ARTURO TEDESCHI

AAD_

ALGORITHMS-AIDED DESIGN

PARAMETRIC STRATEGIES USING GRASSHOPPER®

Foreword by Fulvio Wirz

LE PENSEUR PUBLISHER

ARTURO TEDESCHI

AAD_

ALGORITHMS-AIDED DESIGN

PARAMETRIC STRATEGIES USING GRASSHOPPER®

Foreword by Fulvio Wirz

LE PENSEUR PUBLISHER

AAD_Algorithms-Aided Design

Parametric Strategies using Grasshopper®

Arturo Tedeschi

with contributions by Stefano Andreani, Antonella Buono, Maurizio Degni, Lawrence Friesen, Andrea Galli, Francesco Lipari, Davide Lombardi, Ludovico Lombardi, Arthur Mamou-Mani, Alberto Pugnale, Antonio Turiello, Brian Vesely, Lorenzo Vianello, Fulvio Wirz.

Text revision by Brian Vesely

Each author bears responsibility exclusively for the contents he wrote and might dissent from contents of other authors featured in this book.

ISBN 978-88-95315-30-0

First edition 2014

Printed in Italy (september 2014)

This book mentions several projects and buildings not necessarily designed using the algorithm editor Grasshopper®, but whose complexity was suitable for illustrating the potentials of the algorithmic modeling.

CONTENTS

Acknowledgements

I would like to express my gratitude to the following people and institutions. Without their help and advices it wouldn't be possible to write this book in its current form.

First of all I would like to thank Maurizio Degni and Davide Lombardi who assisted me during the writing and editing. Their support and comments have been essential for this publication. I'm also indebted to Brian Vesely for the final revision and to Alessia De Luca who proofread the manuscript.

This book gathers experiments, methods and strategies that I developed as an independent researcher, designer and tutor. Specific topics are part of a research led at AA Rome Visiting School organized by the Architectural Association School of Architecture (London) and directed by Lorenzo Vianello, Lawrence Friesen and the author, under the coordination of Christopher Pierce.

The book also benefits from the original contributions of several people: Stefano Andreani, Antonella Buono, Lawrence Friesen, Andrea Galli, Francesco Lipari, Ludovico Lombardi, Arthur Mamou-Mani, Alberto Pugnale, Antonio Turiello, Lorenzo Vianello, Fulvio Wirz. I would also thank their respective companies and institutions.

Particular thanks go to the McNeel staff: Elena Caneva, Bob McNeel, Jody Mills, Carlos Perez, Giulio Piacentino, Delia Robalo, David Rutten. I wish to acknowledge the supportive Grasshopper and Rhino communities, in particular: Michele Calvano, Yannis Chatzikonstantinou, Jissi Choi, Paul Cowell, Arturo de la Fuente, Simon Flöry, Ursula Frick, Riccardo Gatti, Thomas Grabner, Giorgio Gurioli, Rodrigo Medina Garcia, Nathan Miller, Michalatos Panagiotis, Daniel Piker, Kaijima Sawako, Will Pearson, Clemens Presinger, Michael Pryor, Filipe Reis, Mateusz Zwierzycki.

I would also acknowledge people that, in different ways, allowed me to start and enhance my experience as a lecturer over the last four years, in particular: Rosetta Angelini, Paolo Cascone, Giancarlo Di Marco, Paolo Fusero, Andrea Giordano, Matteo Gobbi, Lorenzo Massimiano, Giuseppe Massoni, Vittorio Paris, Attilio Pizzigoni, Livio Sacchi, Antonino Saggio, Simone Simonelli.

My gratitude also goes to several people that inspired me or encouraged my work over the years: Eugenio Aglietti, Fabrizio Aimar, Antonietta Andriuoli, Giuseppe Aquino, Andrea Balducci Casté, Chiara Cola, Peter Cook, Rodolfo Beltran, Alessio Biagi, Luca Biselli, Federico Borello, Paolo Franco, Niccolò Casas, Nicholette Chan, Carlo Coppola, Mario Coppola, Paolo Cresci, Maurizio Crespan, Pellegrino Cucciniello, Emanuele Custo, Dominiki Dadatsi, Gaetano De Francesco, Davide Del Giudice, Enrico Dini, Gregory Epps, Michela Falcone, Cristian Farinella, Christian Florian, Andrea Foti,

Simone Giostra, Lorena Greco, Cesare Griffa, Ali Habibian, Marcin Kasiak, Maria e Vincenzo Lamberti, Mario Losasso, Giuseppe Losco, Massimiliano Manno, Emanuele Mantrici, Benedicta Mariani, Maria Elisa Marini, Maria Rosaria Melia, Flavia Migani, Jon Mirtschin, Andrea Morgante, Filippo Moroni, Josef Musil, Charlotte Newman, Federica Ottone, Michael Peng, Emmanuele Jonathan Pilia, Marco Poletto, Sergio Pone, Luigi Prestinenza Puglisi, Faith Robinson, Rinaldo Rustico, Stéphanie Santini, Antonio Pio Saracino, Francesco Schiavello, Patrik Schumacher, Gabriele Stancato, Davide Severi, Danecia Sibingo, Aldo Sollazzo, Alessio Spinelli, Liz Stinson, Giovanni Viggiano, Davide Vitali, Moritz Waldemeyer, Paolo Zilli.

Finally, I would like to thank all the students that I met in these years. This book is dedicated to them.

Arturo Tedeschi

Foreword

In writing the foreword to Arturo Tedeschi's first book, "Parametric Architecture with Grasshopper," the scenario I portrayed on the use of digital tools and generative algorithms in architecture was rather different from today. It was just a few years ago, but the discipline had yet to complete the portrayed revolutionary cycle: both academic research and advanced architectural practices around the world were still in the process of discovering the newness of the digital era with little real focus on the building's industry. From the early nineties to the beginning of the new millennium, architecture went through an incredible period of creative acceleration which sometimes took the discipline progressively beyond product design, car design and art in general. In fact many spatial languages, processes and patterns developed in this period by students and digital pioneering architects with generative design tools, have been subsequently incorporated and produced in other fields with great success. It might then sound a little paradoxical that the discipline which did generate this revolutionary design shift is also the one which took longer to benefit from its implications in real world scenarios like manufacturing and construction. This delay was due to a number of issues intrinsic to architecture: the complexity of a built organism, its delicate bond with socio-economic and political context, and last but not least the timeframe and costs involved in constructing ambitious, unforeseen buildings.

Today the popularity of generative tools and the way students can now absorb information and train their skills has grown exponentially. At the beginning of the new millennium computational training were yet to be included within academic programmes and the average student could only passively watch the results of the new digital era on the internet, considering those visionary experiments as part of an elitarian education only accessible by attending expensive schools.

A changed paradigm can now be depicted. Not alternative or contrasting, just evolved. Looking back at the advances of digital tools in architecture in the last decade two aspects are clear. Firstly, computational research and knowledge have been widely spread thanks to those architects, researchers and universities who shared their experiences. The web has become an invaluable source for tutorials published in blogs or websites which are inspiring to young generations of digital designers. At the same time students have the opportunity to learn processes and theory through workshops arranged by experts and visiting schools showcasing didactic methods used by the most advanced universities all over the world. This gives students, coming from a more conservative background, the opportunity to practically understand this new approach to design. In parallel

the wider interest towards the topic has triggered the activation of many advanced architecture programs, computational departments and digital fabrication facilities from international universities which were previously known to foster a more conservative approach towards architectural research. Better awareness of the creative possibilities of digital tools has also strengthened the global sensibility of architects for the rising aesthetic of computational design which is increasingly becoming accepted as a global emerging style.

The affirmation of digital design culture de facto concludes the speculative era started in the nineties with pioneers using animation software to generate parametric forms and ending with agent-based designs programmed using Processing. Thirty years enriched by highly experimental architecture which have changed forever the way we perceive and design spaces.

Second aspect, which I believe could possibly trigger an evolution of the way architects make use of digital design, is the shift towards an integrated use of computational design which is already underway. Trying to contrast an unstable socio-economic scenario, the architects' main interest is moving from a mere pursuit for spatial complexity and newness to embrace collaborative workflows, sustainability and constructability.

The outcome of architectural research can no longer be a beautiful image printed in a magazine or a book, waiting to be eventually converted in reality after years of reverse engineering and money spent. Digital design can and must confront reality as early as in concept stages giving answers to both performance and functional criteria and achieving solutions tailored to overcome the constraints of contemporary building industry.

The rise of pre-assembled pipelines on construction sites, the necessity of reducing the number of special parts, the economical restrictions deriving from an era of constant recession and, more notably, the necessity of planning a sustainable lifecycle for the building are all part of the equation. These are concepts which can be seen as in opposition to the advancement of architectural research; however, digital architecture must be brought to this next level: form must reflect the integration between multiple disciplines. Ultimately, leading to a strong convergence between architectural languages, structural performance as well as building sub-systems. For the designer this means, to evaluate the output of their Grasshopper definition, Mel or Processing script for variables beyond personal spatial vision. In order to succeed in the current market it is crucial to go beyond. It is essential to dominate tools and processes in order to achieve a higher level of coordination within the design.

The need for digital designers and artists to redefine their agenda is quite well described in the article written by Michael Parsons for Design Review: "Tolerance and Customization: a Question of Value". Even if I cannot completely adhere to author's vision I think he is right when arguing that

as digital tools become more popular and ordinary the complex forms made possible by these tools will lose their status of cutting-edge research unless their aesthetic and spatial qualities are supported by meaningful constructive and functional strategies. Parsons sites as an emblematic case the Voronoi algorithm. Formerly a territory only explored by the most skilled computational designers, the Voronoi algorithm has now become an accessible tool. It is obvious that only a deep understanding of its geometrical properties and intelligent implementation of them can push forward and justify its use. Anything else is just a mere proof of knowing how to use a definition programmed by others, in which case, as Michael Parsons says, "there is no skill involved in it" and I would add there is no progress either. The goal, instead, must be to achieve a perfect equilibrium between complexity and coherency.

In this book Arturo Tedeschi delivers a text that is both a theoretical and practical reference for architectural professionals and students. The text demonstrates techniques of advanced digital form finding workflow, which enables readers to take a step forward, towards the mature implementation of digital tools in architecture. Computational design has refined the architectural language, and it is now time for designers to make dramatic yet meaningful poetry with it.

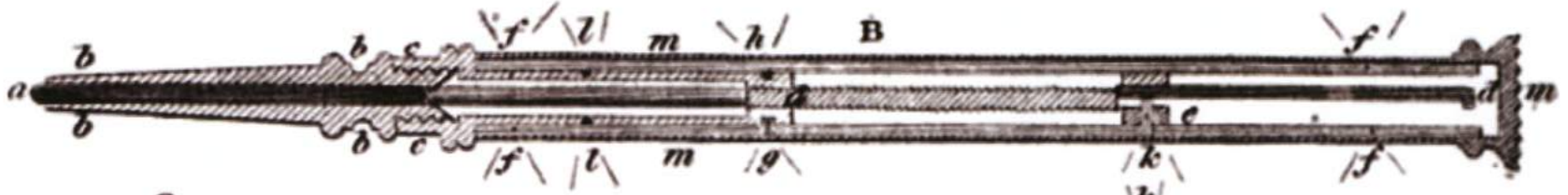
Fulvio Wirz

Fulvio Wirz, born in Naples in 1977, graduated in architecture from Federico II University in Naples. He earned a master in "Public Areas Design" in 2003 and in "Architectural Design for Photovoltaic Integration" in 2004. Since 2002 complemented the work experience in his father's studio with teaching at design courses with Lucio Morrica and starts a personal research focused on relations between architecture and new media, put into practice through several competitions and culminated with the achievement, in 2008, of the PhD in Architectural Design with the thesis "Digital representation and architectural composition". Since 2005 he cooperated at Zaha Hadid Architects office in London where he has progressed his career over the years earning the position of Associate in 2014. Within ZHA Fulvio has won prestigious international competitions as Seville's library, the Masterplan for "Kartal-Pendik" area in Istanbul, the "Lilium Tower" in Warsaw, the "Eli & Edythe" museum in Michigan, the KAPSARC (King Abdullah Petroleum Studies and Research Center) in Saudi Arabia. He has been project designer on "Aura", an installation exhibited at Biennale di Venezia in 2008, and on a number of cutting edge product designs including Zephyr Sofa, Z-Chair and Liquid Glacial Table. Currently he's working on several projects, such as the design of a mosque in Kuwait and the KAFD Metro Station in Riyadh. Fulvio has been co-leading Research Cluster 6 at the GAD (Graduate Architectural Design) at Bartlett School of Architecture and has been teaching at a number of universities in Italy and UK. Since 2013 he is lecturer at the London South Bank University and he has been guest lecturer at AA Rome Visiting School.

Detail of the specification of the patent granted in december 1822 to Sampson Mordan and John Hawkins, for improvements on pencil-holders.

Fig. 1. A.


a
b
b
c
m
m


B
a
b
b
b
b
c
c
f
l
m
h
f
f
l
m
g
k
f
d
m
e


C
a
b
b


k
e
G
e
d


l
D
h
c
G
l
h


e
H
k


d
F


K
n


E
d
d


L


f
g
k
f
I


M
N

Fig. 2.

AAD

Algorithms-Aided Design

From traditional drawings to the parametric diagram

"Architects do not make buildings, they make **drawings** of buildings".

Robin Evans

Architects have always drawn before building, an act that diferentiates architecture from the mere construction. Drawings have been the architects medium to organize ideas, resources, space, etc. and represent the architects' faculty to predict design outcomes. As methods of representation have evolved, new styles have emerged. Tools such as perspective in the Renaissance and projective geometry in Modernism have marked leaps forward in design. However, these tools have been dependant on a stable set of instruments for centuries: paper, drawing utensils, ruler and the compass. In this model each creative act is translated into a geometric alphabet by gestures which establish a direct link between the idea and the sign.

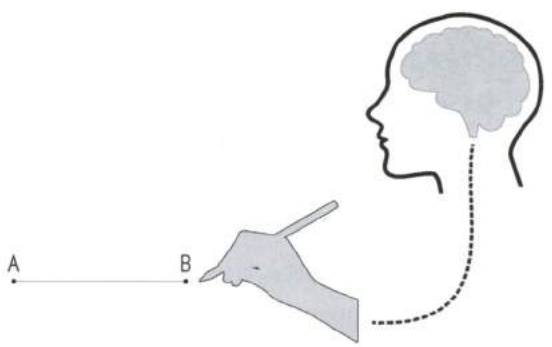


FIGURE 0.1

The act of drawing is a natural gesture when executed using traditional drawing tools which establish a direct link between ideas and signs. A natural interaction is characteristic of those tools which can be considered as a *hand mould*.

An additive process

The traditional drawing is an **additive process**, in which complexity is achieved by the addition and overlap of independent signs traced on paper. No **associative relations** can be managed. The **internal consistency** of a drawing is not guaranteed by the medium, but is entrusted to the designer. As follows, the drawing is not a **smart medium**, but rather, a **code** based on standards and conventions.

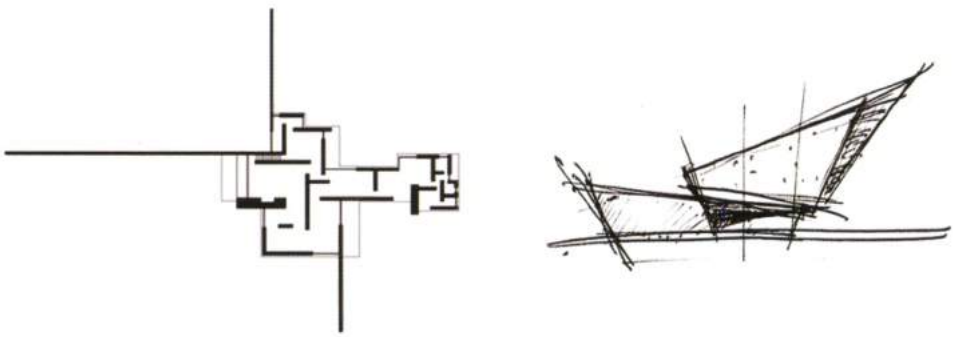


FIGURE 0.2
The traditional drawing is based on adding and overlapping independent signs on a paper. The meaning and the overall consistency of these signs is entrusted to the designers and is based on conventions. The drawing by Mies van der Rohe (on the left) is a "plan" while the sketch on the right is a "draft", nevertheless both are (ontologically) just ***signs on a paper***.

The additive logic of the traditional drawing implies two limits: first, the act of drawing differs from cognitive mechanisms underlying the creative process, which works by establishing interrelations rather than adding information. Second, the drawing process excludes physically relevant aspects that in the real world drive the generation of forms. For example, the traditional drawing cannot manage **forces** (such as gravity) and constraints which affect and restrict deformations and displacements. These limits have restricted the exploitation of the drawing and designers have been forced to reiterate definitive tectonic systems rather than innovating. Initially these limits were not overcome by the computer; CAD software simply improved the ability to perform repetitive tasks without affecting the method of design. Similar to traditional drawing, CAD entrusted the designer to determine the overall consistency by adding digital signs or geometric primitives on a digital sheet/space and controlling **CAD *layers*; this method can be seen as the translation of the additive logic within the digital realm.**

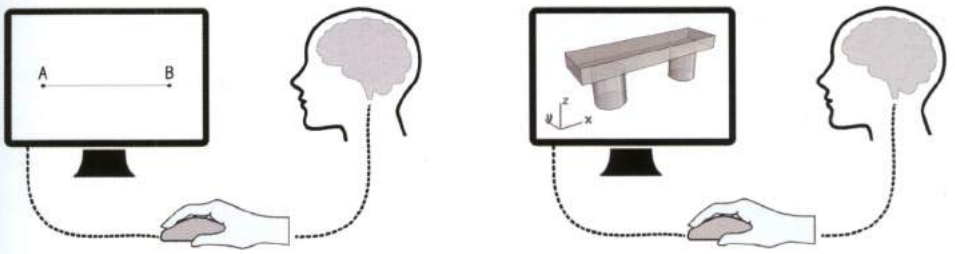


FIGURE 0.3

The mouse is still an extension of the brain. It simulates the "presence" of the hand in the digital environment.

From 60's, the architecture avant-garde tried to "force" drawing's limits using several methods to represent forces and processes that drive the generative process. For example, Eisenman's diagram for House IV impressed the entire sequence of geometric operations that led to the final object.

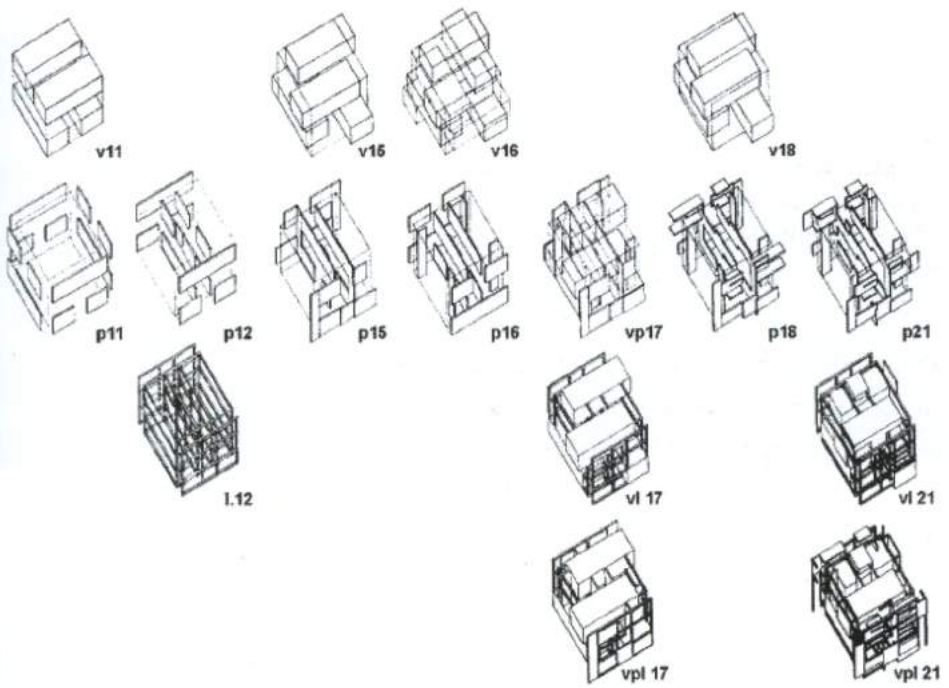


FIGURE 0.4

Peter Eisenman, House IV, Falls Village, Connecticut, 1971.

From conventional drawing to the analogue (smart) apparatus

Despite the limitations, drawings have been the stable medium of architecture over the centuries and this was possible as the architects have relied on **typology**, i.e. the use of well proven, preconceived solutions and tectonic systems. Typology made the drawing not only a communication medium but a system that enabled designers to explore and refine variations (**form-making** approach) within a specific set of formal and structural constraints.

The conventional drawing was first attacked by a new approach, the **form-finding** – emerged in architecture in late 19th century – which aimed to investigate novel and optimized structures found through complex and associative relations between materials, shape and structures.

Pioneers like Gaudi (1852-1926), Isler (1926-2009), Otto (1925-) and Musmeci (1926-1981) have rejected typology and looked to self-formation processes in nature as a way to organize buildings. Since the form could not descend from proven solutions, the traditional drawing could not be used as a tool to predict design outcomes.

FIGURE 0.5
Heinz Isler. Service Station in Deitingen, Solothurn, Switzerland (1968). Image by David P. Billington.

For this reason form-finding pioneers relied on physical models such as: soap films which found minimal surfaces, and suspended fabric which found compression-only vaults and branched structures. In other words, the drawing as a medium to investigate form was replaced with physical form finding relying on analogue devices which demonstrated how dynamic forces could mold new self-optimized architectural forms.

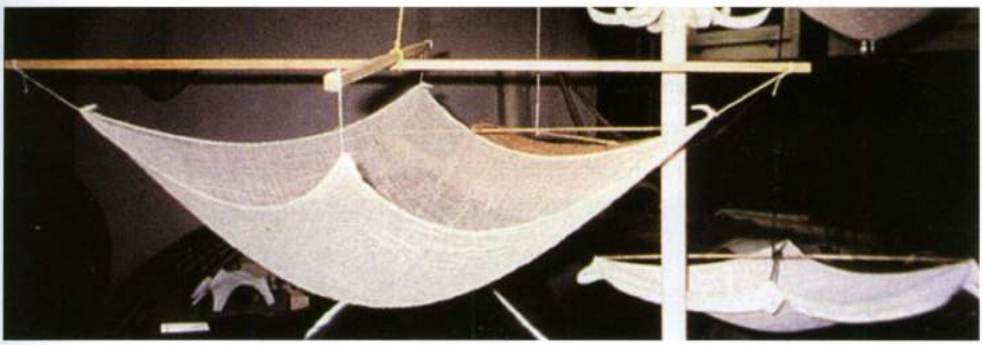


FIGURE 0.6
Forces and forms are correlated.

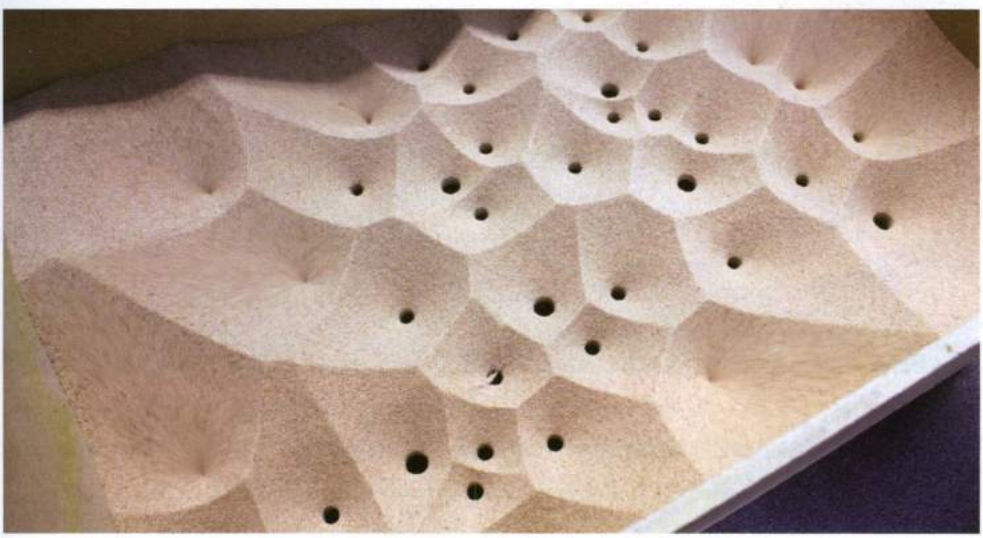


FIGURE 0.7
"Any granular material falling from a fixed point forms a cone on the surface below and a funnel within the granulate mass with the same angle of inclination, the natural angle of repose, 35 degree." Frei Otto, 1972. Sand Experiment Inspired by Frei Otto - WeWantToLearn.net, studio ran by Toby Burgess and Arthur Mamou-Mani at the University of Westminster, student: Jack Munro.

Over the last decades the increasing complexity of buildings has made ***form-finding*** an important strategy in determining the shape and form of indeterminate structures. Structural optimization through physical modeling was **mono-parametric** (gravity based) and marked a trajectory towards **multi-parametric** form-finding which aims to interact with heterogeneous data: geometry, dynamic forces, environment, social data.

Parameters: from additive to associative logic

Luigi Moretti, the Italian architect, invented the definition for "**Parametric Architecture**" in 1939. His research on "*the relations between the dimensions dependent upon various parameters*" culminated in an innovative exhibition of his models of stadiums for soccer, tennis and swimming at the 1960 Twelfth Milan Triennial. Moretti's design parameters were linked to viewing angles and economic feasibility in these projects: the final shape was generated by calculating pseudo isocurves, that attempted to optimize views from every position in the stadium.

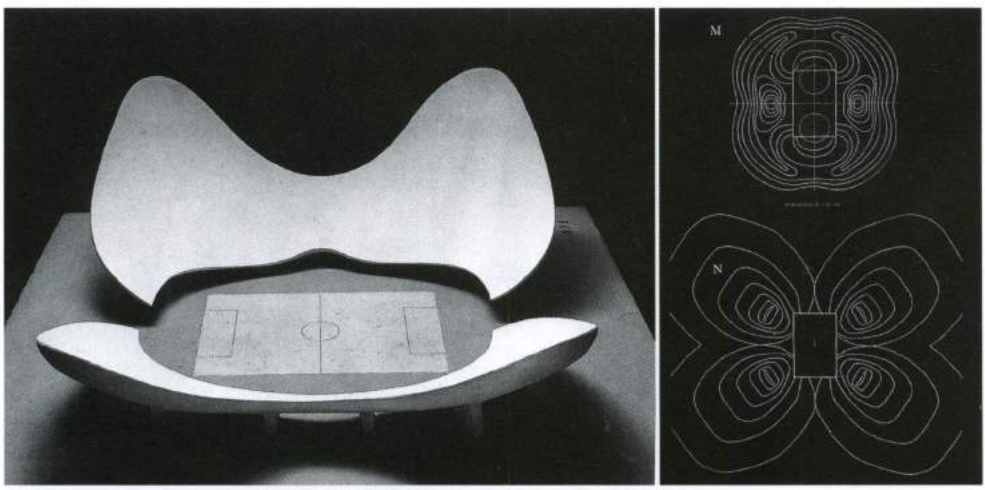


FIGURE 0.8
"Architettura Parametrica" research. Milan Triennale exhibition, 1960. Solution for a soccer stadium and diagrams drawn to generate the geometry.

Moretti's research was a collaboration with the mathematician Bruno De Finetti, wherewith he founded the Institute for Mathematical Research In Architecture (I.R.M.O.U.). Moretti said:
"The parameters and their interrelationships become [...] the code of the new architectural language, the "structure" in the original sense of the word [...]. The setting of parameters and their relation must be supported by the techniques and tools offered by the most current sciences, in particular by logics, mathematics [...] and ***computers****. Computers give the possibility to express parameters and their relations through a set of (self-correcting) routines".*[1]

NOTE 1
F. Bucci and M. Mulazzani, *Luigi Moretti opere e scritti* (Milano: Electa, 2006), 204–208.

It is evident from this quote, that Moretti immediately understood the potentials of the computer applied to design. Following Moretti, the first application for design utilizing the computer occurred in 1963. The American computer scientist Ivan Sutherland developed the ***Sketchpad***, defined as "A Machine Graphical Communication System," creating the first interactive Computer-Aided Design (CAD) program.

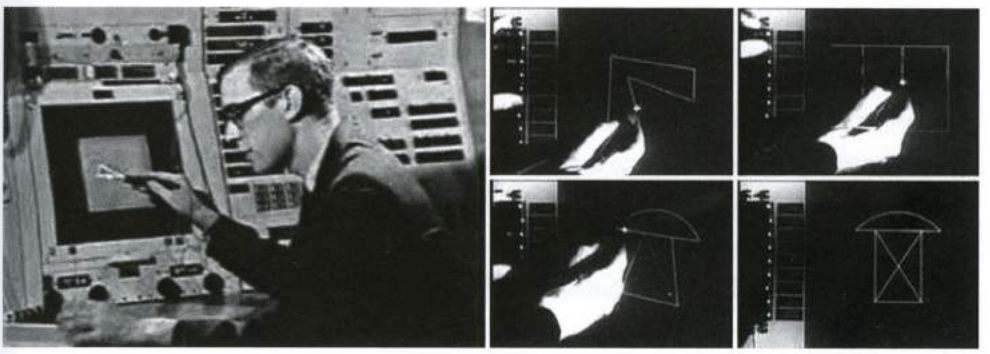


FIGURE 0.9
Ivan Sutherland on MIT Lincoln Labs' TX-2 computer (1963). The *Sketchpad* interface.

Considered as one of the most influential computer programs ever written, the sketchpad was designed to test human-computer interaction and allowed designers to draw basic primitives such as: points, lines and arcs, using a light-pen for input. The program featured many of the now typical CAD operations such as: blocks managing, zoom and snaps. Moreover, it was based on an advanced **associative logic**, the so called *atomic constraint*.

It was an innovative feature which facilitated links between objects; for example, if two lines (or more appropriately two vectors) were drawn starting from the same point A, every movement of A implied change in magnitude and direction of the lines. Constraints such as points could be combined to generate relationships between objects, overcoming the limits of the **additive logic** of traditional drawings.

The introduction of the computer to design by Moretti, and the graphical interface of Sutherland marked a revolution in architectural design techniques and moreover it upgraded the architects tools. However, the innovations brought by early CAD programs were not immediately embraced by commercial software for almost three decades.

For instance, the associative capabilities introduced by the Sketchpad were not embedded in commercially successful software, such as Autocad (1982). Autocad met the architects need to speed up repetitive tasks and manage multiple drawing layers, by in effect, digitalizing the drawing board.

The next important step forward occurred in 1987, with the introduction of Pro/ENGINEER® software,

developed by Samuel Geisberg for mechanical system design. The program allowed users to associate tridimensional parametric components which were controlled by user input constraints. For example, it was possible to create a link between a rivet and the relative hole. The user changing the rivet input size implied a propagation of modifiers which updated the tridimensional model as well as the bidimensional output.

Pro/ENGINEER reduced the cost of making design changes, and overcame the rigid constraints of tridimensional modeling.

The most profound progress has happened from the late 1980's to present day. Academic research and avant-garde practices - trying to escape simple **editing** limitations of software applications - explored new ways to manipulate software "from the inside" aiming to find unexplored solutions and forms through **programming**. Many designers soon realized that more sofisticated programs could manage **complexity beyond human capabilities** by structuring routines and procedures.

This type of modeling relies on programming languages which express instructions in a form that can be executed by the computer through a step-by-step procedure: the **algorithm**.

Algorithmic modeling

What is an algorithm? An algorithm[2] is a procedure used to return a solution to a question - or to perform a particular task - through a finite list of basic and well-defined instructions. Algorithms follow the human aptitude to split a problem into a set of simple steps that can be easily computed, and although they are strongly associated with the computer, algorithms could be defined independently from programming languages. For example, a recipe can be considered as something similar to an algorithm. We can set a procedure for cooking a chocolate cake, based on a simple list of instructions:

0. Mix ingredients;
1. Spread in Pan;
2. Bake the cake in the oven;
3. Remove the cake from oven;
4. Cool.

Nevertheless, such a procedure cannot be properly considered an algorithm since the instructions

NOTE 2

The term "Algorithm" is named after the 9th century Persian mathematician Al-Khwarizmi.

are far from being well-defined and contain ambiguities: "mix ingredients" but which ingredients? How long should the cake cook? This basic example points out some important properties of algorithms:

- *An algorithm is an **unambiguous** set of properly defined instructions.*
  Algorithms depend on entered instructions. The result will be incorrect if the algorithm is not properly defined. Put another way, if steps in the cake are inverted or skipped, the chances of a successful cake diminish.

- *An algorithm expects a defined set of input.*
  Input can be different for type and quantity. The step {0} requires ingredients, the step {2} requires quantitative information such as baking temperature and time. Moreover, each input has a precondition, e.g. a requirement which must be met, such as a range of baking temperatures, for example: 160°C – 200°C.

- *An algorithm generates a well defined output.*

FIGURE 0.10
Schematic representation of an algorithm.

If an ambiguos recipe leads to an inedible cake, in the digital realm:

- *An algorithm can produce error messages and warnings within the specific editor.*
  Input is specific. If preconditions are not met, e.g. numbers are imputted instead of text, the algorithm will return an error.

Although algorithms are often studied abstractly they harness the potential of the computer which has the capacity to perform tasks according to a set of instructions. When algorithmic calculations are executed by a computer, a specific **editor** is used to type instructions. Editors can be **standalone** applications or embedded in a software application. For example, standalone editors include C#, Python etc. and embedded editors are script editors provided by programs such as Rhinoceros and Autocad that allow users to write instructions to automate tasks.
Algorithms consist of different classes, an algorithm class which leads to a number is called a

computation procedure, while an algorithm that generates a *yes* or *no* is called a *decision procedure*.

Algorithms can also lead to geometries. For instance, if an integrated editor is used within CAD or another modeling software, a 3D geometry is created by manipulating the standard set of primitives provided by the software or procedurally defined by a sequence of instructions. For instance, a line can be defined by two points, a start and an end; points in turn can be defined by their coordinates {x,y,z}. For example, a vase model can be defined as a revolution of a profile curve around an axis, and more complex objects can be obtained by establishing a set of rules.

Objects are no longer manipulated with a mouse, instead they are defined by procedures expressed in a specific program language: AutoLisp® in Autocad®, RhinoScript® in Rhinoceros®, MEL® in Maya® or other cross platform languages such as Python®.

Such an approach – usually referred to as **scripting** – is completely new for designers and transforms the link between the idea and the final output.

Scripting consists of two working environments:

- the editor (A);
- the 3D modeling environment (B).

Moreover it produces **two outputs**:

- the algorithm;
- the output of the algorithm, constituted by **associative** 3D or 2D geometry.

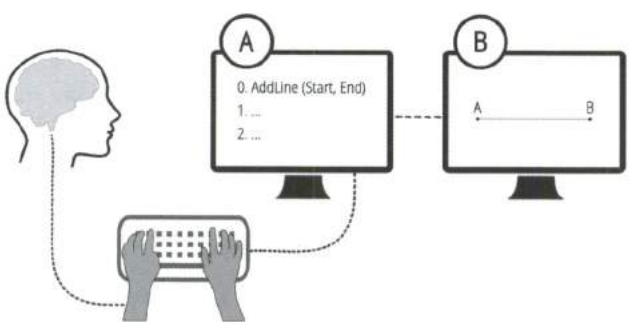


FIGURE 0.11
The algorithmic modeling based on scripting consists of two main "windows": the editor and the 3D modeling environment.

The final output is not just a "digital sign" but it can be considered as an interactive digital model responding to variations in the input by manipulating the entire system. For example, if the points

coordinates are changed from {x,y,z} to {x^1,y^1,z^1} of the mentioned line, the algorithm maintains the established relationship that the line is defined by the two points not their location. Algorithms establish associative relations between different entities such as numbers, geometric primitives and data. For example, complex geometries can be defined by an unambiguous sequence of instructions which drive interrelations. **Algorithmic design enables users to design a process rather than just a single object.**

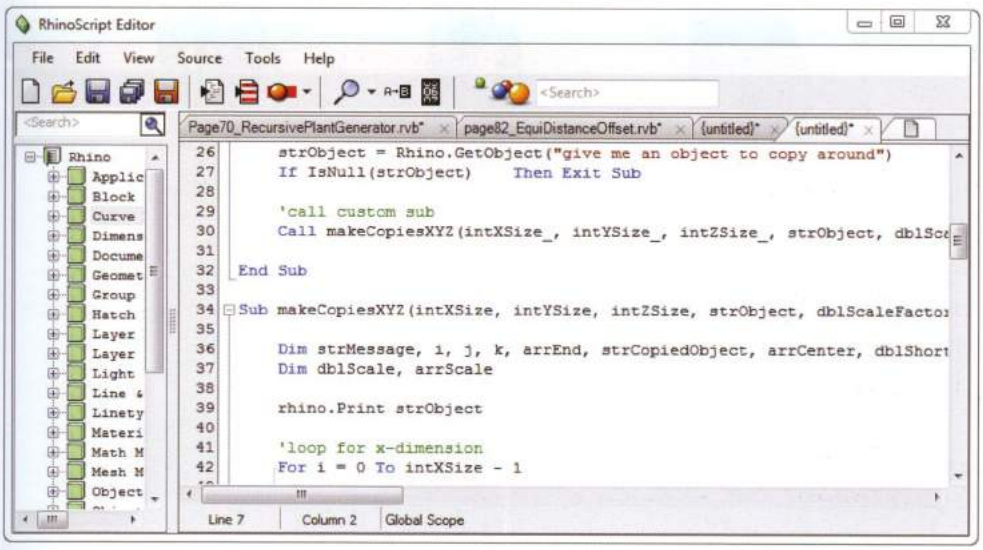


FIGURE 0.12
The RhinoScript editor.

Bruce Mau in his 1998 *Incomplete Manifesto for Growth*, states that a *"process is more important than outcome. When the outcome drives the process we will only ever go to where we've already been. If process drives outcome we may not know where we're going, but we will know we want to be there"*. Mau's quote summarizes the core concept of algorithmic design; the potential to generate and control design-complexity beyond human capabilities. A set of well defined associative rules and constraints can lead to unprecedented shapes or unpredictable results that are coherent with the parameters established. Algorithmic design allows designers to find new solutions and step beyond the limitations of traditional CAD software and 3D modelers.

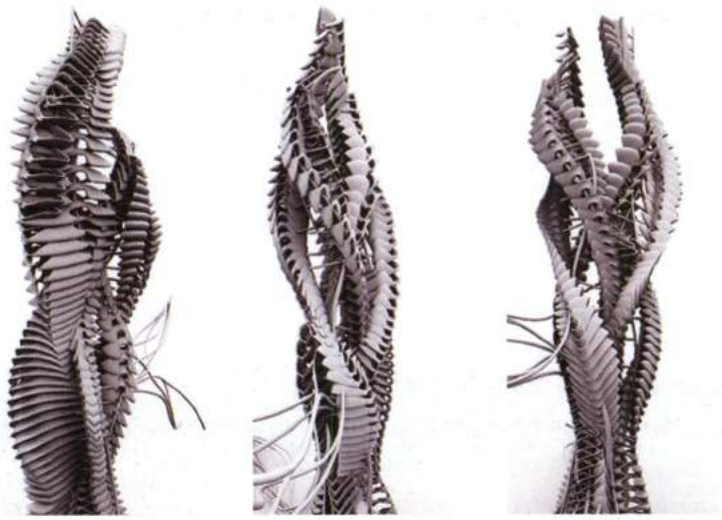


FIGURE 0.13
Parametric Urbanism - Dominiki Dadatsi, Fountoulaki Elrini, Pavlidou Eleni. The image shows three different configurations of an algorithmic-designed tower.

Algorithms can define every type of geometry. The method to construct geometries procedurally is based on writing a rough draft and translate it into a programming language. For example, the image below can be sketched by writing the following list of instructions:

0. Draw four circles;
1. Subdivide the four circles into N parts; we get N points for each circle;
2. Connect the corresponding points.

The same element could be defined by different parameters, but it is natural to write an algorithm in a way that establishes relations between the variable parts of an object. In the example the number of lines is affected by the number of subdivisions (N), which is the main parameter. Nevertheless, the algorithm is still ambiguous since it does not specify unique origins with respect to the z coordinate: $\{x,y,z^1\}$, $\{x,y,z^2\}$ and $\{x,y,z^3\}$, a radius for each circle (r^1), (r^2) and (r^3), and the method of connecting the lines. These refinements are made when transitioning from the rough draft to the final algorithm.

The Parametric Diagram as a smart medium

In recent years many software houses have developed **visual** tools in order to make scripting more accessible to users with little to no programming skills. In effect, associative rules and dependencies can be expressed using a graphical method based on **node diagrams**.

Sutherland's *Sketchpad* represented all the constraints defined during the drawing process. Through a special diagram - a *flow chart* - the user could not only visualize the tree of dependencies but could manipulate the graph with instant effects on the drawing.

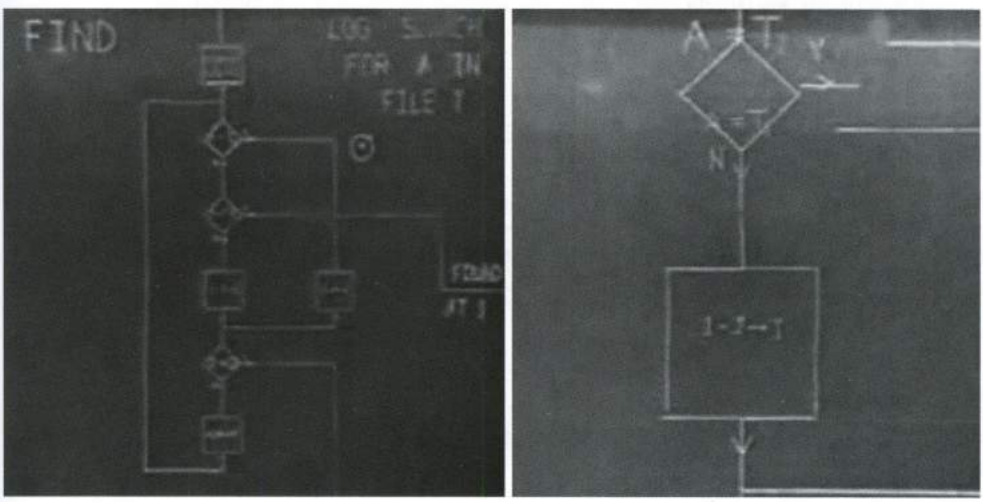


FIGURE 0.14

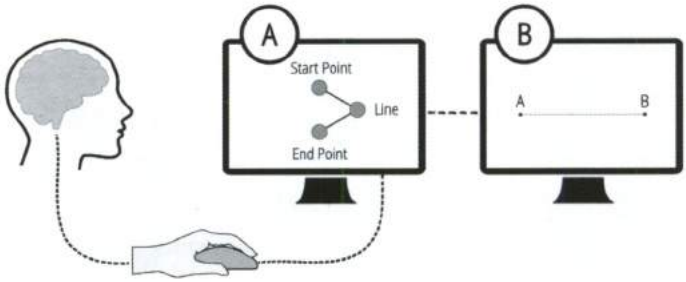
The Sketchpad (1963) provided a graphic visualization of design-constraints through a diagram called *flow chart*.

Many software have enabled users to interact with digital objects either by a direct manipulation and via node-based diagrams. Node based software systems such as Generative Components® by Bentley Systems and Grasshopper® by Robert McNeel & Associates, are two softwares that enable users to build up complex geometries by associating parametric primitives. Visual scripting makes possible a process were a line can be built by connecting two point objects, a square by connecting four line objects etc.

"In principle any conceivable network of relations between a given set of element attributes can be constructed".[3]

NOTE 3

P. Schumacher, *The Autopoiesis of Architecture, A New Framework for Architecture*, (John Wiley & Sons, 2010), vol. I, p. 353.



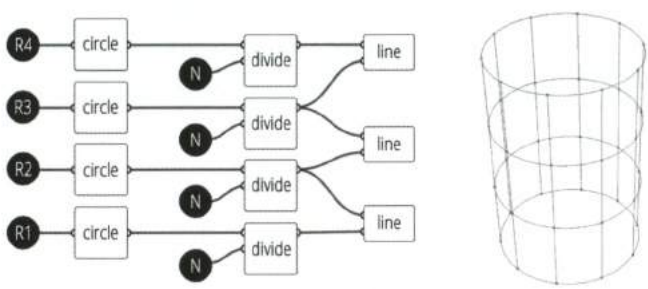
Similar to scripting, visual scripting is based on two main working environments:

- the Visual editor (A);
- the 3D modeling environment (B).

Such a process generates **two outputs**:

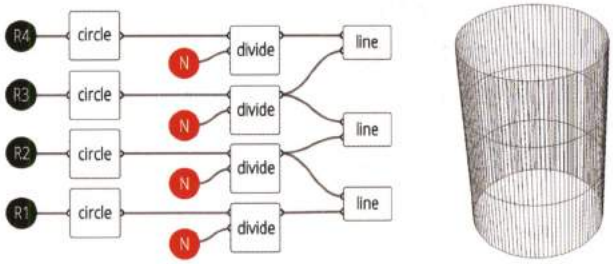
- the node diagram, also called **parametric diagram** or visual algorithm;
- the output of the parametric diagram constituted by parametric 3D or 2D geometry.

Node diagrams can be used to create geometries. For example, the following figure is the visual transposition of the algorithm "drafted" at page 26.

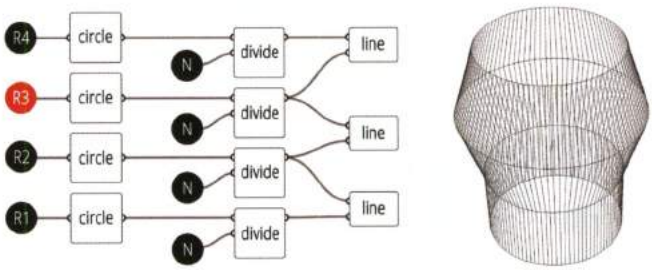


The diagram consists of nodes and connections. Square nodes are the main functions: draw a circle, divide a circle, create a line. The circular nodes are the parameters: the radius of each circle, and the number of subdivisions. The diagram's output is the same geometry generated through the step-by-step procedure shown previously.

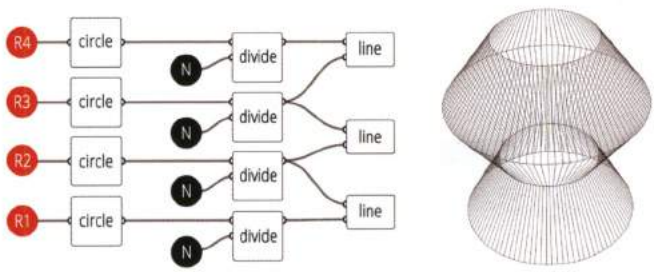
The advantage of a node diagram resides in the intuitive logic which allows you to quickly interact with parameters. For example, if the N parameter is modified, more lines are generated.



The geometry can be further modified by manipulating one of the (R) parameters, the radius of (R3) is increased in figure.



The geometry can be once again be modified by manipulating (R1, R2, R3, R4) as shown in figure.



The parametric diagram has the potential to create associative models that explore multiple configurations through control of the input parameters.

Patrick Schumacher is quoted:

"While the attributes of the graphic/digital primitives [...] are fully determined and fixed at any time, within the parametric diagram they remain variable. This variability might be constrained within a defined range

on the basis of associative functions that imbue the diagrammatic process with an in-built intelligence"[4]. The parametric diagram can be considered a smart medium for architecture and design, since it provides an internal self-consistency transposed in a graphic language which can be easily manipulated, enabling designers to explore form-finding and form-making strategies.

Jerry Laiserin is quoted:

"Form-making, loosely defined, is a process of inspiration and refinement (form precedes analysis of programmatic influences and design constraints) versus form-finding as (loosely) a process of discovery and editing (form emerges from analysis). Extreme form-making is not architecture but sculpture [...]. Extreme form-finding also is not architecture but applied engineering, where form exclusively determined by function".[5]

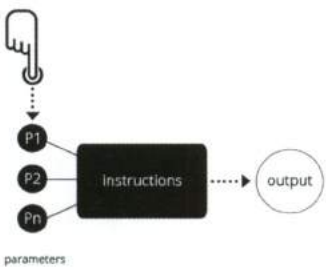


FIGURE 0.15
Conceptual representation of the form-making approach.

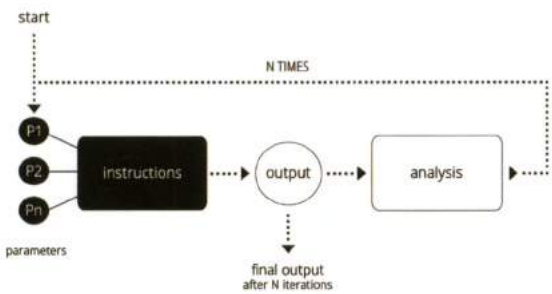


FIGURE 0.16
Conceptual representation of the form-finding approach.

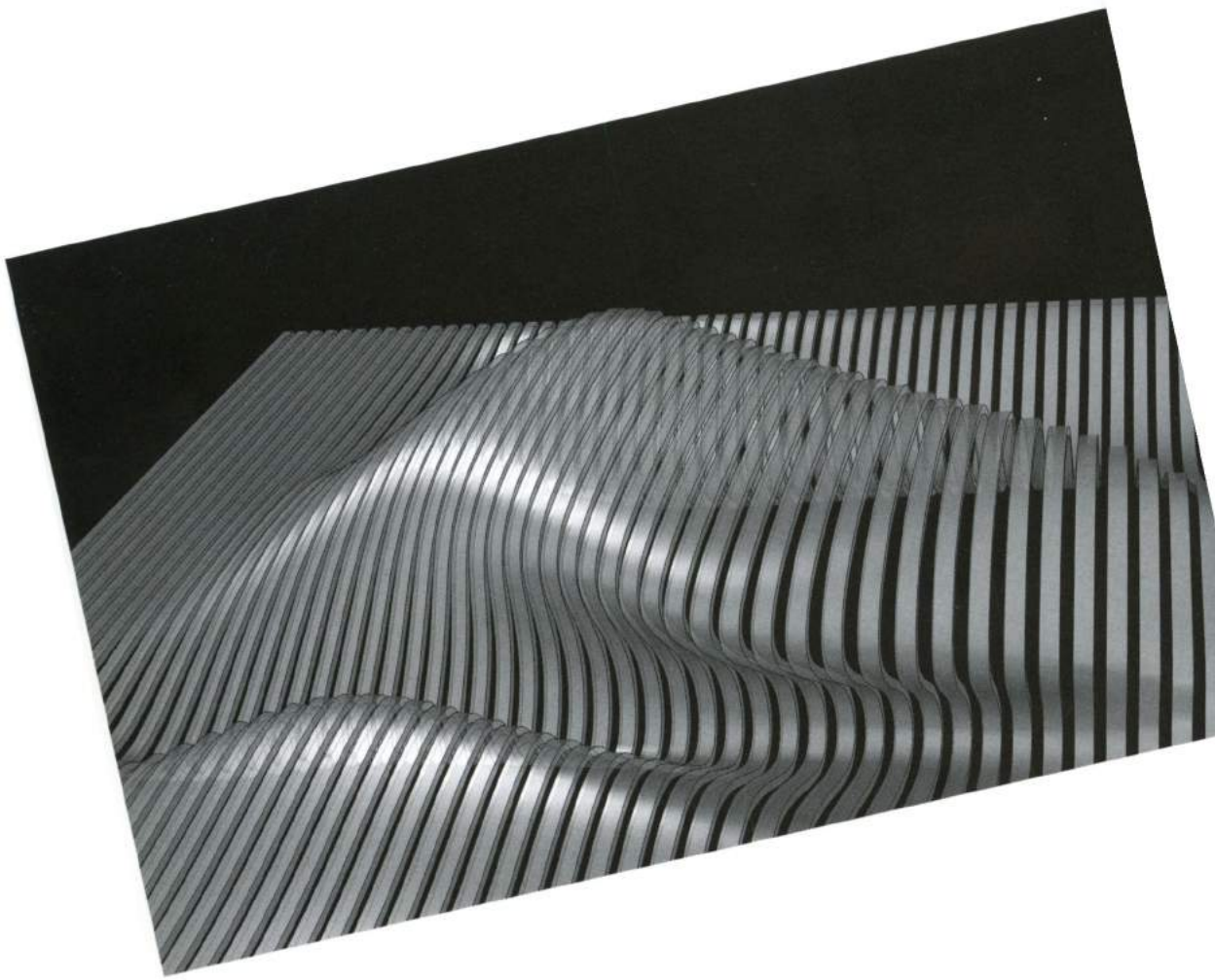
NOTE 4

P. Schumacher, *The Autopoiesis of Architecture, A New Framework for Architecture* (John Wiley & Sons, 2010), vol. I, p. 352.

NOTE 5

J. Laiserin, 2008, *Digital Environments for Early Design: Form-Making versus Form-Finding*. First International Conference on Critical Digital: What Matters(s)? - 18-19 April 2008, Harvard University Graduate School of Design, Cambridge (USA), pp. 235-242.

Architects do not make buildings, they make
diagrams of buildings



1_algorithmic modeling with Grasshopper®

> "Grasshopper seems to be winning out in the competitive struggle for domination as the preferred tool for scripting, at least in the avant-garde segment of the discipline".
>
> Patrik Schumacher

The aim of this publication is to provide techniques and strategies which enable designers to master visual scripting utilizing **Grasshopper®**, one of the most popular and advanced algorithmic modeling tool.

This book is intended to guide readers through the essential concepts of algorithmic modeling both for *form-making* and *form-finding*.

Grasshopper, a plug-in for Rhinoceros®, is a node-based editor developed by David Rutten[6] at Robert McNeel & Associates. Created in 2007 for Rhino 4.0 – originally named as *Explicit History* – was re-branded as Grasshopper in 2008. Within a few years the plug-in gained a vast community of users and developers, including students, academics, and professionals. Grasshopper is available as a free download and it runs on a licensed copy of Rhinoceros 5.0 or higher.

NOTE 6

David Rutten is a graduate of TU Delft Architecture and Urbanism faculties. He works for Robert McNeel & Associates since 2006 on several programs, the most important of which is the Grasshopper visual programming environment for Rhinoceros 3D. See A. Tedeschi, 2010, Interview with David Rutten. MixExperience magazine, "Tools" issue.

http://content.yudu.com/Library/A1qies/mixexperiencetoolsnu/resources/index.htm.

Grasshopper's is an unrivaled platform for formal exploration, that benefits from:

- **Wide, dynamic and growing community**

  Grasshopper is not just a piece of software, it includes a dynamic network of users that share works, knowledge, ask questions, and discuss challenging problems at **www.grasshopper3d.com**.

- **Constant updating**

  Grasshopper benefits from constant updating and improvements. Moreover, bug fixing and new features are commonly based on users feedback.

- **Ecosystem**

  A wide set of plug-ins developed for Grasshopper are available from independent programmers. Loosely speaking, Grasshopper benefits from a plug-in ecosystem that extends the software's potentials. For example plug-ins are available for dynamic simulations, physics, structural and environmental analysis.

- **Software interaction**

  Grasshopper has the potential to interact with other software; not only for file-compatibility and interchange, but for real-time interaction between the algorithmic modeling environment and external software (Excel®, Photoshop®, Revit®, Ecotect® etc.).

- **Hardware interaction**

  Grasshopper's plug-ins enable interactions in data input/output between Grasshopper and hardware (Arduino®, Kinect® etc.).

This book will cover visual algorithmic modeling using Grasshopper, from the basic concept to more advanced strategies. The main ambition is to provide the computational skills - based on a solid understanding of mathematics, logics and geometry - required to face the new design challenges.

1.1 Prerequisites and installation

Grasshopper is not a standalone application, it operates as a plug-in for Rhinoceros. The installation file can be downloaded from the website **www.grasshopper3d.com** (*download* section). Presently, Grasshopper exclusively runs on Windows OS (Grasshopper is not currently compatible with Mac OS). Grasshopper is offered as a free download – without an expiration date – and is required to run on a licensed copy of Rhinoceros 5.0 or higher. Future releases of Rhinoceros might embed Grasshopper as a native feature.

Once installed, Grasshopper can be opened by typing *grasshopper* in the Rhino command line.

1.2 Grasshopper user interface

The Grasshopper editor consists of a window (A) that **always works in parallel** with the Rhinoceros 3D-modeling environment (B). Within the editor (A), users can build visual algorithms (C) by properly connecting graphical objects, called **components**. Components are actually the nodes of a parametric diagram that defines and controls a 3D geometry (D) which is displayed in the Rhinoceros window (B). Components represent primitives, geometric operations, logical functions etc.

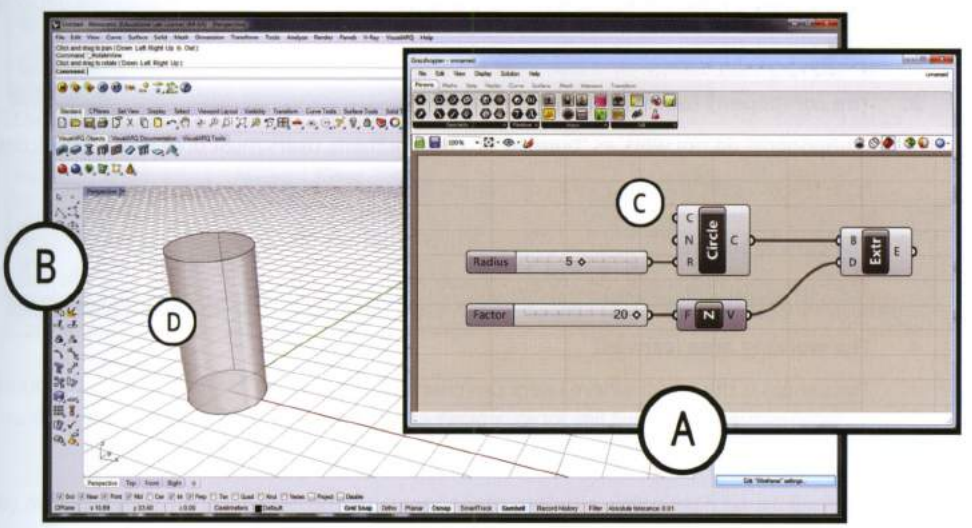
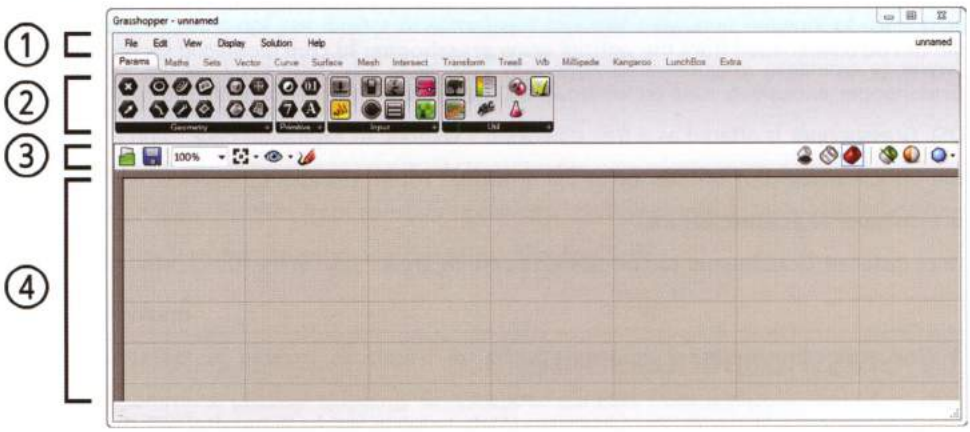


FIGURE 1.1

The image shows the Rhino modeling environment (B) and the Grasshopper editor (A) which is a window that works in parallel with Rhinoceros. D is a geometry "generated" by the visual algorithm (C).

The Grasshopper window is split into four sections, as shown in the following image.



1. **The menu bar**

   Following Windows OS typical bar layout, the menu bar performs basic operations (open, save, etc). On the far right, users can find the *file browser button* which can be used to switch between different loaded files; Grasshopper allows multiple files to be loaded simultaneously. By default a new file is displayed as *unnamed* until it is saved under a different file name.

2. **The component tabs**

   Components do not work as "buttons". To enable them, users must drag the relative icons onto the working area (4). Every component becomes a node of a visual algorithm.

3. **The canvas toolbar**

   The canvas toolbar hosts visualization options.

4. **The working area (canvas)**

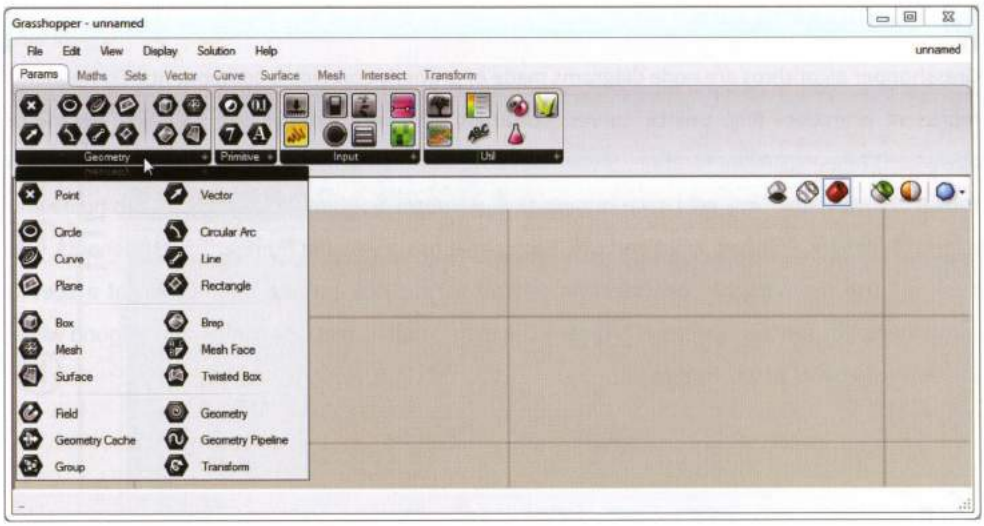
   The canvas is the editor where users can create algorithms.

1.2.1 Component tabs

Grasshopper algorithms are node diagrams made of properly connected components. Components represent primitives (e.g. points, curves, surfaces), geometric entities (e.g. vectors), geometric operations (e.g. extrusion, rotation, revolution) and other categories. They are grouped in several **tabs** (*Params*, *Maths*, *Sets*, etc.) each organized in **panels.** For example, the *Params* tab holds four panels: *Geometry*, *Primitive*, *Input* and *Util*. Each panel has a variable number of components. This book will use the notation *component name* (Tab > Panel) to indicate the location of a specific component. Put another way, *Line* (Params > Geometry) means that *Line* component is found within the *Geometry* panel of the *Params* tab.

FIGURE 1.2
The image shows three different component tabs – Params, Vector and Curve – with relative panels.

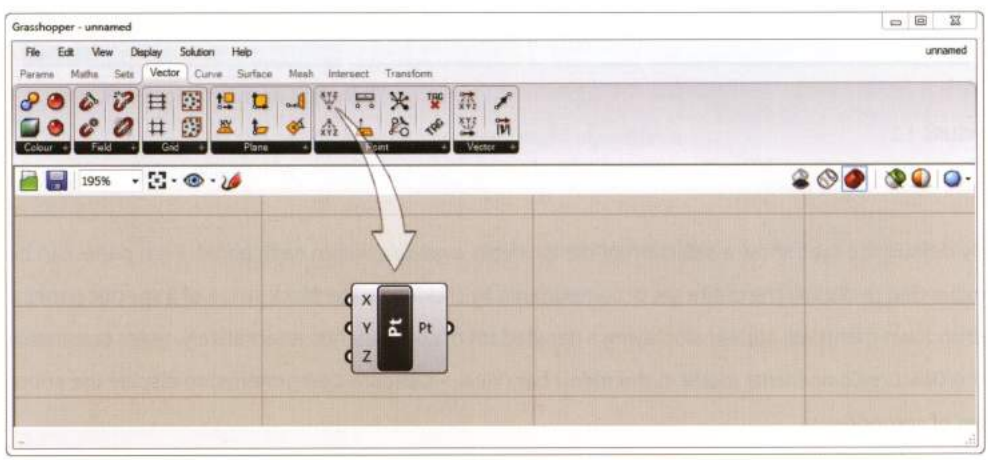
By default the tabs show a selection of components available within each panel. Each panel can be expanded to display the entire set of components by clicking on the black stripe of a specific panel: a dropdown menu will appear displaying a detailed list of components. Alternatively, users can enable the *Obscure Components* mode in the menu bar (View > Obscure Components) to display the entire set of components.



1.2.2 Working area: canvas

The canvas is the "blackboard" where users can build algorithms. Users can place a component on the canvas in one of two methods:

- By dragging the relative icon onto the canvas;
- By double-clicking any free area of the canvas and typing the component's name in the pop-up search box referred to as the **canvas search box**. As the first letters of the components name are typed in canvas search box, Grasshopper suggests a list of components that match the search criteria.



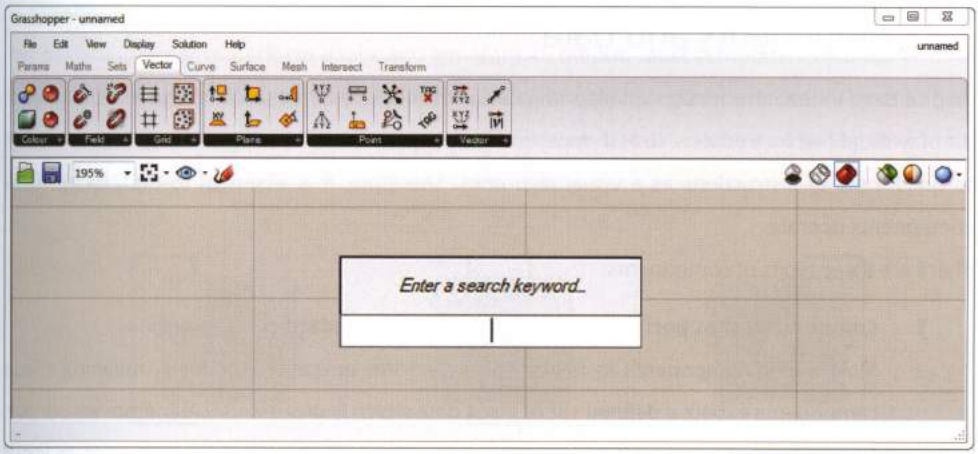


FIGURE 1.3

The image shows the *canvas search box* which allows you to find components by name.

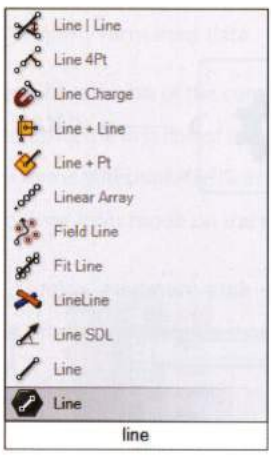


FIGURE 1.4

As you enter the first letters, Grasshopper suggests a list of components that match your search.

The canvas toolbar – positioned above the working area – provides the open/save files buttons as well as quick access to display and preview options.

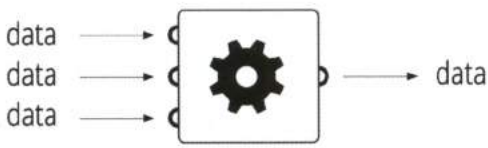
1.3 Components and data

As discussed in the introduction, an algorithm is a procedure that splits a complex task into a basic list of well defined instructions. Grasshopper provides users with a friendly interface of components to build a set of instructions as a visual algorithm. Therefore, it is essential to understand how components operate.

There are three types of components:

1. **Components that perform operations on data (standard components)**

   Most of the components in Grasshopper perform operations on data, meaning these components expect a defined set of input data which is processed to generate an output. For example, the point-component requires a set of numeric {x,y,z} coordinates as input to generate a point as an output, while a loft-component requires a defined set of curves as input to generate a surface as an output. The output of a component can be used as an input for another component.



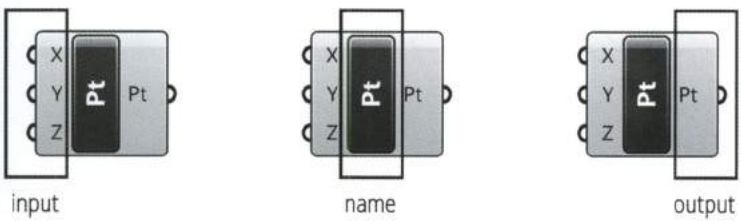
2. **Input-components**

   Input-components provide data (numbers, colours, etc) which can be modified by the user. They are hosted within the *Input* panel of the *Params* tab. Input-components do not expect input data.

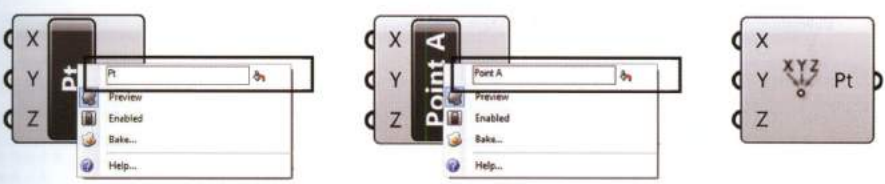
3. **Components that store data (container components)**

   These components can be imagined as containers for data. They can collect data in several ways, and can be used as input for other components. Container components are hosted within the *Geometry* and *Primitive* panels of the *Params* tab and they are characterized by a **black-hexagonal icon**.

A **standard component** placed on the canvas is represented by a node that requires a defined set of data in order to perform a task and generate an output. Most components consist of three main sections: ***input, name,*** and ***output***. The following image shows the component *Construct Point* (Vector > Point) which generates a point in the Rhinoceros 3D environment according to its three coordinates:



- The **input section** contains a variable number of input slots that are specific to each component. For example, the *Construct Point* component requires {x,y,z} coordinates as input. Each input requires properly formatted data.
- The **name section** shows an abbreviation of the components full name (e.g. Pt for Point). A component can be renamed using the first line of the context menu, which appears by right-clicking on the component's name. Alternatively, Grasshopper can display an icon instead of a name if users activate the *Draw Icons* mode on the menu bar (*Display* > *Draw Icons*).



- The **output section** shows a variable number of outputs that are specific to each component. For example, the *Construct Point* component has one output slot that generates data in the form of a correctly formatted point defined by {x,y,z} coordinates.

If the *Construct Point* component is placed onto the canvas a point appears immediately in the origin of Rhino's space. This happens because *Construct Point*, like many components, contains default data. In particular, *Construct Point* contains, by default, the following coordinates as input: x=0, y=0, z=0. **The objects (points, lines, surfaces, etc.) generated by Grasshopper are displayed by default as red geometries in Rhino.**

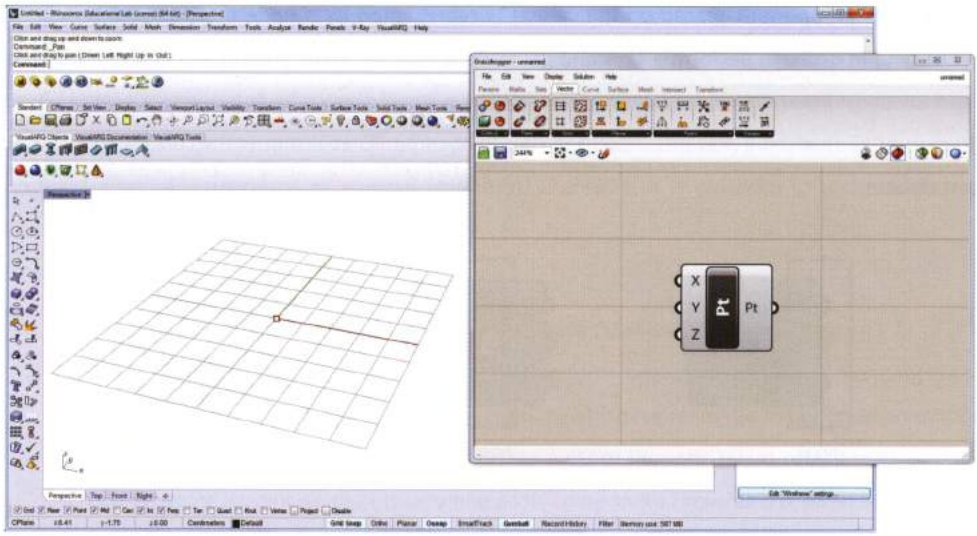


FIGURE 1.5
By default, the *Construct Point* component generates a point in the origin of axis.

To change the position of the point, users have to input values for the {x,y,z} coordinates. In other words, **data must be set inside the component**.

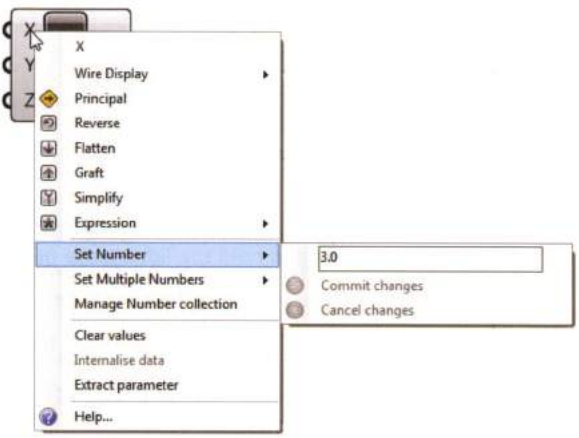
Grasshopper allows users to set data in three ways:

- **Local setting** (1.3.1);
- **Wired connection** (1.3.2);
- **Setting from Rhino** (1.3.4).

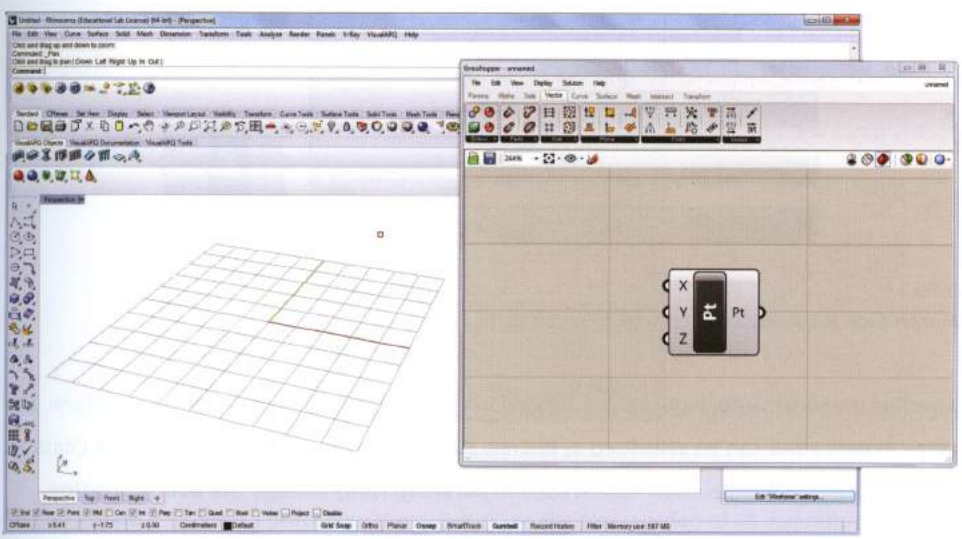
In general, an algorithm is constituted by components which use all three strategies. The next paragraphs discuss each method in detail.

1.3.1 Local setting of data

Data can be set **locally** by the *context pop-up menu* which appears by right-clicking on the specific input slot. For Example, to change the X input, right-click on the X and select the *Set Number* option. A new value (e.g. X=3.0) can be entered and confirmed by clicking on *Commit Changes*. As a result, the point moves three **Rhino units** in the x-direction, creating a point with coordinates {3.0, 0.0, 0.0}.



If we repeat the procedure for Y and Z by entering, for example, Y=3.0 and Z=2.0, the *Construct Point* component will generate a point with the coordinates {3.0, 3.0, 2.0}.



1.3.2 Wired connection

To build more complex algorithms **wired connections** are used to set data between components. Wires conduct data from the output of a component to the input of another component.
The *Construct Point* component can receive data from other components that output numeric data. For instance, the **input-component** *Number Slider* (Params > Input) generates a domain of numeric data. By adjusting the central grip, a number can be output within the domain and can be used as input for other components.

To transfer data to another component, press and hold the left mouse button on the output slot of the number slider. A connecting wire will be extracted which can be used to transfer data to the *Construct Point* component by dragging the end of the wire to the specified input slot. Once a connection is established, the *Number Slider* will be renamed based upon the input the wire is connected to.

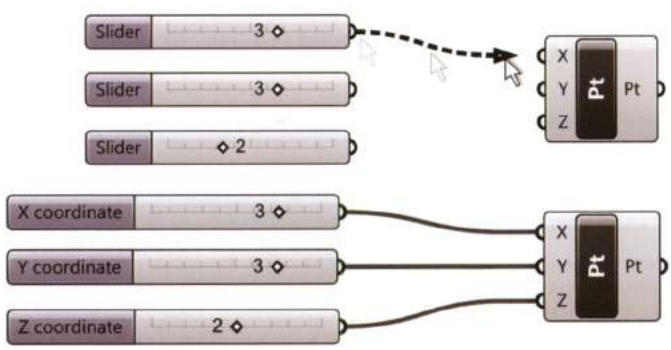
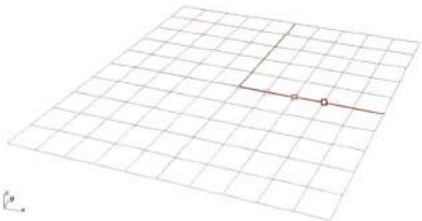
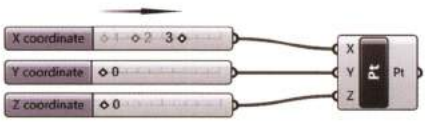


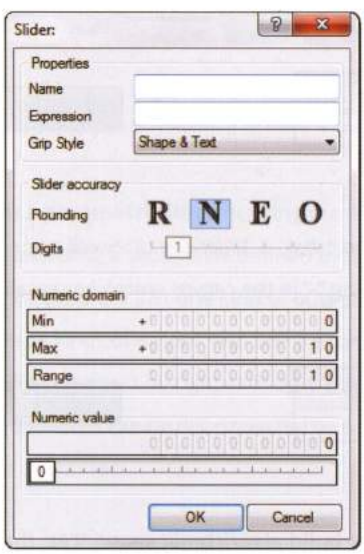
FIGURE 1.6
A *Number Slider* is automatically renamed according to the input name.

By moving the sliders grip, input values are updated changing the points position in real-time. The change in data input can be visualized in Rhino's 3D environment. Once connected to a *Construct Point* component, a *Number Slider* can be used to **parametrically change the points position**.





Slider properties can be changed through the context menu accessible by double-clicking the slider name (not on the grip). Within the context menu the *Numeric Domain* or the Min/Max value of the slider as well as numerical *Rounding*: *Floating Point Numbers* (R), *Integer Numbers* (N), *Even Numbers* (E) or *Odd Numbers* (O), can be specified. If the *Number Slider* rounding is set to R, the amount of displayed *Digits* (decimal places) can be set.



Setting a *Number Slider* is a procedure that slows down the construction of an algorithm: users have to change the *Numeric Domain* (min and max) and define the numerical set.

A quicker and easier way to insert a slider with a specific value is to double click on a free area of the canvas to recall the *canvas search box*. If a number (e.g. 5) is typed into the *search box* Grasshopper suggests the *Number Slider*. If the slider is added to the canvas the component will be set on the typed number (i.e. 5) with a defined default domain (i.e. 0-10).

The domain and a value for a slider can specified by typing 0<5<50 in the *canvas search box*. In this case, the domain will be set between 0 and 50 and the slider will be set to a value of 5.

If 0.5 is typed in the *canvas search box* a slider set to *Floating Point Numbers (R)* will appear on the canvas with one digit after the zero, a default domain between 0 and 1 and a value of 0.5.

If 0-5 is typed in the *canvas search box,* a *Slider* set to -5 will appear on the canvas with a default domain between 0 and -10. Typing "-" in the *canvas search box* recalls the component: *Subtraction*.

Wired connections are used to build algorithmic sequences, they transfer data between output and input of the components. For example, a line can be generated by connecting two points. In this case a specific component is used: *Line* (Curve > Primitive). If the sliders values are changed, the points {x,y,z} positions are recalculated and the magnitude and direction of the line updates associatively.

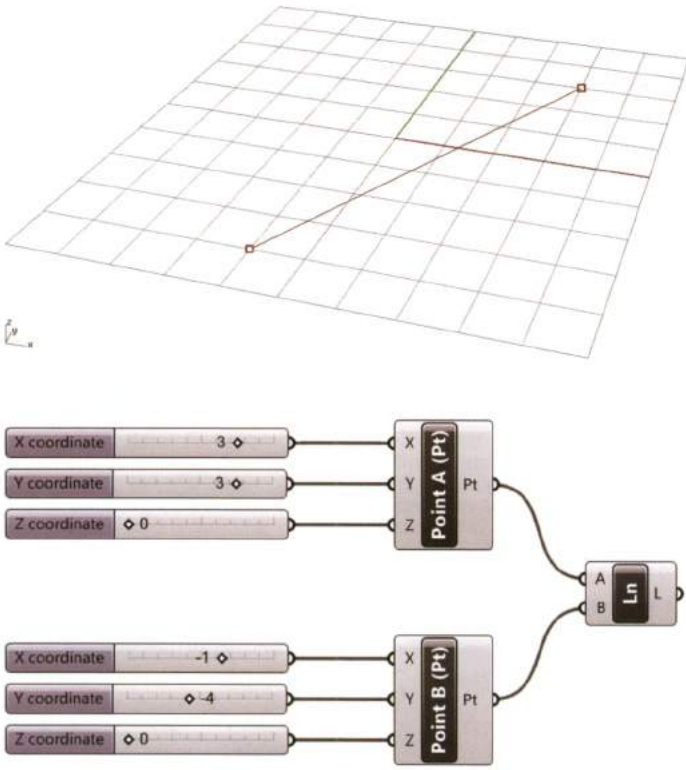
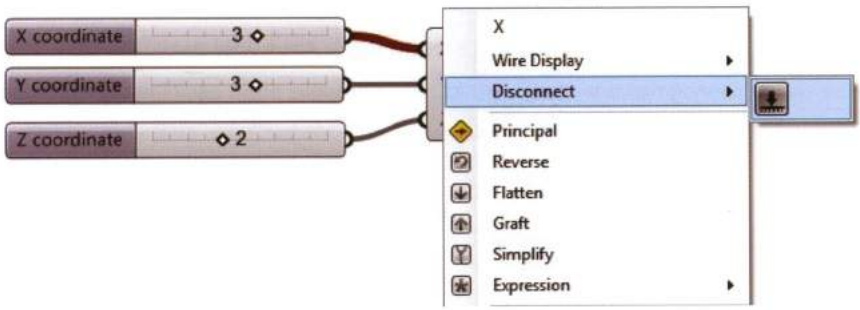


FIGURE 1.7
A line generated as a connection between two points (renamed as Point A and Point B).

The *Line*-output can be defined as a **parametric line**, meaning the line is tied associatively to the six slider parameters of the points, *generating* a set of lines defined by the possible combinations of the sliders domains. Algorithms do not generate just one kind of output.

The algorithm above generates three kinds of output: two point-geometries and one line-geometry.

Established connections can be disconnected by right-clicking on the specific input slot: the *context pop-up menu* will appear, and components can be disconnected by selecting the *disconnect* submenu and specifying the component to disconnect.



1.3.3 Warnings and errors

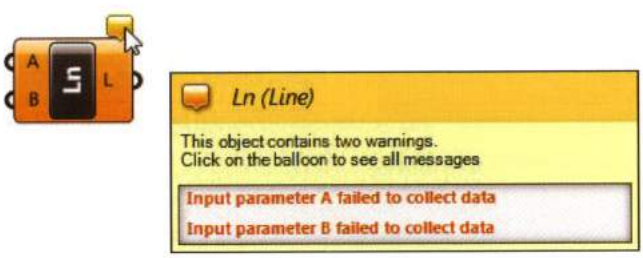
Algorithms can produce error messages and warnings. Grasshopper uses the background color of components to indicate their status and report possible errors. Components display status based on three different colors:

- **Correct status (gray)**

  A component which is properly connected displays a **gray** background. A working algorithm is composed of components displaying *correct status*.

- **Warning status (orange)**

  A warning status is indicated by an **orange** background. Usually, **a warning is related to a lack of data**. For example, when a *Line* component is placed on the canvas it is in warning status: displayed as orange. This is because the component requires two points (A and B), in order to create a line. If points A and B are connected to input A and B, the *Line* component turns gray and generates a line in Rhino.

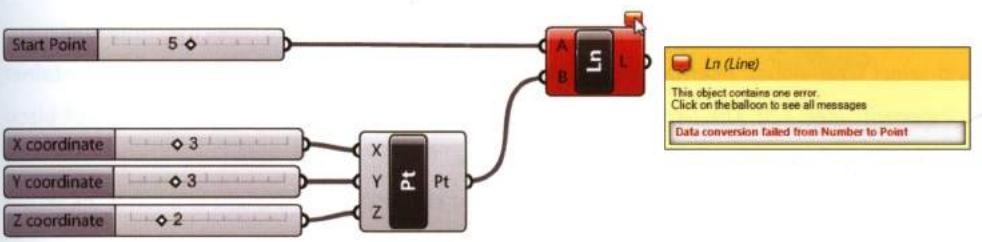


A component in warning status always displays a "balloon" (if it does not appear users can activate this function through *Display* > *Canvas Widgets* > *Message Balloons*). Hovering the mouse over the balloon activates a message-box which suggests the possible causes of the warning.

An algorithm with "orange components" may still work in some cases, but in general, a lack of data leads to unexpected or null results.

- **Error status (red)**

  An error status occurs if users do not fulfill the input requirements. In the comparison between algorithms and recipes it was noted that some steps require "ingredients", others "temperatures", still others require "times". Similarly, each component's input expects a defined type of data. For example, if users connect the A-input of the *Line* component using a numeric value instead of a described point {x,y,z}, the *Line* component turns **red** because input A and B must be formatted as a point.



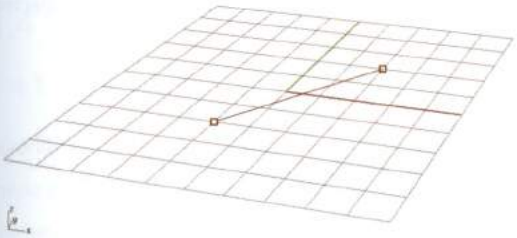
A component in error status does not generate any result. If we connect a component with a "red component" no data will be created or transferred between them. For this reason, a component in error status affects the entire set of components that are directly or indirectly connected with the component in error status. An error status can be fixed by correcting the input data.

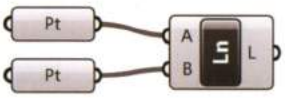
1.3.4 Setting from Rhino

Data can also be set from existing Rhino geometry. For example, a line can be created between two points that exist in Rhino. Using the *Line* component users can right-click on the A-input slot and select the *Set one Point* option from the contextual menu. The *Set one Point* option allows users to select a point from Rhino. When users click on *Set one Point*, Grasshopper's window minimizes, allowing the selection.

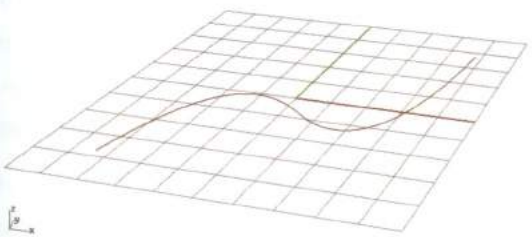
Once users set both A and B input points, the component turns gray and generates a line. Such a method directly stores a Rhino's objects inside an input. If the points move in Rhino the line will be associatively updated. To "empty" an input, right-click on the input slot to recall the contextual menu and select *Clear values*. Alternatively, Rhino's geometry can be collected using specific **container** components (see 1.3). The *Geometry* and *Primitive* panels of the *Params* tab hold a set of black-hexagonal icons for **components that store data.** To set a point, select the component *Point* (Params > Geometry) and place it on the canvas, right-click on the component and select the *Set one Point* option within the context menu.

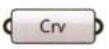
For example, to create a line between two Rhino points: collect the points data with two *Point* components and connect them with wires to a *Line* component.



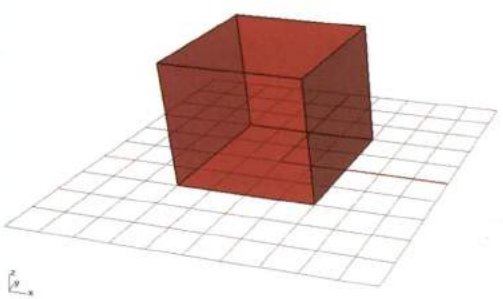


The container components do not perform any operations, they just store data. These components can be used to collect any type of geometry or object from Rhino. Each container component expects a defined type of geometry. In order to collect curves, the *Curve* component would be selected. Correspondingly, a *Point* component is unable to collect curves.





The *Geometry* container is a versatile component which can be used to store any kind of geometry (points, curves, solids etc.). To set a geometry or multiple geometries right-click on the *Geometry* component and select the *Set one Geometry* or *Set Multiple Geometries* option within the context menu, respectively.



The container components also provide an input slot that allows components to store data collected from other components.

Grasshopper's geometry, displayed in red, overlaps the imported geometries that exist in Rhino. When geometry is "set", Grasshopper establishes a continuous link with Rhino, meaning if geometry is deleted in Rhino, the specific container component turns orange because it is no longer collecting data.

Container components allow data to be permanently stored by right-clicking on the geometry component and selecting the *Internalise data* option from the context menu. If this option is selected, Grasshopper will import the geometries data, breaking the link with Rhino, meaning if the original geometry is deleted the specific component will continue to store the data. For example, if a point is imported from Rhino using the *Point* component and internalised, the point can be deleted in Rhino and still exist as stored data in Grasshopper. The points {x,y,z} location can be changed by selecting the *Point* component and using Rhino's *Gumball* to change the point's position.

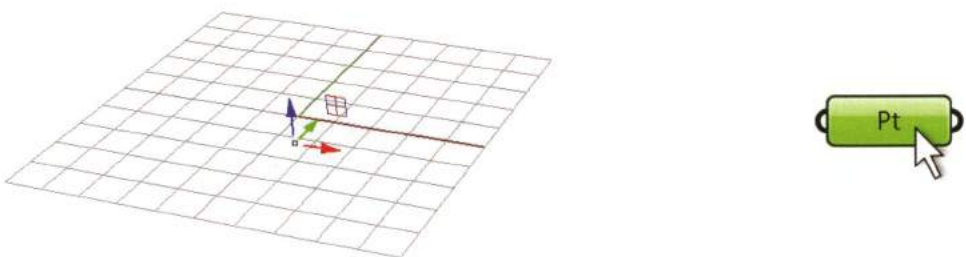


FIGURE 1.8
If you *internalise* a point in Grasshopper and then you click on the relative box-component, the Rhino's Gumball appears on the point, so you can easily change its position.

1.4 Save and bake

1.4.1 Save

Algorithms can be saved using the *Save Document* or *Save Document As* options available in the *File* menu of the *Menu Bar*. The default extension for Grasshopper's files is **.gh**.
The .gh files are not executable. Therefore, to open a previously saved algorithm, users have to start Rhinoceros, load Grasshopper and open the desired file.

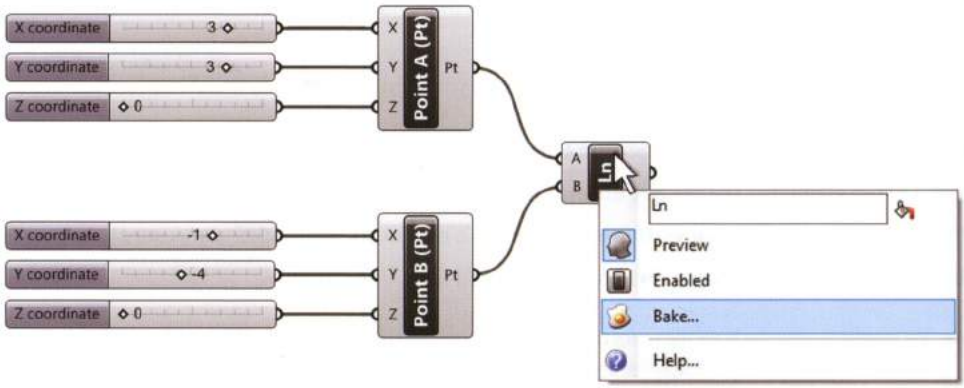
grasshopper file gh.gh

grasshopper file ghx.ghx

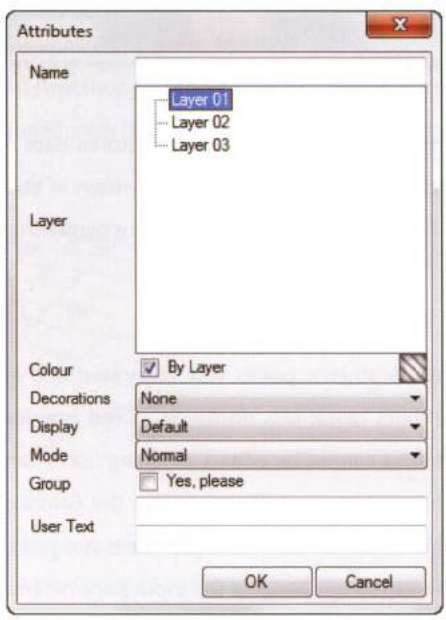
The .gh file extension is formatted in binary, meaning it stores data as pure bytes. Grasshopper files can also be saved as **.ghx**. The .ghx file extension is written in XML format, meaning it can be modified using text-editors. For this reason, the .ghx files are bigger than .gh files.

1.4.2 Bake

In the previous sections, two parametric points was generated and associated to a line creating a basic algorithm. The algorithm's result was displayed as **red preview geometries** in the Rhino environment. Preview geometries cannot be edited, meaning users cannot select them, save them as a Rhino file, render them, etc. Moreover, if users close the Grasshopper file these objects will disappear. The reason for this is that Grasshopper generates not just **one** geometry but an entire set of geometries that can be varied by changing the input parameters. To edit one of the possible configurations generated by an algorithm, users have to "**bake**" the Grasshopper preview into the Rhino environment. Every component that generates a preview can be baked by right-clicking on the component and selecting the *bake* option from the context menu. For example to bake the *Line* component, right-click on it and select *Bake...*. Baking the component makes the line editable in Rhino.



The *Bake Attribute* window, which opens when *Bake...* is selected allows users to set several attributes, such as: the **target layer**, the color of the baked geometry, and the possibility to group objects.



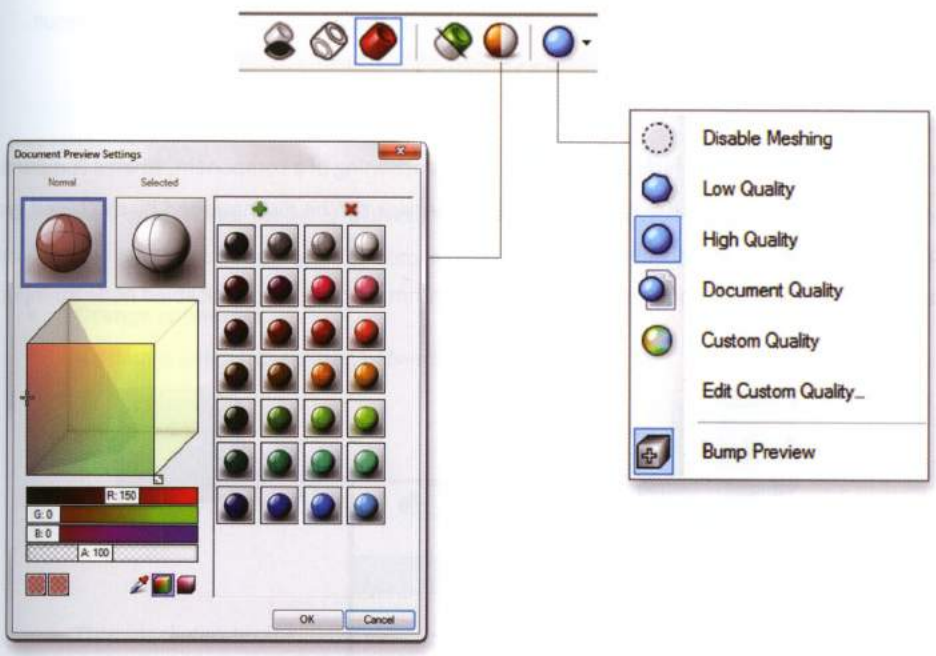
After baking, the geometry can be changed by manipulating the input parameters within Grasshopper. Grasshopper geometry is parametric and the number of solutions are only limited by the input parameters combinations. To undo a bake operation, simply delete the geometry in Rhino or type *undo* in the Rhino command line. Grasshopper cannot bake geometry if any command is running on Rhino.

1.5 Display and control

Grasshopper uses colors and graphics to report component states and display-modes.

1.5.1 Preview quality. Enable/disable preview

Working components generate a result that are, by default, represented by a red preview color in Rhino. Color and quality of the preview can be set through the *Canvas Toolbar*. In particular, the preview color can be set by the *Document Preview Setting* option, while the quality of visualization can be modified through the *Preview Mesh Quality option*. Both options are contained in the canvas toolbar.



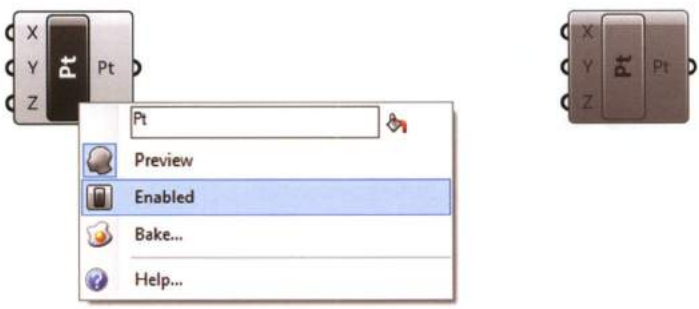
The preview of a geometry can be disabled by right-clicking on the name of the component and selecting the *preview* option from the context menu. When you disable the preview the component turns **dark grey**.

Enabling/disabling preview is crucial when algorithms become complex. Algorithms can be imagined as a construction history, it is unnecessary to visualize all the steps that lead to the final result.

1.5.2 Enable / disable objects

Unlike the *Preview* option, which does not affect the functioning of a component, when a component is disable it will no longer operate. A disabled component can be distinguished by the **faded gray** color of the background and name section.

Disabling a component disables all parts of the algorithm that rely on the disabled node for input.



Many productivity tools so far discussed are included in the ***Radial Menu*** which can be opened by clicking the spacebar, including: navigation, preferences, group, cluster, preview/hide, enable/disable, bake, zoom, disable solver, recompute, and find. For example, to disable the preview of two or more components: select the components, press the spacebar and click on the "masked face" to disable the preview. Other tools of the *Radial Menu* follow a similar logic.

1.5.3 Wire display

If the ***Draw Fancy Wires*** mode is activated (*Display menu* > *Draw Fancy Wires*), Grasshopper differentiates types of connecting wires according to their type of data structure. In particular:

- **Orange connector**

  No data is being transmitted between components.

- **Thin black connector**

  Just one datum (one number, one geometry, etc) is being transmitted between components.

- **Wide black connector**

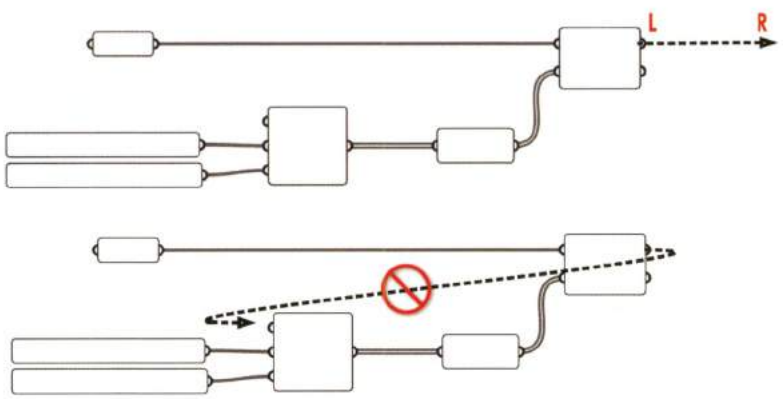
  Two or more data items are being transmitted between components.

A fourth type, the dashed wire will be discuss later in chapter 5.

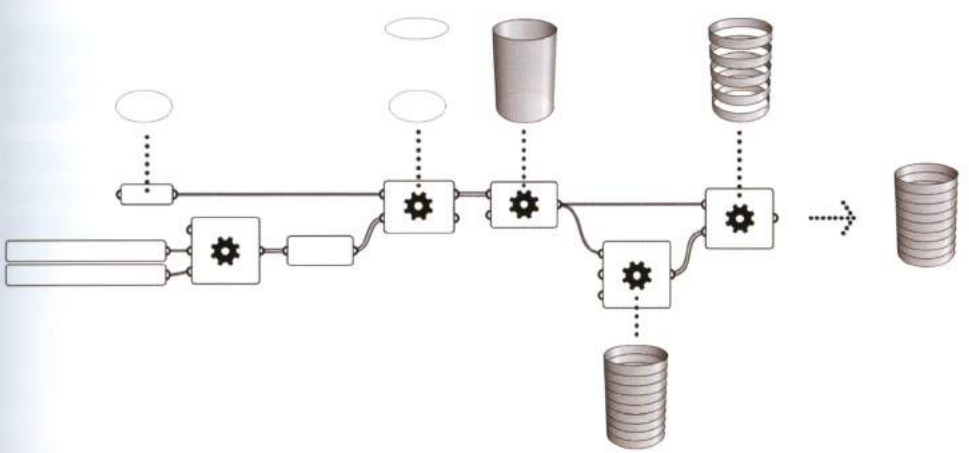
Grasshopper allows users to set "wireless" connections, making a wire invisible unless one of the connected components is selected. This option can be activated within the context menu of every input slot by selecting *Wire Display* > *Hidden*.

