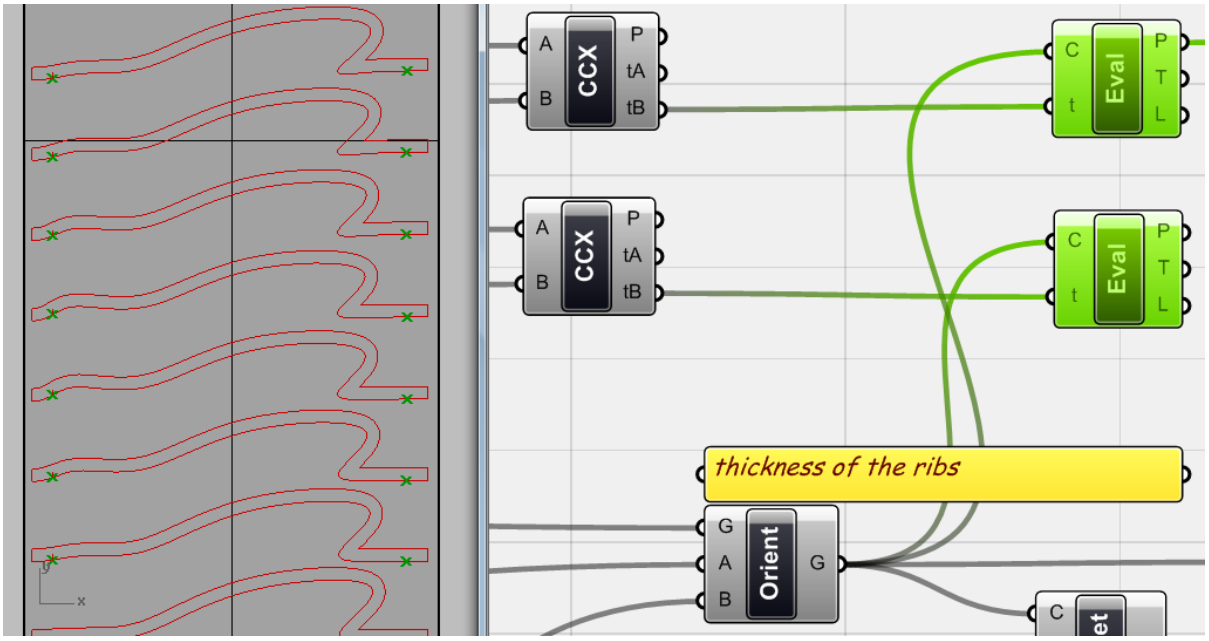


Fig.8.32. If you look at the outputs of the <CCX> component you would see that it gives us the parameter in which, each curve intersects with the other one. So I can <evaluate> nested or <orient>ed curves with these parameters to find the joints' positions on cutting sheets.

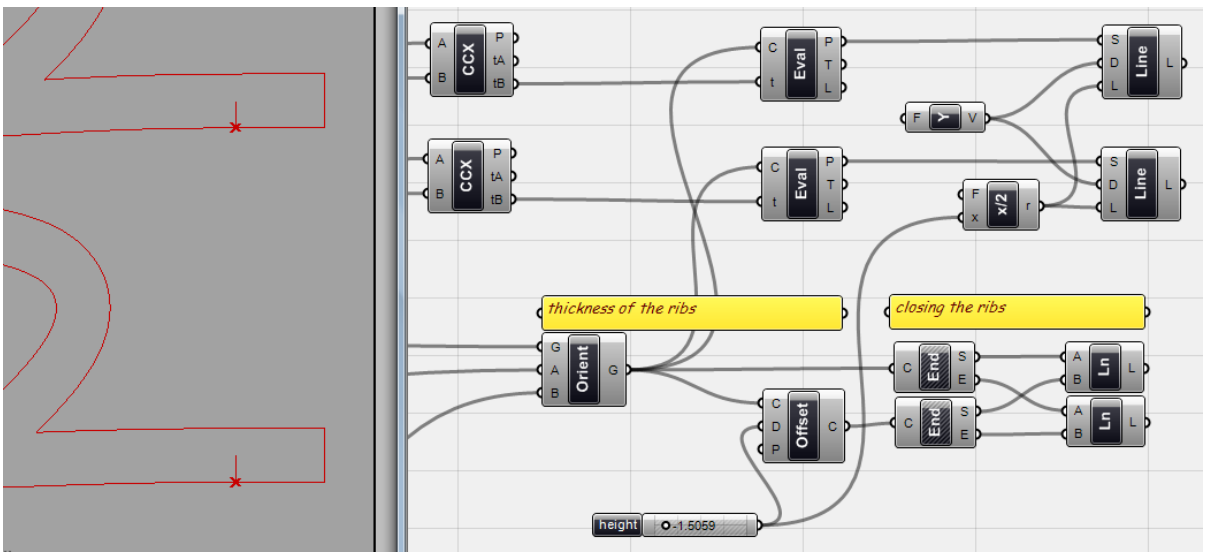


Fig.8.33. Now we have the joint positions, we need to draw them. First I drew lines with <line SDL> component with the joint positions as start points, <unit Y> as direction and I used half of the rib's height as the length of the line. So as you see each point on nested curves now has a tiny line associated with it.