1.6 Grasshopper flow

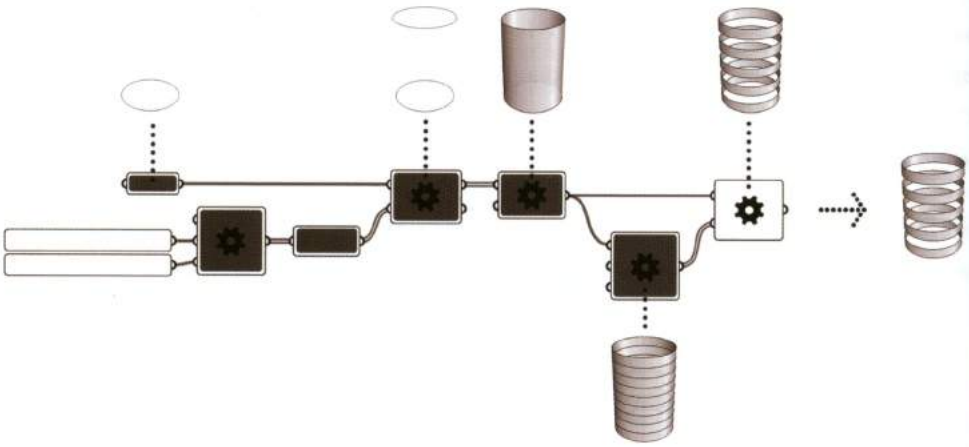
It is important to point out that wires can only connect the output of a generic component (A) with the input of a component (B) that doesn't precede (A) in the algorithmic sequence. In other words, **the data stream can be imagined as a fluid that flows through the components from left to right**. Consequently, is not possible to create a **loop** in Grasshopper, except using special components which can be discussed in chapter 7.



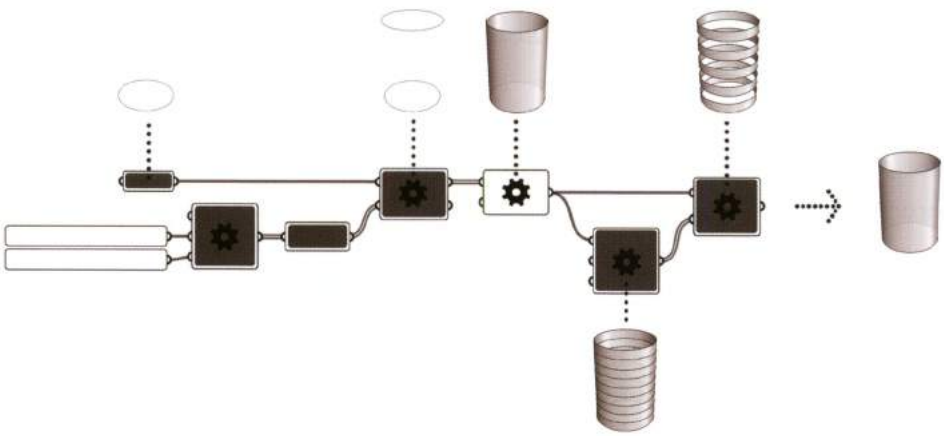
The Grasshopper logic sequence can be imagined as a ***construction history***: Grasshopper stores the output of each instruction. To display the final output of an algorithm all nodes except last component must be un-previewed. If users do not hide components the algorithm will generate geometry in which the final step overlap the previous steps.



In the following image all the components are turned off except the last component, allowing the final output of the algorithm to be visualized without overlap the previous steps.



In the following image all the components are turned off except an intermediate step.



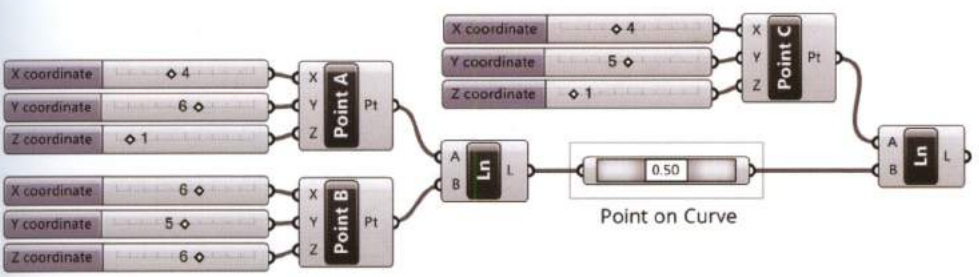
1.7 Basic concepts and operations

This section is a preview of topics that will be discussed, in depth, in following chapters. Here we will focus on components that find specific points on objects or perform basic operations such as moving objects, as well as visualization components.

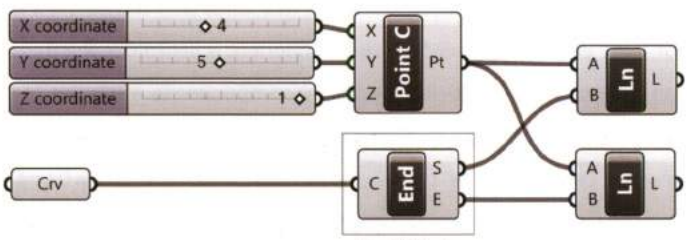
1.7.1 Object snap in Grasshopper

The drawing accuracy of CAD software relies on *Object Snaps* or *Osnap controls*. The *Object Snap* tools allows users to select strategic locations to connect objects. For example, a line has three main points that can be snapped to: a start point, a mid point, and an end point. Grasshopper does not function under the CAD snap logic, instead, it relies on defined algorithmic logic to create the same functionality.

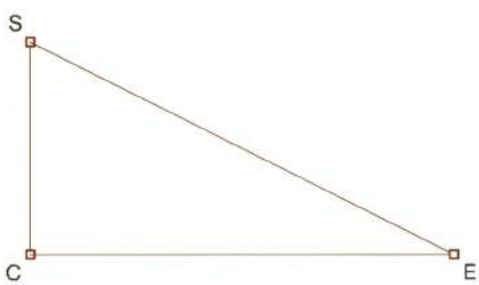
Specific points on a curve can be found using the component *Point on Curve* (Curve > Analysis). The *Point on Curve* slider defines a point on a curve by "reparameterizing" a curve's length as a unit of 1. Correspondingly, the *Point on Curve* component set to a value of 0 will define a point at the start point and value of 1 the end point. By default the component is set to 0.5, the mid point. A specific numeric value can be chosen by right-clicking on the component to open the context menu and selecting the desired input value *(start, quarter, third, mid, two thirds, three quarters, end)*. Arbitrary points along a curve can be "snapped to" by adjusting the slider's grip between 0 and 1. The following example shows how to create a line which connects a point C to the midpoint of a line between two points A and B.



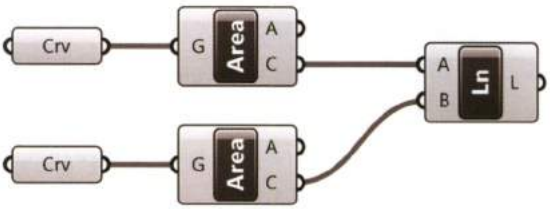
The component *End Points* (Curve > Analysis) can be used to quickly exctract the start point (S-output) and end point (E-output) of a given curve. The following example demonstrates how the *End Points* component can be used to access the end points of a curve set in Rhino (the curve could alternatively be described in Grasshopper), then use the *End Points* component output to generate two lines between the start and end points and a third defined point.

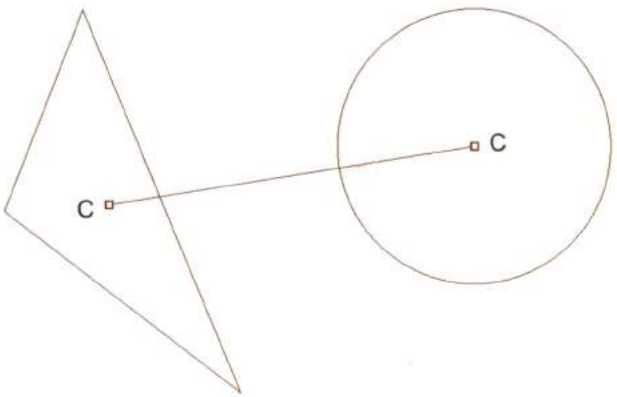


End Points

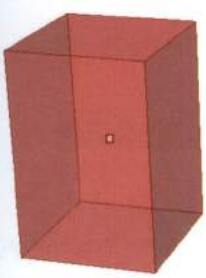


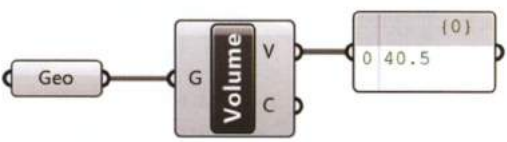
The component *Area* (Surface > Analysis) can be used to extract the geometric *centroid* of a planar closed curve. The component also calculates the area (A-output) of the input curve. The following example demonstrates how the *Area* component can be used to access the centroids of two closed curves, then connect the points to create a line.





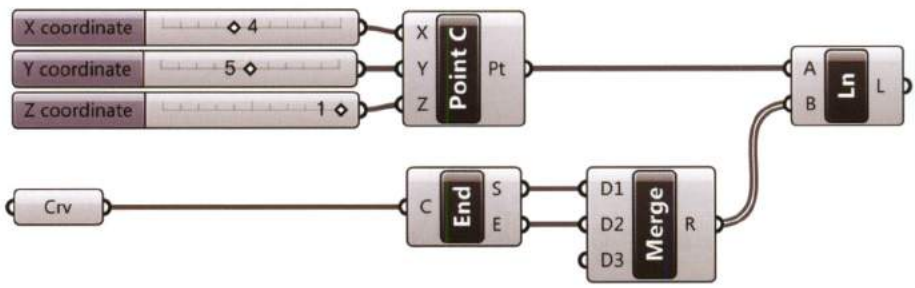
The component *Volume* (surface > Analysis) can be used to extract the geometric *centroid* of a closed three-dimensional geometry. The *Volume* component output calculates volume (V-output) and centroid (C-output). Data output, such as area and volume for the *Volume* component, can be displayed using the component *Panel* (Params > Input). The following example calculates the volume of a Rhino set geometry and displays the result as a single output in a panel. If multiple geometries were set, the panels would calculate the volume of each object, and display the results as specific items in the panel.





1.7.2 Merging data

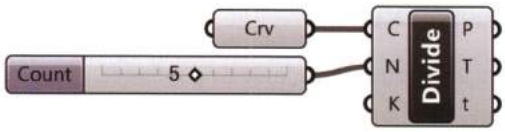
The component *Merge* (Sets > Tree) allows two or more data flows to converge in a single list. With reference to the previous exercise, the following example shows how to use just one *Line* component, instead of two, by collecting the start point and end point by the ***Merge*** component.

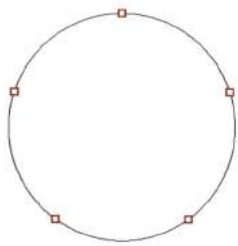


Wires containing more than one item are displayed as a thickened line. For instance, after the merging operation the wire contains more than one item and is displayed as a thicker line (see 1.5.3).
When the *Merge* component is placed on the canvas, it displays two inputs (D1 and D2). Once the D2-input is receiving data (i.e. has a wired connected), *Merge* automatically adds a D3-input. The *Merge* component maintains a record of the connection-order via the progressive index of the D-input.
Additional D-inputs can be manually added by using the ***Zoomable User Interface***. The interface (available on some components) adds an input by clicking on the "+" buttons which are visible after zooming-in on the specific component.

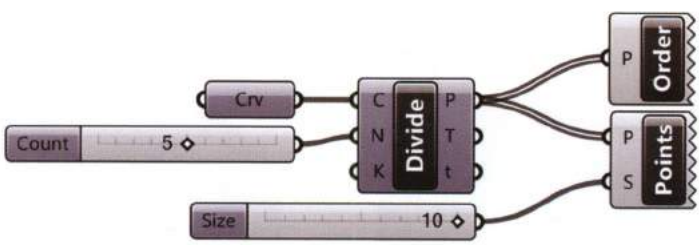
1.7.3 Dividing a Curve

The component *Divide Curve* (Curve > Division), divides an open or closed curve into equal arc length segments. More specifically, given a curve set from Rhino (C-input) and an integer for the number of subdivision (N-input), N+1 points (P-output) will be generated in case of an open curve and N points in case of a closed curve. The points will be stored in a single list.





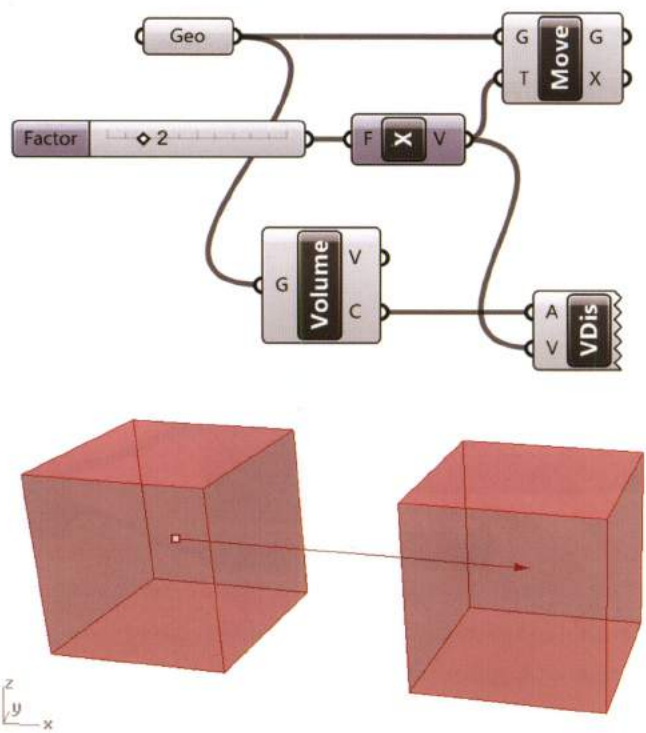
The index number of each point and the points' order can be displayed using the component *Point List* (Display > Vector). Display components do not have an output section as they are used for visualization. The component *Point List* allows users to set the text-size by the S-input.



1.7.4 Moving an object / vectors

When manipulating objects using a traditional modeling software, many mathematical/geometrical operations are blindly performed. For example, when vertically moving an object, the movement is performed according to a specific vector. **Vectors** are a geometric entities defined by **direction, sense and length (or magnitude)**.

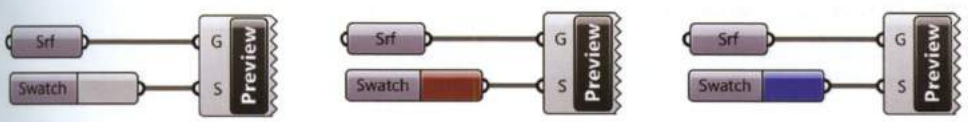
The component *Move* (Transform > Euclidian), moves geometries according to a specified vector. Vectors can be defined by several components (Vector > Vector). For instance, vectors can be defined by three component parts using the component *Vector XYZ* (Vector > Vector) or by specifying a default unit vector according to an axis: *Unit X*, *Unit Y* or *Unit Z* (*Vector > Vector*). The following example uses the *Move* component to translate an input geometry (G) along a vector (T). The vector is defined through a *Unit X* component which, by default, is a unit vector whose direction and sense is parallel to x-axis and length equal to 1. In order to amplify the vector (and consequently, the movement) it is enough to connect a *Number Slider* to the F-input of *Unit X*. According to the *construction history* (see 1.6) Grasshopper displays the original and the translated geometry. In order to display just the moved geometry, the *Geometry* component must be turned off.

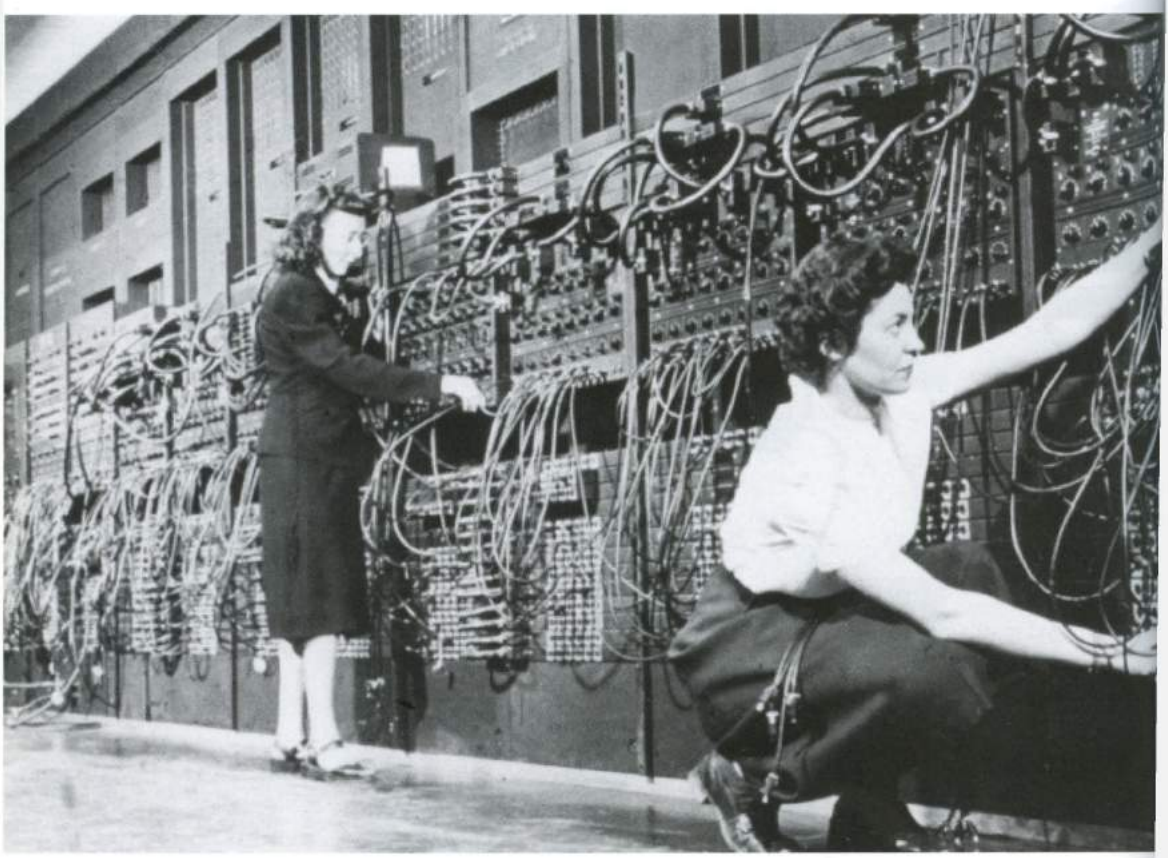


By default vectors are not previewed, they can be displayed using a specific visualization component: *Vector Display* (Display > Vector). *Vector Display* requires a start point (A) and the vector to display (V).

1.7.5 Custom preview

The default red-preview of Grasshopper geometries can be modified through the *Document Preview Settings*. **Users can alternatively customize the preview of single components using Custom Preview (Display > Preview).** *Custom Preview* relies on the component *Colour Swatch* (Params > Input). The color and transparency level can be set by left-clicking on the node to recall a context menu.





The world's first electronic digital computer, the Electronic Numerical Integrator and Calculator (ENIAC),1946.

2_data
how to manage data in Grasshopper

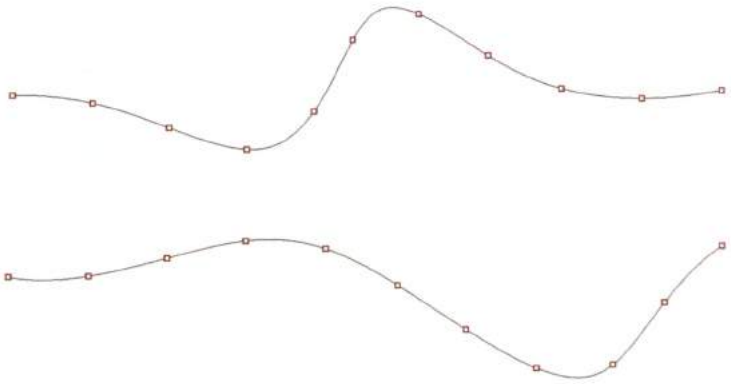
> "Our ability to do great things with data will make a real difference in every aspect of our lives".
>
> Jennifer Pahlka

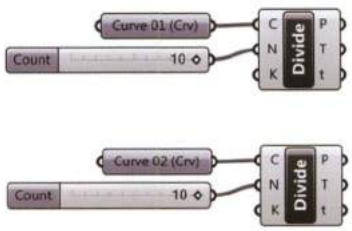
Digital modeling has traditionally been based on virtual manipulation, by the pseudo physical presence of the mouse and keyboard in the digital environment. Algorithmic modeling, conversely, is based on logic and is void of "physical" manipulation. Algorithms rely on the ability to establish conceptual associations between geometry and mathematics. In other words, **data is manipulated instead of digital objects**.

An algorithm can be imagined as a network of data streams, and in order to selectively operate on the network, is necessary to filter, divert and modify data to virtually manipulate geometry. Within this context mathematics and logic are the mouse and keyboard.

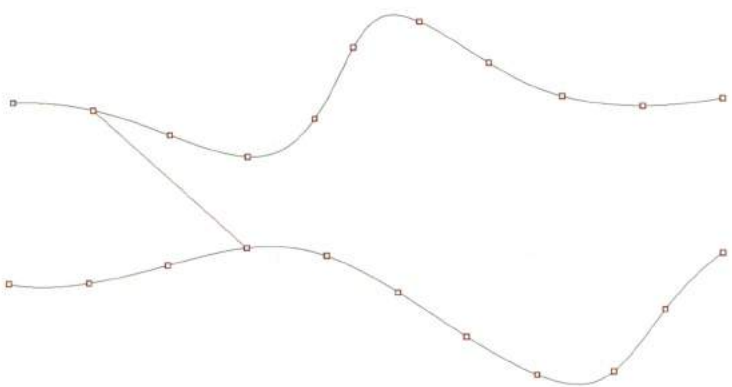
2.1 Filters

One of the consequence of the lack of "physical" interaction in Grasshopper is the seeming difficulty in selecting single items among a set of objects. In the following example two curves drawn in Rhino are imported into Grasshopper (by two *Curve* components renamed as Curve 01 and Curve 02) and then divided into ten parts. As result, eleven points are generated on each curve.

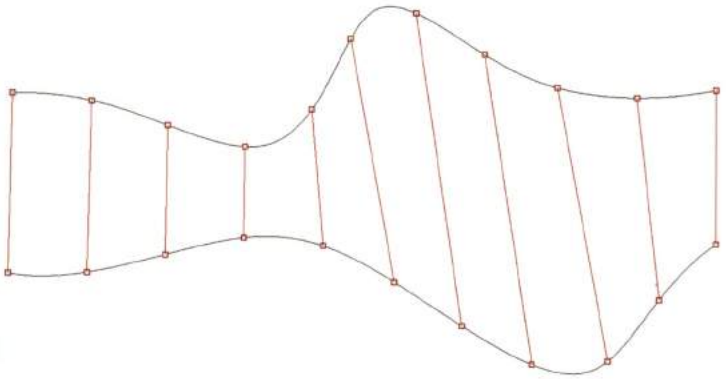


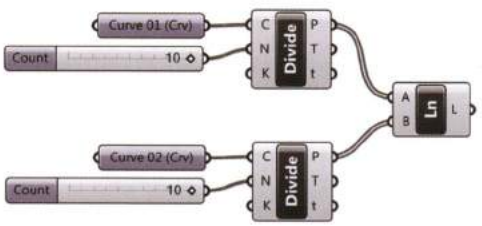


A simple operation such as connecting two arbitrary points among the set of 22 division points (see following image), seems impossible to get using the *Line* component, since we don't know yet the techniques to select single items.

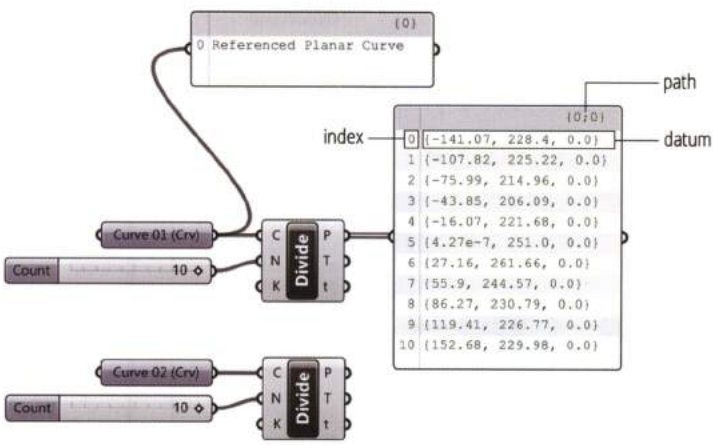


In fact, connecting a *line* component to the P-output of the two *Divide Curve* components will instruct Grasshopper to generate eleven lines connecting the entire set of corresponding points.



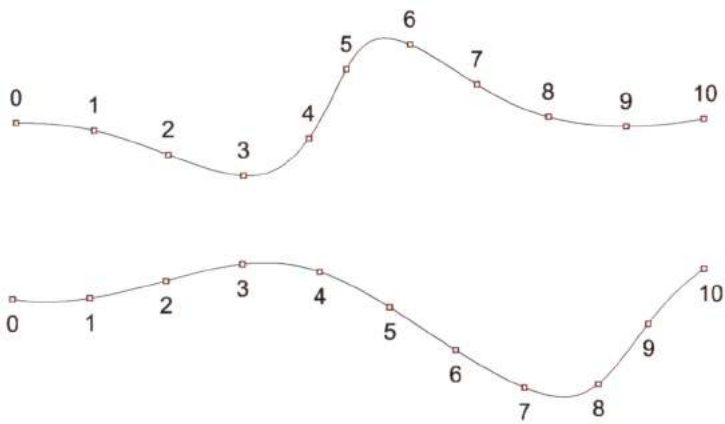


In order to analyze the output of a component a ***Panel*** (Params > Input) can be connected to a component's output to display the data structure. Panels are visualization tools that can be used to visualize output data.



Grasshopper structures data into paths i.e. hierarchical levels that contain **Lists**. Lists are further subdivided into items which are indexed with a natural number (0 to i) to reference an item's position

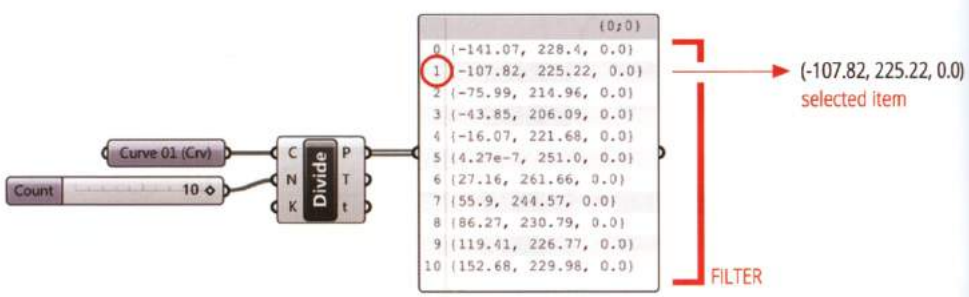
within the list. For example, the data output of the *Divide Curve* components slot (P), is a triplet of coordinates mathematically representing an {x,y,z} point. Each point or datum is referenced by an index, the first point in the list (-141.07, 228.4, 0.0) has an index of 0. Indices always start from 0 and count in integers to i. Paths will be discussed later (chapter 5).



Grasshopper sorts data based on the way data is set. For example, objects set from Rhino will be organized in Grasshopper in the same order they are selected from Rhino. In the case of *Divide Curve*, points are sorted according to the curve direction.

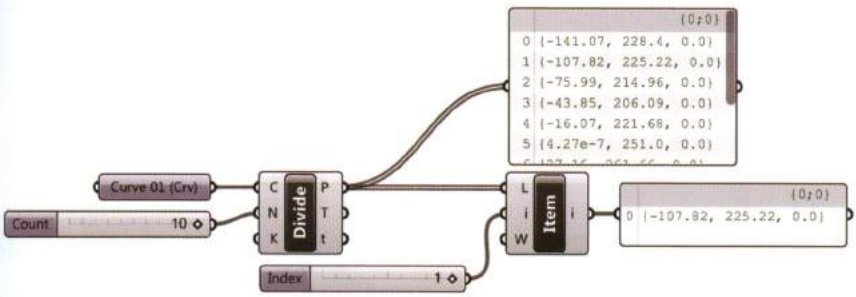
2.1.1 List Item: select one item from a list

Understanding lists enables users to easily select a specific datum within a list by referring to its index by a **filter**. Grasshopper has many components to filter and direct data.

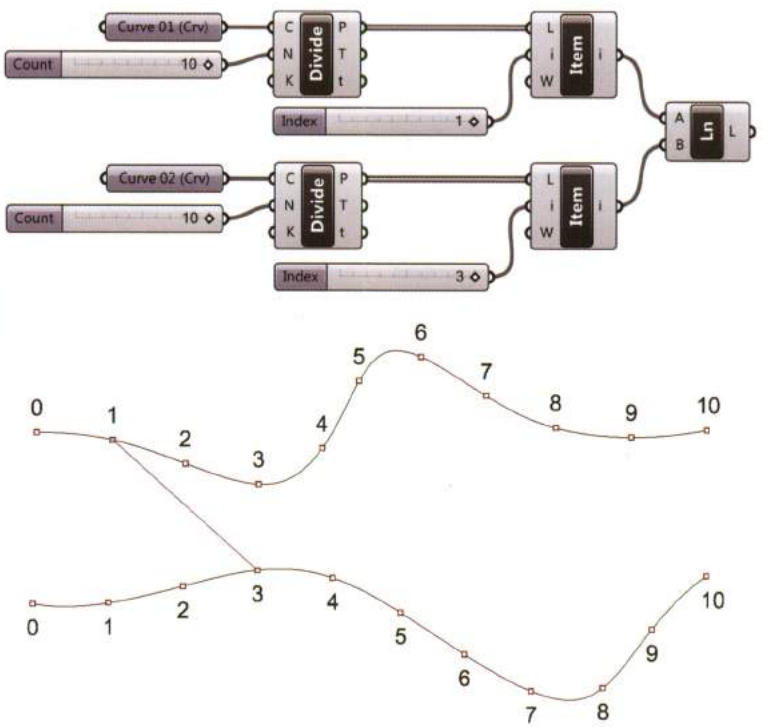


For example, the component ***List Item*** (Sets > List) is a filter to select a specific index item. To select

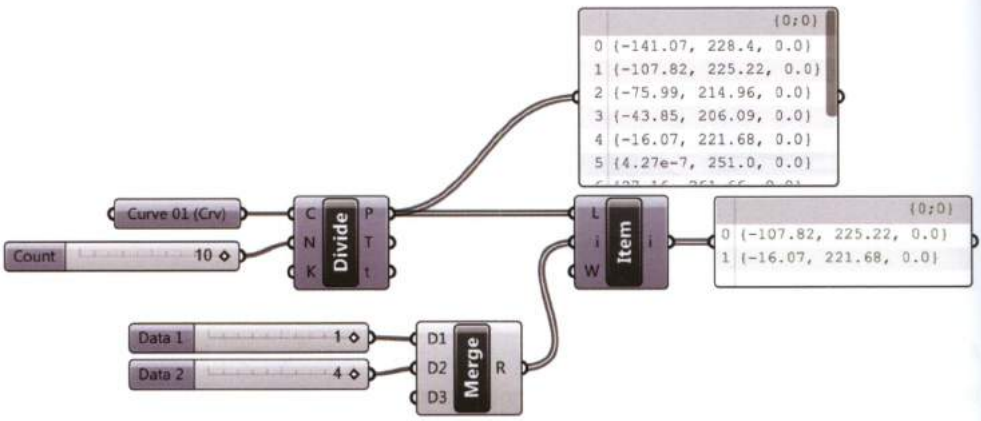
an item, connect a list to select from into the L-input of *List Item* and an integer to the item slot (i). The integer's value specifies the item that the filter selects.



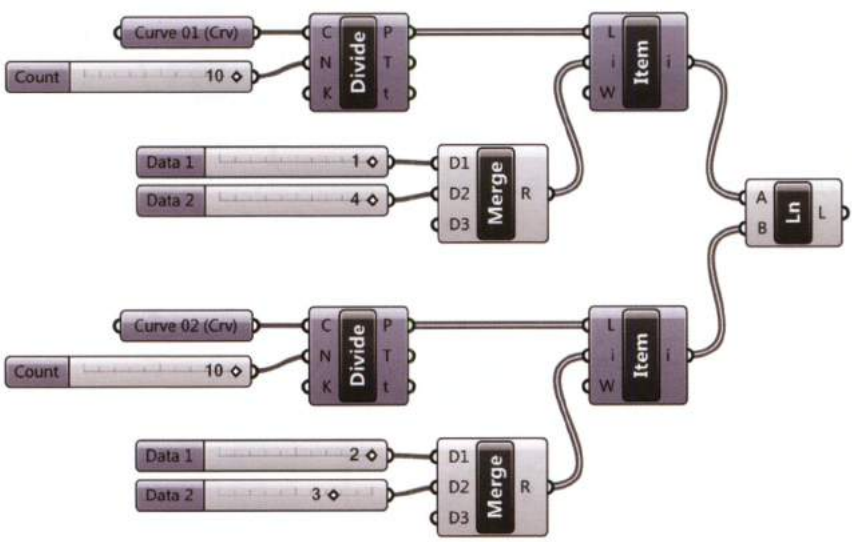
A Line can be generated between the *List Item* outputs (i) by connecting wires from the i-outputs to the A and B inputs of *Line*. The "Index" slider denotes the point selected from the list to be connected as endpoints of the line.

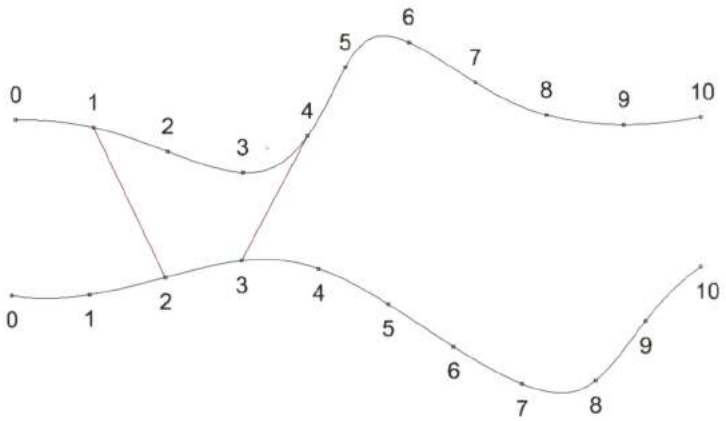


In order to select two or more items, several sliders can be connected to the i-input of the component *List Item*, by using the ***Merge*** component, or by pressing and holding the *shift* button on the keyboard while connecting multiple wires.



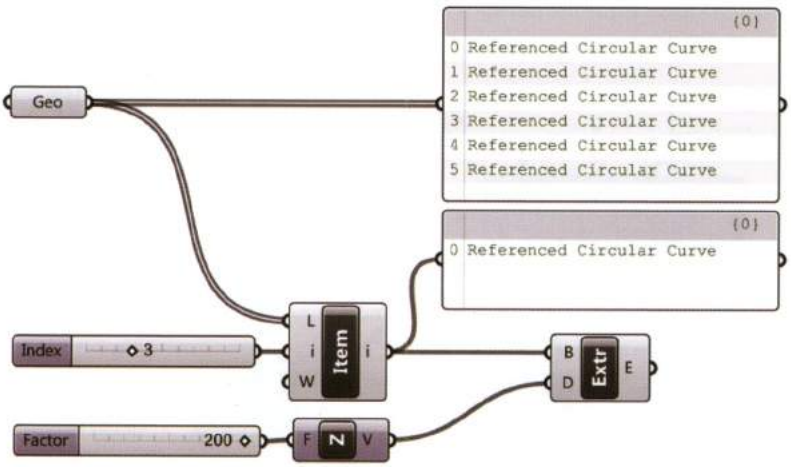
In this way, two lines can be generated by specifying two index items from the P-output using the *List Item* component.





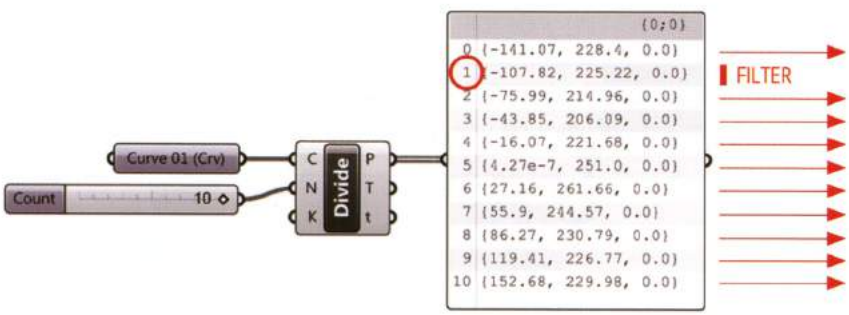
The component *List Item* can be used to filter any data formatted in a list. For instance, the *List Item* component can be used to select a specific circle within a list of circles.

For example, using a *Geometry* container six circles are set from Rhino, using the *Set Multiple Geometries* option from the context menu. Grasshopper sorts the circles according to their position on the x-axis. In order to select and extrude the fourth circle, the index item 3 needs to be specified as the input of the *List Item.* The component *Extrude* (Surface > Freeform) is used to extrude the circle according to a distance and direction (D) specified by a vector. In this case the component *Unit Z* (Vector > Vector) specifies a vector in the z-direction with a magnitude of 1. Using scalar multiplication a specified magnitude is set using a number slider (factor).

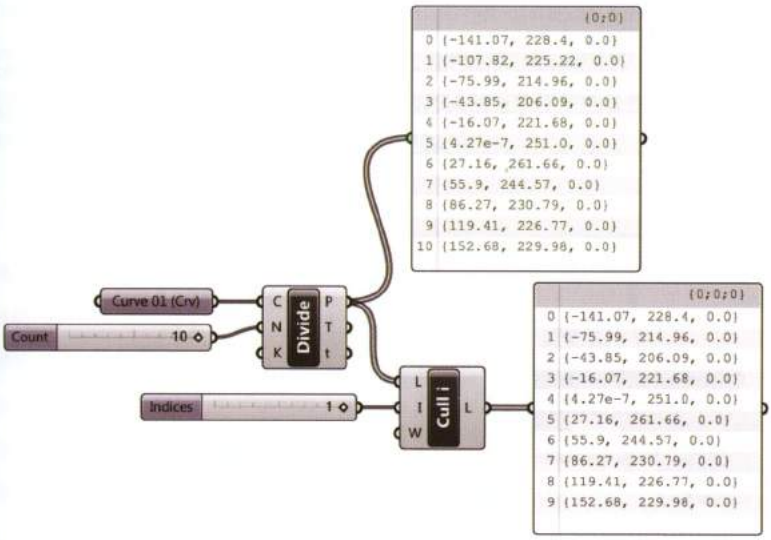


2.1.2 Cull Index: select all data except one item

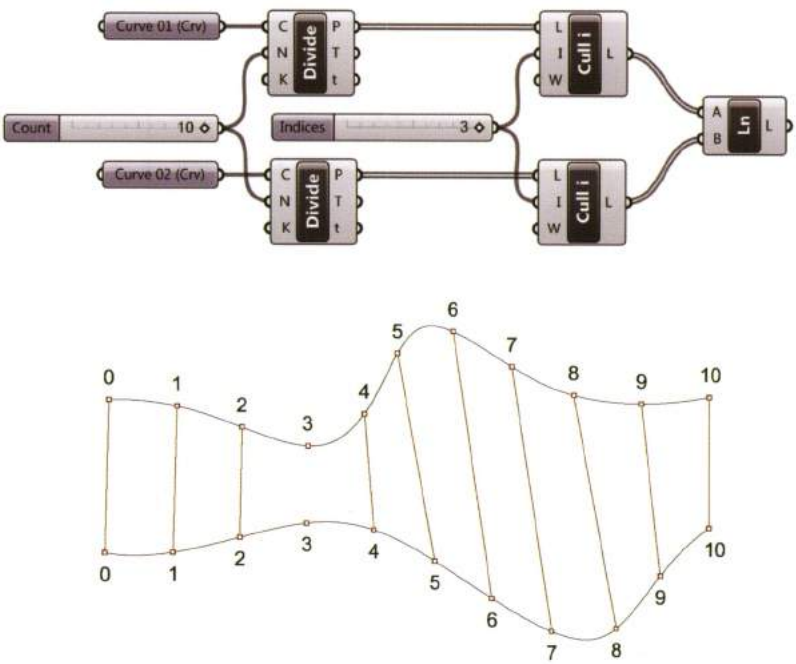
The component ***Cull Index*** (Sets > Sequence) performs the reverse fuction of the *List Item* component: *Cull Index* deletes specified index items from a list.



For instance, to cull index 1 from the list, a slider set to 1 is connected to the i-input of the *Cull Index* component (see following image). As a result, if the initial list has N items the output of *Cull Index* is a new list which hosts N minus the number of culled items.

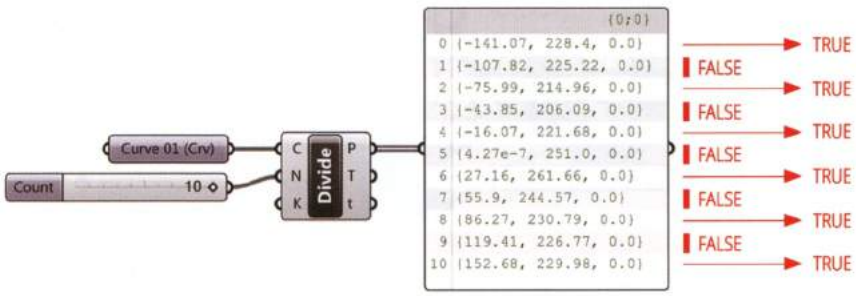


In the following example a set of lines is created between couples of corresponding points excluding the points which are identified by the index 3.

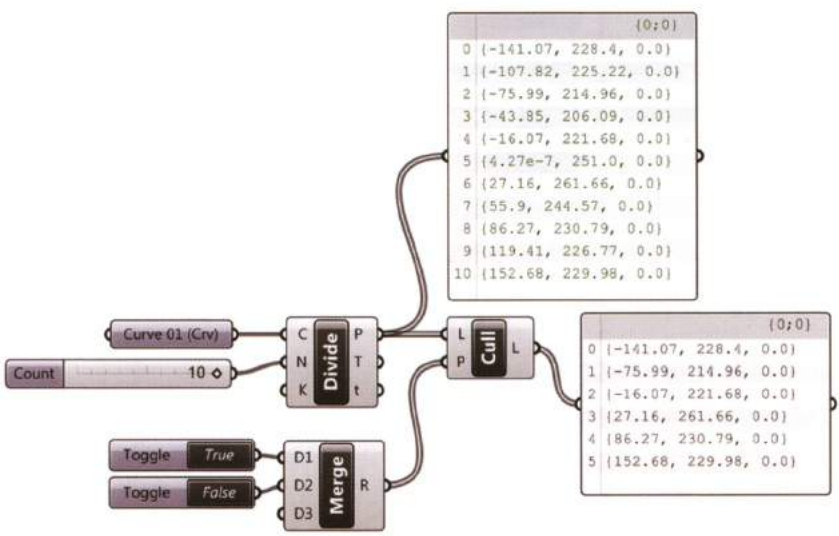


2.1.3 Cull Pattern: select items using a repeating model

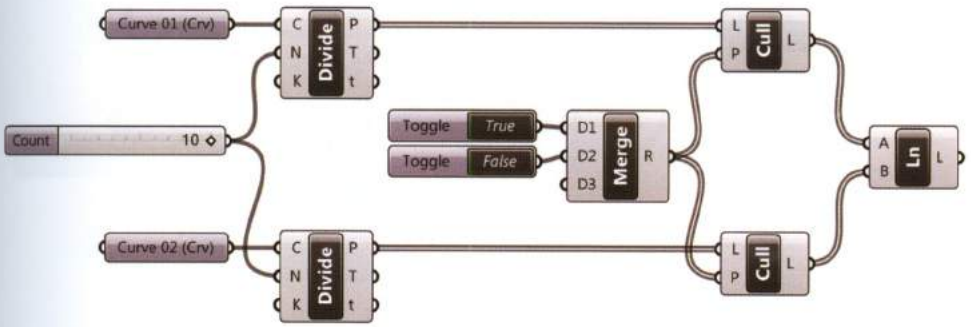
Repeating patterns of True and False values (***Boolean*** values) can be used to select and exclude, respectively, items from a list. The component ***Cull Pattern*** (Sets > Sequence) is used in Grasshopper to filter lists according to a repeating pattern of Boolean values.

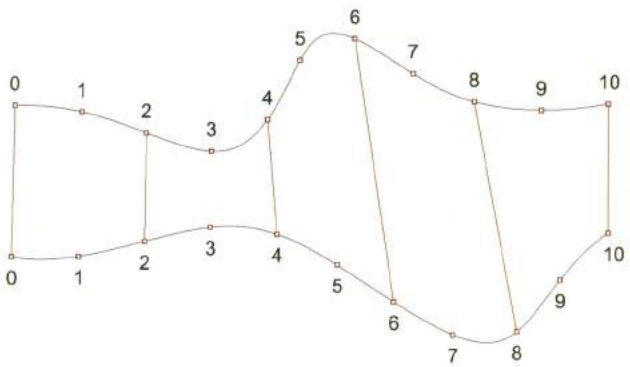


The component ***Boolean Toggle*** (Params > Input), can be used to define culling patterns. The component output can be changed by left clicking the True/False portion of the button, to declare either True or False respectively. Multiple *Boolean Toggle* components can be combined into a single list using the *Merge* component.



The declared True/False pattern is repeated for the length of the list, meaning if the pattern is (True, False) and the list has a length of 4 (contains index numbers 0,1,2,3) the pattern will repeat resulting in a culling pattern (True, False, True, False). Referring to the previous example, lines can be drawn between alternate couples of corresponding points.





Repeating models can be based on patterns greater than two declarations. As shown in the following image the *Cull Pattern* is based on a repetition of three Boolean values (True, True, False).

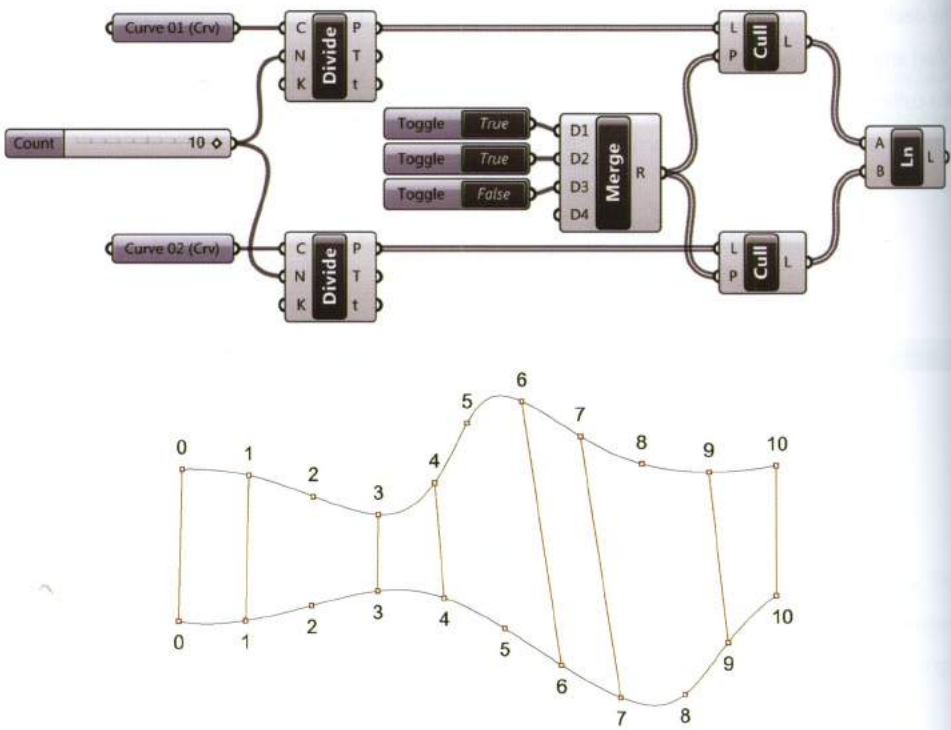
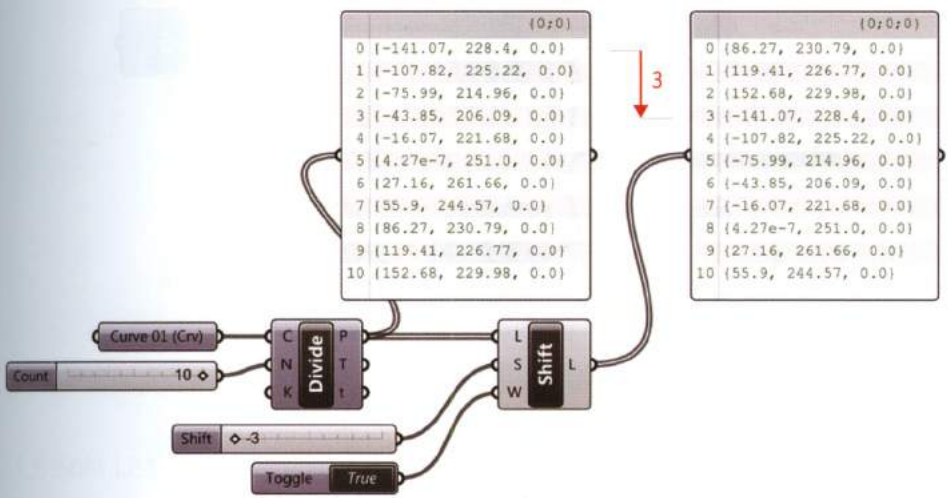


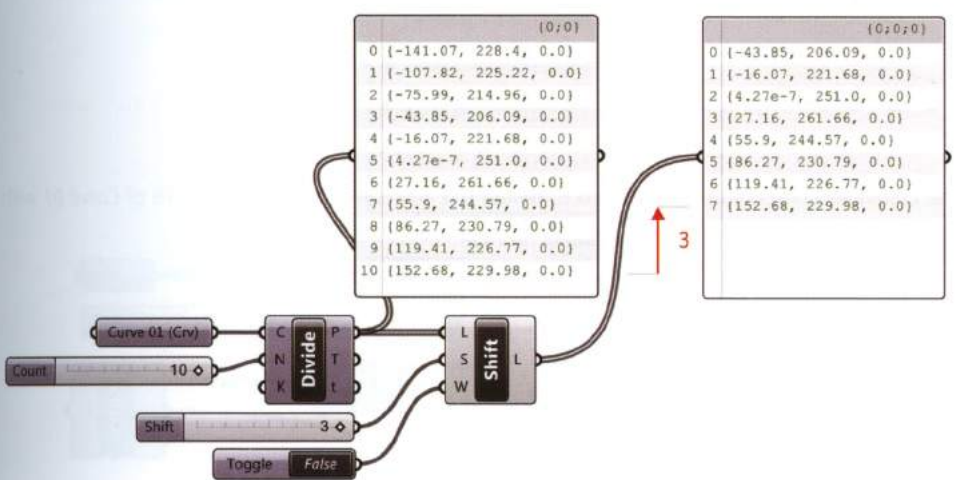
FIGURE 2.1
The image shows the result of *Cull Pattern* based on a repetition of three Boolean values: True, True, False.

2.1.4 Shift List: offset data in a list

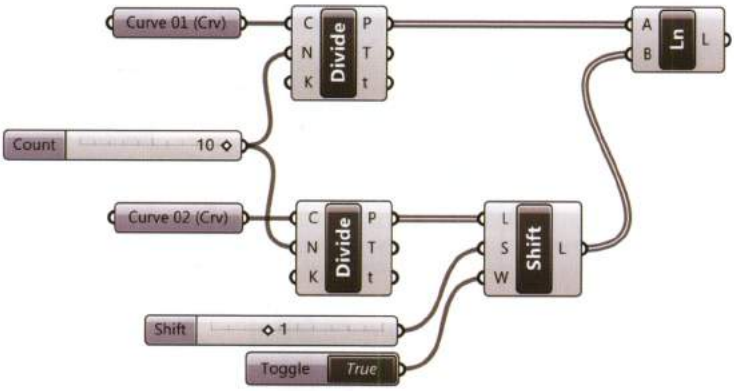
The component ***Shift List*** (Sets > List) shifts index numbers, upward or downward. The component's S-input is the shifting offset, the L-input is the list to shift, and the W-input (Wrap) is a Boolean Parameter which declares if the resulting shifting should be re-appended. Negative numbers shift the list downward and positive numbers upward. For example, If the shifting offset (S) is set to -3 and the wrap (W) is set to true, the list's original indexes are shifted 3 units towards the end of the list and the list is re-appended, meaning the bottom three indexes 8, 9, and 10 become indexes 0, 1, and 2 respectively.

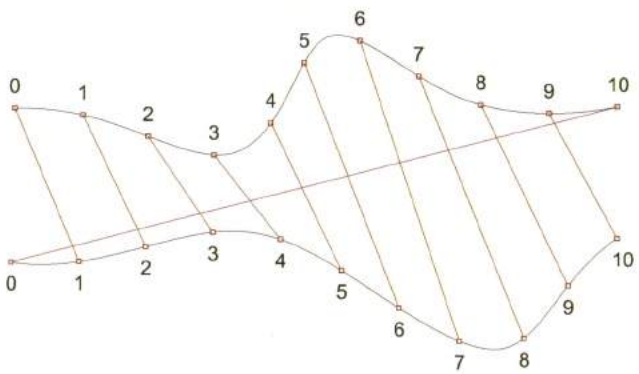


If the shifting offset is set to 3 and the wrap (W) is set to false, the original indexes are shifted 3 units toward the beginning of the list, and the first three values are culled.

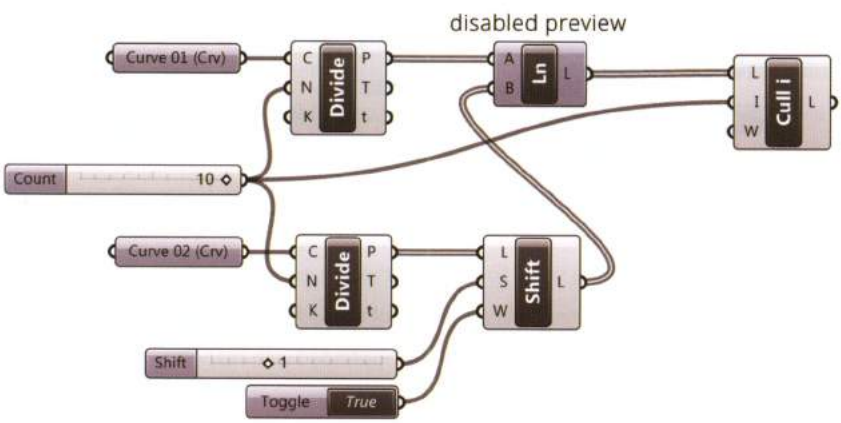


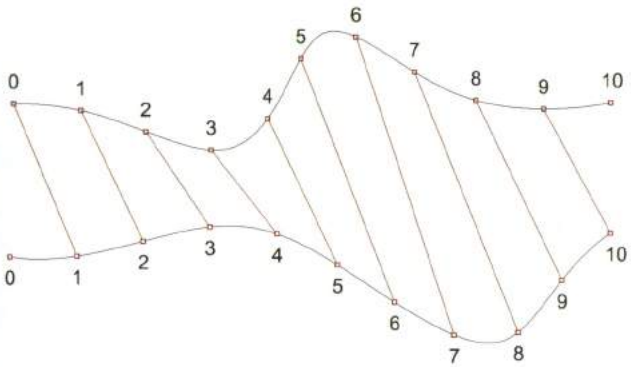
The following algorithm generates a set of lines connecting the generic point i on the first curve with the point i+1 on the second curve. The resulting output list of the *Shift List* component will be shifted 1 value towards the beginning of the list and the list will be re-appended with respect to the input list, meaning the index 10 of *Curve 01* will be connected to index 0 of *Curve 02*.





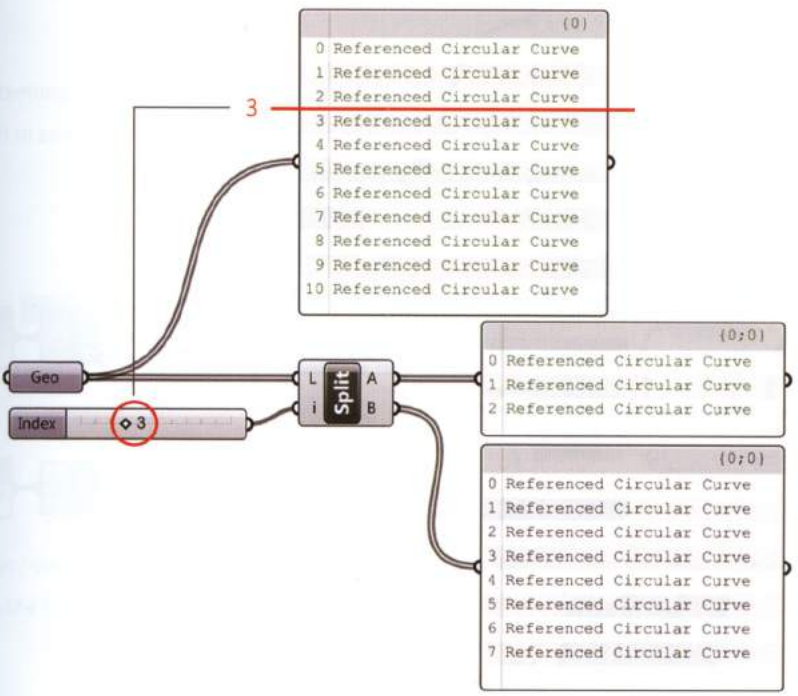
The *Cull Index* component can be used to delete the line connecting the index 10 of *Curve 01* with index 0 of *Curve 02*, by specifying the index number.



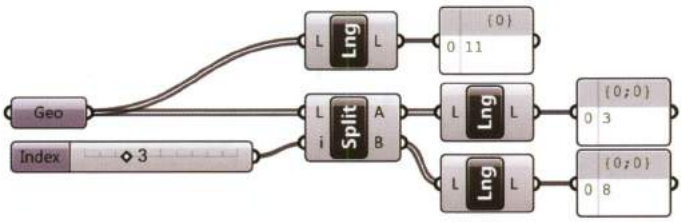


2.1.5 Split List / List Length

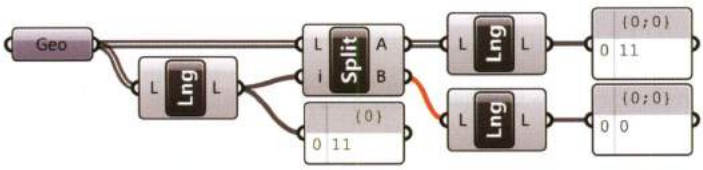
The ***Split List*** component (Sets > List) splits a single input list at a specific index into two lists A and B.



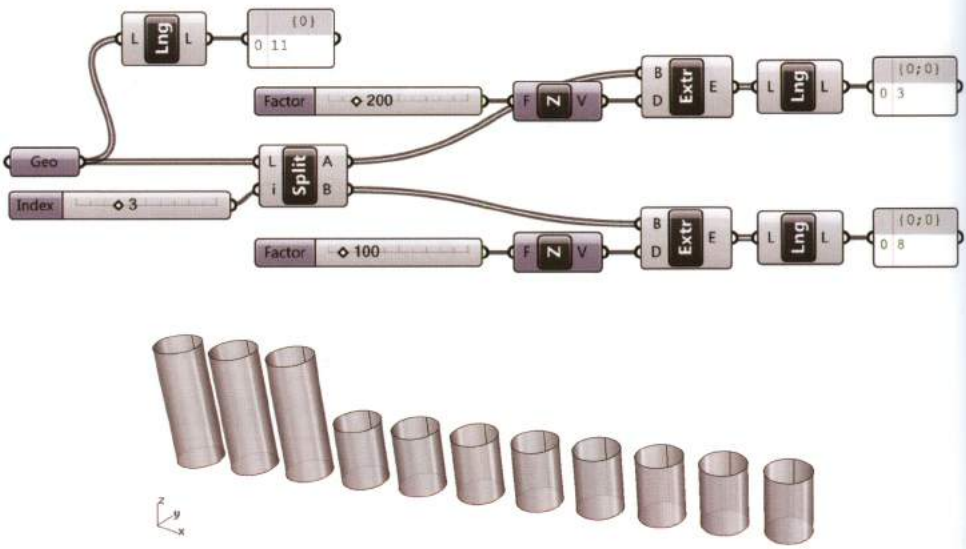
The Component ***List Length*** (Sets > List) calculates how many items there are in a list. To display the lists length, connect a *Panel* to the L-output of *List Length*.



If the splitting index is set equal to the *List Length*, List A will be the same as the initial list and List B will be empty.



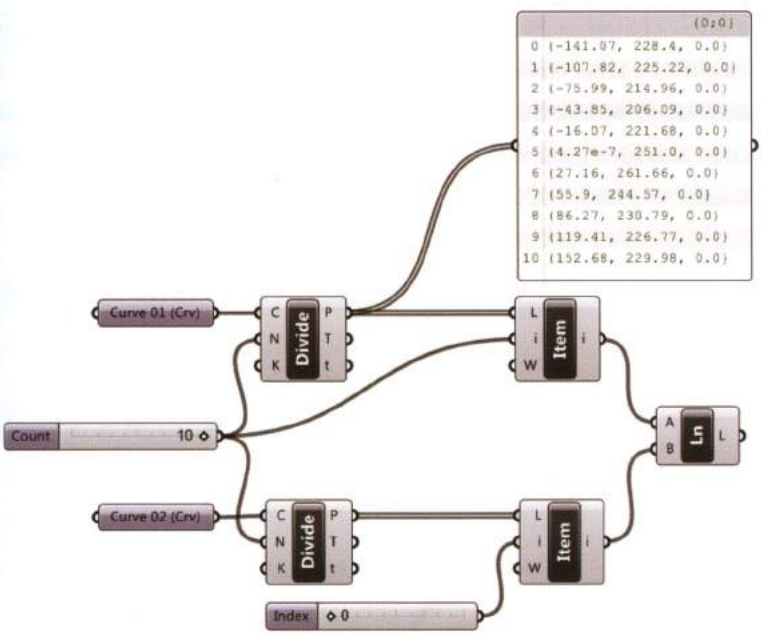
The *Split List* component can be used to carry out different operations on a set of geometries. For instance, after the set is split into two lists, list A is extruded a magnitude of 200 units in the *Unit Z* direction and list B is extruded a magnitude of 100 units in the same direction.



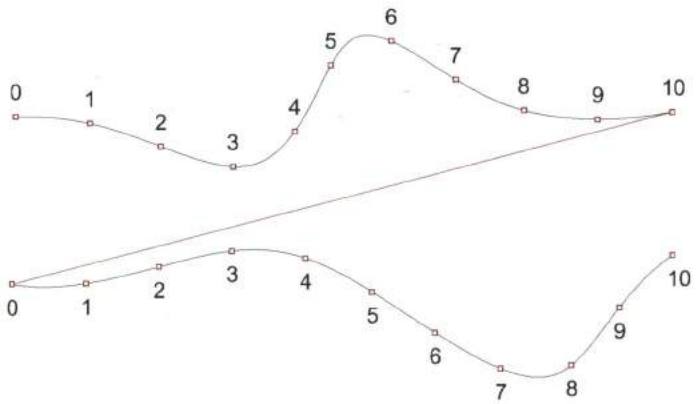
2.1.6 Reverse List

Frequently is useful to reverse the order of a list of data. Referring to the previous example, we want to create a line which connects the "last point" on the *Curve 01* with the "first point" on the *Curve 02*. In addition, such a rule must work even if changing the number of subdivisions through the N-input of *Divide Curve*.

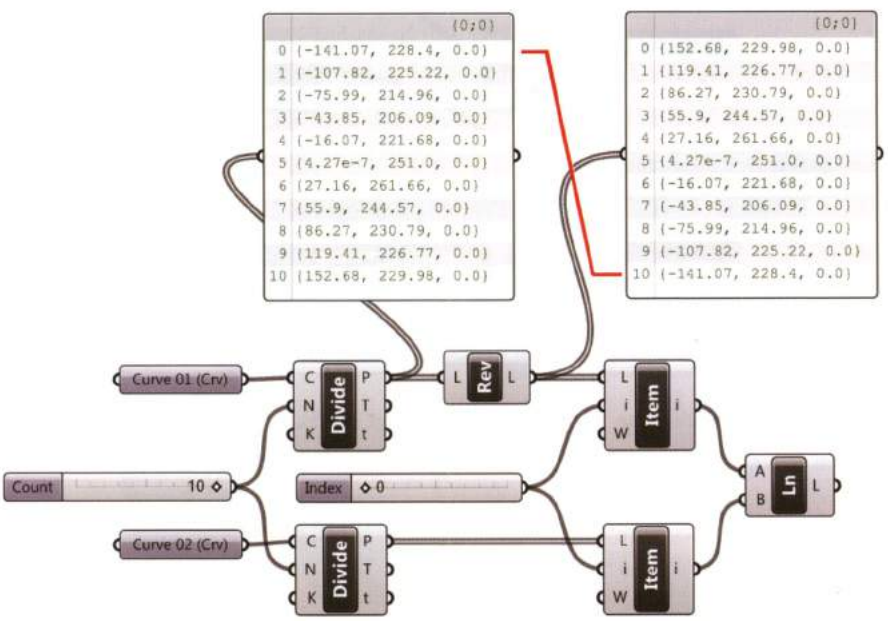
The "first point" can be easily found by a *List Item* component set to 0. The "last point", instead, depends on the number of subdivisions (N). To find this latter, we can rely on two main strategies. The first one consists in creating a "parametric link" using the same slider to "feed" the N-input and the i-input of the first *List Item*.

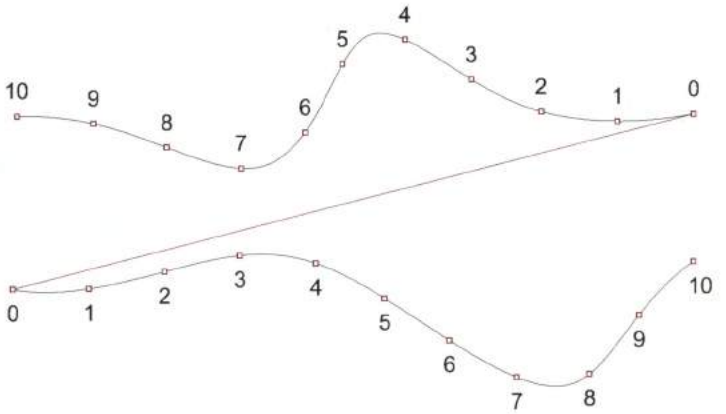


In this way, even if changing the number of subdivisions, the List Item will always select the "last point" on the Curve 01.



The second strategy is based on the component ***Reverse List*** (Sets > List) which reverses the order of a list (in this case the P-output list), meaning the index 10 is converted into 0, the index 9 into 1 etc. Every input slot embeds the functionality to reverse input data by right-clicking the slot and choosing *Reverse* from the contextual menu. Using *Reverse List*, the same slider set to 0 can feed the two *List Item* components.





2.2 Numerical sequences

The components *Series, Repeat Data, Random, and Range* create and manipulate numerical sequences to create lists that manage geometry.

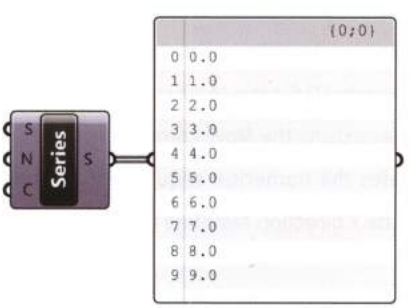
2.2.1 Series

The component ***Series*** (Sets > Sequence) generates numerical sequences according to:

- The first number in the series **(S)**;
- The step size for each subsequent number **(N)**;
- The number of steps in the series **(C)**.

The *Series* component default settings are: S=0, N=1, C=10, resulting in the sequence:

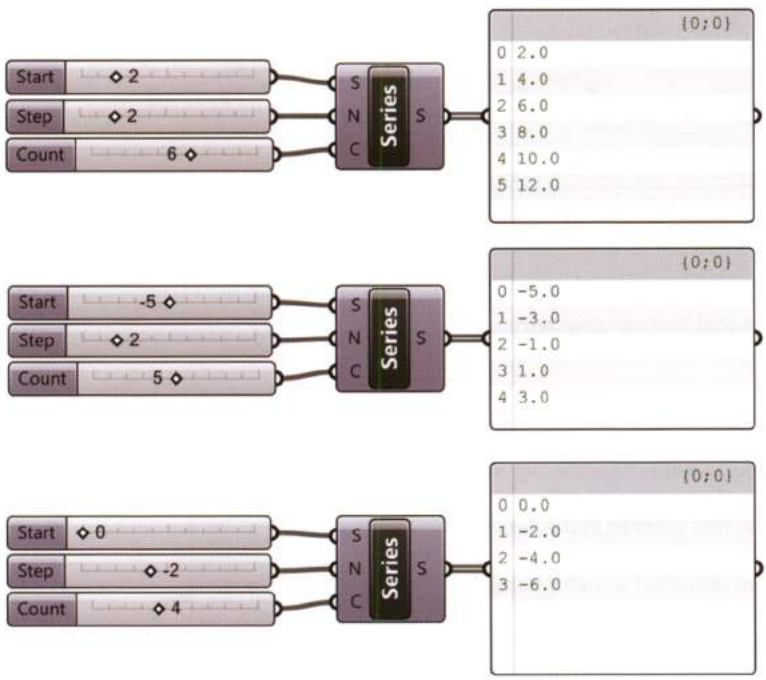
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).



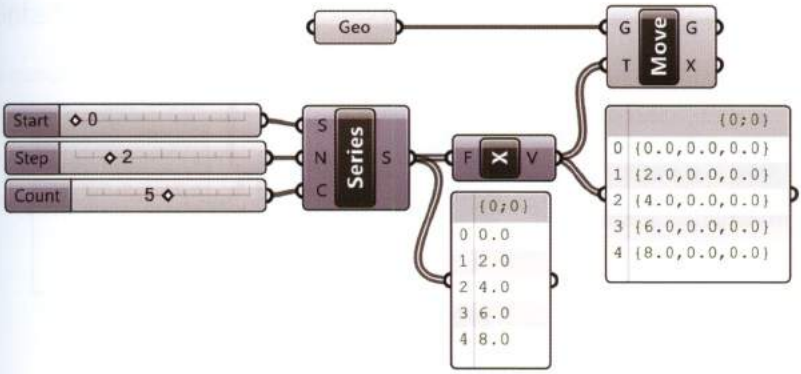
For instance, by setting the parameters to: S=2, N=2 and C=6, a series of 6 numbers is generated which starts from 2 and has a distance of 2 units between a number and its subsequent:

2, 4, 6, 8, 10, 12.

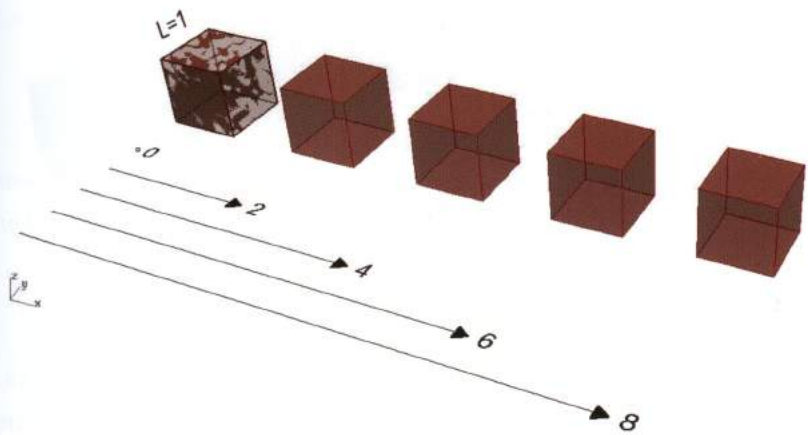
The *Series* component can create infinite unique numerical sequences that are increasing or decreasing based on the input parameters.



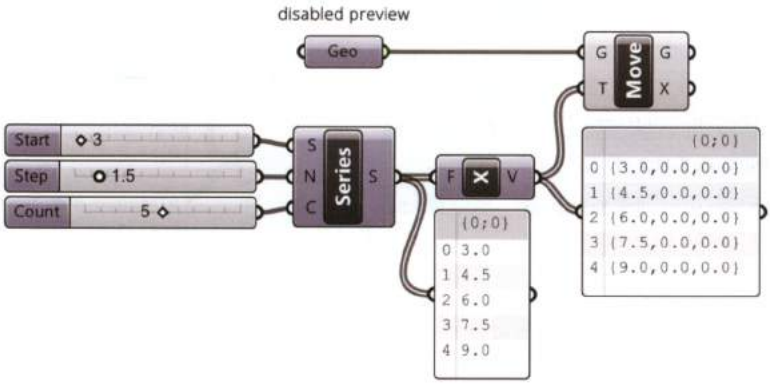
Series is often used along with transformation components, such as *Move*, in order to get multiple transformations. The component *Move* (Transform > Euclidian) translates a geometry (G) according to a vector (T). The *Series* component describes a list of scalar values which can be multiplied by a unit vector to define a translation vector (T) for the *Move* component. For example, a Rhino defined cube can be translated along the x-axis using the *Move* component in conjunction with *Series* component. The *Series* component generates the numerical sequence (0, 2, 4, 6, 8) of scalar values which are multiplied by a unit vector in the x direction resulting in the translation vectors (0x, 2x, 4x, 6x, 8x).



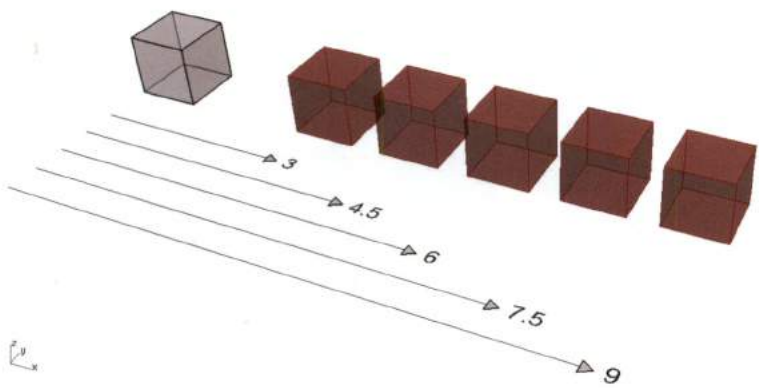
The translation vectors are connected to the T-input of the *Move* component, yielding a series of cubes translated *One Dimensionally*. The first translation vector is 0x, meaning the first translated geometry overlaps with the Rhino set geometry.



If you want to display just the translated cubes (output of *Move*), you have to disable the preview of *Geometry* as well as the preview of the initial object in Rhino (*Hide Objects* command).



The *Move* and *Series* components, used in combination, can create infinite unique vector sequences. The sequences can start at any value, have any step size and count to any length.

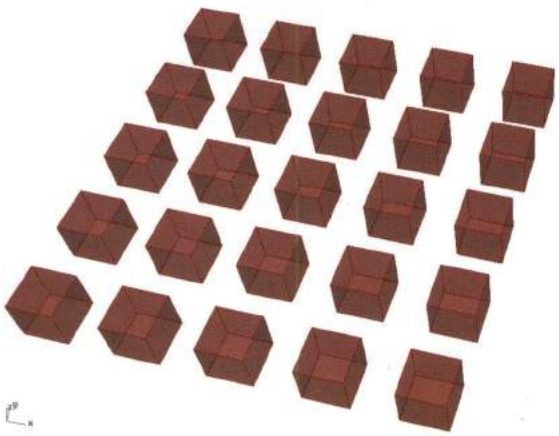


For instance, if (S) is equal to 3, (N) is equal to 1.5 and (C) is equal to 5, the sequence (3.0, 4.5, 6.0, 7.5, 9.0) will result. If the sequence is multiplied by the *Unit X* vector the translation vectors (3.0x, 4.5x, 6.0x, 7.5x, 9.0x) will be defined. Inputting the translation vectors into the T-input of the *Move* component defines the *One Dimension* translation. The first translation vector is 3.0x, meaning the first translated geometry will not overlap with the Rhino set geometry.

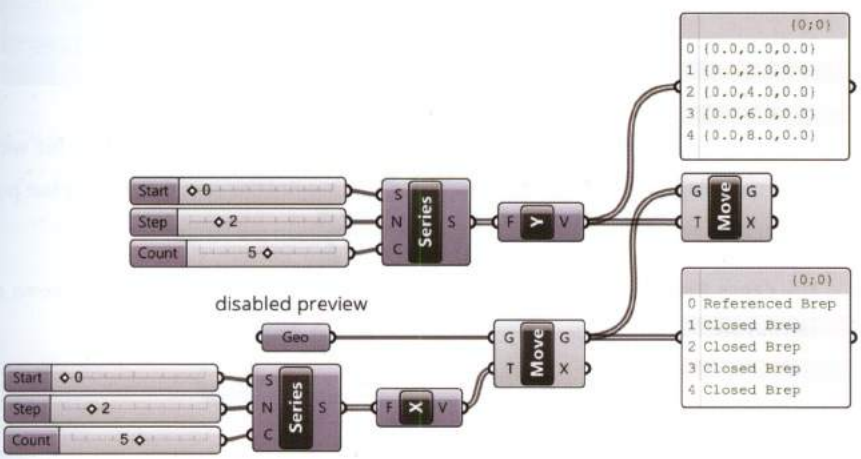
2.2.2 Data Matching

Usually a component receives two or more wires and as many data streams. When data enter into a component they are matched according to a default *logic* which must be understood in order to get specific results. An example will explain this concept.

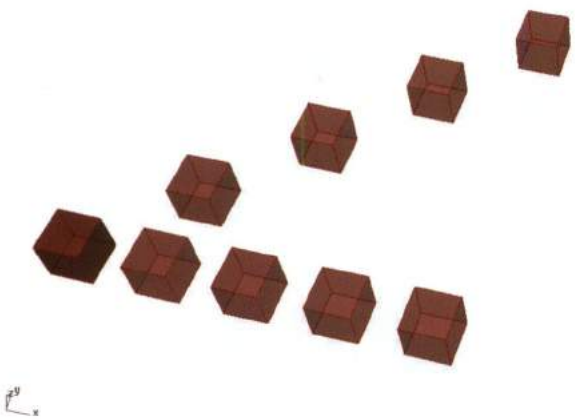
Starting from the previous algorithm we aim to get the following cube grid (based on a 5 x 5 grid).



As first attempt we can modify the algorithm by adding a new *Move* component connected to a *Series* in order to translate the five cubes (output by the first *Move*) according to the y-axis.

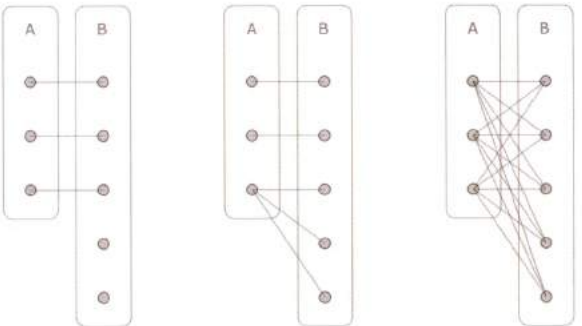


Nevertheless, the final output is quite different from a cube grid: the boxes will create a diagonal arrangement.



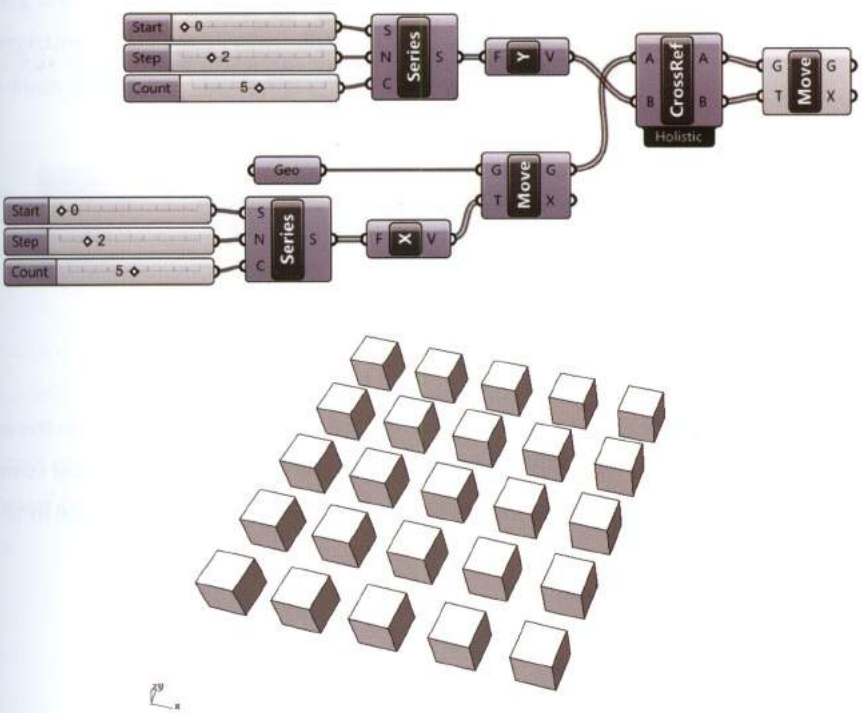
Such a result depends on a particular logic called "**Data Matching**" which, by default, matches – one by one – corresponding items from different lists entering into the same component.
In the example, the second *Move* component matches two flows, the first one transferring geometries (G-input), the second one transferring vectors (T-input). According to this logic the *Move* component moves the first geometry according to the first vector (0 length), the second geometry according to the second vector (whose length is 2) and so on.
In order to get the square grid, Grasshopper must move each cube according to all vectors. This implies that we have to manipulate the matching logic.
Grasshopper matches data according to three main methods:

- The **Shortest List** mode matches each item in the first list with the corresponding item in the second list. If the lists are not the same length the shorter list length will shrink the longer list to the same length.
- The **Longest List** mode (default method) matches each item in the first list with the corresponding item in the second list. If the lists are not the same length the last point in the shorter list will be connected to remaining points in the longer list.
- The **Cross Reference** mode matches each item of the first list with all the items of the second list.

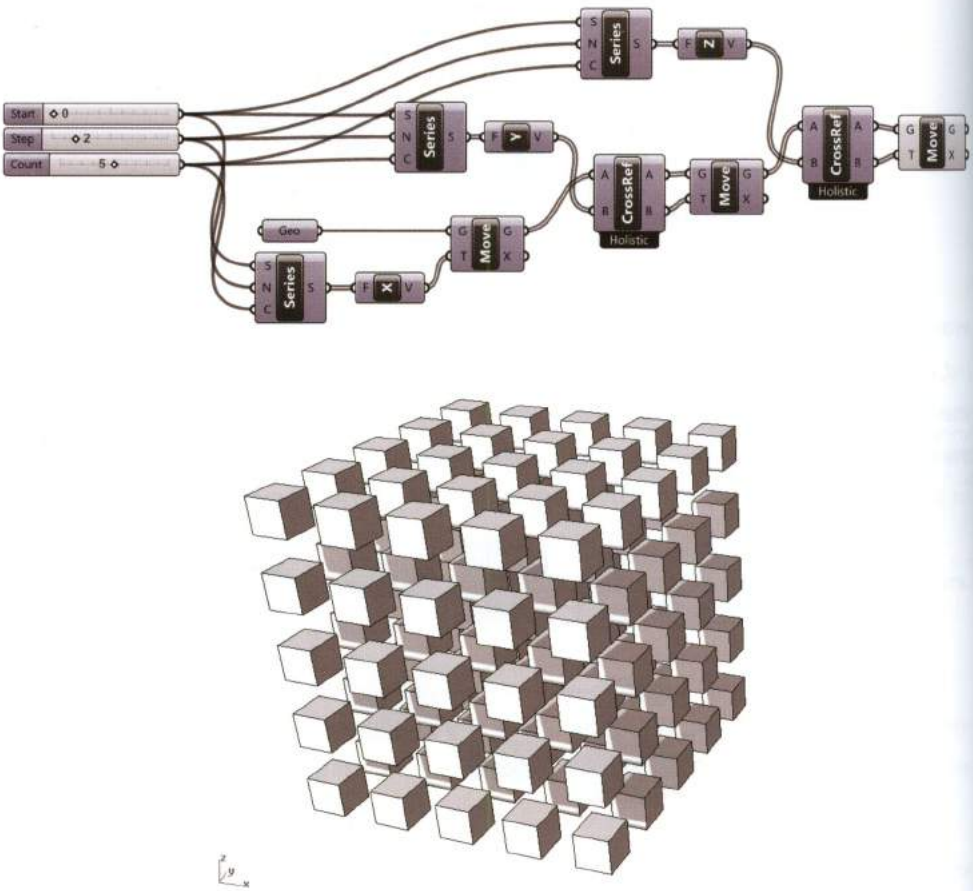


In order to modify the matching logic Grasshopper provides us of three specific components: *Shortest List*, *Longest List* and *Cross Reference* all hosted within the *List* panel of the *Sets* tab.

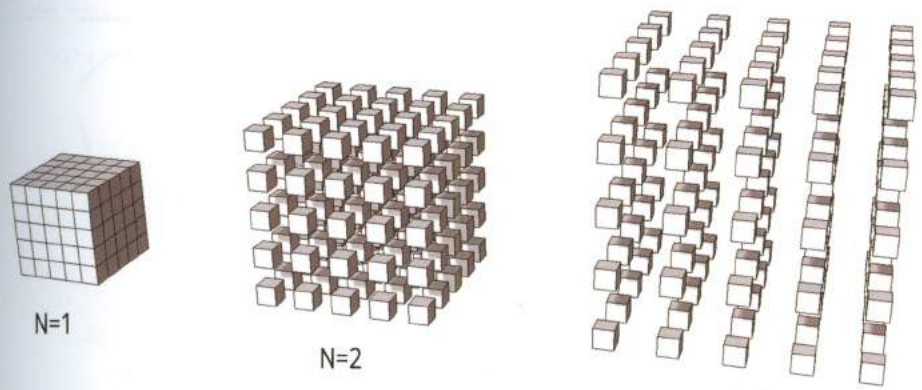
The component *Cross Reference* can be used, in the example, to create a two-dimensional array i.e. the desired box grid, as illustrated in the following image.



The component *Cross Reference* can also be used to create three-dimensional arrays, by connecting a z translation vector sequence and "cross referencing" the data with the second move component.

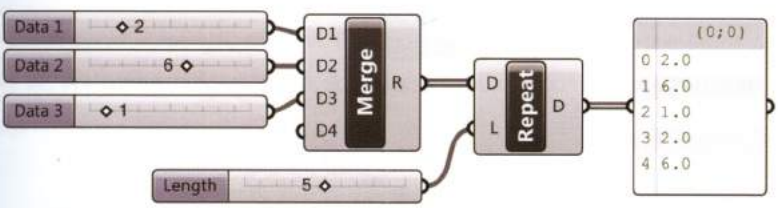


The algorithm uses the same three *Number Sliders* to feed the three *Series* components. In this way we can change the distance between cubes (in all the three directions) by acting on the slider connected to the N-input of the three *Series* (in our example the initial cube has a side length equal to 1).



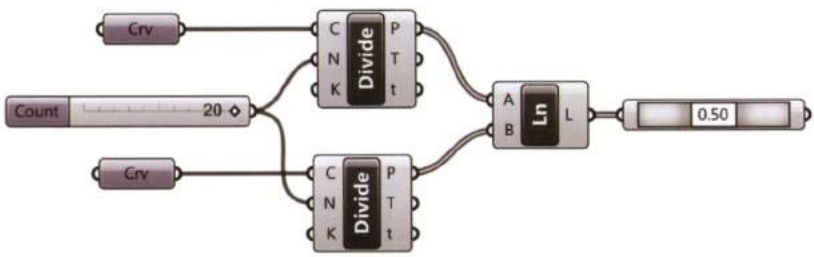
2.2.3 Repeat Data

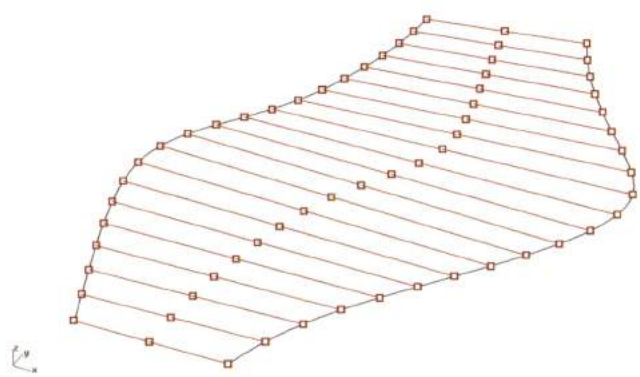
Another useful component is ***Repeat Data*** (Sets > Sequence) which extends a numeric sequence to a specified length by repeating the input data. The component's D-input collects a defined set of numbers, and the L-input defines the length of the output list. For example, if the numbers (2,6,1) are merged into a single list connected to the D-input of *Repeat Data*, the resulting list is (2, 6, 1, 2, 6) if the L-input is set to 5.



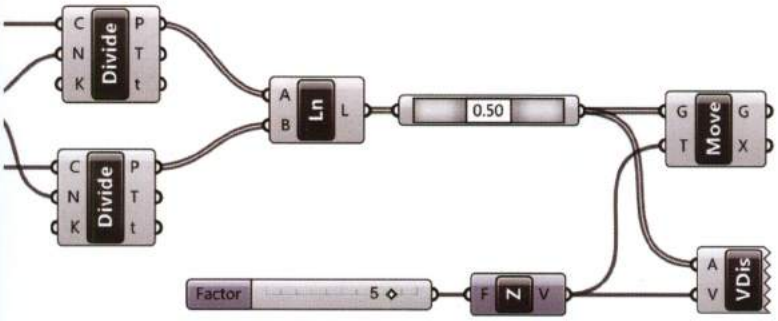
The output of the *Repeat Data* component can be used to define sequential geometry. To visualize how *Repeat Data* can manipulate geometry, a simple algorithm will be used to get the following surface.

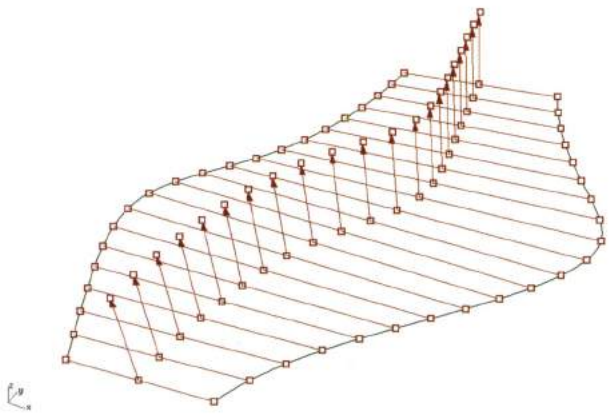
Two curves set from Rhino are divided using the *Divide Curve* component, and lines are connected to corresponding points in the lists using the *Line* component. The lines are evaluated using *Point on Curve* to calculate the midpoint of each line.



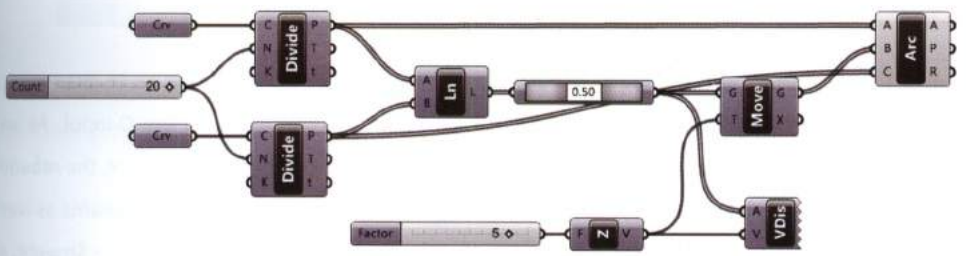


Each midpoint is moved by a translation z-vector (visualized by the *Vector Display* component), using the *Move* component.

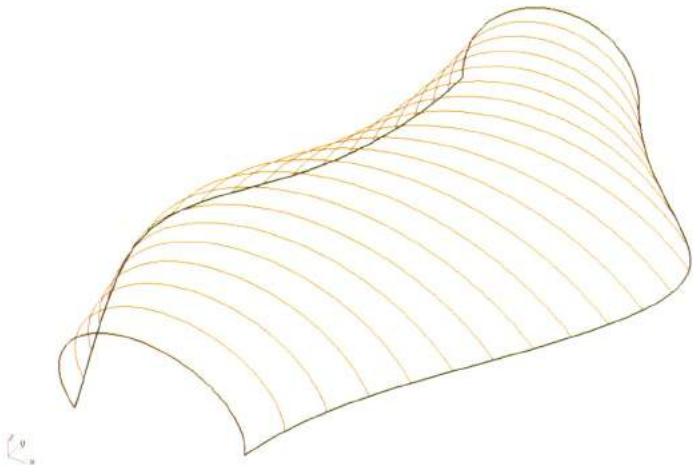




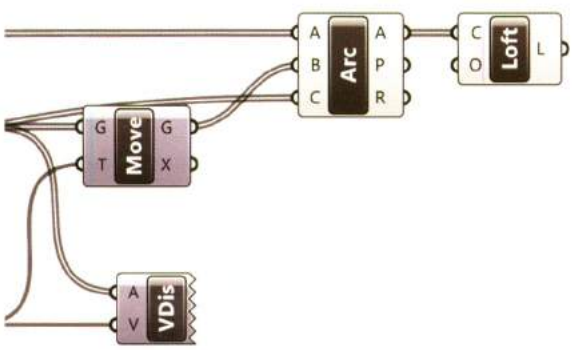
Then, the *Divide Curve* output (P) of *Curve 01*, the translated points, and the *Divide Curve* output (P) of *Curve 02*, are connected into the A, B, C slots of the component *Arc 3Pt* (Curve > Primitive) respectively.



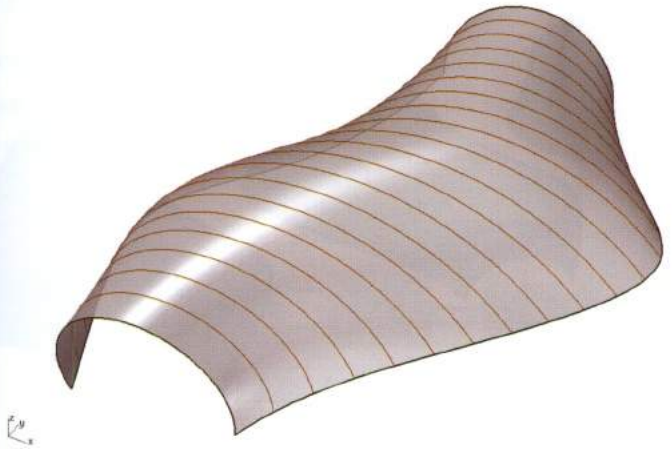
The following image displays the final set of arcs. We turned off the preview of all the components except *Arc 3Pt* (you can also see the two curves drawn in Rhino).



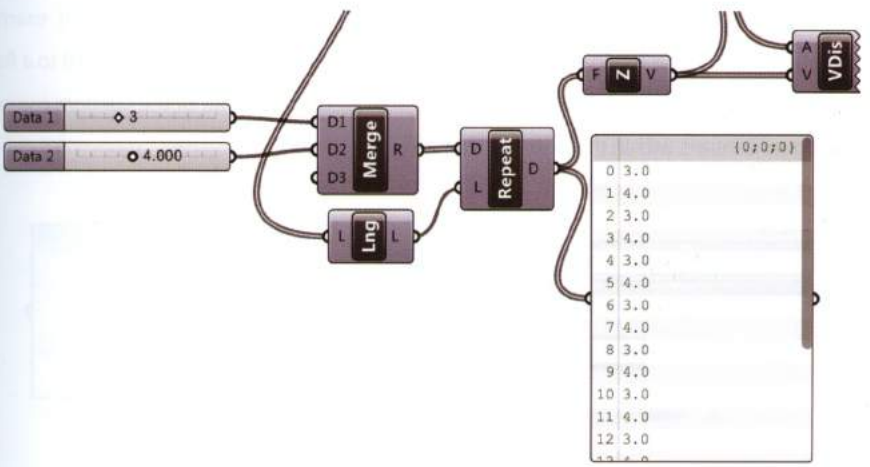
Lastly, the A-output of the *Arc 3Pt* component is connected to the C-input of the component ***Loft*** (Surface > Freeform) to generate a lofted surface, defined as a surface through a set of section curves.



The O-input of *Loft* is used to modify surface options. Options can be locally set or, in alternative, modified by the ***Loft Options*** component (Surface > Freeform) connected to the O-input. As an instance, when the number of section curves is not enough to create a "smooth" surface, the ***rebuild*** option (Rbd-input) must be set. *Loft Options* also allows to close the loft, to adjust the seams as well as select the loft type, which is defined by an integer (0 = Normal, 1 = Loose, 2 = Tight, 3 = Straight, 4 = Developable, 5 = Uniform).



The algorithm is now complete. To modify the surface, the lofted arcs can be manipulated using the *Repeat Data* component. Modifying an algorithm by changing the inputs is a common working method in Grasshopper. For example, if the *Number Slider* connected to the *Unit Z vector* component is replaced with a list of repeated numbers (with the *List Length* set equal to the number of items of the *Line* component) the lofted geometry will be updated, and express the change to the z-translation in its surface shape. The surface can be varied through infinite changes of the sequence input into the *Unit Z* component.



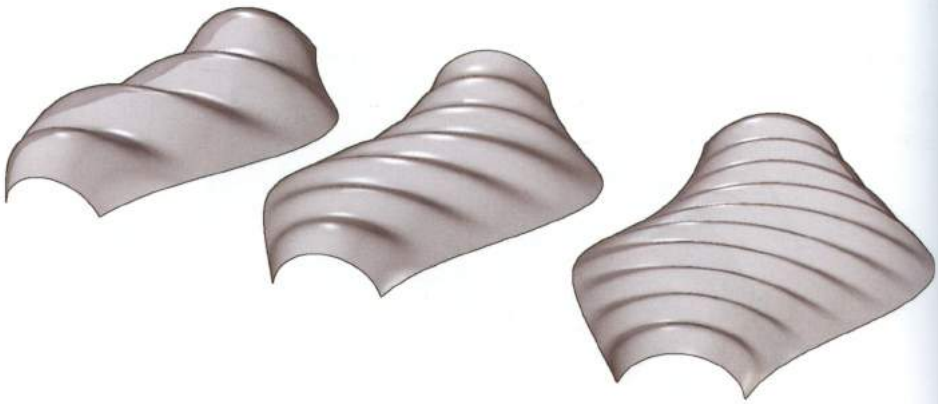
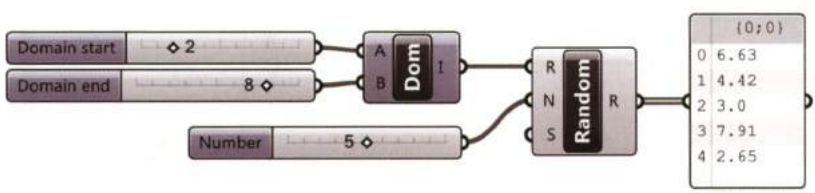


FIGURE 2.2
Shapes informed by data.

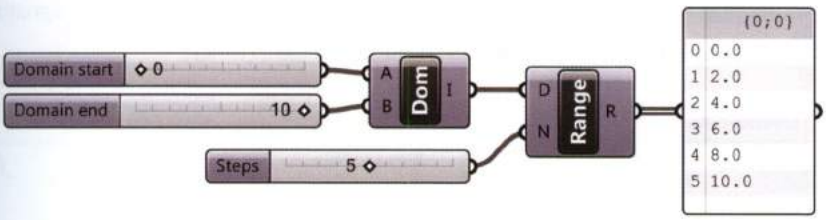
2.2.4 Random / Construct Domain

The component *Random* (Sets > Sequence) generates a list of (N) random numbers within a defined numeric domain (R). The S-input specifies the seed value of the component, if the seed value is changed and all other inputs remain the same a new list of random values will be generated. By default, the *Random* component generates real numbers. To specify integers right-click on the *Random* component and select the *Integer Numbers* option from the context menu and a black *Integers* flag will appear at the bottom of the component. The component *Construct Domain* (Maths > Domain) is used to define a numeric domain between two numeric extremes. For example, a domain defined between a minimum value of 2 and a maximum value of 8 is connected to a *Random* component specifying a list with five items is to be generated. The output of the *Random* component is five random numbers within the numeric domain.



2.2.5 Range

The component *Range* (Sets > Sequence) defines a range of numbers within a numeric domain (D) with (N) subdivisions. In other words, numeric domain is divided into equal parts. When the domain is divided into N parts N+1 values are defined.

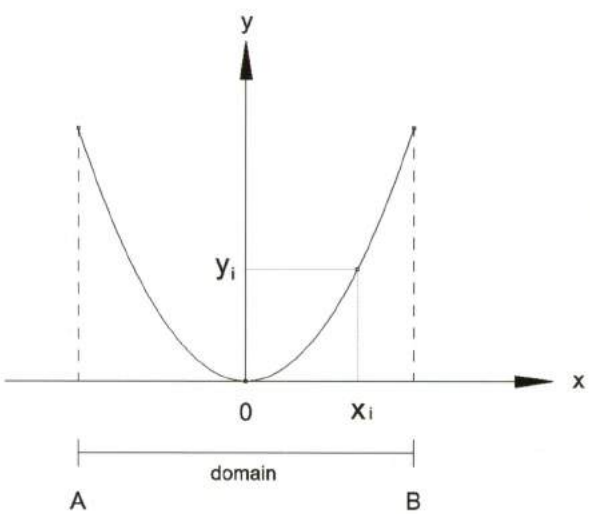


2.3 Mathematical Functions

The component *Evaluate F(X)* (Maths > Script) can be used to define numerical sequences that follow **mathematical functions**.

2.3.1 Functions of one variable

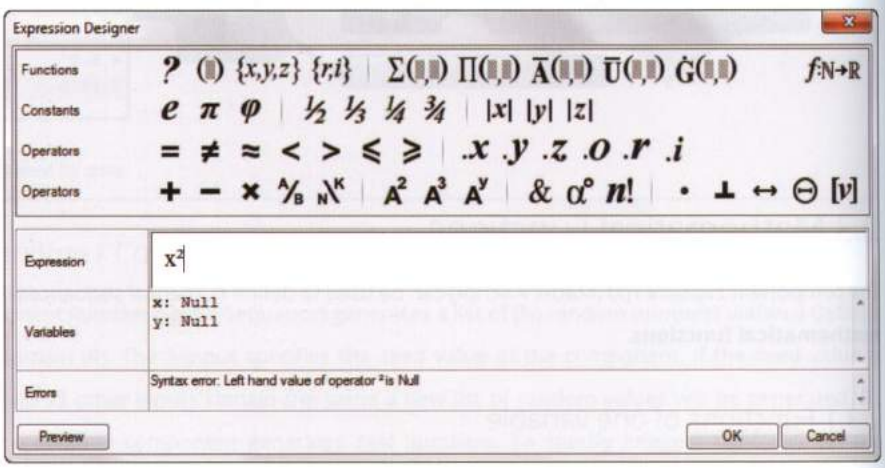
A function of one variable has as a **domain** of values in the x axis and a **codomain** of values in the y axis, meaning for each input value (xi) the function provides an output value (yi).



To build an algorithm with a mathematical function:

1. **Defining a function**

   The *Evaluate F(X)* component, can define a function by double-clicking on the node to display the *Expression Designer*. For instance, a parabola can be defined by the *explicit equation* $y=x^2$, so x^2 would be typed in the *Expression Designer*. The square operator can be found within the *Operators* library and a complete list of functions and their signatures are available by clicking the *f:N>R* button (at the upper right corner).



   Alternatively, a function can be defined by entering an equation in a *Panel* and connecting its output to the F-input slot of the *Evaluate* component.

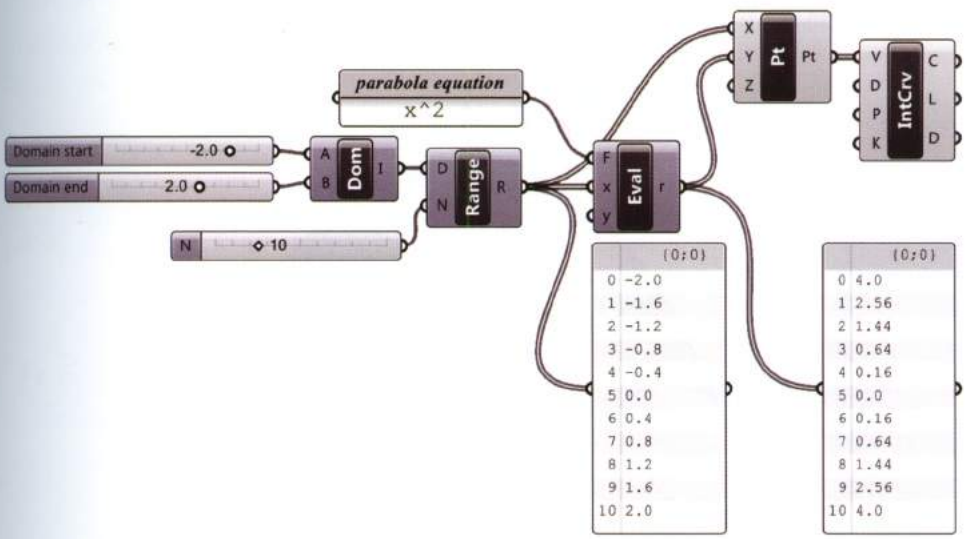
2. **Defining a domain**

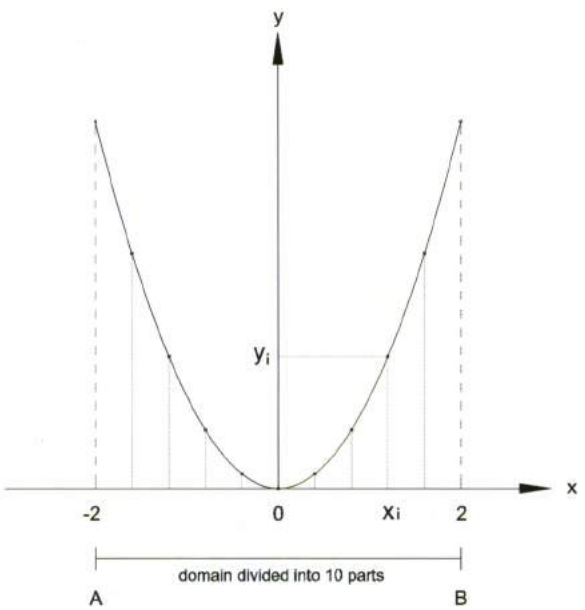
   In order for a function to return a result, the function requires numerical values as input. The component *Construct Domain* in conjunction with the component *Range* can generate numerical output. For example, if a domain [-2, 2] is connected to a range component with N set to 10, 11 values will be generated which are equally divided with respect to the domain. The output numerical values can be used as input for the *Evaluate F(X)* component.

The r-output of the *Evaluate* component is the codomain of the function. Examining the output of the *Evaluate* component, the list of numbers decreases until zero and then increases again after zero; the expected results of the function x^2.

Each point of a planar mathematical curve has two coordinates – x and y – where x is a value from the domain and y the output codomain. If the outputs of *Range* component (R) and the *Evaluate*

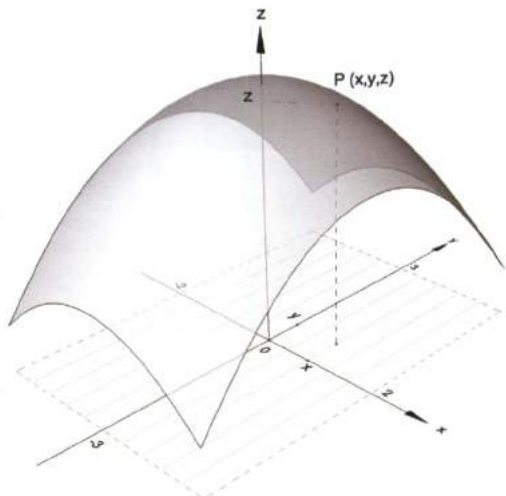
component (r) are connected to the *Construct Point* components (X) and (Y) slots respectively a set of points which define the graphical appearance of the function x^2 will be generated. A curve can be drawn through these points using the component *Interpolate Curve* (Curve > Spline).





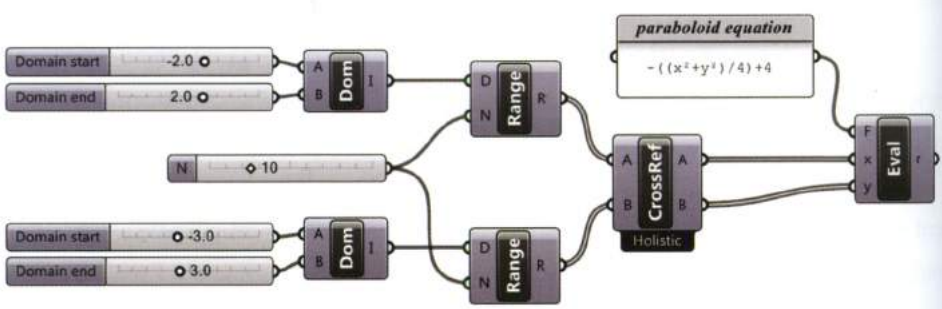
2.3.2 Functions of two variables

A function of two variables has a domain composed of a subset of points belonging to the XY-plane and a codomain of values in the z axis, meaning for each input value (xi,yi) the function provides an output value (zi).

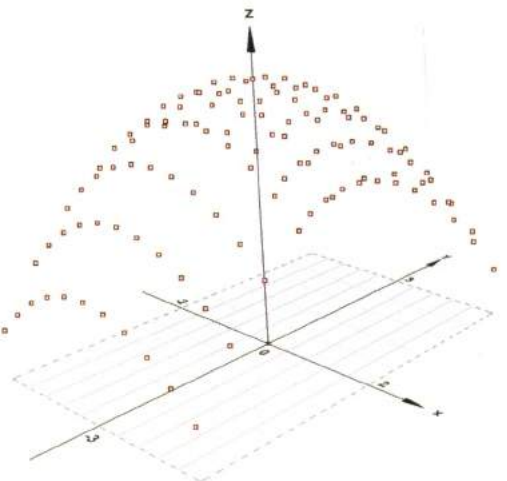


To generate a surface from a function of two variables: first, define an equation and second define the domain. Then construct points with respective (xi,yi,zi) output and interpolate a surface through the points. Functions of two variables have a bi-dimensional domain, meaning input values for (xi) and (yi) are required to be defined.

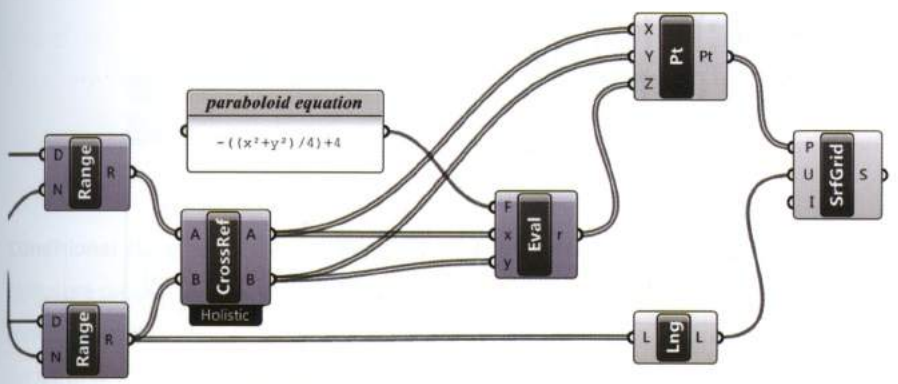
For example, a paraboloid can be defined by the equation: $z=-((x^2+y^2)/4)+4$. To generate a grid of points to define a paraboloid surface, the (xi) and (yi) input lists are required to be "cross referenced" before evaluating the function.



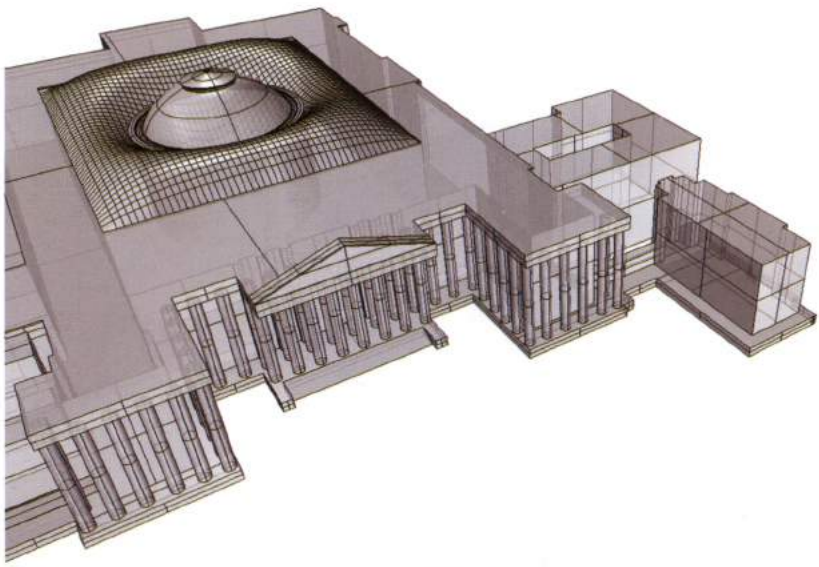
The *Evaluate* component generates a (zi) coordinate for each (xi,yi) input pair based upon the specified equation. The *Construct Point* component combines the data from the (xi), (yi), and (zi) output lists into a point (xi, yi, zi).



The component *SrfGrid* (Surface > Freeform) creates a surface from a grid of points (see 3.6). To generate the paraboloid surface the Pt-output of *Construct Point* is connected to to the P-input of the *SrfGrid* component, and the *List length* component is used to specify the number of points in the x or y direction to satisfy the (U) input of the *SrfGrid* component.



The following image illustrates how is possible to get the actual configuration of the **British Museum Great Court Roof** referring to the equations published by Chris JK Williams[7]. The roof's shape is generated by a sum of three equations of two variables.



$$z=\frac{h\left(1-\frac{x}{b}\right)\left(1+\frac{x}{b}\right)\left(1-\frac{y}{c}\right)\left(1+\frac{y}{d}\right)}{\left(1-\frac{ax}{rb}\right)\left(1+\frac{ax}{rb}\right)\left(1-\frac{ay}{rc}\right)\left(1+\frac{ay}{rd}\right)}$$

$$z=H\left(1-\frac{x}{b}\right)\left(1+\frac{x}{b}\right)\left(1-\frac{y}{c}\right)\left(1+\frac{y}{d}\right)\left(\frac{r}{a}-1\right)$$

$$z=\frac{\eta\left(\frac{r}{a}-1\right)}{\left(\begin{array}{l}\frac{\sqrt{(b-x)^2+(c-y)^2}}{(b-x)(c-y)}+\frac{\sqrt{(b+x)^2+(c-y)^2}}{(b+x)(c-y)}\\+\frac{\sqrt{(b-x)^2+(d+y)^2}}{(b-x)(d+y)}+\frac{\sqrt{(b+x)^2+(d+y)^2}}{(b+x)(d+y)}\end{array}\right)}$$

NOTE 7

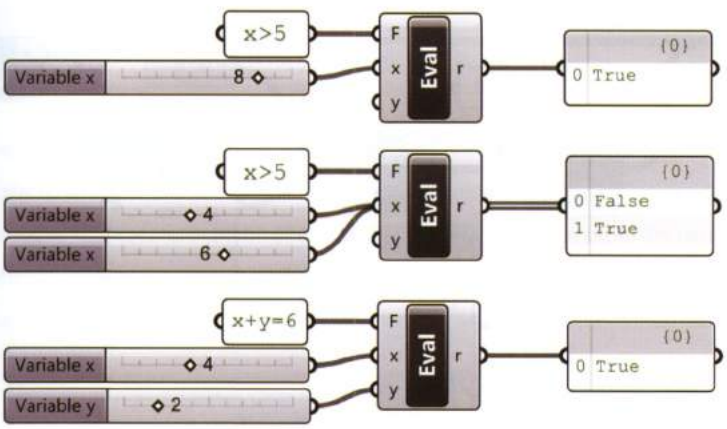
C.J.K. Williams, *The analytic and numerical definition of the geometry of the British Museum Great Court Roof*, (Mathematics & design, 2001), 434–440.

2.4 Conditions

The ***Evaluate*** component is not only useful for creating curves and surfaces from specific equations, but it also allows us to define **conditions** in Grasshopper. Often it's crucial that an algorithm performs different operations if a **condition** is met or not. A condition can be set in the F-input of *Evaluate* (in alternative, you can type the condition inside the *Expression Designer* editor) and it may include an arbitrary number of variables (x, y, ...).

2.4.1 Logical operators/Boolean values

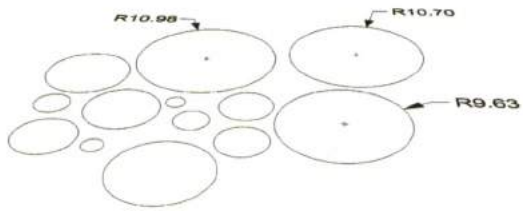
Conditions are build through variables and logical (comparison) operators such as: *Equality* (=), *Smaller Than* (<), *Larger Than* which can be found within the *Operators* panel of the *Maths* tab. Other operators (e.g. *inequality, smaller than or equal*) can be found within the *Expression Designer* editor. For example, the *Evaluate* component can be used to test if a set of numerical variables meets a defined condition. *Evaluate* returns **Boolean values**[8]: **True** if the condition is met, and **False** if the condition is not met.

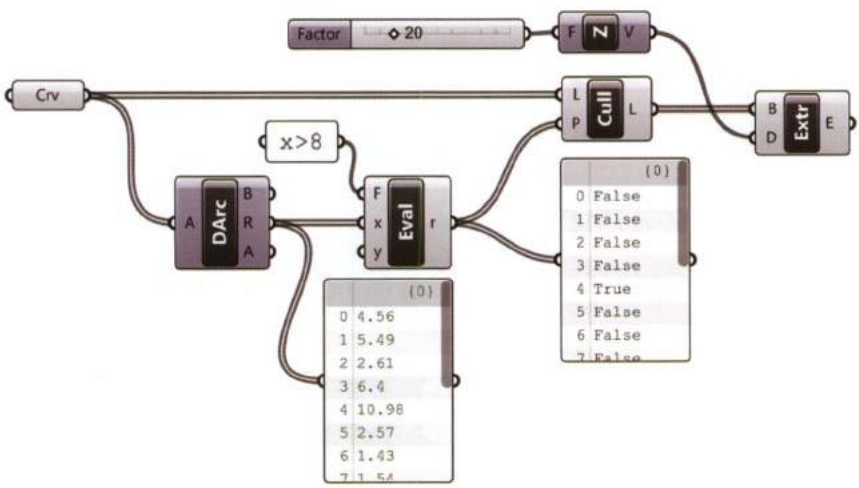


Conditional statements can be used to define *Cull Patterns*. For example, a set of circles whose radius are calculated using the component *Deconstruct Arc/DArc* (Curve > Analysis) are compared to the conditional statement R > 8, circles which satisfy this condition are extruded 20 units in the z direction, circles which do not satisfy this condition are culled.

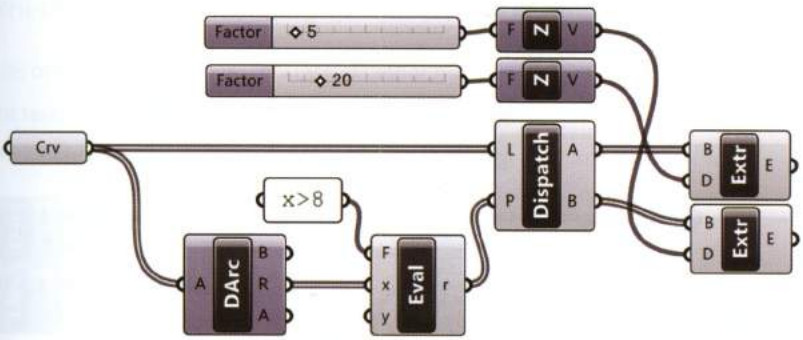
NOTE 8

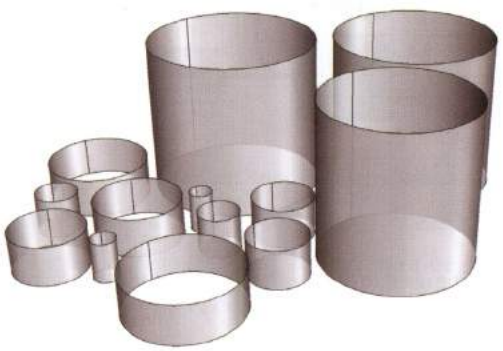
Named after the English mathematician George Boole.



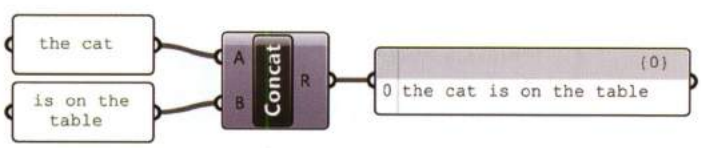


Alternatively, the component ***Dispatch*** (Sets > List) can be used to filter data with a corresponding Boolean values list. The *Dispatch* component returns two lists: **list A** contains data which returned True i.e. met the conditional statement, and a **list B** contains data which returned False i.e. data that did not meet the conditional statement. Referring to the previous example, *Dispatch* compares the radius of circles to the conditional statement (R>8) and sorts the circles so that circles which return True are extruded 20 units in the z direction and circles which return False are extruded 5 units in the z direction.

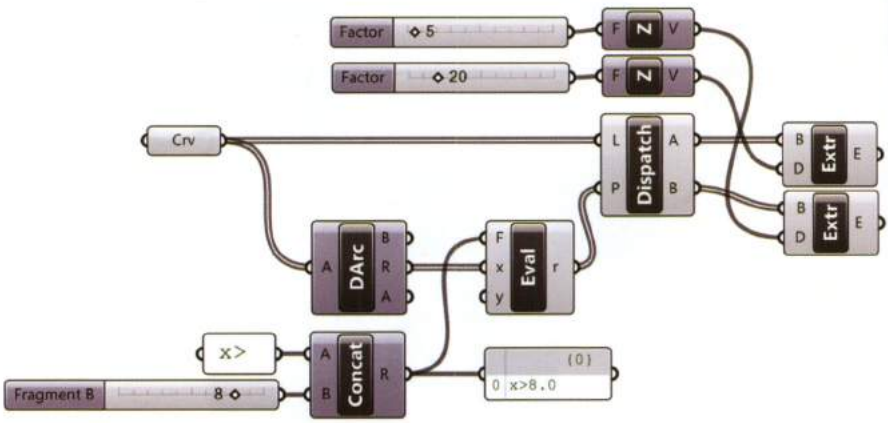




The component ***Concatenate*** (Sets > Text), can be used to join two or multiple fragments of text. For instance the statement "the cat is on the table" can be created by concatenating the panels "the cat" and "is on the table." *Concatenate* can be used to write a "parametric" condition by concatenating the text "x>" typed inside a *Panel* (do not press ENTER on your keyboard after typing "x>" but confirm with OK within the *Panel Properties* window) with a *Number Slider* or a generic number input.

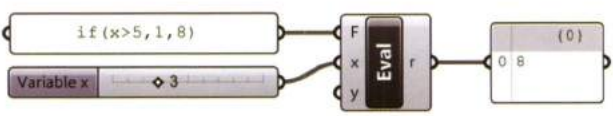


The *Concatenated* output can then be used as the F-input of the *Evaluate* component.

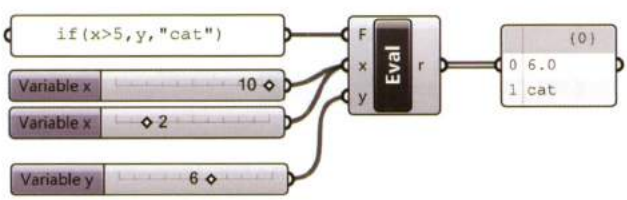


2.4.2 Conditional: if / then

The *Evaluate* component can be used to define if/then statements, meaning the conditional statement **if (test, A, B)** returns A if test is True, B if test is False.

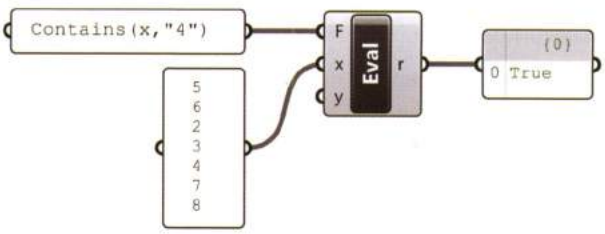


Inputs A and B do not have to be numbers. For instance, the text "cat" can be returned when the statement is False.

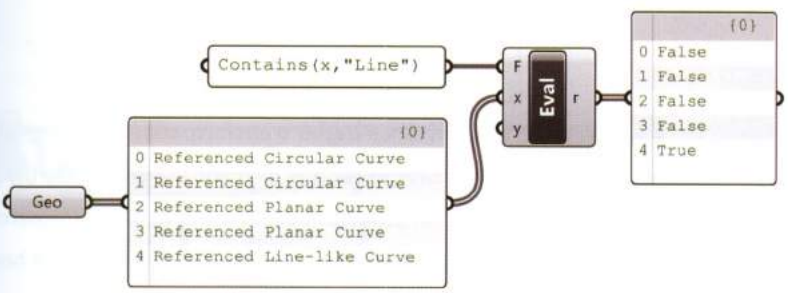


2.4.3 Other operators: Contains

The *Contains* operator **Contains (s, "p")** tests whether the string **p** is in the list of data **s**. The *Evaluate* component tests the operator and returns a Boolean value. The following example tests if "4" occurs within the list of numbers connected to the x-input, and yields True as a Boolean value.

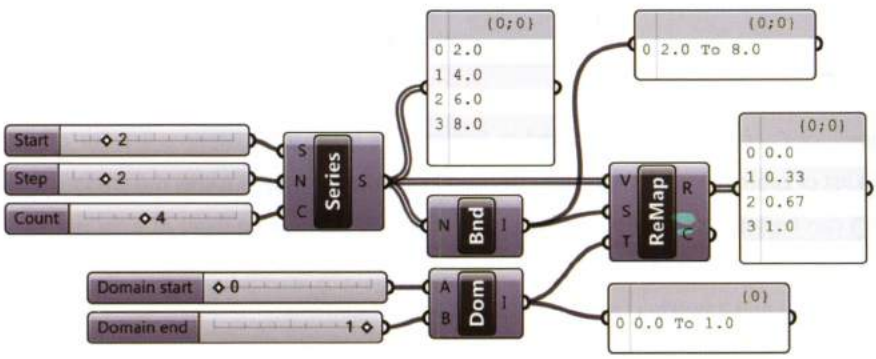


The operator *Contains* can also be used to find specific geometries within a list. For example, to test if a list of geometries (imported by a *Geometry* container) contains a line, the operator Contains (x,"line") can be set in the F-input of *Evaluate*. Then, a *Panel* must be used as a "bridge" between *Geometry* and the x-input of *Evaluate*. Panel's descriptions containing the text "Line" will result in a True statement and any other text will result in a False statement.



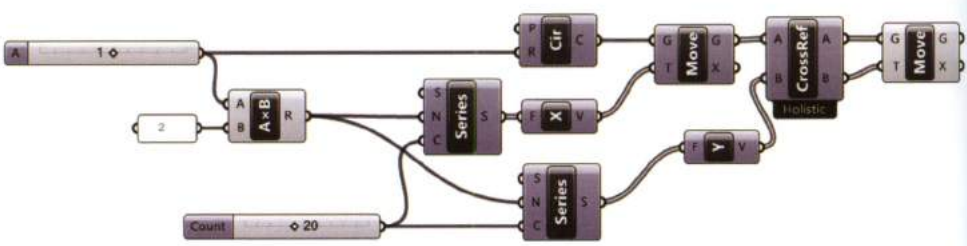
2.5 Remapping numbers / Attractors

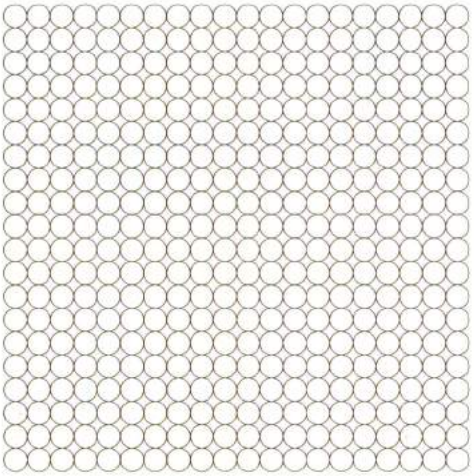
The component ***Remap Numbers*** (Maths > Domain) evaluates a list of numbers ranging from A to B, and resizes them proportionally to a new numeric domain A' to B'. The *Remap* component requires a list to remap (V), a source domain (S), and a target domain (T). The source domain of the numerical sequence can be found using the component *Bounds* (Maths > Domain), while the target domain is specified using the *Construct Domain* component. For example, the list of numbers (2, 4, 6, 8) whose source domain is [2,8] can be remapped to a new domain [0,1], yielding the list of values (0.0, 0.33, 0.67, 1.0).



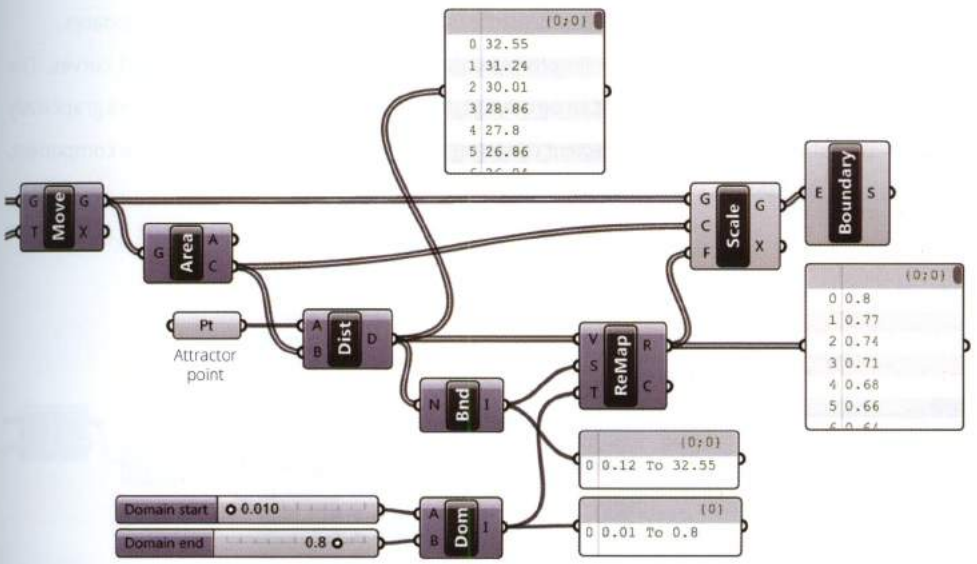
2.5.1 Attractors

The *Remap* component is used to perform **distance-based transformations**, utilizing "attractors." An **attractor** is a geometric entity: a point, a curve or another element, used to modify the geometry around it within defined limits. The impact an attractor has on other geometry depends on the distance between the defined attractor and the object it is manipulating. To visualize how a point attractor can scale a grid of circles a simple algorithm will be used as an example.



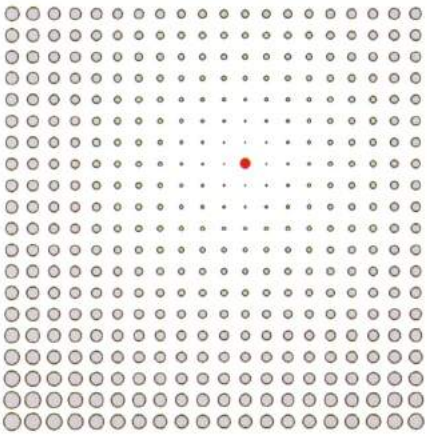


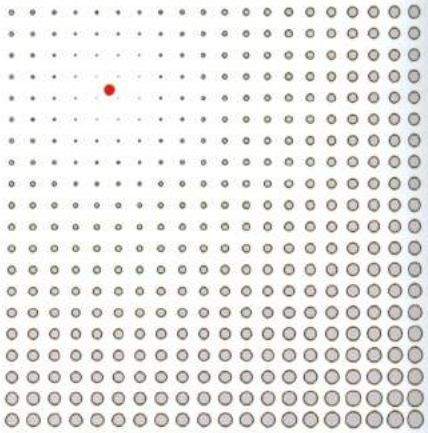
The component ***Circle*** (Primitive > Curve) generates a circle with the center at point {0,0,0} and a radius of 1. The defined circle is two dimensionally arrayed using *Move* and *Series* in conjunction, creating a grid of circles.



A point set from Rhino is used as the attractor mechanism by measuring the distance between the attractor-point and the centroid of each circle returning a list of numeric values. To scale the circles a scaling factor between 0 and 1 is required. The *Remap* component is used to remap the actual

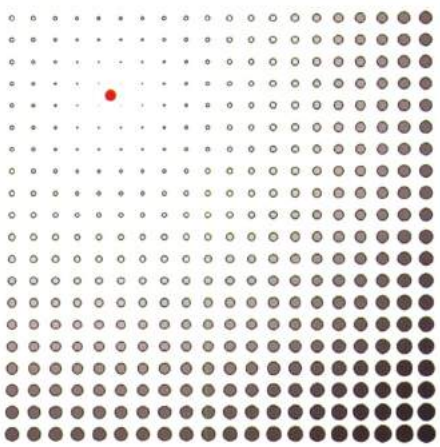
distances (between the attractor point and each centroid) proportionally to a new domain between (0,1]. A scaling factor of zero would yield null results; consequently a value close to zero is used.

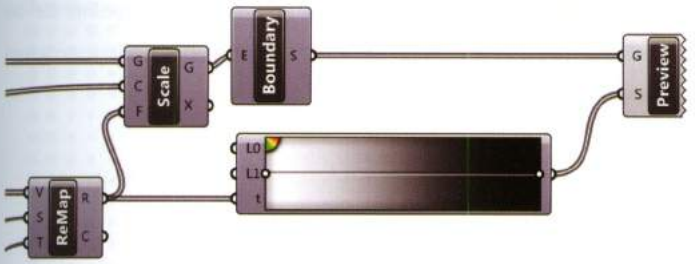




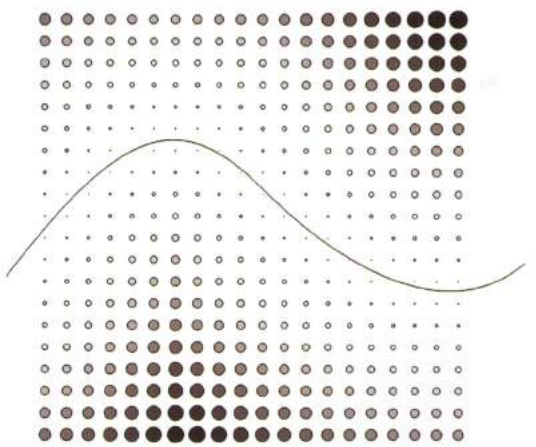
The component ***Scale*** (Transform > Euclidean), scales geometry (G) according to a center of scaling (C) and a scale-factor (F). The remapped domain (0,1] is used as the scaling factors. As the attractor points position moves, the respective distances change and the scaling associatively updates.

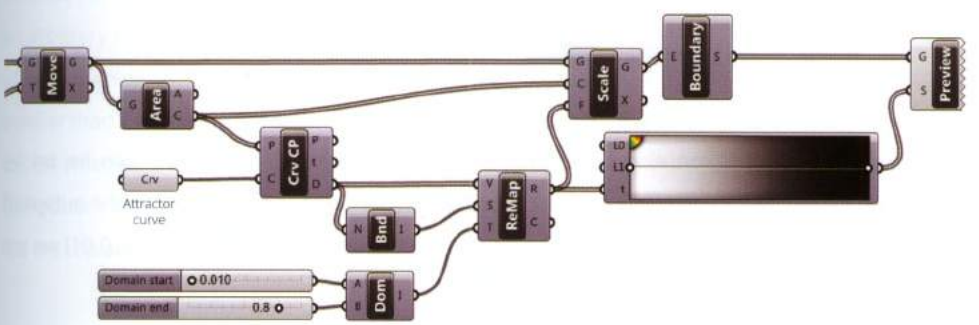
The component ***Boundary*** (Surface > Freeform) creates planar surfaces from closed curves. The component ***Gradient*** (Params > Input) can be used to display the respective scaling factors graphically by a color gradation. To change the *Gradient* component's color preset: right-click on the component, select preset from the sub-menu, and specify a color gradient.



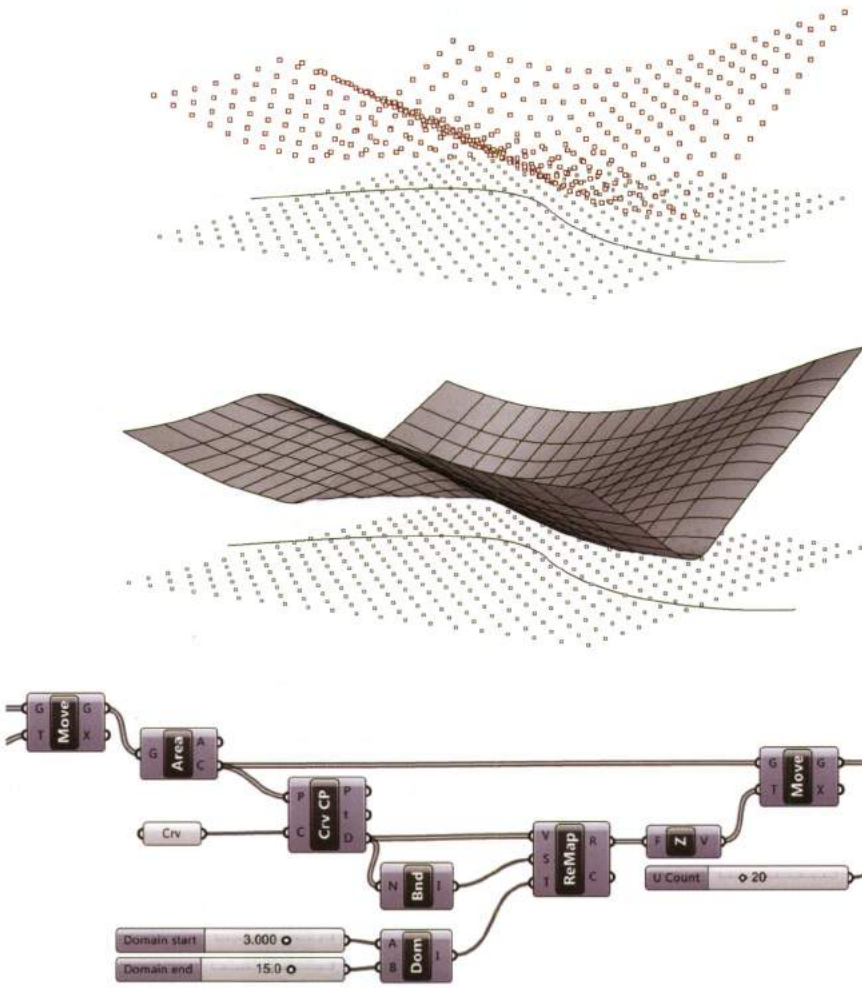


A curve can also be used as an attractor by means of the component ***Curve Closest Point*** (Curve > Analysis). The *Curve Closest Point* component measures the distance (D) between each circle's center and the closest point on the curve. The scaling factors can be visualized by using the *Gradient* component.

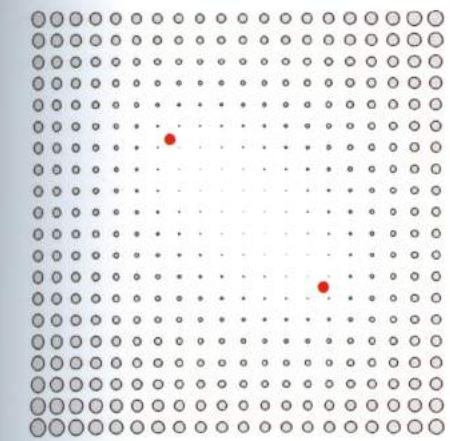




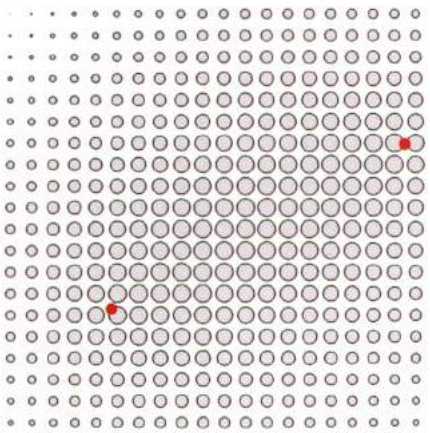
The remapped values can also be used to move the centroid points. For instance, the points can be moved in the z-direction. Changing the remapped domain to [3,15] emphasis the translation when scalar multiplication is carried out within the *Unit Z* component.



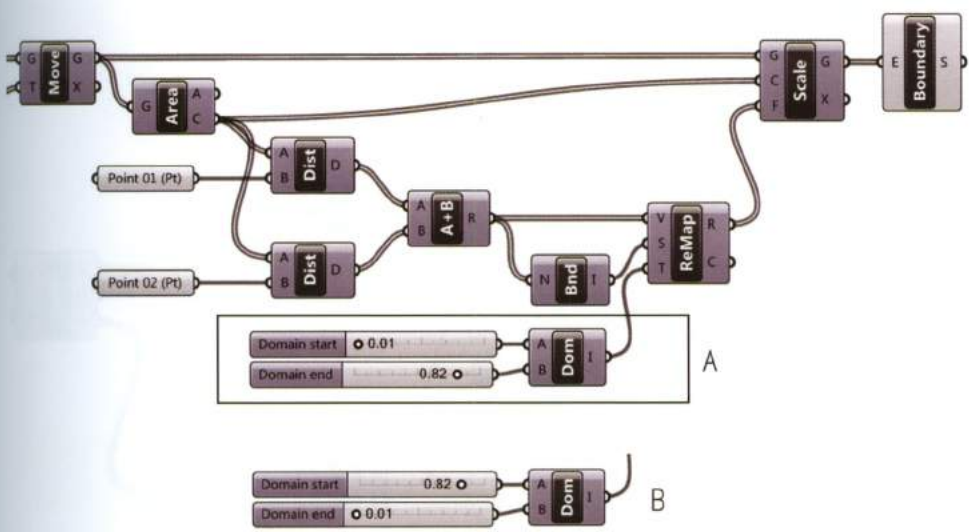
Multiple attractor points can also be defined. For example, two attractor-points can be set by summing their relative distances to each circles centroid. The list to remap (V) will be the output of the component *Addition* (Maths > Operators). If we invert the domain's extremes [0.82, 0.01] we get an opposite scale effect (image B).



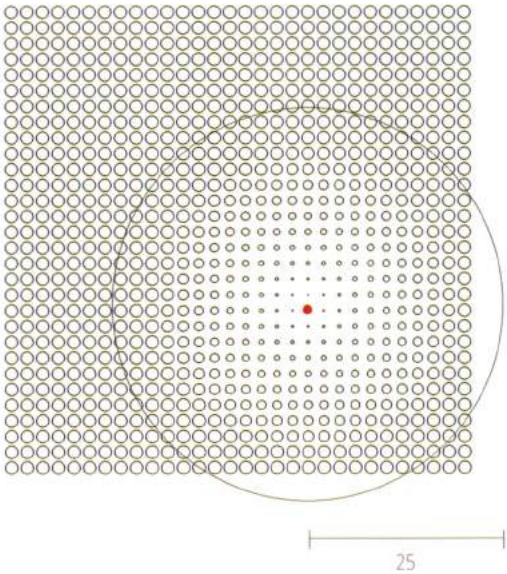
A

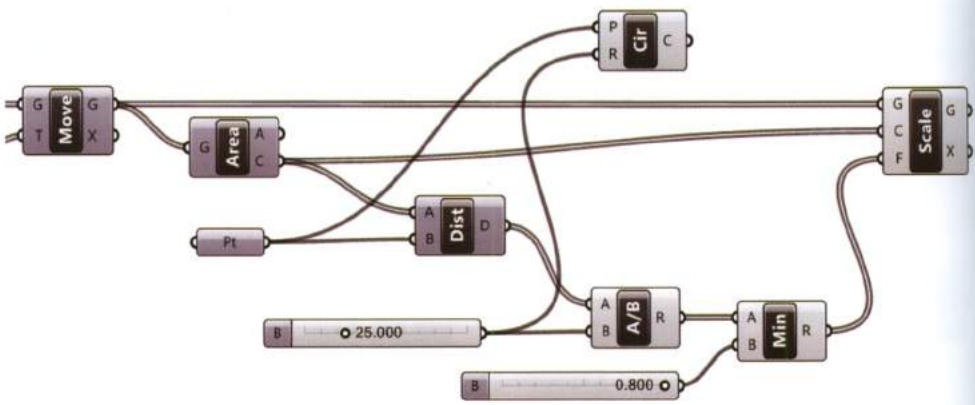


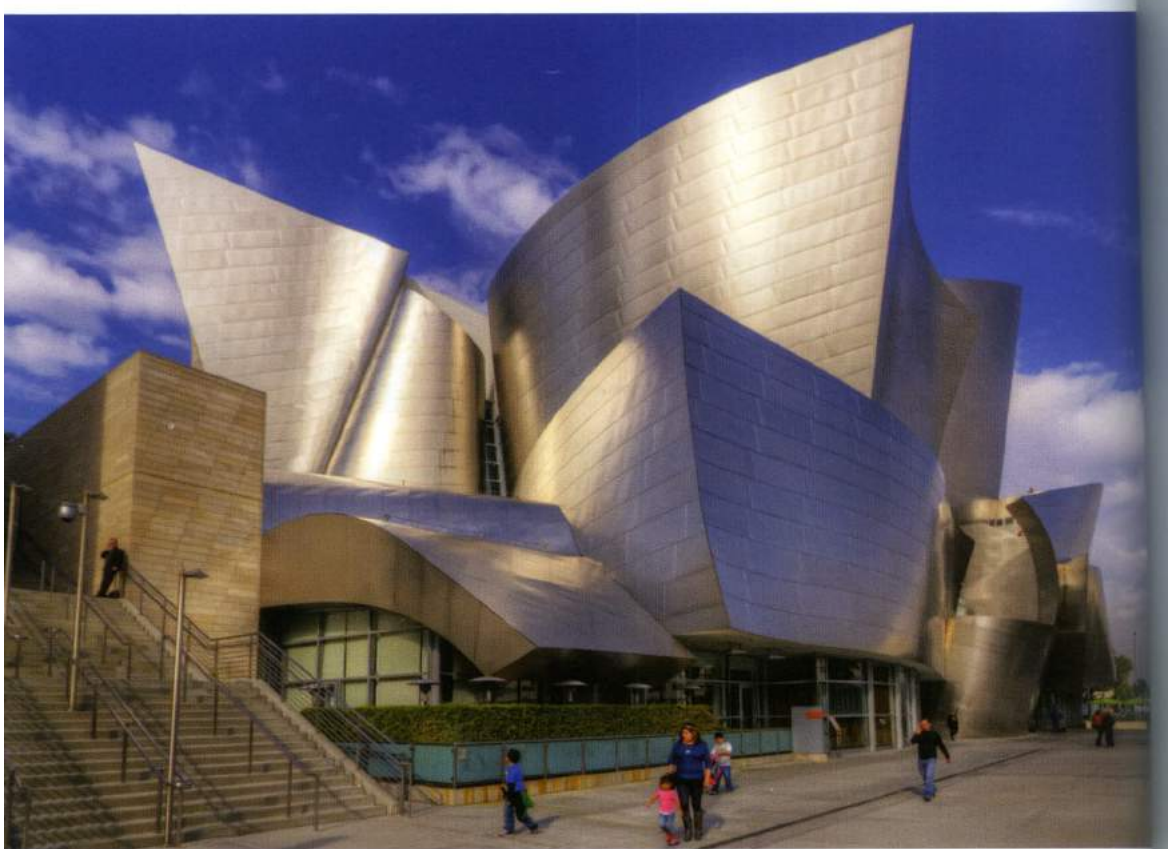
B



To **limit** the effect of an attractor the distance between the point and the centroid can be divided by an arbitrary number, which becomes the *operating range* of the attractor. The division returns a list of values which are connected to the F-input of the *Scale* component. The scaling factors must be smaller than 1 to reduce the original geometry. The component ***Minimum*** (Maths > Util) can be used to test whether a list of numbers are larger than a comparison number (B). Values that satisfy the component are replaced with the minimum value (B). For instance, distances that are greater than 0.8 are replaced with 0.8 as a scale factor.


25


Move
G
T
G
X
Area
G
A
C
Cir
P
R
C
Pt
Dist
A
B
D
A/B
A
B
R
25.000
B
Min
A
B
R
0.800
B
Scale
G
C
F
G
X



Frank Owen Gehry, Walt Disney Concert Hall building.
Picture by Pedro Szekely.

3_control

curves and surfaces in Grasshopper

> "...I was looking for a way to express feeling in three dimensional objects".
>
> Frank Gehry

Prior to the advent of digital tools, formal investigation was limited to traditional drawing techniques relying on geometric primitives and basic mathematical models. Emerging tools and fabrication techniques have changed the way designers conceive of architecture, opening new formal trajectories. This paradigm-shift from pencils, squares and compasses to computer-based tools has happened over the last three decades.

The computer or more specifically, 3D modeling software has pushed the boundaries of formal exploration by digitizing manually iterated design. The next epochal transition – Algorithmic Modeling – requires a robust knowledge of the principles behind form generation; specifically an understanding of NURBS geometries.

3.1 NURBS curves

NURBS or ***N**on-**U**niform **R**ational **B**-**S**plines* are *mathematical representations of 3D geometry that can accurately describe any shape from a simple 2D line, circle, arc, or curve to the most complex 3D organic free-form surface or solid.*

A NURBS curve is a mathematical curve defined by its degree, a set of weighted control points, a knots vector and an evaluation rule.

- **Degree**: a positive, integer number. For example, NURBS-lines and NURBS-polylines are degree 1, NURBS-circles are degree 2, and most free-form curves are degree 3 or 5. The value equal to degree+1 is called the ***order***.
- **Control-points**: A NURBS curve is conditioned by the position and ***weight*** of control points Weights regulate the attraction between the control-points and the curve; a weight >1 attracts the curve and a weight between 0 and 1 repels the curve. The number of control points cannot be smaller than order (degree+1).
- **Knots:** are a list of (degree+N-1) numbers, that control the smoothness of a curve.
- **Evaluation rule**: is a formula that inputs: degree, control points and knots and outputs a point location.

NURBS curves can be drawn by positioning control-points in three dimensional space or on a 2 dimensional plane. The image below is a planar NURBS-curve with a degree of 3 controlled by the position of the 7 control-points.

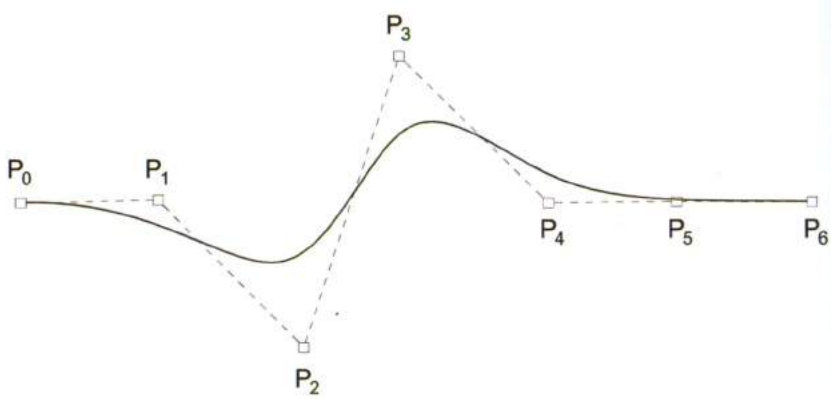


FIGURE 3.1
A NURBS curve with 7 control-points.

If the curve's degree changes different curves will be generated. A curve with the degree 1 coincides with the control-polygon.

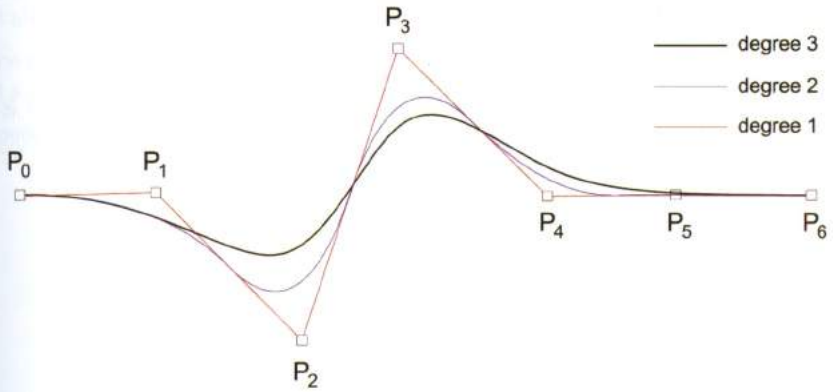


FIGURE 3.2

NURBS curves with 7 control points and different degree.

The image below shows how the *weight* of control-points affects a NURBS curve.

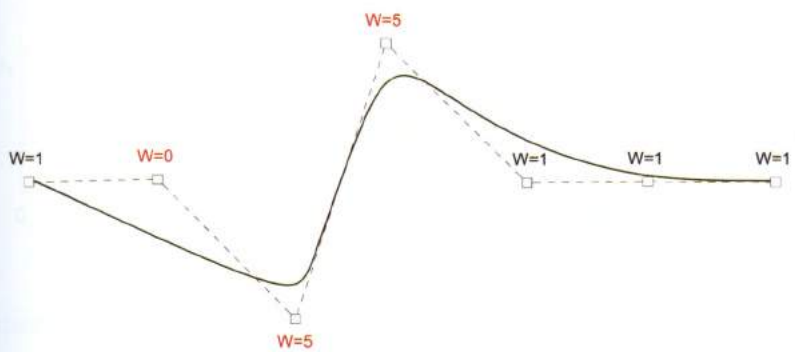


FIGURE 3.3

A control-point with a weight > 1 attracts the curve, on the contrary a control-point with a weight between 0 and 1 push off the curve.

3.2 Parametric representation of a curve

The Rhino environment is based on a **World Coordinate System (WCS)** and points on a NURBS curve are defined by a triplet of coordinates (x_i, y_i, z_i). If z=0 for each point, the curve is a *planar curve* and is defined by the coordinates (x_i, y_i).

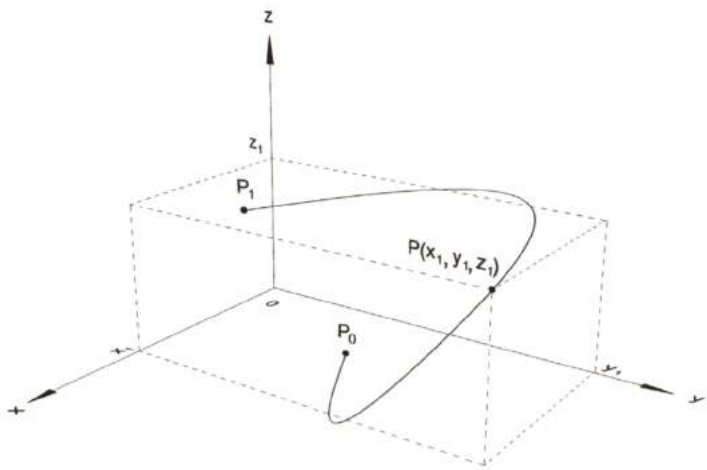
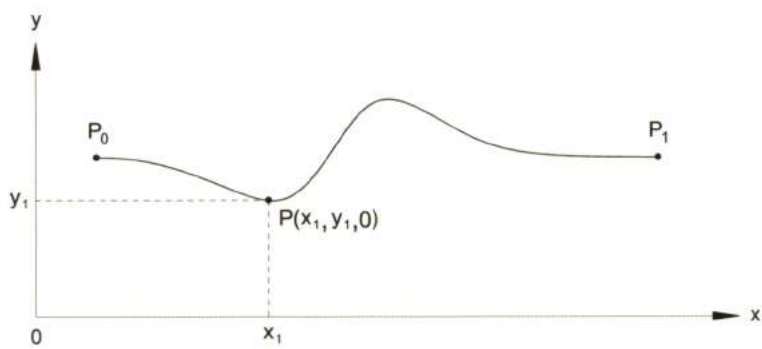
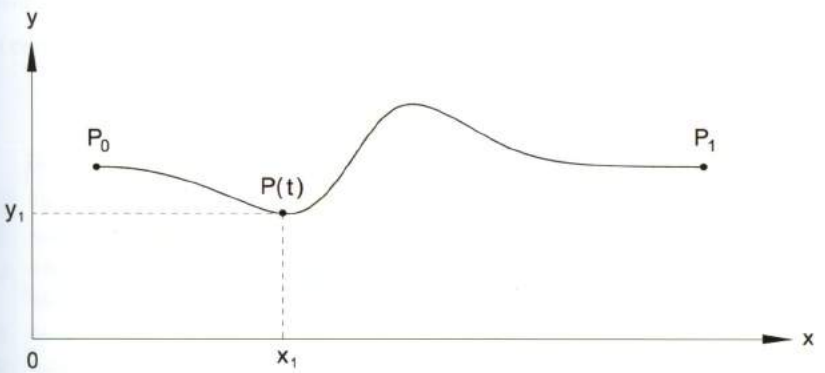


FIGURE 3.4
Each point of a NURBS curve is defined by a triplet of coordinates.

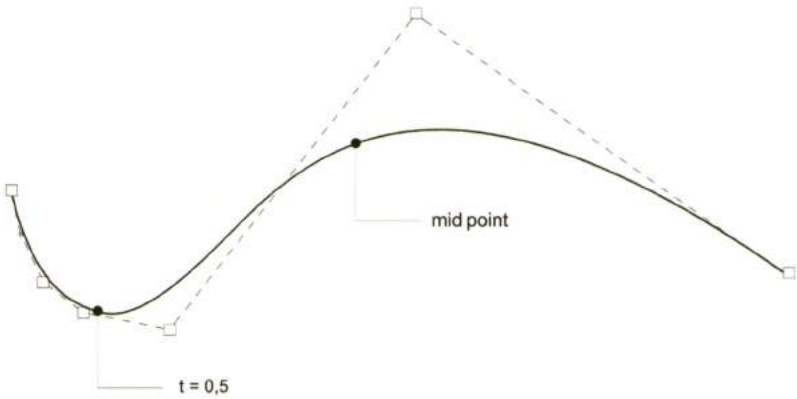


Another way to find points on a curve is based on the **Parametric representation**. The coordinates of an arbitrary point P are expressed as function of a variable ***t*** that ranges between 0 and 1. For t=0 and t=1 we get the curve's end points and for *t* ranging between 0 and 1 we describe the whole curve.

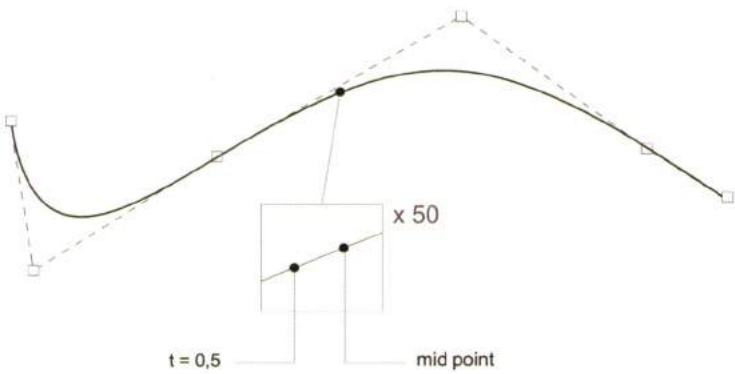
We can also say that a curve is *parameterized* between 0 and 1 or, in alternative, that its **domain** is [0,1]. The parametric representation can be considered as a **Local Coordinate System (LCS)**. Roughly speaking, it is a system set "on" the curve which have great advantages since it requires just one parameter to identify a point, which is still defined in the World Coordinate System (see images below).



It's important to point out that *t* doesn't measure distances. We can imagine *t* as the *time* that a "particle" takes to go from t=0 to the instant position P(t). This time is affected by the position of control-points and, in particular, the motion of the "particle" slows down when it passes through a concentration of control points (see following image). For this reason t=0,5 is not the parameter that specifies the curve's mid point.



Even if the curve is *rebuilt* using Rhino, resulting in a uniform redistribution of control points, the mid point is not coincident with the point corresponding to t=0.5.

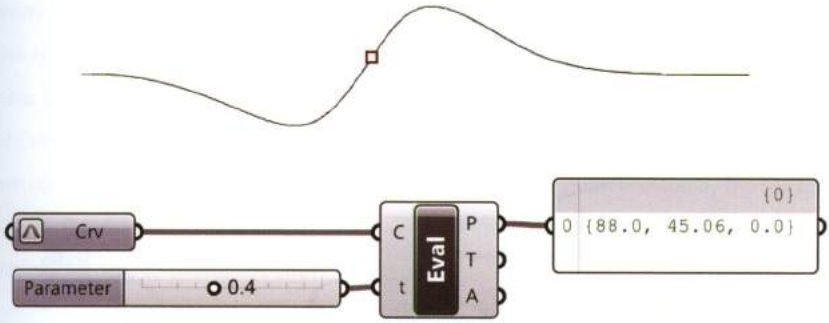


3.3 Analysis of curves in Grasshopper

The following paragraphs will introduce components, logics and strategies to control NURBS curves using *Parametric Representation.*

3.3.1 Finding points on a curve: *Evaluate Curve* component

Parametric Representation enables users to find points along a curve. The component *Evaluate Curve* (Curve > Analysis) requires a curve (C) to analyze and a parameter (t) on the curve domain to evaluate, which can be defined by a *Number Slider*.



The *Parametric Representation* assumes that a curve is parameterized between 0 and 1. However, curves in Rhino typically have different domains. The domain of a curve can be set to [0,1] by reparameterizing the curve. The ***Reparameterize*** option is available by right-clicking on the curve component and selecting *Reparameterize* from the contextual menu. If multiple curves are set with different domains, their local domain will be set to [0,1] by selecting the *Reparameterize* option.

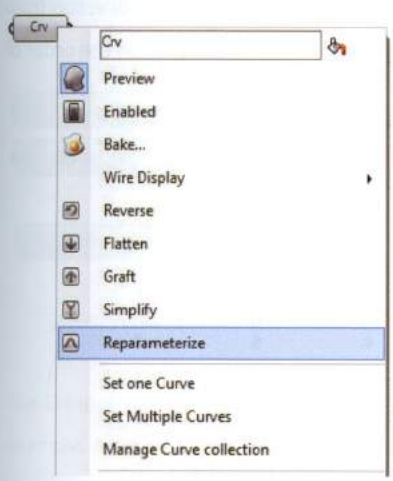
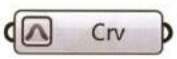
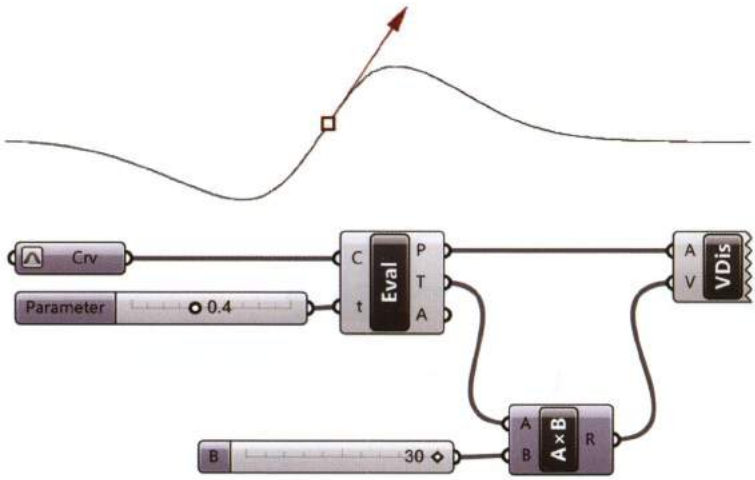


FIGURE 3.5
The Reparameterize option is available within the context menu of the Curve component.

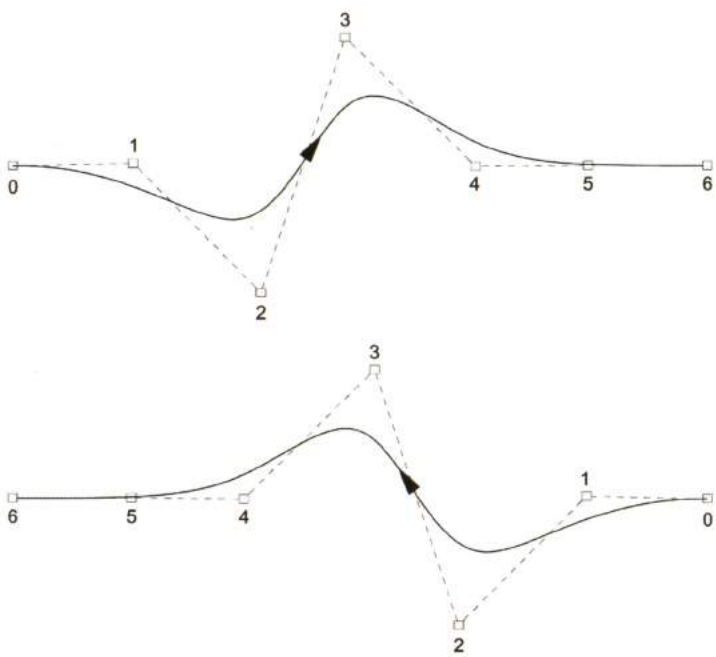


The *Evaluate Curve* output (P) expresses the points location in the World Coordinate System, the T-output is the tangent vector at *t*. The *Vector Display* component can be used to visualize the tangent vector (T). Tangent Vector (T) is a unit vector that can be amplified by scalar multiplication using the *Multiplication* component.

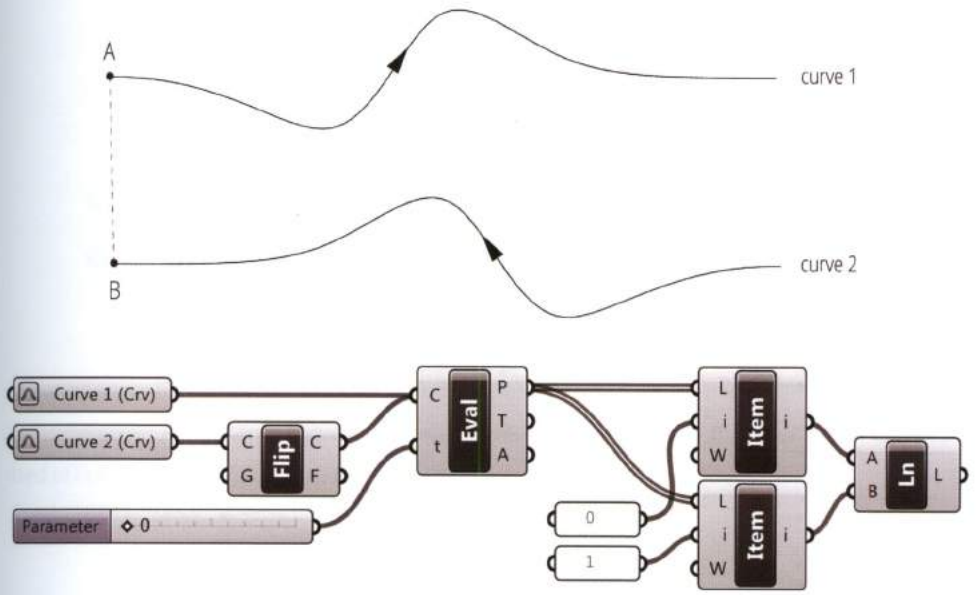


3.3.2 Inverting direction: *Flip Curve* component

Parameterized curves are described by a start point (t=0), an end point (t=1) and a *direction* from start point to the end point. The *direction* depends on how the curve was drawn, i.e. on the order of control points that define the curve.

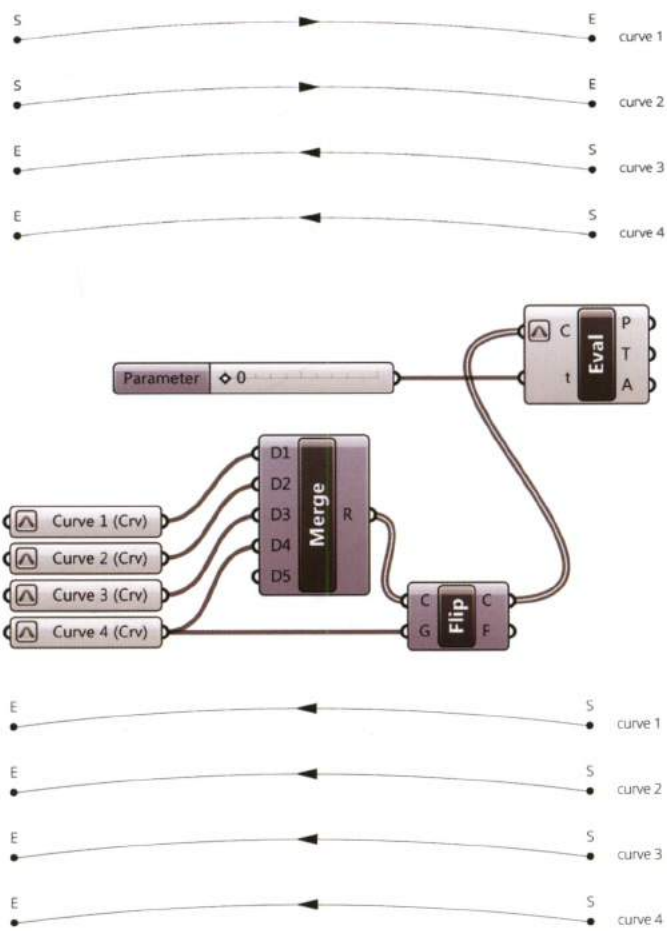


Reparameterizing a curve does not change the direction. The direction of a curve can be changed using the component *Flip Curve* (Curve > Util). The following image displays 2 curves with opposite directions: Curve 1 has left-to-right direction while Curve 2 has right-to-left direction. A is the start-point of Curve 1 while B is the end-point of Curve 2. Our aim is to connect the points A and B by a line. If we use an *Evaluate Curve* component to extract points, we must previously flip the direction of either Curve 1 or Curve 2 by *Flip Curve*. In this way, the point B will become the start-point of Curve 2 and we will be able to create the line.



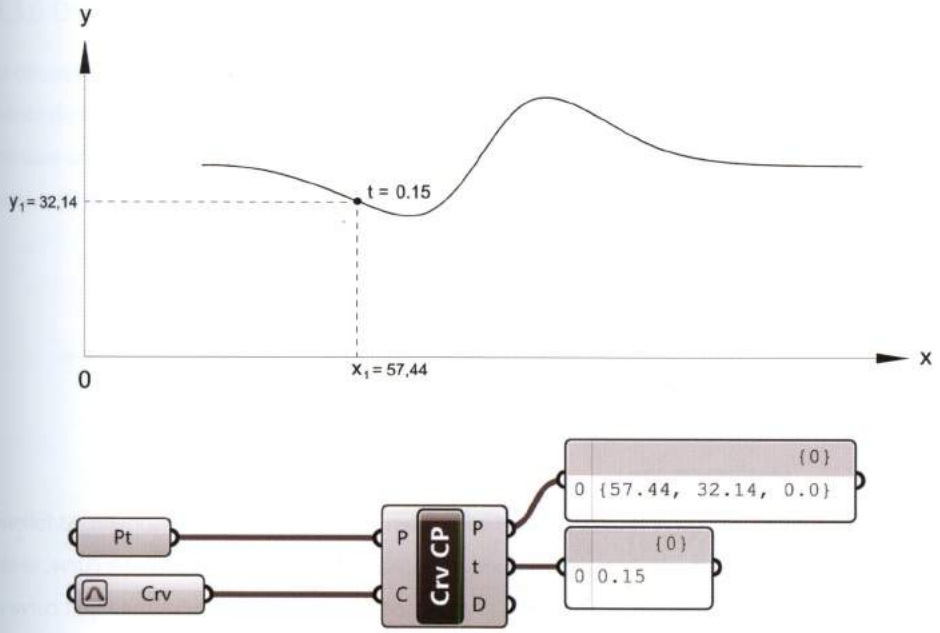
The G-input of *Flip* specifies a reference curve used to unify the direction of set of curves. For example, a set of curves with different directions – curves 1 and 2 are left-right oriented and the remaining are right-left oriented – are set and merged (see following image).

The merged list is connected to the C-input of a *Flip* component and Curve 4 is set as the guide curve in input (G), resulting in a set of right-left oriented curves. The *Evaluate Curve* component can be used to specify points along the curves (it is important to reparameterize the C-input).

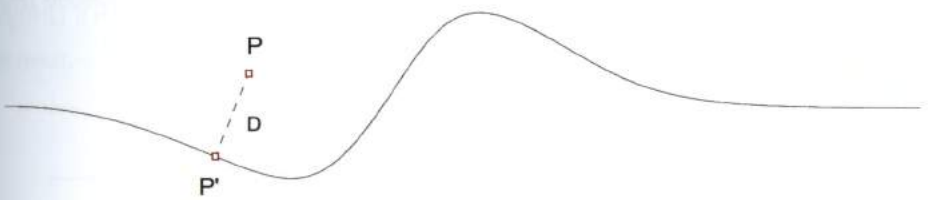


3.3.3 Finding points on a curve: *World to Local* conversion

By default, set points from Rhino are expressed in the World Coordinate System. The component *Curve CP* (Curve > Analysis) is used to convert a point expressed in the WCS to the *t* local parameter of a given curve. The *t*-output of *Curve CP* is the parameter on the curve's domain expressed in the Local Coordinate System.

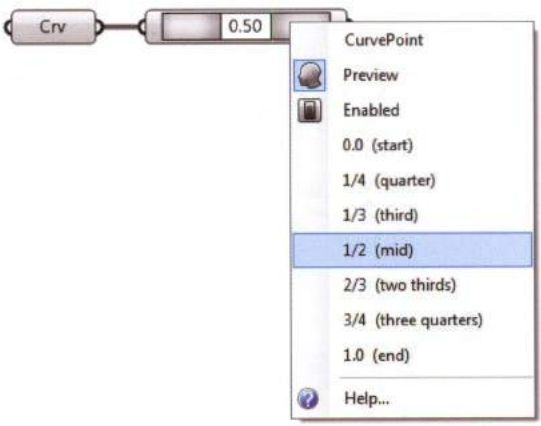


The component *Curve CP* enables users to find the closest point P' on a curve, given an external point P. The displacement distance between points P and P' is provided by the D-output. When *Curve CP* is used to convert from WCS to LCS: P=P' and D=0.



3.3.4 Finding points on a curve: *Point on Curve* component

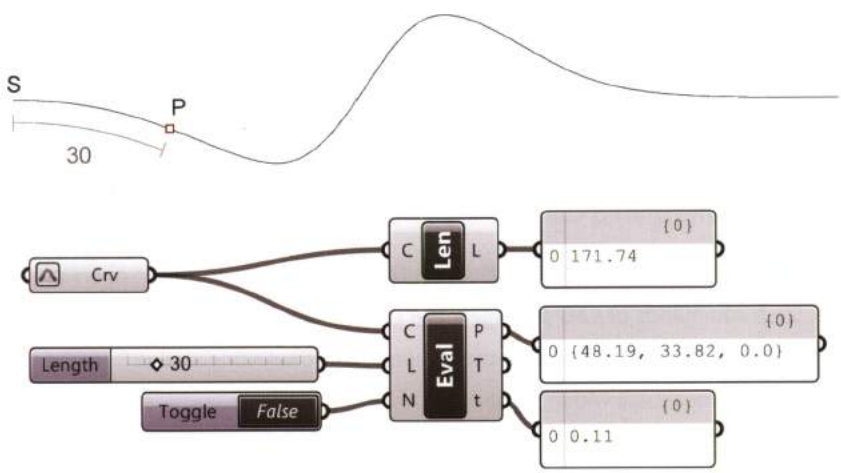
As discussed in chapter 1 the component *Point On Curve* (Curve > Analysis) can operate similarly to the Object Snap commands of CAD packages. The component can be used to find points such as start, quarter, mid etc., by right-clicking on the component and selecting a value from the contextual menu, or by specifying a value within the slider.



The component *Point On Curve* is based on the curve's length not on the curve's domain. As follows, a value of 0.5 expressed by the components slider corresponds to the mid point of a curve, unlike the *Evaluate Curve* component. The *Point On Curve* component does not require the input curve to be reparameterized.

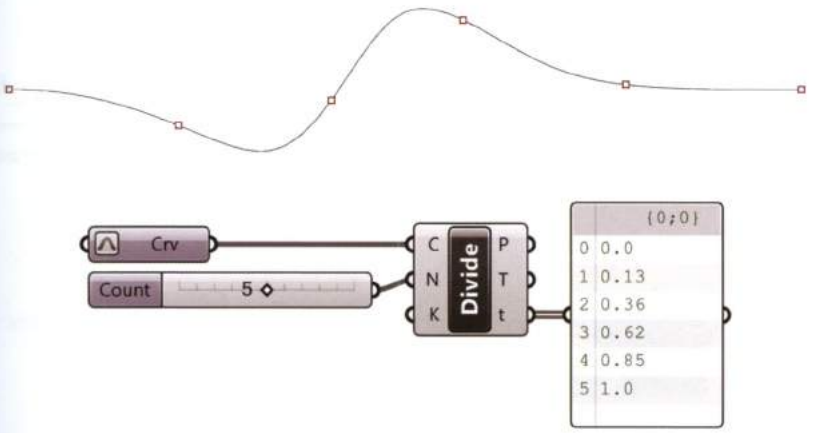
3.3.5 Finding points on a curve: *Evaluate Lenght* component

The component *Evaluate Length* (Curve > Analysis), finds a point (P) a distance measured as an arc-length (L) from the start point t=0. The input (L) is a number between 0 and the curve's length; which can be calculated using the component *Curve Length* (Curve > Analysis). The *Evaluate Length* component outputs coordinates in the WCS (P-output) as well as in the LCS (t-output).



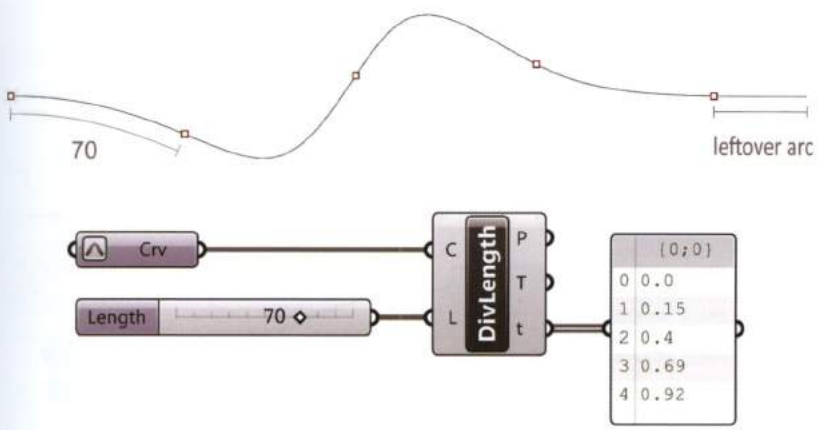
3.3.6 Dividing a curve: *Divide Curve* component

As discussed in chapter 1 (1.7.3), the component *Divide Curve* (Curve > Division) generates a set of points (P) by dividing a curve into (N) equal length arcs. As a result, N+1 points are generated in the case of open curves, and N in the case of closed curves. The component calculates the tangent vectors (T) at division points as well as the LCS (t) parameters.



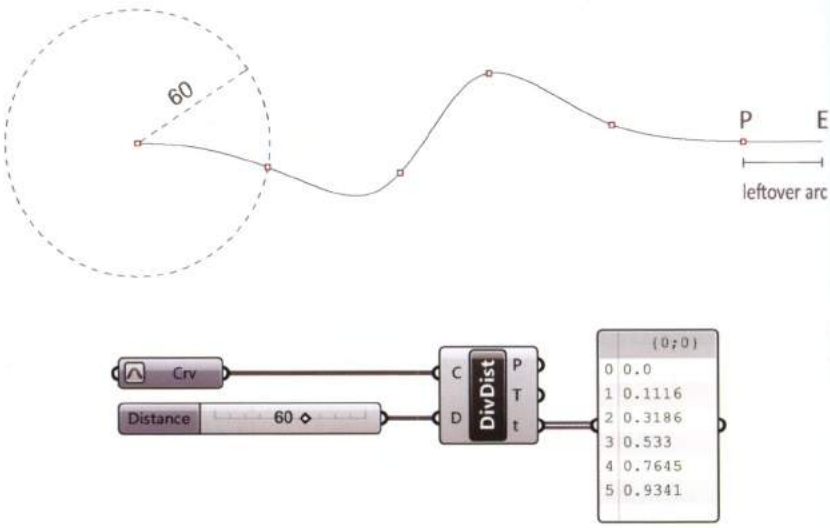
3.3.7 Dividing a curve: *Divide Length* component

The component *Divide Length* (Curve > Division) divides a curve into segments specified by the arc-length (L). If the curve's length is not a multiple of (L), a "leftover-arc" with a length different from (L) will result.



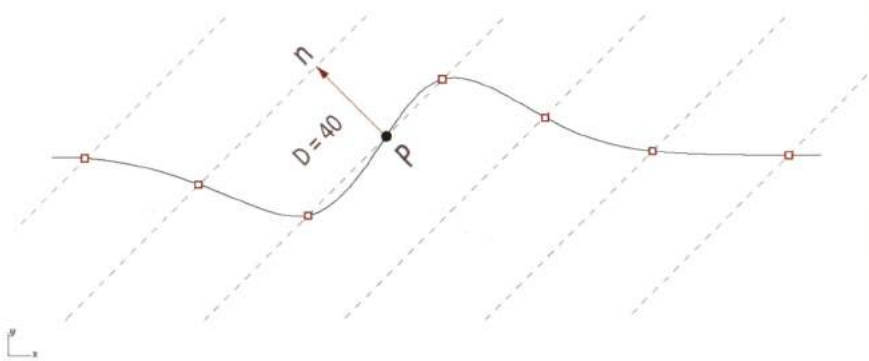
3.3.8 Dividing a curve: *Divide Distance* component

The component *Divide Distance* (Curve > Division), divides a curve into segments by calculating sequential intersections of circles and the curve. Depending on the value of the radial distance (D) a "leftover-arc" will result having a distance PE different from (D).



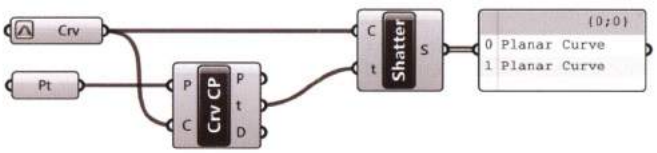
3.3.9 Dividing a curve: *Contour* component

The component *Contour* (Curve > Division), creates a set of curve contours given a curve (C), a start point (P), a vector (N) and a distance (D). The resulting intersections are output as point coordinates in the WCS (P) as well as in the LCS (t).

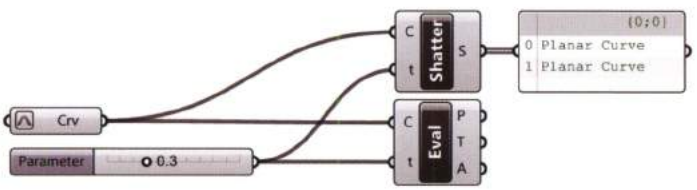


3.3.10 Splitting a curve: *Shatter* component

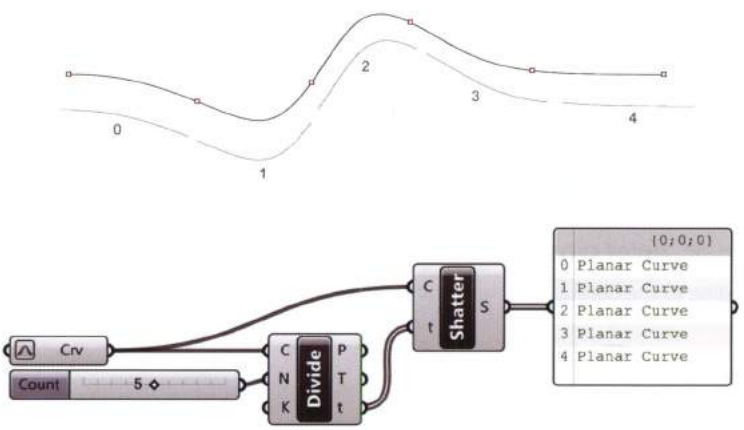
The component *Shatter* (Curve > Division) splits a curve into segments at (t) input parameters outputting segments, instead of points. The parameters to split at (t) can be set by specifying a value within the LCS domain [0,1] or supplied by the output of other components. For example, a curve set from Rhino can be split into two parts using a splitting point (set from Rhino) by connecting the *t*-output of the Curve CP to the *t*-input of the *Shatter* component, resulting in two curves as observed in a panel.



The *t* parameter can also be satisfied by using a slider to specify a value within the numeric domain [0,1].



Similarly, the *t*-output of the *Divide Curve* component can be used as a splitting parameter.



3.4 Notion of Curvature for planar curves

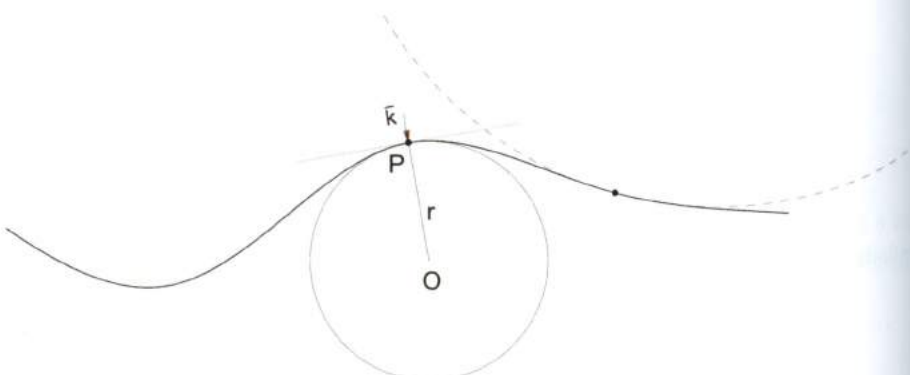
The curvature of a planar curve *c* measured at a point P calculates the deviation of *c* from its tangent line near P. The curvature of a generic curve varies from point to point. In order to mathematically define curvature there are two basic assumptions:

- The curvature of a straight line is zero at every point;
- The curvature of a circle is constant. In particular, curvature is equal to zero when the radius approaches infinity (i.e. a straight line is a circle with an infinite radius).

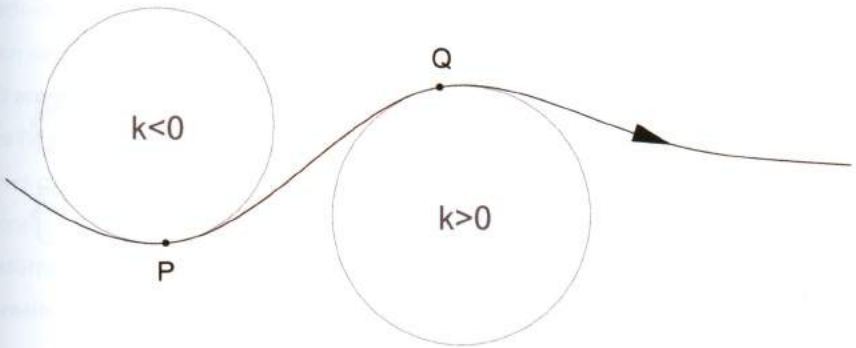
As follows, the curvature (k) can be defined as the reciprocal of the radius (r):

$$\boldsymbol{k = 1/r} \; [1]$$

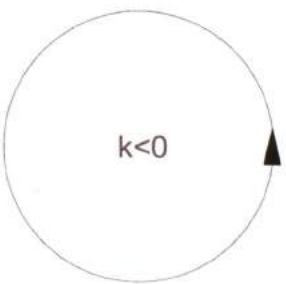
In order to extend this concept to generic curves a geometric construction is used: the *osculating circle*. Given a point P of a generic curve *c*, the circle which most closely approximates the curve near P is defined as the *osculating circle* at P. The curvature's value *k* of curve *c* at point P is defined as the reciprocal of *osculating circles* radius.



Curvature can also be defined as the vector $\bar{k}$ with direction PO and a magnitude equal to the reciprocal of the *osculating circle's* radius. Since curvature is the quotient of two positive numbers - and by definition the radius is a positive number – the curvature of a planar curve is always a positive number. However, ***signed curvature*** convention denotes a positive sign if the *osculating circle* lies to the left according to the default direction and has negative sign if the *osculating circle* lies to the right of the curve.

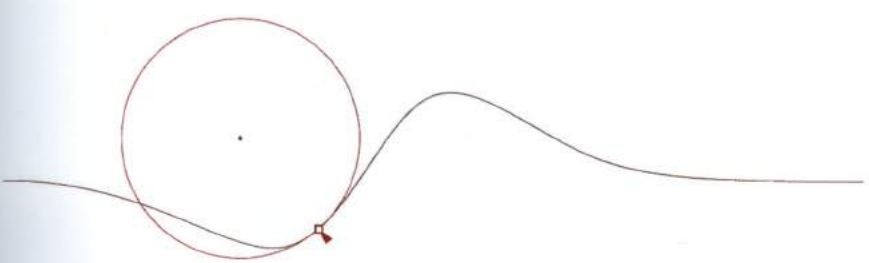


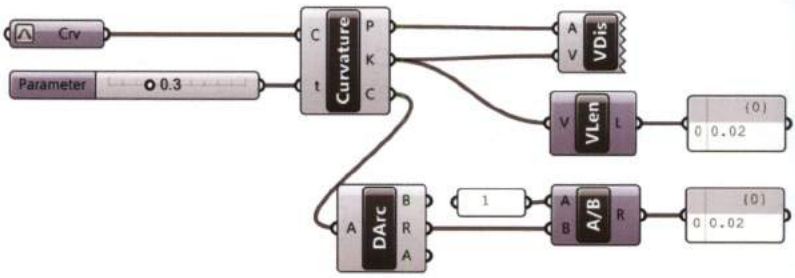
According to the default counter clockwise direction, the circle has a negative curvature sign.



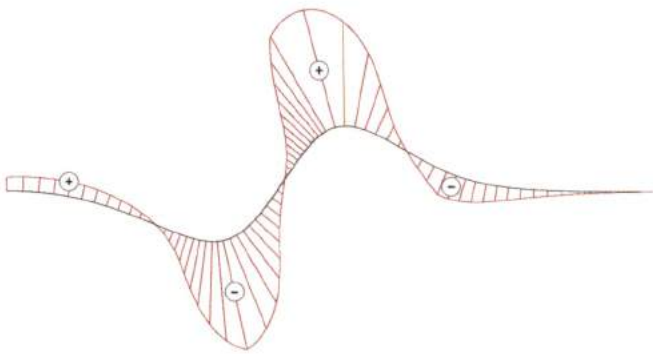
The component *Curvature* (Curve > Analysis) graphically displays the curvature vector $\bar{k}$ and the *osculating circle* at a generic point P expressed in the Local Coordinate System (LCS).

The curve's curvature value can be calculated in one of two ways: as the reciprocal of the radius of *osculating circle* (C), which can be extracted using the R-output of the component *Deconstruct Arc-DArc* (Curve > Analysis), or as the magnitude of the vector $\bar{k}$ which can be calculated using the component *Vector Length* (Vector > Vector).



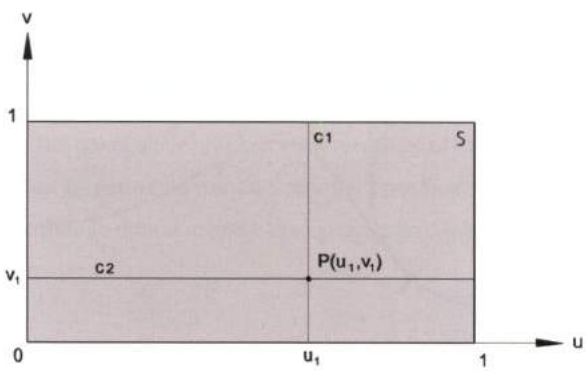


The component ***Curvature Graph*** (Curve > Analysis) visually displays curvature as a graph.



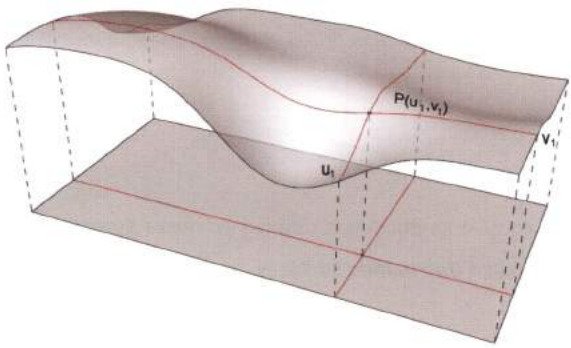
3.5 Parametric representation of a surface

Similar to curves, surfaces can be associated to a LCS. The parameters *u* and *v* (ranging between 0 and 1) operate similarly for surfaces as the parameter *t* operates for curves.

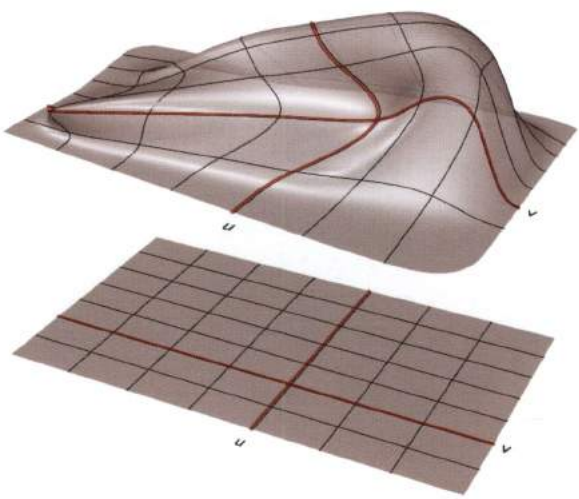


For each value of {u}, points P(u1,v) can be found to constitute the "sectional curve" C1. As follows, for each value of {v} points P(u,v1) can be found to constitute the "sectional curve" C2. Curves C1 and C2 are called ***Isocurves*** or *Isoparametric Curves*. In particular, C1 and C2 are the isocurves of the surface (S) at P(u1,v1). An isocurve is a curve with constant *u* or *v* values on a surface.

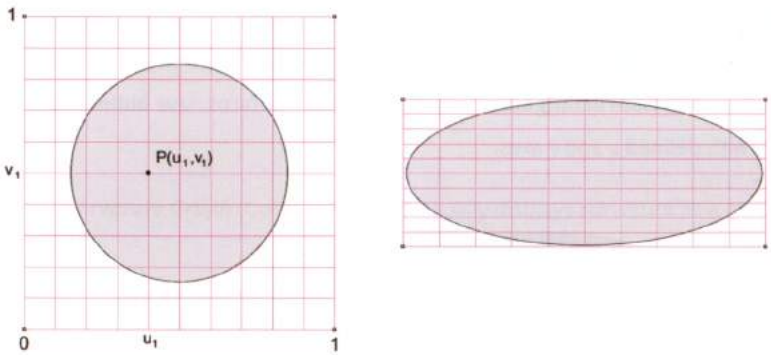
Isocurves create a rectangular grid generalizing the notion of the *Cartesian Grid* on a curved surface. This concept is valid for every freeform surface since a **NURBS-surface can always be imagined as the deformation of a flat-rectangular surface**. Moreover, surfaces have **bidimensional domains** (or **domain²**) with defined *u* and *v* axes.



As shown below, we can get a complex, freeform surface starting from a rectangular flat surface. The rectangular-orthogonal grid of isocurves becomes a rectangular-distorted grid.



Non-rectangular surfaces are however based on rectangular grids. This can be visualized by drawing a surface from a circle or from an ellipse and activating the rectangular grid of control-points. Rhino hides the trimmed area, but the actual domain is always rectangular.



A surface that does not contain its domain is called a ***Trimmed Surface***. Conversely, surfaces that contain their domain are called ***Untrimmed Surfaces***.

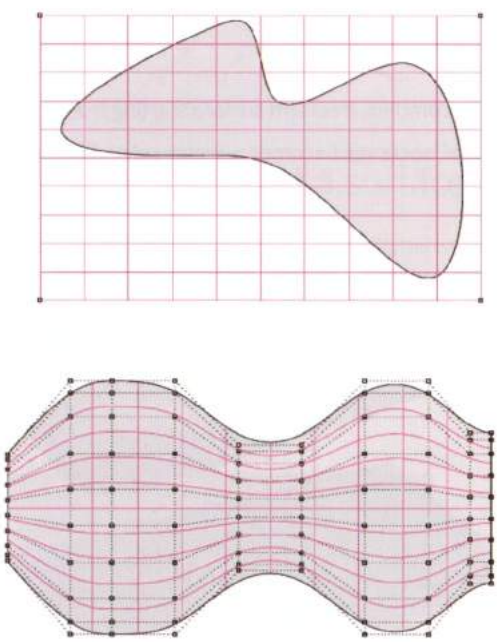
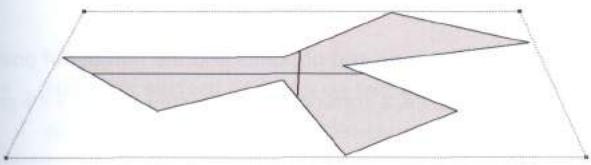


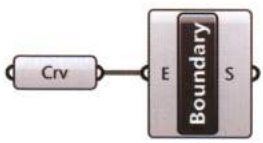
FIGURE 3.6
A trimmed (above) and an untrimmed (below).

3.6 Surface creation

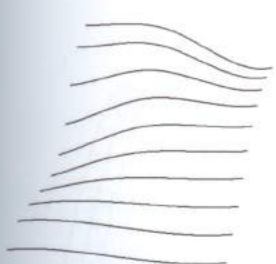
Grasshopper provides several components to create surfaces; they are hosted within the *Freeform* panel of the *Surface* tab:

- **Extrude:** is the easiest way to create a surface. The component *Extrude* (Freeform > Surfaces) requires a profile curve (open or closed) or a surface to extrude according to a vector. Extrude yields a surface with a constant section.
- **Boundary surface**: creates a trimmed or untrimmed **planar** surface from a collection of boundary edge curves. The component *Boundary Surfaces* (Freeform > Surfaces) requires planar and continuous closed curves to result in a surface.





- **Loft**: creates a surface through a set of ordered section curves. The component *Loft* (Freeform > Surfaces) requires at least two curves to generate a surface. The component *Loft Options* (Freeform > Surfaces) controls the method of surface creation, or the default method will operate. *Loft Options* specifies whether a surface is closed, if the seams should to be adjusted, if surfaces should be rebuilt, a refit tolerance, as well as one of six loft types specified by an integer: 0=Normal, 1=Loose, 2=Tight, 3=Straight, 4=Developable, 5=Uniform.

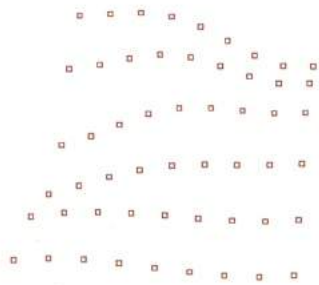


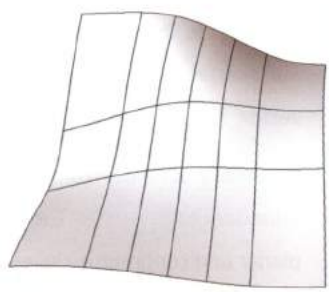
section curves

regular loft

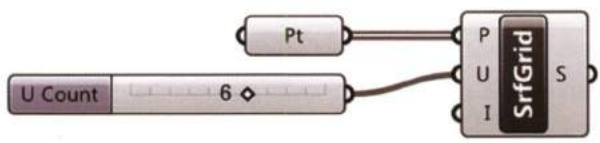
straight loft

- **Surface from points**: creates a surface from a grid of points. The component *Surface from Points* (Freeform > Surfaces) is useful for creating surfaces from other surfaces, or as discussed in 2.3.2 from mathematical functions.

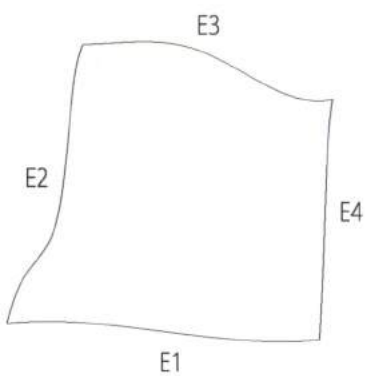


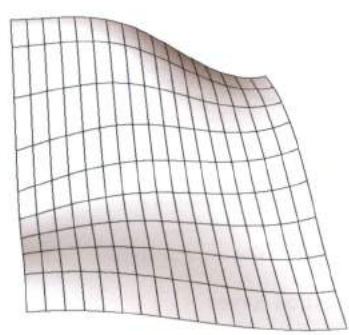


The component *Surface from Points* requires a grid of points and the number of points in the *u* direction to be declared. For instance, a surface can be created from a 9 x 6 grid of points by specifying a U-count of 6.

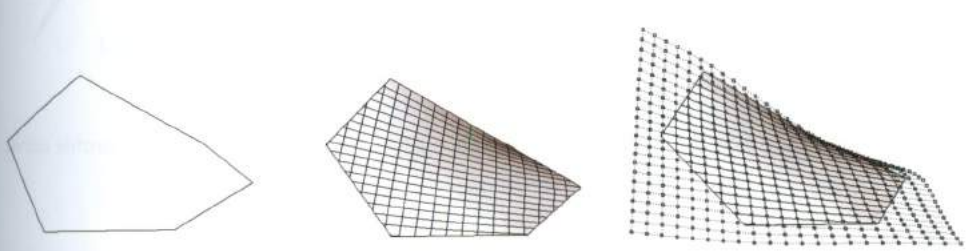


- **Edge surface** creates a surface from two, three or four curves. The component *Edge Surface (Freeform* > Surfaces) interpolates a surface from ordered edge curves.