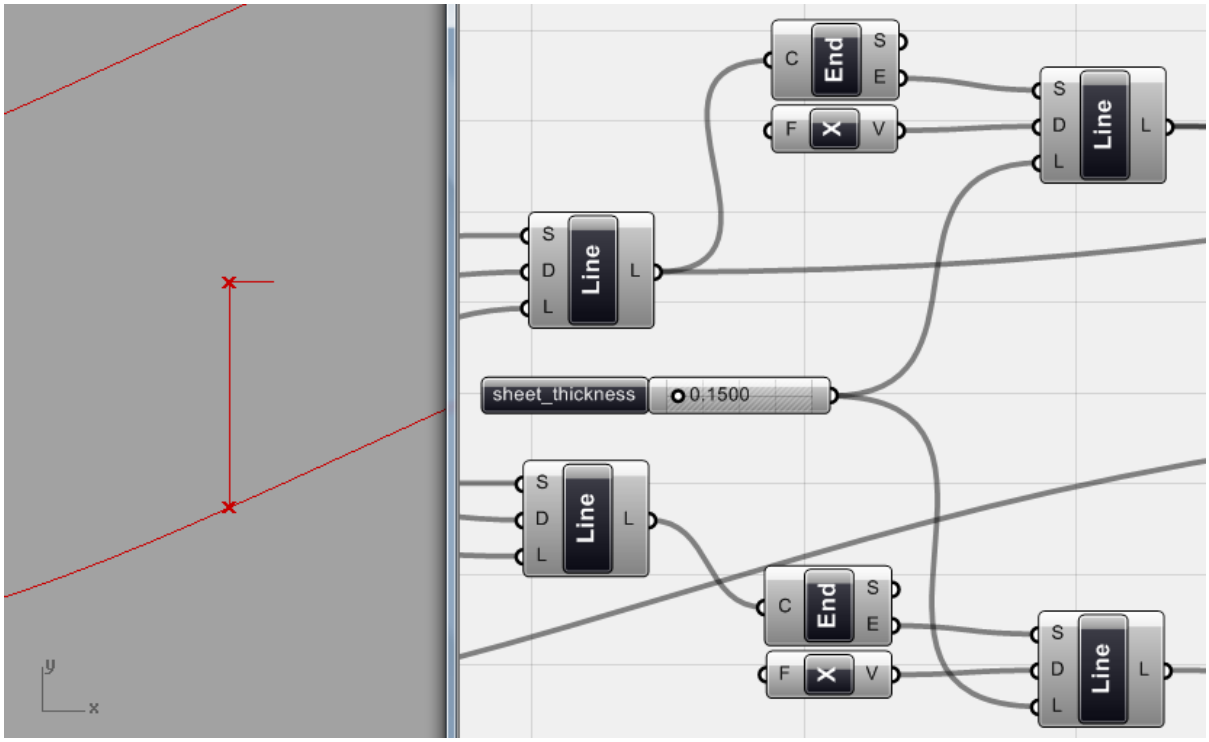


Fig.8.34. Next step, draw a line in X direction from the previous line's end points with the length of the <sheet_thickness> (depends on the material).

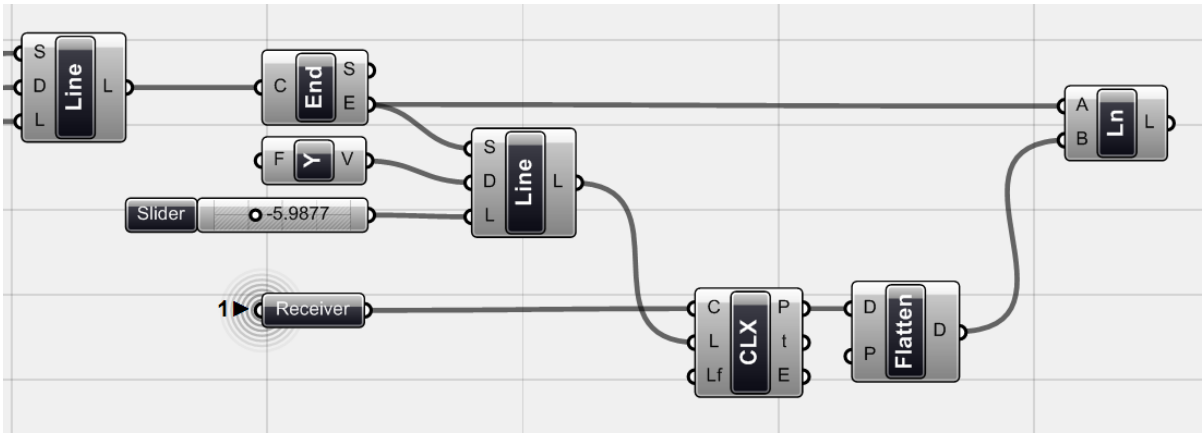
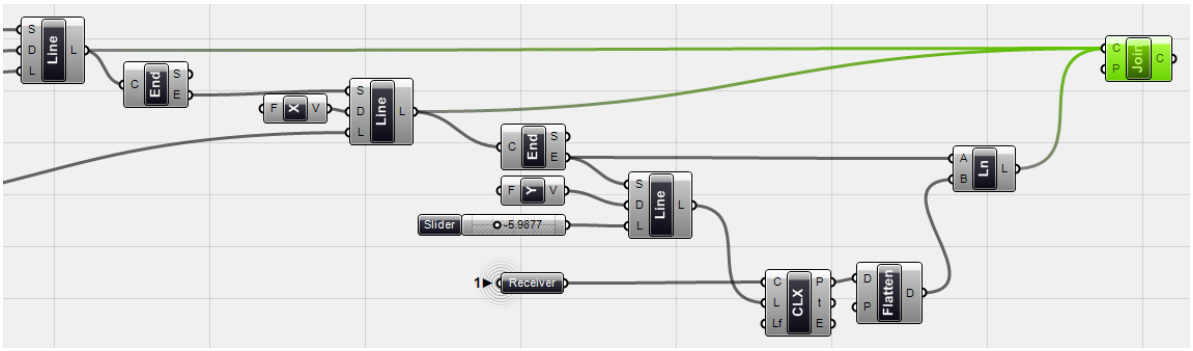


Fig.8.35. To draw the third line I need to find the point where this line connects to the base curve because I don't know its length exactly. In this step, I added another <line SDL> with Y direction and minus value to draw the third line, but a bit longer than needed, to cross the base curve, to find the intersection point. The <receiver> connected to the oriented curves. So I used <CLX> (Intersect > Mathematical > Curve | line>) to find intersection positions with base curve. I <Flatten>ed these points and added <line>s, again from the end point of the second line to this intersection point. As a result, joints are completed now. I have to complete this for both side joints.



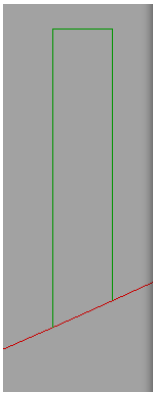


Fig.8.36. Using a <join curves> component (Curve > Util > Join curves) now as you can see I have a slot shaped <join curve> that I can use for cutting as bridle joints inside ribs. I am applying the same method for the other end of the curve (second series of joints on the other side of the oriented curve).

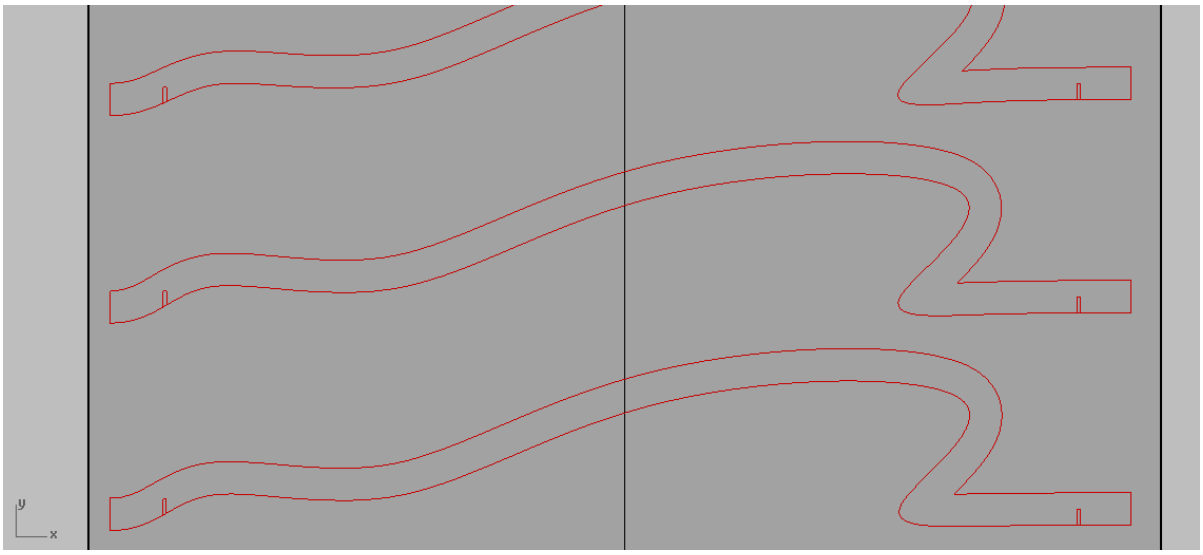


Fig.8.37. Ribs with joints drawn on their both ends. I can trim the tiny part of the base curve inside joints but because it does not affect the geometry I can leave it.

Labelling

While working in fabrication phase, it might be a great disaster to cut hundreds of small parts without any clue or address that how we are going to assemble them together, what is the order, and which one goes first. It is obvious that because all parts are different, we need to label them in order to assemble easily.

It could be simply a number or a combination of text and number to address pieces. If the object comprises of different main parts each divided into pieces, then we can name main parts, so we can use these names or initials with numbers to address the pieces (i.e. left_wing_01). We can use different hierarchies of project assembly logic in order to name parts as well (i.e. layer_01_p_45).

Here I just need a series of numbers to show the position of ribs in the list. I can use <text tag> component in order to add text to my geometry and for that, I need text to display, position of the text, and height of the text.

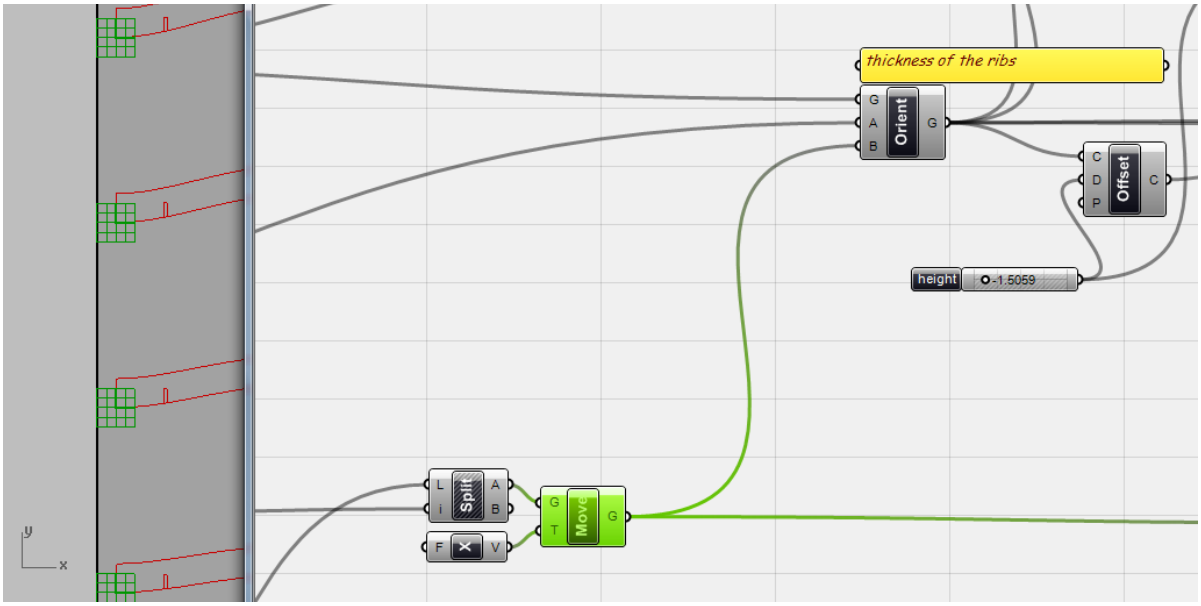


Fig.8.38. As you remember I had a series of planes which I used as target planes for orientating my section curves on the sheet. I am going to use same planes to make positions of the text. Since these planes are exactly at the corner of ribs I have to displace them first.