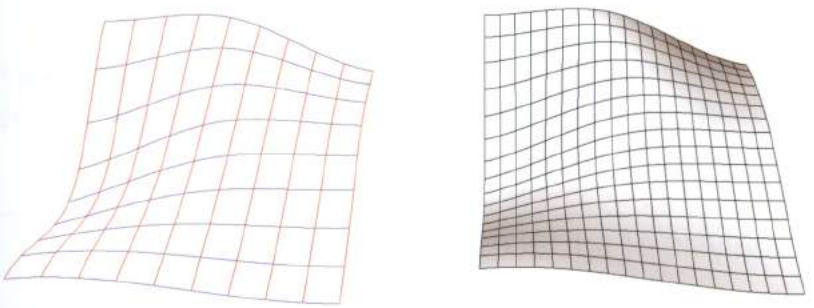




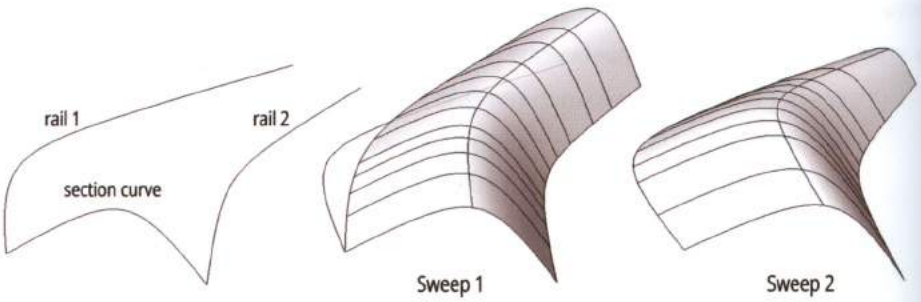
- **Patch:** fits a surface through a set of points, or curves that are open or closed. The component *Patch* (Freeform > Surfaces) generates a series of perpendicular curves, called spans, from input curves which compose a trimmed or untrimmed surface. Span curves are rarely parallel to any edge of the output surface so the *Patch* component is used when it's not possible to perform other methods.



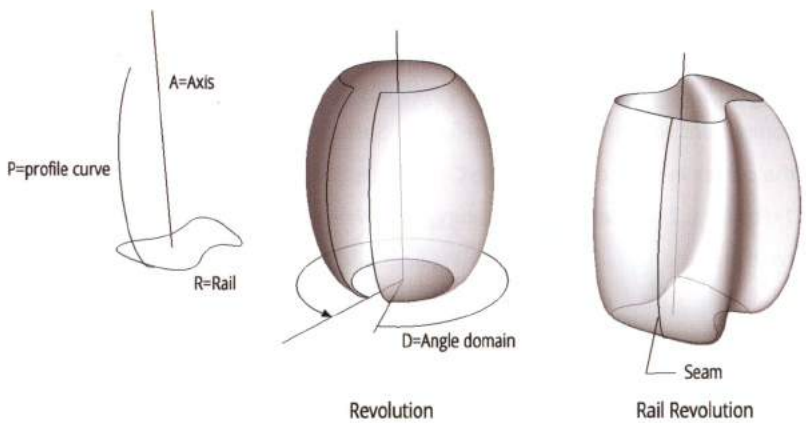
- **Network surface** operates on two sets of ordered curves to create a single surface. The component *Network Surface* (Freeform > Surfaces) requires curves in the *u* direction and the curves in the *v* direction. The C-input specifies the type of surface continuity (0=*Loose*, 1=*Position*, 2=*Tangency*, 3=*Curvature*. The *Network Surface* component allows for more control of the surfaces edges, when compared to the *Loft* component.



- **Sweep 1 and Sweep 2:** generate a surface using a section curve S swept along one or two rail curves R. *Sweep1* (Freeform > Surfaces) should be used with one rail curve and *Sweep2* (Freeform > Surfaces) with two rail curves.



- **Revolution and Rail Revolution:** generate surfaces from the revolution of a profile curve P around an axis or using a sweep rail respectively.



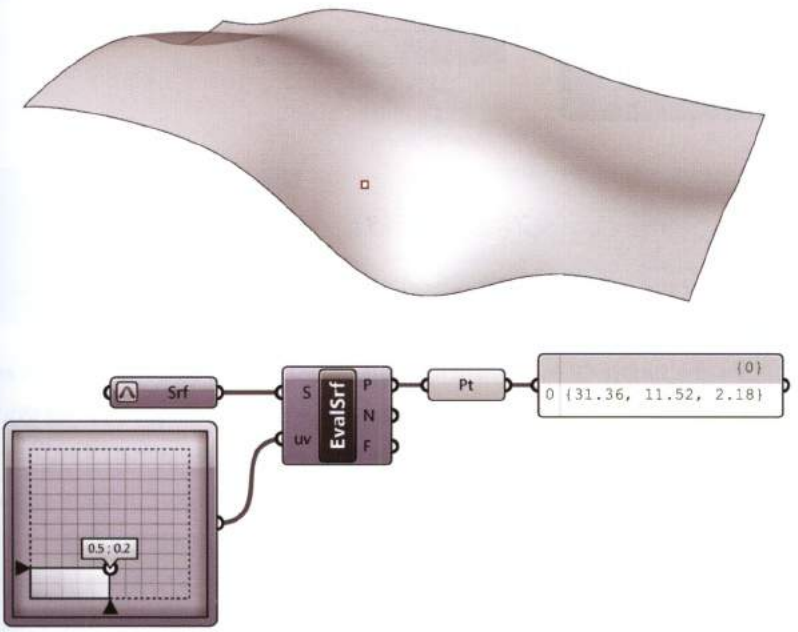
3.7 Analysis of surfaces using Grasshopper

3.7.1 Finding points on a surface: *Evaluate Surface* component

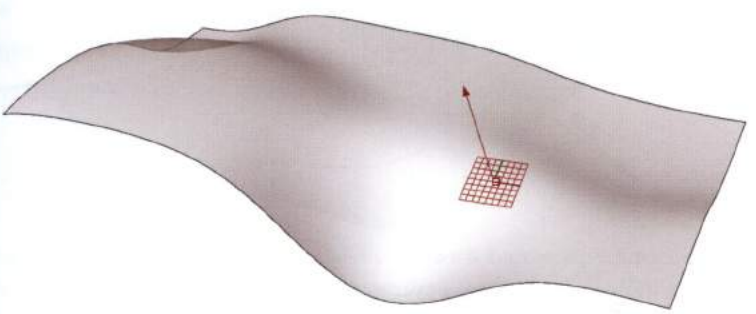
Parametric Representation is used to find points on a surface. The component *Evaluate Surface* (Surface > Analysis) requires a surface (S) to analyze and two LCS coordinates *(u and v)*. The component *MD Slider* (Params > Input) can be used to assign *u* and *v* coordinates.

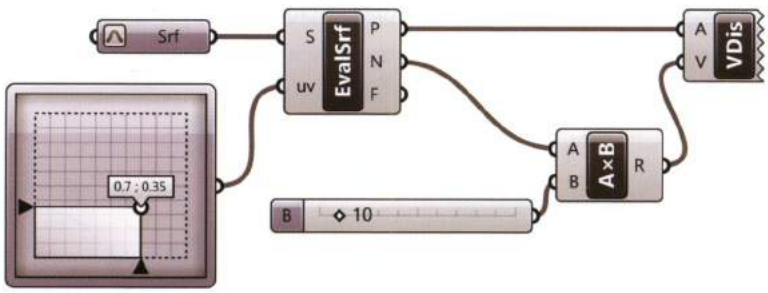
The *MD Slider* (or multi dimensional slider) is a bidimensional extension of the *Number Slider*. By default its values range between 0 and 1 in *u* and *v directions*, assuming the input surface is parameterized between the same interval.

As for curves, the *Surface* component and the S-input of *Evaluate Surface* allow us to reparameterize the input surface by right-clicking the *Surface* component or the S-input of *Evaluate Surface* and selecting *Reparameterize*.



The *Evaluate Surface* component outputs: a point (P) expressed in the WCS, a normal vector (N) at P and a tangent plane (F) at P. The displayed dimension of planes can be set from *Display* > *Preview Plane*.

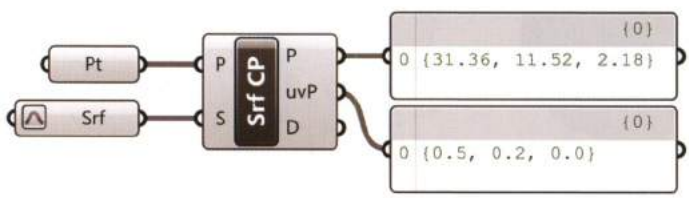


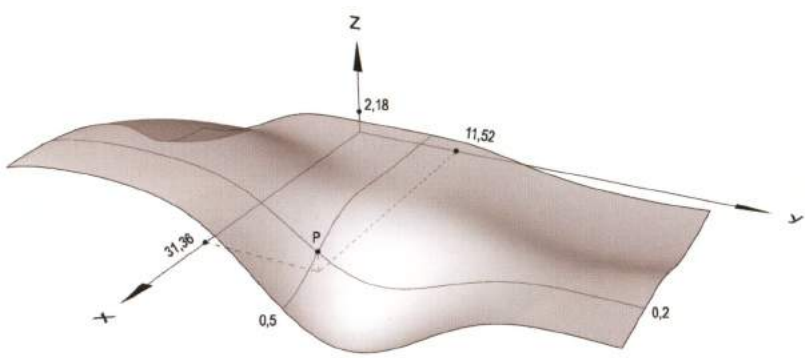


3.7.2 Finding points on a surface: *World to Local* conversion

The component *Surface CP* (analysis > surface) can be used to convert a point P on a surface expressed in the WCS to a point expressed in the LCS.

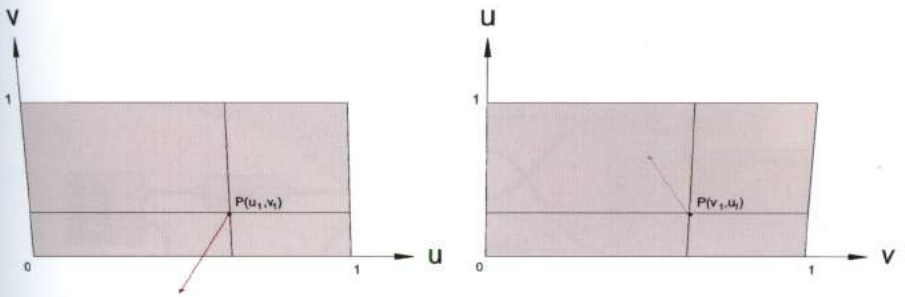
More specifically, the *Surface CP* component finds the closest point P' on a surface, given an external point P. The displacement distance between points P' and P is provided by the output (D). When the component *Surface CP* is used to convert between WCS and LCS: P=P' and D=0.





3.7.3 Inverting direction: *Reverse Surface Direction* component

The component *Surface Direction* (LunchBox > Util) inverts the *uv* direction of a surface. The Tab **Lunch Box**, developed by Nathan Miller[9] is available as a plug-in for Grasshopper and includes utilities for mathematical forms, paneling systems, structures, and workflow.



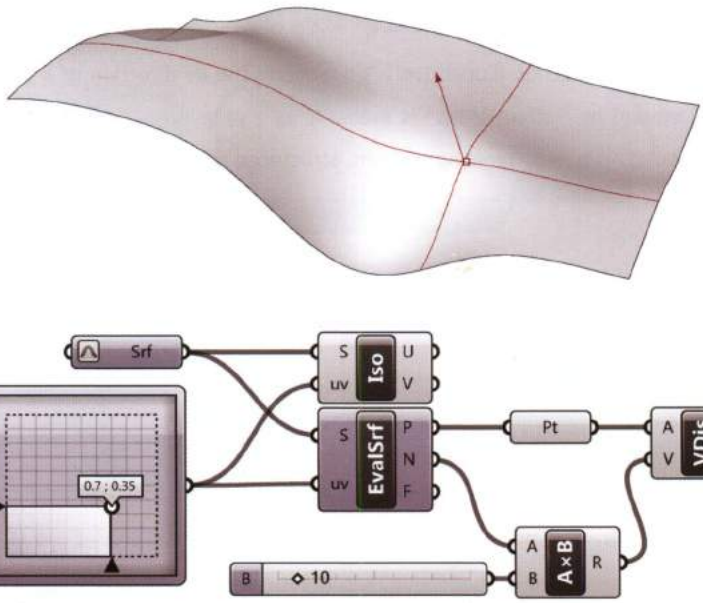
The R-input of *Surface Direction* specifies the reverse option: if R=0 the directions will not be inverted, if R=1 the *u* direction will be inverted, if R=2 the *v* direction will be inverted and if R=3 both *u* and *v* will be inverted. Often it is required to *reparameterize* the RevSrf-output.

3.7.4 Extracting Isocurves: *Isocurve* component

The component ***Isocurve*** (Curve > Spline) extracts the isocurves of a surface at a specific point (P), given its local coordinates (uv). Isocurves U (parallel to *u* direction) and V (parallel to *v* direction) are extracted separately. The U or V isocurves can be visualized independently by connecting a curve component to the U or V output and hiding the *Isocurve* component.

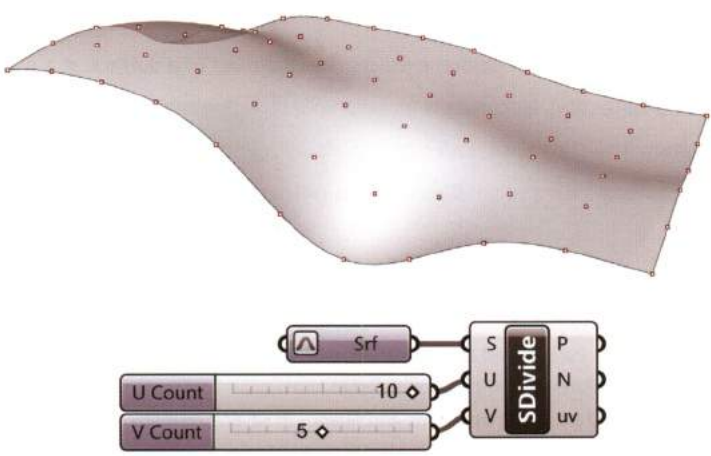
NOTE 9

Nathan Miller is an Associate Partner and Director of Architecture & Engineering Solutions at CASE where he is responsible for leading the efforts on computational design strategy and complex modeling and geometric rationalization. His expertise in 3D modeling programs and fluency in several scripting and programming languages provides leading-edge solutions and capabilities for significant AECO clients. Explorations on computational design are published on his blog: http://www.theprovingground.org/



3.7.5 Dividing a surface: *Divide Surface* component

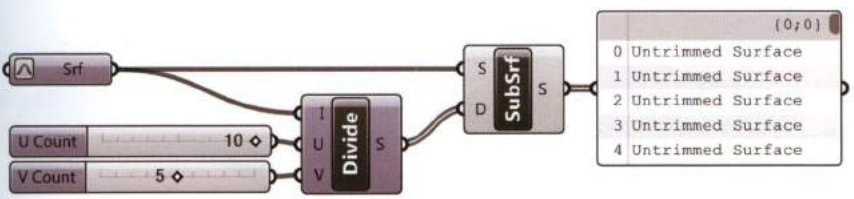
The component *Divide Surface* (Surface > Util) generates a grid of points on a surface. The points (P) are the intersection vertices of a grid of isocurves calculated by dividing the *u* and *v* axes evenly by a positive integer input. The *Divide Surface* component outputs: normal vectors (N) at points (P) as well as their local coordinate (uv).



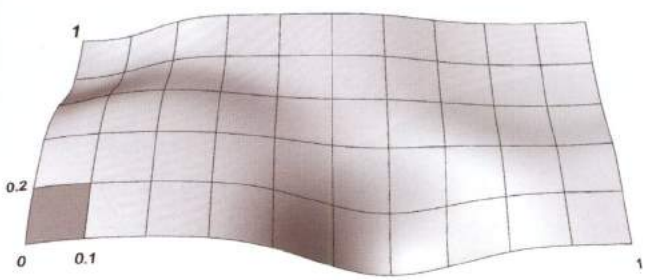
3.7.6 Splitting a surface: *Isotrim* component

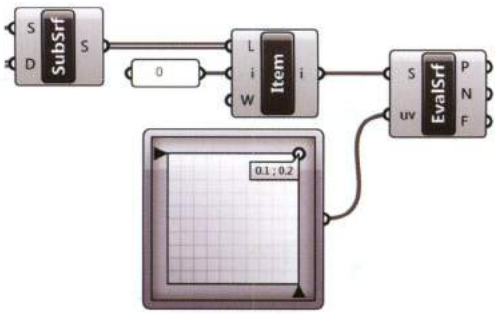
The component *Isotrim-SubSrf* (Surface > Util) splits a surface into a set of contiguous sub-surfaces based on a grid of splitting isocurves. For example, the image below is a surface split into 50 sub-surfaces.

The *Isotrim* component is used in conjunction with the component *Divide Domain²* (Maths > Domain). The *Divide Domain²* component sets the number of divisions in both *u* and *v* directions generating a cutting-grid on the surface for the *Isotrim* component to operate on.

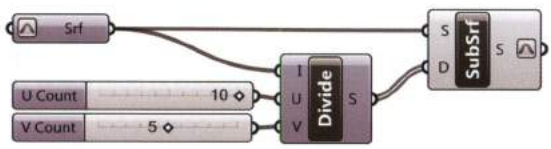


The *Isotrim* component outputs a set of *untrimmed* surfaces parameterized according to the domain of the parent-surface. For example, the domain of the bottom-left sub-surface is [0;0.1] in the *u* direction and [0;0.2] in the *v* direction.

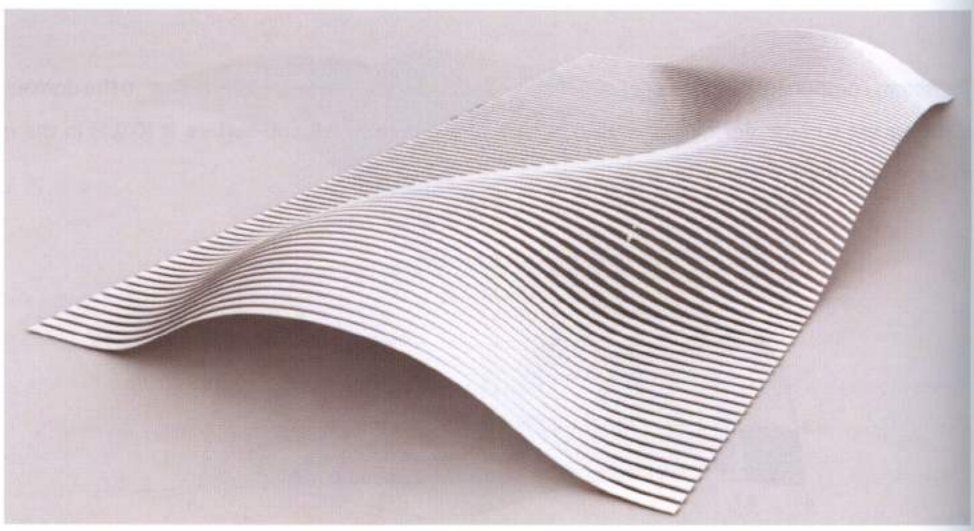


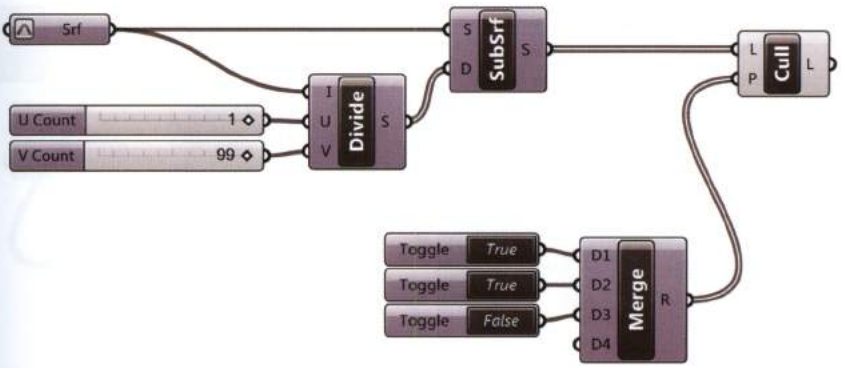


Each sub-surface can be parameterized between 0 and 1, by right-clicking on the S-output of the *Isotrim* component and selecting *Reparameterize* from the context menu.



If 1 is set as the U-input or the V-input of *Divide Domain²* component, a series of contiguous stripes that follows the parent-surface will be generated. Patterns can be created using the component *Cull Pattern* (Sets > Sequence) with a list, for example, of (True, True, False) Boolean inputs.





3.7.7 Strategy: uneven splitting

The cutting-grid generated by *Divide Domain²* is evenly spaced because the *u* and *v* axes are divided into equal lengths by the U and V input values respectively. To generate an unevenly spaced grid the distribution of points within *u* and *v* are required to be adjusted.

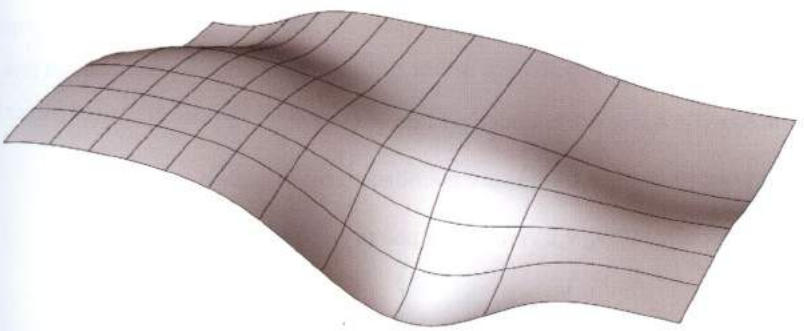
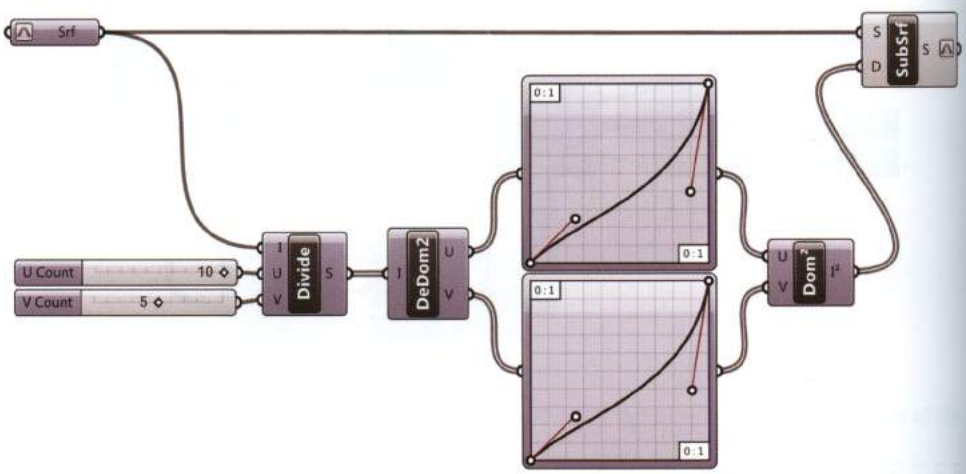


FIGURE 3.7
Uneven splitting of a NURBS surface.

To construct an unevenly spaced grid: first, the bidimensional domain must be divided into its mono-dimensional components U and V using the component *Deconstruct Domain²* (Maths > Domain), then the distribution of points within the two domains (U and V) are changed by the component *Graph Mapper* (Params > Input) set to *Bezier*, by right-clicking on the component and specifying *Bezier* from the context menu; next, the component *Construct Domain²* (Maths > Domain) reassembles the mono-dimensional domains into a bidimensional domain, and finally the surface is split using the *Isotrim* component. Acting on graph's grips will change the grid spacing.



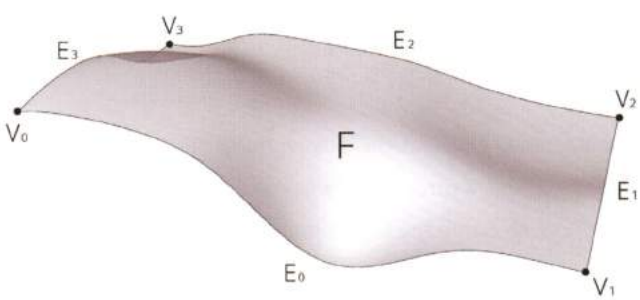
3.7.8 Decomposing a surface: *Deconstruct Brep* component

NURBS-surfaces can be imagined as a geometrical object composed of three geometric entities: ***faces***, ***edges***, and ***vertices***.

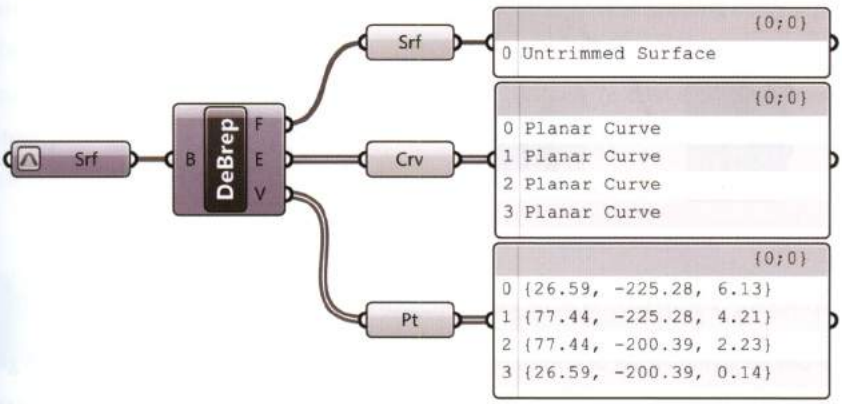
For every surface:

- One *Face* (F_0). A face is a bounded portion of surface;
- Four *Edges* (E_0, E_1, E_2, E_3). Edges are bounded pieces of curves;
- Four *Vertices* (V_0, V_1, V_2, V_3). Vertices are points that lie at corners;

can be extracted.

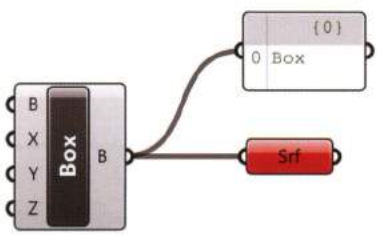


The component *Deconstruct Brep* (Surface > Analysis) is used to extract *faces*, *edges*, or *vertices* from a surface.

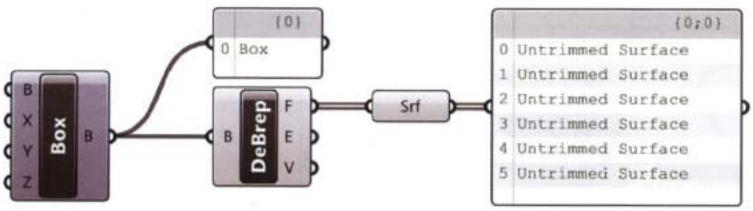


The term **Brep** (or **boundary representation**) describes a way of defining geometric forms using boundaries. Primitives such as Boxes, Cylinders, Cones as well as other geometric solids are defined by boundary representation. Meaning, that a solid or a polysurface is composed of a collection of edges and faces "stitched" together to form a solid object. For instance, a NURBS surface can be considered as a collection of one face.

For example, if the component *Center Box* (Surface > Primitive) is used to define a box with a specified domain and the *Surface* component is connected to the box output (B) the *Surface* component will turn red to display a data mismatch error.

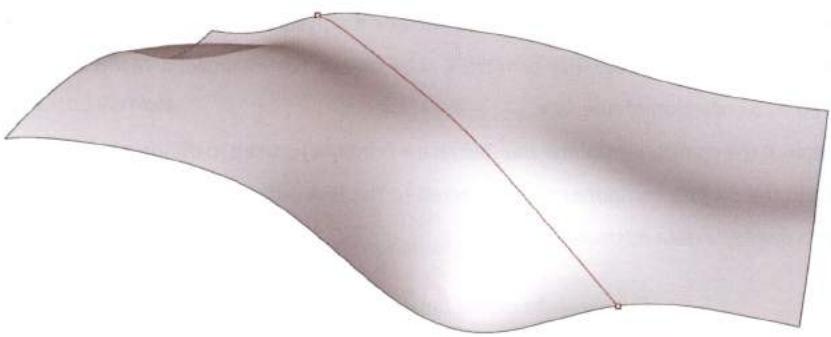


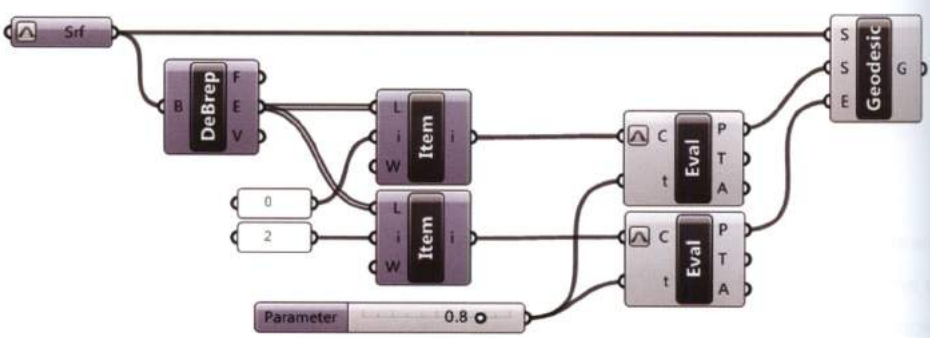
The error occurs because the box is constituted by six surfaces joined together into a Brep. In order to extract surfaces, the box or Brep has to be deconstructed into surfaces using the component *Deconstruct Brep*.



3.7.9 Splitting a surface using generic curves: *Surface Split* component

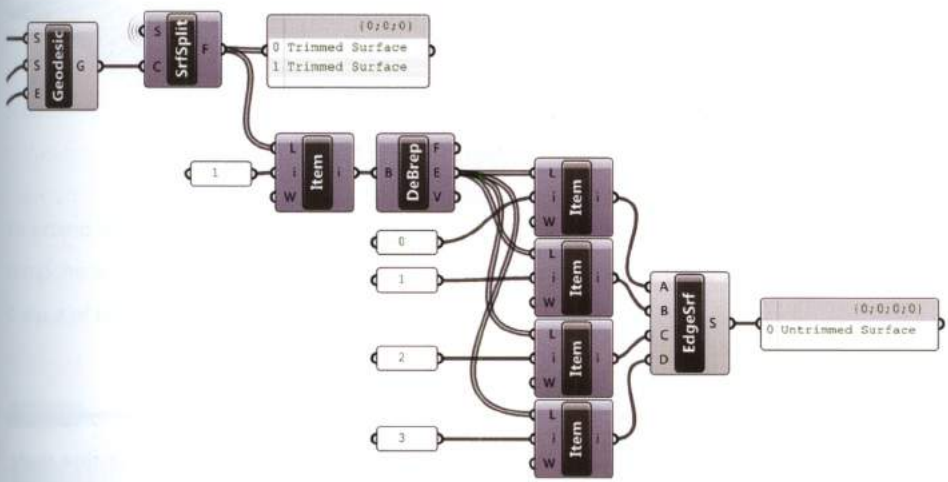
A surface can be split by a set of curves that are coincident to the surface. For example, a ***geodesic*** curve can be used as a cutting curve to split a surface into two parts. Given a surface (S) and two points (S) and (E) placed on opposite edges, a geodesic (or a shortest path curve) can be constructed between points (S) and (E) on the surface. A geodesic line can be considered a straight line in curved space.

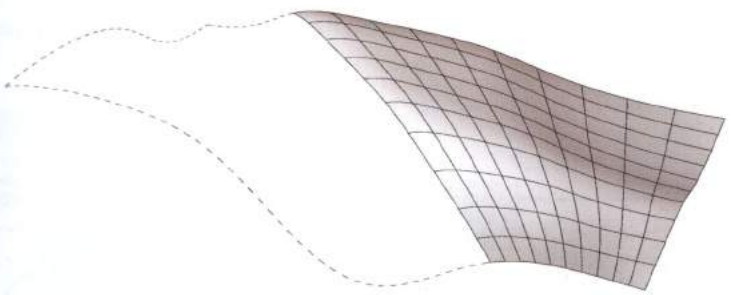




The component *Surface Split* (Intersect > Physical) splits a surface (S) at splitting curve (C) which in this case is defined by the geodesic curve.

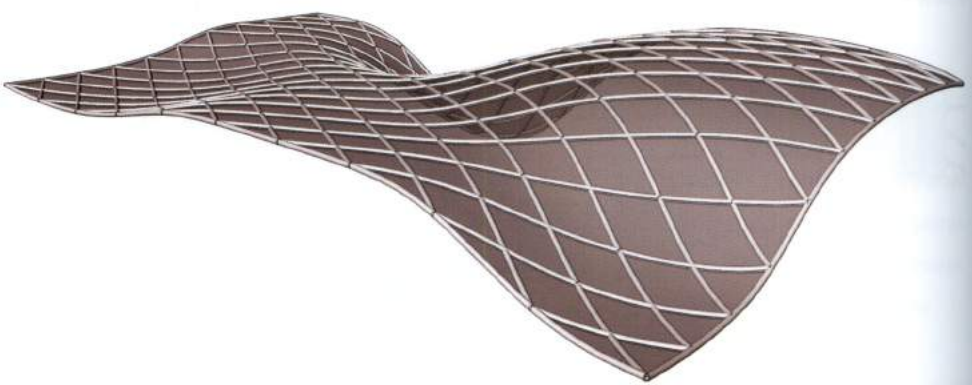
The result of the *Surface Split* component is two *trimmed* surfaces. To create an *untrimmed* surface from a *trimmed* surface the *Deconstruct Brep* component can be used to extract the edges and create a new surface using the component *Edge Surface* (Surface > Freeform).



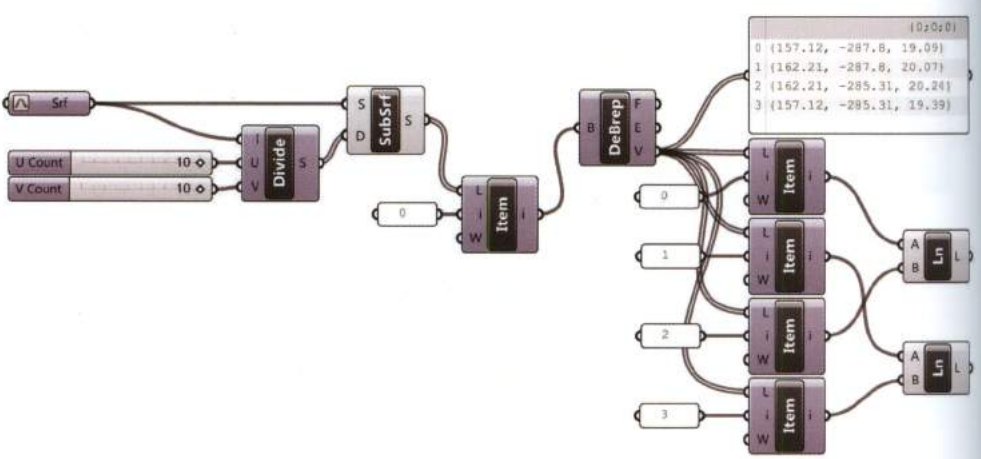


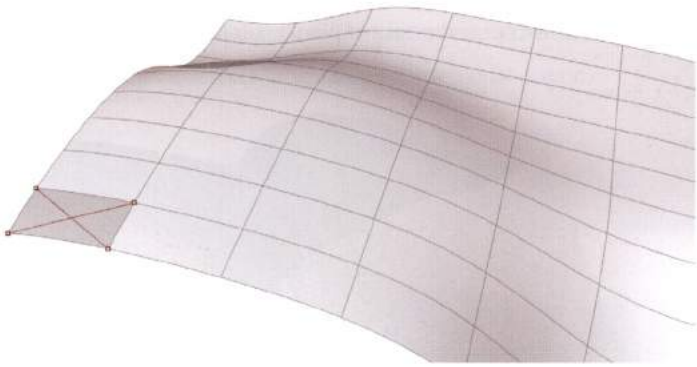
3.7.10 Example: diagrid

A diagonal grid or a *diagrid* can be created on a target surface by using the decomposition logic of Brep geometry.

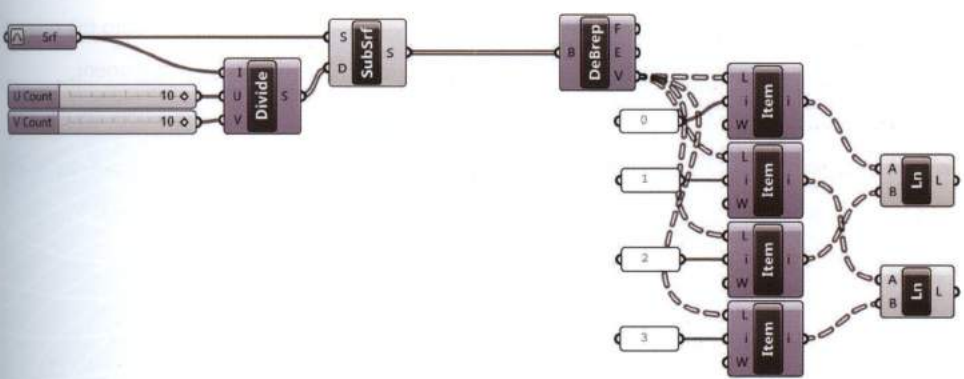


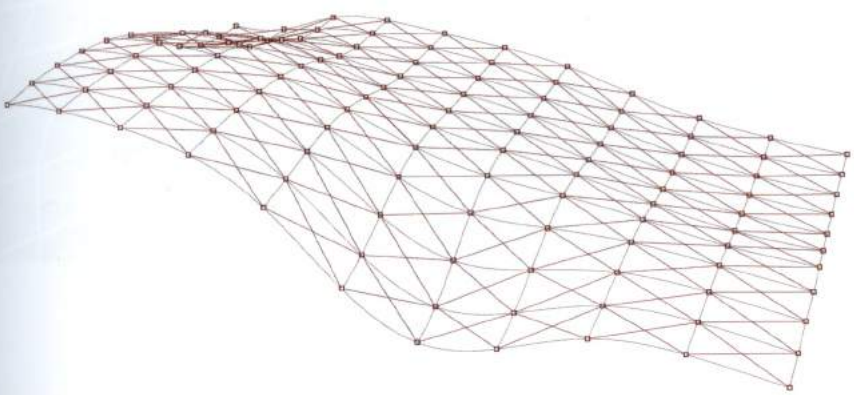
A surface set from Rhino is evenly divided into a set of sub-surfaces using the *Isotrim* component. To understand the logic of creating a diagrid, one surface item (0) will be examined as a case study. A diagrid is created by connecting opposite vertices for each sub-surface. The vertices are extracted using the *Deconstruct Brep* component and itemized separately by four *List Item* components. A line is created between the items (0 to 1) and (1 to 3), creating one module of the diagrid pattern.



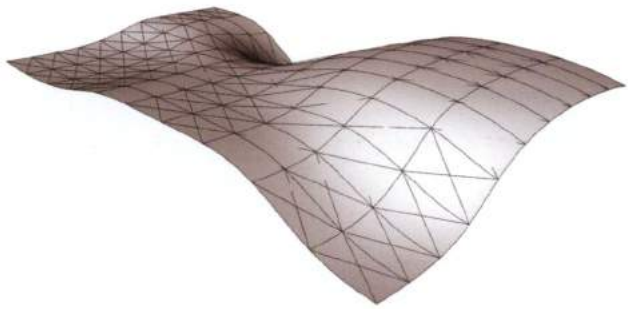


To extend the diagrid logic to the overall surface the *List Item* component, placed after the *SubSrf* component, is deleted. Then a direct connection is made between the S-output of *SubSrf* and the B-input of *Deconstruct Brep*.

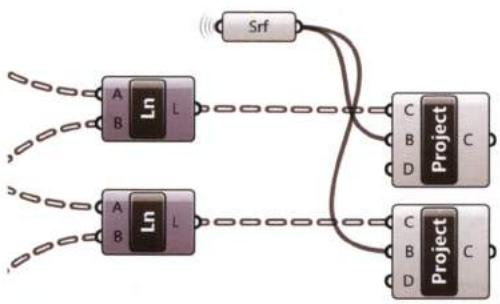


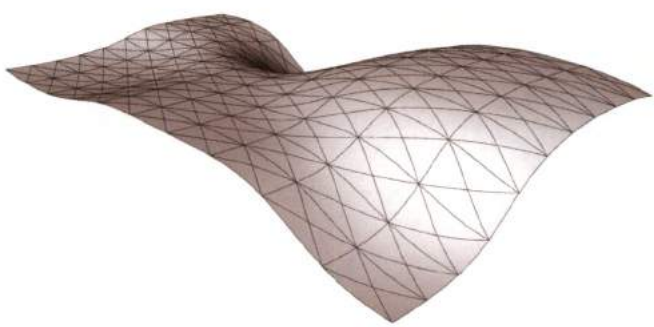


Since the algorithm is defined by connecting the vertices of sub-surfaces, the diagrid is not coincident to the surface, except in the case of planar sub-surfaces.

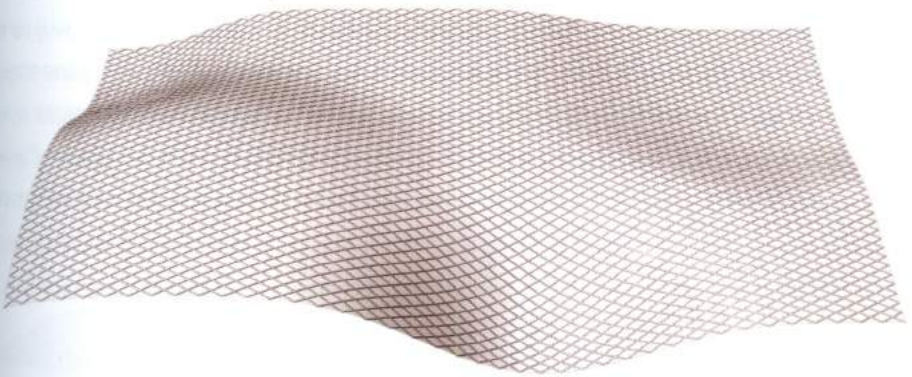


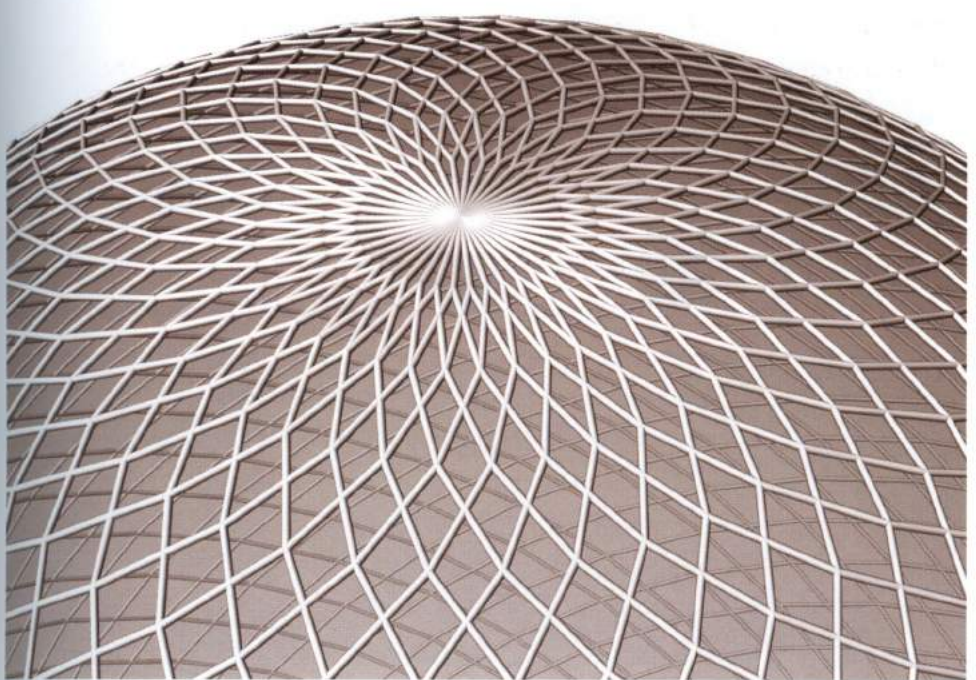
In order to generate a diagrid that is coincident with the target surface, the component *Project* (Curve > Util) is used to project the curves onto the target surface. The output of the *Line* component (L) is connected to the C-input of the *Project* component, and a receiver (wireless connection) of the original surface is connected to the B-input of the *Project* component, projecting the diagrid onto the original surface. Line segments will be replaced by curves as the output (C) of the *Project* component.





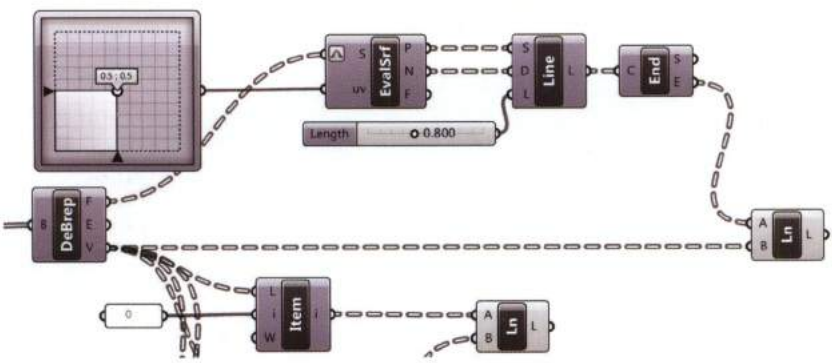
The diagrid algorithm is not bound to a specific surface, the algorithm can be used as a diagrid generator for any surface. The component *Pipe* (Surface > Freeform) will sweep a circle of a specified radius along the diagrid curves (used as rail curves) creating a three dimensional diagrid.

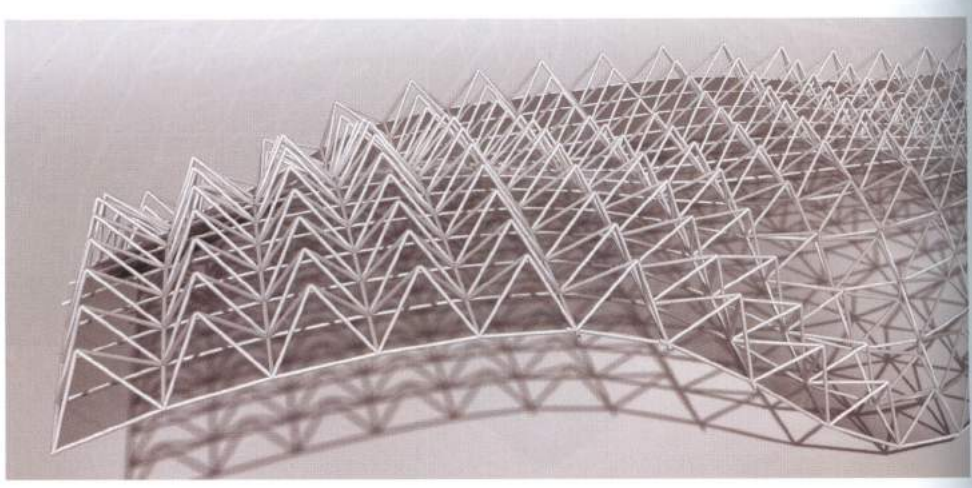




3.7.11 Example: space frame

The diagrid-algorithm can be extended to create a ***space frame***. The F-output of the *Deconstruct Brep* component is a set of sub-surfaces, so the approximate center point of each sub-surface can be calculated using the *Evaluate Surface* component connected to an *MD Slider* set to (0.5;0.5). Since the surfaces returned from the F-output of *Deconstruct Brep* are parameterized according to the domain of the parent surface they must be **reparameterized**. The P-output of the *Evaluate Surface* component is a set of points approximating the centers, and N-output the vector normals to the surfaces. The component *Line SDL* (Curve > Primitive) is used to create a set of line segments given: the start points (S) declared by the P-output, the directions (D) declared by N-output and lengths (L) specified by a slider. The component *Endpoint* is used to extract the end points of the lines which are finally connected to the V-output of *Deconstruct Brep*, generating the remaining components of the space frame.





3.7.12 Grid of equidistant points on a generic surface

Parametric Space can be imagined as a deformed *Cartesian Grid* which perfectly fits an arbitrary freeform surface. The dimensions of the grid are dependent on the initial domain as well as its subdivision. As a consequence, **edges do not have the same length** unless the surface curvature is equal to zero.

To generate a grid with equal-length edges, a grid of equidistant points on a surface is required to be declared. The Russian mathematician **Pafnuty Lvovich Chebyshev** (1821-1894) developed a method called the **Chebyshev-net** to clothe curved surfaces by cutting and sewing flat pieces of fabric. The method is published in the book *On the cutting of our clothes* (1878).

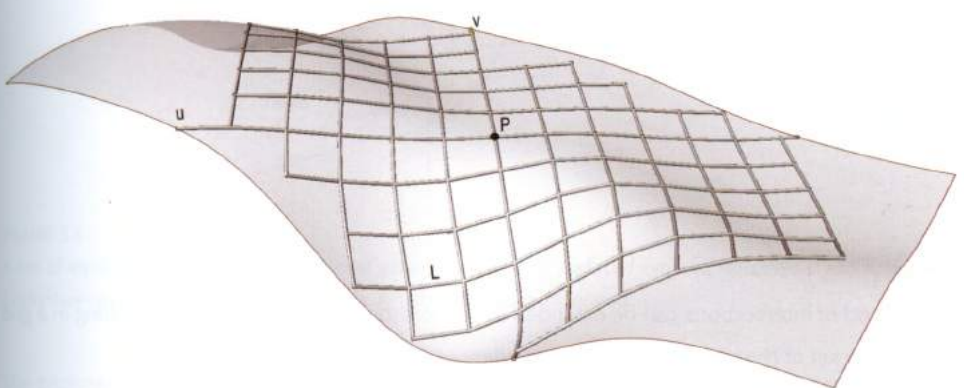


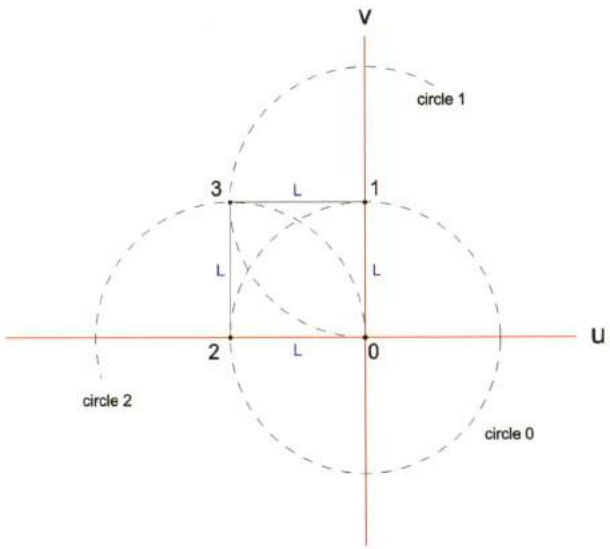
FIGURE 3.8
A grid of equal-length edges generated on an arbitrary freeform surface.

Given two arbitrary transverse curves *u* and *v* starting from an arbitrary point (P) of a surface, and a specified edge-length (L) a unique Chebyshev-net can be found. The Chebyshev-net method can be applied to the plane as well as any freeform surface. The geometric approach[10] to the Chebyshev-net can be imagined as a tridimensional extension of the *Divide Distance* logic (see 3.3.8).

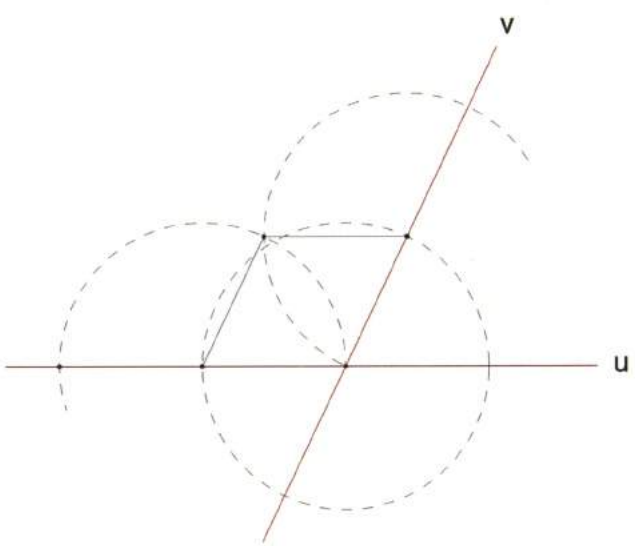
To find a grid of equidistant points in a bidimensional space a generic point 0 is used to draw two orthogonal curves *u* and *v*. The desired edge-length (L) is defined by drawing a circle *(circle 0)* with radius (L) originating from *point 0*; then intersecting the circle with the isocurves *u* and *v* to define *point 1* and *point 2*. At the intersection points circles (*circle 1* and *circle 2*) are drawn with the same

NOTE 10
E.V. Popov, 2002, *Geometric Approach to Chebyshev Net Generation Along an Arbitrary Surface Represented by NURBS.*

radius as *circle 0*. The circles intersection defines *point 3*. The procedure is repeated for the remaining three quadrants to create a grid.



The method of intersections can be extended to non-orthogonal curves *u* and *v*. Resulting in a grid formed by a set of rhombi with equal-length sides.



Moreover, curves *u* and *v* are not required to be straight. The following image displays a grid of equidistant points calculated from two isocurves with a degree of 3 on a flat surface originating from an arbitrary point 0. An equidistant grid generates a leftover area. Leftover areas do not happen in the case of domain-based grids since the segment length is dependent on equally dividing the domain.

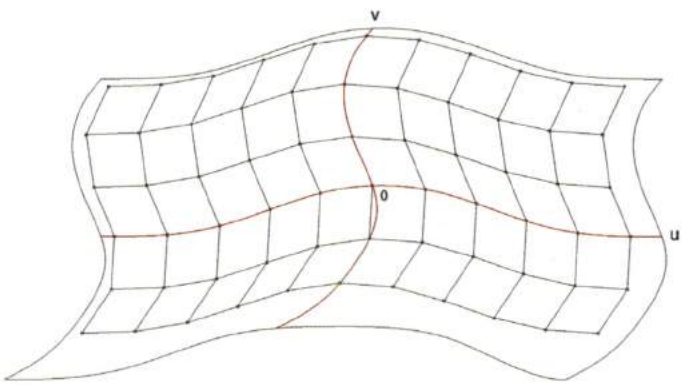
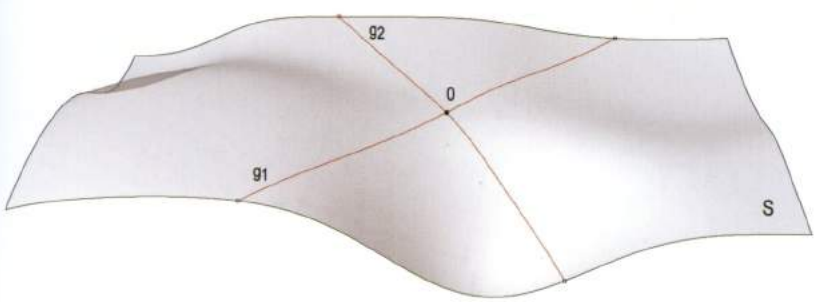


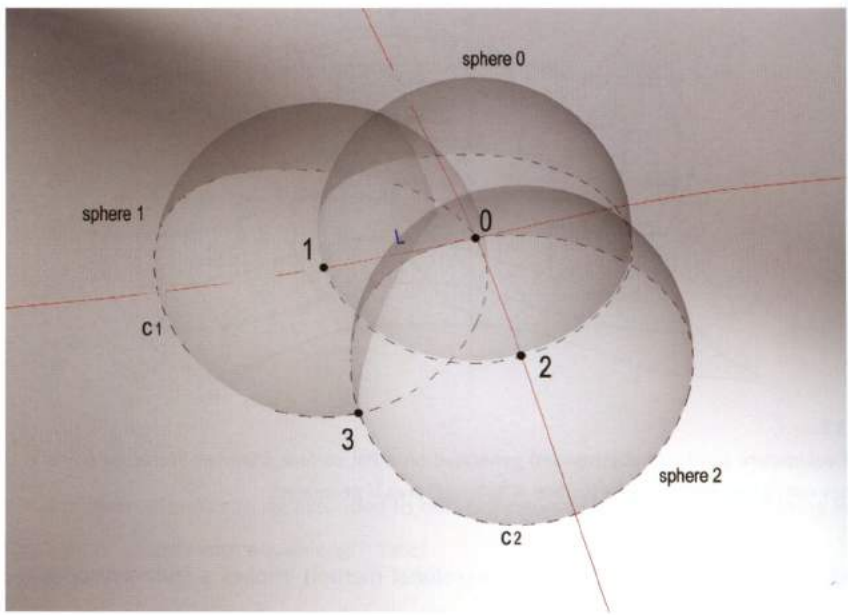
FIGURE 3.9
A net of equidistant points (Chebyshev-net) generated on a flat surface from two isocurves *u* and *v*. Usually a Chebyshev-net cannot perfectly fit a surface. A "leftover" area is generated.

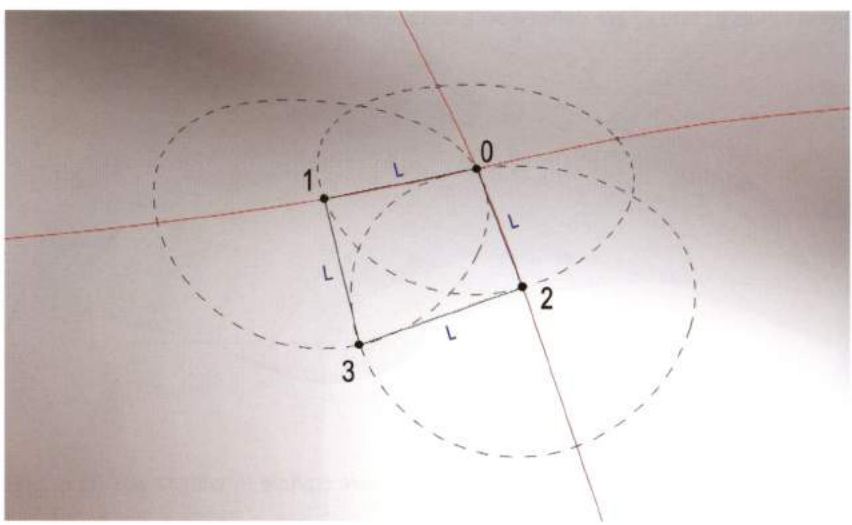
The tridimensional extension of the bidimensional method implies a tridimensional geometric construction. Meaning, a set of spheres are used instead of circles. To construct a tridimensional net, two generic curves *g1* and *g2* are defined on a surface S and their intersection (*point 0*) is found.

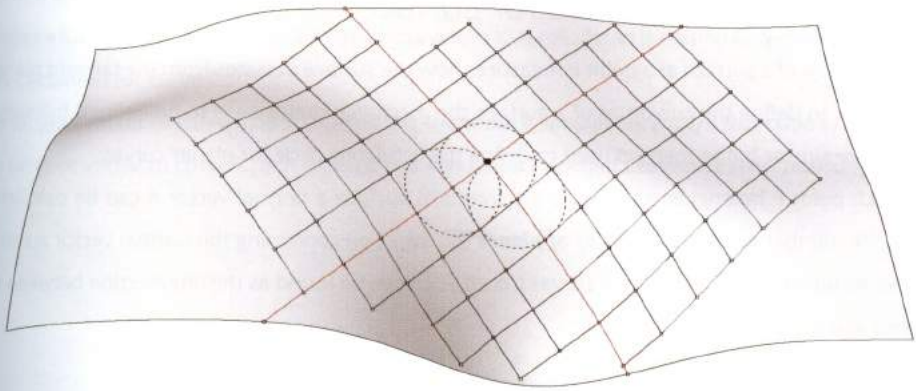


Then – expanding the bidimensional method – a sphere (*sphere 0*) with radius (L) is drawn and intersected with curves *g1* and *g2* to define *point 1* and *point 2*. *Point 3* is found by intersecting two

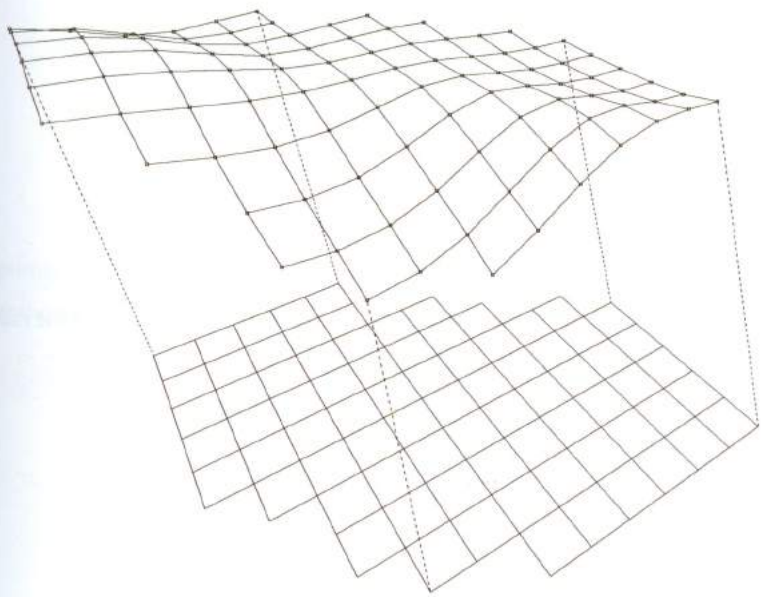
spheres (*sphere 1* and *sphere 2*) wih the same radius (L) with the surface, generating curves which are intersected to define *point 3*. The procedure must be iterated and repeated for the remaining three quadrants to generate the complete grid. Grasshopper does not provide a built-in component that allows users to perform iterations and create a Chebyshev-net. Anyhow, as we will see in chapter 7, a specific add-on is available to perform iterative procedures.







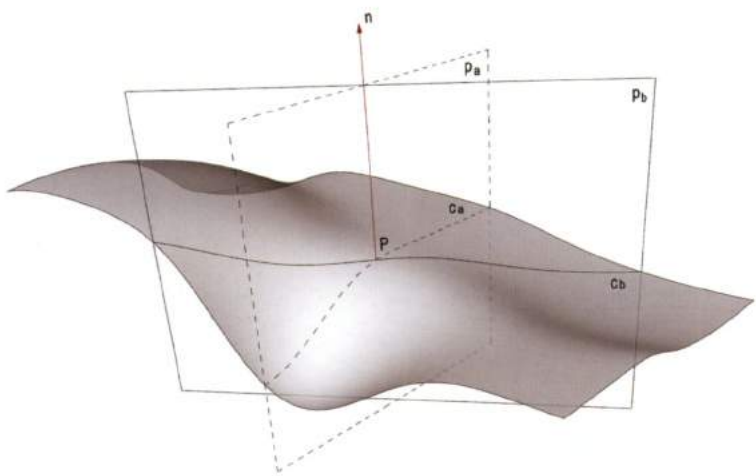
It is important to point out that an equidistant point grid can be flatten in a regular square grid as displayed below. This characteristic is crucial to form freeform structures (gridshells) starting from planar and deformable elements.



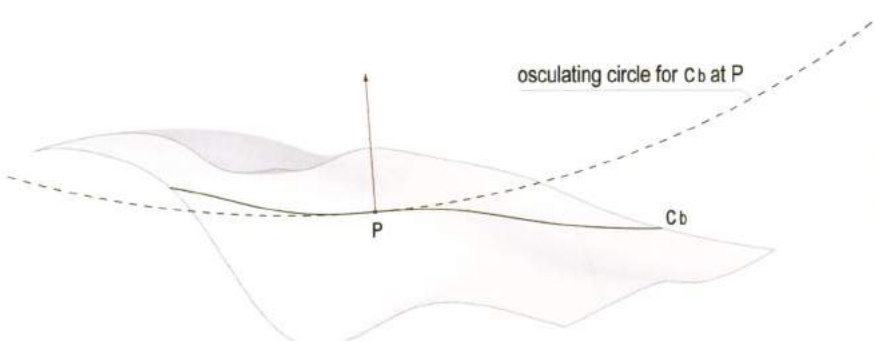
3.8 Notion of Curvature for surfaces

The curvature of a surface at a point P measures how the surface deviates from the tangent plane at P. In order to define the curvature of a surface the *osculating circle* needs to be defined following a procedure similar to the method used to define the *osculating circle* for planar curves.

For each point P belonging to an arbitrary freeform surface a normal vector *n* can be calculated. A infinite number of planes or ***sheaf of planes*** (*pa, pb,..., pi*) containing the normal vector *n* can be found. As follows, an infinite set of curves (*ca, cb,..., ci*) can be found as the intersection between the planes and the surface.

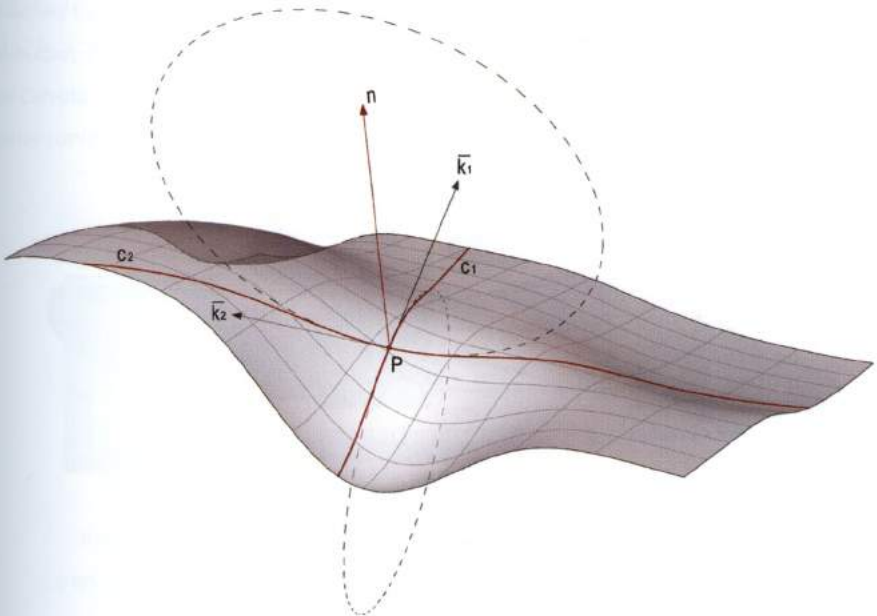


For each curve (*ca, cb,..., ci*) the curvature can be calculated at P as the reciprocal of the radius of the *osculating circle*, resulting in a set of curvature values: ka, kb,..., ki.



Within the set of curvatures (ka, kb,...ki) a *Minimum Principal Curvature* k_1 and a *Maximum Principal Curvature* K_2 can be found. The curve in which curvature K_1 is calculated is named C_1. Similarly curvature K_2 is calculated for curve C_2.

The tangent vector to curve C_1 at P is called the ***Minimum Principal Curvature Direction*** $\overline{k1}$. Similarly, the tangent vector to curve C_2 at P is called the ***Maximum Principal Curvature Direction*** $\overline{k2}$.



The two primary methods of defining curvature, ***Gaussian Curvature*** and ***Mean Curvature***, can be defined by the equations:

$$G = \text{Gaussian Curvature} = K_1 \cdot K_2 \quad [2]$$

$$M = \text{Mean Curvature} = (K_1+K_2)/2 \quad [3]$$

3.8.1 *Gaussian Curvature*

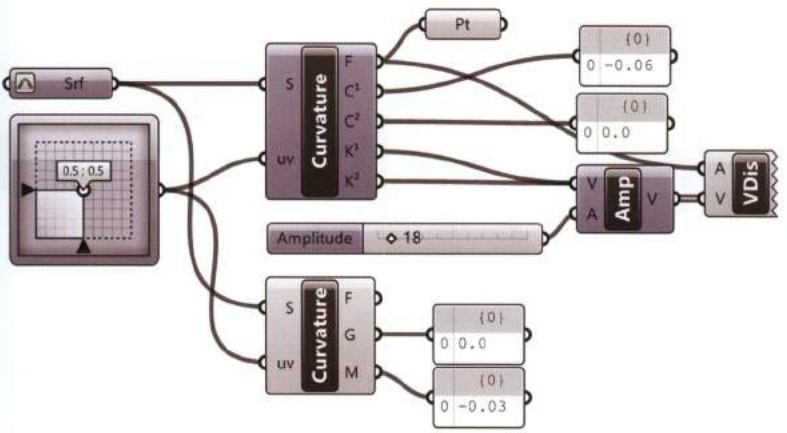
Gaussian and *Mean* curvature are significant to the theory of surfaces. **Surfaces with null *Gaussian Curvature* at all points are *Developable Surfaces***; meaning the surfaces can be flattened onto a plane without deformation. Developable surfaces are practical in manufacturing and fabrication since forms can be constructed by bending flat sheets of flexible materials such as: metal, cardboard, plywood etc. Additionally, these surfaces contain a set of straight lines which also simplify the structure's construction (linear beams).

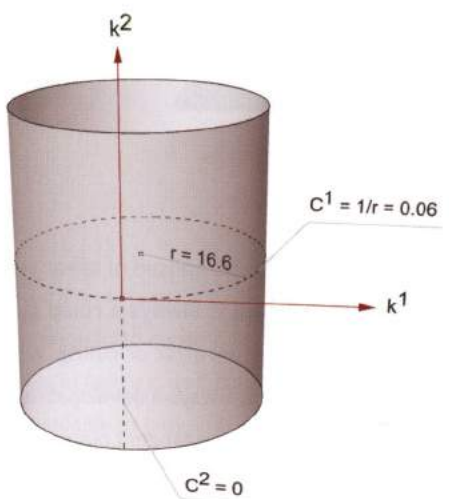
A surface is *developable* if the ***Gaussian Curvature* is equal to zero (G=0)** at each point.
The condition G=0 is satisfied when $\mathbf{k_1=0}$ or $\mathbf{k_2=0}$, since the product of zero is always zero.
For Example, a cylinder's curvature is measured at a point (P):

- *Minimum Principal Curvature* $\mathbf{k_1}$ corresponds to the generatrix *g*. Since generatrix *g* is a ruled line $k_1=0$.
- *Maximum Principal Curvature* $\mathbf{k_2}$ corresponds to *d'* which is parallel to the directrix *d*. Since *d'* is a circle its curvature k_2 has a constant value equal to the reciprocal of cylinder's radius.

The product of K_1=0 and K_2=(1/r) is equal to zero. Thus, the curvature of a cylindrical surface is zero at each point meaning *Gaussian Curvature* is null. Similarly, the cone's Gaussian Curvature is null since K_1=0 and K_2=(1/r_n).

The components ***Principal Curvature*** and ***Surface Curvature*** (Analysis > Surface) calculate curvature for surfaces. The *Principal Curvature* component outputs: K^1 and K^2 measuring the principal curvature directions as vectors, as well as C^1 and C^2 which calculate principal curvature values by definition. The *Surface Curvature* component calculates *Gaussian Curvature* as G-output and the *Mean Curvature* as M-output. For instance, for each point of the cylinder the *Gaussian Curvature* is zero whereas the *Mean Curvature* is the average of C^1 and C^2: resulting in a negative value since C^1 – a circle – has a negative curvature according to the *signed curvature* convention.

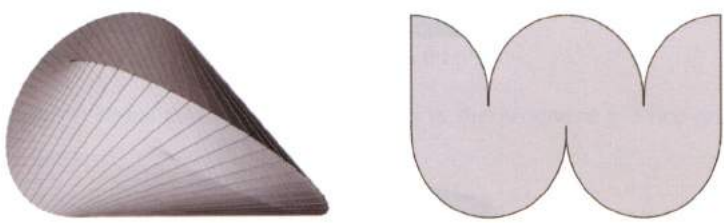




Developable surfaces include:

- ***Planes;***
- ***Cylinders;***
- ***Generalized Cylinders:*** surfaces that have the cross-section of an ellipse, a parabola or, in general, any smooth curve;
- ***Cones;***
- ***Generalized Cones:*** surfaces created by the set of lines passing through a vertex and every point on a smooth curve;
- ***The Oloid.*** A geometric object that was discovered by Paul Schatz in 1929;
- ***Tangent developable surfaces***: surfaces formed by the union of the tangent lines of a free-form curve.

generalized cylinder generalized cone tangent developable

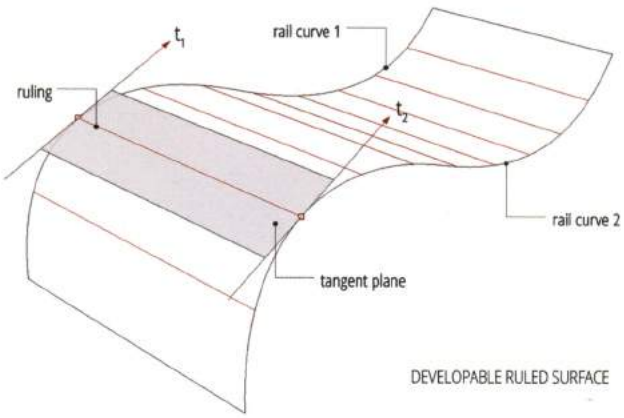


Oloid Developed oloid

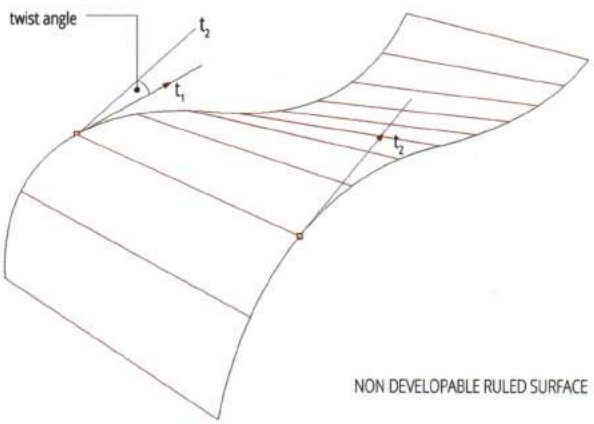
By definition developable surfaces are composed of linear cross sections in one primary direction. In other words, a **developable surface is always a ruled surface**, i.e. a surface generated by the motion of a straight line called the **generatrix** or **ruling**. A developable surface is always a ruled surface; however a ruled surface is not always a developable surface.

A ruled surface is developable when unique tangent planes can be calculated along a ruling. In other words, the **twist angle** (see following images) between tangents at rulings' end-points is null (in this

case a ruling is called a ***torsal ruling***). For example, to create a *developable ruled surface* starting from an arbitrary curve (*rail curve 1*), the geometry of the second curve (*rail curve 2*) is dependent upon outputting a null twist angle.

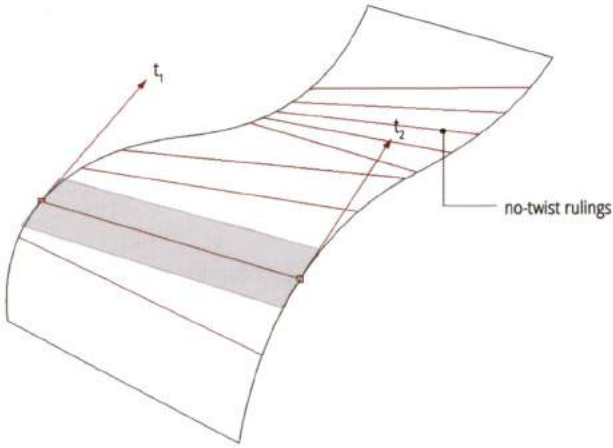


DEVELOPABLE RULED SURFACE



NON DEVELOPABLE RULED SURFACE

A different strategy must be applied when aiming to get a *developable ruled surface* from two arbitrary and fixed rail curves. Of course, it's not enough to divide both curves into equal parts and connect the resulting points through "ruling" lines. In fact, it is not guaranteed that twist angle is null for each ruling. In this case, specific algorithms (available as plug-in software) must be used in order to search for the *developable ruling lines* which have no "twist".



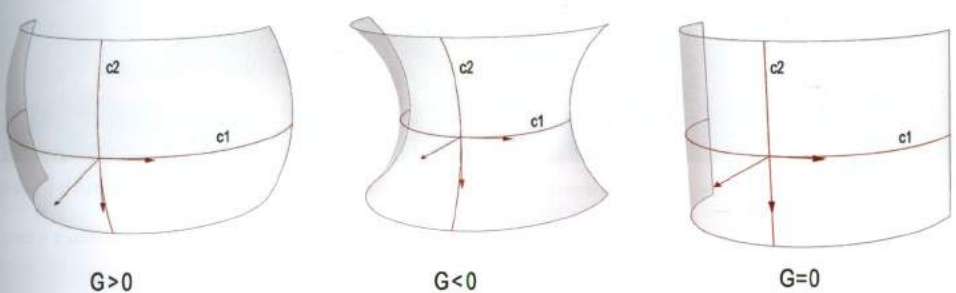
Many architects, designers and artists use developable surfaces in their work. For example, Frank Gehry often studies geometry using physical models composed of flexible sheet materials to developable geometries for his buildings.

FIGURE 3.10
F. Gehry, MARTa Herford Museum, 2005. Image by Oliver S. Wittekind.

FIGURE 3.11
F. Gehry, Richard B. Fisher Center for the Performing Arts, Bard College, Annandale-on-Hudson, New York, USA. Image by Daderot.

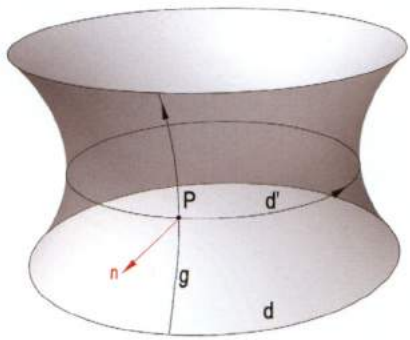
3.8.2 Sign of *Gaussian Curvature*

Gaussian Curvature is null for developable surfaces. *Un-developable surfaces* can have positive or negative *Gaussian Curvature*. *Gaussian Curvature* is positive when K_1 and K_2 are both negative or positive meaning c1 and c2 have their osculating circles lying on the same side of the generatrix. *Gaussian Curvature* is negative when K_1 and K_2 have different signs, meaning c1 and c2 have their osculating circle lying on different sides of the generatrix. If *Gaussian Curvature* is null, c2 is a straight line.



3.8.3 *Mean Curvature*

Surfaces that have a null *Mean Curvature* at all points are called ***Minimal Surfaces***. For Instance, a ***catenoid*** *minimal surface is* formed by the revolution of a catenary-curve *g* about its directrix *d*.



By definition a ***minimal surface*** has **M=0**, meaning $\mathbf{K_1=-K_2}$. In other words, the *Minimum Principal Curvature* and the *Maximum Principal Curvature* have the same value and opposite signs at each point. For instance, the catenoids curve *g* has its osculating circle on the positive side of generatrix and the curve *d'*, a circle, has negative curvature by definition.

A soap film enclosing no volume and formed between boundary constraints is a minimal surface. Soap films have a vanishing *mean curvature* because the surface area is constantly approaching a minimal surface area.

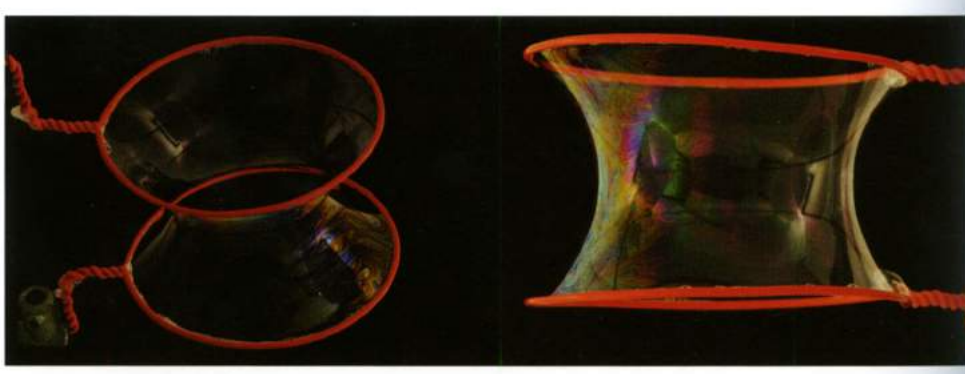


FIGURE 3.12

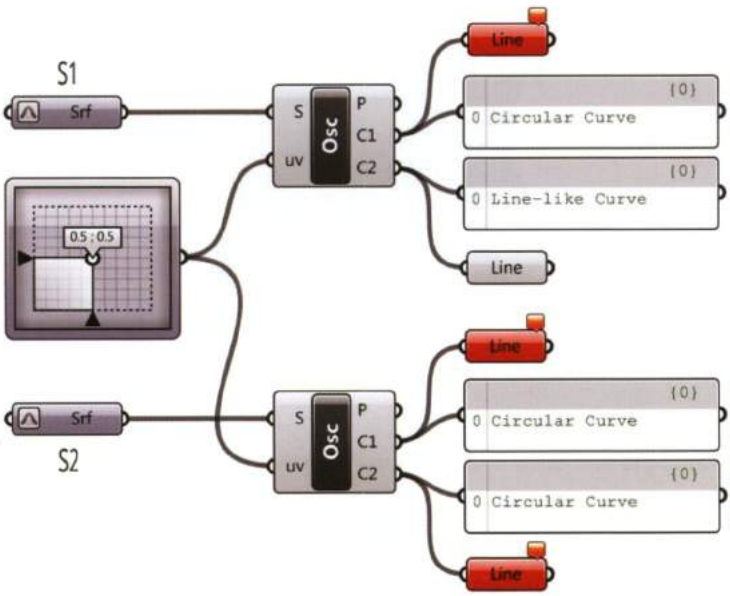
The mathematical shape "the catenoid" made of a soap film. In every point of its surface the *Mean Curvature* is zero. Image courtesy of soapbubbledk (www.soapbubble.dk).

3.8.4 Strategy: developable test

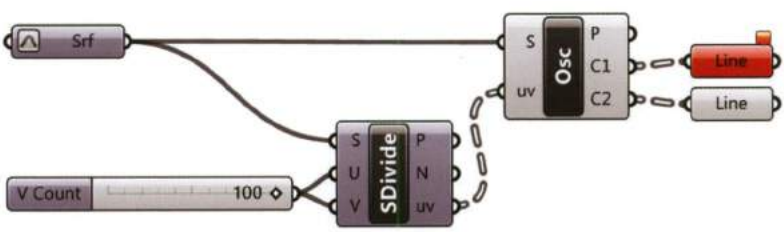
Developable primitives can be used to isolate developable sub-surfaces, however this method can lead to false results if the geometry is not trimmed from a primitive.

FIGURE 3.13
In case of a developable surface S1, at least one of the osculating circles becomes a straight line and Grasshopper draws a short segment. In a similar (not developable) surface S2 both osculating circles are represented as curves.

To indentify developable surfaces the component *Osculating Circles* (Surface > Analysis) can be used to draw the two *osculating circles* at a generic point P expressed in the LCS. If the surface is developable at least one of the *osculating circles* will be a straight line at each point. In other words, at least one of the *Line* components connected to C1 and C2 will be in "correct" status.

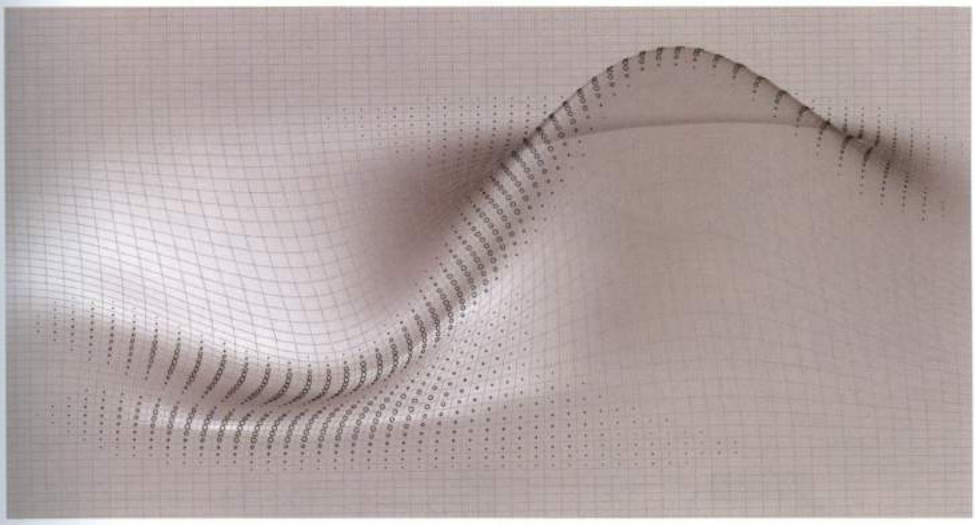


To generalize the method we can generate a grid of points on an input surface by the *Divide Surface* component and calculate the *osculating circles* at resulting points expressed in the LCS.



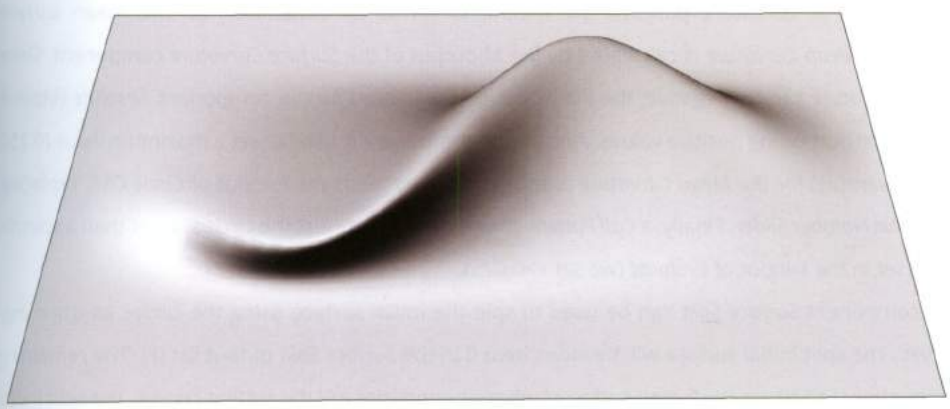
3.8.5 Example: curvature pattern

Mean Curvature analysis can be used to generate complex patterns on surfaces by using measured curvature to inform a pattern.

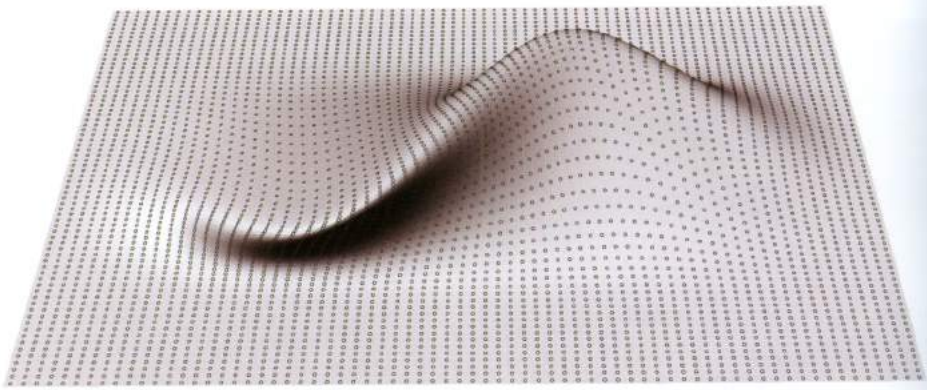


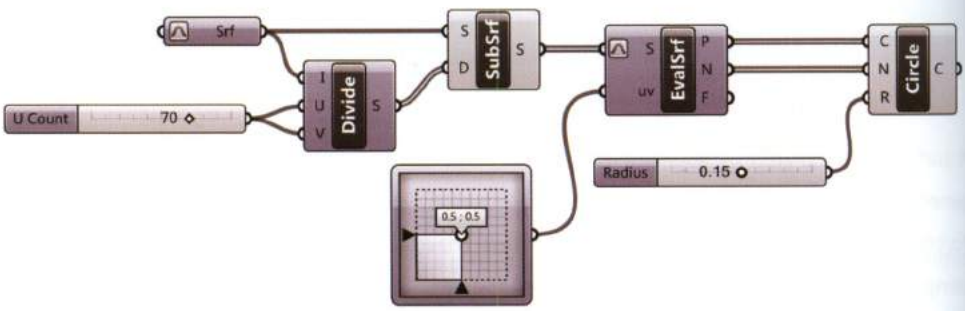
A surface manipulated by adjusting points using the *Gumball* tool, is set from Rhino. In order to generate relevant changes in curvature, surface ridges are created.

To create the pattern the set surface is reparameterized, and divided into a set of N sub-surfaces using the *Divide Domain* and *Isotrim-SubSrf* components. The *Evaluate Surface* is used in conjunction with a *MD Slider* set to (0,5;0,5), to create a point approximating the center of each surface.



To generate the pattern of circles the C-input and the N-input of a *Circle CNR* component are connected to the P-output and N-output of *Evaluate Surface*, respectively. The radius of circles (R) is set to an arbitrary value through a slider.

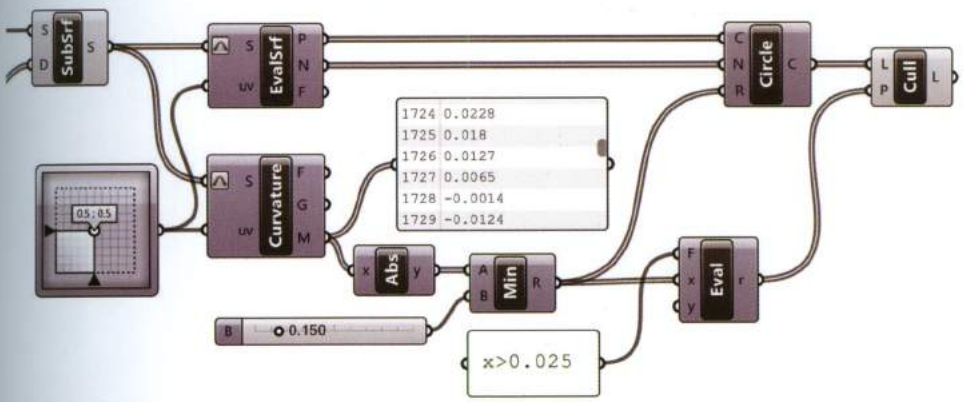




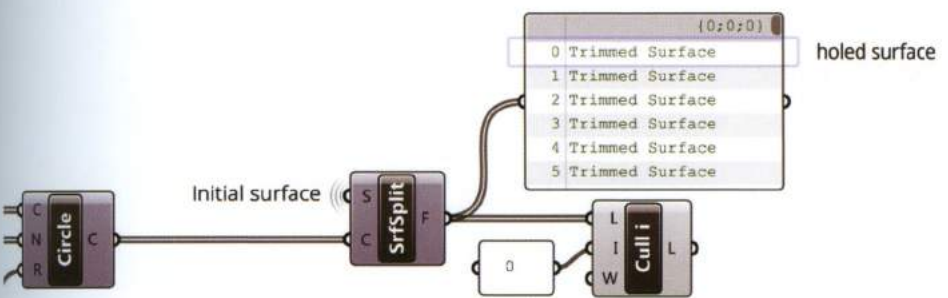
To generate a "curvature pattern" the R-input is set to be dependent on the mean surface curvature. *Mean Curvature* is calculated by the M-output of the *Surface Curvature* component. Since *Mean Curvature* can be negative, the list of values are filtered by the component *Absolute* (Math > Operators) outputting positive values. A *Minimum* component is used to set a maximum value (0.150 in the example) for the *Mean Curvature* output which now feeds the R-input of *Circle CNR*, replacing the initial *Number Slider*. Finally, a *Cull Pattern* component selects just the circles larger than a specific value set in the F-input of *Evaluate* (we set x>0.025).

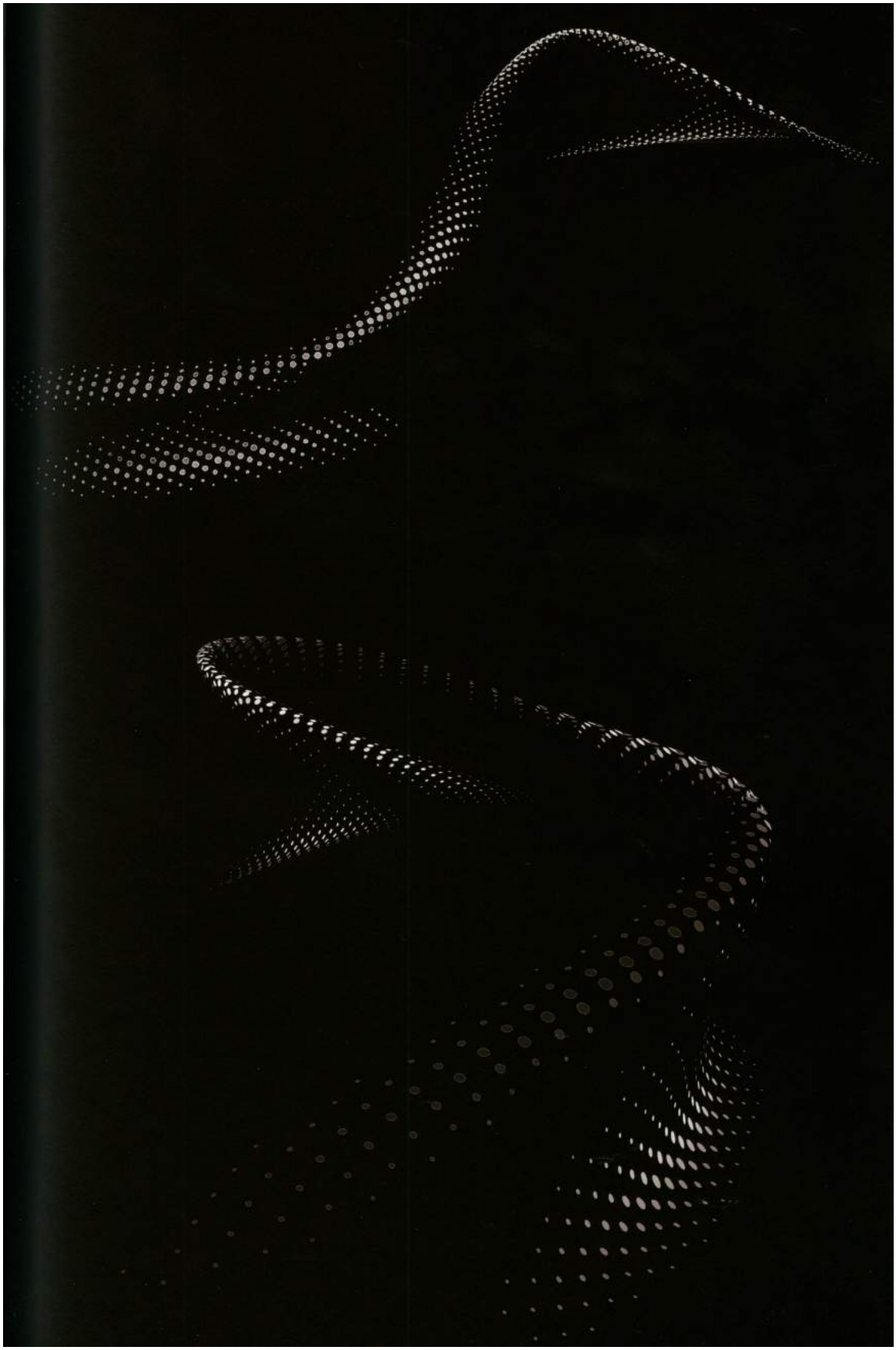
The component *Surface Split* can be used to split the initial surface using the circles as trimming-curves. The split initial surface will be index item 0 in the *Surface Split* output list (F). The remaining geometries (1 to N) are the "scrap surfaces" whose boundaries are the cutting curves. The split initial

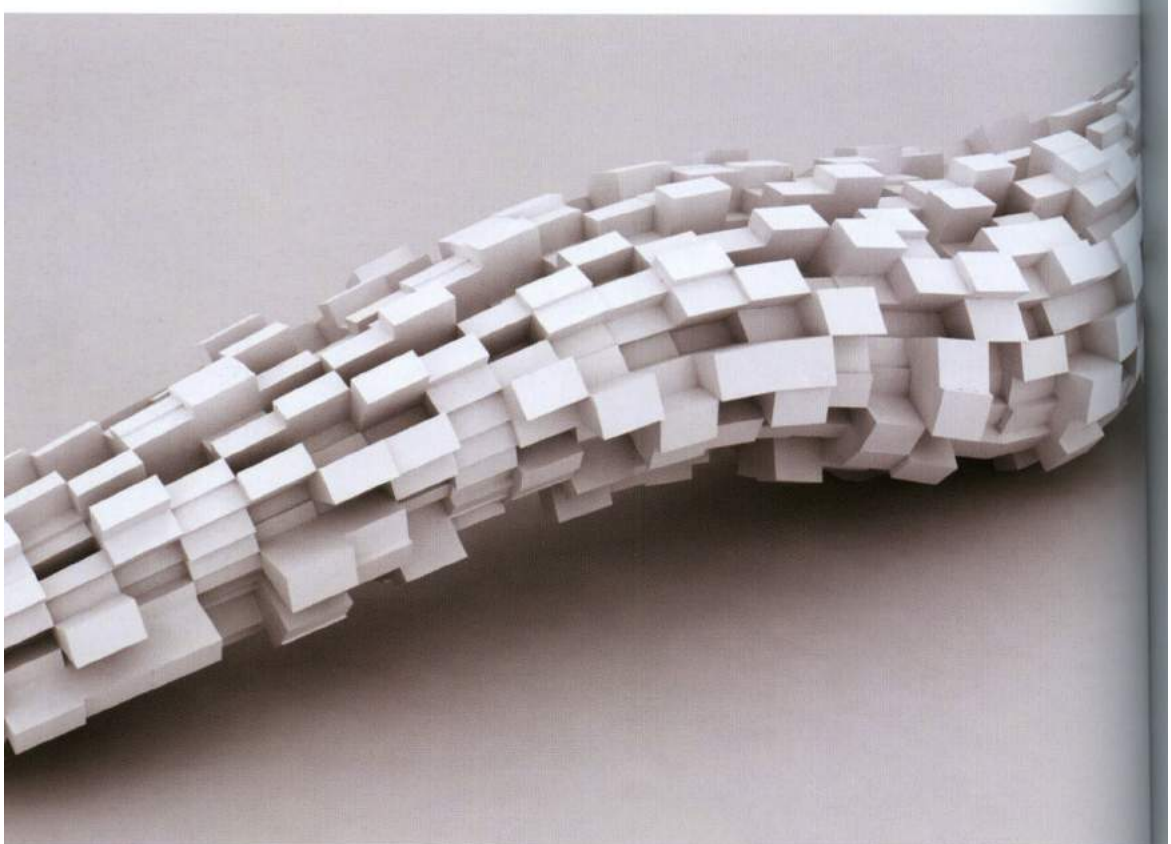
surface can be visualized by using the *List Item* component set to 0.



In order to visualize just the "scraps" index 0 – the split initial surface – can be removed from the list using the *Cull Index* component.







4_ transformations

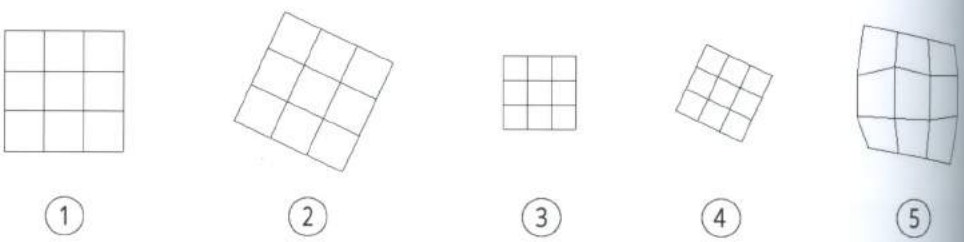
"The orthogonal features, when combined, can explode into complexity"

Yukihiro Matsumoto

Previous chapters have demonstrated how mathematics and logic are the basis of complex 3D models. In this section a new layer of complexity will be introduced, *geometric transformations*. In particular, the chapter will examine:

- **Euclidean transformations:**
  Euclidean transformations preserve lengths or angle measures. Euclidean transformations include translations, rotations, and reflections.
- **Affine transformations**
  Affine transformations do not preserve lengths or angle measures. Affine transformations include scaling, shearing and projections.
- **Other transformations**
  More complex transformations can be performed by combining transformations or by specific components such as *Morph* component.

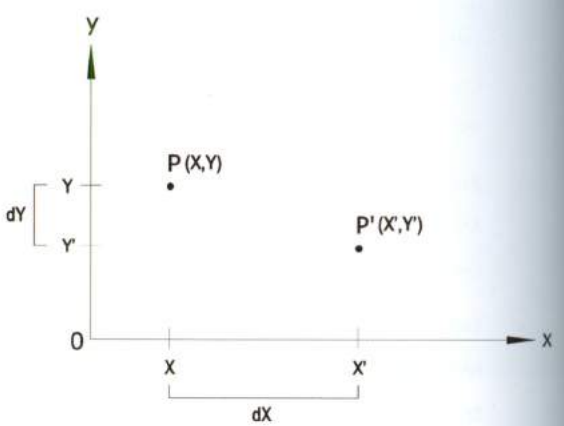
Transformation	Maintain	Does not maintain
Euclidean	shape, size	position
Affine	parallelism	shape, size, position
Similarity	shape	size, position
Morph	topological relationships	geometrical properties



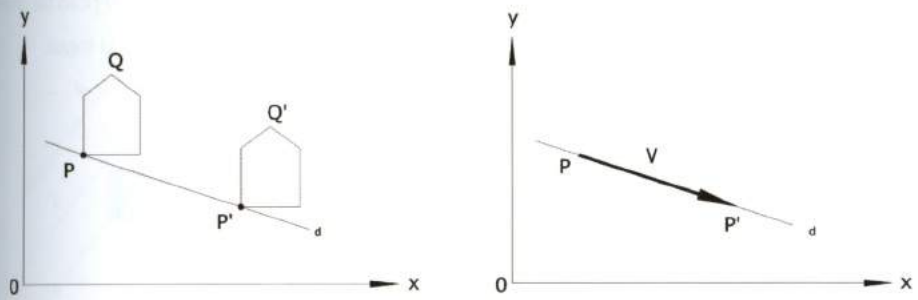
1. Original geometry;
2. Euclidean transformation: rotation;
3. Affine: scale. Scale is an affine transformation which maintains shape in addition to parallelism;
4. Similarity (scale + rotation);
5. Morph.