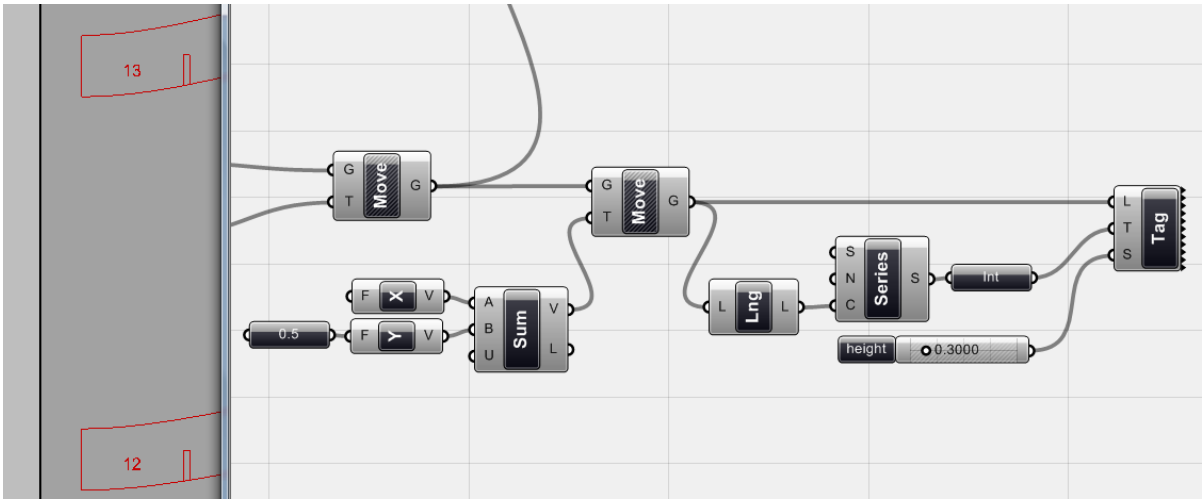


Fig.8.39. I moved corner planes 1 unit in X direction and 0.5 unit in Y direction (as <sum> of the vectors) and I used these planes as positions of text tags. Here I used <text tag 3D> and I generated a series of numbers as much as ribs I have to use them as texts. The <integer> component that I used here converts 12.0 to 12 but you can do it with functions as well. As a result, you can see all parts have a unique number in their left corner.

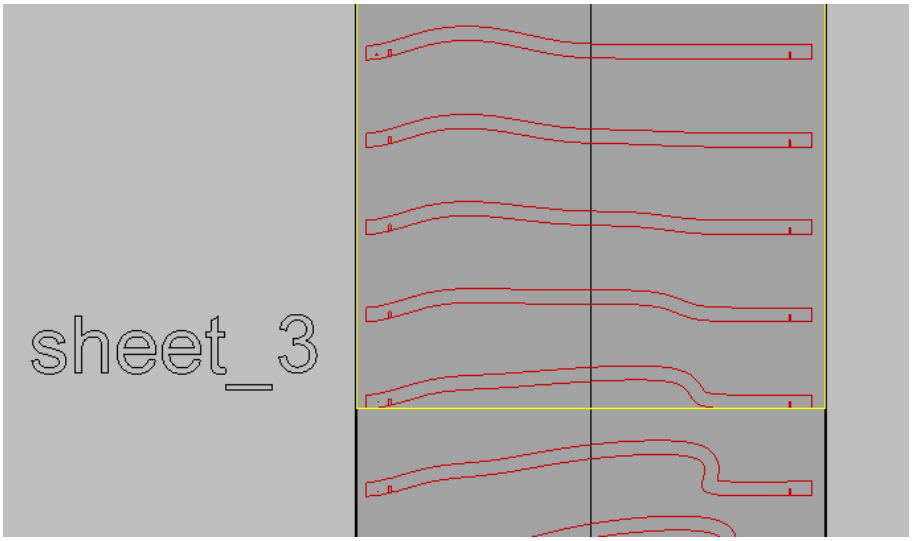


Fig.8.40. Now you can change division factors of the cutting surface to compress ribs as much as possible to avoid wasting material. As you see in the above example, from the start point of the sheet_3, ribs started to be more flat and you have more space in between. Here you can split your ribs in two different cutting surface and change the division points of them, to compress them based on their shape. But if you are not dealing with lots of parts you can always do this type of stuff manually in Rhino; all parts do not need to be Associative! Now I have ribs in one direction, and I am going to do the same for the other direction of ribs as well. The only point that you should consider here is that the direction of joints flip around, so basically while I was working with the <orient>ed geometry in the previous part here I should work with the <offset> one.

Cutting

When all geometries become ready to cut, I need to Bake them and manage them a bit more on my sheets. As you see in Figure 8.41 they all nested in three sheets. I generated three different shapes for ribs in the width direction of the object to check them out. The file is now ready to be cut.

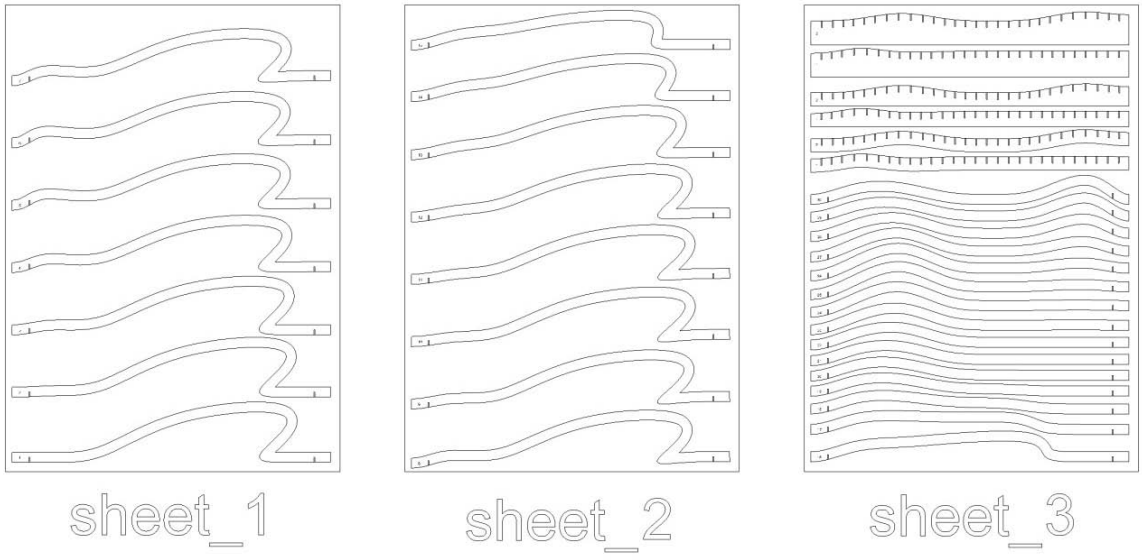


Fig.8.41. Nested ribs, ready to be cut.