A transformation measures the change between two points P(x,y) and P'(x',y').
To translate a point to a different position a specific quantity is required to be to added or subtracted from its original coordinates.

$$x' = x + dx$$
$$y' = y + dy$$



An entire object can be translated performing the previous transformation to all its points or, more simply, to its vertices.



This means that a point reaches a new position by moving it of a certain distance, according to a specific direction and sense. These three characteristics define a "vector", that is a geometrical object characterized by ***direction, sense and magnitude (or length)***.

Therefore, a translation can be defined by specifying a start point P and a translation vector V.

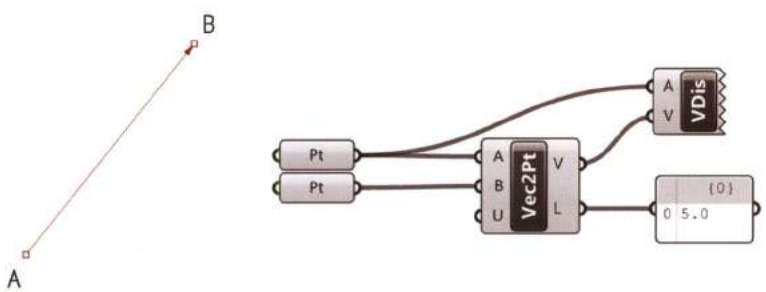
Grasshopper provides several methods to create vectors and perform operations on vectors.

4.1 Vectors

The panel (Vector > Vector) hosts the components used to create and modify vectors.

- **Vector 2Pt:**

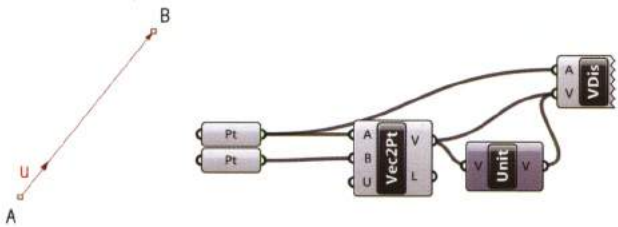
  The component *Vector 2Pt* (Vector > Vector) creates a vector between two described points: A (start point) and B (end point). The L-output of the *Vector 2Pt* component returns the magnitude of the vector and the V-output the vector. The *Vector Display* component can be used to visualize the vector in Rhino.



- **Unit Vector:**

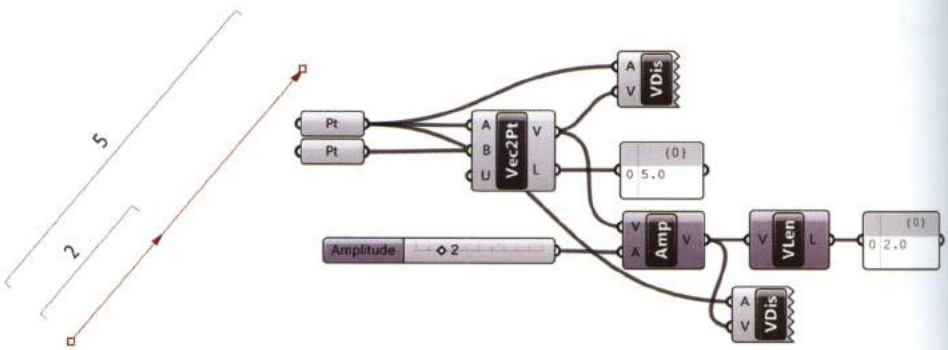
  Given a vector with a specified direction and magnitude (L) as described by two points, the component *Unit Vector* (Vector > Vector) can be used to unitize the vector or change the

magnitude to 1 unit, without modifying the direction. The component *Vector 2Pt* can be used to unitize an output vector by connecting a Boolean toggle set to True to the U-input.



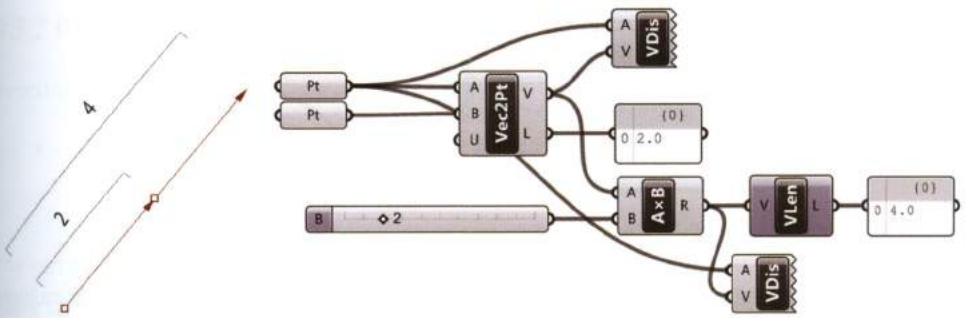
- **Amplitude:**

  The component *Amplitude* (Vector > Vector) sets the length of a given vector to an input value (A). For instance, a vector with an initial length equal to 5 is connected to the input (V) of the *Amplitude* component, the component will return a new vector with a magnitude (A) while maintaining the original direction and sense. Sense will change using a negative value as an input (A).



- **Scalar multiplication:**

  Scalar multiplication refers to the multiplication of a vector by a scalar number N. If N>0 a vector in the same sense will result, if N<0 a vector in the opposite sense will result. The vector's new length is equal to the product of the initial vector's length and N.



- **Unit X, Unit Y, Unit Z:**

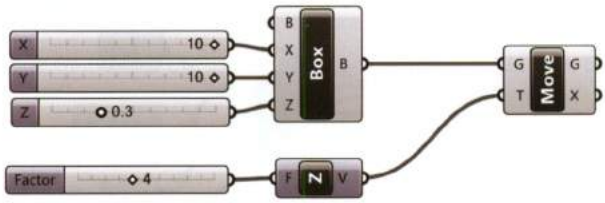
  The components *Unit X, Unit Y and Unit Z* (Vector > Vector) define unit vectors along the x, y and z axes. The input (F) can be used to set a length through embedded scalar multiplication.

4.2 Euclidean transformations

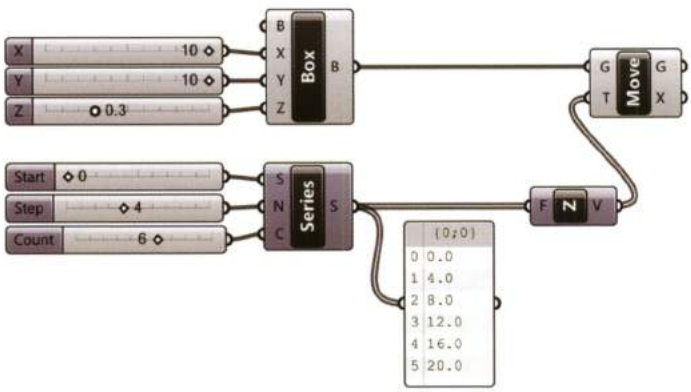
Euclidean transformations (Transform > Euclidean) include translations, rotations, orientation, and mirror components.

4.2.1 Translations: *Move* component

The component ***Move*** (Transform > Euclidean) translates a geometric entity (G) according to a vector (T). For example, a box created using the component *Center Box* (Surface > Primitive) is translated in the z direction using a *Unit Z* component.



If input (F) of the *Unit Z* component is connected with multiple numeric values it will return multiple vectors, these vectors can be used to perform multiple translations. For instance, a multistorey building can be modeled by connecting a *Series* component to the F-input of *Unit Z* component. In this simplified model the N-input of the of the *Series* component sets the distance between floors, the C-input sets the total number of floors and the S-input defines the starting value of the numerical sequence. If (S) is set to 0 the first floor will coincide with the original box.

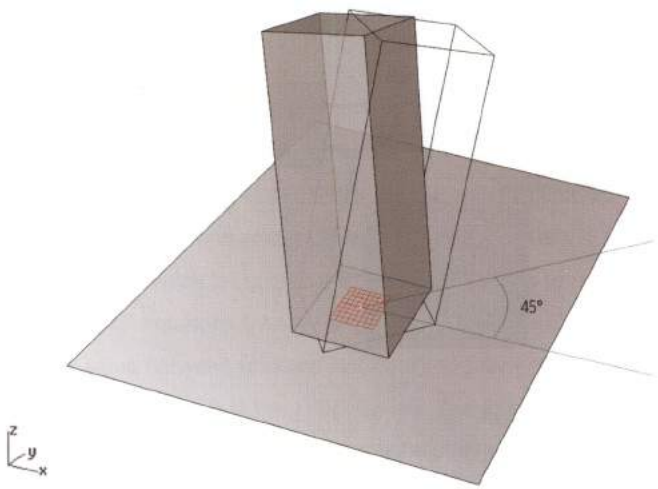


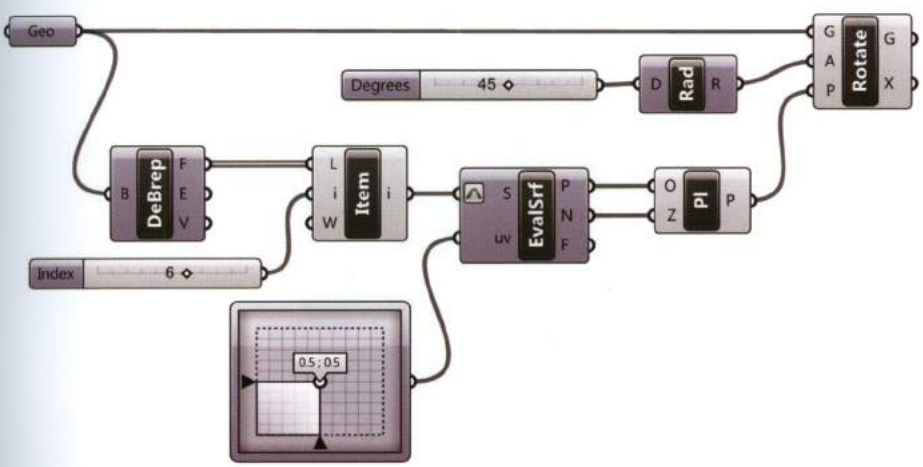
4.2.2 Rotations: *Rotate* and *Rotate Axis*

A rotation is the motion of a rigid body around:

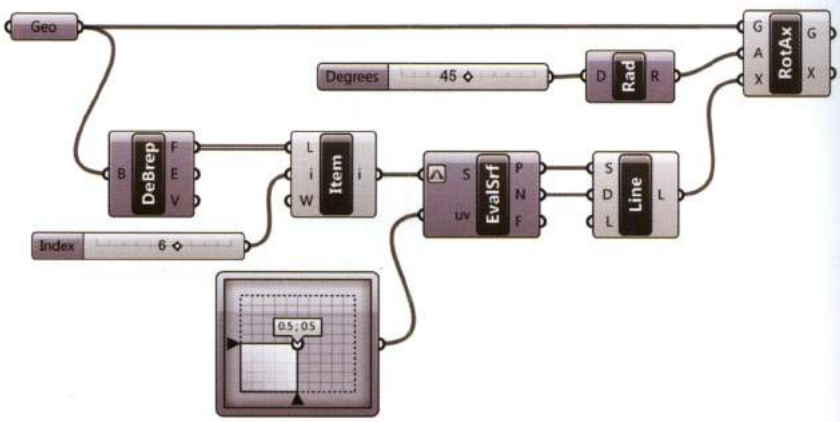
- a fixed point (called rotation center);
- a straight line (called rotation axis).

The components *Rotate* and *Rotate Axis* (Transform > Euclidean) rotate a geometric entity (G) in a plane or around an axis respectively. For instance, a prism can be rotated 45° with respect to its original position on an arbitrary-oriented surface by using the component *Rotate* in conjunction with other components that define a rotation plane.

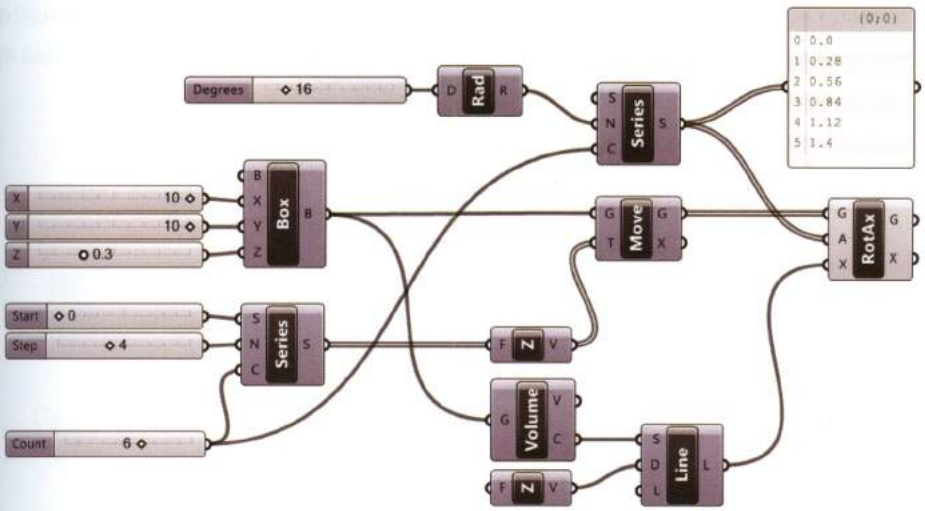




The *Geometry* container imports the prism. We can decompose it by using a *Deconstruct Brep* component and then we can extract the lower face by *List Item* set to 6. *Evaluate Surface* calculates the normal vector of the lower face at its center; then *Plane Normal* (Vector > Plane) returns the plane to rotate according with. Finally, ***Rotate*** is connected to the prism (G-input) and to the plane in (P). The rotation angle is set through a *Number Slider* ranging between 0° and 360°. Since *Rotate* accepts angle in radians, we have to convert degrees into radians by the Radians component (Maths > Trig).
In alternative, we can rely on the ***Rotate Axis*** component which needs a line (X-input) to perform a rotation. The line is created by a *Line SDL* component.



The component *Rotate Axis* used in conjunction with the *Series* component, can create progressive rotations.



With reference to the previous example (see 4.2.1), it is possible to define the rotation axis as a vertical line, starting from the gravity point of the initial box (by the *Volume* component). The progressive rotation is made using different rotation angles (*Series* outputs) instead of just one value. The C-input of *Series* must be set so that its value is equal to the number of objects to rotate.

To calculate the incremental rotations (*relative angle* between consecutive floors) as a function of the *twist angle* (angle value between first and last floor), this latter is divided by the number of floors minus one; the division's result is the *relative angle*. The following images display different incremental rotations referred to the wireframe model of a multistorey building: the angle between the first and last floor (twist angle) is 90°.

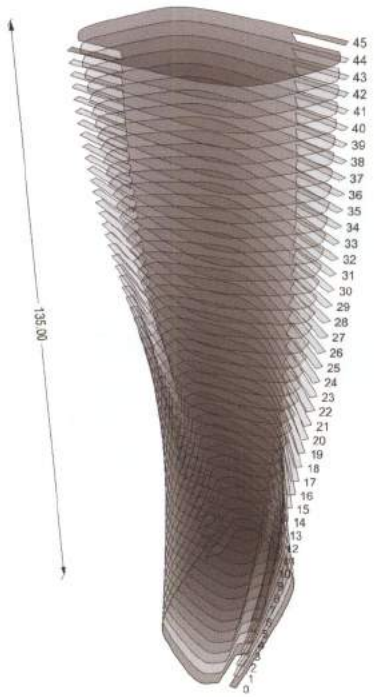


FIGURE 4.1
Progressive rotation applied to a simplified model of a multistorey building.

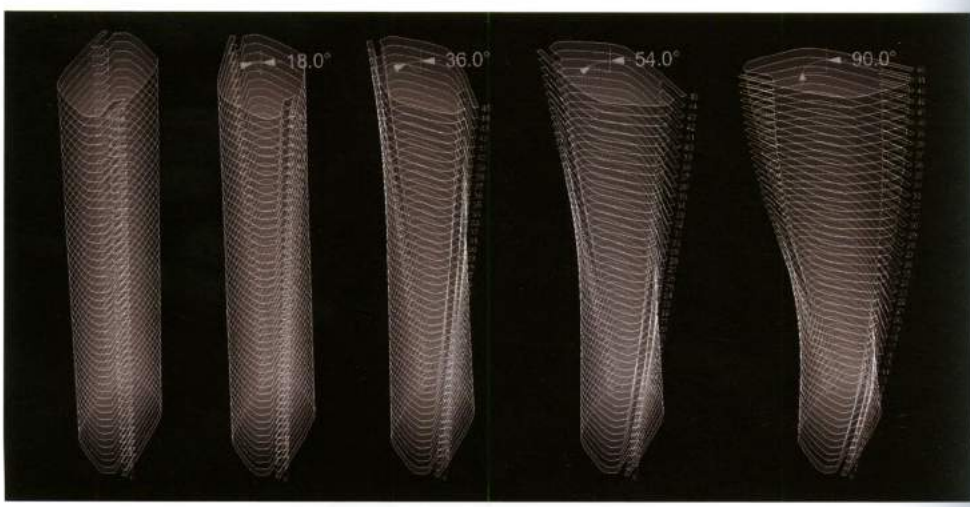
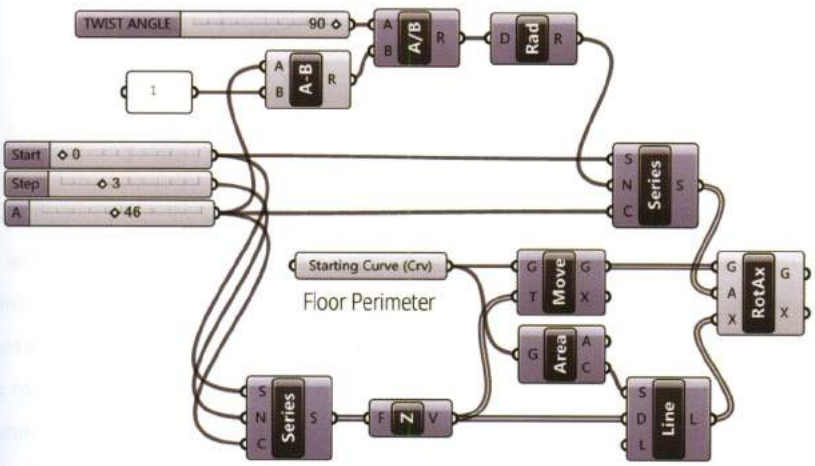


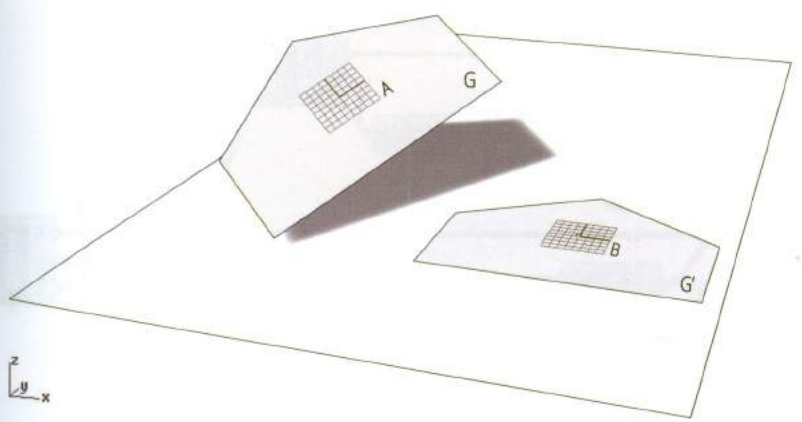
FIGURE 4.2
The image illustrate different configurations of the model as the *twist angle* changes.

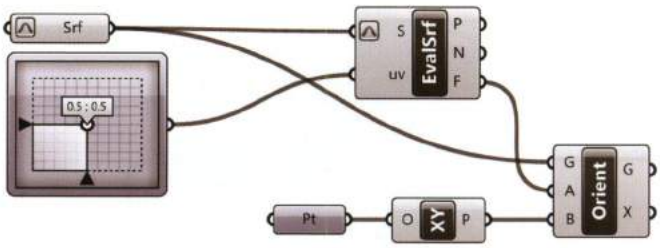
The Equation *relative angle = twist angle / (number of floors-1)* calculates incremental rotations for each floor.



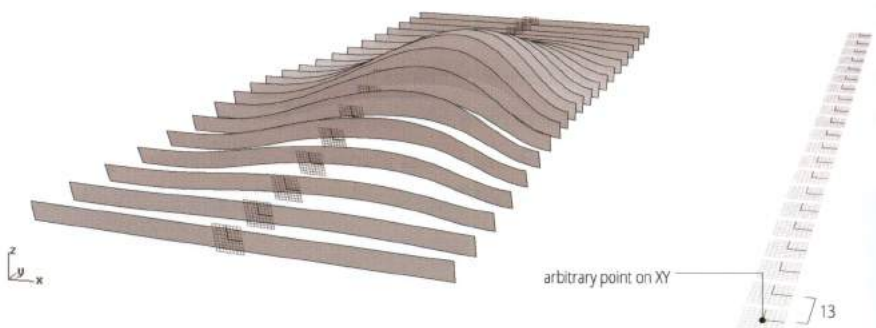
4.2.3 *Orient* component

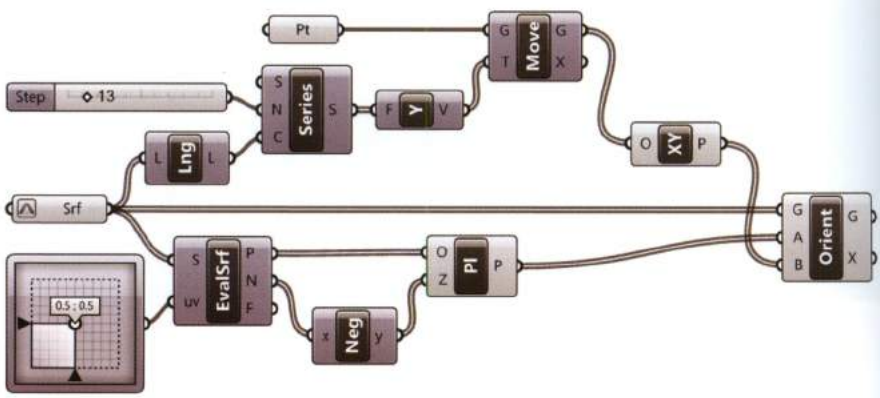
The component ***Orient*** (Transform > Euclidean) combines translations and rotations into a single transformation. A **geometric entity (G)** arbitrarily oriented in the space, can be oriented to new position (G') by specifying an **initial plane A** and a **inal plane B**. To orient geometry G to a new position G' the *Evaluate Surface* component is used to find an initial plane defined as a tangent plane output (F) at LCS point (0.5;0.5), and a final plane described by the component *XY Plane* (Plane > Vector). The orient component adjusts the initial plane to be coincident with the final plane.

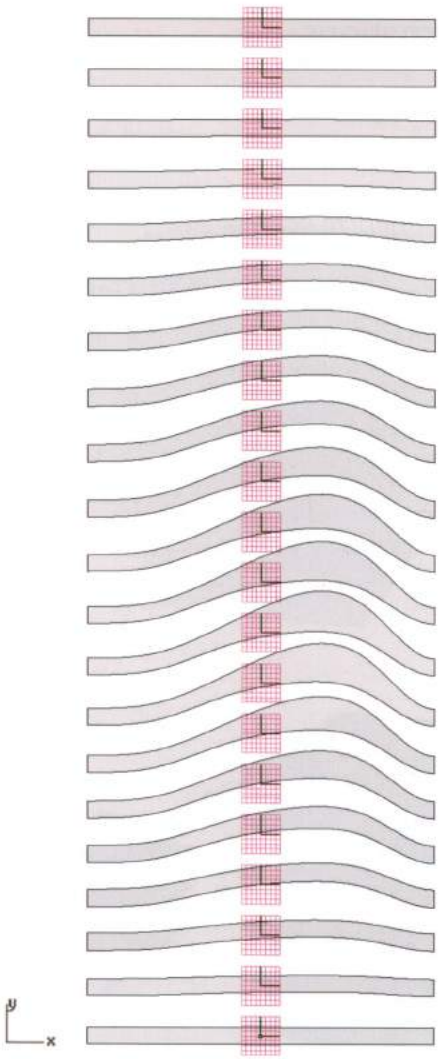




The orient component can also be used to orient a list of arbitrarily located ribs-surfaces to the XY plane. The *Evaluate Surface* component is used define a plane on each of the ribs. The *Evaluate Surface* normal vector output (N) is set as the z-axis direction (Z-input) of the *Plane Normal* component (Vector > Plane) defining the initial planes. To define the final planes, an arbitrary point is translated in the Y direction using the *Series* component as a scalar multiplier with: N-input set to the step size (13) and the C-input value coincident with the number of surface-ribs (calculated through *List Length*). At each point a *XY Plane* is defined. Lastly, the ribs-surfaces are oriented to the final planes.






y
x

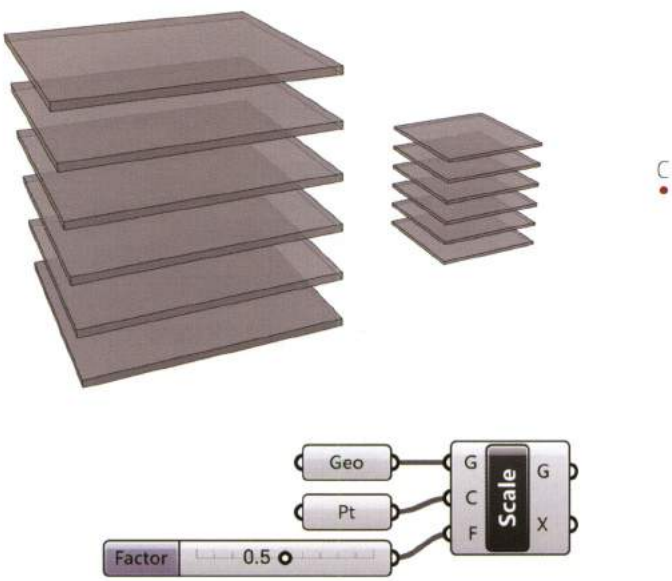
4.3 Affine transformations

Affine transformations (Transform > Affine) include scaling, shearing, projection and mapping components.

4.3.1 *Scale* component: uniform scaling

The component ***Scale*** (Transform > Affine) is a geometric transformation that enlarges or reduces objects uniformly in x, y and z directions. The ***Scale*** component resizes an initial geometry (G) by a scale factor (F) relative to a **center of scaling point (C)**. The center of scaling (C) can be any point. The **scale factor** is a positive number and **cannot be 0** (a null scale factor is a mathematical error). In particular, we have three main cases:

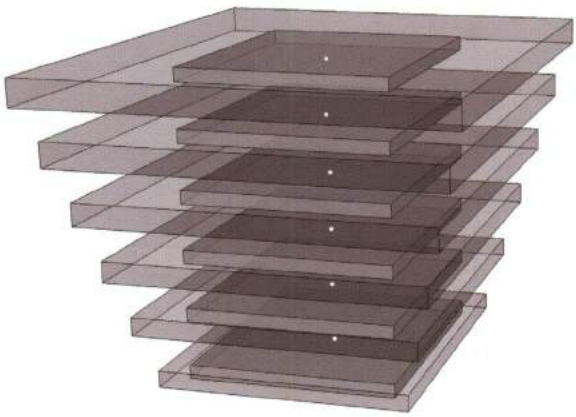
- **0 < F < 1**: objects reduce;
- **F = 1**: objects do not change;
- **F > 1**: objects enlarge.

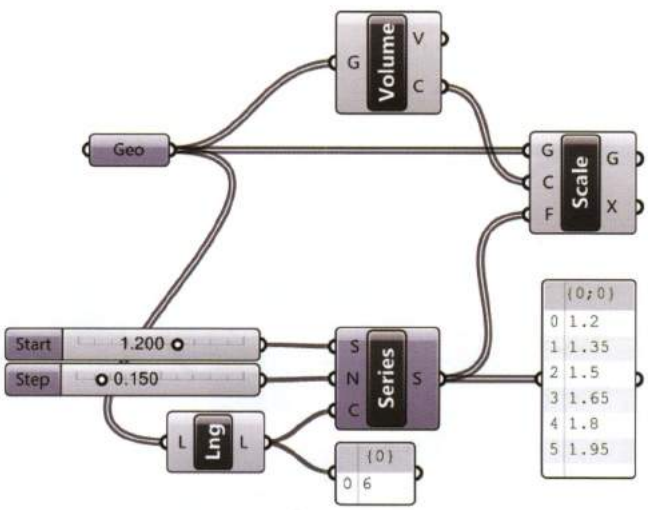


Scaling can also be performed on several objects. For instance, if the same scale factor is applied to the entire set of geometries the entire set of geometry will be scaled by the same proportion.

Alternatively, different scale factors can be applied to each geometric entity.

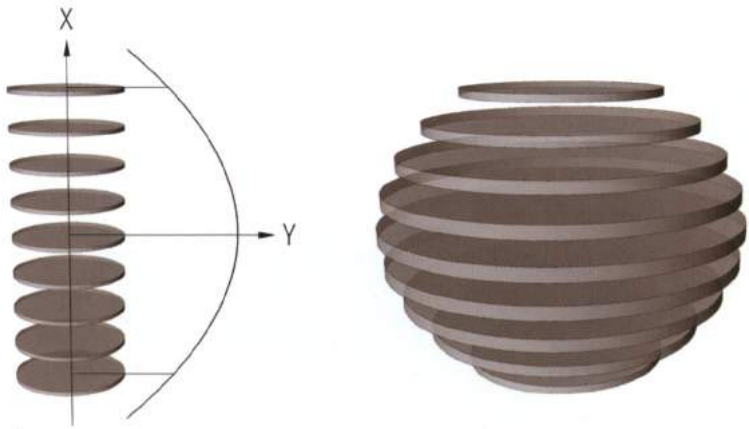
For example, to progressively enlarge a set of six boxes the input (F) of the *Scale* component is supplied with a list of increasing scaling factors by the *Series* component. The center of scaling input (C) is defined as the centroid of each box respectively.





The *Series* component generates a list of increasing or decreasing **linear** scaling values.

To generate more complex forms of scaling using mathematical functions the component *Graph Mapper* (Params > Input) can be used.

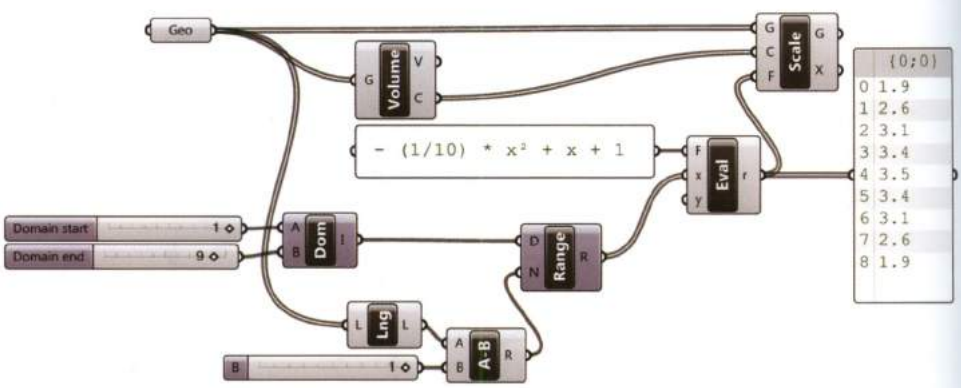


For instance, if the input (F) of *Scale* component is supplied with a **symmetrical** numerical sequence: (1, 2, 3, 4, 5, 4, 3, 2, 1) a parabolic form will result.

More specifically if the equation,

$$y = -(1/10) * x^2 + x + 1 \text{ with } 1 < x < 9,$$

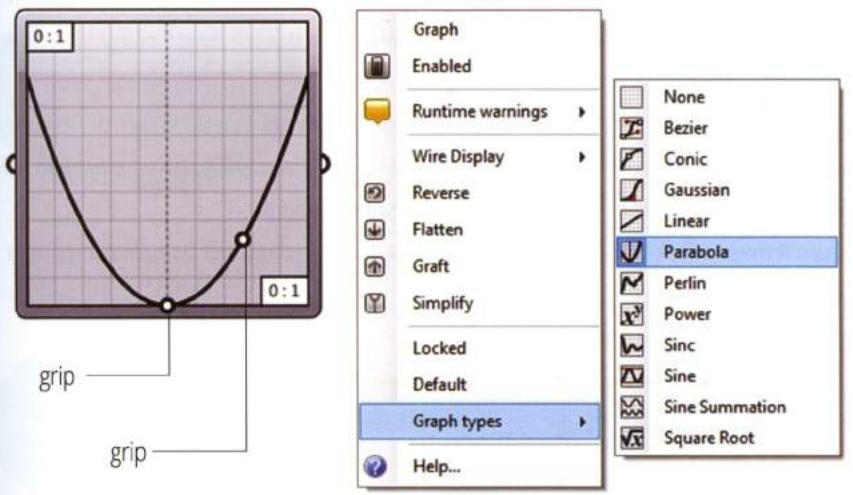
is supplied to the input (F) of the *Evaluate* component, the output (r) will return a symmetrical sequence co-domain which can be used as the scaling factor (F).



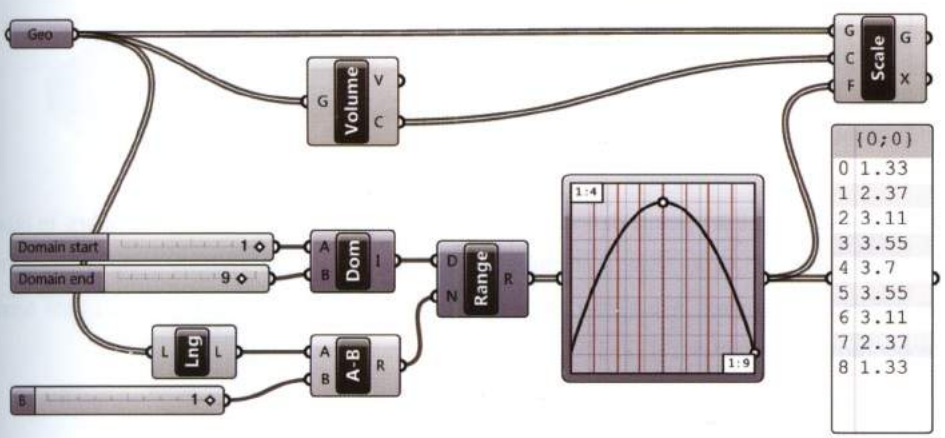
Of course it's not easy to manipulate complex equations to get a specific objective, but fortunately we can rely on an alternative tool available in Grasshopper: the *Graph Mapper*.

4.3.2 *Graph Mapper* component

To more easily define mathematical functions the component ***Graph Mapper*** (Params > Input) can be used to select a desired mathematical expression from a list.



The *Graph Mapper* component replaces the *Evaluate* component in the previous example. Instead of defining an equation, a desired function can be selected from the list provided by the *Graph Mapper* component. The function can be manipulated by adjusting the graphs grips.

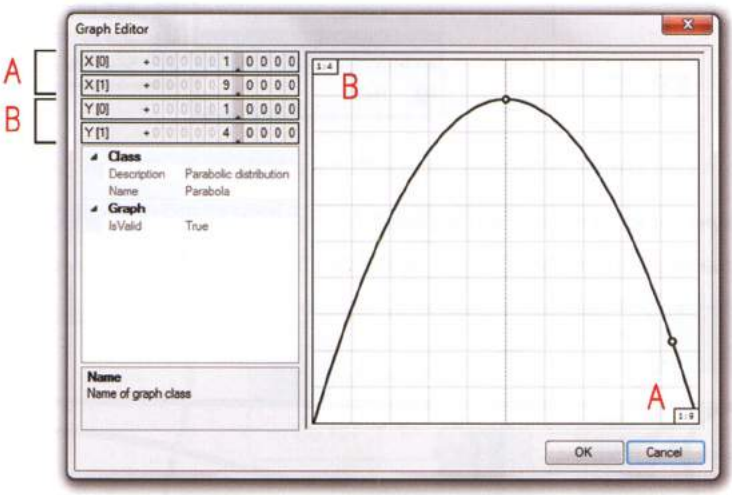


When you change the graph you actually change the mapping function, getting different results.

As for the *Evaluate* component so for the *Graph Mapper*, a proper domain must be supplied for the function using the *Construct Domain* component in conjunction with *Range*. It's important to point out that the *Graph Mapper* must be feed by a number of values equal to the number of geometries to scale. Since *Range* generates N+1 values (where N is the number of values defined in the N-input), we must subtract 1 from the number of objects coming from the *List Length* component connected to the initial geometries.

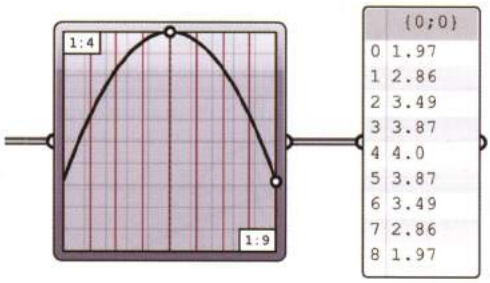
Nevertheless, the numeric domain is not the only thing to pay attention to. In fact, the graph has two "internal" domains to set. Double-clicking the *Graph Mapper*, a window appears and, with reference to the following image, we must change the two domains: A and B.

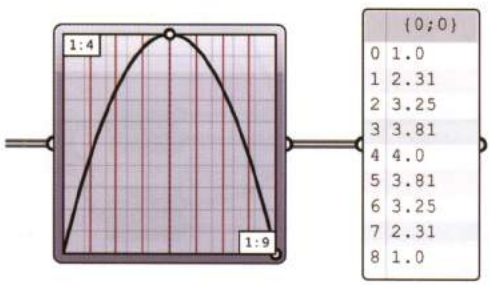
The domain A must be set according to the values used for the *Construct Domain* component (in our example 1 and 9).

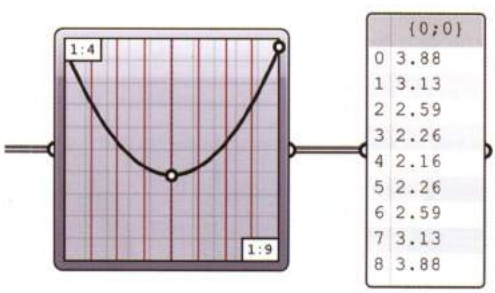


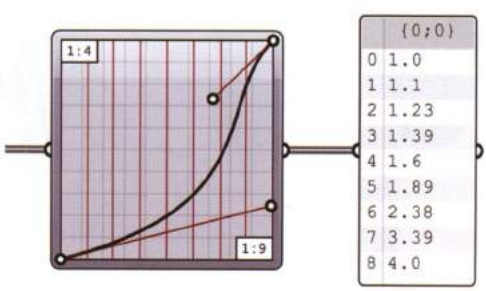
The domain B can be arbitrarily defined. It controls the extremes of the list of numbers. In other words, if B is set to [1,4] the lower scale factor will be 1 and the maximum scale factor will be 4. The two internal domains are displayed in the upper left and in the lower right corners of the *Graph Mapper* component contextual window.

The following images demonstrate how different graphs and different domains output varying geometric sets.









The *Graph Mapper* component can also be applied to the multistorey building example.

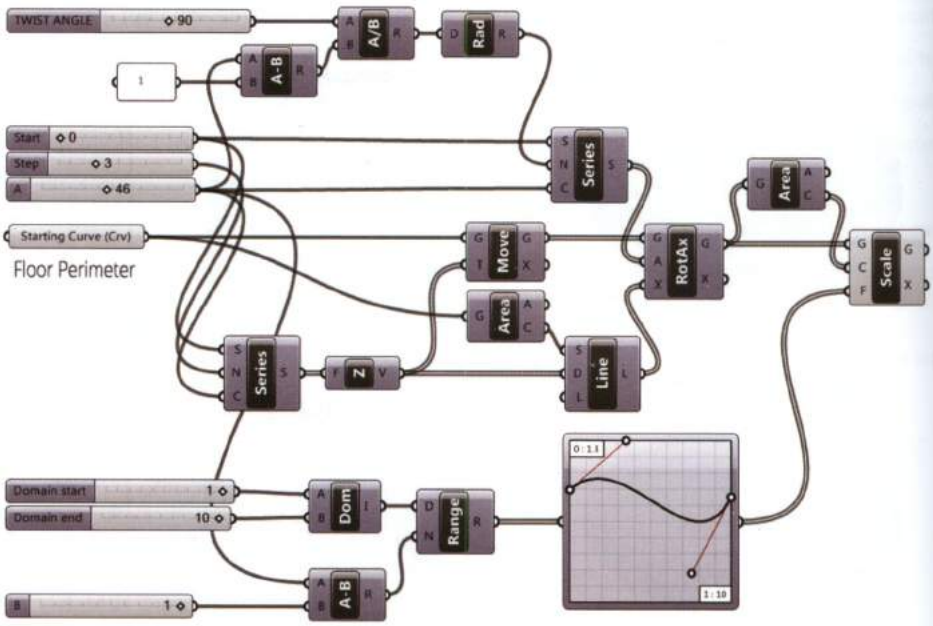


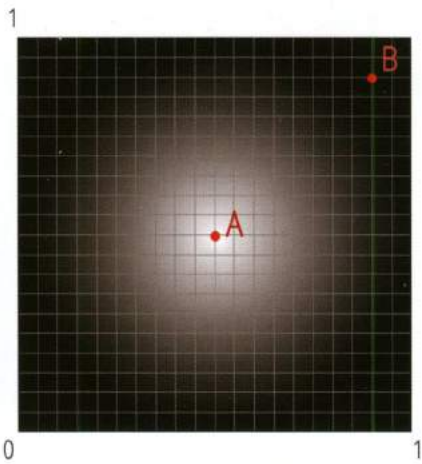
FIGURE 4.3

The image illustrate different configurations of a building model as the *Graph Mapper* changes, affecting the scale factor for each floor.

4.3.3 *Image Sampler* component

The component ***Image Sampler*** (Params > Input) is an input component which converts chromatic information of an image into numerical values.

Every rectangular picture can be imagined as a bidimensional domain ranging by default between 0 and 1. If a rectangular grid is superimposed onto the image each grid point P – defined by its UV coordinates in the LCS – provides an *intensity* value as an output.



point A

coordinates: (0.5, 0.5)

intensity: 1

point B

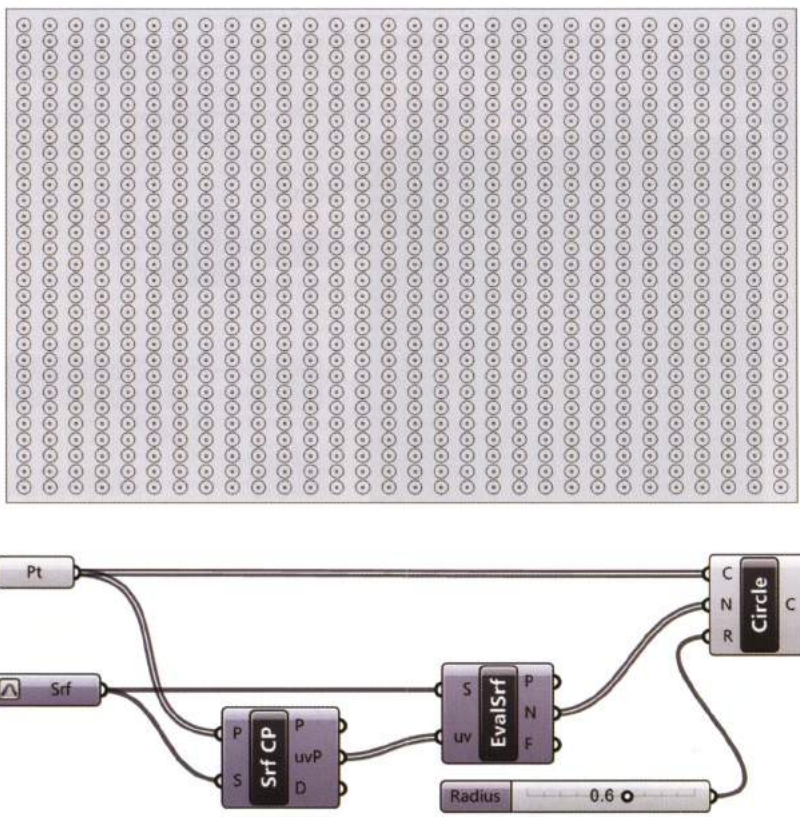
coordinates: (0.9, 0.9)

intensity: 0

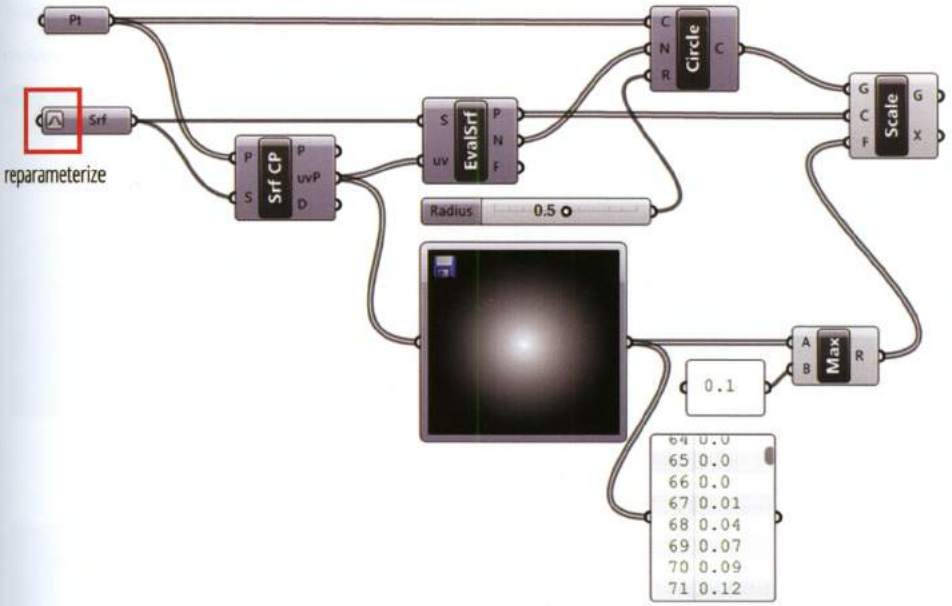
Intensity measures the color of the image at P dependent upon the specific *color model*: RGB, grayscale, etc., and outputs a value. For instance, the grayscale color model measures the intensity of a pixel and outputs values **ranging from 0 (black) to 1 (white) with fractional grayscale values in between.**

The *Image Sampler* editor – accessed by double left clicking on the component – is used to load an image, select a *color model* and specify whether to interpolate. The *Image Sampler* component requires a set of points with UV coordinates as input, and returns a list of numbers or *intensity* values.

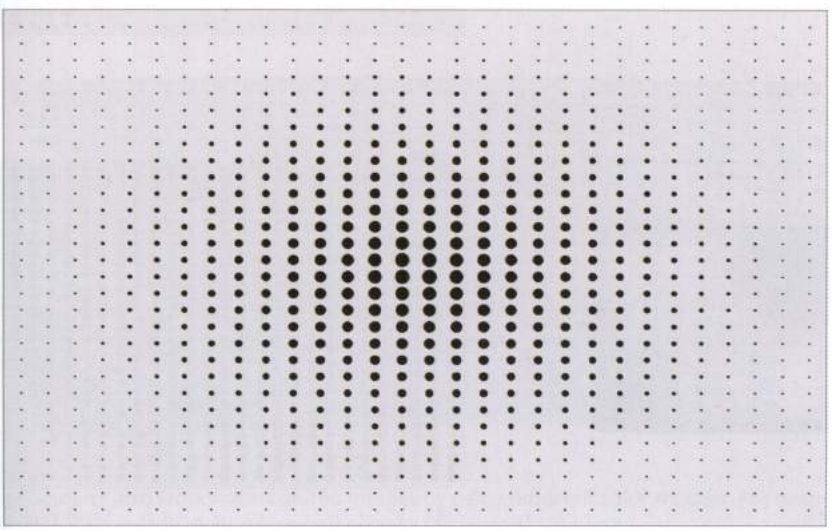
For example, a freeform surface and an array of points is set from Rhino. The component *Surface CP* is used to convert points coordinates from WCS to LCS, allowing the surface is evaluated at the uv resulting coordinates. The normal vectors (N) are connected to the N-input of *Circle CNR*. The radius of the circles is set to a constant value defined by a slider.



A geometrical pattern can be obtained by scaling the current circles using a scale factor that reduces geometries, i.e. a scale factor between 0 and 1. Since the output of the *Image Sampler* component is a list of *intensity* values ranging between 0 and 1, the *Image Sampler* output can be used as scale factors. To generate the scale factors the output (uvP) of the *Surface CP* component is connected to the input of the *Image Sampler*. The *Image Sampler* component outputs a list of values ranging between 0 and 1 according to the sampling of the set grayscale image. The *Image Sampler* output is connected to the scale factor input (F) of the scale component generating a pattern of circles. The initial surface must be reparameterized.



Images with pure black areas return null **intensity** values which are null scale factors. To cull null scale factors the component *Maximum* is used to replace every 0 with a different value such as (0.1).



The following images compare generated patterns to their corresponding grayscale images.

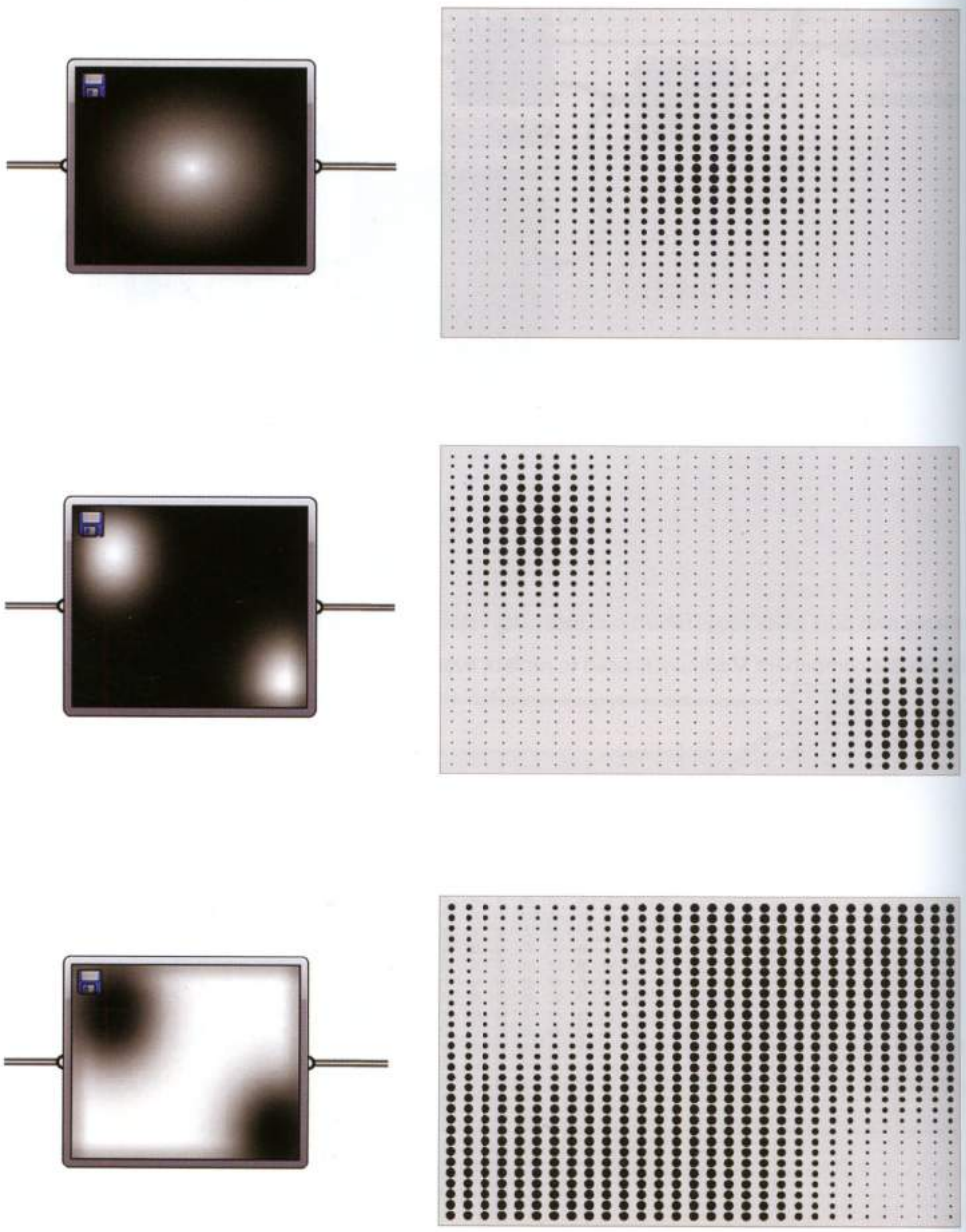
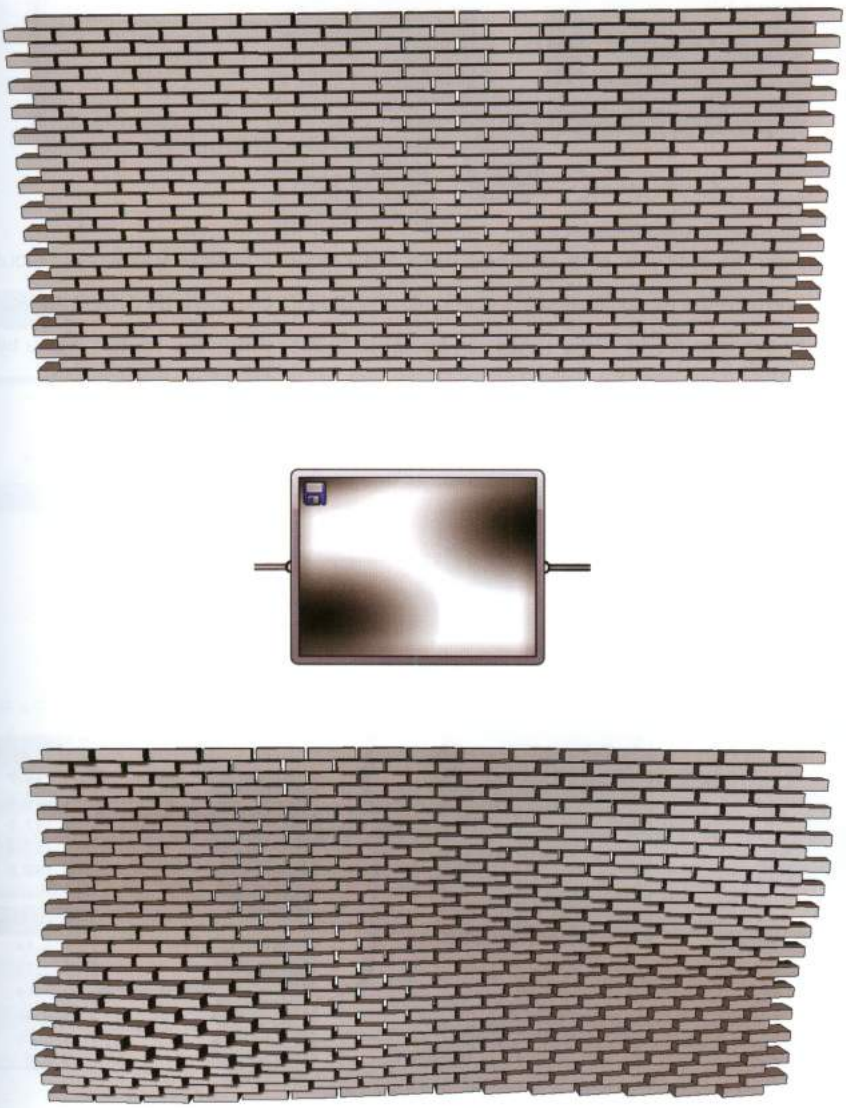


FIGURE 4.4

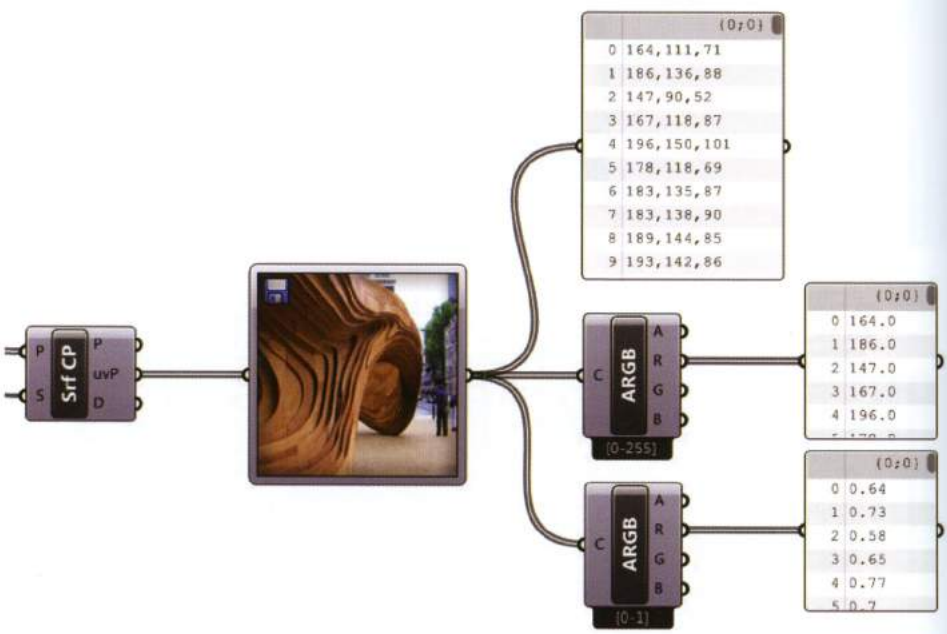
Different configurations corresponding to different grayscale images. Bright areas (close to white) return values close to 1 (no scaling), while dark areas (close to black) give values close to 0 (maximum reduction).

Below the results of the combination of the *Image Sampler* with different transformations. Each brick moves and rotates according to a grayscale image. The bricks' gravity points (whose coordinates were reparameterized between 0 and 1) were used as UV points for the *Image Sampler*.



The *Image Sampler* also works with colored images by using different color models. For example, The RGB color model returns a list of RGB values – or a triplet of values (channels) – ranging between 0 and 255.

RGB values can be split into four channels using the component *ARGB* (Display > Colour). The component returns the Alpha (A), Red (R), Green (G) and Blue (B) channels. By default *ARGB* outputs values range between 0 and 1, the range can be set to [0,255] within the context menu by right-clicking on the component and selecting *Integer Channels*.



The following example demonstrates how to generate a variable offset from a flat surface using a colored image. In particular, the algorithm offsets parts of the surface that correspond to the red areas of the image.

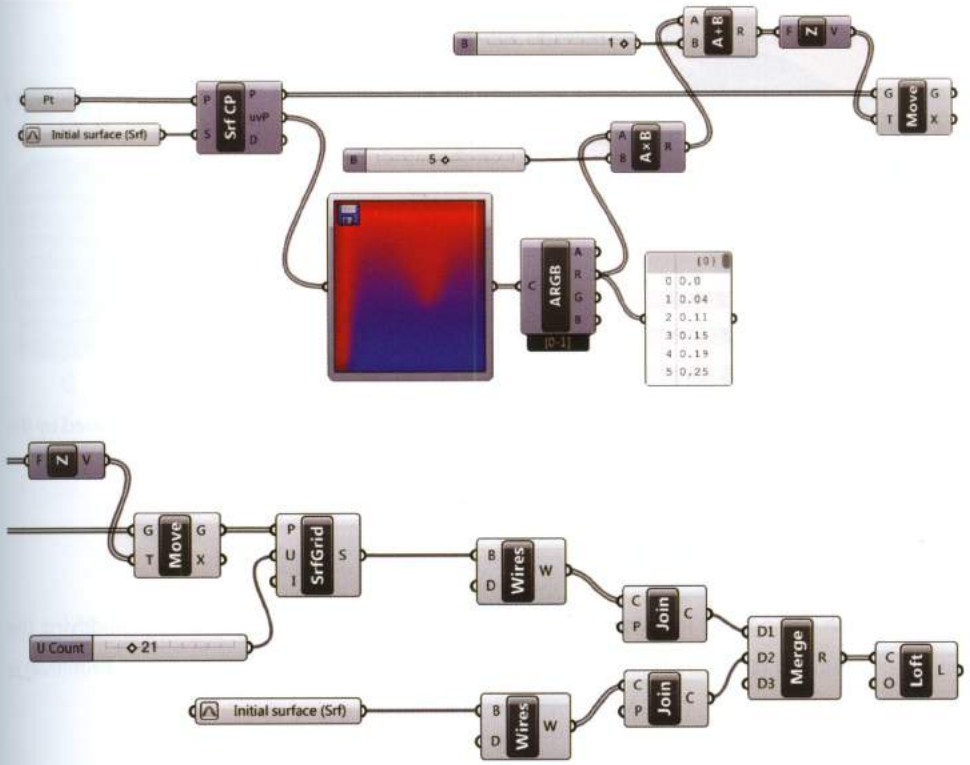
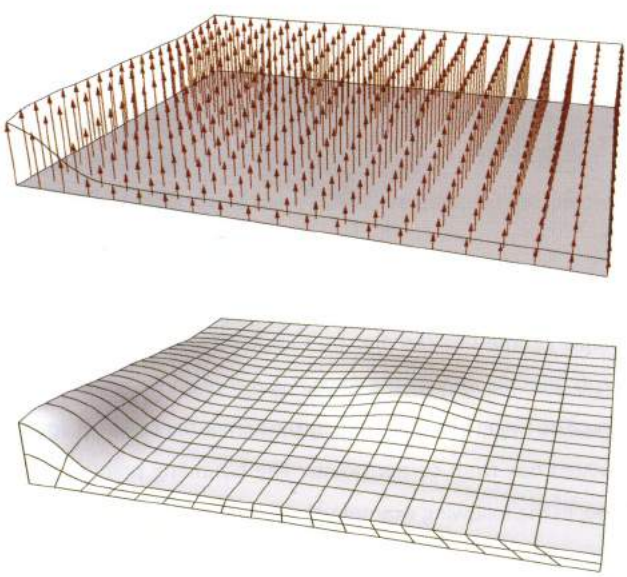


FIGURE 4.5
Variable offset obtained through the *Image Sampler*. The algorithm performs the offset for those parts of the surface corresponding to red areas in the image.



4.4 Other transformations: *Box Morph*

The component ***Box Morph*** (Transform > Morph) uses a reference "pliable" box to deform geometry to a target box. *Box Morph* is likened to Rhino's *Cage* and *CageEdit* commands.

The *Box Morph* requires three data inputs:

1. **A geometry to morph (G)**

   A geometric entity or a set of geometric entities;

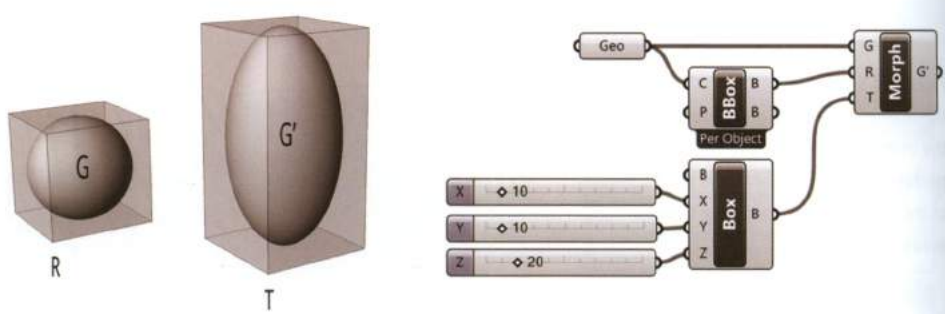
2. **A *reference box* (R)**

   A box which encompasses the geometry to morph. The *reference box* can be defined by the *Bounding Box* component (Surface > Primitive);

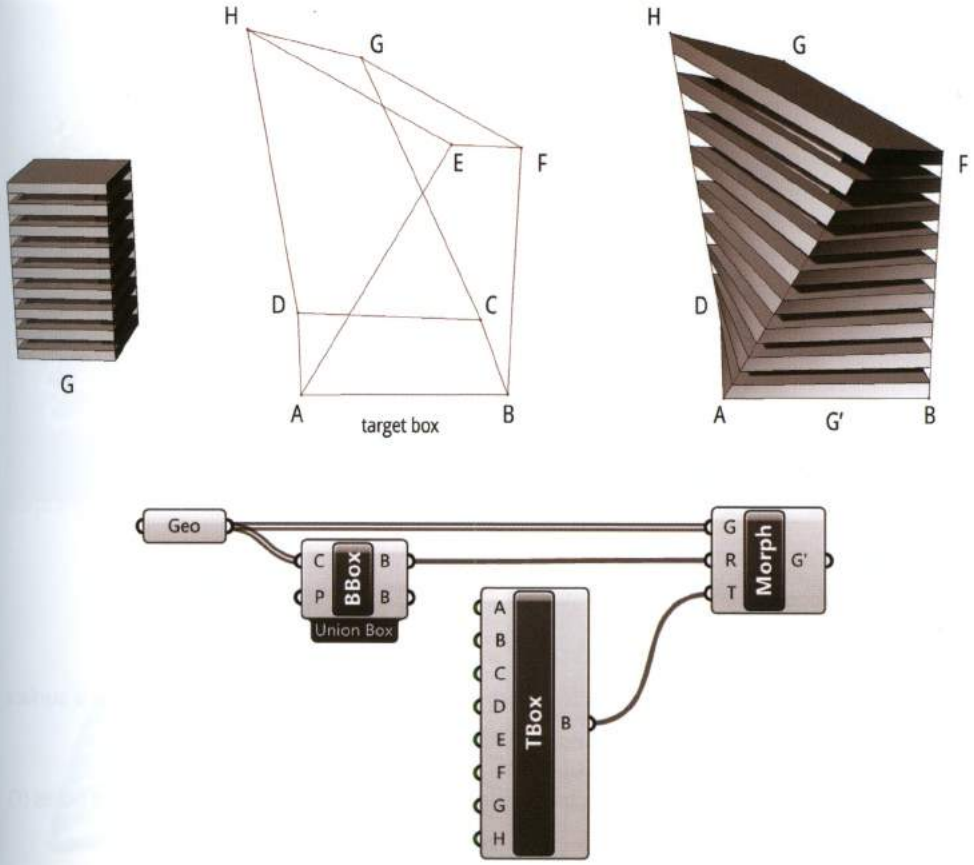
3. **A *target box* (T)**

   Any box that can be created using Grasshoppers box-components.

The *Box Morph* component transforms the *reference box* (R) into the *target box* (T) modifying the contained geometry (G). In the following image, the component *Center Box* (Surface > Primitive) is used as the *target box*.



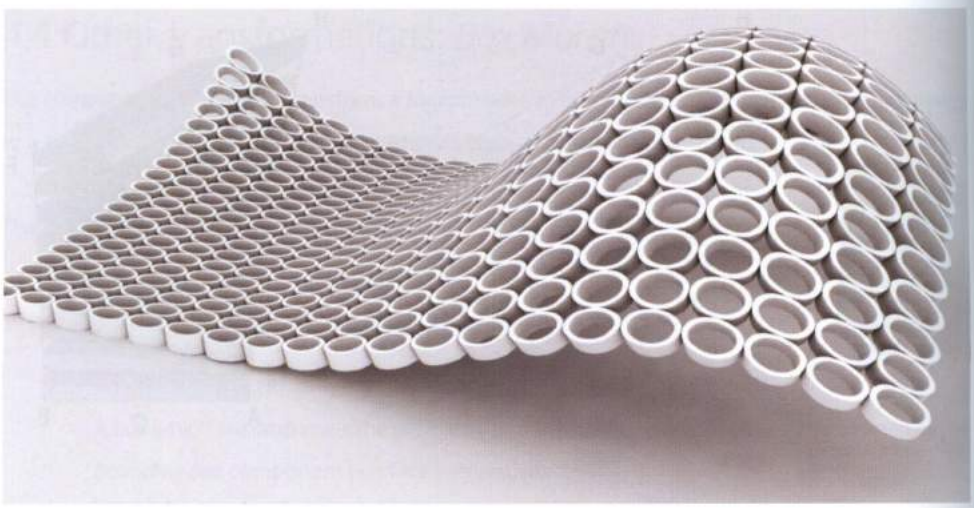
Multiple geometries can also be simultaneously morphed to primitive or non-primitive boxes. To morph multiple objects the *Bounding Box* must be set to *Union Box* mode through the contextual menu. The target box is defined by the component *Twisted Box* (Transform > Morph) which generates a twisted box from corner points. The bounding points can be defined in Grasshopper or set in Rhino. In the following example, the points are set from Rhino using the *Twisted Box* local setting.



In this case, multiple geometries are set from Rhino using the *Geometry* component. The *Bounding Box* component set to *Union Box* mode encompasses the geometries. The *target box* is created by setting eight points drawn in Rhino using the local setting, if these points are adjusted the geometry will deform accordingly.

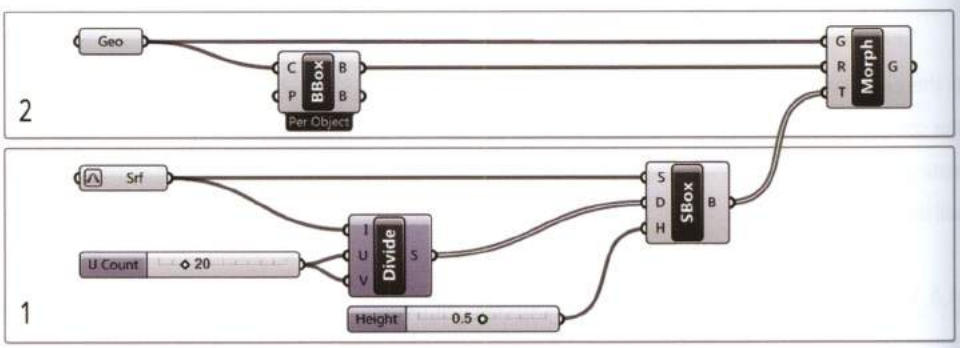
4.4.1 *Paneling*

Paneling extends the *Box Morph* logic to multiple *target boxes*. Given an arbitrary surface with a set of twisted *target* boxes a surface-panelization can be defined using a single or a set of merged geometries.

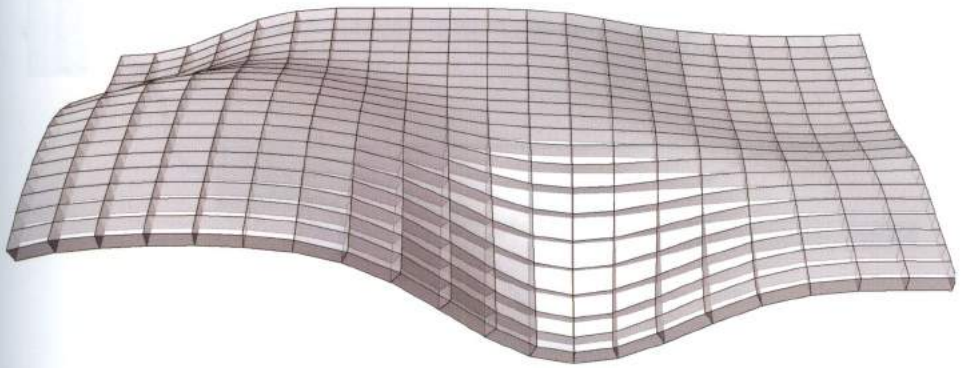


The *paneling* procedure consists of two steps:

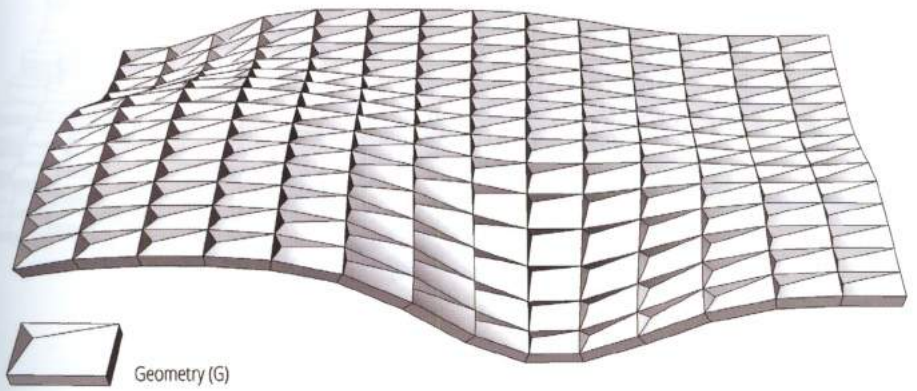
1. Creating a set of twisted boxes (T) on an arbitrary freeform surface.
   The component *Surface Box* (Transform > Morph) generates twisted boxes on a surface with a defined bidimensional domain (D) and a height (H).
2. Morphing a geometry (G) – encompassed by a *Bounding Box* (R) – to twisted *target* boxes (T).



For example, a grid of 20 x 20 surface boxes – as specified by the bidimensional domain – with a height of 0.5 units are created on the surface using the component *Surface Box*. The *Surface Boxes* are the *Target Boxes* for morphing.



Morphing manipulates the base geometry (G) to match the target boxes (T), creating panels.



If random values are connected to the H-input of *Surface Box* different heights for the target boxes will result; yielding a panelization of random height base geometries (see following images).

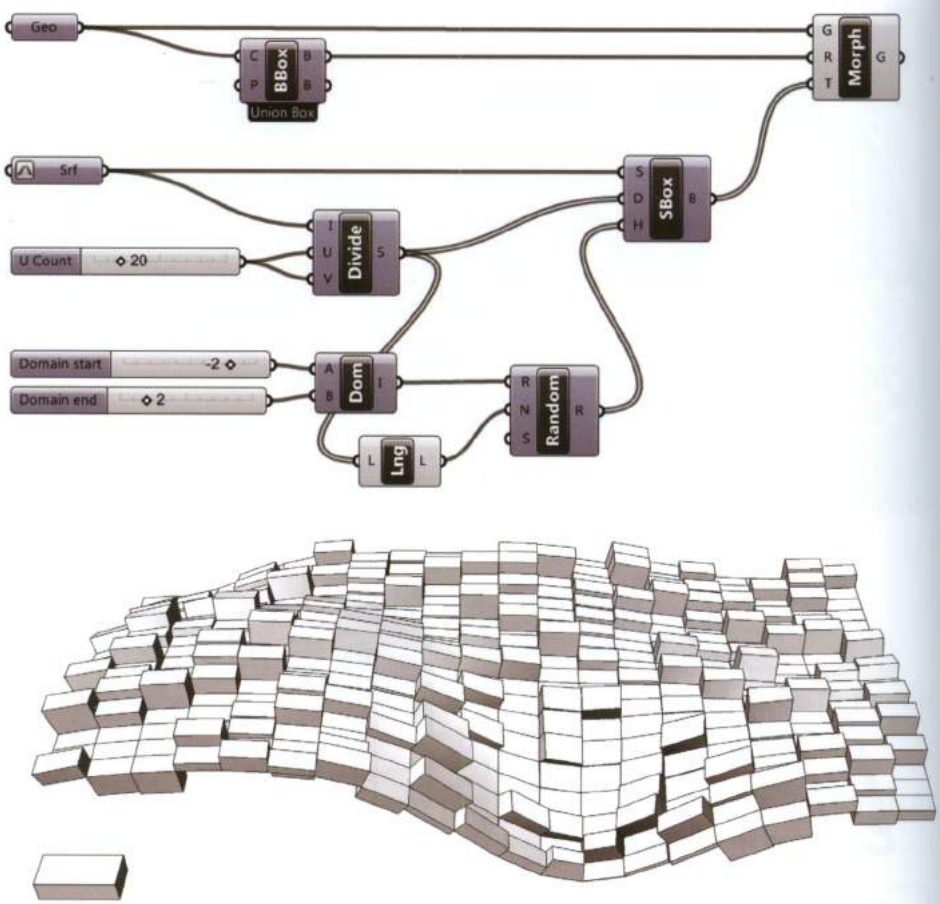


FIGURE 4.6

Interesting output can be achieved using a non constant value for the H-input of *Surface Box*.

Herzog & De Meuron, Beijing National Stadium (Bird's nest). Picture by Jorge Láscar.

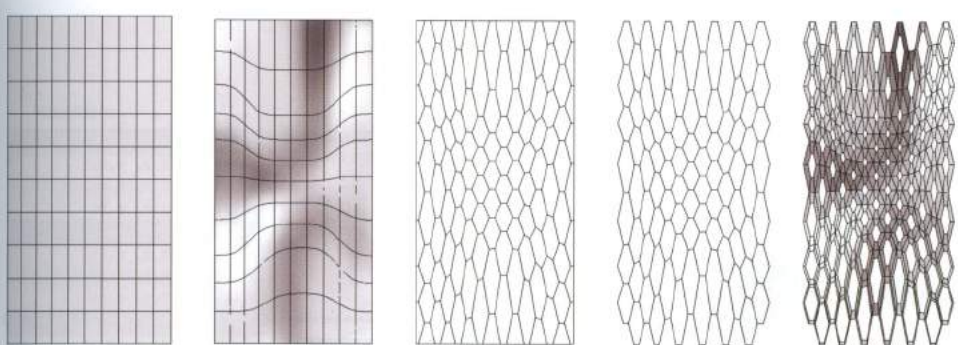
5_skins

advanced data management

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

Kevin Kelly

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



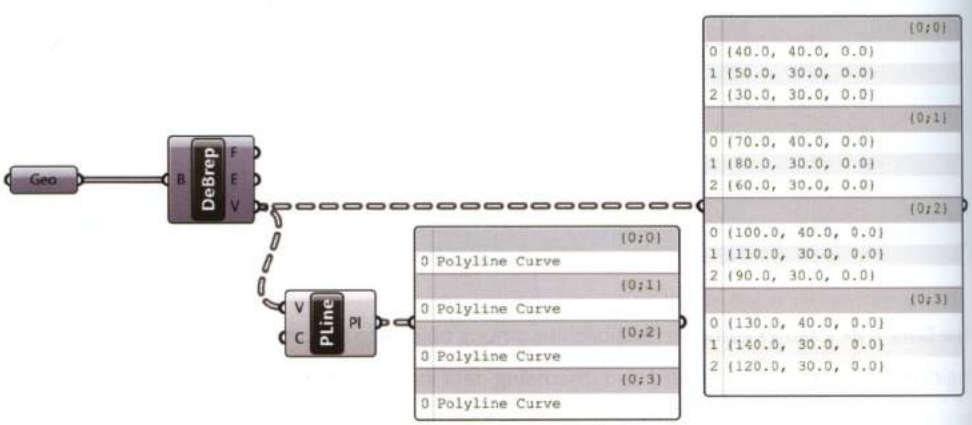
The Data Tree enables complex relationships among objects to be formed. Thus far, computational logics have been straightforward and discussing data structure was unnecessary. This part of the text will elaborate on the basic concepts of using data structure to control form.

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

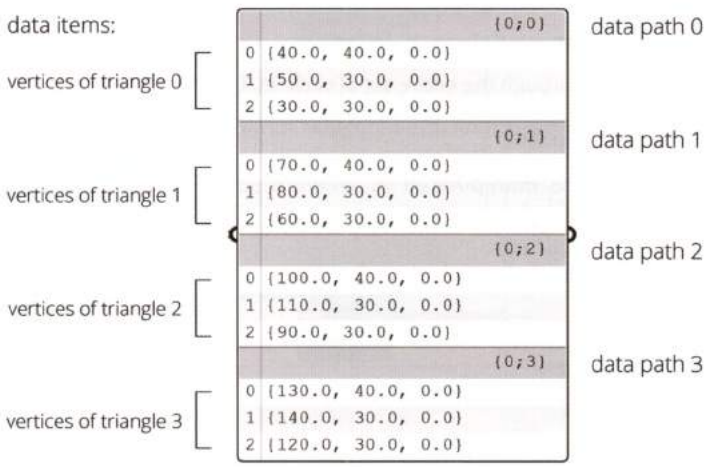
The result differs from the desired final output. The defined logic did not create a single polyline through the twelve points. Instead as expected, the data structure defined four polylines each with three vertices.

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

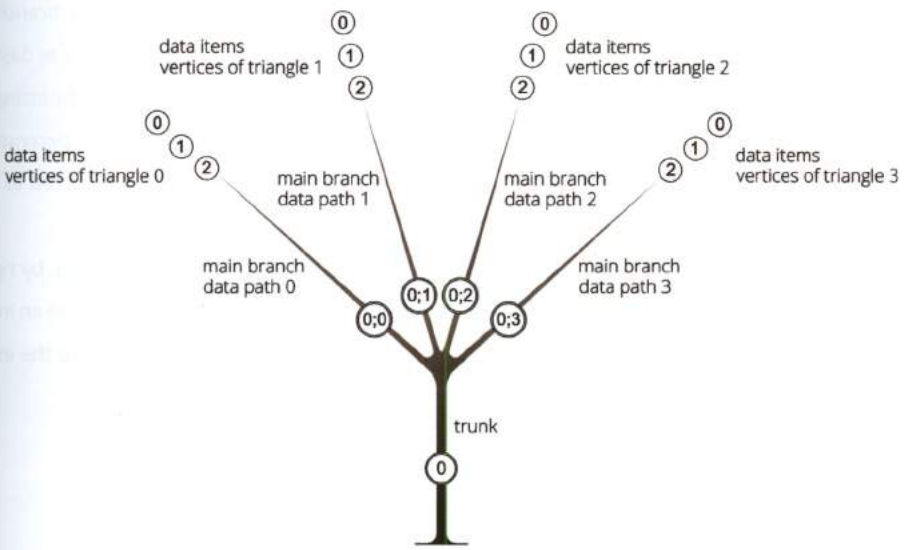
1. The outgoing wire generated by the *Deconstruct Brep* component is *dashed;*
2. Data within the lists are split into four subsets.



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



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



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

Data Trees follow two fundamental rules:

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

5.1 Manipulating the *Data Tree*

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

5.1.1 *Flatten Tree*

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

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

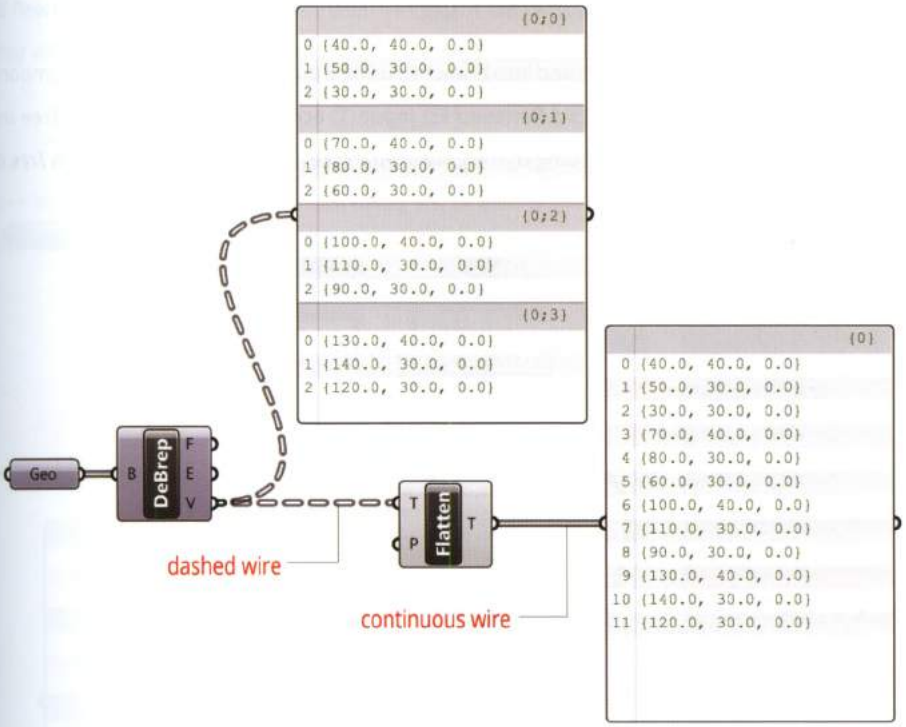
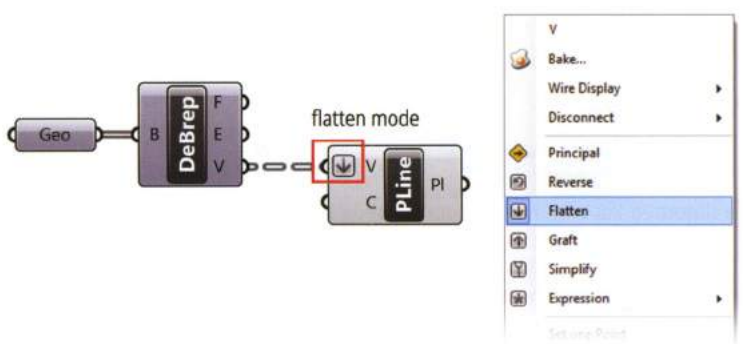
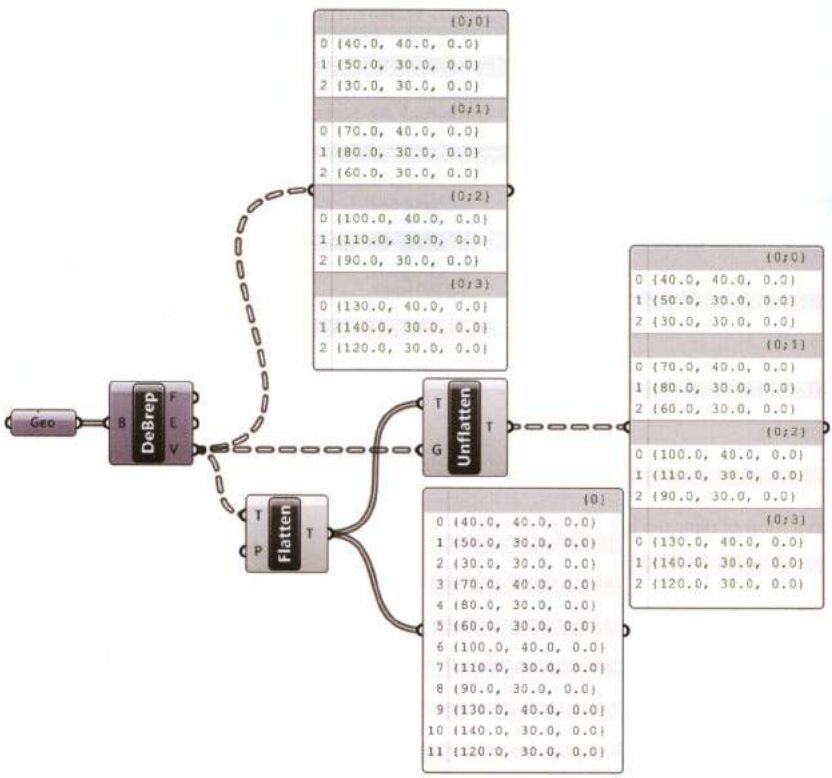


FIGURE 5.1
A "structured" flow of data can be identified by a dashed wire, while a flattened flow by a continuous line.



5.1.2 *Unflatten Tree*

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

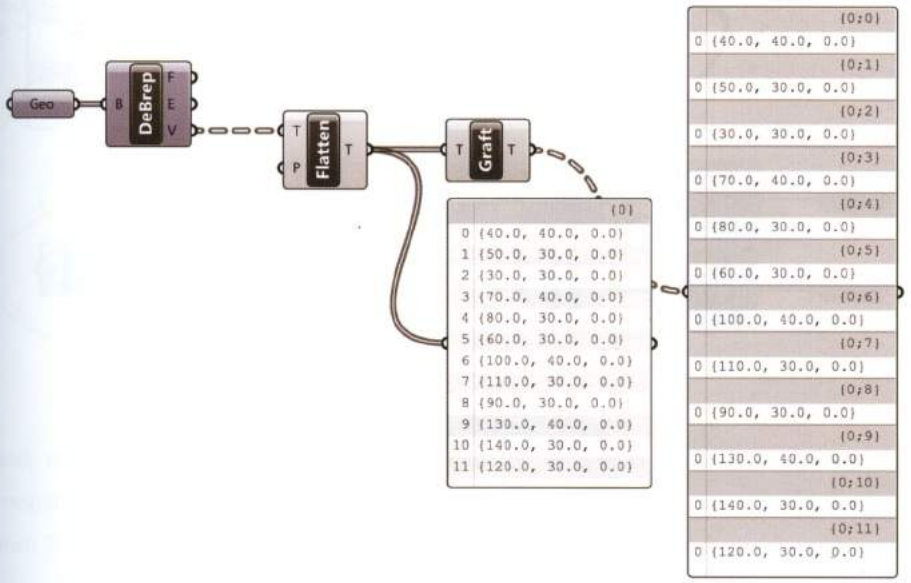


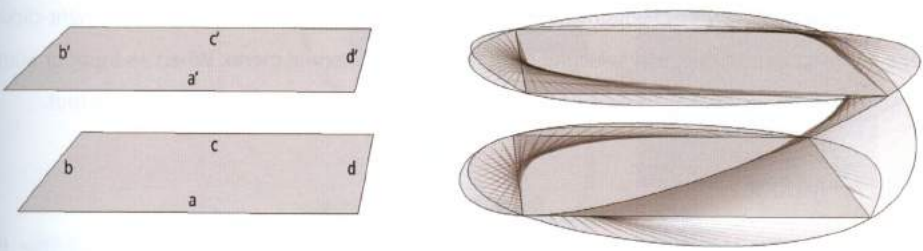
5.1.3 *Graft Tree*

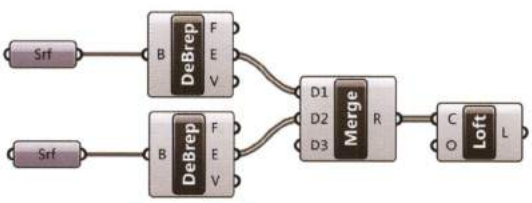
The component ***Graft Tree*** (Sets > Tree) creates a branch for every item within an arbitrary list. Accordingly, a flattened list with N items connected to the T-input of the *Graft Tree* component will return a new list with N branches, one branch for each item.

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

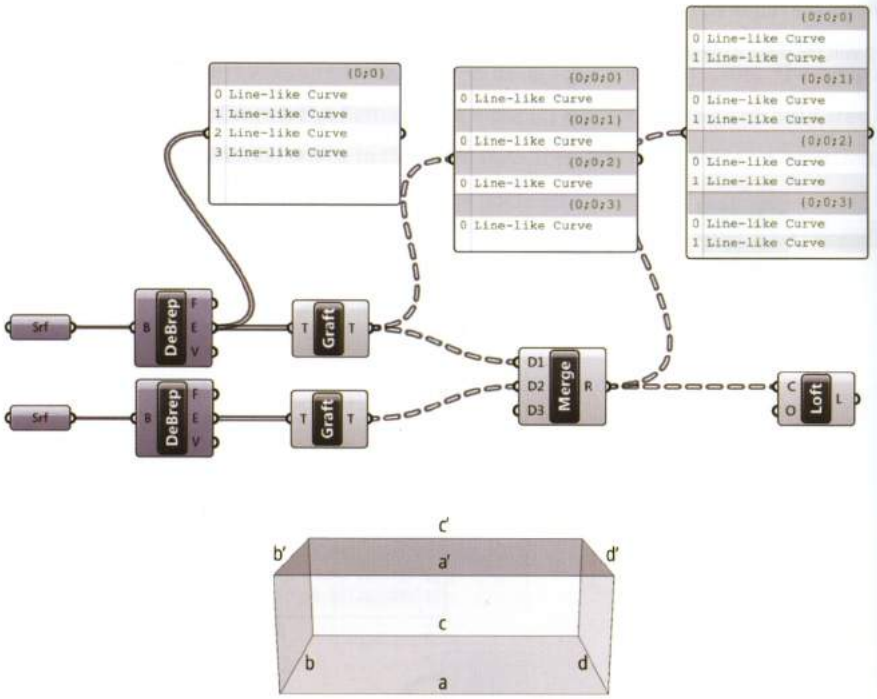
data flows are required to be grafted, then merged. If the two data flows are *merged* and not *grafted*, *Lofting* will be performed through the entire set of curves according to the order: a'-b'-c'-d'-a-b-c-d.







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

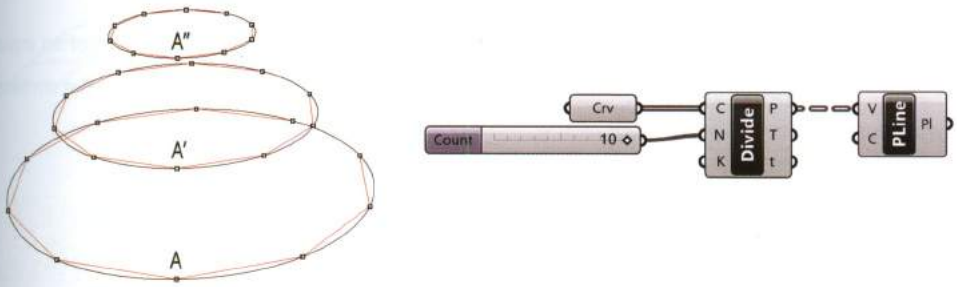


Every component provides the possibility to internally *Graft* incoming or outgoing data by right-clicking on the specific input/output and selecting *Graft* from the contextual menu. When an input or output data stream is being *grafted* an upwards arrow symbol will appear next to the input or output.

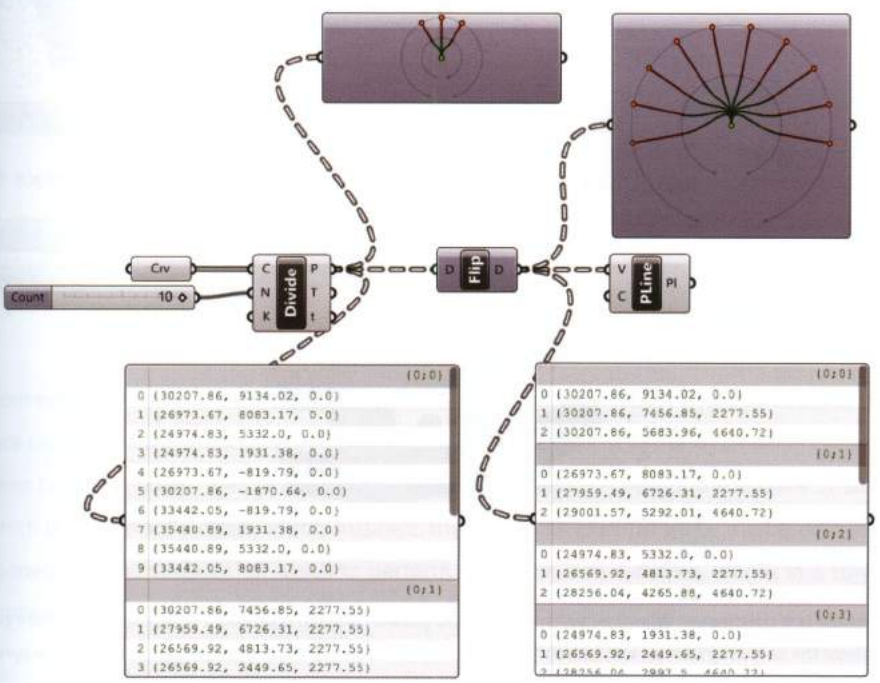
5.1.4 *Flip Matrix*

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

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



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



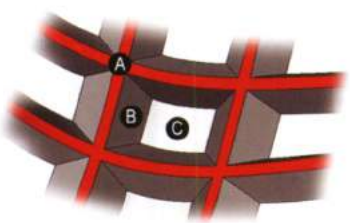
5.2 Skins

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

5.2.1 Rectangular based pattern

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

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

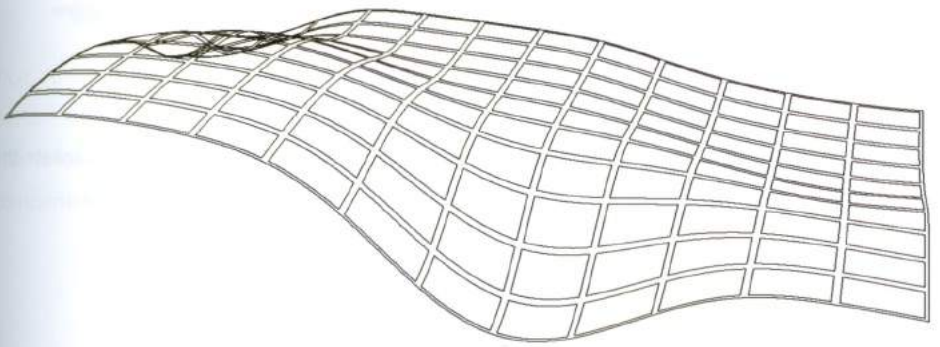


NOTE 11

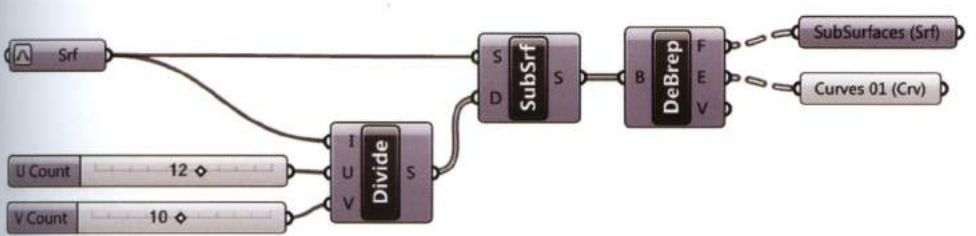
A *prismatoid* is a polyhedron where all vertices lie in two parallel planes. A *prismoid* is a *prismatoid* where parallel planes host the same number of vertices and lateral faces are trapezoids. An example of *prismoid* is the *pyramidal frustum*.

SURFACES A

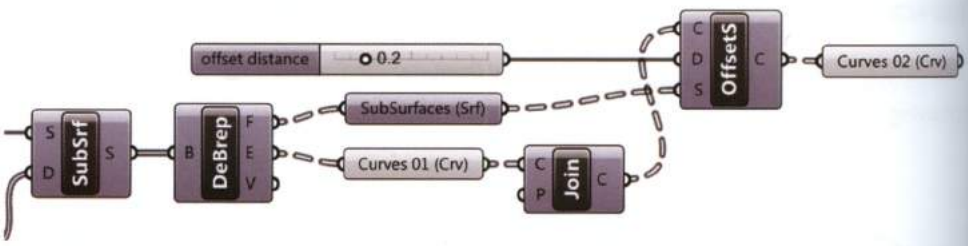
The surfaces denoted as "A" frame the *pyramidal frustums;* the frame's edges are displayed in the following image.



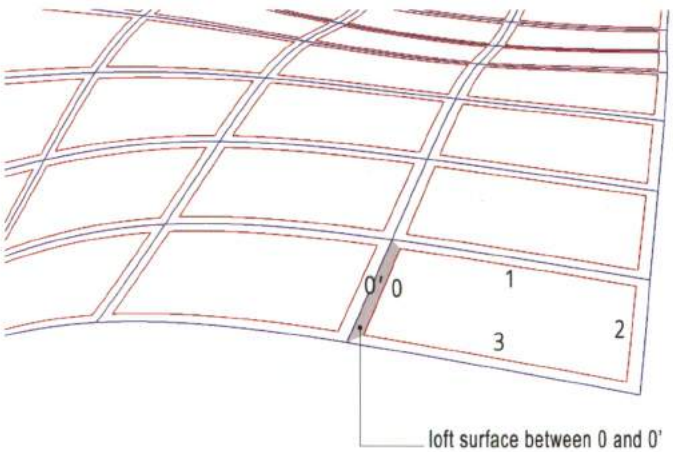
The first step in the definition is to set and reparameterize a surface from Rhino using the *Surface* "container" component. Then the *Isotrim-SubSrf* output is deconstructed using the *Deconstruct Brep* component to extract the defined faces and edges. To easily identify the *Deconstruct Brep* output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



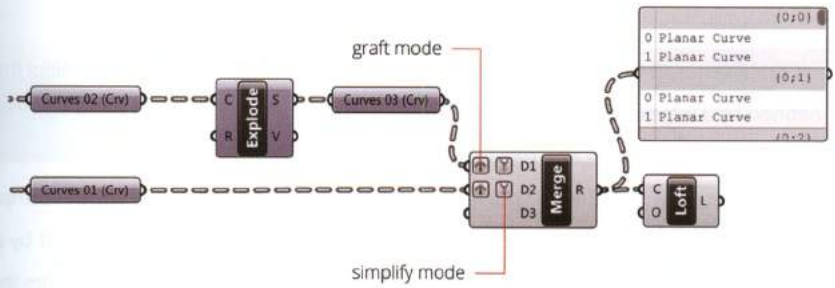
The four edges of each sub-surface are joined, forming a polyline which is offset coincident to the set surface using the component ***Offset on Srf*** (Curve > Util). The C-input of the *Offset on Srf* component requires curves to offset, the D-input the **offset distance**, and the S-input the surface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



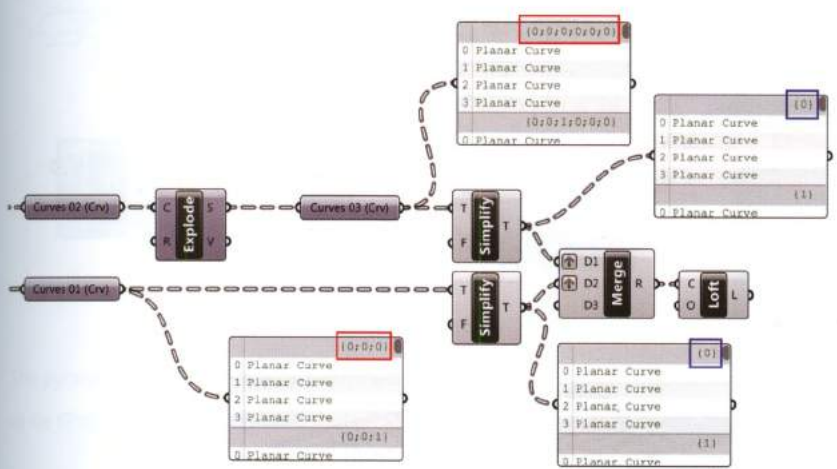
The following image depicts the *red* offset curves and the *blue* sub-surface edges. To create the frame surfaces a **loft** operation is performed between corresponding curves (e.g. 0 to 0').



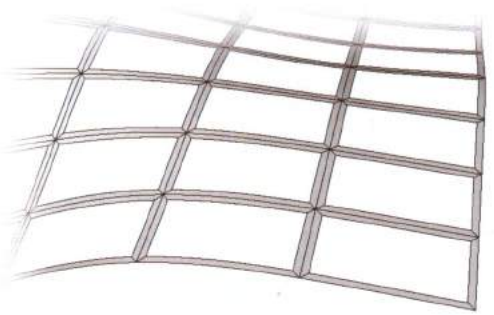
Since the output (C) of the *Offset on Srf* component stored as *Curves 02* are joined polylines, the curves must be exploded to define a loft operation between corresponding curves. The component *Explode Curve* (Curve > Util) explodes the polyline to define four edges. The output (S) of *Explode Curve*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



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



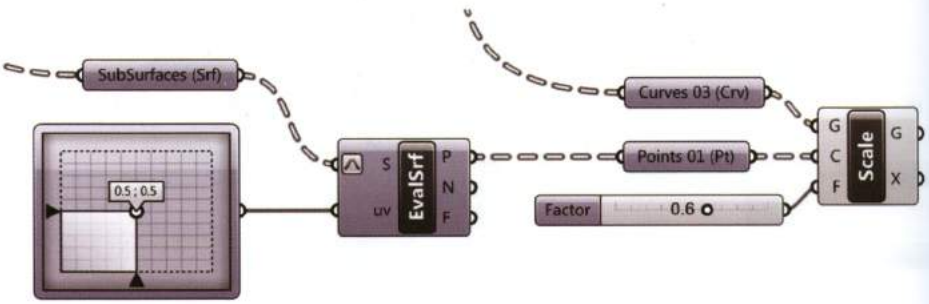
The properly formatted data is lofted returning the set of surfaces "A". If the offset distance input (D) of the *Offset Srf* component is modified the frames width will parametrically change.

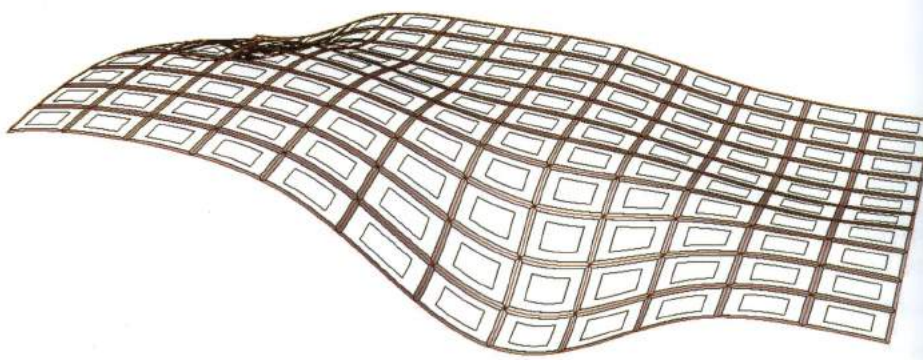


SURFACES B

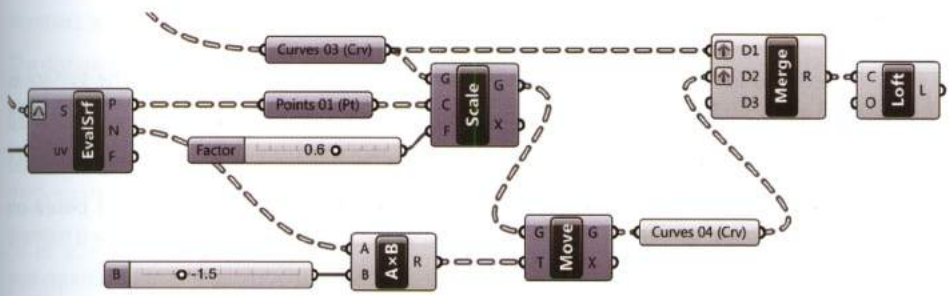
The surfaces denoted as "B" form the lateral faces of the *pyramidal frustums*. The *pyramidal frustums* faces are defined by a loft between data set *Curves 03* and an additional data set *Curves 04*. The data set *Curves 04* is defined by scaling and moving *Curves 03*.

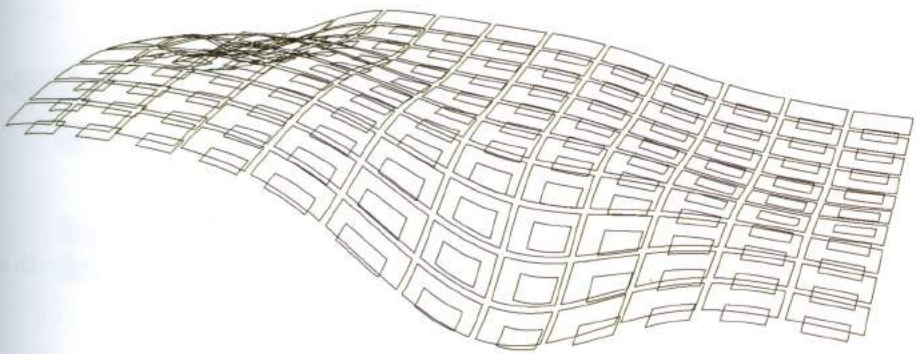
To define the data set *Curves 04*, the mid point of each sub-surface is calculated using the component *Evaluate Surface*, with the input slot (S) set to reparameterize, and the input (uv) defined by an *MD Slider* set to (0.5;0.5). The *Evaluate Surface* component output (P) calculates a set of points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





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

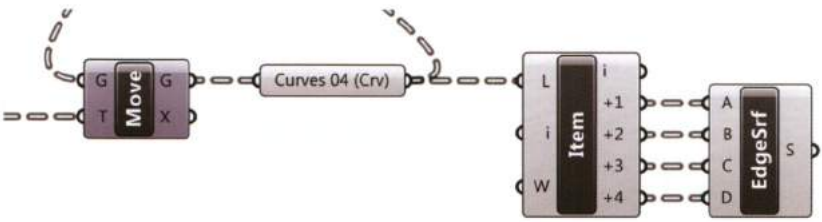




The *pyramidal frustums* faces are defined by lofting *Curves 03* and *Curves 04*. The curves are required to be *Grafted* but not simplified.

SURFACES C

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



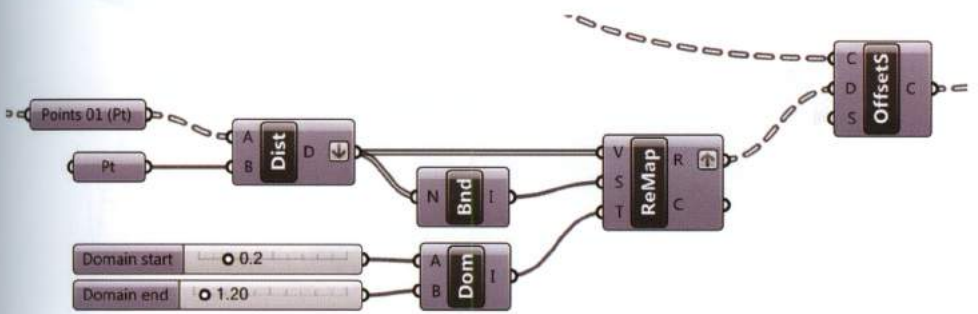
The desired output can be manipulated parametrically by changing the input parameters, such as: the offset distance, the scale factor, the translation factor or even the initial surface.

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

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

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

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

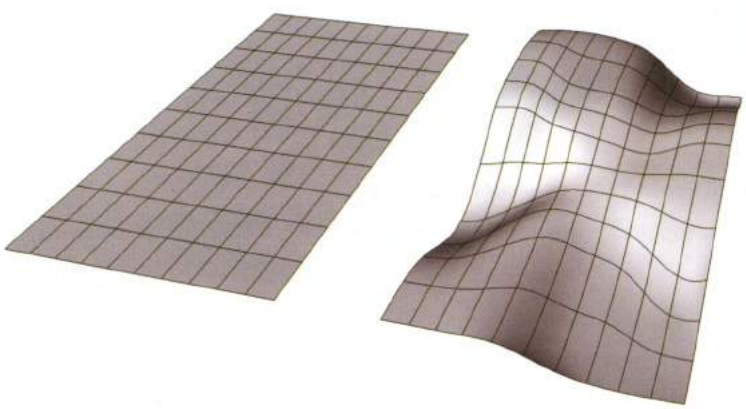


The resulting responsive frame is controlled by an external point. As the distance between the attractor point and the sub-surfaces decreases the frames width will also decrease.

5.2.2 Hexagonal based pattern

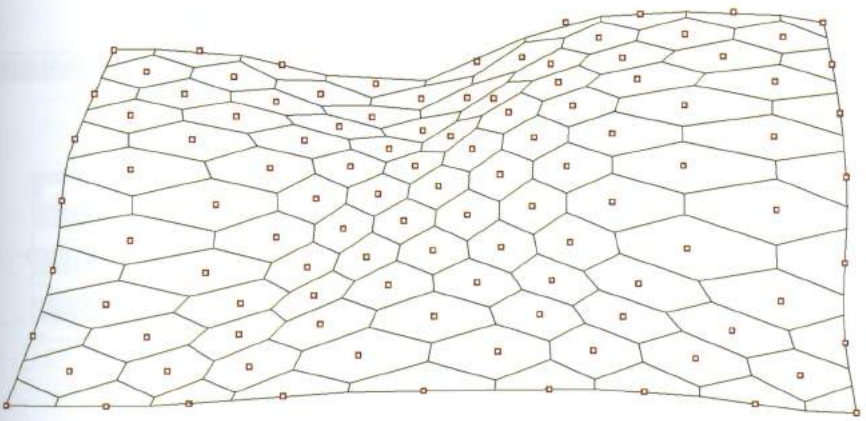
The second example will create a tridimensional pattern on a surface by means of a hexagonal subdivision. The procedure is similar to the first example, however there are several pattern based differences.

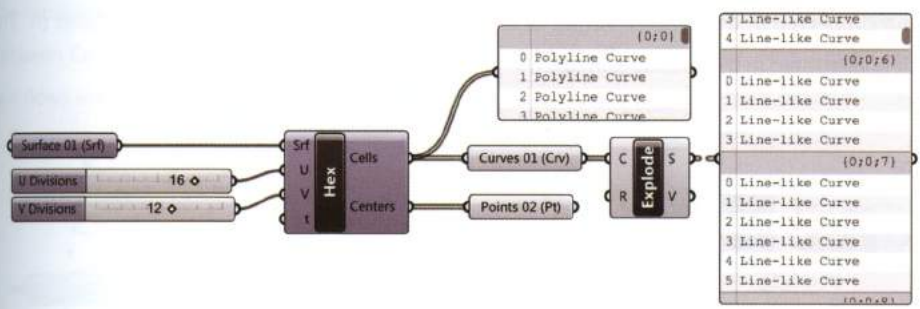
The first difference is that initial surface has an uneven grid of isocurves. The uneven set of isocurves will result in a non-regular hexagonal grid, with smaller hexagons around the surfaces center and larger hexagons near the surfaces edges.



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

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



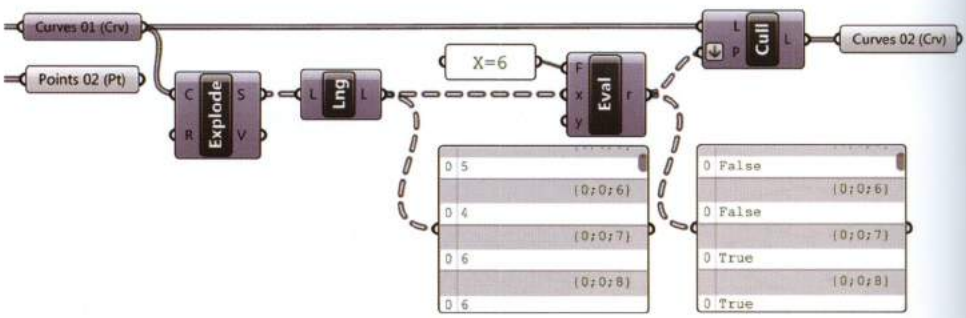


The first step in the definition is to cull the non-hexagonal cells (i.e. cells with less than six sides) from the data set defined by the *Hexagon Cells* output stored in the container component *Curves 01*. To remove the cells with less than six sides, a conditional Boolean statement is defined.

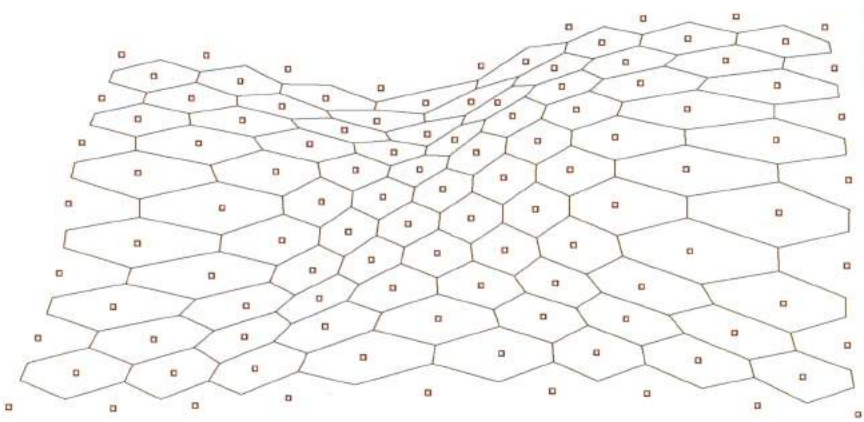
To count the number of sides comprising each cell, the cells are exploded using the component *Explode Curve*. The list length is calculated for each branch of the Data Tree using the *List Length* component. The resulting lengths: four, five or six, are tested against the condition X = 6 using the

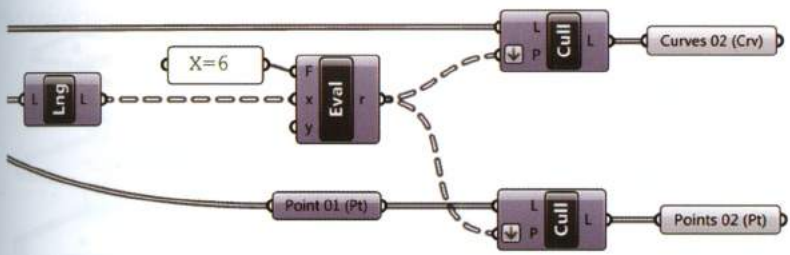
Evaluate component. The resulting output (r) of the *Evaluate* component will be a list of Boolean statements which define the P-input of the *Cull Pattern* component. The culling list input (L), or list to cull, is set as *Curves 01*.

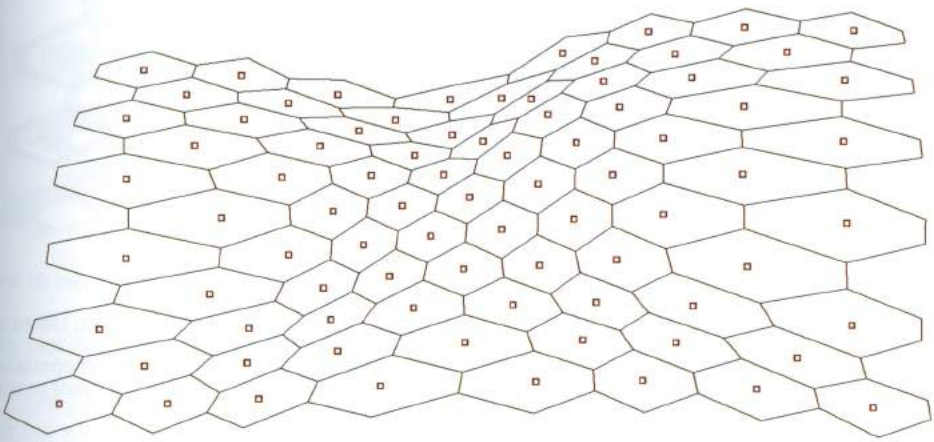
The data contained in the *Curves 01* container component has no branches, only a trunk; this is graphically visualized by the continuous wire. While the output of the *Evaluate* component is a Data Tree with multiple braches, graphically visualized by the dashed wire. In order to match the data structure,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



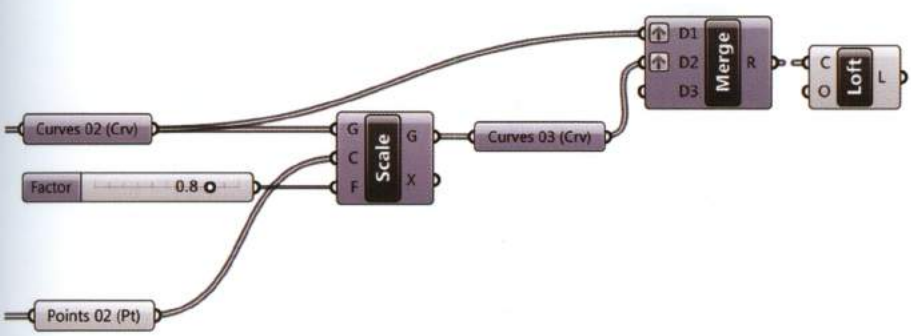
Similarly, the centers of the non-hexagonal cells are culled by another *Cull Pattern* component; with L-input connected to the Centers-output of the *Hexagon Cell* component collected as *Points 01*. The result is illustrated in the following images.

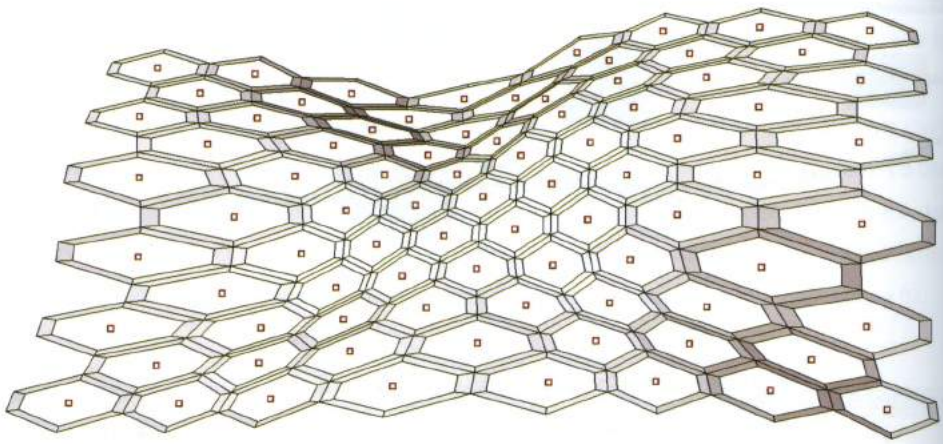




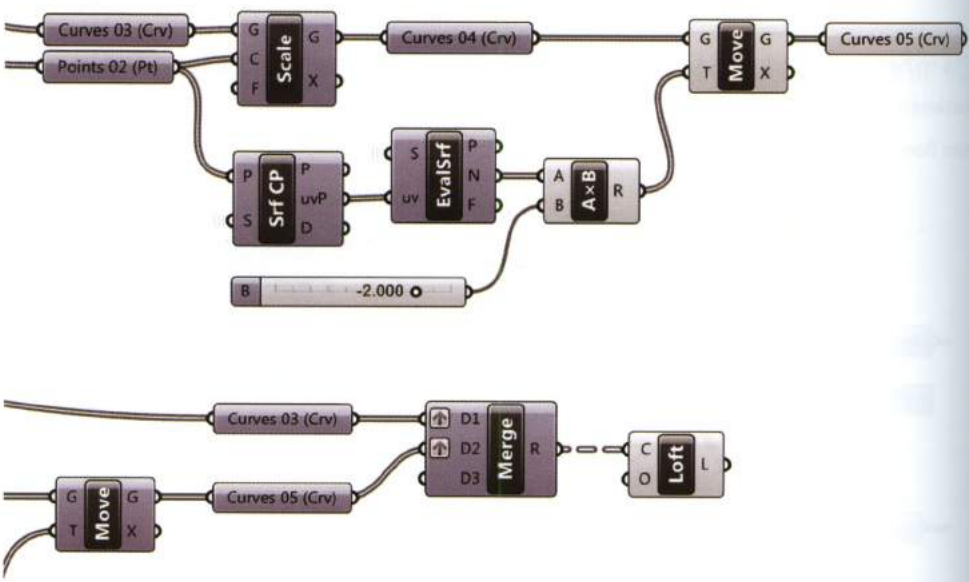


The remaining procedure is similar to the first example. The hexagonal frame is achieved by lofting between *Curves 02* and *Curves 03*. *Curves 03* are obtained by scaling *Curves 02*. Also in this case the two flows are grafted before merging.

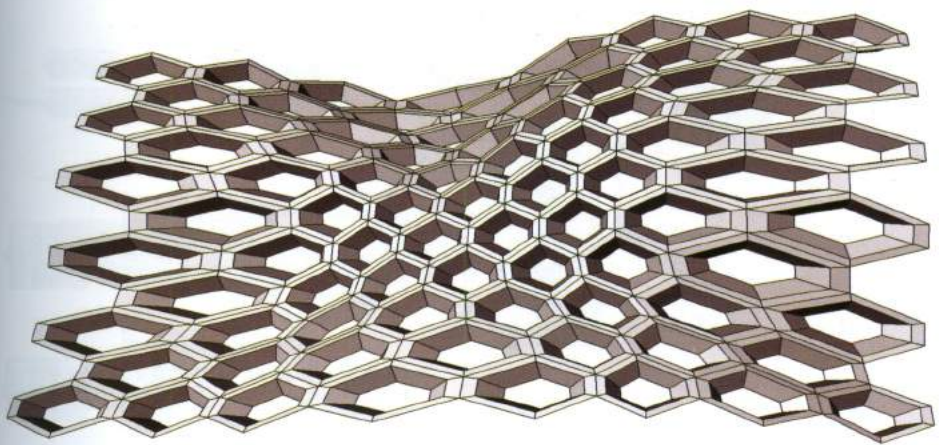




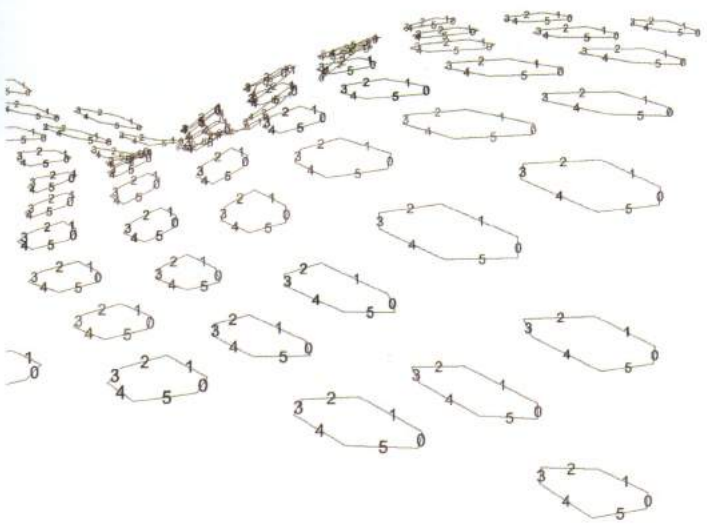
The next step is to generate the lateral faces of the hexagonal frustums; by scaling *Curves 03* and then translating the scaled curves according to the surface normals. The resulting curves, stored as *Curves 05*, are the lower edges of the hexagonal frustums. To create the lateral faces a loft operation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



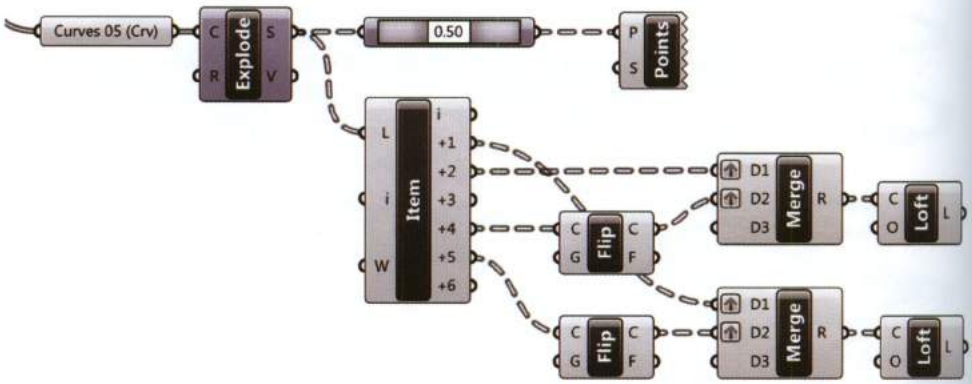
The result is illustrated below.

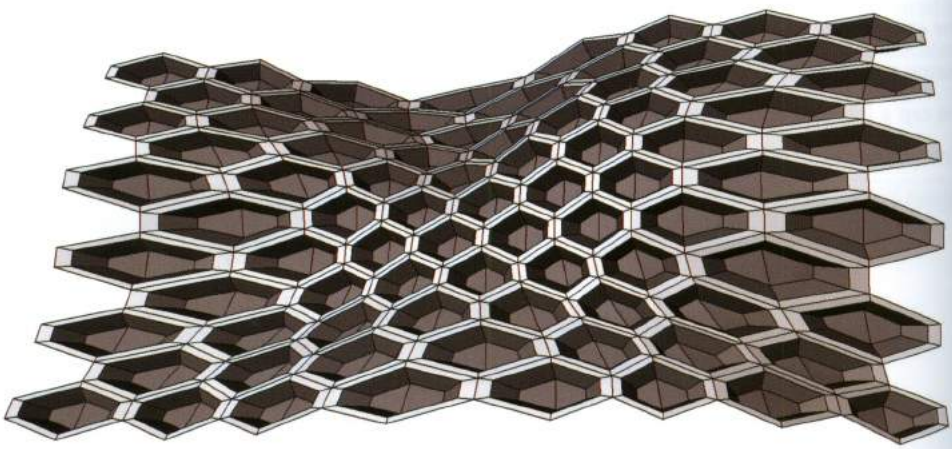


The caps are defined differently from the previous example. The caps have six edges and for this reason they cannot be created as a single surface without using the *Patch* component. To generate the cap a loft operation is performed between curves 2-4 and 1-5 respectively (see image below). Since the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

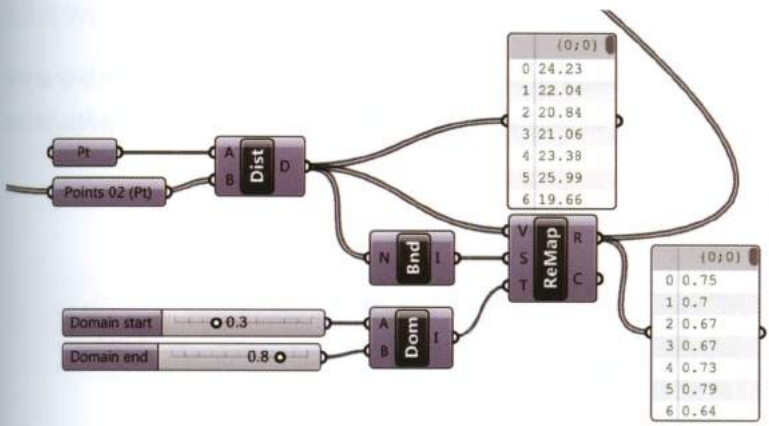


For this reason, curves 1 and 2 must be flipped using the component *Flip Curve*. The following images show the complete sequence and the final geometry.

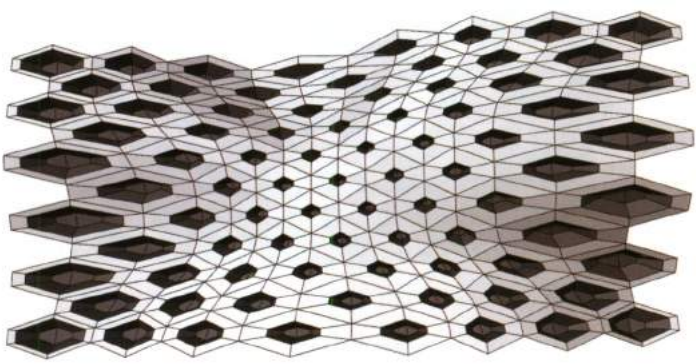




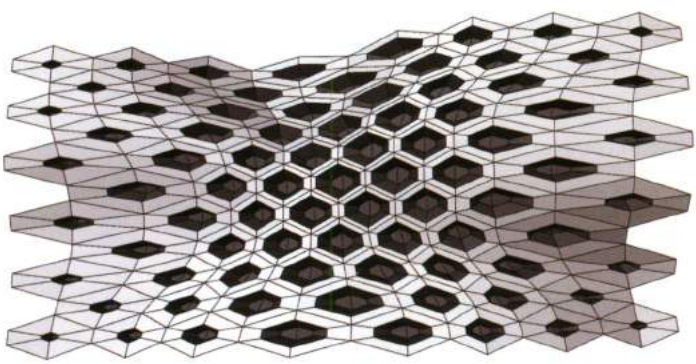
Similar to the previous example, the frames width can be controlled by an attractor point. In this case, the remapped distance values are used as scale factors for the first *Scale* component. The two *Domain* values A and B are the minimum and maximum frame width respectively.



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



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



5.2.3 Further Study: Responsive facade

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

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

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

1

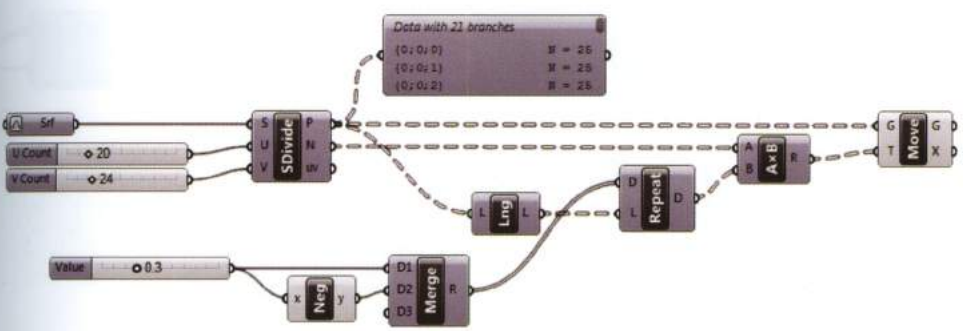
NOTE 12

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

5.2.4 Weaving

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

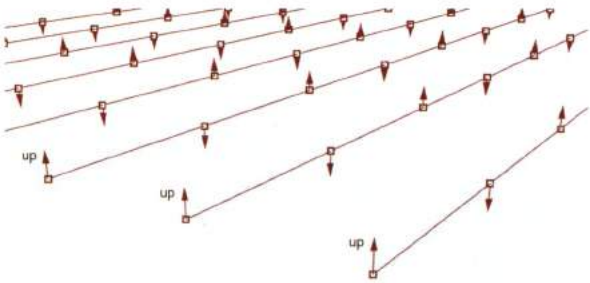
In general, digital weavings are defined by translating a grid of points according to alternating positive and negative surface normal vectors. The first step in the weaving definition, is to generate a grid of points on a set reparameterized surface using the *Divide Surface* component. The defined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



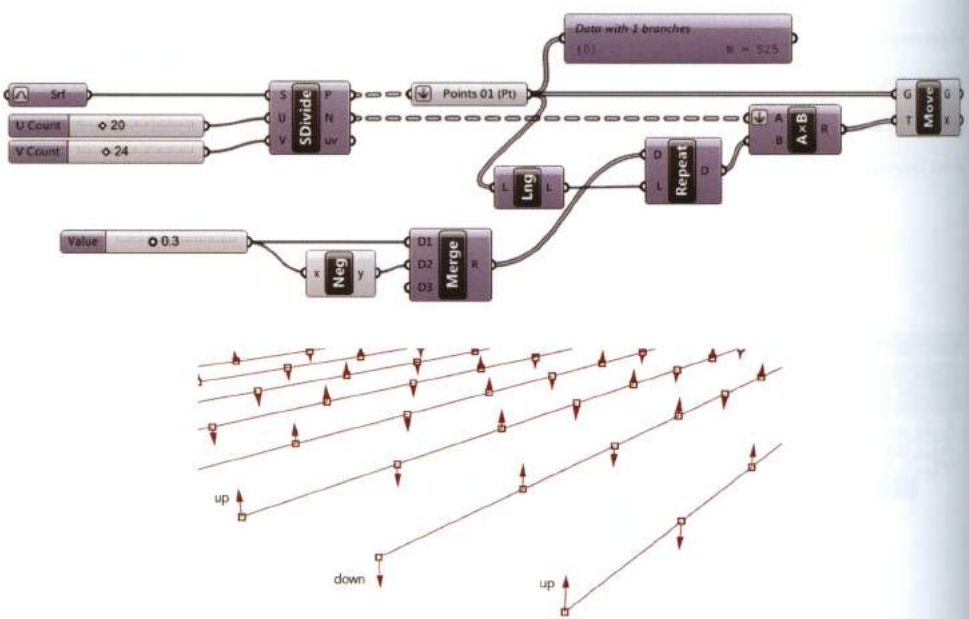
WARP

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

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

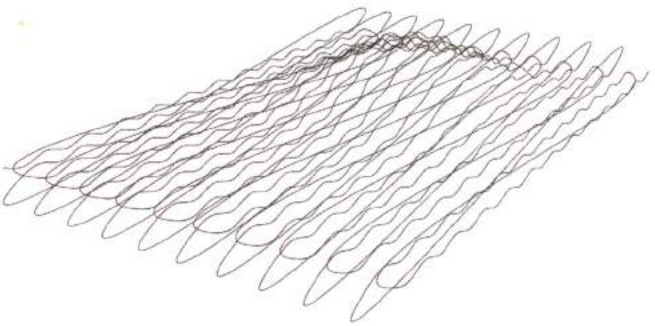


For this reason the P-output of Divide Surface is required to be flattened. The flattened list of points *Points 01*) are then translated using the *Move* component. The translation vectors are defined by multiplying a repetitious scalar list of alternating positive and negative values by a list of normal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

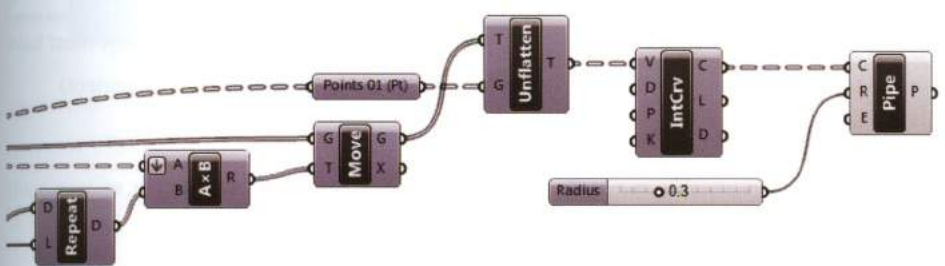


The weaving *warp* is created by interpolating a line through the G-output of the *Move* component using the *Interpolate* component. Since the points data structure has been flattened the *Interpolated*

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

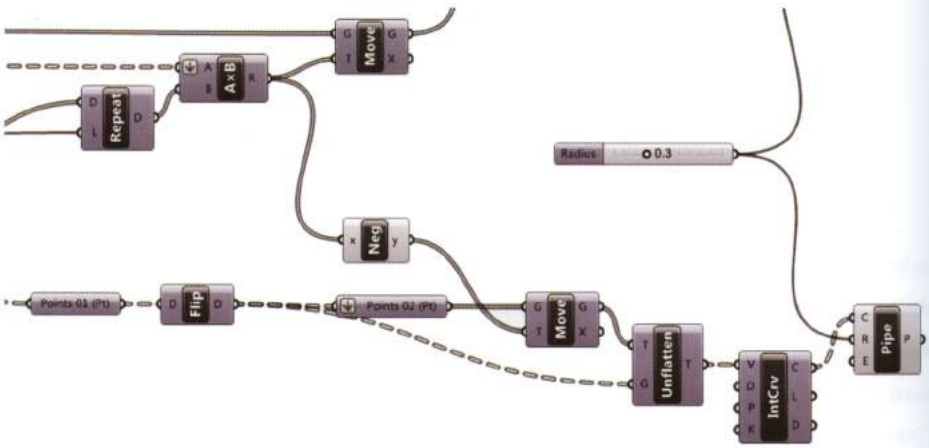


In order to interpolate a curve through each V direction column, the data must be unflattened using the *Unflatten Tree* component. The *Divide Surface* output *Points 01* is used as the guide Data Tree structure input (G-input of *Unflatten Tree*). The resulting data is structured according to the initial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



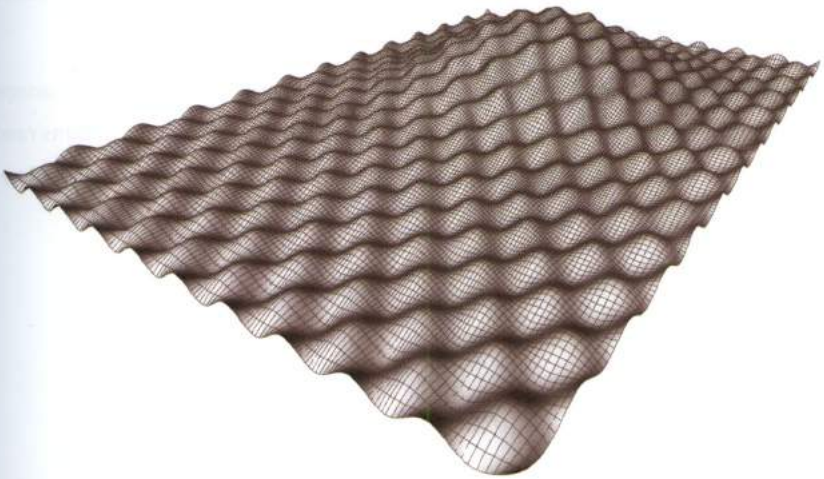
WEFT

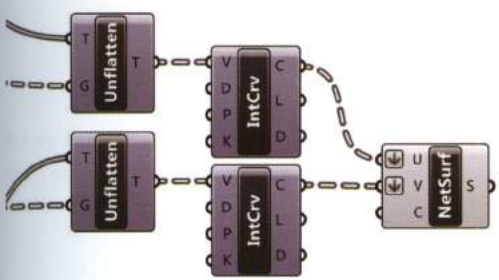
Using a similar logic the *weft* can be defined in the U direction. It is not necessary to repeat the entire algorithm to define the *weft*, instead flip the initial *Points 01* matrix and copy the end of the algorithm as shown below.



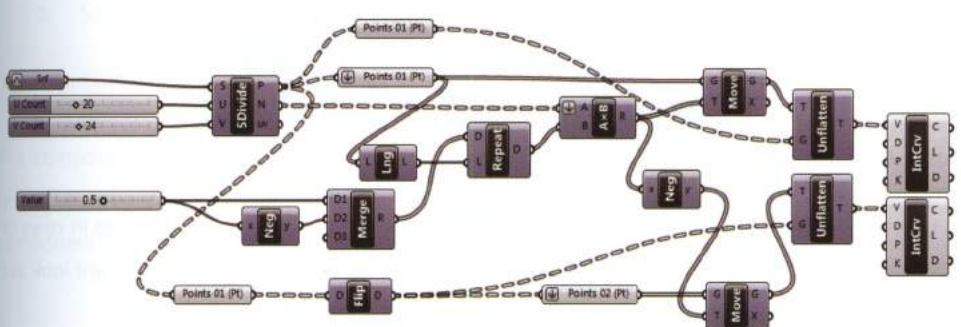
Since the *warp* and *weft* threads should never touch; the translation vectors of the *weft* must have the opposite sense as compared to the translation vectors of the *warp* (*Negative* component).

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





The complete algorithm is illustrated below.



5.3 Sorting strategies using Data Tree

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

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

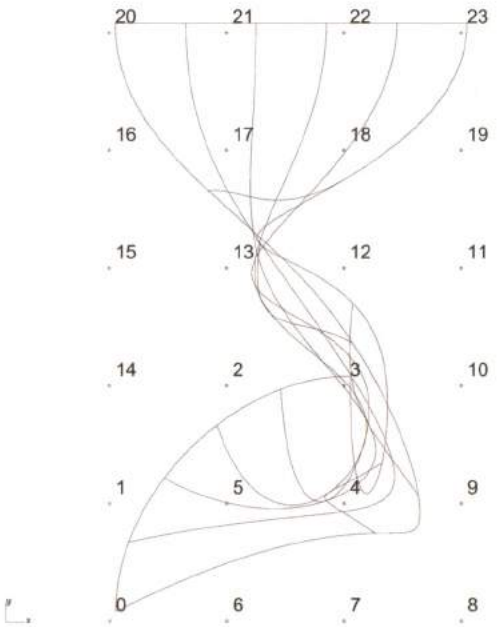


FIGURE 5.3
When points are not organized in rows or columns, *Surface From Points* component could generate incorrect results.

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

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

STEP 1

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

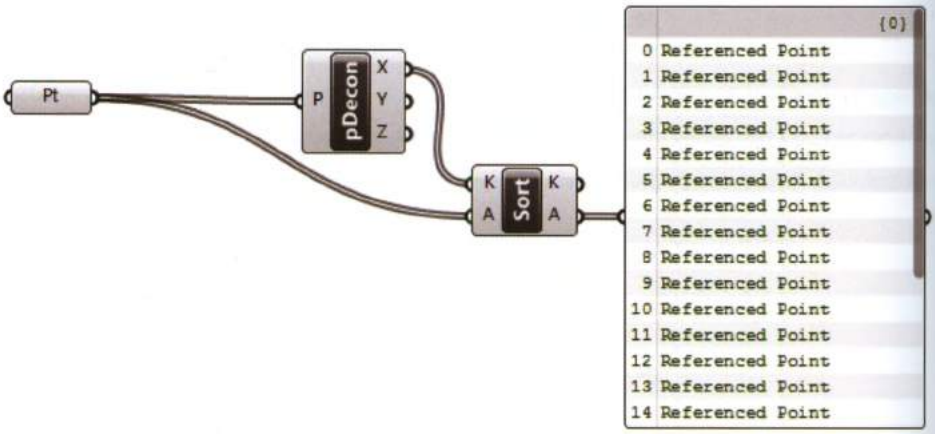


FIGURE 5.4
Sorting points according to the x-axis.

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

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

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

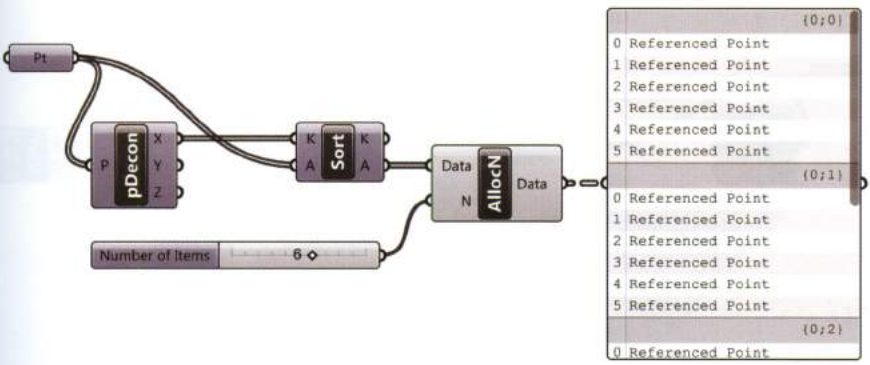
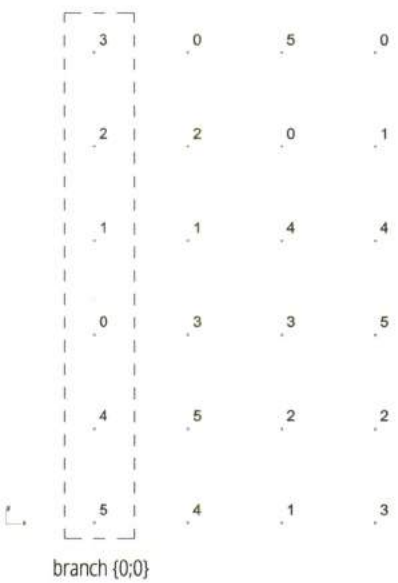


FIGURE 5.5
Splitting data using the *Allocate N* component. Each branch is related to a different column of the points grid.



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

STEP 2

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

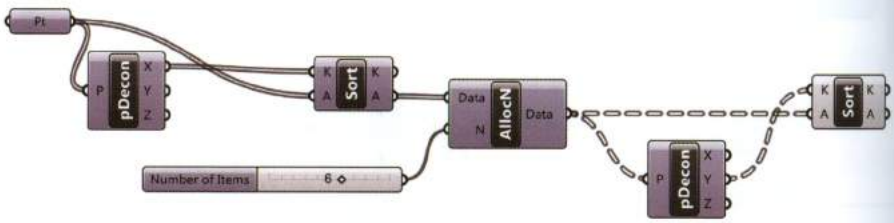
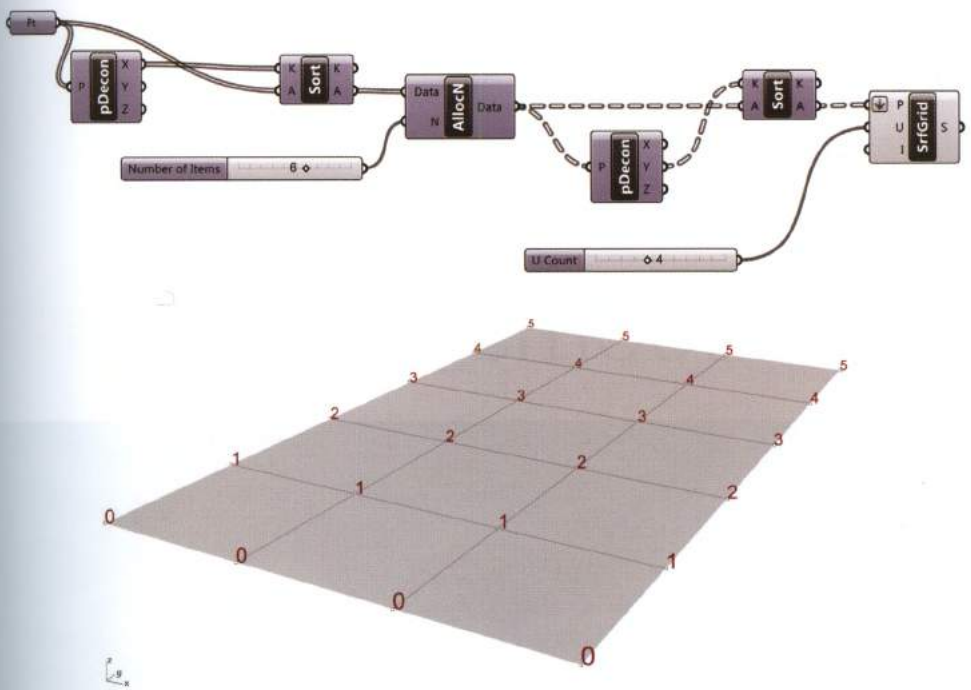
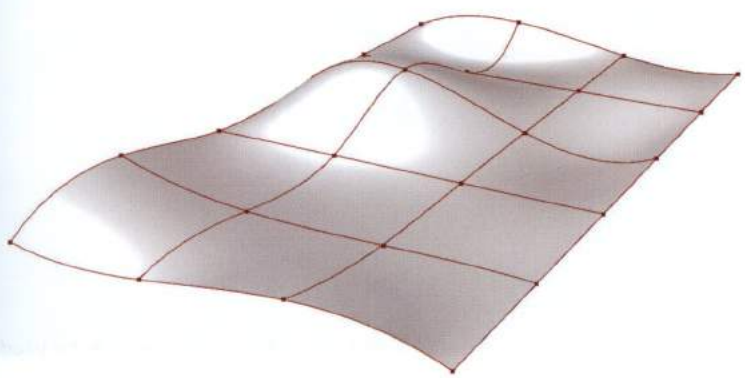


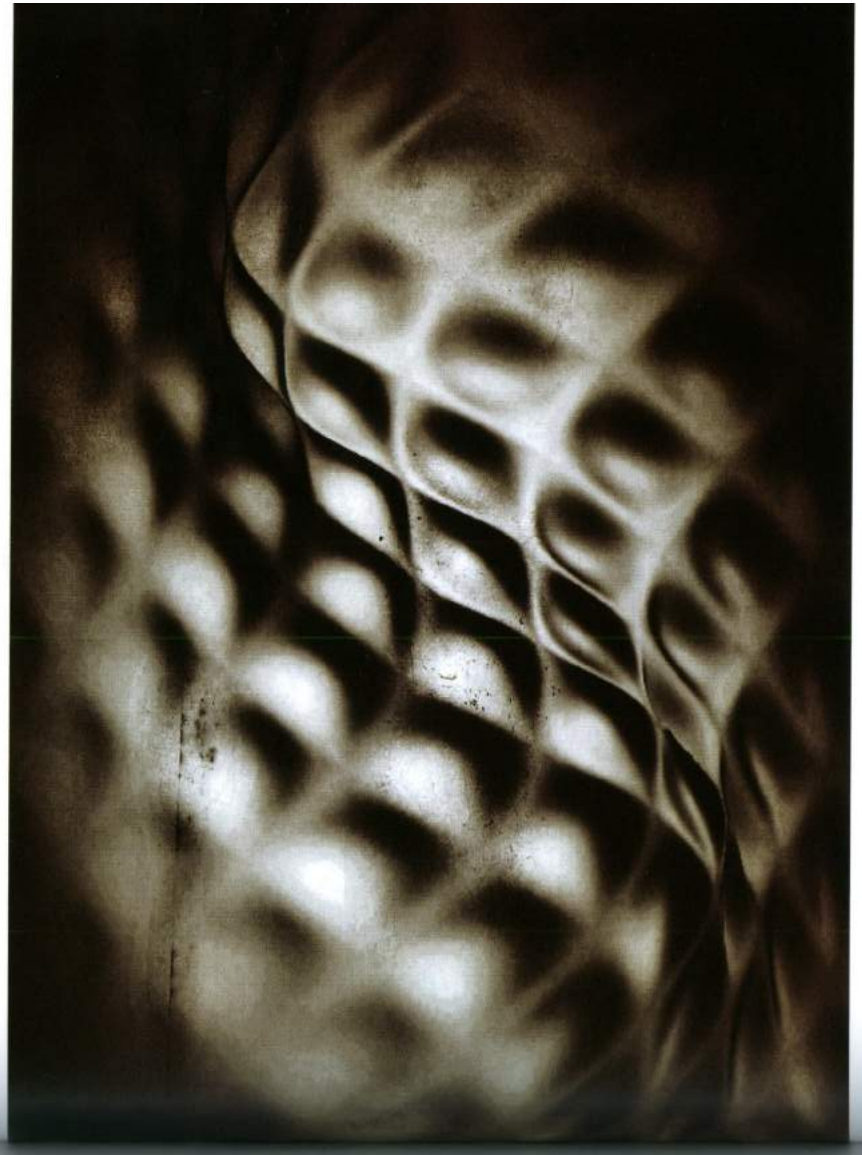
FIGURE 5.6
Points sorted according to the y-axis.

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



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





6_smoothness

"...there are enough indications right now that subdivision software packages are going to be relevant in the next ten years".[14]

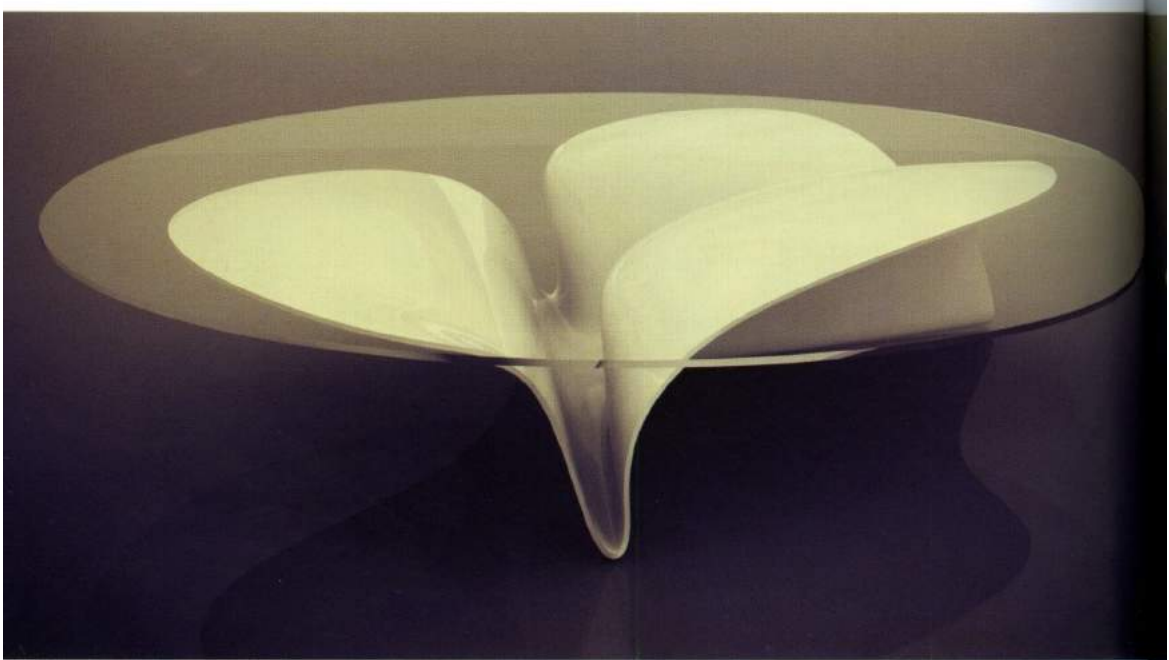
Greg Lynn

NURBS modeling is based on structuring mathematical curves or points to build surfaces. This method of surface creation is not always the best procedure, especially for freeform geometries. For example, the coffee table and the centerpiece shown on the following pages could be defined by NURBS surfaces; however, because of the surface complexity, another method relying on polygon meshes can be employed: the Subdivision Surface Modeling (SubD). This is a technique developed in the late 1970's and used extensively within the animation industry for character animation. The SubD method is based on the defining a schematic polygon mesh which is iteratively refined using specific algorithms. This technique allows users to model low-polygon meshes and generate smooth (high face count) meshes. The SubD method enables a level of smoothness which is quite difficult to obtain using NURBS techniques. Though NURBS are the dominant standard for CAD and engineering applications, SubD is becoming increasingly important in several fields such as industrial design.

NOTE 14

S. Converso, *Il progetto digitale per la costruzione. Cronache di un mutamento professionale* (Maggioli, 2010), 148–153.

FIGURE 6.1
Blossom Table, Arturo Tedeschi (2011).



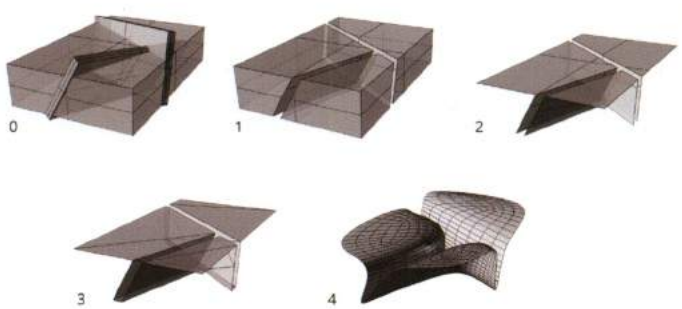
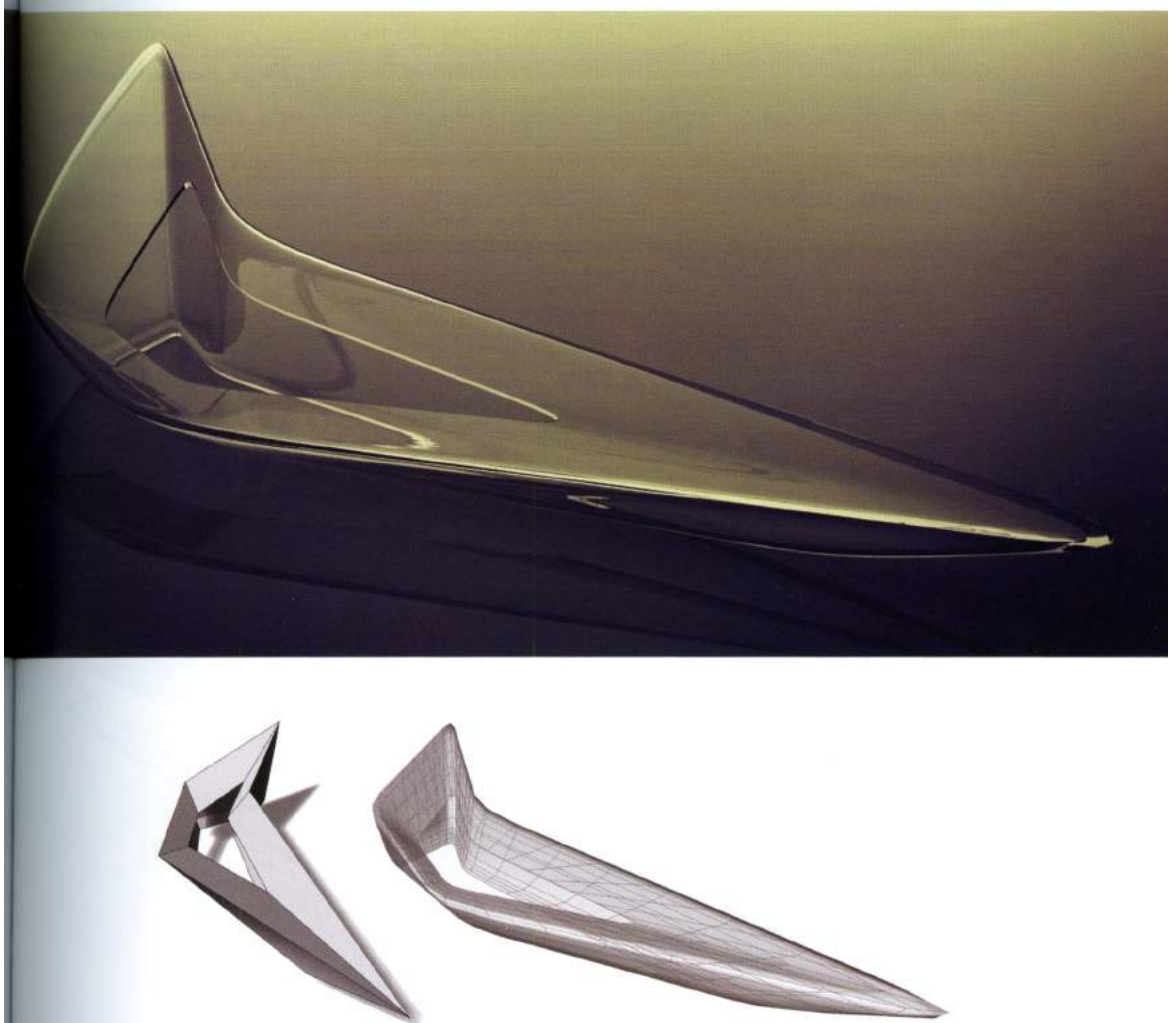


FIGURE 6.2
Centerpiece, Arturo Tedeschi (2011).



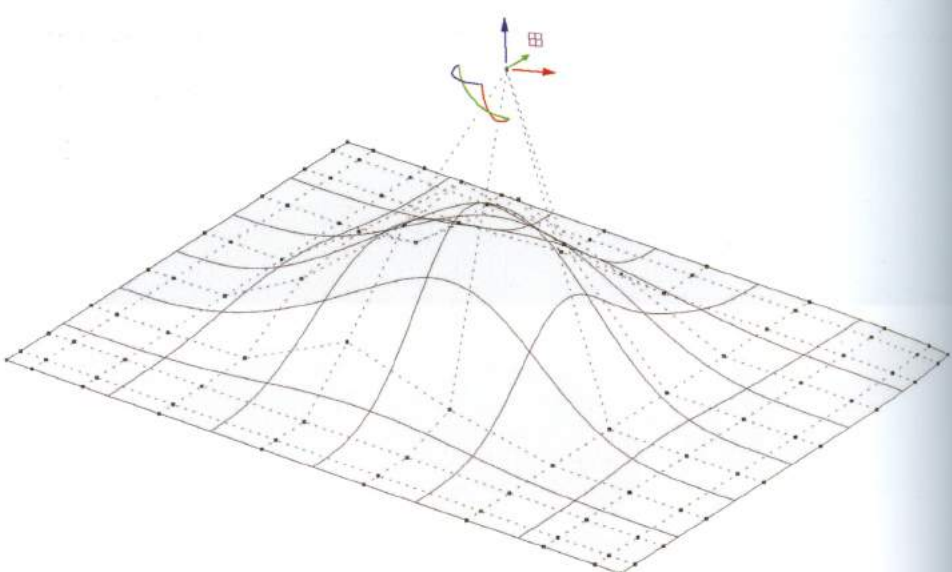
6.1 NURBS and Polygon Meshes

Polygon modeling is conceptualy different from NURBS modeling. Each method has strengths and weaknesses when defining tridimensional geometry.

Modeling software defines 3D objects by one of two methods:

- Representation by curves and NURBS surfaces;
- Approximation by polygon meshes.

Each curve, surface or freeform geometry discussed so far, whether created in Rhino or by Grasshopper components, are NURBS geometries (see chapter 3). NURBS are defined by a mathematical formulation which connects degree, control points, weights and knots in order to accurately represent any geometric object. Moreover, NURBS geometry is a **single geometric entity**. NURBS geometry can be modified by manipulating the object's control points. When control points are displaced or their relative weight is changed, a local deformation will result. The computation of coordinates is based on one or more parameters: *t* for curves and *u* and *v* for surfaces. NURBS surfaces, since they are based on mathematical formulations, define flexible and accurate freeform geometry; which are **inherently smooth**.



Alternatively, **polygon meshes** are not strictly defined by mathematical logic or curves. Polygon meshes are not single geometric entities, but they are made through a **set of adjacent polygons** which determines the global shape. A mesh is not actually smooth, the greater the number of polygon the smoother the mesh.

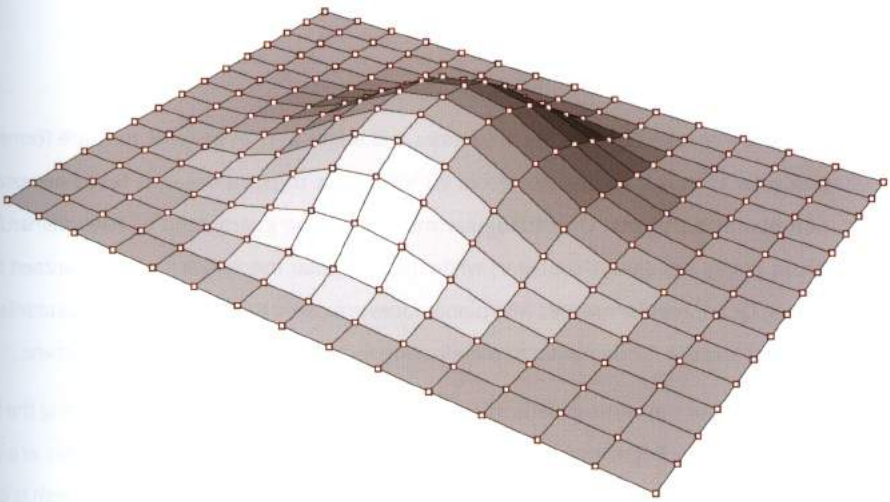


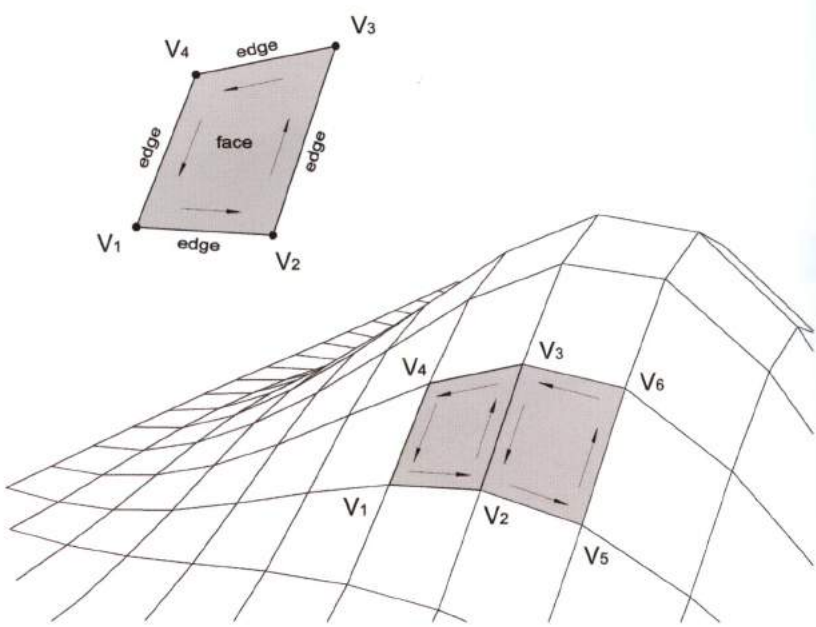
Figure 6.3
A polygon mesh is not actually a smooth geometry: the smaller are the polygons (below) the smoother is the mesh.

6.2 Polygon meshes

The elements of a polygon mesh are:

- **Vertices**: are points defined in the World Coordinate System by relative coordinates (x, y, z). The vertices provide positional information and affect the shape of the polygonal geometry;
- **Edges**: are segments that connect vertices;
- **Faces**: are constituted by a set of vertices and edges and, in general, they are formed by triangular or quadrangular polygons. It's important to point out that **faces are usually non-planar polygons**. Only triangular meshes can be guaranteed to have planar faces (three points are always coplanar) while quadrangular meshes are not guaranteed to be planar. Quadrangular meshes with planar faces are called **PQ Meshes** (Planar Quadrilateral Meshes) and are fundamental to panelization techniques of freeform architecture.

The connection-order of the vertices determines the orientation of the face, distinguishing the front face from the back face. Edges that connect vertices in the counterclockwise direction are front facing. Adjacent faces are defined as **compatible** if they have the same orientation. A mesh is called **orientable** if it is constituted by compatible faces.



For each face a **normal vector** can be defined. A front-facing mesh has an **outgoing** normal vector, while a back-facing mesh has an **ingoing** normal vector. As follows, an orientable mesh has normal vectors oriented in the same direction.

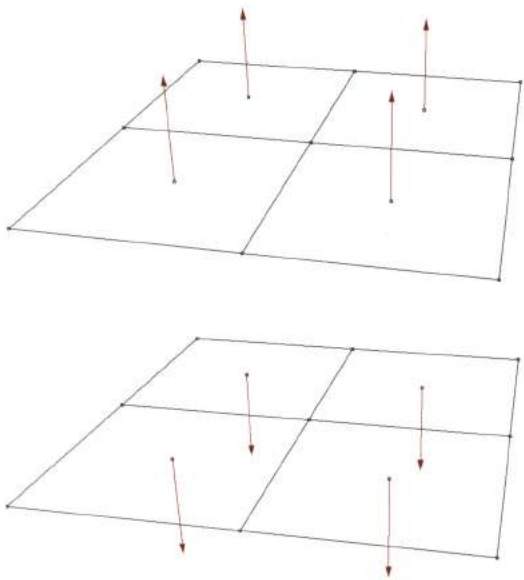


FIGURE 6.4
Two orientable meshes: the normals are oriented according to the same direction.

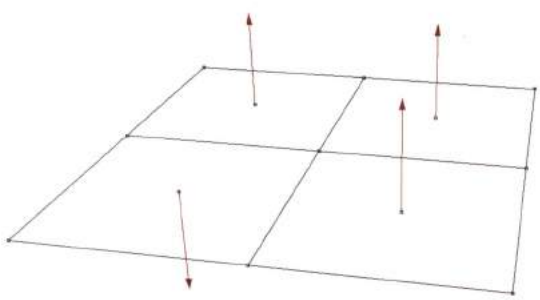


FIGURE 6.5
A non-orientable mesh.

6.2.1 Geometry and topology

A mesh is basically a geometric entity defined by a collection of points that determine its **geometry**, as well by the connection-logic that determine its **topology**. For instance, in figure 6.5, two mesh-boxes having the same vertices but a different connection-logic (topology). Conversely, image 6.6 illustrates two mesh-boxes with the same topology, but different geometry.

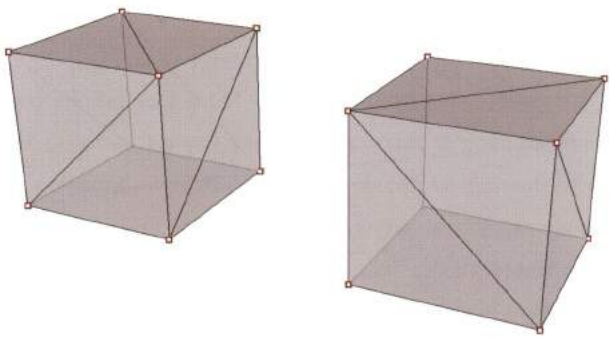


FIGURE 6.6

Two mesh-boxes with the same geometry but different topology.

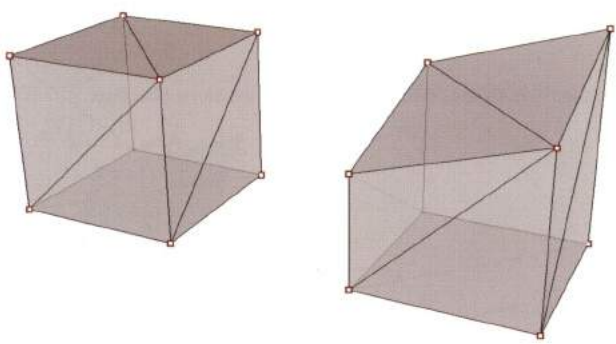


FIGURE 6.7

Two mesh-boxes with the same topology but different geometry.

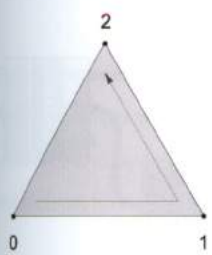
6.3 Creating meshes in Grasshopper

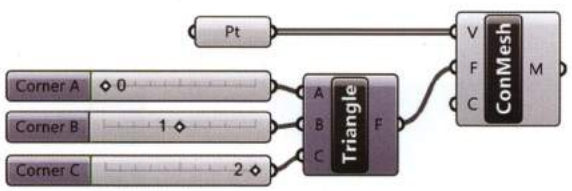
Several Grasshopper components are used to create meshes. Excluding mesh primitives (plane, box, sphere, ect.) which can be defined using a single component, there are three main strategies to define meshes in Grasshopper:

- Creating meshes by topology;
- Creating meshes by triangulation;
- Creating meshes by a NURBS to mesh conversion.

6.3.1 Creating meshes by topology

A mesh is defined by a collection of vertices connected sequentially, determining the meshes **topology**. The most basic mesh is a single triangular face with edges defined by three vertices: 0, 1, 2. The component *Construct Mesh* (Mesh > Primitive) is used to build a mesh. The component has two main inputs: (V) and (F). The V-input is defined by a set of three vertices, and F-input is defined by the component *Mesh Triangle* (Mesh > Primitive) which specifies the connection order i.e. the mesh topology. A is the first vertex specifying point index 0, B is the second vertex specifying point index 1 and C the third vertex specifying point index 2.



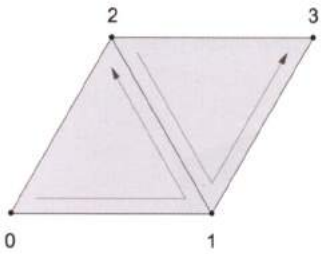


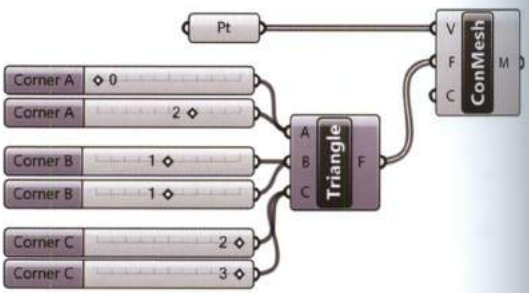
The above definition describes a three vertex triangular mesh with an anticlockwise direction. To visualize the mesh edges the *Preview Mesh Edges* option must be activated from the *Display* menu. If a fourth vertex is added to the list of set points, a new face will be generated by properly defining the mesh topology. In order to create two compatible faces the connection-order has to be set as follows:

Face 1: 0, 1, 2

Face 2: 2, 1, 3

In this case the A-input, B-input and C-input are all fed by two values as shown in the following algorithm:



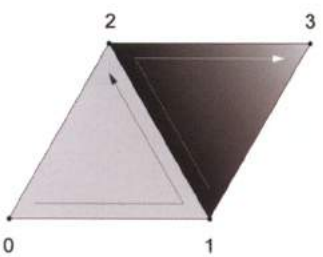


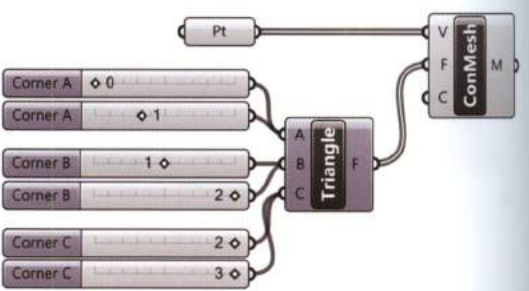
If the second face vertices are connected in the clockwise direction a non-orientable polygon mesh will result. Meaning, if the connection-order is set as follows, the face will be oriented in the opposite direction:

Face 1: 0, 1, 2

Face 2: 1, 2, 3

Grasshopper displays a non-compatibility warning by chromatically differentiating that two adjacent faces. The shading graphically represents the orientation: front (clear face) and back (dark face).





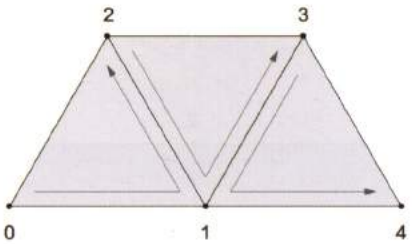
As follows, a triangular mesh with three faces can be created by setting five vertices topologically defined as follows.

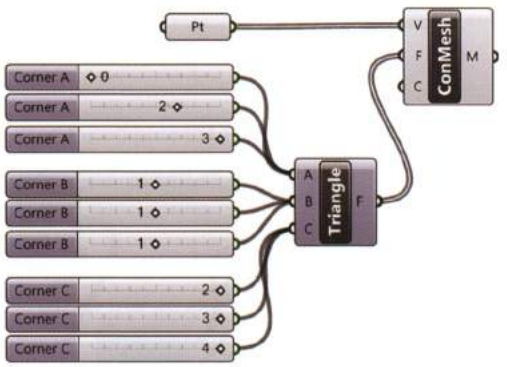
Face 1: 0, 1, 2

Face 2: 2, 1, 3

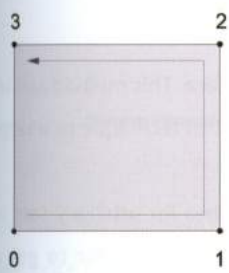
Face 2: 3, 1, 4

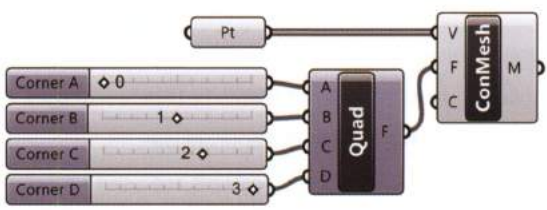
Since the three faces have anticlockwise-orientation an orientable mesh made up of three triangular faces results.





To create a quadrangular mesh face, the input (F) of the *Construct Mesh* component must be satisfied by the component *Mesh Quad* (Mesh > Primitive). The *Mesh Quad* component has four inputs to specify face topology and operates similar to the *Mesh Triangle* component. For example, to create a quadrangular mesh face, four vertices are set and connected according to the anticlockwise topology rule.

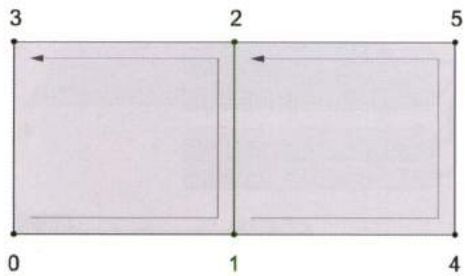


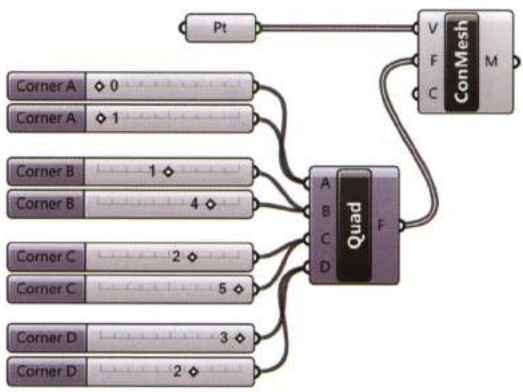


By adding two additional vertices to the set list of points, two mesh faces can define an orientable mesh by specifying the connection topology as follows:

Face 1: 0, 1, 2, 3

Face 2: 1, 4, 5, 2





6.3.2 Creating meshes by triangulation: Delaunay algorithm

Mesh topology specifies the connection order of vertices to define a mesh face. This method can be applied to small number of points or repeating logics. However, applying this technique to a large number of points is often not appropriate.

Alternatively, *triangulation algorithms* are used to define triangulated meshes for arbitrary sets of points. **Triangulation algorithms minimize differences between triangles in order to get a pseudo-regular mesh, avoiding skinny triangles.** In fact, skinny triangles can lead to inaccurate results if the mesh is used to calculate simulations or conduct analysis such as particle-spring

systems (see chapter 9) or finite element method (FEM) analysis.

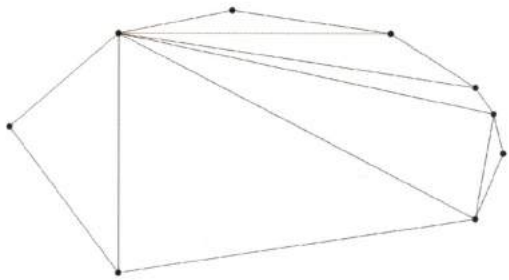
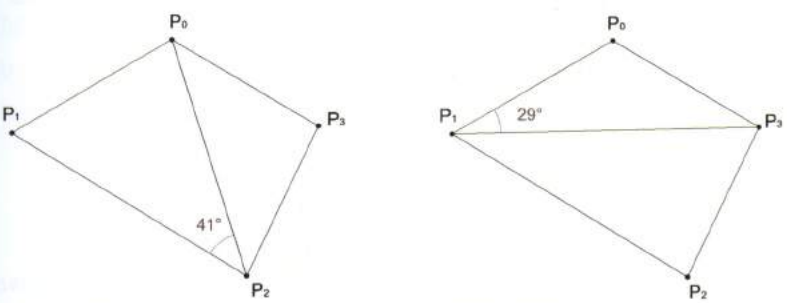


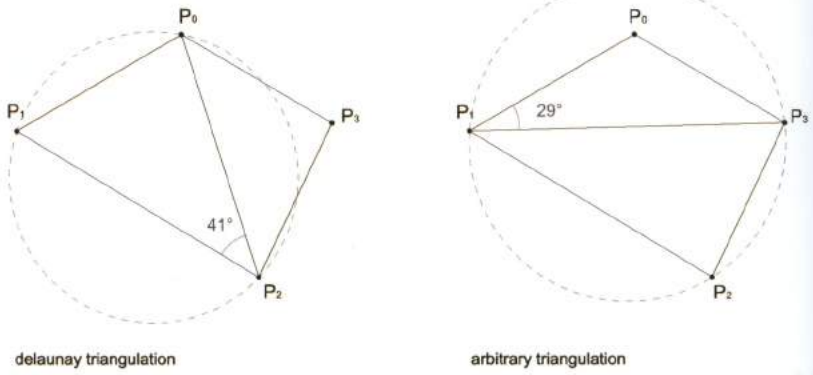
FIGURE 6.8
The image shows a mesh formed by skinny triangles. This kind of mesh is usually considered "bad" for simulations or analysis.

Several algorithms can be used to compute triangulations. The ***Delaunay algorithm***, named after the mathematician Boris Delaunay, also known as ***Delaunay triangulation*** is one of the most popular methods because of the outputs geometric properties.

The *Delaunay triangulation* connects an arbitrary set of points by a geometric procedure that maximizes the minimum angle of triangulation, so that the resulting mesh tends to avoid "thin" triangles. The following figure compares the *Delaunay triangulation* logic to an arbitrary triangulation, applied to the same set of points.



The *Delaunay algorithm* creates the triangulation "choosing" triplets of points that meet the following rule: once defined the circumscribed circle (*Delaunay* circle) through three vertices of the same face (P_0, P_1, P_2) the *Delaunay* circle through P_0, P_1, P_2 cannot contains other points inside.



The component *Delaunay Mesh* (Mesh > Triangulation) generates a triangular mesh according to the *Delaunay algorithm* for a series of defined points (P). The component's input (Pl) is the plane or planes where the algorithm operates.

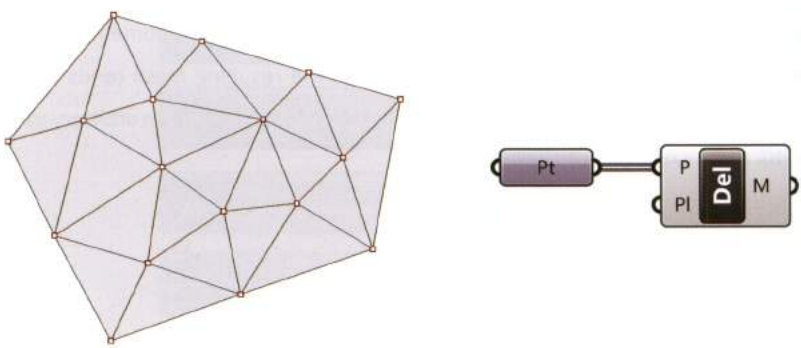
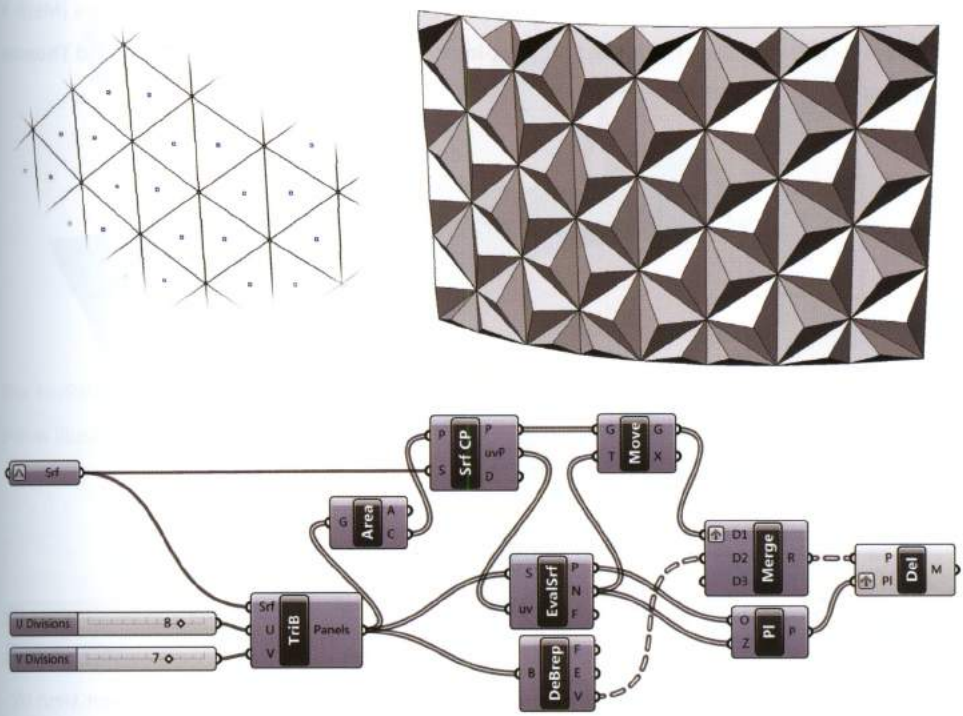


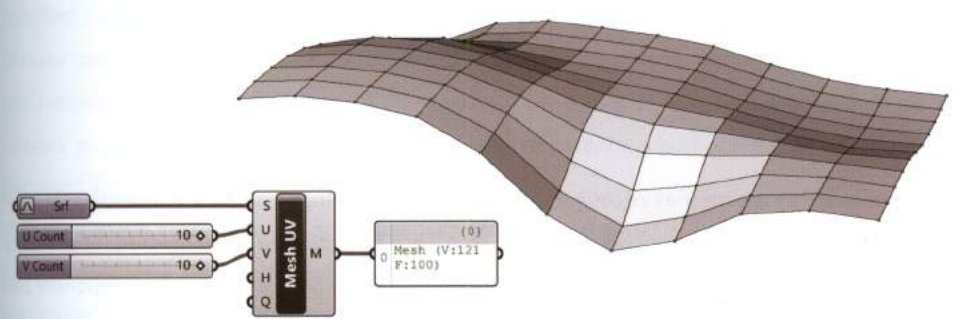
FIGURE 6.9
The Delaunay Mesh algorithm applied to a planar set of points.

In the following page you can see a tridimensional pattern get from a freeform surface using a *Delaunay algorithm*. As first step a triangular grid is created using the *Triangle panels B / TriB* component provided by the **Lunch Box** plug-in. For each triangle the centroid is calculated and subsequently moved according to the surface's normal vectors. The *Delaunay algorithm* is performed on the three vertices of each triangle merged with the translated centroids. The *Delaunay Mesh* component composes the vertices using the defined planes into a collection of meshes with 3 faces.

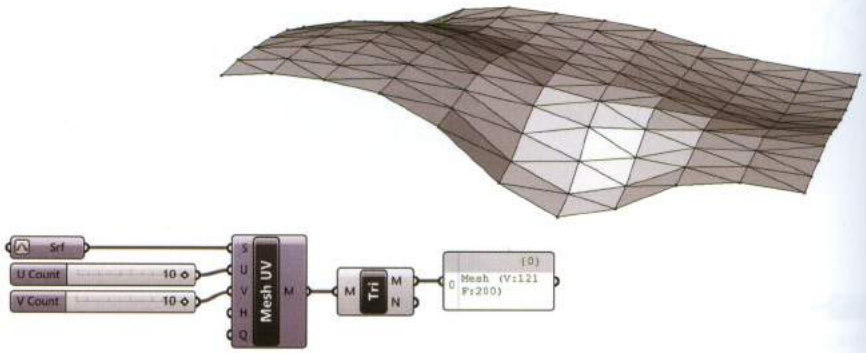


6.3.3 Creating meshes by a NURBS to mesh conversion

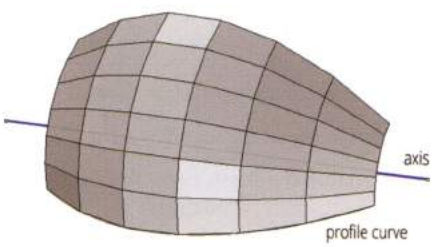
Meshes can also be defined from a base NURBS surfaces. The component *Mesh Surface /Mesh UV* (Mesh > Util) converts a NURBS surface into a **Quadrangular Mesh** by specifying the number of quads in U and V directions. To display the mesh edges enable the *Mesh Edges* option (Display > Preview Mesh Edges).



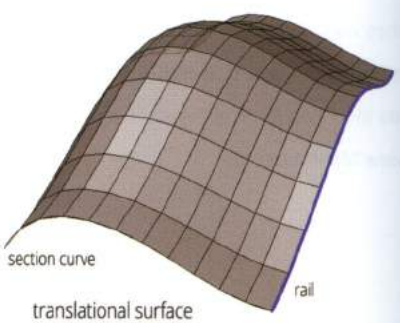
A **triangular mesh** can be obtained from a quad mesh using the *Triangulate* component (Mesh > Util) available after installing the **Mesh Edit plug-in, developed by [uto] (Ursula Frick and Thomas Grabner)**.



As discussed in section 6.2, triangular meshes are always planar, while Quadrangular meshes are planar only under certain circumstances. For instance, rotational and translational NURBS surfaces return **planar quadrilateral** meshes (or **PQ Meshes**) when converted using the component *Mesh UV*.

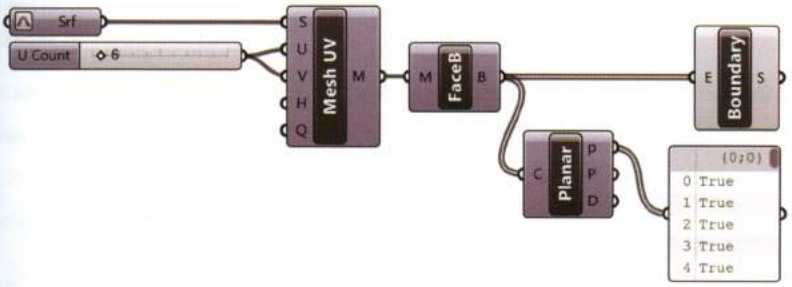


rotational surface

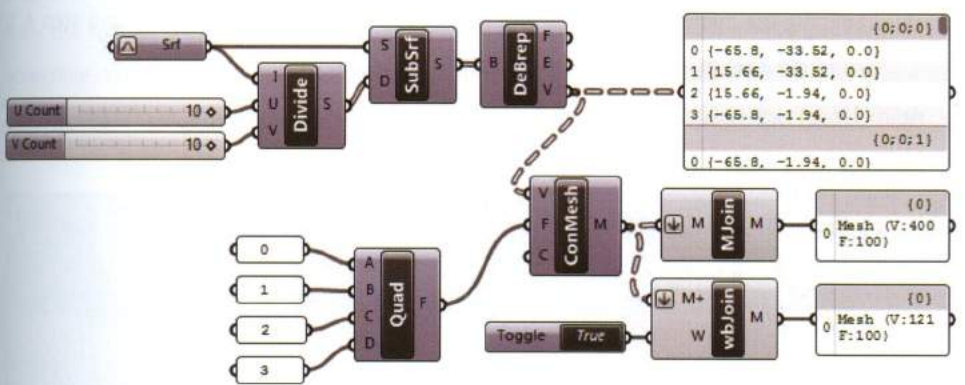


translational surface

To test for planarity the meshes edges are extracted using the component ***Face Boundaries*** (Mesh > Analysis) returning a boundary polyline for each face. Each boundary polyline is tested for planarity using the component ***Planar*** (Analysis > Curve). Alternatively, the *Boundary Surfaces* component can be used to test for planarity, since by definition the component will draw a surface only if the input curves form a planar boundary.



The NURBS to Mesh conversion can also be performed by specifying topology, as the definition below illustrates.

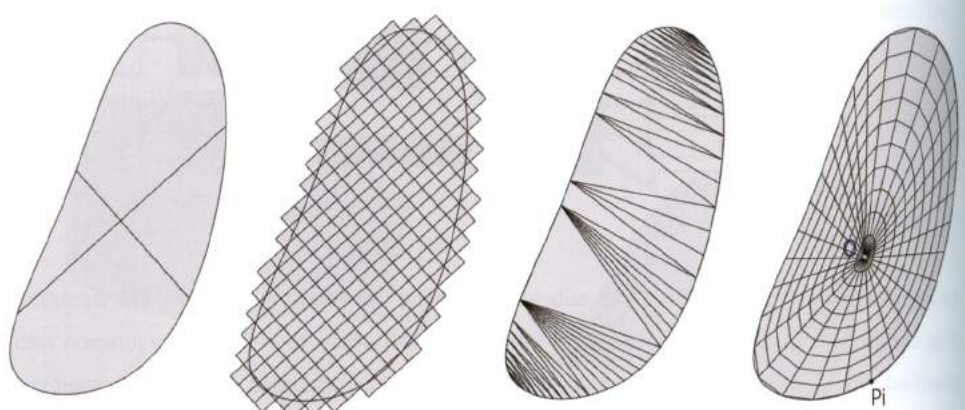


An initial A set surface is divided into sub-surfaces using the *Isotrim* component. The *Deconstruct Brep* component extracts the four vertices from each sub-surface, to define (in conjunction with a specified topological order) the *Construct Mesh* component, as discussed in 6.3.1. The result is set of quad meshes which can be joined into a single mesh by the component *Mesh Join* (Mesh > Util). Alternatively, *Join Meshes and Weld* (Wb > Extract) can be used to join meshes and weld coincident vertices, as specified by a Boolean toggle. *Join Meshes and Weld* is a part of the Weaverbird plug-in, discussed in next section.

NURBS to mesh conversion is compatible with **untrimmed** surfaces (see 3.5), since untrimmed surfaces contain a rectangular isogrid that can be converted into a set of quad meshes. Unlike untrimmed surfaces, **trimmed** *surfaces* are not compatible with the NURBS to mesh conversion method. Trimmed surfaces converted using the *Mesh Surface* component will return a polygon mesh

that approximates the original boundary geometry. Trimmed surfaces converted using the *Mesh Brep* (Mesh > Util) component will return a polygon mesh that preserves the original boundary geometry but allows for thin triangles. The NURBS to mesh conversion method for a trimmed surface must be considered on case by case basis; a possible strategy to convert a trimmed surface into a mesh is summarized as follows:

1. Adding an arbitrary point O;
2. Dividing the surface's boundary into N parts getting N points P_i;
3. Creating a set of lines from O to the subdivision points P_i;
4. Dividing the lines into equal parts and connecting the resulting points with a set of polylines;
5. Splitting the initial surface using the network of lines obtained in step 3 and step 4, to generate a set of trimmed sub-surfaces;
6. Extract the sub-surfaces' vertices and create the mesh relying on topology method.



Trimmed Surface Mesh Surface Mesh Brep

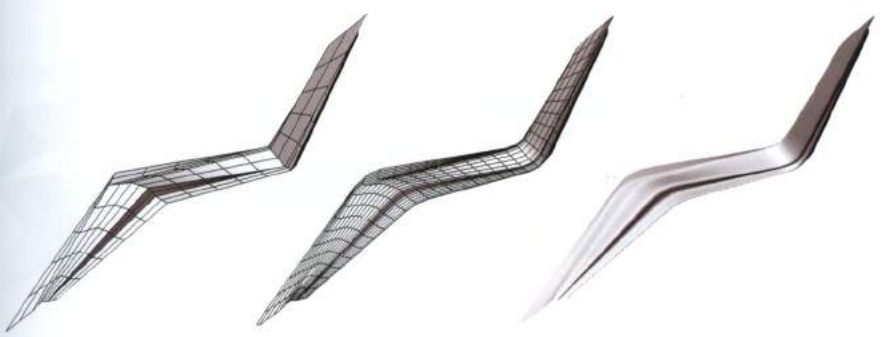
6.4 SubD in Grasshopper: Weaverbird plug-in

The basic idea behind Subdivision Surfaces is that we can get a smooth polygonal surface (limit surface) associated with any given input mesh. SubD algorithms process an input mesh and generate an output mesh that is closer to the limit surface. Depending on the type of the input mesh (triangular, quadrilateral etc.) we must use a different SubD algorithm. We will particularly focus on two method:

- Loop subdivision algorithm;
- Catmull-Clark subdivision algorithm.

Triangular meshes are commonly subdivided using the *Loop Subdivision* algorithm, while quadrilateral meshes are generally subdivided using the *Catmull-Clark* algorithm. Both algorithms are part of the plug-in **Weaverbird** developed by Giulio Piacentino[15]. The plug-in is available for free download from the following website: www.giuliopiacentino.com. After installation a new tab (Wb) will be accessible. Weaverbird's toolbar allows users to access components to perform subdivisions and transformations as well as define polygonal primitives and extract main elements from a mesh.

FIGURE 6.10
Weaverbird components toolbar.



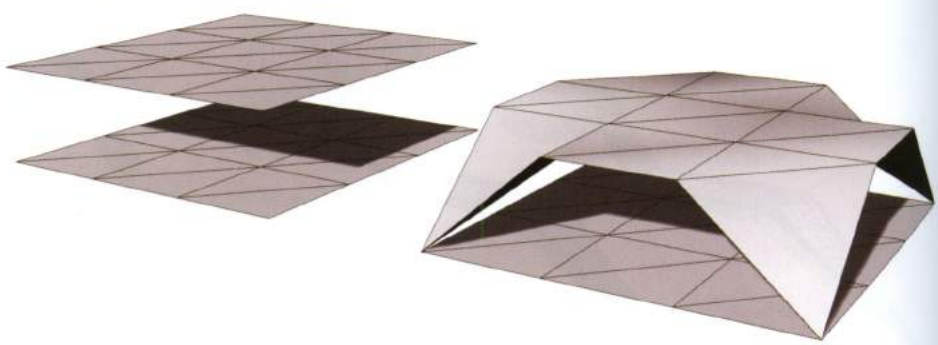
NOTE 15
Giulio Piacentino, architect, graduated from Politecnico di Torino, continued his education at the University of Technology of Delft, Holland, then began a co-operation with NIO Architecten. He currently develops for McNeel and teaches at geometrydepth.com. He has trained students in geometry all around Europe.

6.5 Subdivision of triangular meshes: Loop algorithm

The Loop algorithm is an iterative subdivision method for **triangular** meshes, developed by Charles Loop[16] in 1987. For every edge in the source mesh, the algorithm adds a new vertex at the mid point. The mid points are connected and every triangle is replaced by 4 sub-triangles. Each iteration of the Loop algorithm approaches the limit surface, i.e. the surface defined by infinite iterations. The number of iterations is counter proportional to the change in geometry. As a result, only a few iterations are usually required to approximate a limit surface.

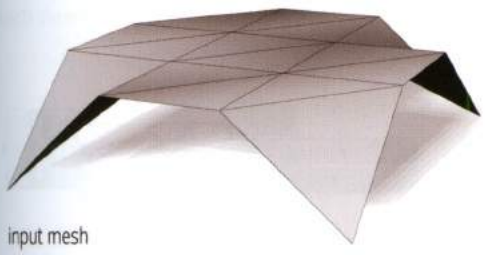
FIGURE 6.11
Loop subdivision scheme. Each triangle is divided into 4 sub-triangles, adding new vertices in the middle of each edge.

The Loop algorithm is performed by the component *wbLoop* (Wb > SubD). The M-input is the mesh to subdivide, the L-input the number of subdividing iterations and the S-input specifies how to treat the naked edges of the input mesh. The number of iterations can vary from 1 to 3. Generally, a simple and schematic input mesh will yield a more refined and polished output mesh.

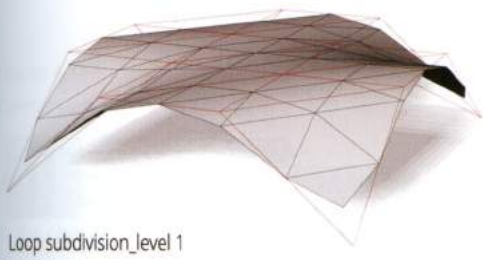


NOTE 16
C. Loop, "Smooth Subdivision Surfaces Based on Triangle", M.S. Mathematics thesis, University of Utah, 1987.

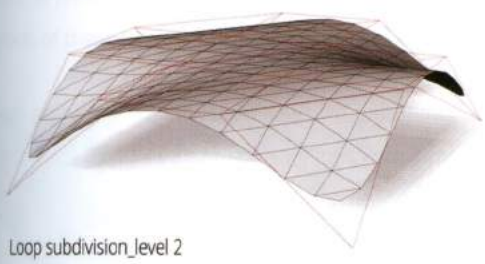
For example, a triangular mesh set from Rhino (previous image) can be subdivided using the following definition. The L-input varies from 1 to 3 illustrating the iterative refinement of the mesh, S-input is set to 1 (*Smooth*) and as a result the naked edges tend towards a spline.