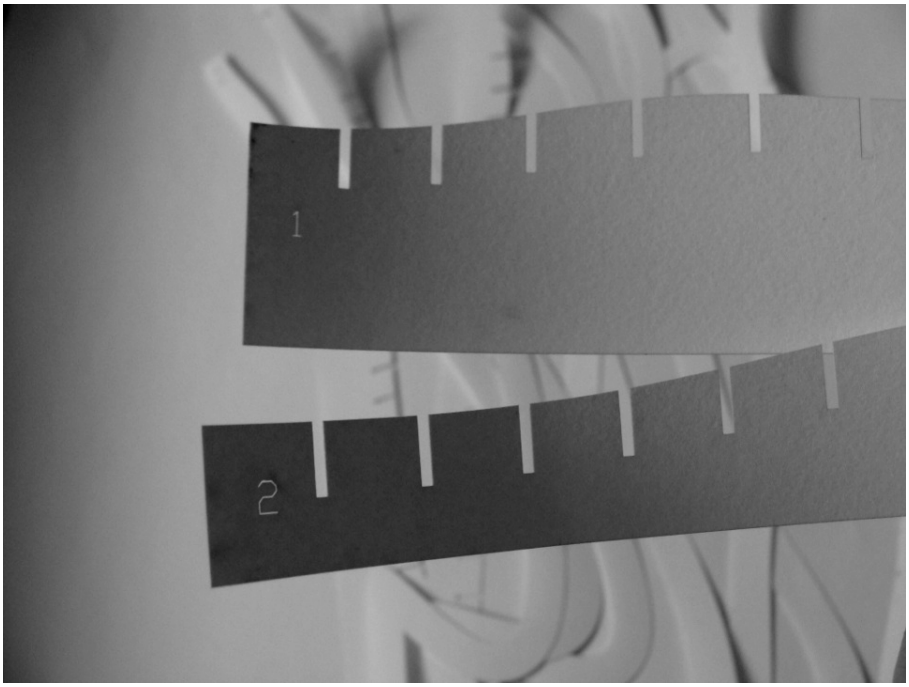
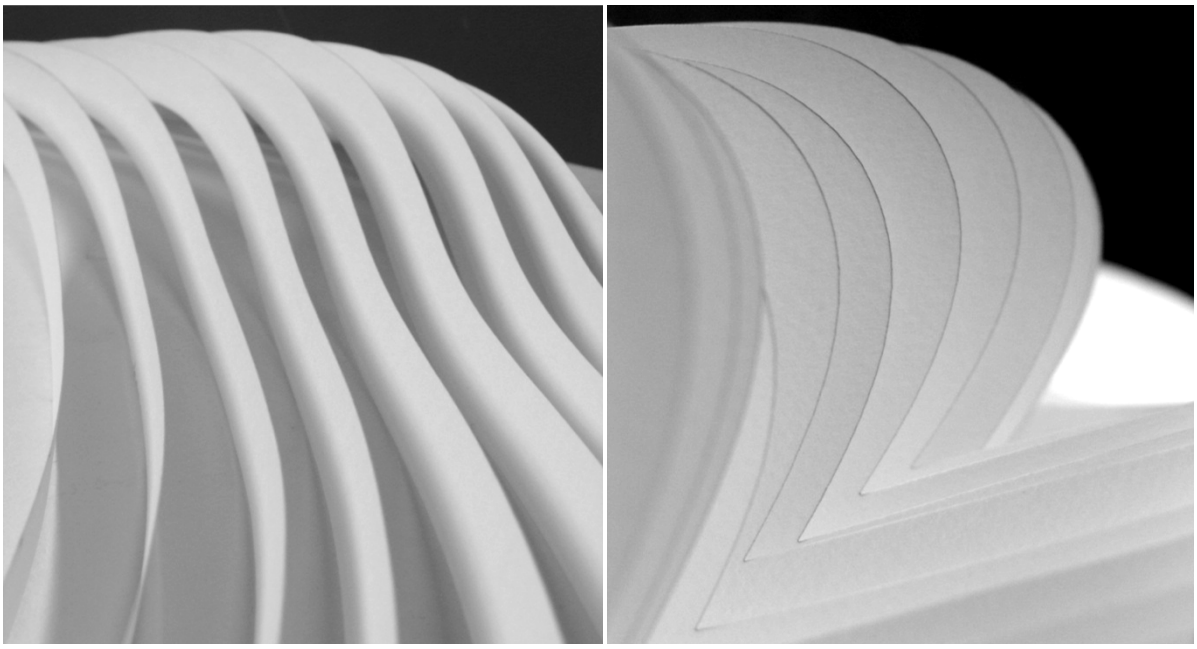
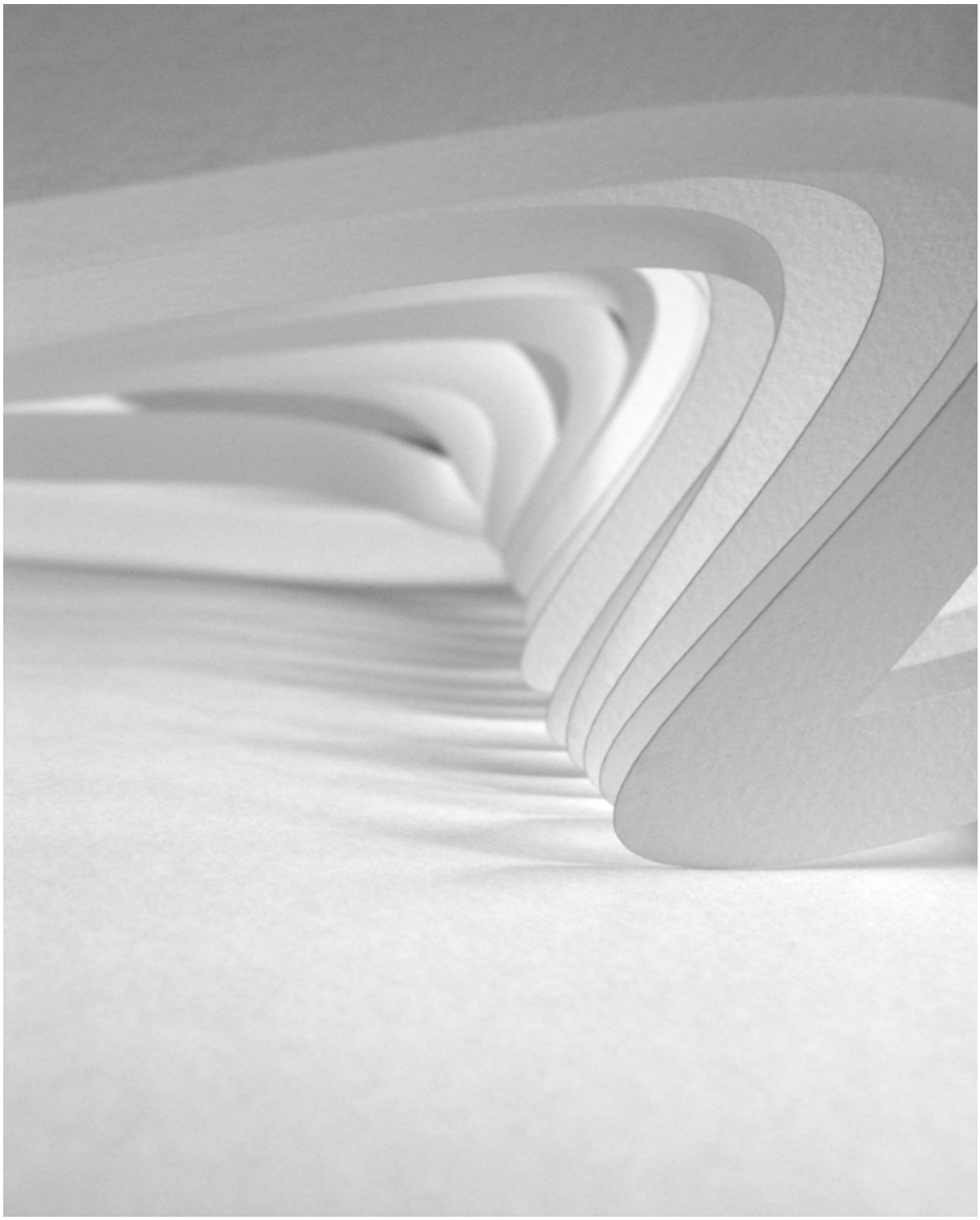