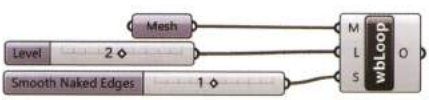


input mesh

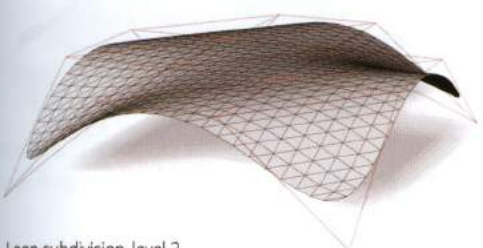


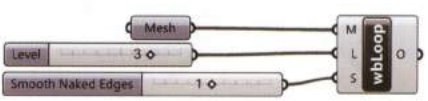
Loop subdivision_level 1





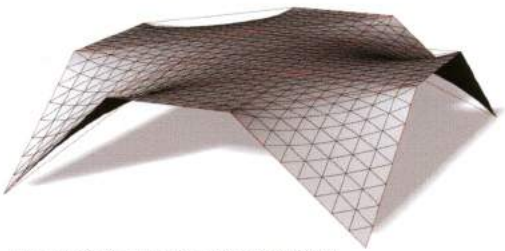
Loop subdivision_level 2

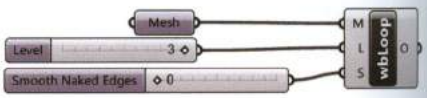




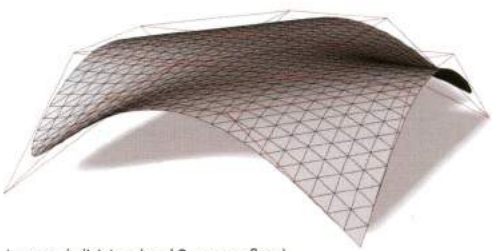
Loop subdivision_level 3

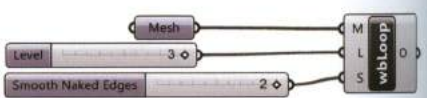
If the S-input of *Loop* component is set to 0 (*Fixed*) the naked edges will be fixed and do not move from the original position, 1 (*Smooth*) the naked edges tend towards a spline, or 2 (*Corner Fixed*) the naked edges tend towards a spline and two vertices are fixed. The three types of naked edge options are illustrated in the previous example or below.





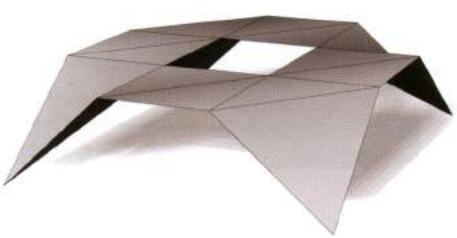
Loop subdivision_level 3_naked edges fixed





Loop subdivision_level 3_corner fixed

As we previously discussed a refined mesh can be generated from a simple input mesh using the Loop subdivision algorithm. The Loop SubD algorithm also works for meshes with holes. Once the mesh is subdivided the hole, as demonstrate below, will become pseudo rounded.



6.6 Subdivision of quadrangular meshes: Catmull-Clark algorithm

The Catmull-Clark is an iterative subdivision algorithm for **quadrangular** and **triangular** meshes, developed by Edwin Catmull and Jim Clark[17] in 1978. The algorithm iteratively adds new vertices for each face to approximate a smooth surface.

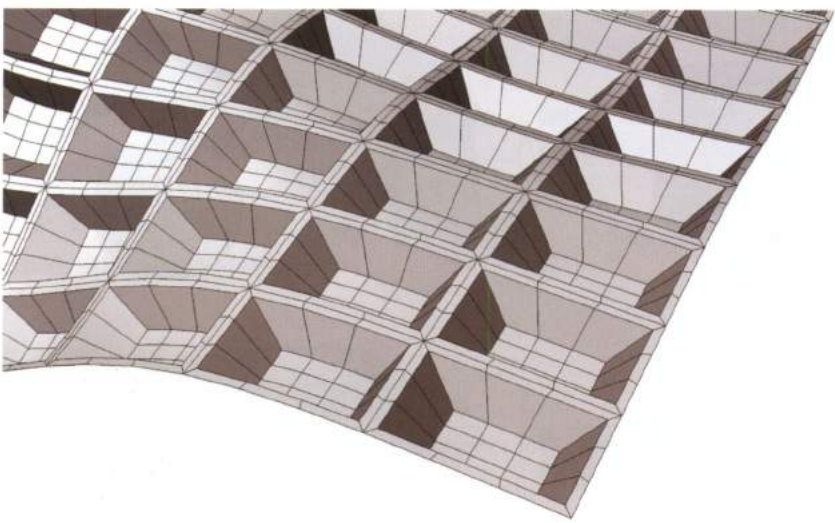
FIGURE 6.12
Catmull-Clark subdivision scheme.

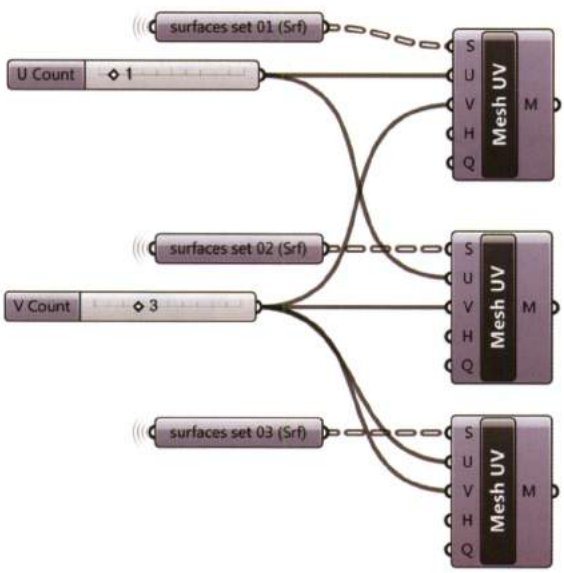
The following image, displays the final result of converting the tridimensional NURBS skin – created in the chapter 5 – into a smooth and continuous polygon mesh using the Catmull-Clark algorithm. As explained in 5.2.1 the final skin is composed of three sets of surfaces: the frame (*surfaces set 01*), the lateral faces (*surfaces set 02*) and the lower faces (*surfaces set 03*). Before subdividing the surfaces, each of these surfaces must be converted into a mesh using the component *Mesh Surface*.

FIGURE 6.13
Smoothing a tridimensional skin by a subdivision strategy based on the Catmull-Clark algorithm.

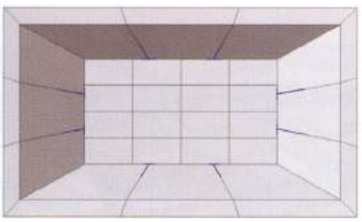
NOTE 17
E. Catmull and J. Clark, *Recursively generated B-Spline surfaces on arbitrary topological meshes*, Computer Aided Design, 1978.

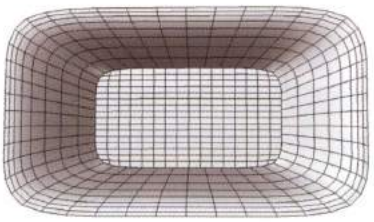
The component *Mesh Surface* requires a specified number of quads in U and V direction. When converting a single surface the number of U and V subdivisions is arbitrary. Conversely, for multiple merged surfaces, **the number of quads in U and V direction should be set so that edges always meet in the vertices, avoiding T-nodes**.



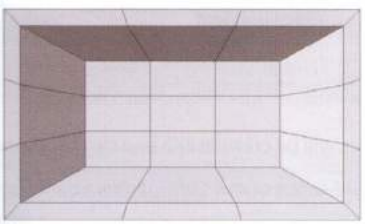


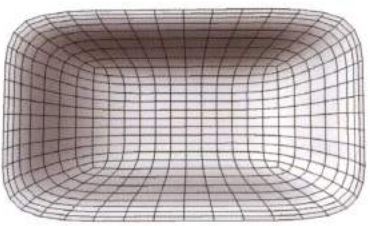
When a set of contiguous meshes have T-nodes (image below), they cannot be **welded** correctly. Under this condition, the subdivision algorithm operates on each individual disconnected mesh.



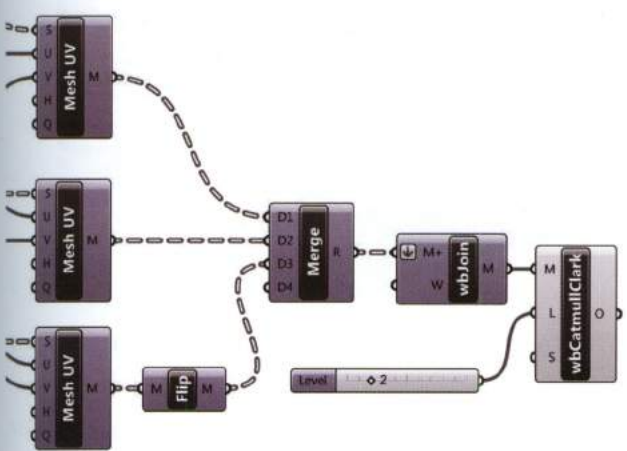


If the U and V count is consistent such that the nodes can be welded, the subdivision algorithm will operate as expected.





If the three mesh nodes align and can be welded, the component ***Join Meshes and Weld*** (Weaverbird > Extract) can be used to create a single mesh that can be smoothed by the component *Weaverbird Catmull-Clark Subdivision* (Weaverbird > SubD).

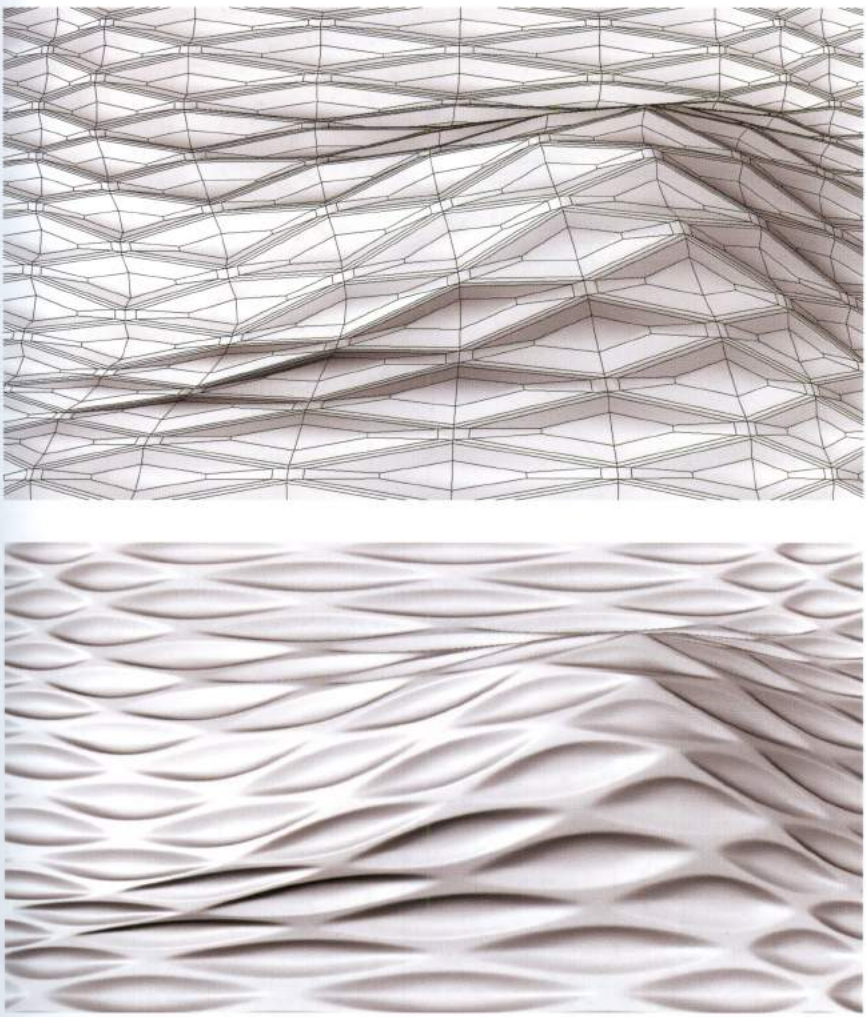


The component *Mesh Flip* (Mesh > Util) reverses the direction of the faces created from the *surfaces set 03*. Mesh Surfaces must be compatible in order to be joined and welded, as discussed in section (6.2). If the surfaces are not compatible a shadow will appear along the mutual edge, graphically displaying an error.

By flipping the faces created from *surfaces set 03*, the faces will be compatible and can be joined, welded and then smoothed. Similar to the *Loop* component the M-input of the *Catmull-Clark* component is the mesh to subdivide, the L-input the number of subdividing iterations and the S-input specifies how to treat the naked edges of the input mesh.

FIGURE 6.14
The final mesh resulting from the subdivision operated by the Catmull-Clark component.

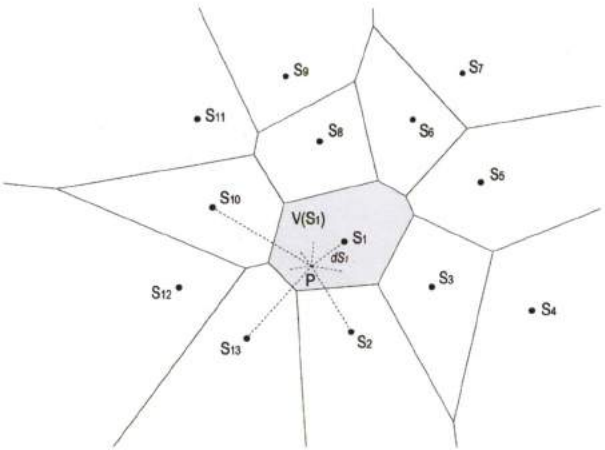
The subdivision strategy developed in the previous example can be applied to the hexagonal skin discussed in 5.2.2. The following image displays the hexagonal skin before and after subdividing using the Catmull-Clark algorithm.



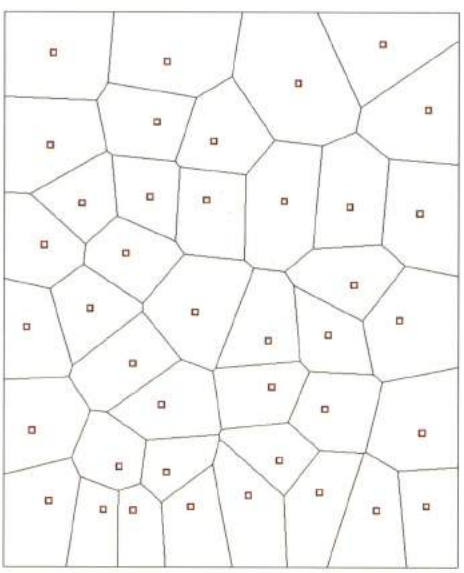
6.6.1 Voronoi skin

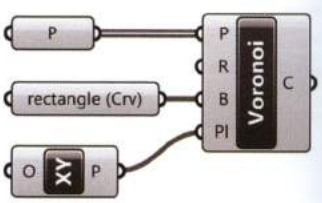
The ***Voronoi diagram***, named after the mathematician Georgii Voronoi, is a decomposition of a metric space according to proximity criteria. Given a specified set of *n* points S (S_1, S_2, ..., S_n), the Voronoi diagram for S is the decomposition of the bidimensional space which associates a region

$V(S_i)$ called Voronoi cell, to each point of S, so that all the points of $V(S_i)$ are closer to S_i than any other point of S. The Voronoi diagram has practical applications in different fields, from physics to city planning (territorial division based on distances from a specific center). Such a diagram has also fascinated designers for its intrinsic beauty.

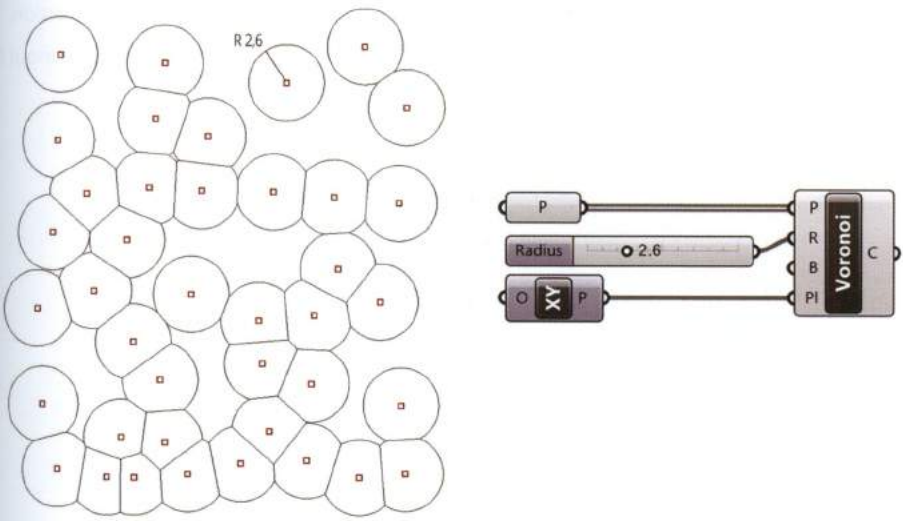


The component *Voronoi* (Mesh > Triangulation) creates a planar Voronoi diagram given a set of points (P), on a plane (Pl) within a containment boundary (B).

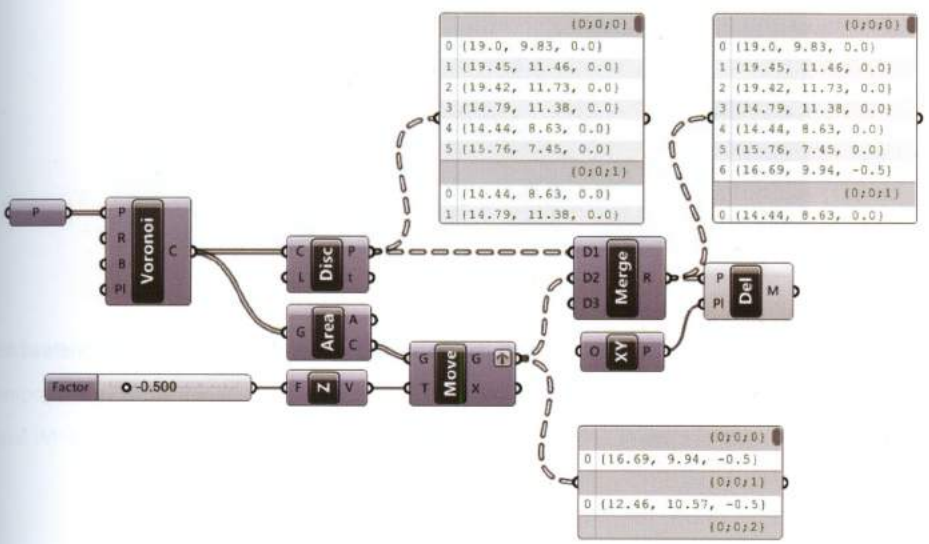




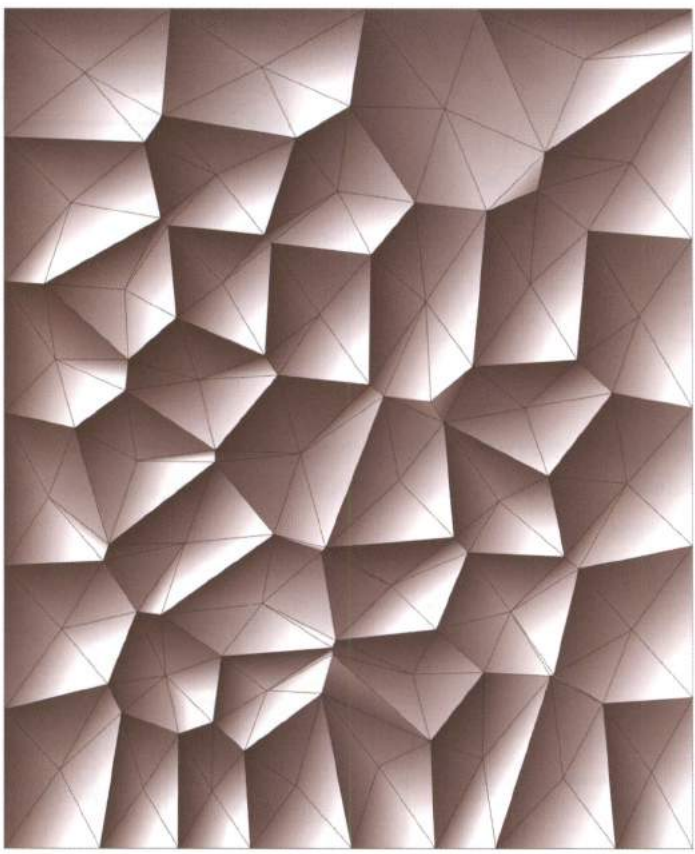
An optional radius R can be set for the diagram, as shown below. If no radius is specified the Voronoi component defaults to an infinite radius input (R) resulting in merged cells.



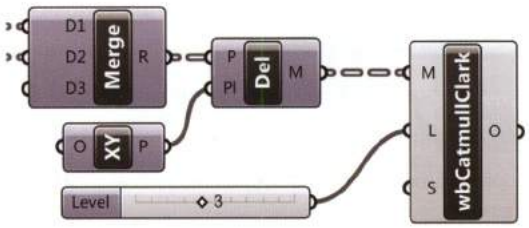
Similar to previous examples a Voronoi skin can be created from a planar Voronoi diagram. The procedure is to define a set of pyramids to smooth by a subdivision algorithm. Each pyramid has a Voronoi cells as a basis.

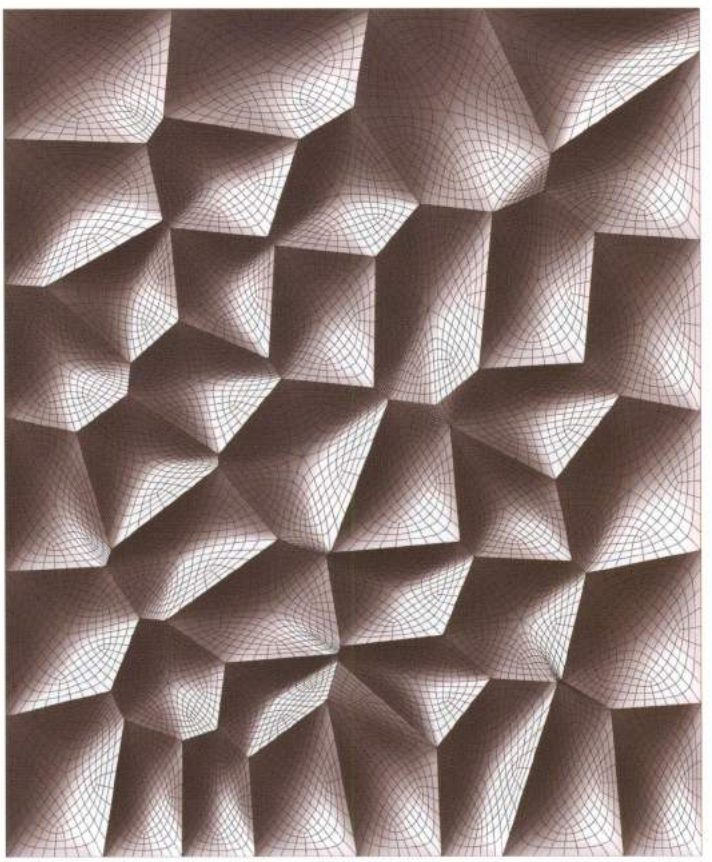


The first step in the Voronoi skin procedure is to use the component *Discontinuity* (Curve > Analysis) to return the discontinuity points or the vertices that define each Voronoi cell. The discontinuity component creates a *branch* for each cell with n items depending on the number of discontinuity points for each cell. Next, the centroid of every cell is translated in the negative Z direction. The discontinuity points and the translated centroid points are merged into a single list, with input (D1) and (D2) set to *graft*.

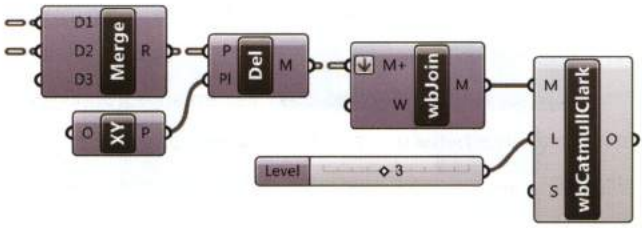


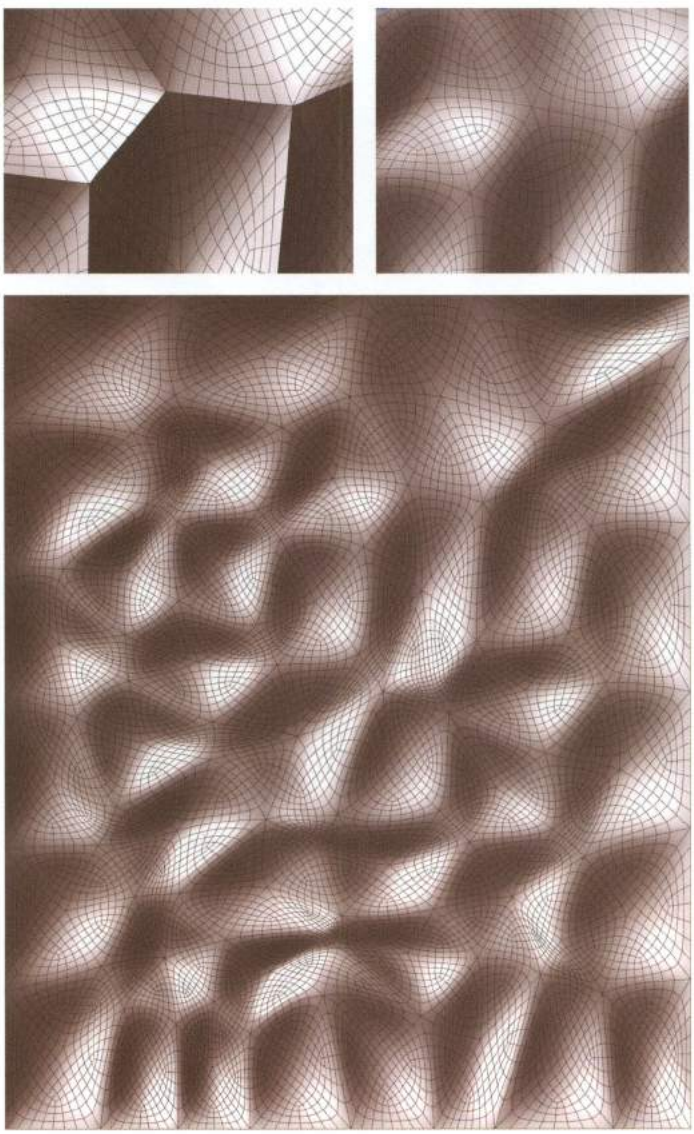
The polygon mesh defined after triangulation is not a single joined and welded mesh, instead it is formed by 40 unique discontinuous meshes.



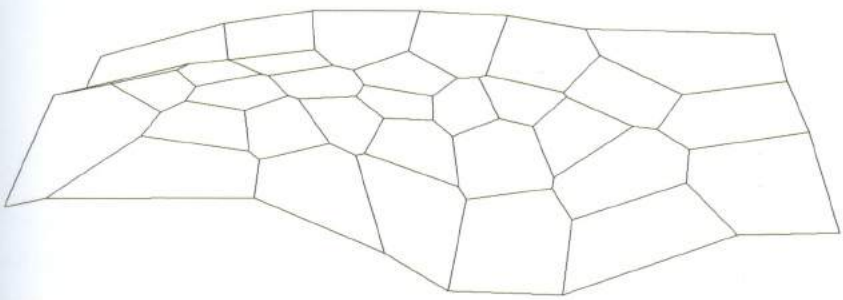


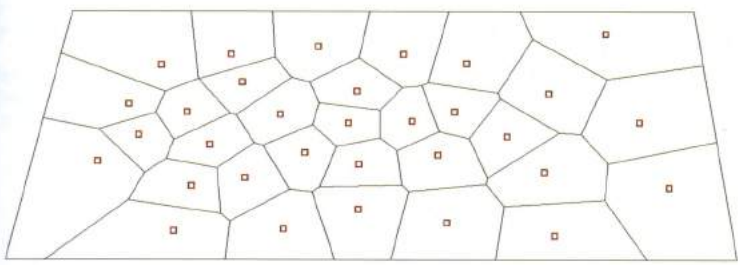
To create continuity between the unique meshes the M-output of the *Delaunay* triangulation component must be joined and welded using the component *Weaverbird Join Meshes and Weld*, with input (M+) set to flatten. The final result is shown in the following images.

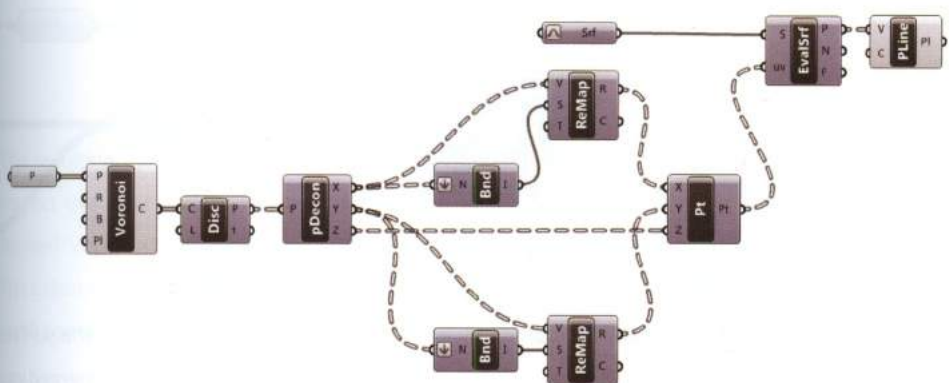

D1
D2
D3
Merge
R
P
Pl
Del
M
M+
W
wbJoin
M
M
L
S
wbCatmullClark
O
O
XY
P
Level
3



Such a strategy can be also applied to non-planar Voronoi diagrams which can be achieved using plug-in software or relying on several strategies. One of them is based on *remapping* (see 2.5). As first step we create the Voronoi on the plan and then we remap Voronoi's discontinuity points (*Discontinuity* component) in a range between 0 and 1 (*Remap Numbers* component). Since the points of a reparameterized target surface range between the same domain, we can use *Evaluate Surface* and *Polyline* to reproduce the Voronoi diagram onto a freeform surface. The complete algorithm is shown below.

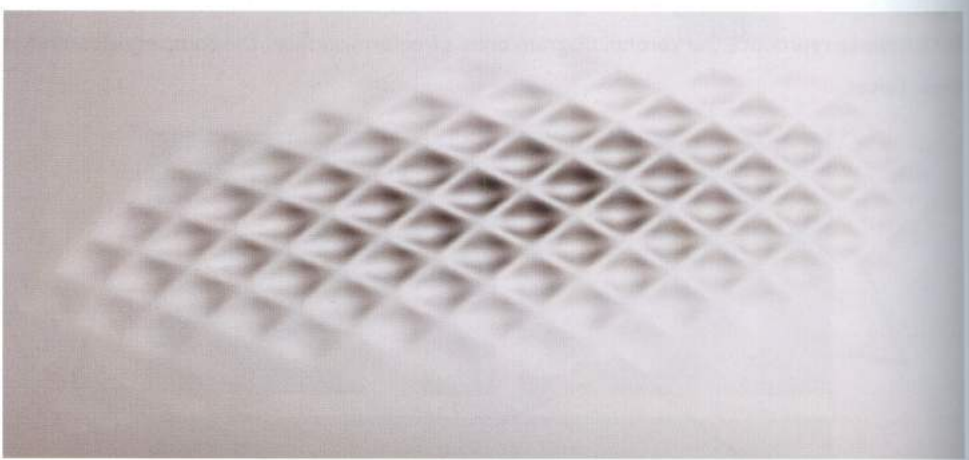




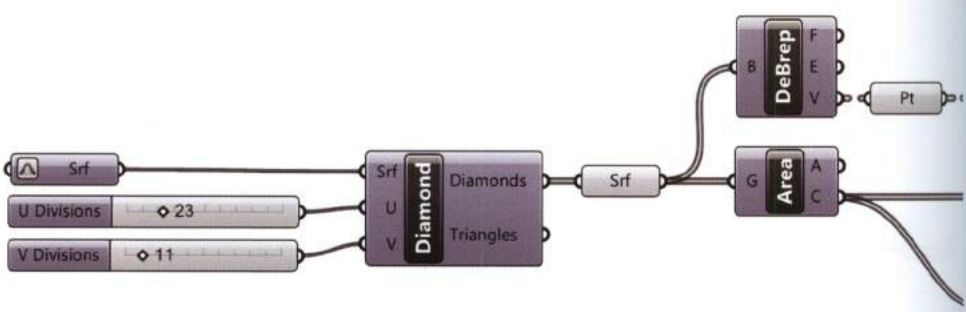


6.6.2 Fading pattern

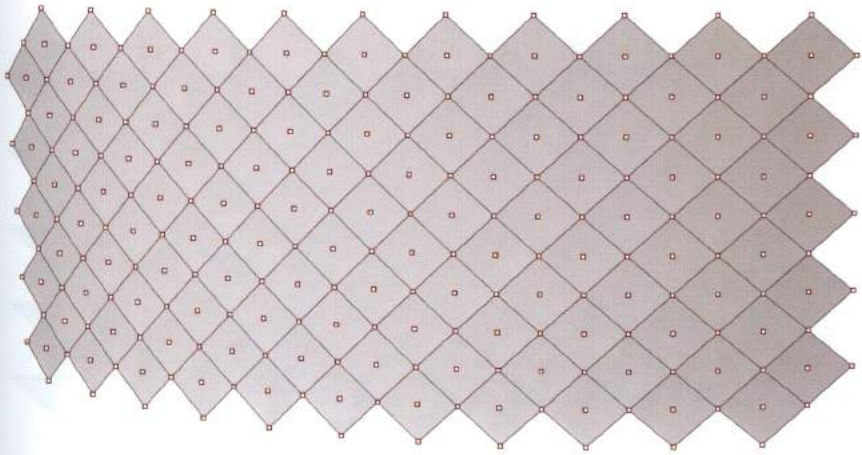
The final example combines a subdivision algorithm with the **Image Sampler** component (4.3.3), to generate a fading tridimensional pattern informed by a grayscale image



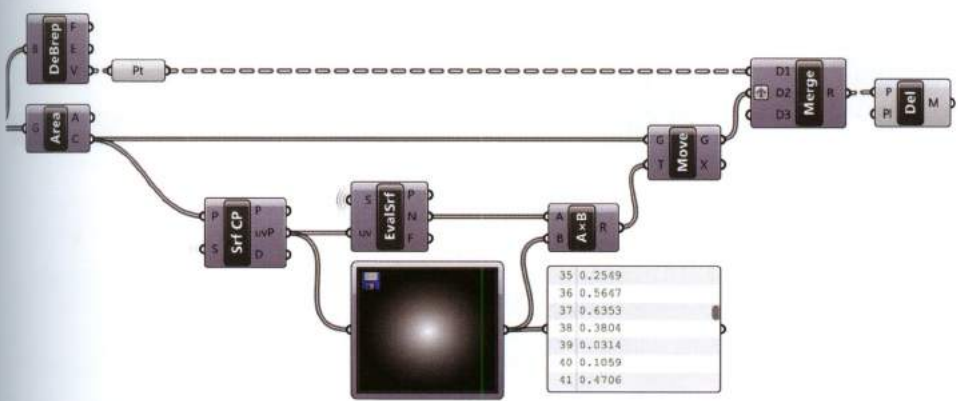
Similar to previous exercises, a bi-dimensional grid is generated from a NURBS surface. The **Lunch Box** plug-in component *Diamond Panels* (LunchBox > Panels) outputs quadrangular and triangular surfaces; the first step in creating a fading tridimensional pattern is to extract the surface vertices and calculate the centroid of every quadrangular (or diamond) surface.



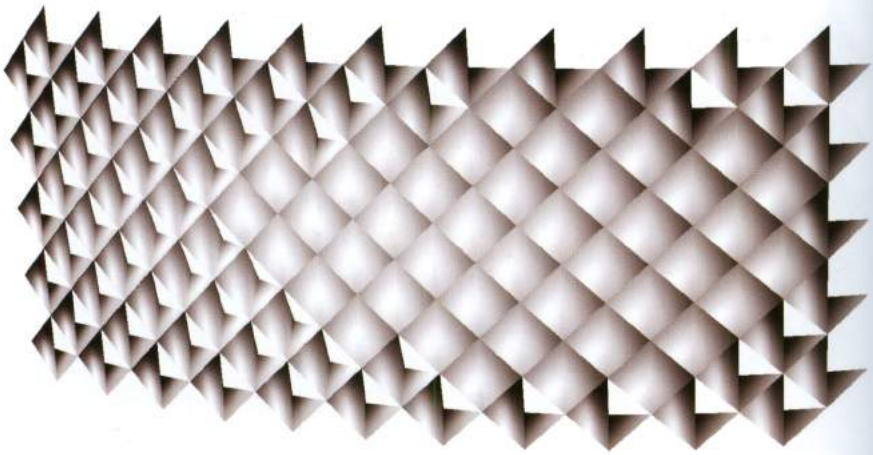
Next, the centroid of each surface is translated according to a scalar factor multiplied by the surface normal unit vector. The four bounding vertices and the translated centroid for each face are merged into a single Data Tree branch. The component *Delaunay Mesh* creates a mesh pyramid from each branch.

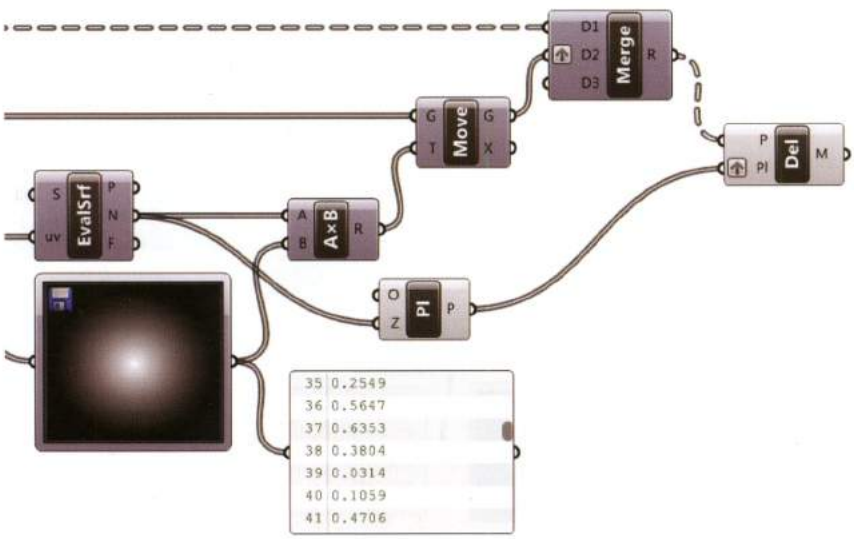


To create a fading tridimensional pattern a non-constant centroid translation is required. As displayed below the *Image Sampler* component can be used to provide non-constant scalar factors.



The resulting pyramids display an error because of the different orientations of each pyramid. To correct this fault a plane must be defined for each diamond-surface using the component *Plane Normal* (Vector > Plane); with input (Z) defined by the output (N) of the *Evaluate Surface* component. The calculated plane for each surface output (P) is connected to the grafted input (Pl) of the *Delaunay Mesh* component. Grafting is required to match the outgoing data of the *Merge* component.





After properly structuring the pyramid data to triangulate, the output of *Delaunay Mesh* component is merged with the second output (Triangles) of the *Diamond Panels* component, to define regular edges. The final step is to smooth the flattened list of joined and welded meshes using the *Catmull-Clark* component.

P
Pl
Del
M
M
Flip
M
D1
D2
D3
Merge
R
M+
W
wbJoin
M
M
L
S
wbCatmullClark
O
Triangles (Srf)
V Count
1
S
U
V
H
Q
Mesh UV
M
Toggle
True

6.6.3 Strategy: cull adjacent faces

Complex shapes are often composed of an aggregation of mesh-boxes, as shown in the image below. Once the boxes are joined together, internal overlapping faces can be removed according to the following algorithm.

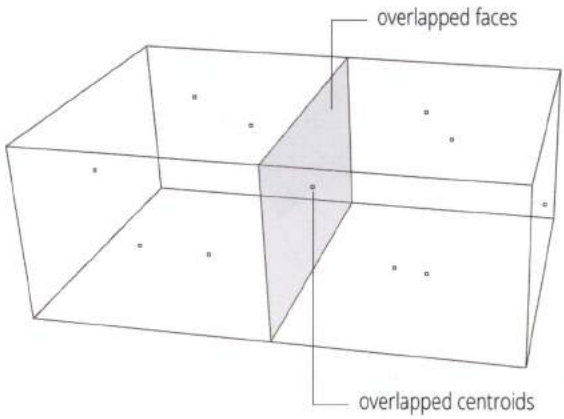


FIGURE 6.15
Often we need to remove the internal faces of a joined mesh made up of mesh-boxes.

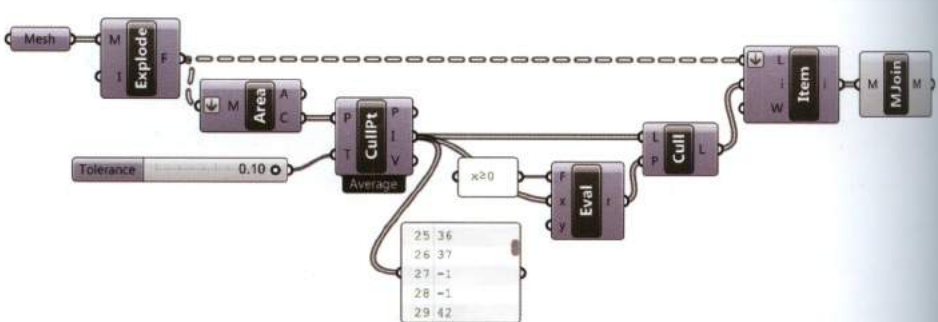


FIGURE 6.16
Cull Duplicates component culls points that are coincident and gives us the index of non-overlapped faces.

Digital informing creativity

Ludovico Lombardi

Lead Architect at Zaha Hadid Architects

The relationship between the tools used to investigate space and the way we understand space, and therefore the way our mind engages the creative process of designing, is a constant dialectic and dynamic process. The influence of the tools, understood both as theoretical researches and, in this case, as computational platforms, is marking the avant-garde experimentations in today's architecture, product design and fashion scenes.

Throughout history theoretical and technological research has influenced the creative mind of architects and artists, and the technical progress informed in a broader way in a mental set up that informed and reconfigured artist and architect's creativity. The theorizations of the perspective system triggered a visual and spatial revolution that defined the Renaissance period spatially both in its physical and intellectual spatial articulation and organization. In the same way the parallel rules defined the modernist way of understanding and designing spaces. The same influence is visible today in the use of computational and code based design.

The great advantage of computational design is not cad based systems, but the possibility to engage with generative systems, understood as generic code based system, able to manifest themselves in a dynamic and adaptive way based on the basic codes and parameters that define them. If generative systems allow designer's creativity to redefine shapes, forms, configurations and organization systems as relational, it is also reflects in a new aesthetic register, the aesthetic of relations or the aesthetic of sublime (understood as an aesthetic of legible forces).

The sublime is derived by an aesthetic that is the result of forces applied into a generative system, which are legible in the configurations and variations resulting by the system itself. Examples of this aesthetic are present in different fields and different domains, from the sublime aesthetic of a flock of birds in the sky constantly reconfiguring and morphing in new formations, to the gothic constant morphing of the same element from column to vault nodes or ornament in a continuous way, or the landscape representations of Ruskin.

While none of the examples described above necessitate computational design to manifest themselves, the legibility of their sublime aesthetic remain true to what I was trying to define as the

new aesthetic informed by computation design. Today's avant-garde designers use computational tools to manage the complexity present in natural system, as in the gothic or as in Ruskin paintings. It is also the tool that allows to negotiate the complexity of contemporary heterogeneous and dynamic society of multitude.

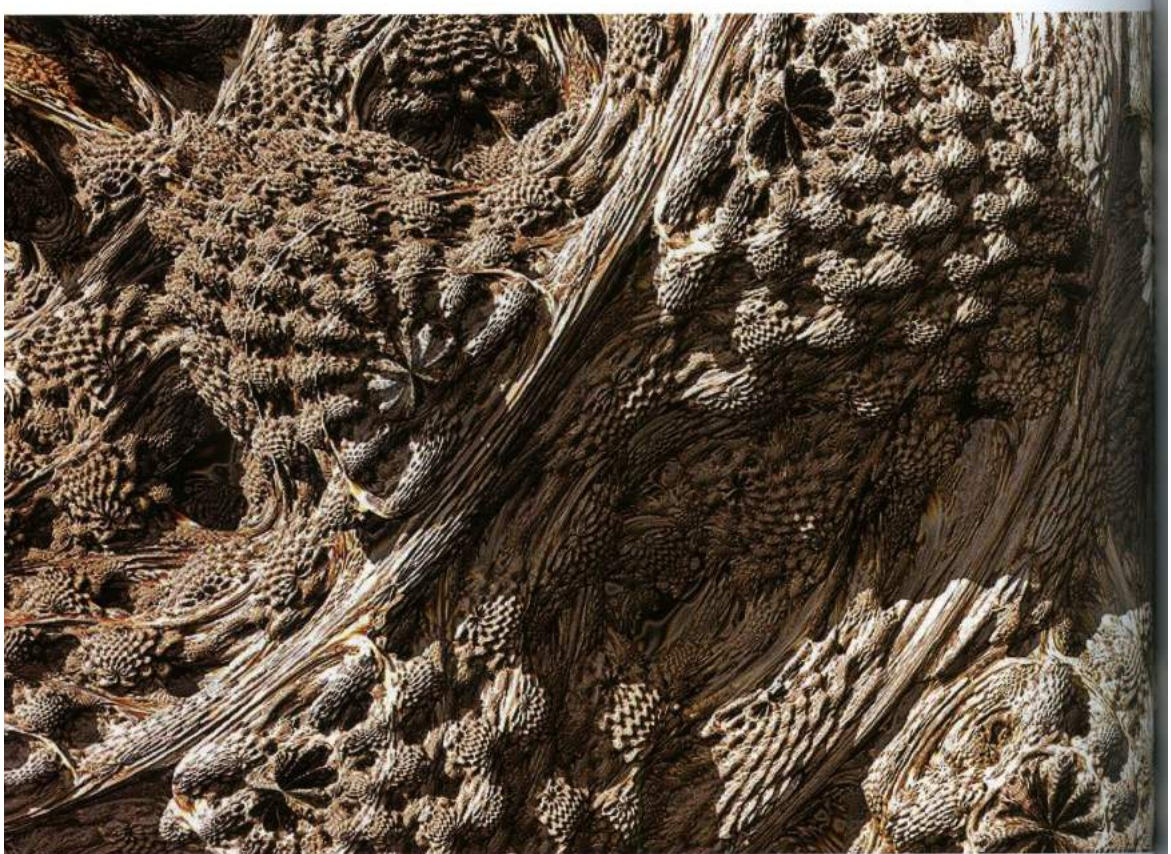
Black Swan necklace, Ludovico Lombardi (2014).

On the other hand there is a superficial and Manneristic overproduction of computational based designs that are not relevant to the above discourse, but that can be defined as a simplistic experimentations triggered by the easy access of the new code based computational platforms. This is probably the most pertinent way to explicitate the need and the role of a designer, with its sensibility and its rational. The creative process is enriched and informed by computational tools, but those do not substitute the role of the creative mind. The misinterpretation of generative system, as not driven by the designer creativity, but in the specific just by computational tools, is the triggering factor of most of today's criticism to computational based design. Understanding that design is relational first than even its physical manifestation, is the paradigm that brings back the role itself of the designer creativity while coding the relations that generates the outcomes of those complex system.

The repositioning of the role of the designer in the above statement hints also to the counterpart of the new design approach, which is the distinction between designer and technical virtuoso. While both have different merits, one does not implies the other and the distinction is still valid as it always was. There is the need of a common understanding and knowledge sharing, but the two roles are not always or don't always to coincide. The specificity and technicality of code based design is still something that necessitates a strong logic and mathematical background, while the understanding of its set up and rules is what a designer today needs to be exposed and engage with. Variations, inflection, self-similarity and integrations become parameters of evaluation of computational design aesthetic, and their definitions is an a-priori from the their formal representation, embedded in the set-up of generative systems and the relations within the system itself. The creative process is readable at different stages, from the moment when the designer is establishing those relations within the set-up of the system, to the moment when the manifestation in the physical domain gets informed by materiality, engineering and production methods. All of those factors are now integrated in what can be defined as a complete creative moment. Today's designers are borrowing tools and referring to a vast number of fields tangent to design, allowing for a re-definition of both aesthetic and logic of space and forms. There is a strong common research manifesting in parallel in different fields, from architecture to design to fashion, and in the most progressive courses in universities and design schools. This legible academic direction is something that will shape the future generation of designers and will define a new type of creativity and aesthetic, which will reflect in our daily life in a constantly increasingly way.

Ludovico Lombardi (LDVC) promotes design by research through computational and material processes. The avant-garde design research operates on multiple scale, from urban design, landscape design, architecture and industrial design. The articulation of complexity and organization through elegance and sophistication become the paradigm to negotiate contemporary heterogeneous and dynamic society of multitude.
Ludovico's work experience in Spain, Italy and England includes Carlos Ferrater, Arata Isozaki and Zaha Hadid architects, where he currently works. In 2008 he graduated with a Master's degree in Architecture and Urbanism from the DRL Design Research Lab of the Architectural Association in London after having previously studied in Italy at the Politecnico of Milan and in England at the Bartlett. His work has been featured on a series of international magazines, such as AD, AJ, Abitare among others. His works have been presented and exhibited at the Architecture Foundation in London and in touring exhibitions in Italy and in the US. Ludovico has appeared as guest critic at the Architectural Association, at the Bartlett UCL and for the DRL and has been invited to lead workshop and lecture in Turin, Bologna, Florence and internationally at the Tate Britain in London and at RISD Rhode Island School of design. Ludovico was recently invited to collaborate with Abitare magazine and was appointed unit master for the M.arch in Urban Design program at the Bartlett UCL for the academic year 2009/10. He was recently appointed Faculty Professor at RISD architecture department.

http://www.ldvc.net/



7_loops

"...this is a curious state of affairs and a reflection that ordinary rules break down at infinity".

Cecil Balmond

Recursive algorithms define earlier input values by later output values, which are determined by the algorithms execution. The procedure can repeat itself N times, generating an output at the end of each iteration or step. The first iteration starts with an *edge condition*, i.e. one or more elements (numbers, geometries, data) defined not recursively; later iterations are defined by data loops. Recursive algorithms are very powerful because they can be used to generate a multitude of geometries by a short and simple process.

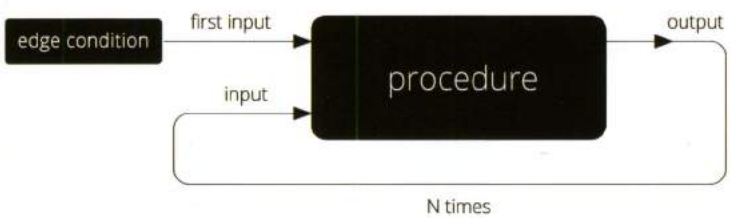


FIGURE 7.1
A recursion starts with an *edge condition* (first input) that generates, by a specific procedure, the first output. The procedure repeats itself N times and, at every step, it generates an output that becomes a new input.

Several numerical sequences can be calculated recursively. For instance, the Fibonacci sequence is

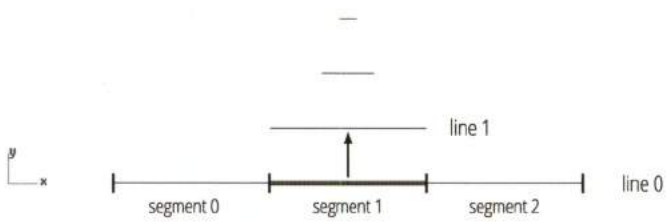
defined recursively by the procedure: $F_n = F_{n-1} + F_{n-2}$. Once the start values or *edge conditions* $F_0 = 0$ and $F_1=1$ are defined the sequence can be calculated.

0, **1**, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

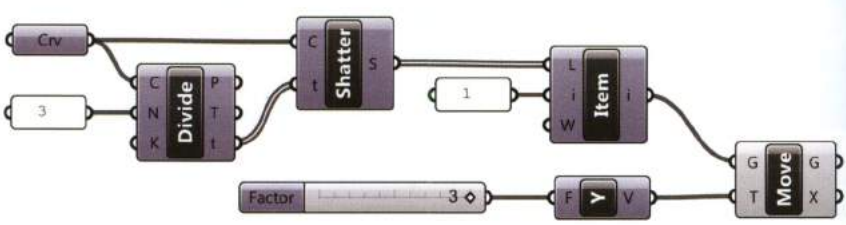
Recursions can be used to achieve ruled complexity, a condition of natural geometries at all scales. The following image shows a geometric configuration based on a simple recursion.

A recursive procedure that repeats itself three times to divide a line is as follows:

- Step 1: draw a line to define an edge condition, the line 0;
- Step 2: divide the line into 3 segments: segment 0, segment 1, segment 2;
- Step 3: move the segment 1 by 3 units according to the y-axis. Outputting line 1;
- Step 4: repeat the step 2 and step 3 using the line 1.



The procedure can be converted into a Grasshopper definition as shown below.



Then again, the converted recursive definition is able to perform just one iteration, outputting the

line 1 as returned from the G-output of the *Move* component. In order for the definition to execute three times a data **loop** is required. However, a data loop conflicts with Grasshoppers linear-flow logic (cfr.1.6).

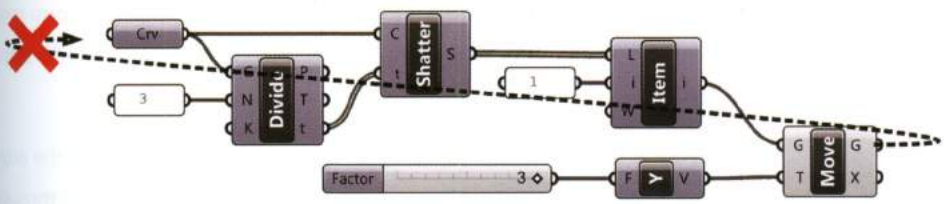
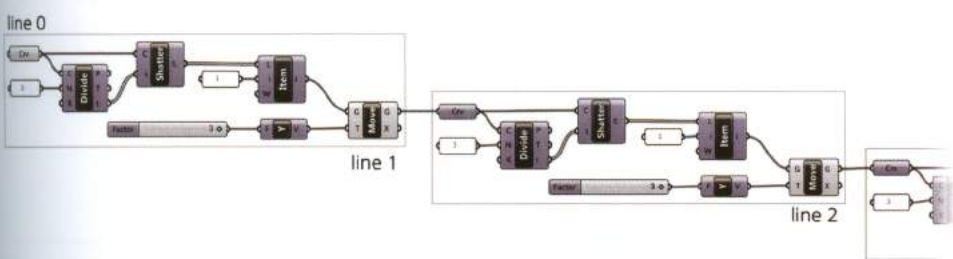


FIGURE 7.2
Loops are not allowed in Grasshopper because of the "left-to-right" connection logic.

According to Grasshoppers left-to-right connection logic, the only way to execute the procedure three times would be to repeat the algorithm. Fortunately, the plug-in components ***HoopSnake*** and ***Loop*** enable feedback loops to be defined in Grasshopper.

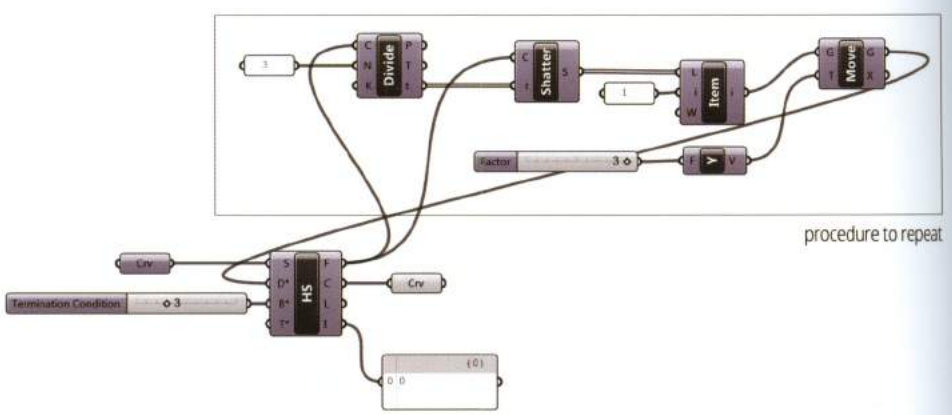


7.1 Loops in Grasshopper: HoopSnake component

HoopSnake is a plug-in for Grasshopper developed by Yannis Chatzikonstantinou. The plug-in is available for free download on his website[18]; after installation a new component will appear in the Extra tab. *HoopSnake* enables users to defy the left-to-right connection logic and perform loop operations. With reference to the previous example and to the next image we can easily understand how the component works:

1. The linear curve (line 0) set from Rhino by a *Curve* container component is the edge condition. The curve is connected to the S-input of *HoopSnake* and passes through the component without any change, meaning the output (F) is equal to the input (S).
2. All the inputs that in previous example were connected to *Curve* (C-input of *Divide Curve* and C-input of *Shatter*) must be connected to the F-output of *HoopSnake*.
3. The G-output of the *Move* component (last component of the procedure) is connected to the D*-input of *HoopSnake*, defining the loop.
4. The input (B*) specifies the number of loops to perform, and is best supplied by a *Number Slider*.
5. The output (C) returns the cumulative output of all iterations, each as a branch of a Data Tree.

The procedure is composed in the following definition:

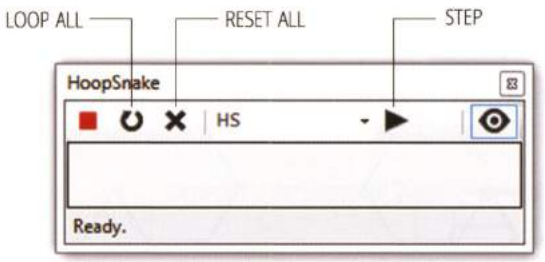


NOTE 18

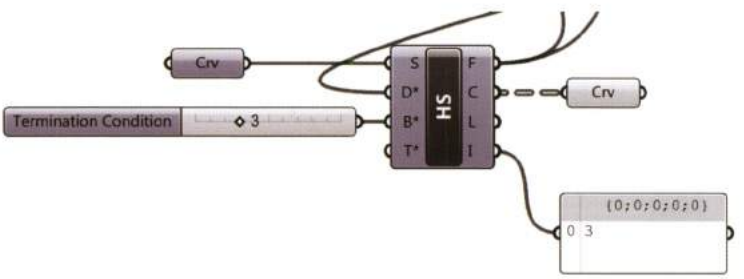
http://yconst.com/software/hoopsnake/.

In alternative, you can download HoopSnake at: http://www.food4rhino.com/project/hoopsnake

HoopSnake operates as an engine and, in order to perform iterations, the component must be started. To start the engine double-click the component to recall the control panel. Clicking the control panel's *Step* button performs a single loop while clicking the *Auto Loop All* button performs the number of iterations as specified by B*.



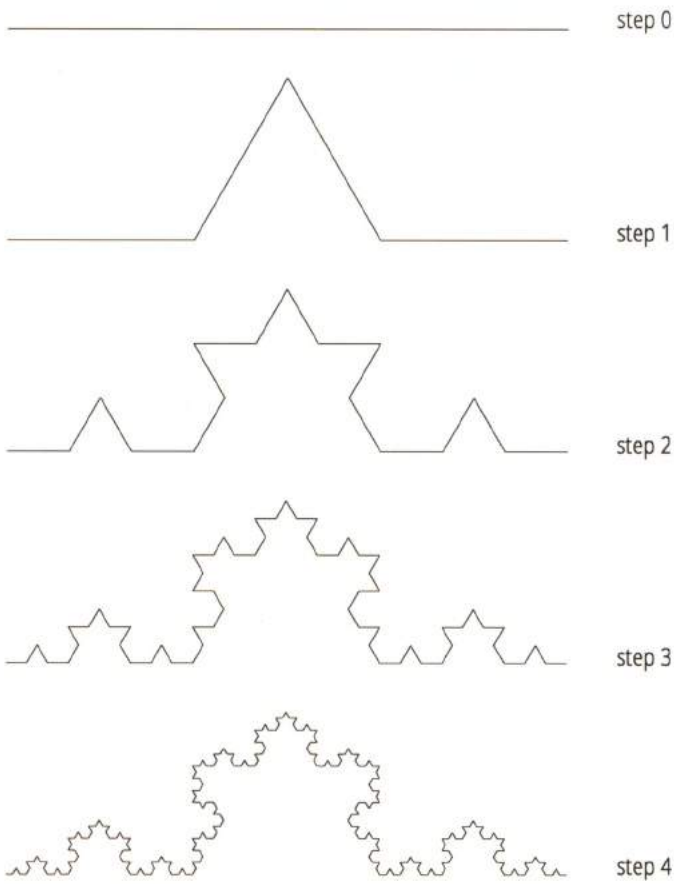
The output (I) provides the iteration count, when *Auto Loop All* button is run the recursion terminates when the counter is equal to the value set in B*.



7.2 Fractals

Fractals are patterns that are self-similar across scales. They are generated by repeating a simple procedure using feedback loops. Fractals repeat at increasingly smaller scales producing complex shapes.

A well known example of a fractal pattern is the **Koch snowflake curve**. The *snowflake curve* studied by the Swedish mathematician Helge von Koch (1870 - 1924) is one of the earliest fractal curves to have been described.



The Koch curve is found as the limit of an infinite sequence of approximations. The first approximation is a straight line segment (step 0). The middle third of this segment is then replaced by two segments (whose length is equal to the middle third) which are joined like two sides of an equilateral triangle (step 1). In the step 3 each line segment has its middle third replaced by two segments which form an equilateral triangle. *"We can regard the limit of this infinite process as being a curve that actual exists, if not in physical space, then at least as a mathematical object"*[19].

NOTE 19

R. Rucker, *Infinity and the Mind: The Science and Philosophy of the Infinite*, Princeton University Press, 2004.

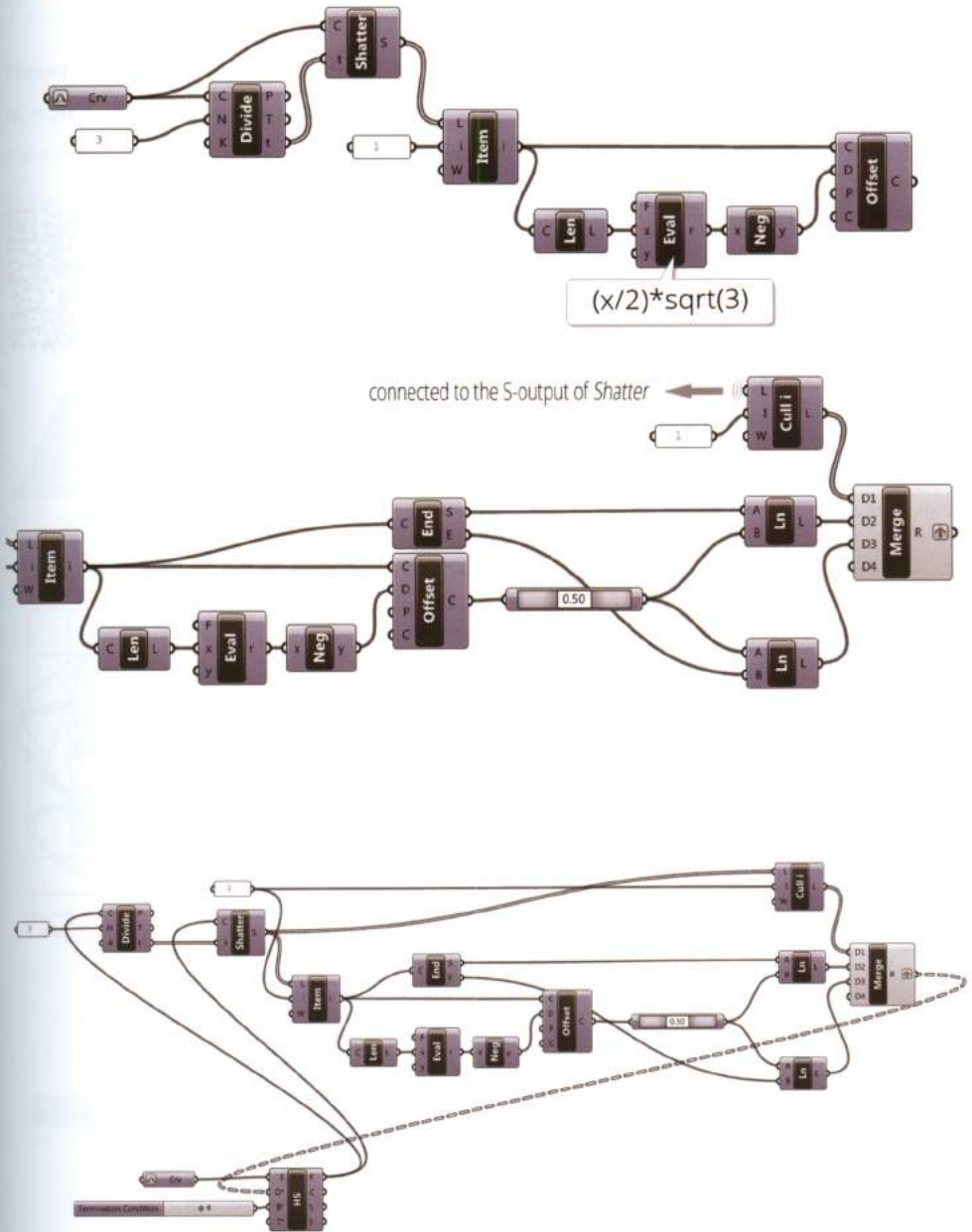


FIGURE 7.3
Koch curve algorithm developed using HoopSnake.

7.2.1 Further Study: Practical Fractals

In this example, Toyo Ito's and Cecil Balmond's 2002 Serpentine Pavilion will be reproduced. Despite the complexity of the final structure, the apparently random pattern was in fact derived from an algorithm based on a simple rule:

"I propose an algorithm: half to a third of adjacent sides of the square. The 1/2 to 1/3 rule traces four lines in the original square that do not meet [...]. The half to a third rule forces one to go out of the original square to create a new square so that the rule, the algorithm, may continue [...] and a primary structure is obtained. Then if these lines are all extended, a pattern of many crossings results. Some are primary for load bearing, some will serve as bracings to secondary and the rest will be a binding motif of the random across the surface of the box typology". (Cecil Balmond)

2

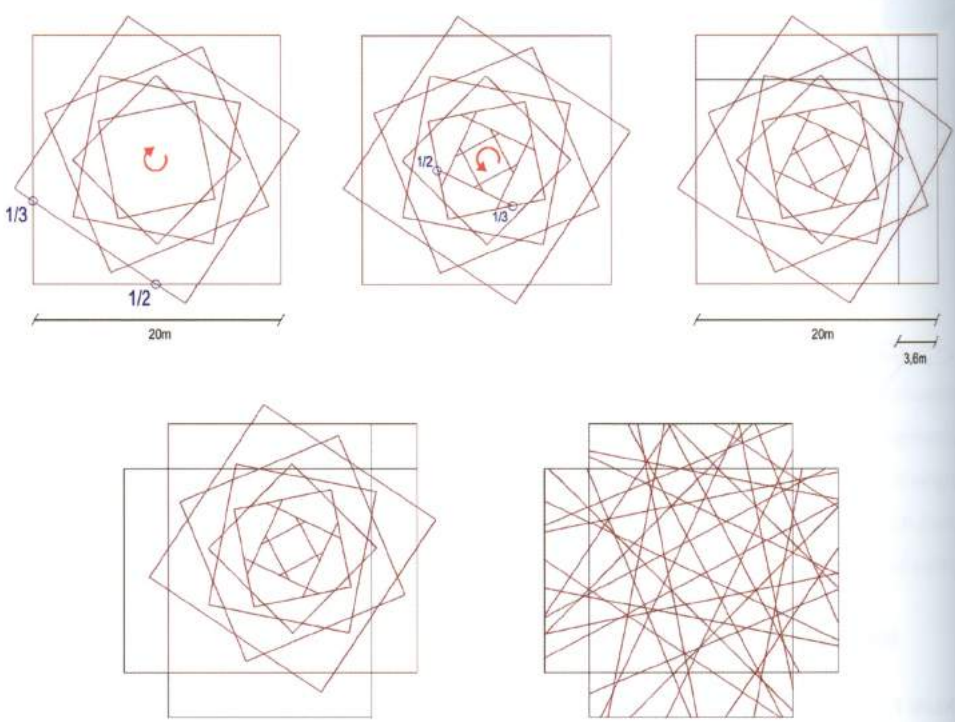


FIGURE 7.4
Serpentine Pavilion 2002, schemes of iteration.

7.2.2 Tridimensional fractals

Tridimensional fractals follow the same logic as bidimensional fractals. In the following example a 3D fractal is created from three points and performing three iterations. The procedure to create the tridimensional fractal is as follows: first, a triangular mesh is defined through three points by *Delaunay Triangulation*, second, each face's centroid is translated, and finally, three new faces are defined using *Delaunay Triangulation*. This process can be repeated through *n* iterations.

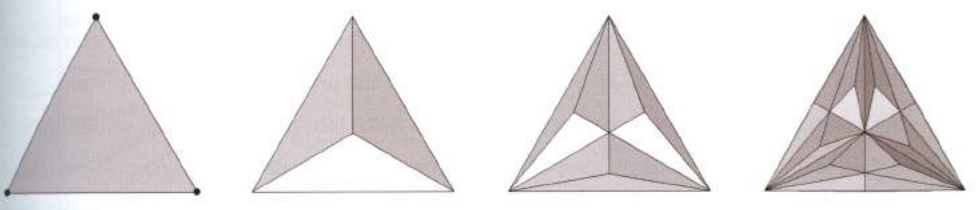
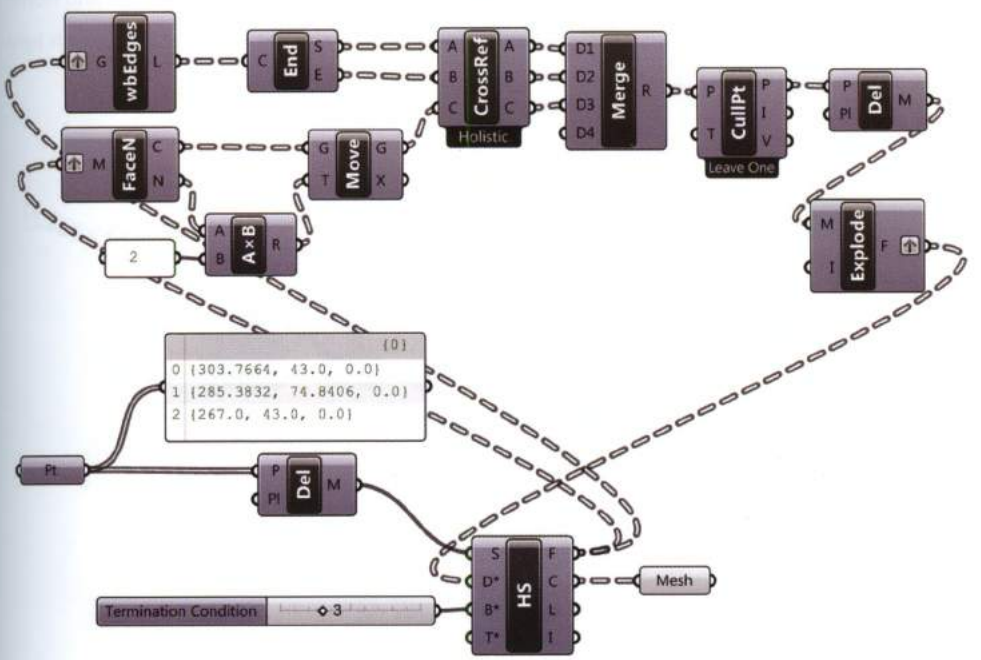


FIGURE 7.5
A tridimensional fractal obtained by three iterations of a recursive algorithm.



7.3 Loops in Grasshopper: Loop component

An alternative to *HoopSnake* is provided by the *Loop* plug-in developed by Antonio Turiello[20]. The *Loop* method uses three components: *Rnd*, *Store* and *Loop*. The first two components, *Rnd* and *Store*, represent the explorative part of *Generation*[21], an add-on which provides additional tools to explore, animate and fabricate generative shapes with Grasshopper. *Rnd* generates a list of pseudo random numbers updated by a *Timer* (Params > Util) or by the *Recompute* command (Solution > Recompute). The *Store* component stores data between updates.

The component *Loop*[22] is used to iterate a procedure by replacing an input parameter of the procedure's initial component with an output parameter of the procedure's final component.

In the following definition, the *Rnd* component generates three interpolated random values within a domain. Two of these three values correspond to the parametric coordinates of a point P(u,v) placed on an initial sphere. The third value corresponds to the radius of a new sphere (stored in the *Store* component) which is tangent to the initial sphere at point P(u,v). The *Loop* component (connected to a *Timer*) updates the *Rnd* component and replaces the reparameterized S-input of *Evaluate Surface*, with the G-output of *Move*. Following this logic, in each iteration a new sphere is created and stored, exploring different compositions of dark-colored spheres with smaller radii combined with light-colored spheres with larger radii.

NOTE 20

Antonio Turiello is a Generative Designer, Independent Researcher and Authorized Rhinoceros Trainer.

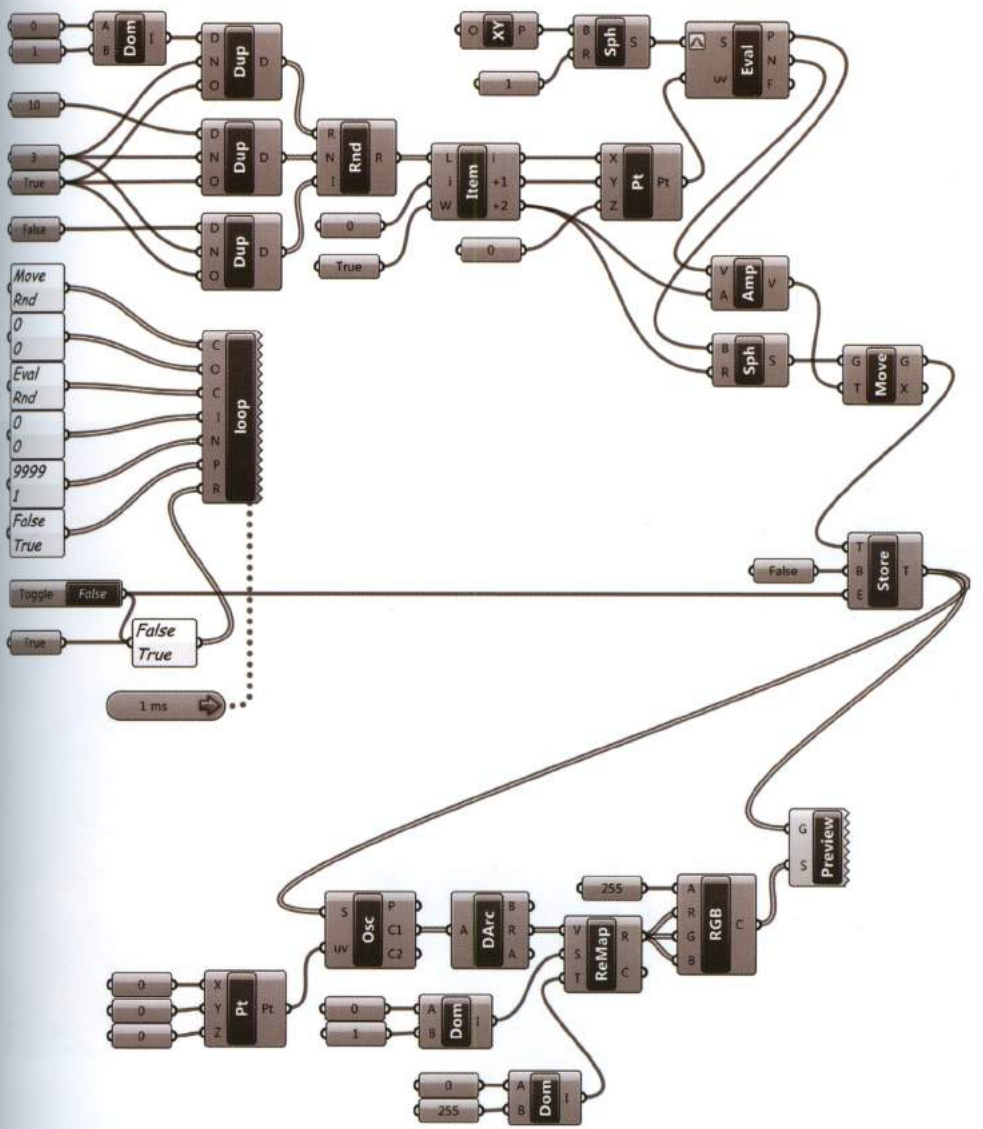
http://antonioturiello.blogspot.com/

NOTE 21

http://www.food4rhino.com/project/generation

NOTE 22

http://www.food4rhino.com/project/loop


Dom
Dup
Dup
Dup
Rnd
Item
XY
Sph
Eval
Pt
Amp
Sph
Move
Move
Rnd
Eval
Rnd
9999
False
True
loop
Toggle
False
True
False
True
1 ms
False
Store
Preview
Osc
DArc
ReMap
RGB
Pt
Dom
Dom

Danecia Sibingo (under the supervision of C. Walker and M. Self), Driftwood Pavilion. AA School, London 2009.

8_digital fabrication

make ideas come true

"Any sufficiently advanced technology is indistinguishable from magic".

Arthur C. Clarke

The link between design and fabrication has always been crucial for architecture and industrial design. New manufacturing techniques, assembly processes, and materials, often prompt paradigm shifts in design. For instance, the introduction of structural steel fabrication to the design of structures enabled new formal ambitions; but required new expertise in detailing and pre-fabrication. This new approach, based on the design of component, progressively marginalized handcraft and made the designer essential within the building process.

Before the *digital revolution*, the designer managed the complexity of building projects by breaking down components into individual parts, and studying part to whole assembly strategies through scale models and drawings. The scale models and subsequent construction drawings were then transferred to suppliers and contractors to interpret and fabricate the assembly components. This process was limited to orthogonal structures, and lacked the capacity to respond to complex shapes, with few notable exceptions such as Jorn Utzon, Heinz Isler, Antoni Gaudí etc.. The digital revolution liberated this constraint by integrating design output directly with fabrication.

The first stage of *digital revolution* focused on controlling the project by generating holistic digital tridimensional models. The digital model was then interpreted by the computer to generate bi-dimensional plans, sections, elevations and details. Informed **digital fabrication**, and in particular **CNC**[23] processes, directly link 3D geometry to the final components bypassing the production of drawings. Digital fabrication or the automated production of components improves accuracy and makes complexity ordinary; since a complex operation will have the same level of machining difficulty as a simple operation.

Since building processes are directly interpreted from 3D geometry, designers produce traditional drawings to merely aid in the assembly of components. Drawings can also be omitted by printing assembly instructions directly on components, or more futuristically by providing robots or drones with coded instructions. The ambition to directly translate an idea into reality is already possible for small scale objects or components. In other words, it is possible to create a physical object from a 3D virtual model using **Rapid Prototyping** (RP). Rapid Prototyping is an additive fabrication technique in which material is deposited in layers to print a component. This technique will likely have a resounding impact on the future of manufacturing and construction.

8.1 Fabrication Techniques

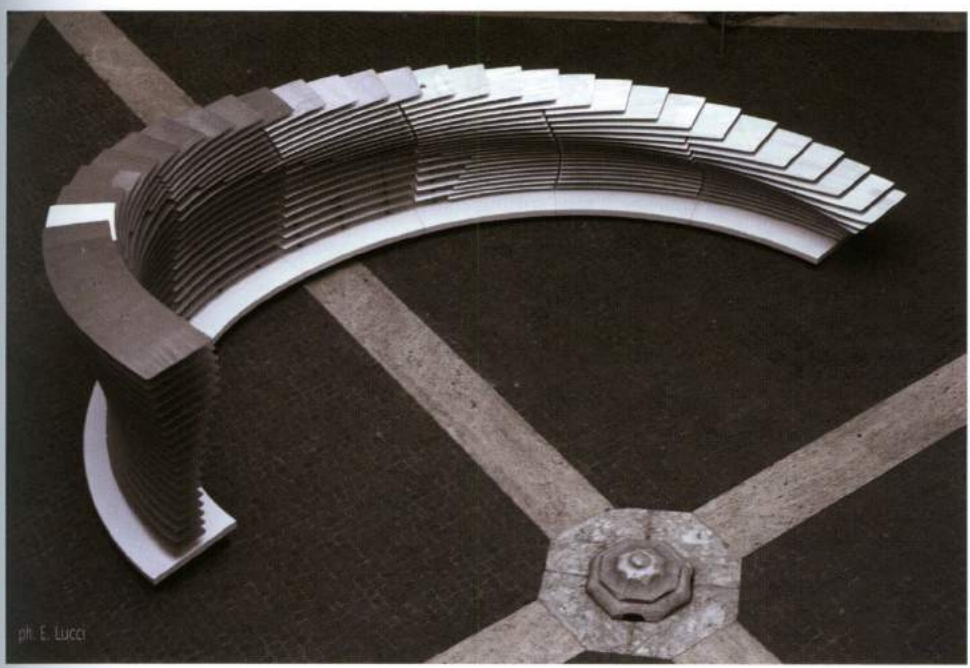
Fabrication techniques can be sorted according to processes and materials.

8.1.1 Bi-dimensional Cutting

Bi-dimensional fabrications transform planar sheets of: aluminum, steel, plywood, acrylic etc. of varying thicknesses into components for tridimensional assemblies. **Planar sheets** can be used to form geometries using techniques such as: stacking, faceting etc.. Flat sheets can also be **bent** to form complex developable geometries, or geometries with null Gaussian Curvature.

NOTE 23

The acronym "Computer Numerical Control" identifies the use of computer in driving and controlling a machine movement. The machine can be a milling cutter, a lathe, a laser or waterjet cutter.



ABOVE. The NU:S installation (2012) designed by Arturo Tedeschi and Maurizio Degni was made by overlapping planar sheets of aluminum cut by a waterjet machine. BELOW. A sculpture by Carlo Borer (2011) created by bending and connecting shaped metallic plates. The sculpure's geometry is a set of zero Gaussian Curvature surfaces. Image courtesy of Galerie Frank Pages.

Cutting-based processes include:

- **CNC Laser[24] cutters**

  Laser cutters burn or melt material using a focused laser set to a specified power and cutting speed. The cutting speed and power are set based upon the materials dimensional and physical properties. Laser cutting cannot be used for every material; for example, aluminum sheets will reflect the laser.

- **CNC Plasma cutters**

  Plasma cutters cut material using a focused stream of super heated gas or plasma. Plasma cutting is widely used to cut steel and aluminum.

- **CNC Waterjet cutter**

  Waterjet cutters cut materials using a focused jet of water combined with an abrasive substance. The major advantage of the Waterjet is that it does not create *heat-affected zones* were the molecular structure is modified. The Waterjet can be used for a wide range of materials from steel to wood.

FIGURE 8.1
A waterjet machine in action while is cutting a 5 mm aluminum sheet. Image courtesy of Lamberti Design.

NOTE 24
The word LASER is an acronym that stands for "light amplification by stimulated emission of radiation".

8.1.2 Subtractive techniques

Subtractive techniques, such as CNC milling, create objects by removing material. Subtractive methods can achieve the same output of cutting machines with the added ability to specify the Z depth of a cut.

FIGURE 8.2
The Driftwood Pavilion was created using 3-Axis mill from large-scale wooden planks.

Solid blocks of materials from wood to polystyrene can be milled to carve described geometry. Not all geometry can be achieved using 3 axis milling; and in many cases objects must be decomposed into several parts to prevent undercutting.

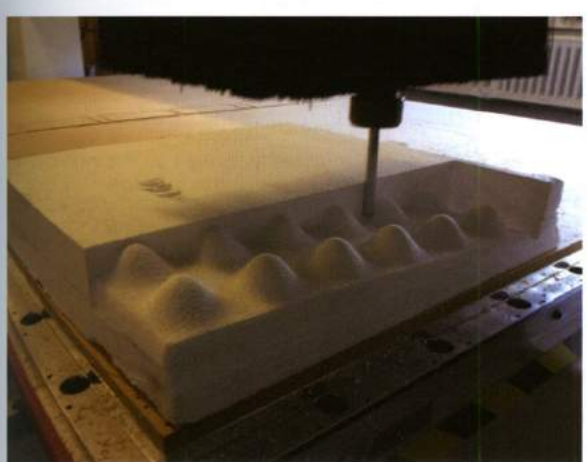


FIGURE 8.3
The Illinois School of Architecture, Rheotomic Farm; Mock-Up Milling. Gaelan Finney-Day, Przemek Swiatek, Brian Vesely.

FIGURE 8.4
The Illinois School of Architecture, Kiva; Mock-Up Milling, Arch 576 Post Digital Strategies, Critic: Brian Vesely.

Subtractive techniques include:

- **CNC milling machines**

  Milling is a tooling process that uses a cutter head to remove material from a sheet or block of material. Mills vary in their size and depth capacity as well as the number of axis that the cutter head can be manipulated. The most commons machines are 2D, 2.5D, and 3D machines: 2D mills cut at a specified Z depth similar to cutting machines, 2.5D (two and a half) mills operate in 3 axes but only perform operations in two axes simultaneously, and 3D mills perform operations in all three axes simultaneously. More advanced 5 axis mills move in four or more axes to create custom parts with limited tooling restrictions. Milling machines are informed by digital geometry which is used to describe tooling paths.

FIGURE 8.5
Aura, by Zaha Hadid Architects (design team F. Wirz and M. Lanza), exhibited at Villa Foscari "La Malcontenta" in Mira, Italy (2008). The sculpture was created relying on a 6 axis CNC machine using polyurethan foam as material to mill.
Image courtesy of Zaha Hadid Architects. Image copyright by Luke Hayes.

- **Hot-wire foam cutter**

  Hot-wire cutters use an electronically heated wire to cut polystyrene foam or similar materials. Several different types of hot-wire cutters exist from wires able to cut on a single plane to cutters able to cut on multiple planes. This method is used to quickly generate tridimensional shapes.

- **Robotic arms**

  Robotic arms can be used to facilitate other fabrication techniques, such as: holding a milling cutter head or heated wires, grasping and folding, forming etc.. Recent research in academia as well as avant-garde professional offices, have investigated design possibilities enabled by the high degree of flexibility provided by the robotic arm.

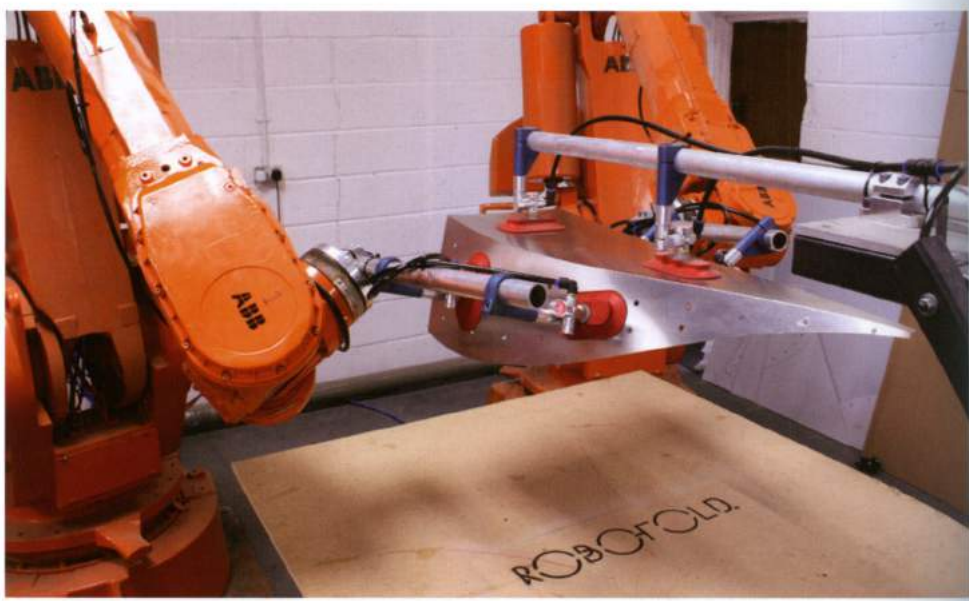


FIGURE 8.6
Objects made from formed sheet metal are usually produced with a stamping press or expensive moulds. The RoboFold technology (www.robofold.com) allows to form sheet metal by using robots. Image courtesy of RoboFold.

8.1.3 Additive techniques

Additive Manufacturing (AM) creates tridimensional objects by an accumulation of successive layers. Terms such as, **Rapid Prototyping** or **3D printing** are increasingly used to describe additive manufacturing. Additive manufacturing enables designers to fabricate objects that are impossible to make using subtractive techniques. Objects such as branching shapes, complex twisting, moving parts with intricate details can be realized, albeit at a small scale.

Additive manufacturing is defined by describing deposition paths. Deposition paths are created by slicing **a digital model, developed in Rhino or another modeling software, into layers with a defined slice thickness or resolution. The contours describe how material is deposited, layer by layer**.

The AM process involves three steps:

1. Creation of a digital 3D model. *A printable model is required to meet the criteria discussed in 8.2.*
2. Conversion of the digital model into a code based machine-readable format.
3. 3D printing the object.

The most commonly used 3D printing format is Standard Tessellation Language (STL), an interface developed by 3D Systems®. An STL conversion triangulates an initial tridimensional model and outputs the coordinates of each vertex according to the right-hand rule. The number of triangulation iterations can be equated to the smoothness of the final model, meaning the more triangles the smoother the model. The output can be described in ASCII and Binary. Binary is the more common language because of the smaller file dimension. The most common 3D printing techniques are: stereolithography (SLA), selective laser sintering (SLS) and fuse deposition modeling (FDM).

- **Stereolithgraphy (SLA)**

  SLA printing was developed in 1983 by Charles Hull. A SLA printer consists of four main elements: a vat containing a photosensitive liquid form resin, a mobile perforated-platform, a UV beam, and a computer that controls both the beam and the platform. The platform is initially positioned on the top of the vat, just below a thin layer of resin. As the UV beam strikes the resin, the resin selectively solidifies and a layer is formed; as defined by one layer of the *sliced* digital model. The platform is then lowered the distance of one layer to print the next slice, this process continues until all the sections are created. After the object is completed, it is rinsed with a liquid solvent and baked in a UV oven to cure the plastic. Resin that is not solidified remains in the initial liquid form. Un-solidified liquid is unable to support overlying parts; for this reason, it is often necessary to create support structures. SLA printers can produce very-high-resolution objects, but are slow and expensive.

- **Selective Laser Sintering (SLS)**

  SLS printing was developed in 1986 by Carl Deckard at The University of Texas Department of Mechanical Engineering. SLS printers (see figure 8.7) use a high-power laser to selectively fuse powdered particles of plastic, metals, ceramic, glass etc.. An SLS printer consists of four main elements: a laser, a powder cartridge, a roller, and a fabrication platform. A thin layer of powdered material is spread by a roller across the fabrication platform where the laser traces a bidimensional section of the object, sintering the material together. The platform is then lowered the distance of one layer thickness and subsequently the roller deposits new material from the powder cartridge to be sintered forming the next layer. This process continues until the part is completed. SLS does not require support structures since the sintered parts are surrounded and supported by the non-fused powder.

FIGURE 8.7
The Selective Laser Sintering (SLS) process is based on a high-power laser beam, a levelling roller, a powder cartridge and a fabrication platform

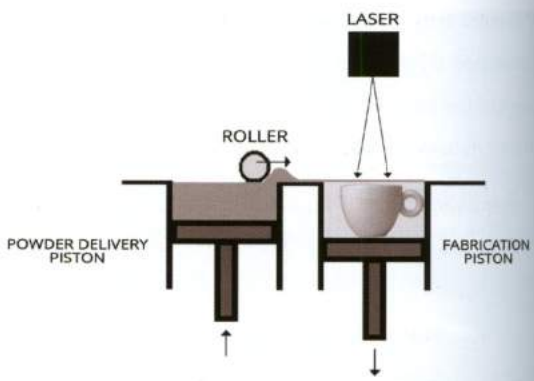


FIGURE 8.8
NU:S parametric shoes (2012) designed by Maurizio Degni, Alessio Spinelli and Arturo Tedeschi were made by a SLS process using nylon powder. Image courtesy of SOLIDO 3D.

- **Fuse deposition modeling (FDM).**
  FDM printing was developed in 1988 by Scott Crump. FDM printers form geometry by melting plastic filament and depositing thin layers additively on a platform. The FDM printers, similar to SLA printers, require support structures for overhanging parts. Some FDM printers can print multiple materials, in this way supports can be printed in a material that is easy to remove; such as water soluble filament. FDM printers use two types of plastic; ABS, or an organic version: PLA.

Other innovative techniques have been tested recently; such as the MX3D-Metal printer developed by Joris Laarman in collaboration with the software company Autodesk. MX3D-Metal printing utilizes

a robotic arm to deposits molten metal onto an existing metal surface. Autodesk notes of this innovation: *"The arm is controlled by new software Autodesk created that can give the robot more fluid instructions for where the metal should go. Because of how quickly the metal hardens, the new object doesn't need an additional support structure"* (Autodesk).

Conceptually, additive techniques revolutionize the topic of **optimization**. Traditionally, optimization has been linked to simplifying fabrication techniques such as: cutting operations, complexity through similarity, describing developable surfaces or planar panels etc.. Optimization of additive techniques involves finding the optimal shape which meets a prescribed set of performance targets; such as minimally using material. Additionally, advanced form-finding strategies such as **topology optimization** (chapter 10), are becoming increasingly important in architecture and product design.

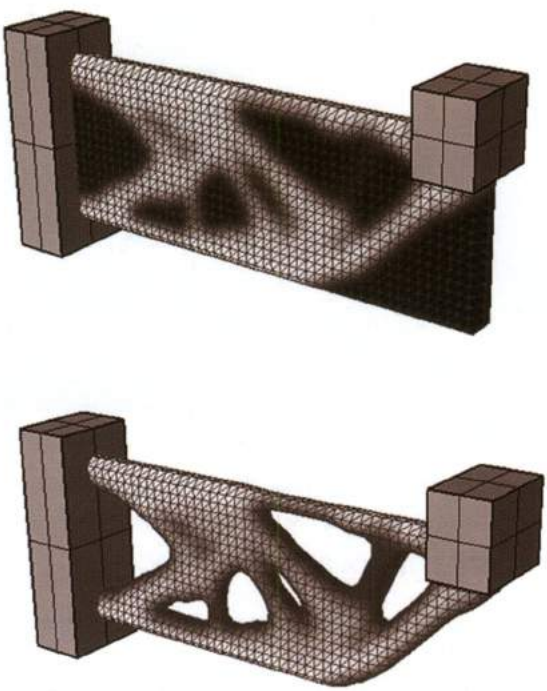


FIGURE 8.9
Topology optimization drives material distribution within the volume of the truss.

8.2 Modeling Printable Objects

Translating digital ideas to physical objects using 3D printing is currently viable for **small scale objects**. The definition of a *small scale object* is constantly changing; since advances in 3D printing technologies are enabling increasingly larger objects to be printed. Currently the single printing-session dimensional limits of high end commercial printers are 1000L x 1000W x 500H mm.

Producing digital models for 3D printing is not technically different from producing conventional 3D models. Similar to all forms of fabrication, when using additive fabrication consideration should be given to the constraints of the selected printing technology as well as real world characteristics. Moreover, the way geometry is defined to create a printable model is crucial.

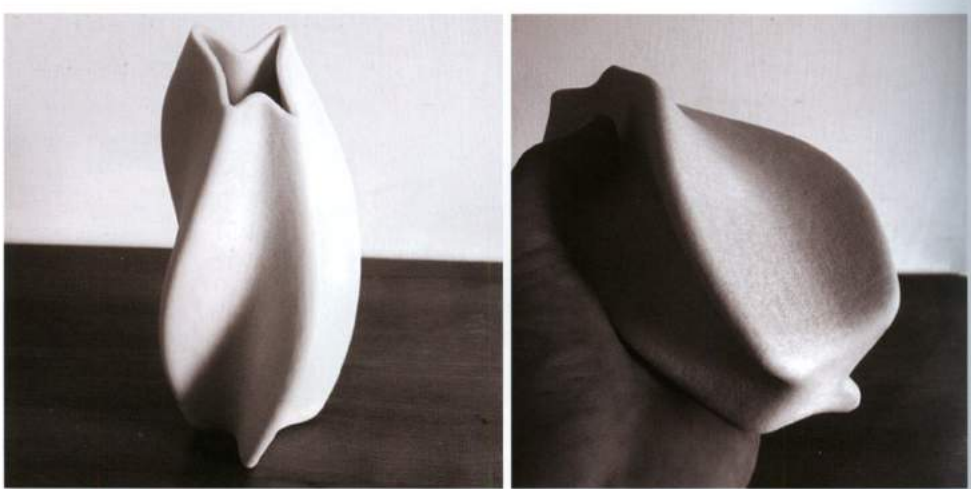


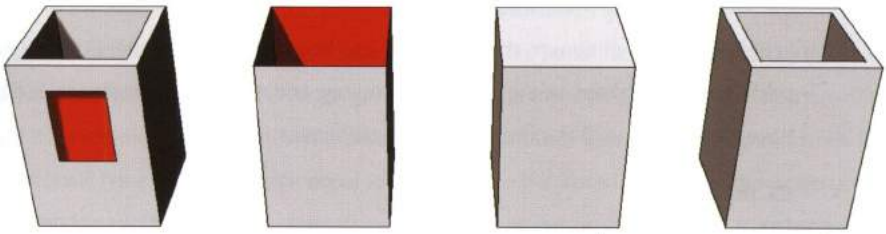
Figure 8.10
Pandora | AL, Arturo Tedeschi (2011), made by a SLS process using nylon powder.

8.2.1 Main characteristics of a printable 3D model.

Printable 3D models differ from 3D models developed for visualization and rendering. Printable objects must consider: geometric digital modeling criteria, the specifics of the machine that will "read" and print the generated code, and the characteristics of the materials used for printing.

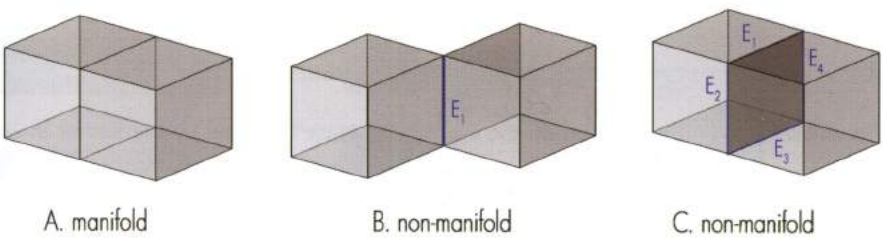
The geometric characteristics of a printable object are:

- **G1. Closed (watertight) geometries**
  Each surface or mesh has an *inside* face (colored in red in the image) and an *outside* face (colored in gray). A 3D object is *watertight* and printable if no inner faces are visible.



- **G.2. Manifold mesh**

  Only manifold models will be printed correctly. A geometry is called manifold if does not contain edges shared by more than two faces. Non-manifold geometries can be the result of inconsistent models (image B) or overlapped objects (image C), issues that are problematic when printing. Paragraph 6.6.3 explained how to remove overlapped faces from a mesh.



- **G3. Orientable mesh (correct normals)**

  The geometry must be an orientable mesh. In other words, the 3D model must be composed of face normals which follow the same directional logic.

- **G.4. No self-intersections**

  The geometry must not be composed of intersecting non-booleaned objects.

A printable object must also meet the following constraints:

- **C1. Maximum size (or build volume)**

  Objects cannot be larger than a maximum size of the ***build volume*** or ***tray size***. These values are measured in XYZ dimensions and are specific to each printer. Models that are larger than *build volume* can be divided into smaller parts which can be assembled into the larger component.

- **C2. Minimum wall-thickness**

  Is defined as the minimum thickness that can be printed by a printer. The minimum wall thickness varies based on the specific technology and material. For instance, SLS prints can have a minimum wall-thickness of 0.7mm using plastic materials.

- **C3. Resolution**

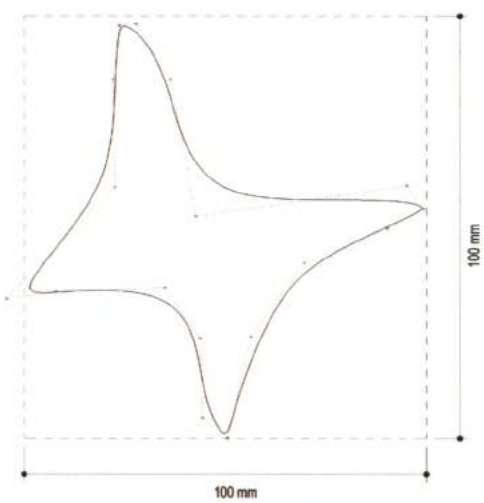
  The *horizontal (XY)* and *vertical (Z)* resolutions are based on: the specific technology, the material, and the overall quality of the desired print. The horizontal resolution is the smallest movement that the extruder or print head can make horizontally and is expressed in microns (100 micron = 0.1 mm). The vertical resolution is the thickness of a 3D printed layer. As follows, details smaller than the horizontal or vertical resolutions cannot be printed.

- **C4. Gravity**

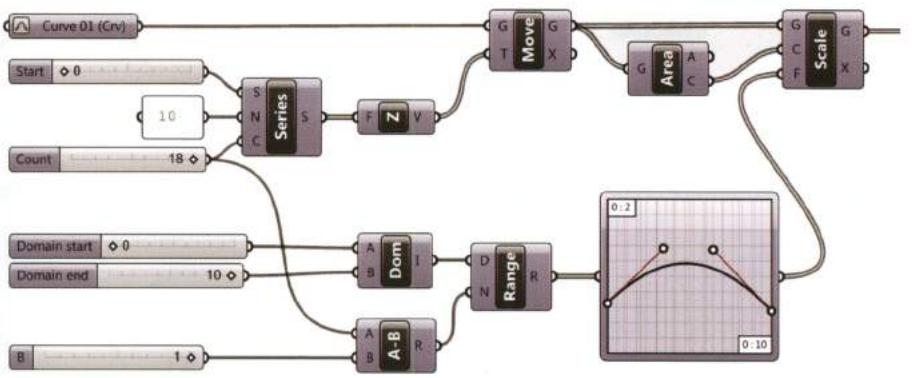
  The printed 3D model is a physical object that must obey natural laws when printing. Basic gravity-point tests and simulations can be performed using external software to check models for successful printability.