Fig.8.42. Sample of Ribs, ready to assemble.

Assembly

In our case assembly is quite simple. Sometimes you need to check your file again or even provide some help file in order to assemble your parts in different fabrication methods. All together, here is the surface that I made by paper sheet.





8.43. Final Model.

Fabrication is a wide topic to discuss. It highly depends on what you want to fabricate, what is the material, what is the machine and how fabricated parts going to be assemble and so on. As I told you before, depends on the project you are working on, you need to provide your data for fabrication stages. Sometimes it is more important to get the assembly logic, for example when you are working with simple components, but complex geometry as a result of assembly.

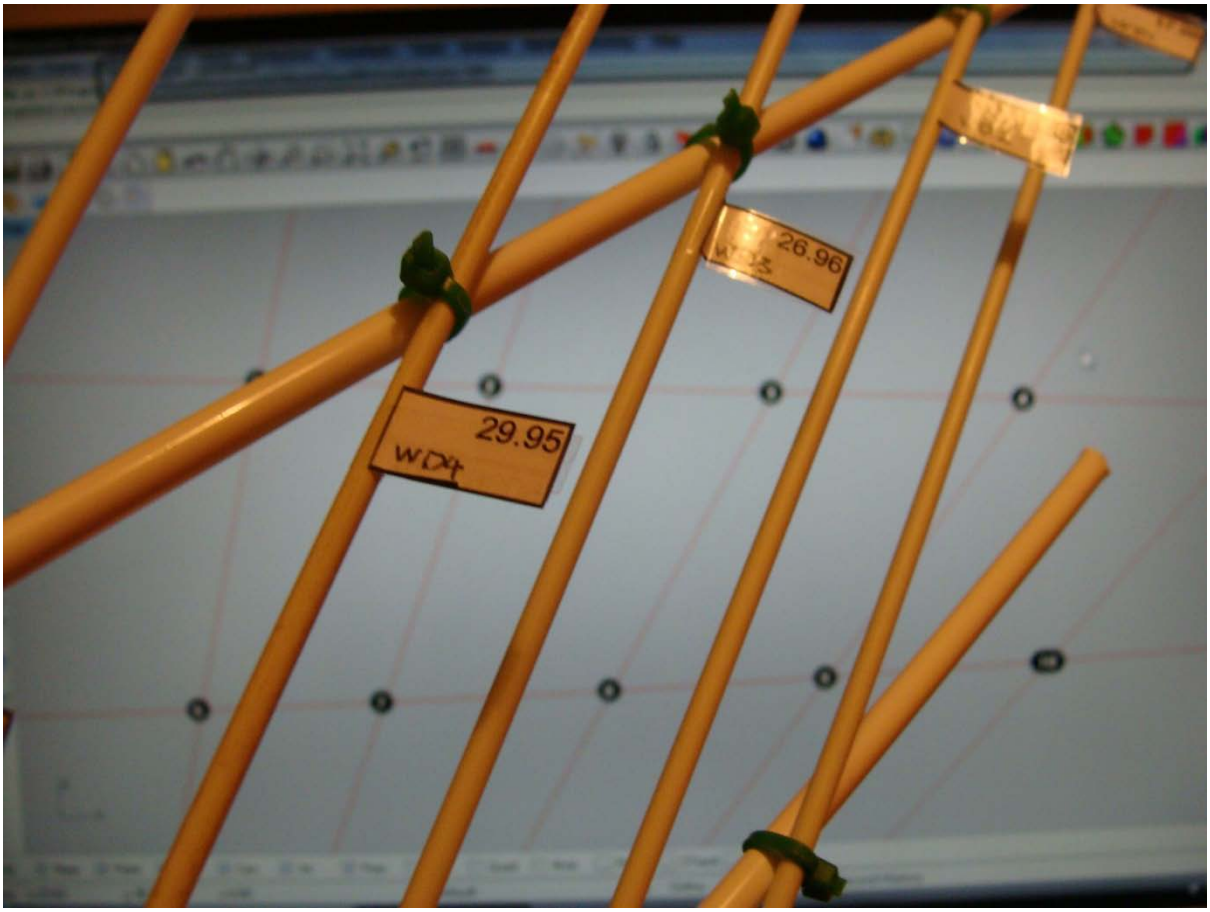


Fig.8.44. Assembly logic; Material and joints are simple; I can work on the assembly logic and use the data to make my model.

Chapter_9_Design Strategy

Design Strategy

Generative Algorithms are algorithmic and Parametric/Associative ways of dealing with geometry in design problems. More than conventional geometrical objects, with this algorithmic method, now we have all possibilities of computational geometries as well as managing huge amount of data, numbers and calculations. Here the argument is to not limit the design in any predefined experiment, and explore infinite potentials; there are always alternative ways to set up design algorithms. Although it seems that the in-built commands of these parametric modelling softwares could limit some actions or dictate methods, but alternative solutions can always be brought to the table, let our creativity fly away of limitations.

In order to design something, having a Design Strategy always helps to set up the best possible algorithm to find the design solution. Thinking about general properties of design object, drawing some parts, even making some physical models, would help for a better understanding of the algorithm so better choice of <components> in digital modelling. Thinking about fix parameters, parameters that might change during the design, numerical data and geometrical objects needed, always help to improve the algorithm. It would be helpful to analytically understand the design problem, sketch it and then start an algorithm that can solve the problem.

We should think in an algorithmic way to design algorithmic!

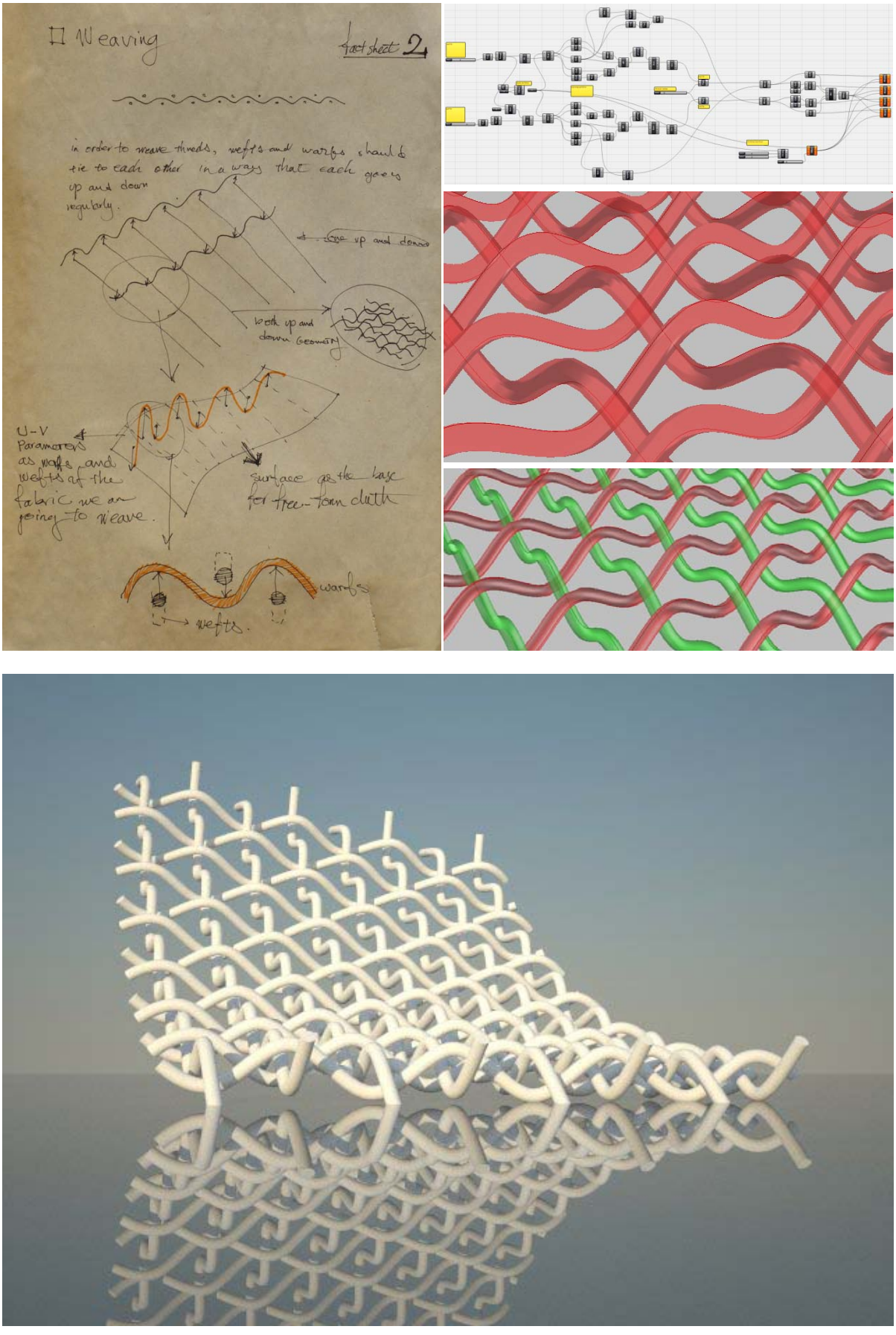


Fig.9.1. Weaving project; From Analytical understanding to Associative modelling.

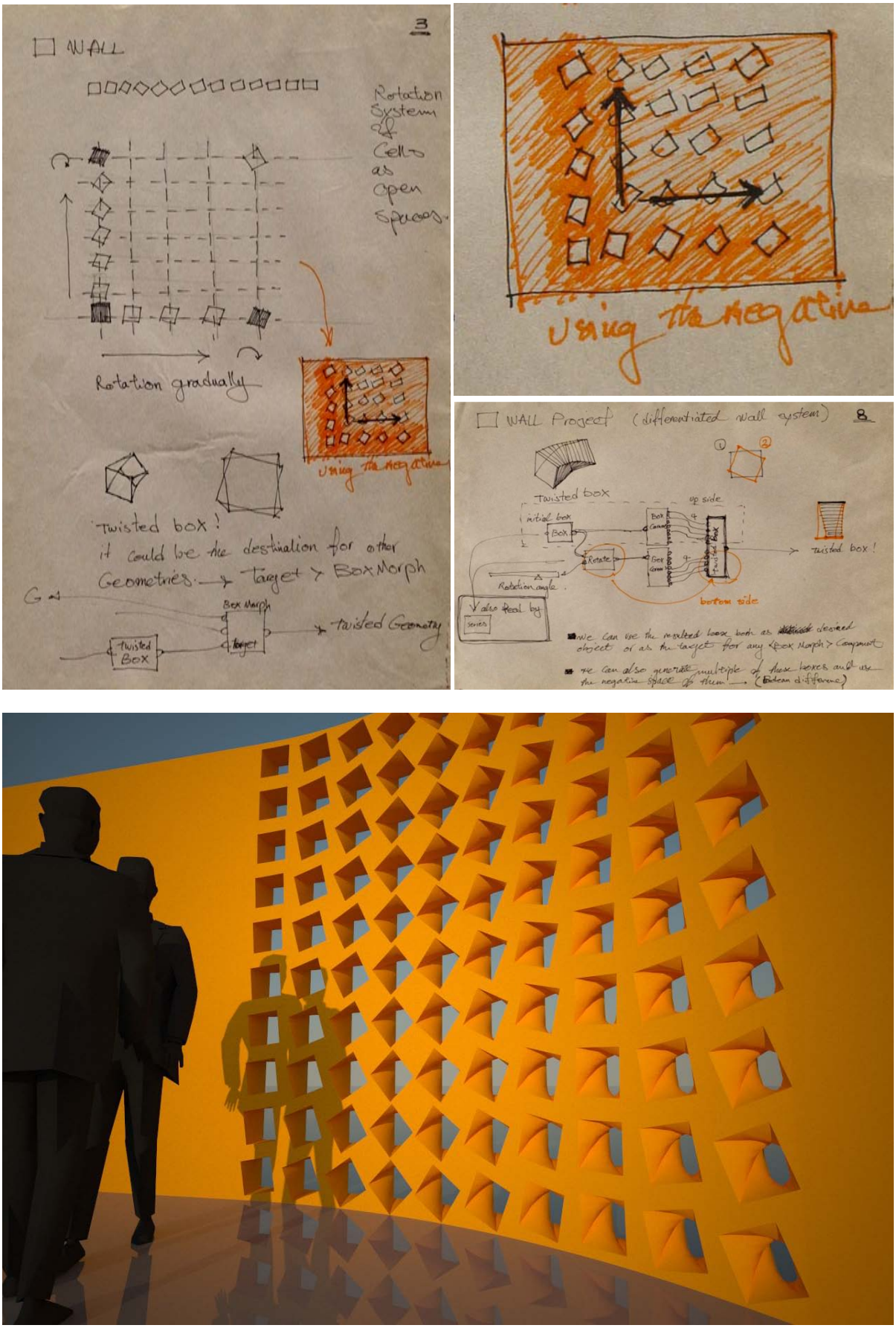


Fig.9.2. Porous wall project; From Analytical understanding to Associative modelling.

Bibliography

Pottman, Helmut and Asperl, Andreas and Hofer, Michael and Kilian, Axel, 2007: **'Architectural Geometry'**, Bently Institute Press.

Hensel, Michael and Menges, Achim, 2008: **'Morpho-Ecologies'**, Architectural Association.

Rutten, David, 2007: **'Rhino Script 101'**, digital version by David Rutten and Robert McNeel and Association.

Flake, Gary William, 1998: **'The computational beauty of nature, computer explorations of fractals, chaos, complex systems, and adaptation'**, The MIT Press.

De Berg, Mark and Van Kreveld, Marc and Overmars, Mark, 2000: **'Computational Geometry'**, Springer.

Grasshopper tutorials on Robert McNeel and Associates wiki:
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryExamples.html

Axel Kilian and Stylianos Dritsas: **'Design Tooling - Sketching by Computation'**,
http://www.designexplorer.net/designtooling/inetpub/wwwroot/components/sketching/index.html

Wolfram Mathworld: http://mathworld.wolfram.com/

Stylianos Dritsas, http://jeneratiff.com/

Main Grasshopper web page: http://grasshopper3d.com/

Notes

GENERATIVE ALGORITHMS

using GRASSHOPPER

Zubin Khabazi

Ver.02

www.MORPHOGENESISM.com