8.2.2 Example: parametric modeling of a vase

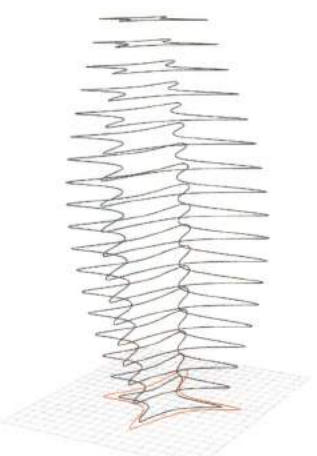
The following example is based on the Pandora|AL vase designed by the author in 2011, using a common desktop printer with a *build volume* of 200 x 200 x 200 mm; and will demonstrate a technique to create printable geometries.



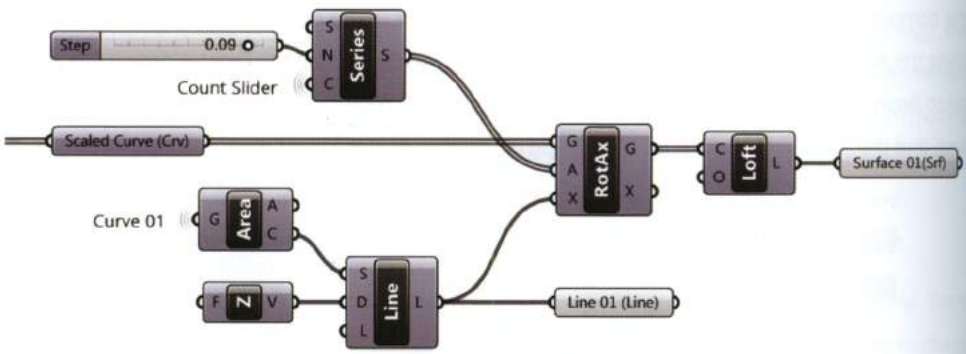
The first step in the algorithm is to define a closed NURBS curve within a 100 x 100 mm square. Next, the curve is translated vertically according to scalar multiplication. Scalar factors generated by the *Series* component are multiplied by the *Unit Z-Vector*, generating a set of translated curves with a step size of 10 mm. The {Z} *Build Volume* constraint is 200mm, as a result the C-input of *Series* component is set to 18. The resulting object will have a {Z} dimension equal to 180 mm. The translated curves are then scaled using the component *Scale* with: center of scaling (C-input) defined as the centroid of each translated curve using the *Area* component, and the scaling factor (F-input) defined using the *Graph Mapper* component set to *Bezier*. Since the {X} and {Y} *Build Volume* constraints are 200mm the Y domain is set between (0,2) such that the maximum scale factor (2) will yield geometry within the build volume.



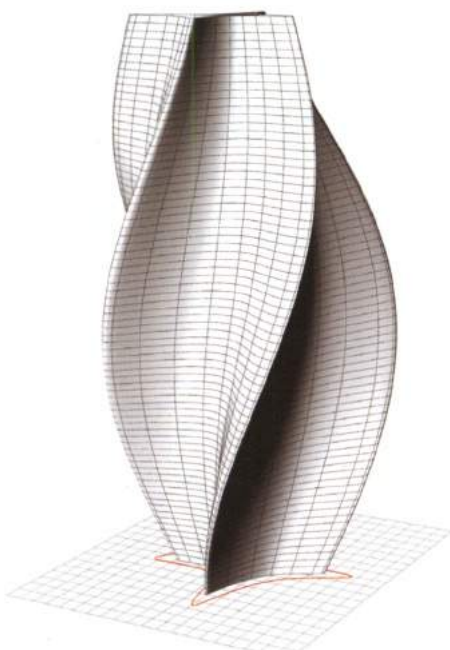
The following image displays the resulting geometry. The initial curve is colored in red.



The second step is to perform a rotation for each translated and scaled curve around a central axis using the component *Rotate Axis* (Transform > Euclidean). The angle of rotation in radians (A-input) for each curve is defined using the *Series* component. Lastly, the rotated curves are lofted using the *loft* component defining an untrimmed surface.



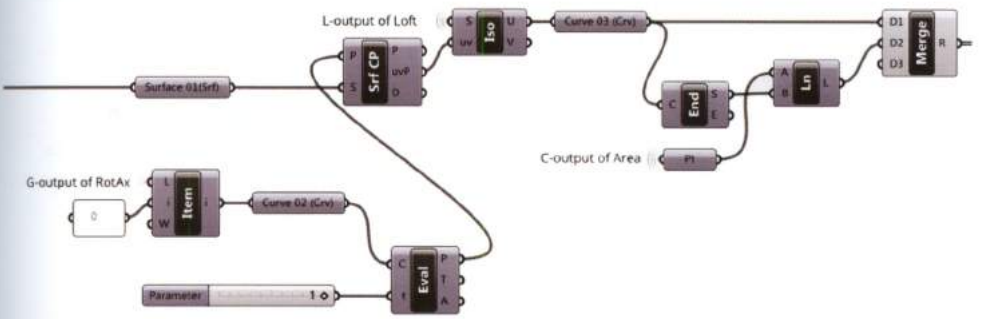
The untrimmed surface is not a printable object since it contains no thickness and is not *watertight*.

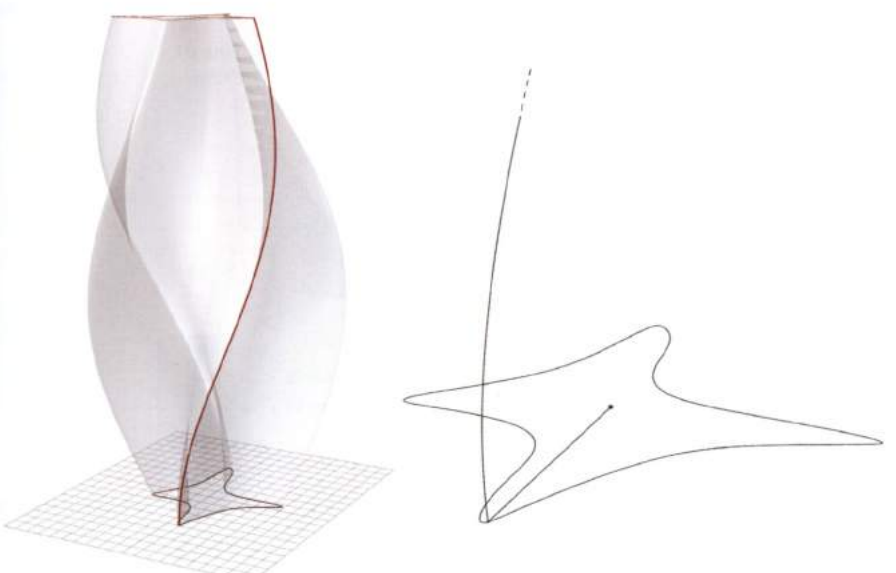


Since the desired output is a vase, a lower cap can be created by the component *Boundary Surfaces*. If the cap and the loft surfaces are joined, then assigned a wall thickness using the component *Mesh*

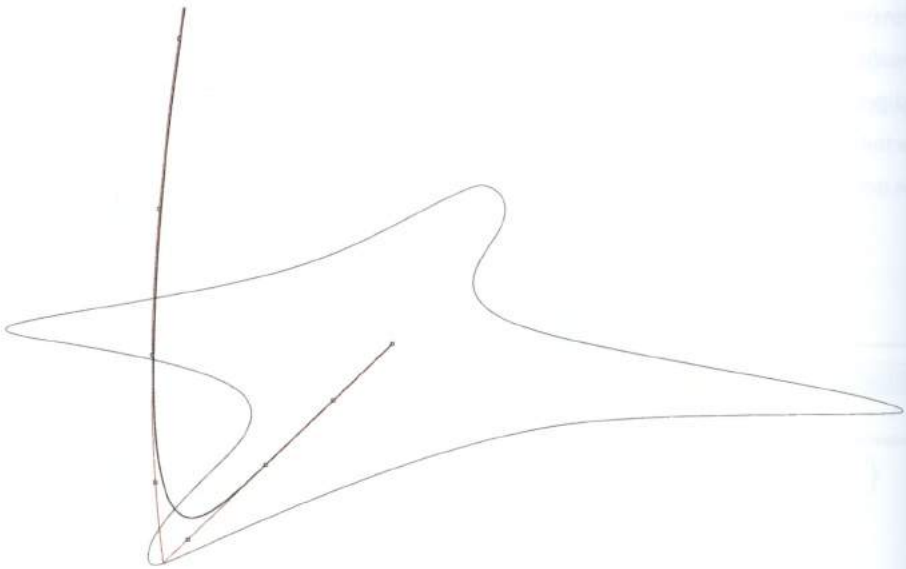
Thicken (Weaverbird > Transform) a printable object will result with sharp edges.

To create a smooth edge an alternative strategy is used. A vertical isocurve is extracted from the lofted geometry at a point defined on curve (0) using the component *Evaluate Curve* with t=1. The extracted isocurve is joined with a curve defined from the centroid of curve (0) and the start point of the isocurve.

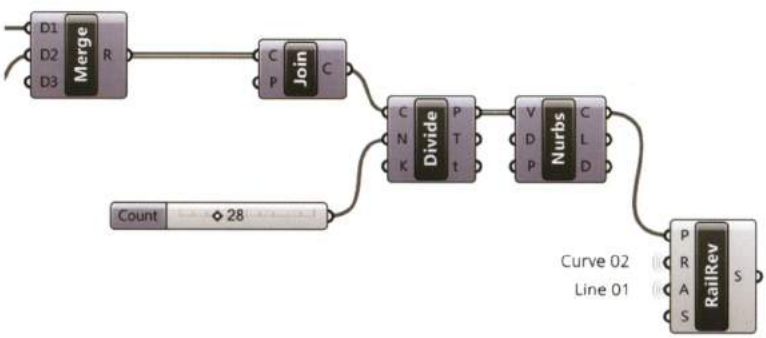




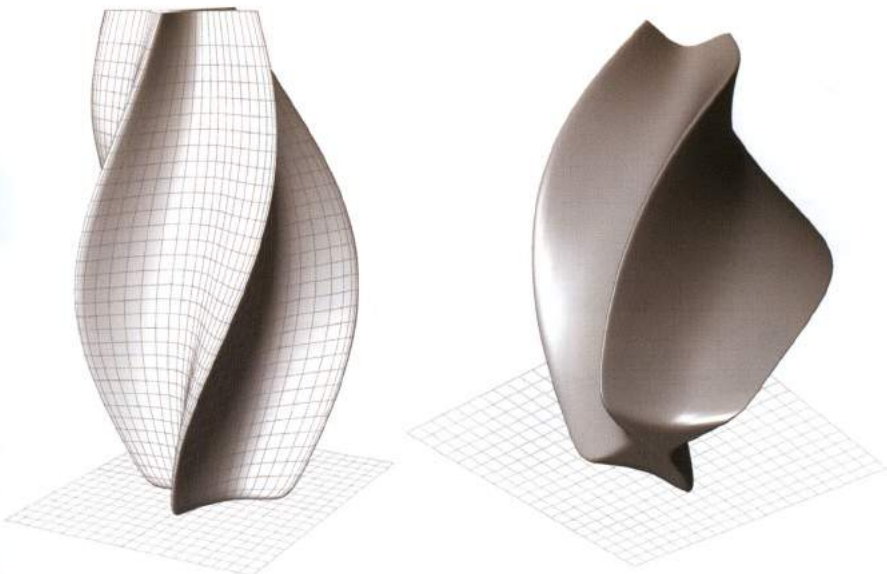
The defined curve is then divided using the component *Divide Curve*. Outputting, in this case 28 points, which are used to generate a smooth, freeform (non-planar) curve using the component *Nurbs Curve*.



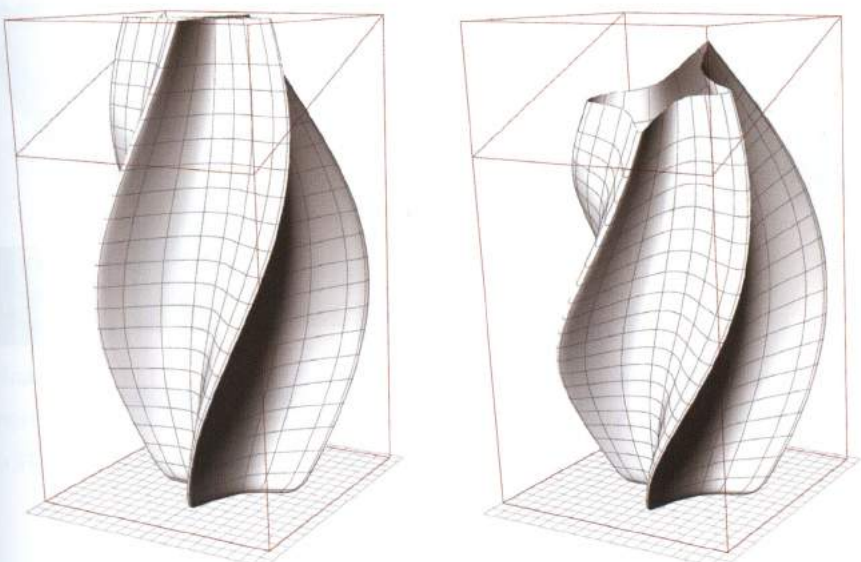
The *Nurbs Curve* is then revolved using the component *Rail Revolution* (Surface > Freeform) along rail curve (R-input) defined by *Curve 02* and a revolution axis defined by *Line 01*.



The output of the rail revolution, displayed in the following image, is a NURBS surface with smooth edges.

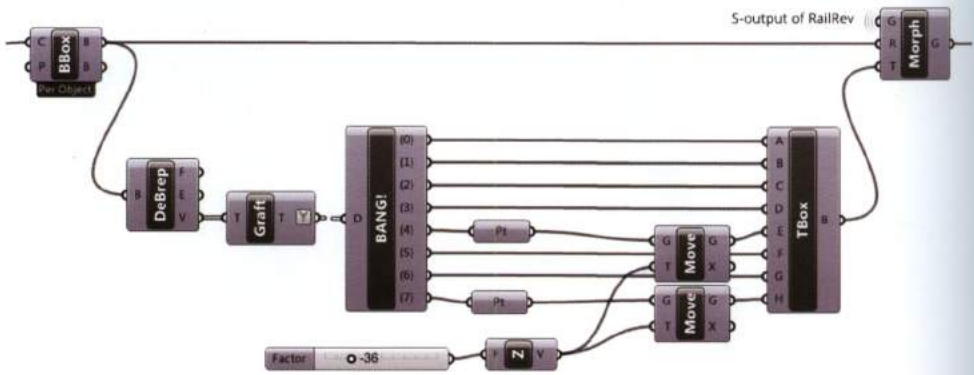


The Resulting NURBS surface is then morphed using the component *Box Morph*.

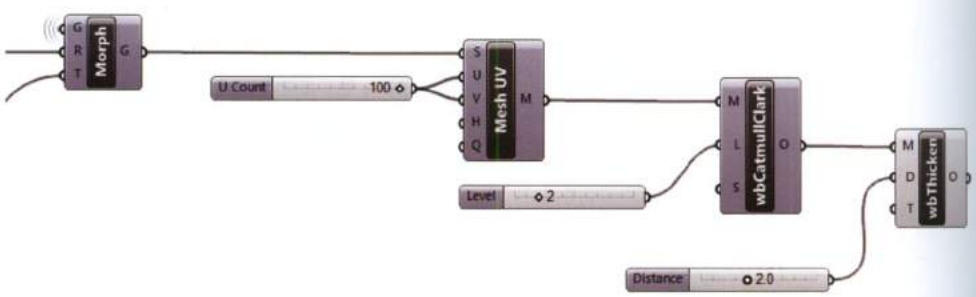


The *Box Morph* component requires a reference box input (R) and a target box input (T) to perform a morphing operation. The R-input is satisfied as the bounding box of the NURBS surface using the

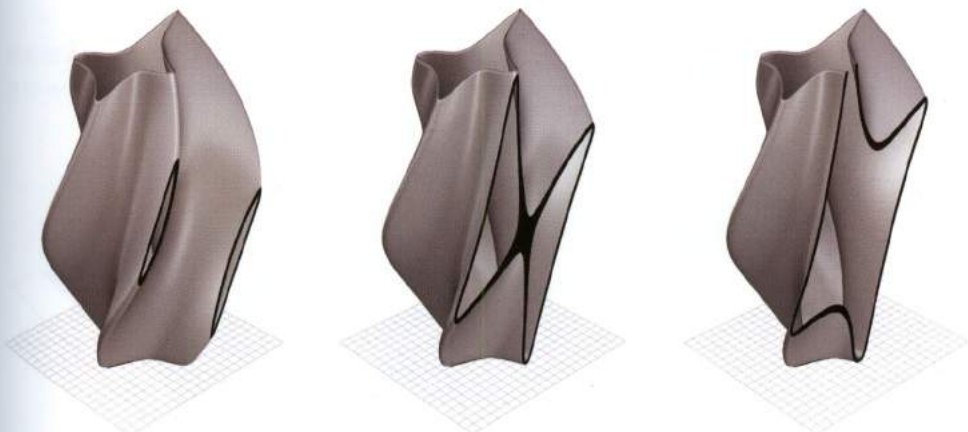
component *Bounding* Box (Surface > Primitive). The target box is a deformation of the bounding box, achieved by extracting and subsequently translating vertices. The manipulated vertices are then rebuilt into a target box using the component *Twisted Box* (Transform > Morph).



The resulting twisted output is then converted into a mesh using the component *Mesh Surface* and is subsequently smoothed using the *Wb Catmull Clark* subdivision method. The resulting smoothed geometry is then assigned a wall thickness of 2.0 mm using the component *Mesh Thicken* (Weaverbird > Transform), defining a *watertight* printable object.



ABOVE. The final mesh is smoothed by the Catmull-Clark component and the watertightness is achievd through a proper thickness.
BELOW. Different vertical sections show the absence of intersections and the perfect watertightness.



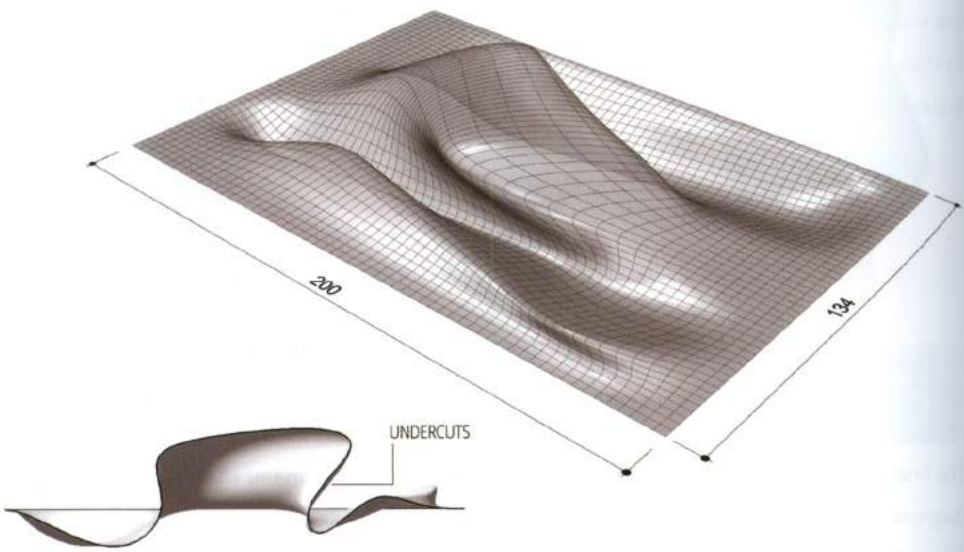
The final mesh can be baked into Rhino, checked through the command *check* which is a tool for diagnosing potential geometry errors. If no errors occur, the mesh can be exported and converted into a STL file (File > Save As or Export > STL) to be transferred to a 3D printer or other software.

8.3 Modeling objects for cutting based operations

Medium-scale objects, or objects that are currently outside of the means of additive processes, can be fabricated using cutting based operations. Medium scale fabrications can be assembled from parts relying on techniques such as sectioning and waffling.

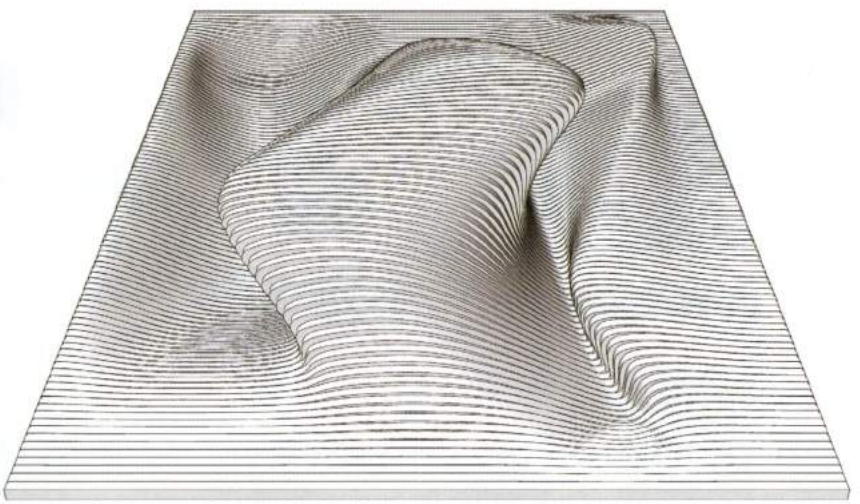
8.3.1 Example: sectioning and waffling

Sectioning or contouring a model in one direction is a common technique used to build complex shapes by defining planar developable surfaces. For instance, a freeform surface can be sliced into a set of parallel and planar sections based on an interval equal to the thickness of the material to be milled. Upon gluing the sections together in sequence, the original surface can be approximated. The surface could also be milled from a single block of material; however undercuts require the use of a 3D milling machines.



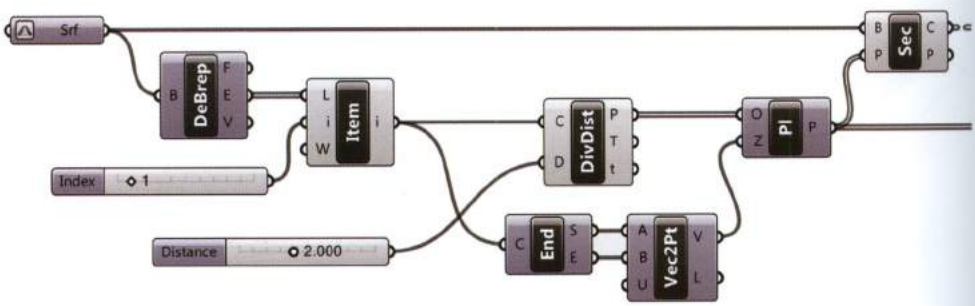
Sectioning creates continuity parallel to the cut section and limited continuity in the direction normal to section-planes. A common fabrication technique is to leave a space between each section to create the illusion of continuity in the normal direction.

FIGURE 8.11
With *sectioning* is impossible to get continuity in the direction normal to section-planes.

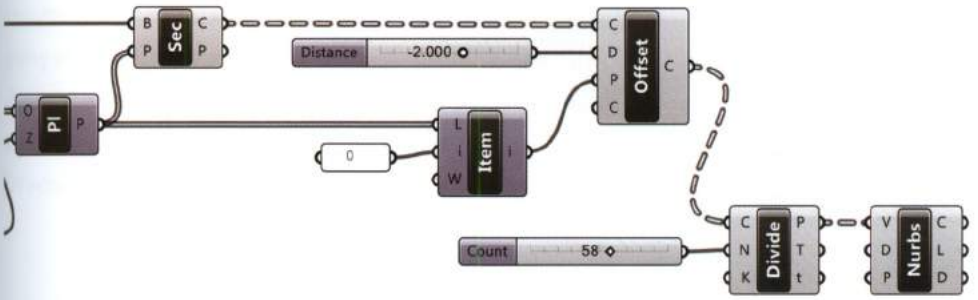


To define an unidirectional sectioning algorithm, a surface created in Grasshopper or set from Rhino is required. The subsequent steps are to define: a slicing direction, a series of planes, and finally intersect the planes with the model; defining the sections. In this instance, the surface's boundary forms a rectangle and one edge can be used to define the origin points for sectioning planes.

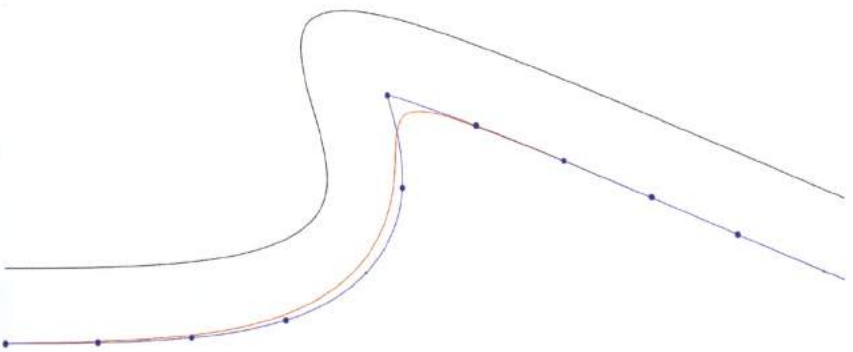
The component *Divide Distance* divides a curve with a preset distance equal to the thickness of the material to mill. At each division point a plane is defined using the component *Plane Normal* (Vector > Plane). Each plane is then intersected with the initial Brep using the component *Sec* (Intersect > Mathematical) to calculated the section curves.



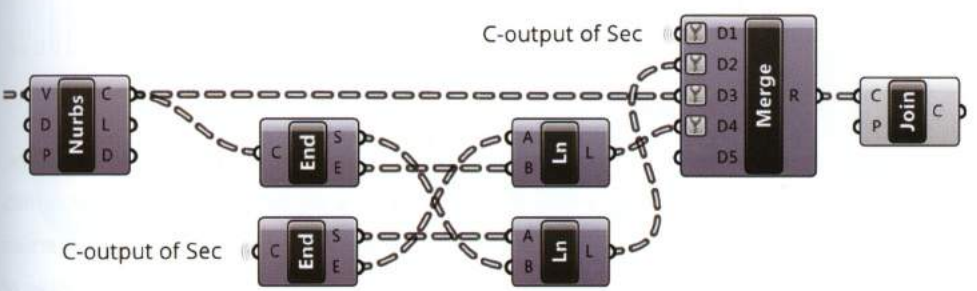
Next, each curve is offset with respect to the defined plane, using the component *Offset* (Curve > Util). Offsetting may yield discontinuities which can be eliminated by dividing the curve into N-parts and rebuilding the curves using the *Nurbs* component.

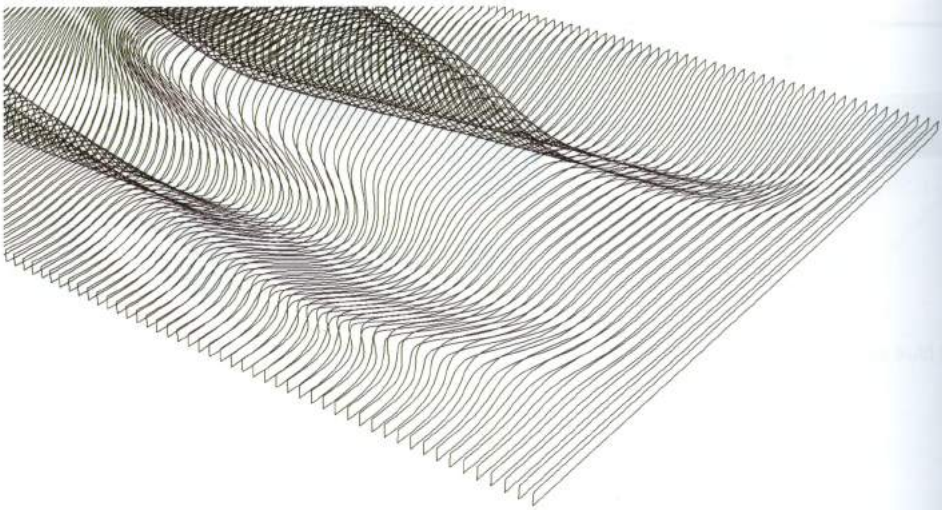


The blue curve is the original curve and the red curve is the rebuilt curve.

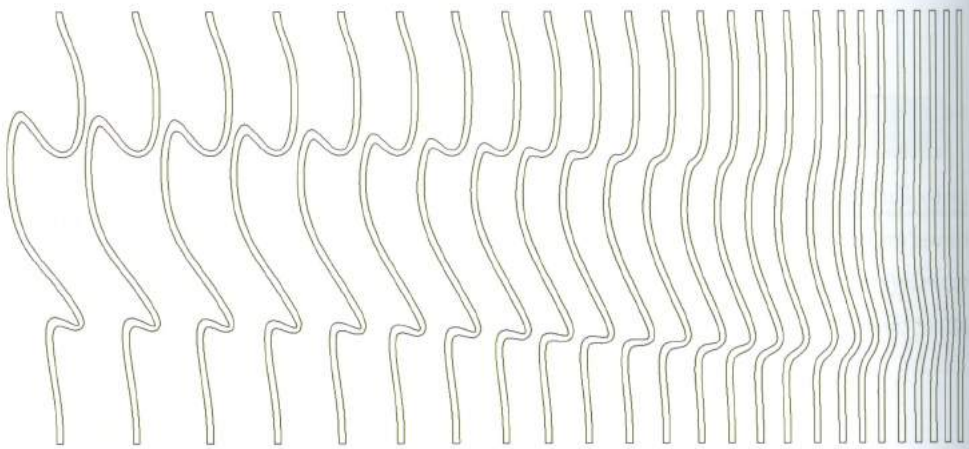


Lastly, each section curve is joined with its smooth offset by defining two segments that connect the curves' end points. The four curves are merged into a single branch using the *Merge* component set to *simplify* mode. The merged branches are joined using the component *Join*, returning a list of *closed planar curves*. These curves can then be oriented to a plane for milling (see 4.2.3).





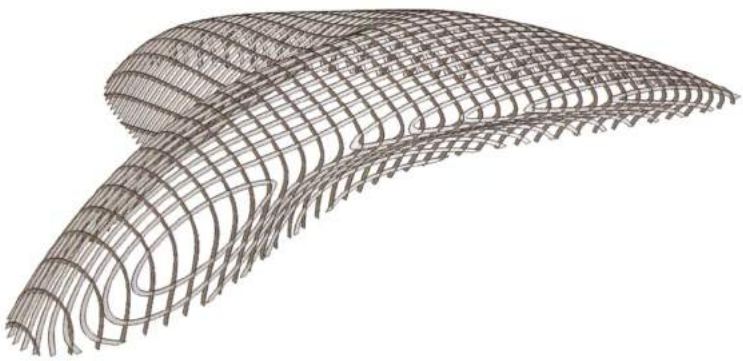
ABOVE. The image shows the output of *Join Curves*, i.e. a set of closed planar curves. BELOW. These curves can be easily oriented on the XY plane relying on the Orient component (4.2.3).



Once the curves are oriented, such that that all parts of every curve are coincident with the XY plane, they can be transferred to a cutting or milling machine. Most CNC machines cannot read NURBS curves. As a result the NURBS curves must be converted into **arcs and segments**. The conversion is based on two main strategies: the first strategy is to divide the curves into a large number of points and then define a polyline through the resulting points, the second strategy is to use Rhino's *Convert* command to convert curves into arcs or polylines with set tolerances. After conversion, the

resulting geometries are *nested* in order to minimize the material waste either manually, by a third party software, or by a plug-in.

Another technique to build freeform objects by planar sections is based on a bi-directional strategy usually referred to as ***waffling***. Waffling performs the section contouring procedure in two orthogonal directions. There are several Grasshopper plug-ins available to automate the process.



The following image shows different waffling alternatives, which are based on the initial shape and the direction of sectioning, to achieve a desired output.

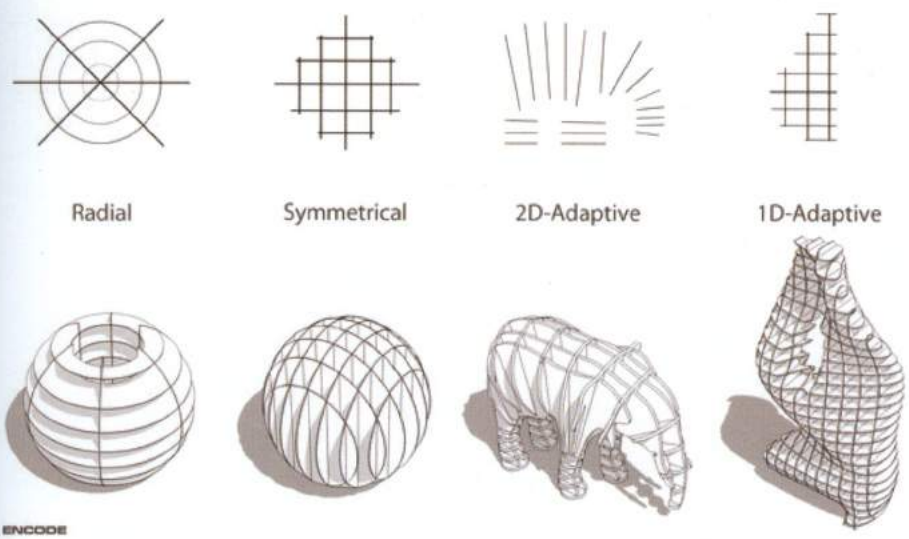
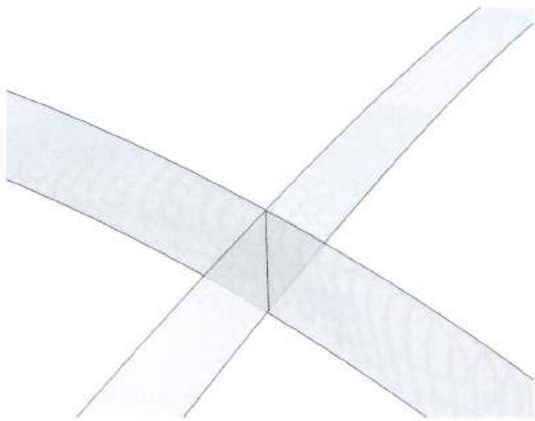


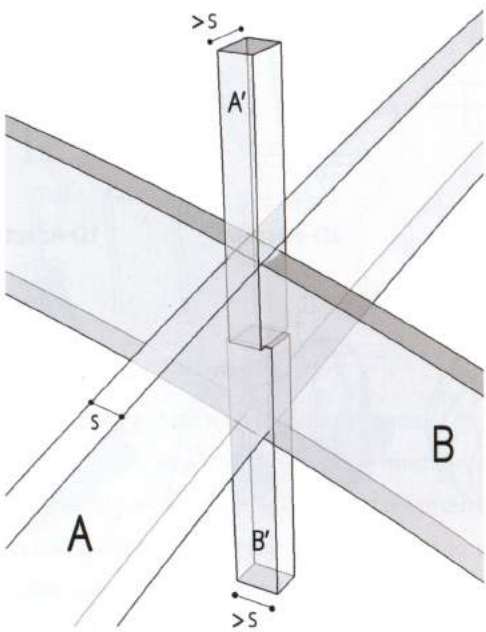
FIGURE 8.12
Different kinds of waffling. Image courtesy of Hassan Ragab.

Waffling requires intersections on which to perform a standard set of operations:

- Every possible intersection between every *rib* (or planar surface) is calculated and a list of intersection-segments are returned.

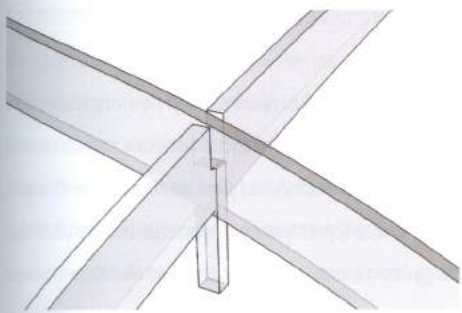


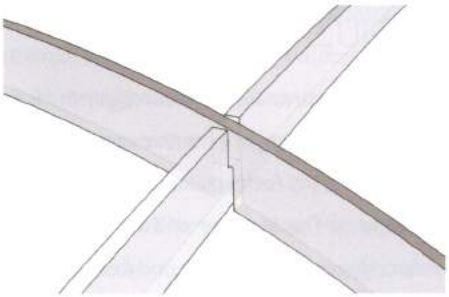
- At each intersection-segment two domain boxes (A' and B') are created. The boxes are subsequently subtracted using the *Solid Difference operation*: A-A' and B-B'.

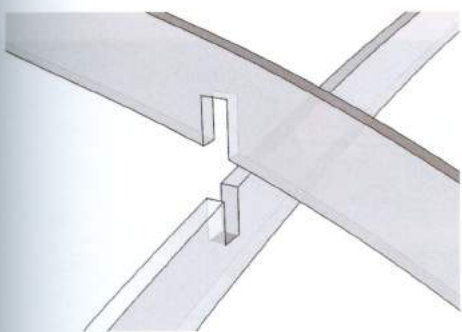


3

A complete waffling exercise can be found using the QR code above.







A solid difference allows to create node-intesections between ribs.

FIGURE 8.13

Parametric Soft Wall by Studio Kami, Rome (2012). Image courtesy of Studio Kami (www.studiokami.it).

8.4 NU:S Installation

"The collective approach is not utopian and is indeed possible. The NU:S project clearly demonstrates that this methodology is both creative and attainable, and is an example of one of the directions artists in Italy are taking. "NU:S Fashion clothes Architecture", is an art installation exhibited first at Bramante Cloister, and later at The Museum of Contemporary Art of Rome. The project came from a cross-fertilization within different artistic fields and fascination for technology, computational design, and the Renaissance philosophies.

FIGURE 8.11
The NU:S installation. Photo by G. Catani and L. Sorrentino.

The work had no preconceived result or over-arching direction, only interaction. It was a herculean experiment, where the goal was the study of the interplay between creative forces, not the result. This is fundamentally different from any other event because what was created was imagined and made within

and for the project. The NU:S installation, designed by Arturo Tedeschi and Maurizio Degni, was inspired by the ascessional flow of the cloister as well as its symbolic strength, these contaminating the mathematical intrinsic logic of Renaissance architecture, reinterpreting it through the new parametric paradigm. The sculpture was also inspired by tailoring expertise since the introduction of digital techniques and fabrication processes in architecture allowed a concept of personalization only similar to haute couture. [...] The interesting aspect about this project is that new media led to artistic innovation, the very same phenomenon that characterized the Renaissance. In the 15th century artists applied perspective to drawing, whereas today computational design is employed to support innovation.

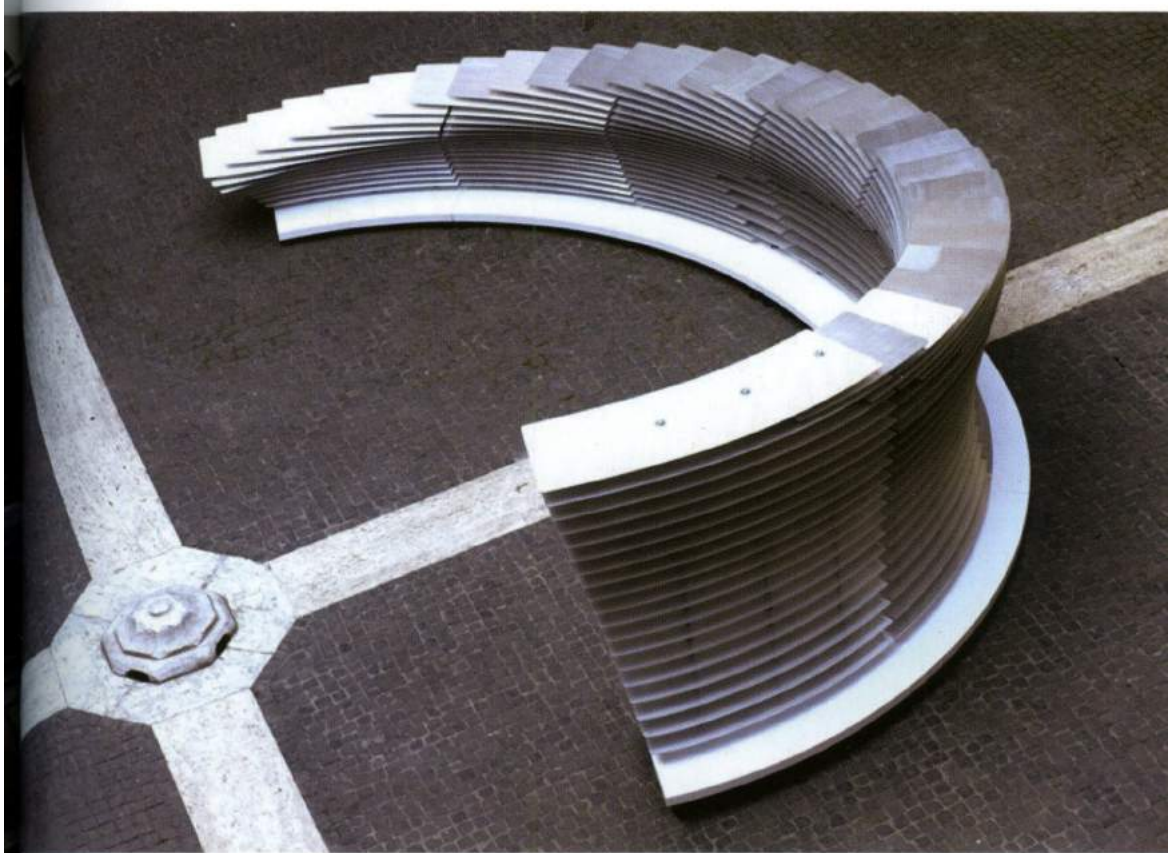


FIGURE 8.12
The NU:S installation. Photo by E. Lucci.

There are different levels of possibilities in the new digital instruments and technical planning methods, and NU:S tried to explore these. Sometimes there was a one directional process, sometimes bi-directional, sometimes even multi-directional. The revolutionary factor in the work was how each

designer maintained his own identity, even while adopting software and hardware foreign to their specific field".

Antonella Buono

Antonella Buono graduated as a Fashion Designer and Product Responsible. Afterward, she focused on costume design and during the last eleven years she has worked in the advertising and movie industry as a stylist and a costume designer. Beyond her work as a stylist, she teaches Fashion Design at UARC for the Philadelphia University as well as courses in Rome. She is interested in cross-fertilization within different artistic fields, which she examines as an independent researcher. www.glamnicism.com

FIGURE 8.13
NU:S parametric shoes (2012) designed by Maurizio Degni, Alessio Spinelli and Arturo Tedeschi (sinterized nylon powder). The particular aesthetic of parametric structures - predominantly found in architectural design - has been expressed in this project through the creation of a "wearable object". Photo by G. Catani and L. Sorrentino.

8.5 Large-scale objects

Large-scale fabrications, or objects that in their articulation require technical, structural and functional issues to be considered, such as pavilions, canopies, shells, houses, and buildings; require additional techniques. The fabrication of large-scale objects usually involves the assembly of individually fabricated parts to transfer loads. Two important trajectories for fabricating large scale objects are currently being researched in academia and in the architecture profession. The first approach aims to reduce complexity and differentiation of components while the second aims to simplify the building process.

1. Computational approach.

   The main challenge of building a freeform facade has traditionally been to approximate a surface using **planar** panels of the same dimensions. This strategy is used to reduce manufacturing costs and optimize the reuse of molds. This process of surface discretization was often achieved through the triangulation of a surface. This technique has several disadvantages including: the weight of the support-structure, the complexity of each structural node etc. Nowadays cutting-edge computational techniques have been developed in order to achieve **planar quad panels** (PQ Meshes) from an arbitrary freeform surface and, under certain conditions, to get **planar hexagonal panels** (P-Hex Meshes). P-Hex Meshes simplifies the fabrication of the underlying structure as each node of an hexagon connect just three rods.

FIGURE 8.14

A freeform surface approximated through a set of planar hexagonal panels.

Developable strategies also play an important role in the construction of freeform objects. Several algorithms have been developed in order to find developable surfaces or sets of surfaces to approximate a freeform geometry. Since every developable surface is a ruled surface, a set of straight lines can always be found. These lines can become the axis of linear beams, ultimately simplifying the support structure.

2. Innovative fabrication.

   Whereas *computational approaches* operate within the traditional fabrication environment, *innovative fabrications* focus on developing new fabrication strategies. For instance, pioneer companies like D-Shape, guided by the italian engineer and inventor Enrico Dini, are successfully testing additive techniques to print large-scale objects. The D-Shape procedure compels designers to rethink the conception and optimization of structures.

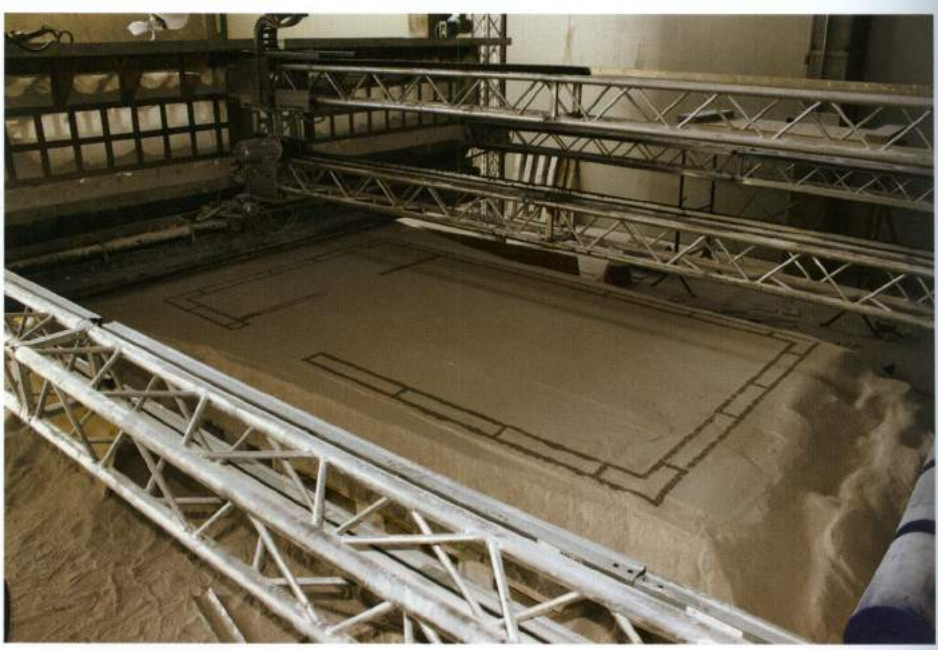


FIGURE 8.15
The 6m x 6m D-Shape printer, built by the italian engineer Enrico Dini. This is the first large-scale 3D printer. Image courtesy of D-Shape (www.d-shape.com).

Over the material, Past the Digital: Back to Cities

Stefano Andreani
Lecturer in Architectural Technology & Research Associate
Harvard University - Graduate School of Design

The history of architecture features a combination of different technological timelines. As Mario Carpo points out in his *The Alphabet and the Algorithm* book,[1] we can identify three technical ages – the age of hand-making, the age of mechanical-making, and the age of digital-making.

This distinction is of particular relevance if we think of the intrinsic dichotomy of architectural design: on the one hand, the built environment is based on the production of material objects, which in turn depend on the evolution of manufacturing technologies; whereas on the other hand, architectural design translates into abstract operations that are very much influenced by cultural environments and medium of representation. The world of hand-making that preceded the machine-made environment was characterized by unique pieces, mainly conceived and produced through a process of imitation rather than replication. The first break, the transition – as Carpo defines it – "from *artisanal variability* to *mechanical identicality*", occurred with the industrial revolution. But the second break in this sequence, the passage from mechanically-made identical copies to digitally-generated differential variations, is happening now. Hand-making creates variations, as does digital making; but the capacity to design and mass-produce serial variations (or *differentiality*) is specific to the present digital environment. Unlimited variability, however, may result in a loss of relevance and ultimately in a loss of meaning.

Back to the nineties, the digital revolution began to narrow down the gap between design and fabrication. Computers, in fact, not only could easily deliver tools for the ideation and manipulation of complex mathematical forms; these new tools could also be directly applied to the conception, representation, and production of objects.

Back then Bernard Cache stated that "*mathematics has effectively become an object of manufacture*,"[2] and Greg Lynn remarked that computer-aided design had "*allowed architects to explore calculus-*

1. Mario Carpo, *The Alphabet and the Algorithm* (Cambridge, MA: MIT Press, 2011).
2. Bernard Cache, "Objectile: The Pursuit of Philosophy by Other Means," in Stephen Perrella, ed., *Hypersurface Architecture II*, special issue (AD Profile 141), *Architectural Design* 69, nos. 9–10, 1999.

based forms for the first time."[3] The concept of *multiple variations* then emerged to define this new idea of the digitally-made object, what Deleuze would refer to as *objectile* – a function that contains an infinite number of objects[4].

> *The objectile is not an object but an algorithm – a parametric function which may determine an infinite variety of objects, all different (one for each set of parameters) yet all similar (as the underlying function is the same for all). Instead of focusing on one instance from a virtual series of many, the new technological paradigm is increasingly dealing with variations that can all be designed and fabricated sequentially: mathematical continuity in this case is set in a manufacturing series, not in a diachronic sequence, and used to mass produce the infinite variants of the same objectile – at the same unit cost as identical copies.*[5]

This technological shift defines the basic principles of a *nonstandard series* – i.e., a set in which each item has some features in common with all others. It thus follows the notion of *mass-customization*, which was born as a marketing strategy well before the rise of CAD-CAM technologies.[6] The term first appeared in 1987, with Stanley M. Davis' *Future Perfect* book, supplying both a name and a conceptual framework for processes initially from the clothing industry and recognizing that mass customization simply extended the capabilities latent in CAD/CAM processes.[7] In 1993, B. Joseph Pine II expanded on Davis' ideas articulating the production systems into three main categories: craft production, mass production, and mass customization – which combined elements of the first two.[8]

However, in the realm of architecture, the increasing power of today's computational tools might lead to discrepancies, rather than proximities, between the digital and physical worlds. As John Frazer argues when referring to the evolution of computational design, *"We went to all this effort in order to solve real social, environmental and technical problems where we believed a computer could significantly assist. But now that there is a massive computer power and software cheaply available, most scripting has become nothing more than an onanistic self indulgence in a cozy graphic environment. Endless repetition and variation on elaborate geometrical schema with no apparent social, environmental*

3. Greg Lynn, *Animate Form* (New York, NY: Princeton Architectural Press, 1999).
4. Gilles Deleuze, *Le pli: Leibniz et le baroque* (Paris: Editions de minuit, 1988).
5. Mario Carpo, *The Alphabet and the Algorithm*.
6. Stanley M. Davis, *Future Perfect* (Reading, MA: Addison-Wesley, 1987).
7. Dan Willis Todd and Woodward, "Diminishing Difficulty: Mass Customization and the Digital Production of Architecture," in *Fabricating architecture: Selected readings in digital design and manufacturing* (New York: Princeton Architectural Press, 2010).
8. B. Joseph Pine II, *Mass Customization: The New Frontier in Business Competition* (Boston, MA: Harvard Business School Press, 1993).

and technical purpose whatsoever."[9] A true paradigmatic shift in architectural design and fabrication may lie instead on the role of, and push for, *innovation* – in the sense of "*an activity that generates vitality*" as proposed by Mark Burry.[10] A conceptual approach to innovation that highlights the role of the environment in affecting an idea, supporting its development, and eventually its implementation and dissemination.

Innovation is what actually drives the research activities pursued by the Design Robotics Group (DRG) at the Graduate School of Design of Harvard University. The research unit, led by Professor Martin Bechthold, promotes the understanding, development and deployment of innovative technologies in the use of design as an agent of change in the quest for a better future. DRG looks at the role of material processes and systems in the built environment, with a special interest in robotic and computer-numerically controlled (CNC) fabrication processes. Combining issues of design computation, materials and assembly processes, the projects result in speculative prototypes as well as applied research geared towards industry integration. DRG work in material systems is best characterized as process-oriented, strategic and focused on performance.

An example of the implementation of the design-oriented research pursued by the Harvard DRG is the 'Ceramics 2.0' Workshop Cluster at the 2012 SmartGeometry, where ceramic material systems were explored through a combination of computational design methods and a six-axis robotic manipulator equipped with a wire-cutting tool. Run by Prof. Martin Bechthold, Jose Luis Garcia del Castillo, Aurgho Jyoti, Nathan King and the author, the workshop built upon the 'Flowing Matter' project developed at Harvard GSD.[11] Going back and forth between manual and robotic clay manipulation, workshop participants developed design intuitions that expanded beyond what would have been feasible when limited to either physical or computational methods. Exploring a material system in this open-ended manner generated a host of powerful ideas and much discussion. The workshop demonstrated that design robotics has matured to a point that brainstorming and sketching are now possible in newly hybridized modes that combine robotics with exploratory hands-on experiments.[12]

This line of research on ceramic building systems eventually resulted in the articulation of the

9. John Frazer, in M. Burry (ed.), *Scripting Cultures: Architectural design and programming* (Chichester: John Wiley&Sons, 2011).

10. Mark Burry, "The Innovation Imperative: Architectures of Vitality," *AD The Innovation Imperative: Architectures of Vitality* 221 (2013): 8-17.

11. Stefano Andreani, et al., "Flowing Matter: Robotic fabrication of complex ceramic systems," in *Proceedings of ISARC 2012: The 29th International Symposium on Automation and Robotics in Construction* (Eindhoven, 2012).

12. Martin Bechthold, "Design Robotics: New Strategies for Material System Research," in Brady Peters et al., eds. *Inside Smartgeometry: Expanding the Architectural Possibilities of Computational Design* (London: John Wiley & Sons, 2013): 254–265.

'[R]evolving Brick' project.[13] Developed at Harvard GSD by Prof. Martin Bechthold and the author, the on-going study introduces the concept of *strategic customization* in the industrial fabrication context.

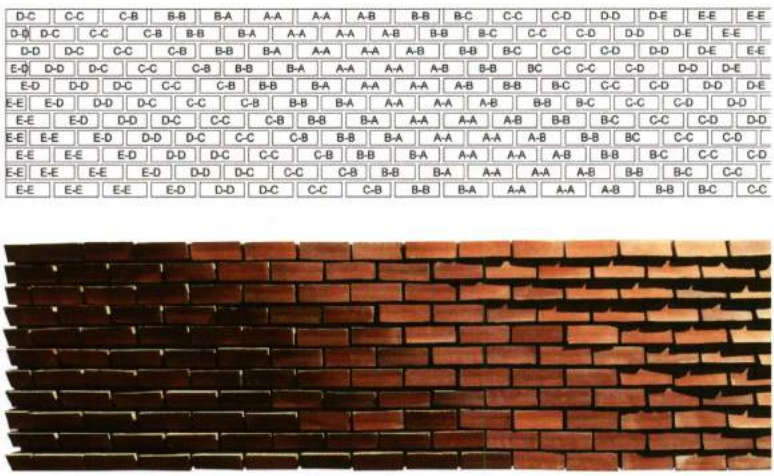


FIGURE 1
© Stefano Andreani.

The project in fact integrates robotic technology on the production side, so that the age-old rectangular form of the brick can be successfully overcome, while maintaining the efficiency of tried and true mass-production methods. The resulting mass-customization of brick forms opens up a new design space in brick construction. In order to combine ornamental effects with sustainable design in architectural ceramic systems, this work also developed strategies to improve the energy efficiency of brick envelopes. In particular, by combining material proprieties and geometric parameters, the research shows that it is possible to optimize the material configuration to generate solar-selective thermal mass systems that include self-shading. Exploiting the advantages of the geometric complexity available through the proposed shaping process, the new material system merges aesthetics and environmental performance by creating design pattern articulations that respond to variable climatic and diurnal cycles. The resulting integrated workflow would eventually let both architects and manufacturers re-think the way brick building systems can be used, and let designers re-create novel and unexpected relationships with this traditional material.

13. Stefano Andreani and Martin Bechthold, "[R]evolving Brick: Geometry and Performance Innovation in Ceramic Building Systems through Design Robotics," in Fabio Gramazio et al., eds. *Fabricate: Negotiating Design and Making* (Zurich: Gta Publishers, 2014).

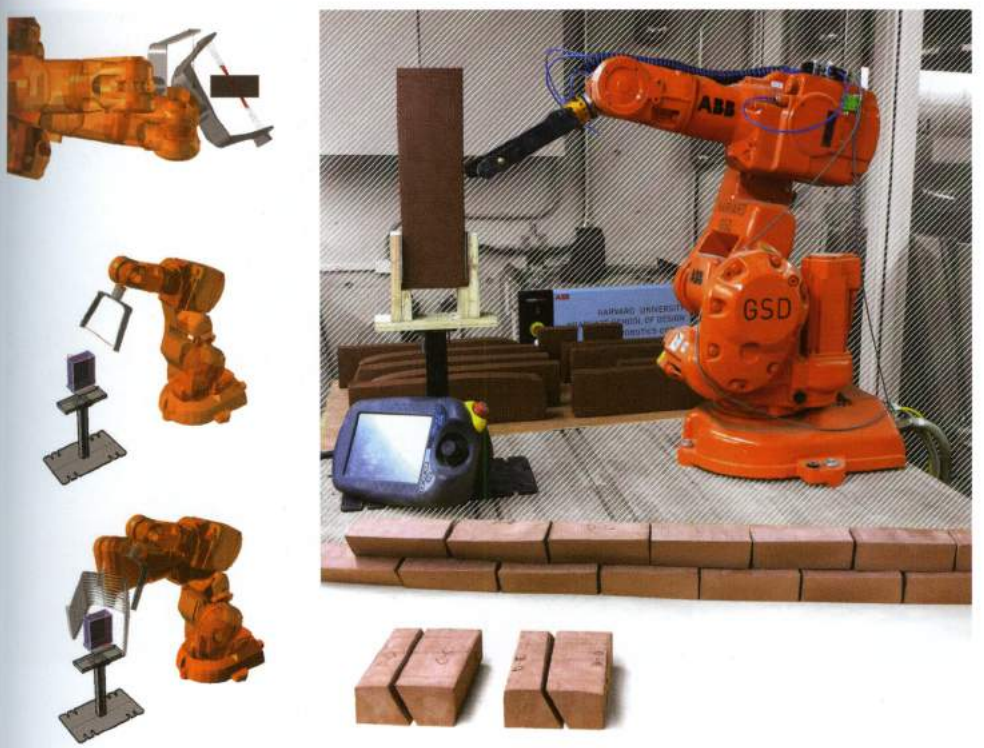


FIGURE 2
© Stefano Andreani.

Concepts such as complex geometry optimization, material structural efficiency, and advanced fabrication methods find unique expression in the design and fabrication of the 'Floating Ceramic Shell' – a structural ceramic and concrete shell system being developed as a collaboration between the Harvard Graduate School of Design and the Institute for Structural Design at TU Graz, Austria. Sponsored by the Valencia Trade Fair Association and by ASCER, the mock-up of the ceramic shell system was exhibited at the 2014 Cevisama in Valencia, Spain as the centerpiece of this year's international show. Floating overhead like a giant pair of wings, the installation consisted of a ceramic deck measuring about 7.8 by 4.5 meters in a plan view, suspended from the centre columns of the fair building. This shell has a double-curvature surface, which is composed of just a single type of ceramic piece – whereas double-curvature surfaces usually need several types of pieces. In order to achieve this simplification, the ceramic units overlap each other, thus accommodating the differences in measurements of the doubly-curved surface.[14]

14. "Trans/Hitos 2014," *Domus*, February 20th, 2014, http://www.domusweb.it/en/news/2014/02/20/trans_hitos_2014.html

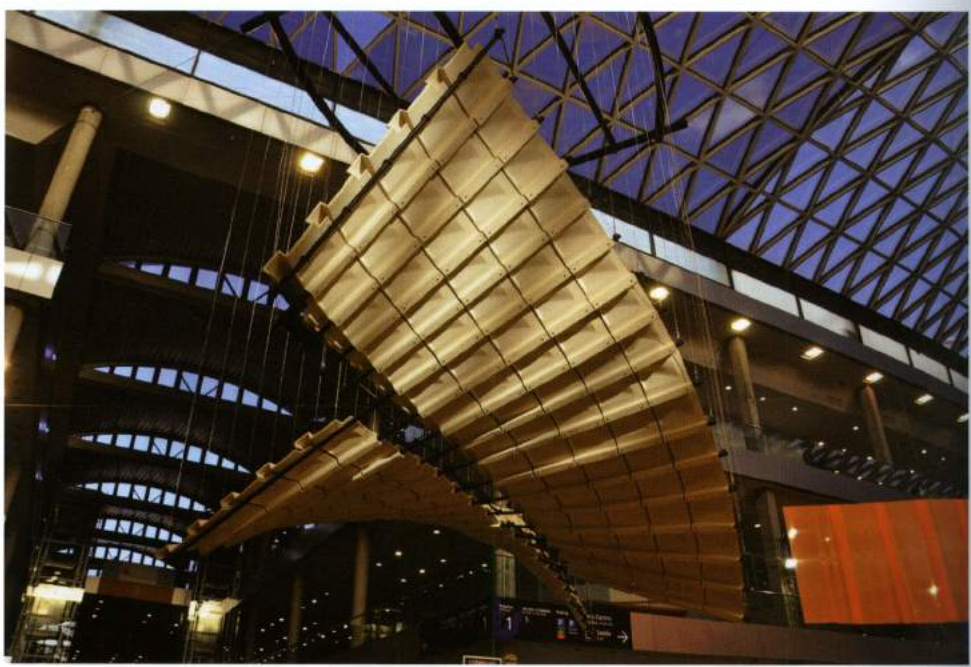


FIGURE 3
© Andreas Trummer.

The large ceramic stoneware elements are designed to enclose channels that form a perpendicular network of connecting ultra-high strength fiber concrete ribs. Ribs and tiles form a composite structural surface. Structural tests will be eventually performed on various-scale prototypes before the actual erection of the ceramic/concrete shell.

Material system research pursued at Harvard GSD thus embraces a variety of materials and techniques. A further investigation is the 'Robotic Casting' workshop at the 2012 Robotics in Architecture in Vienna, run by Prof. Martin Bechthold, Nathan King and the author. The workshop explored novel approaches to serially customized casting processes enabled through the strategic deployment of 6-axis robotic manipulators. Traditional casting techniques currently limit designers to repetitive use of identical elements in order to distribute the cost of the molds. Serialized customization, mass-customization or related techniques that produce highly variable, individualized design expressions are rarely possible, yet increasingly demanded in pursuit of contemporary architectural forms. Robotic casting is a new casting method developed by the Design Robotics Group that uses strategically designed molds that are oriented robotically and, when filled with variable material volumes, produce families of varied yet similar shapes. Studies performed during the workshop proposed modular assemblies

driven by acoustic, lighting, views, or assembly techniques of interlocking and staggering. Robotic code was created with a DRG Grasshopper component that outputs angle rotations and volume measurements for each piece. Following the digital design process, the workshop team produced, in 14 hours, a doubly-curved, perforated wall consisting of 40 individual elements.

FIGURE 4
© Stefano Andreani.

The projects described above, particularly the ones developed in workshop contexts as collective experience, can be seen as manifestations of a fundamental objective at Harvard GSD: design and manufacturing can really permeate each other, blurring the boundaries between the digital and physical worlds. The widespread movement of fab-labs, the increasing use of 3d printing techniques, and the creation of ecosystems or hubs of innovation in cities reinforce the idea that the employment of digital-fabrication technologies is rapidly becoming ubiquitous. Digital design, the use of big data, and nanotechnology can play a valuable role in creating new manufacturing methods, processes and systems, and not just products. Looking at economies as *product space*, Harvard economist Ricardo Hausmann argues that "*developments around information technology, 3-D printing, and networks will*

allow for a redesign of manufacturing. The world will be massively investing in it."[15]

FIGURE 5
© Stefano Andreani.

In this context, the Responsive Environments and Artifacts Lab (REAL) at the Harvard Graduate School of Design, led by Professor Allen Sayegh, looks at how new technologies can impact the built environments in which we live, work, and play. Pursuing the design of digital, virtual and physical worlds as an indivisible whole, REAL investigates the all-pervasive nature of digital information and interaction at scales ranging from our bodies to the larger urban contexts we occupy and the infrastructures that support them. Through multi-disciplinary projects in the U.S., Europe and China, REAL is developing novel theoretical frameworks for tackling pressing problems of contemporary cities, from the redefinition of local manufacturing activities for bringing factories back to cities, through the development of new digital and physical tools for improving health and wellbeing of urban environments, to the deployment of advanced technologies in high-rise, high-density cities

15. Antonio Regalado, "You Must Make the New Machines," in The Next Wave of Manufacturing, *MIT Technology Review*, 2013.

that mediate that scale of the body with the scale of the tall building.

The future dynamics of contemporary cities will likely encompass a seamless integration of smart objects, interactive environments, and digitally-augmented fabrication machines with the everyday life of citizens. As designers, it is then our responsibility to develop ecological models of implementation for ensuring the creation of high-quality environments in our cities.

Stefano Andreani

Harvard University, Boston, 03.25.2014

Stefano Andreani is a licensed architectural engineer and educator interested in the strategic implementation of advanced technologies in architecture for innovative design solutions. As Research Associate at the Graduate School of Design of Harvard University, he pursues research on performative material systems within the Design Robotics Group (DRG) and on the future of learning and healthcare practices and environments within the Responsive Environments and Artifacts Lab (REAL). Andreani received a Master in Design Technology degree from Harvard GSD and a Master in Architectural Engineering from the University of Perugia, where he was Assistant Professor of Architectural Technology. His professional work mainly focuses on high-rise design, as Project Designer at RBA Studio and as Design Technology Consultant for the South China University of Technology.

(Digital) Form-finding

Alberto Pugnale
Lecturer in Architectural design at the University of Melbourne

In architecture and structural engineering, 'form-finding' identifies the process of designing optimal structural shapes by using experimental tools and strategies, i.e. physical models[1], to simulate a specific mechanical behavior. The reverse hanging method is the oldest and probably most diffused form-finding technique for arches, vaults and shells – a physical model, made with elastic cables or membranes with no rotational stiffness, is first subject to gravitational forces to obtain a structural state of pure tension; such a form, which is called "funicular", is then inverted to identify the mechanical compression-only situation. This principle was first mentioned in a publication by Robert Hooke in 1675[2]. With a Latin anagram, only solved in 1701, he proposed to reverse the curve which was generated by a hanging chain, under self-weight and supported only at its ends, in order to define the optimal structural form of an arch (figure 1). Such a curve is called "catenary" when the chain presents a constant distribution of weight[3]. Initially confused with a parabola by Galileo Galilei, the catenary was then mathematically described by David Gregory in 1697[4].

1. A detailed description on the use of small-scale physical models for structural design can be found in: Addis B., *'Toys that save millions' – A history of using physical models in structural design*, in "The Structural Engineer", April 2013, pp.12-27.
2. See: Hooke R., "A description of helioscopes and some other instruments made by Robert Hooke, Fellow of the Royal Society", London: T.R. for John Martyn, 1676, p.31.
3. See: Adriaenssens S., Block P., Veenendaal D., Williams C. (Eds.), "Shell Structures for Architecture", Routledge, 2014, pp. 7-8.
4. See: Kurrer K., "The history of the theory of structures: from arch analysis to computational mechanics", Berlin: Ernst &Sohn, 2008, pp. 213-216.

FIGURE 1
A catenary arch and its inverted shape.

Infinite ideal forms of a compression-only arch can be generated by varying two boundary conditions: (1) the applied loading; and (2) the span/rise ratio. Figure 2 shows how these parameters affect the final geometry individually.

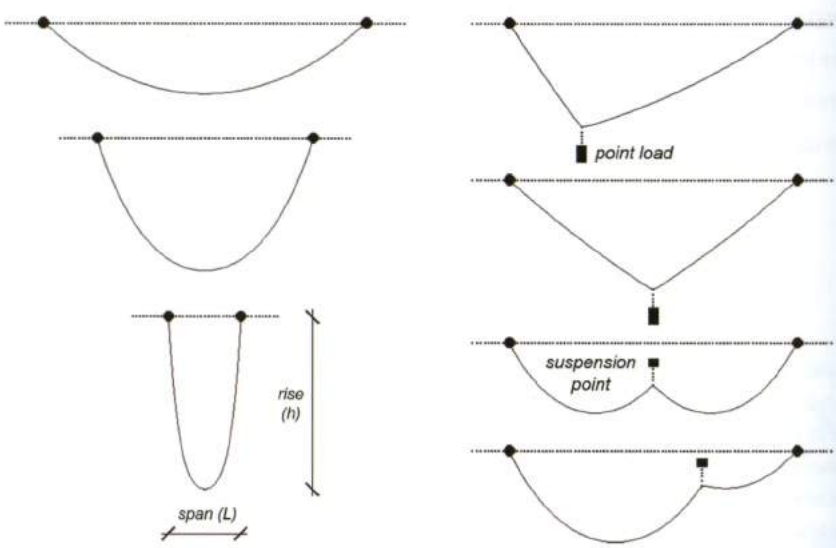


FIGURE 2
Variations of the boundary conditions and respective hanging chains.

The same principle can be extended to the three-dimensions in order to find structural form of Reinforced Concrete or masonry vaults and shells, as well as of steel or timber gridshells. Roughly, three model constructing methods can be distinguished. The first is based on the use of strings, with the function of discretizing either barrel vaults, if placed in parallel, or domes, when disposed in a radial way. Bags of sand are then added to control the distribution of weight. Antoni Gaudí designed several buildings with the aid of this procedure. Rather known is the model he realized for the Church of Colonia Güell, in which two hierarchical orders of strings were used – the first defined the geometry of columns, main arches and supported the second, which established, indeed, the form of walls and vaults. In 1982/83, the Institute for Lightweight Structures in Stuttgart performed a reconstruction of the original Gaudí's model, demonstrating that its preparation is highly time-consuming and precision becomes a key aspect of the simulation in order to get accurate results[5]. The second model making technique takes advantage of non-rigid nets, realized either with chains or elements, in order to find structural form for gridshells[6]. The Multihalle in Mannheim (1973-74) is the biggest and probably most relevant building which was designed with this method. A 1:300 wire-mesh model was initially built by Frei Otto and his research group in Stuttgart to establish the basic geometry – two main halls connected by a tunnel, of which the larger spans about 60 meters. A 1:98.9 scale model was then prepared to refine the shape of the gridshell and determine the precise position of the boundary supports (Figure 3). A rather sophisticated design phase that followed was the survey of the model through photogrammetry – geometrical data were then transferred into constructing drawings and other analytical models for further calculations[7].

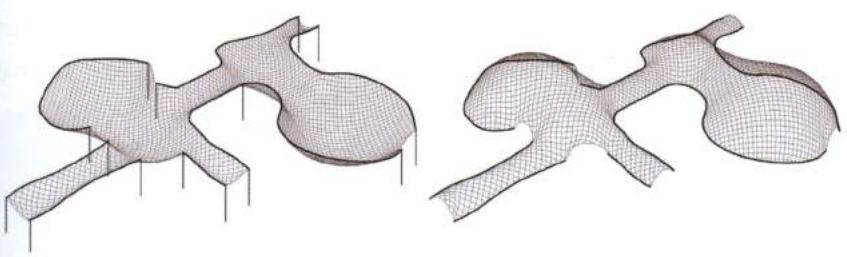


FIGURE 3
Schemes of the second form-finding model of the Multihalle in Mannheim, which represents every third lath of the real structure.

5. See: Tomlow J. (Ed.), "IL 34. The model: Antoni Gaudí's hanging model and its reconstruction – New light on the design of the church of Colonia Güell", Stuttgart: Institute for Lightweight Structures (IL), 1989.
6. See: Hennicke J. et al., "Il 10. Grid Shells", Stuttgart: Institute for Lightweight Structures (IL), 1974.
7. See: Burkhardt B. et al., "IL 13. Multihalle Mannheim", Stuttgart: Institute for Lightweight Structures (IL), 1978, pp. 33-55.

The third method of making hanging models was developed by Heinz Isler in 1955, and it was specific for the form-finding of RC shells which are a continuous type of structures. Such a characteristic was simulated through suspension of wet pieces of fabric or membrane - the derived forms were then frozen and finally inverted[8]. Very different results can be obtained by varying the type of fabric used, i.e. the properties of the model material play an important role in the process.
Amongst the shells Isler designed throughout his career, the ones of the Deitingen service station, built in 1968 on the Bern-Zurich motorway, and the roof of the open-air theatre in Grötzingen, dated 1977 and with a thickness of only 11cm, are the most representative of the structural lightness this form-finding technique can lead to.

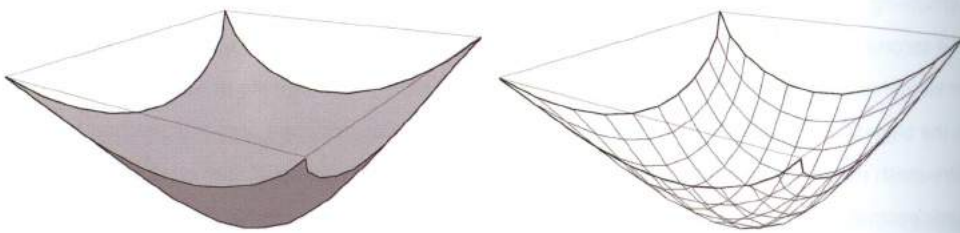


FIGURE 4
Simulation of a hanging membrane model and its corresponding cable net one.

Apart from the reverse hanging method, a few other experimental ways of finding structural form through physical models have been developed. The first is based on stretching cables or elastic membranes across different edge frames – this simulates the pre-stress state typical of cable nets or tensile structures and generates, consequently, their geometry. The shape of small scale tensile structures can also be found dipping closed frames, made for instance of thin wire, into a membrane-forming liquid[9]. This procedure forms soap films, which are mathematically defined as "minimal surfaces"[10]. Frei Otto has been a pioneer in this field. Together with his research group of the Institute for Lightweight Structures in Stuttgart, he defined an extensive set of rules for the generation of soap films. They classified such minimal surfaces according to the number of closed frames they needed to generate them. For instance, saddle shapes can be obtained from a single

8. See: Isler H., *New Shapes for Shells – Twenty Years After*, in "Bulletin of the International Association for Shell Structures", no.71/72, 1980, pp. 9-26. See also: Isler H., *New Shapes for Shells*, in "Bullettin of the IASS, no.8, 1961; and: Isler H., *Concrete Shells Derived from Experimental Shapes*, in "Structural Engineering International, Vol.3, 1994, pp. 142-147.
9. See: Otto F., Rasch B., "Finding Form: Towards an Architecture of the Minimal", Axel Menges, 1996, pp. 58-59.
10. Several examples of minimal surfaces can be found in: Gray A., Abbena E., Salamon S., "Modern Differential Geometry of Curves and Surfaces with Mathematica", 3rd ed., Boca Raton: CRC, 2006 (1993).

closed edge. Two ring frames are necessary to form a catenoid, which is the surface of revolution of the catenary curve, and infinite other different soap films can then be found if further two-dimensional inner edges are added[11].

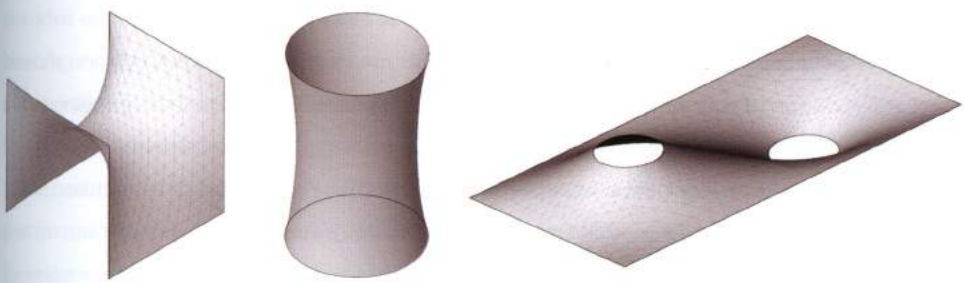


FIGURE 5
Generation of soap films from one (left), two (middle) and several closed frames (right).

Two projects by Frei Otto are worth to be mentioned: the Tanzbrunnen tensile structure in Cologne, dated 1957, and the large roofs built in 1972 for the Olympic Games in Munich. Minimal surfaces have also been used for the form-finding of compression-only structures. A rather interesting example is provided by the RC bridge designed by Sergio Musmeci over the Basento River in Italy (1967-69)[12].

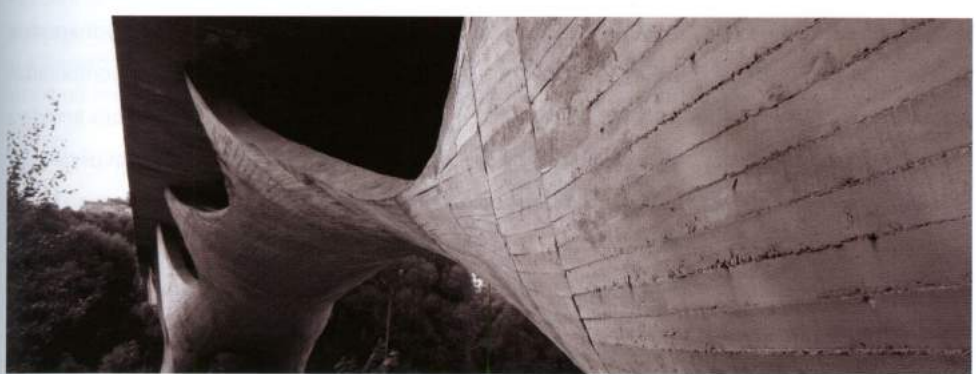


FIGURE 6
Bridge over the Basento River, Potenza (Italy).

11. See: Bach K., "IL 18. Forming bubbles", Stuttgart: Institute for Lightweight Structures (IL), 1988, pp.73-219.
12. No English sources have been published on the work by Musmeci. However, the Italian books are: Nicoletti M., *Sergio Musmeci: Organicità di forme e forze nello spazio*, Torino: Testo & Immagine, 1999, and Guccione M., *Il ponte e la città. Sergio Musmeci a Potenza*, Roma: Gangemi, 2003.

A second alternative to the use of hanging models is called "pneumatic or inflated hill method". The concept is rather intuitive: structural form is obtained through inflation of membranes and can be applied for the design of RC shells, as well as air-supported membrane halls. During the 60ies, Italian architect Dante Bini developed and patented a curious construction technique called "Binishell", which took advantage of this form-finding process in order to erect RC domes in a rapid way - concrete pouring was initially performed over a flat pneumatic preformed formwork, and then inflation allowed rising and roof completion within a couple of weeks[13]. Other alternative form-finding methods are based on flowing forms or on combinations the previous techniques. An extensive description of such experimental strategies can be found in the book: "Finding Form: Towards an Architecture of the Minimal" by Frei Otto and Bodo Rasch[14]. Analytical methods have been developed too. In this case, structural form is defined using analytically well-known geometries, such as cylinders, spheres, ellipsoids, or forms obtained through operations on them. Félix Candela designed several examples of this kind. The Church of Our Lady of the Miraculous Medal, completed in 1955 in Mexico City, and Los Manantiales Restaurant in Xochimilco, dated 1958, are two outstanding buildings resulting from geometrical operations on hyperbolic paraboloids. Through form-finding, design is always directed towards structural optimum. From the conceptual point of view, this cannot result in free-forms, which are 'freely' generated apart from any structural and construction principle. In other words, the representative component of architecture cannot be separated from its conformative core. Digital technologies are radically modifying this aspect. Numerical calculation techniques are replacing entirely experimental structural design and analysis methods - the way now is to use mathematical optimization which, on the basis of one or more chosen criteria, takes advantage of the computation power of the computer to interactively search for optimal solutions to a problem from among a series of possible candidates. This change is relevant, from the architectural design point of view, for at least three reasons. Unlike in classical form-finding, the topology of a structural system no longer needs to be fixed. It can therefore become the object itself of the optimization process, as in the case of the design of the new TAV station in Florence, which was developed by Isozaki and Sasaki on occasion of an international competition in 2003[15].

Compared to the projects by Heinz Isler and Frei Otto, optimization also allows the original form-finding concept, literally aimed at the search of the optimal form, to be changed into what can be

13. Bini D., "Building with air", London: Bibliotheque McLean, 2014.

14. See: Otto F., Rasch B., "Finding Form: Towards an Architecture of the Minimal", Axel Menges, 1996.

15. See: Cui C., Ohmori H., Sasaki M., Computational Morphogenesis of 3D Structures by Extended ESO Method, in "Journal of the International Association for Shell and Spatial Structures, Vol. 44, no. 141, 2003, pp. 51-61. The competition project for the new TAV station in Florence is also described in: Sasaki M., Flux Structure, TOTO, 2005.

defined as "form-improvement"[16] - this new process is instead aimed at improving the performances of an already existing spatial configuration, which does not necessarily mean reaching the structural optimum. For example, as far as the Kagamigahara crematorium is concerned, no physical model used to obtain the inverse of the tension-only hanging membrane would have been able to translate the idea of the architect Toyo Ito into a structure. Instead, through optimization, the floating RC roof, figuratively inspired by a cloud, was first freely modeled as if it were a sculpture and was then structurally honed through a Sensitivity Analysis (SA)[17].

The last fundamental aspect of optimization is that it is not just limited to resolving questions of a static nature, which instead is an intrinsic characteristic of form-finding based on physical models. Techniques like GAs can be used in all those cases in which an architectural performance can be formulated through a mathematical function, such as in the case of acoustics or light, and, technically speaking, it can therefore be "minimized".

From being a simple resolution instrument, optimization is becoming an efficient "form-exploration" tool to support conceptual design. It is forcing the limits of classical form-finding and defining several new research directions that redefine entirely the relationship between architecture and engineering.

Alberto Pugnale is a lecturer in Architectural design at the University of Melbourne, Australia. In 2007, he won the fifth edition of the IASS HANGAI Prize, related to the study of complex architectural/structural bodies. He has been an assistant professor at Aalborg University, Denmark (2010–2012), and an invited lecturer in France and Italy. At present, he is member of the International Association for Shell and Spatial Structures (IASS) and is a licensed architect in Europe. His research interests are in the computational morphogenesis of free-form structures, reciprocal structures and history of construction.

http://albertopugnale.wordpress.com/

16. The term "form-improvement" was coined by the author in March 2007 for a short online article.
17. The Sensitivity Analysis is explained briefly in: Sasaki M., Flux Structures, TOTO, 2005.

Only-compression vault obtained through digital form-finding techniques. The scale model is made of 3D printed "bricks" that work in pure compression. The project was developed by Alessia De Luca under the supervision of the author.

9_digital simulation

particle-spring systems

"Science is a tool for ideas [...] and it is not only a means to verify structural strength. Science must lead us to discover the optimized geometry for that particular static (or dynamic) condition".

Sergio Musmeci

Structures that transmit forces through axial compression or tension have an increased capacity to withstand loads with smaller cross sectional areas. Traditional form-finding strategies for axially loaded structures include: complex physical models, hanging chain networks, stretched fabrics, soap films etc.. These techniques are difficult and time consuming. As a result, few designers investigated these form-finding potentials. Traditional form-finding techniques can now be digitally found using *particle-spring systems* that simulated the physical behavior of deformable bodies. Originally developed for character animation and cloth simulation, *particle-spring systems* have emerged as a powerful technique for form-finding. Whereas traditional techniques were difficult to apply, digital simulations allow designers to investigate form, in real time, by updating forces, supports and physical properties.

A particle-spring system is a **discretization** of a continuous model into a finite number of masses, called particles, connected by perfectly elastic springs. The main components of a *particle-spring system* are:

- **Particles**: each particle in the system is a lumped mass, that changes position and velocity as the simulation evolves.
- **Springs:** a spring is an elastic linear connection between two particles that behaves according to the Hooke's law: *a spring has an initial resting length and a stiffness value (k).*
- **Forces:** weights and external loads are simulated by vectors that are applied exclusively to particles.
- **Anchor Points:** particles that do not change position during the simulation.

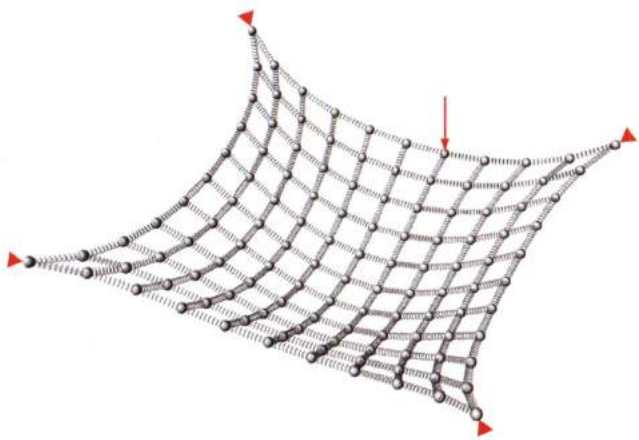


FIGURE 9.1
A particle-spring system that simulates a square membrane anchored at its corners. Force vectors are applied to the particles.

Once the simulation has started, the particles move from their initial position until they reach an equilibrium state which is dependent on the initial geometry, the force vectors, and the springs' defined properties. In accordance with Hooke's law, the lower the *stiffness* or *k value* the greater the spring elongation. Since particles in this system behave like spherical hinges without the capacity to resist moment forces, equilibrium solutions carry defined loads exclusively through **axial forces**. This is the ideal condition for form-finding strategies.

9.1 Kangaroo plug-in

Particle-spring systems (PSS) iterative calculations approach an equilibrium state were the sum of all forces is zero. The iterative calculations are performed by mathematical **solvers**.

Solvers operate iteratively within a main *engine,* meaning every subsequent iteration narrows the position and velocity of particles from the previous step towards an equilibrium solution. This process, similar to a key-frame animation, creates the illusion of movement when frames are calculated in a continuous sequence. Several particle-spring software packages have been developed recently including: CADenary[25] written by Axel Kilian, Dan Chak and Megan Galbraith in 2002. Most particle-spring software packages are standalone and do not fully integrated into a CAD or other modeling software. Furthermore, many of these standalone products are difficult to master. In contrast, **Kangaroo** a physics based particle-spring system engine developed by Daniel Piker[26] (development team: Robert Cervellione, Giulio Piacentino, Daniel Piker), is easy-to-use, designer-oriented, and is integrated as a plug-in for Grasshopper.

FIGURE 9.2
The Kangaroo toolbar.

Kangaroo enables designers to interact with form through *particle-spring system* simulations in real time. There are two interaction methods:

- **Direct Interaction**: includes the manipulation of anchor points, forces and spring properties;
- **Parametric or Associative Interaction**: anchor points, forces and spring properties can be parametrically linked to other parts of the 3D model. For example, the anchor points can be defined as the end points of a set of lines whose position is defined by another part of the algorithm.

NOTE 25
http://designexplorer.net/newscreens/cadenarytool/cadenarytool.html
NOTA 26
Daniel Piker is a researcher at the frontier of the use of computation in the design and realization of complex forms and structures. After studying architecture at the AA, he worked as part of the Advanced Geometry Unit at Arup, and later the Specialist Modelling Group at Foster+Partners. He has taught numerous studios and workshops and presented his work at conferences around the world, and consults and collaborates with a wide range of practices and researchers.

9.2 Kangaroo workflow

The workflow of Kangaroo relies on the same set of rules and operations for low nodal models, such as single digital chains, as high nodal models such as multi-supported membranes. The Kangaroo workflow is illustrated below:

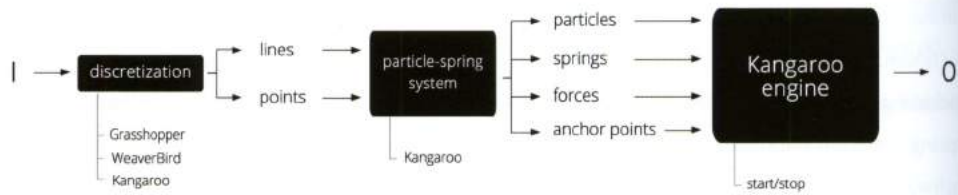


FIGURE 9.3
The Kangaroo workflow.

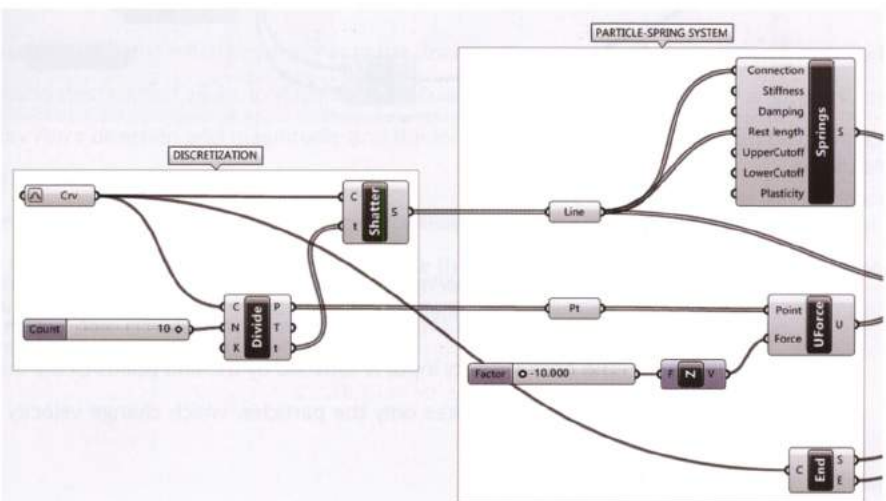
- **Discretization**: a deformable body, e.g. a fabric membrane or a flexible cable, is created by *discretizing* NURBS-geometries and subsequently processing the resulting geometry with a *particle-spring system*. Kangaroo requires that NURBS-curves are converted to lines and NURBS-surfaces are converted to meshes (i.e. points and lines). Kangaroo cannot process NURBS-surfaces and NURBS-curves. The main components used for discretization are hosted within the *Extract* panel of Weaverbird. We can also find useful components within Grasshopper standard tabs or Kangaroo tab.
- **Particle-spring system**: after a geometry has been discretized, lines are converted into springs and points into particles using specific components hosted within the Kangaroo toolbar. Vectors representing forces are applied to particles, and the anchor points are assigned.
- **Kangaroo Engine:** particles, springs, forces and anchor points are connected to the *Kangaroo Engine* in their respective slots. Toggling between True and False statements, using the component *Boolean Toggle* (Params > Input), starts and stops the simulation. While the simulation is running, particles move until an equilibrium state is reached. For this reason the engine's output can be considered as *dynamic*.

9.3 Cable simulation

In the first example, the behavior of a flexible elastic cable – suspended between two ends points and subjected to loads imposed by self weight – is simulated. The first step in the definition is to set a horizontal line drawn in Rhino using the *Curve* container component.

FIGURE 9.4
Representation of a particle-spring system that simulates a flexible cable.

- **Discretization**: the initial geometry is discretized by splitting the set line at division points – calculated by the component *Divide Curve* – using the component *Shatter* (Curve > Division). The N-input of *Divide Curve* sets the number of divisions: the greater the number of division points the greater the final deformation. The *Divide Curve* points are the particles in the system where force is applied or restraints are set. In the following example, N is set to 10.



- **Particle-spring system**: the component *Shatter* outputs a series of unique lines that are converted into springs using the component *Springs From Line* (Kangaroo > Forces), generating the system's springs.

  The output (S) of the *Shatter* component stored in the container component *Line*[27] is connected to the Connection-input and to the Rest Length-input of the *Springs From Line* component. The output (P) of *Divide Curve* is connected to the Point-input of the component *Unary Force* (Kangaroo > Forces) which applies a force-vector to every input point or particle. In this case, a force vector acting in the negative Z direction with a magnitude of 10 is set. Force vectors can be set in any desired direction. Lastly, the anchor points are defined as the end points of the initial curve.

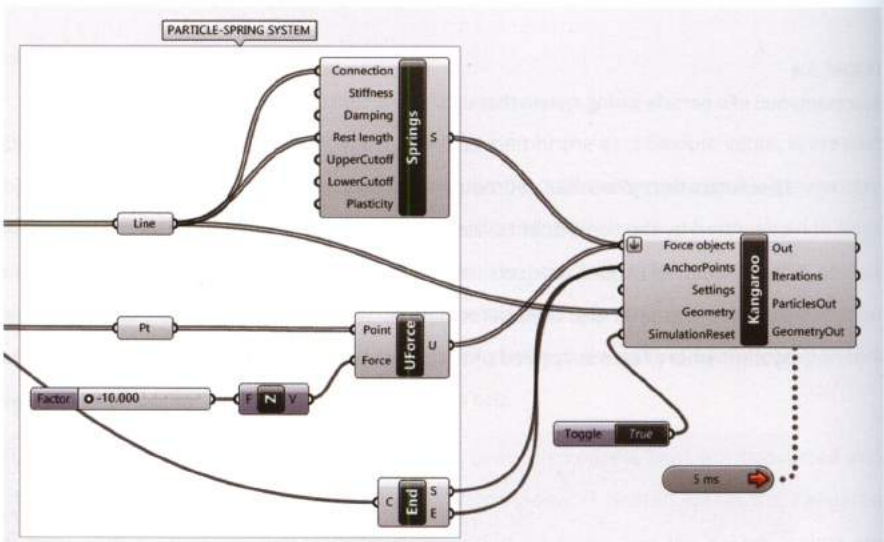


FIGURE 9.5
Particle-spring settings.

- **Kangaroo Engine**: the *KangarooPhysics* engine (Kangaroo > Kangaroo) collects the outputs of *Springs From Line* and *Unary Force* in the Force objects slot, **which must be set to "flatten"**. The AnchorPoints input is satisfied by the end points of the initial curve. By default the simulation influences only the particles, which change velocity while the

NOTE 27
The *Line* container is useful to spot any inputs different from line-geometries (*Springs From Line* can process just line-geometries). If other types of geometries, e.g. curves, come in the *Line* container, it will turn red.

simulation is attempting to reach equilibrium. To visualize the simulation of the cable seeking equilibrium, the output of the *Shatter* component is connected to the Geometry -input of the *KangarooPhysics* component.

To start or stop the simulation a *Boolean Toggle* and a *Timer* (Params > Util) are connected to the *KangarooPhysics* component; which starts (False) and stops (True) the simulation and defines the number of frames per second respectively. Alternatively, if the component is double clicked a contextual panel will appear with a set of buttons: *stop, play, pause*.

In the example, the *Timer* is set to 5 milliseconds (right-click > Interval > 5). The simulation can be started by double-clicking the toggle and changing the Boolean statement to False. Once the simulation is started the particles move in the direction of the force vectors which are restrained by the elastic-linear springs properties. The cable's geometry changes several times until it reaches an equilibrium state.

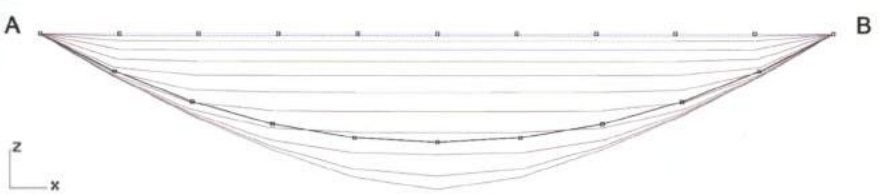


FIGURE 9.6
The frontal view shows the sequence of positions taken by a cable while the Kangaroo simulation is running. The cable bounces several times until it reaches an equilibrium state (dark polyline).

If changes are made to the initial geometry or to the discretization process the simulation is required to be reset and then started again, to visualize the influence of the changes. Other parameters, such as the *Unary Force* direction and magnitude, and the location of the anchor points can be changed during the simulation.

The position of the constraints can be changed manually by setting the anchor points from Rhino instead of relying on the *End Points* component. Once the simulation has started the position of the anchor points can be cchanged "manually": the simulation will react to the change and, once again, seek equilibrium.

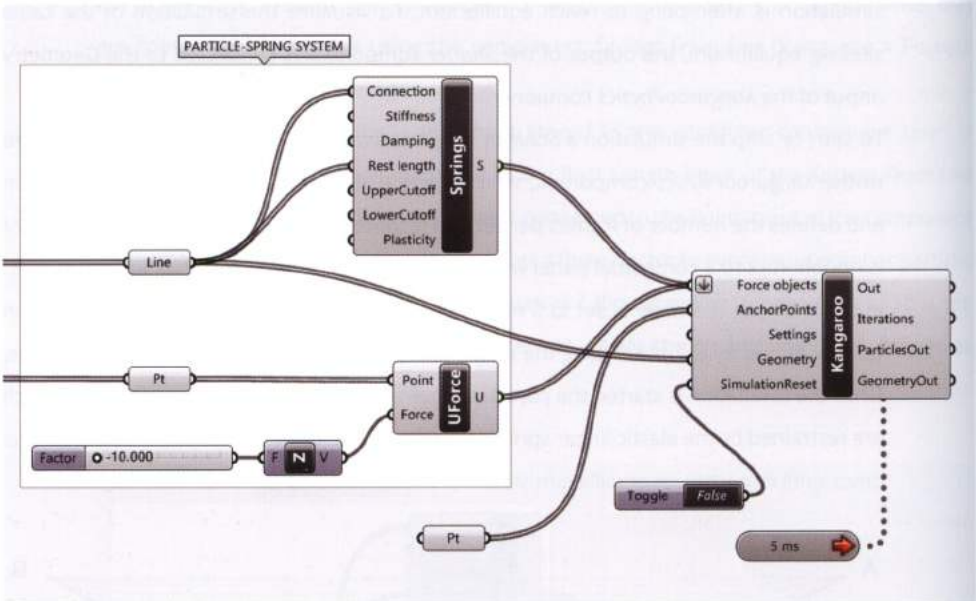


FIGURE 9.7
The anchor points set from Rhino are collected in Grasshopper by the container component *Point*.

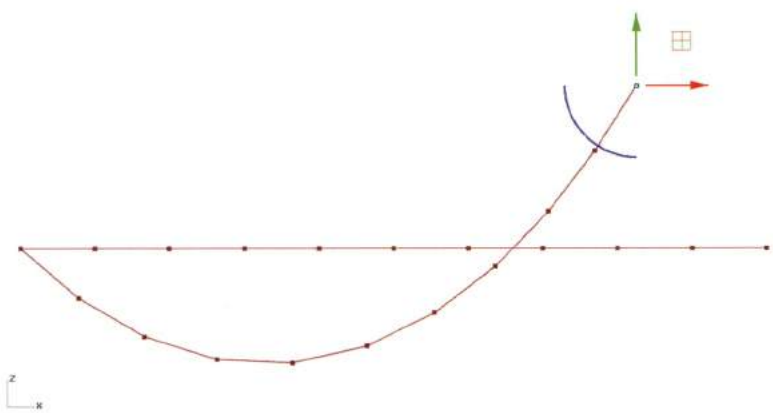


FIGURE 9.8
We can manually move the anchor points set from Rhino: the cable will react as if it were in the physical world.

Manual interaction with the model can lead to errors if the points are not returned to their original position before stopping and starting a new simulation. If the simulation is run without returning the anchor points to their original position the physics engine will not recognize how to restrain the model.

To add additional or new anchor points, points can be set from Rhino or calculated within Grasshopper. Anchor points must be always positioned at particles. For instance, an anchor point drawn in the middle of a spring will have no influence.

FIGURE 9.9
A cable simulation with three anchor points.

9.3.1 Strategy: continuity

Kangaroo's inputs are defined as points, lines or meshes, Kangaroo's output is similarly defined as the same kind of geometries. An elastic cable can be simulated by connecting the ParticlesOut-output to the V-input of the component *NURBS Curve*, defining an interpolated curve through the points. This strategy can be useful to achieve continuity; however, it can lead to physically incorrect results since the Nurbs-curve acts rigidly around point B, which is impossible since particles act as spherical hinges, without moment capacity.

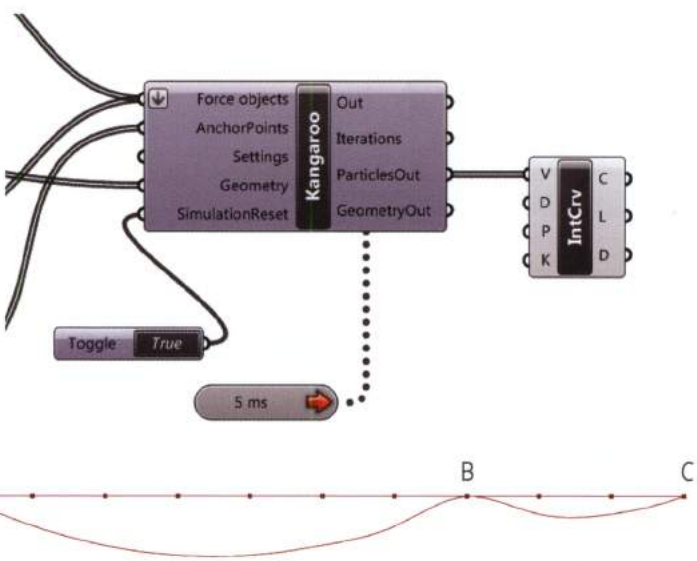


FIGURE 9.10
How to get continuity using an *Interpolate Curve* component connected to Particles Out.

9.4 Elastic behavior: Hooke's law

When the cable reaches an equilibrium state induced by influence of the external forces, and resisted by the elasticity of the springs, the length of each segment increases. For example, the cable illustrated below has a start length of 10 units. The curve is discretized into 5 parts each 2 units in length. Six identical unary forces with a magnitude -20 units in the *Unit Z* direction, are applied to the particles; yielding a final curve with an overall length of 10.46 units, hence a deformation of 0.46 units. This deformation distance is not split evenly among the segments, instead the springs closer to the reactions elongate further. This is because, they bear the weight of the other springs.

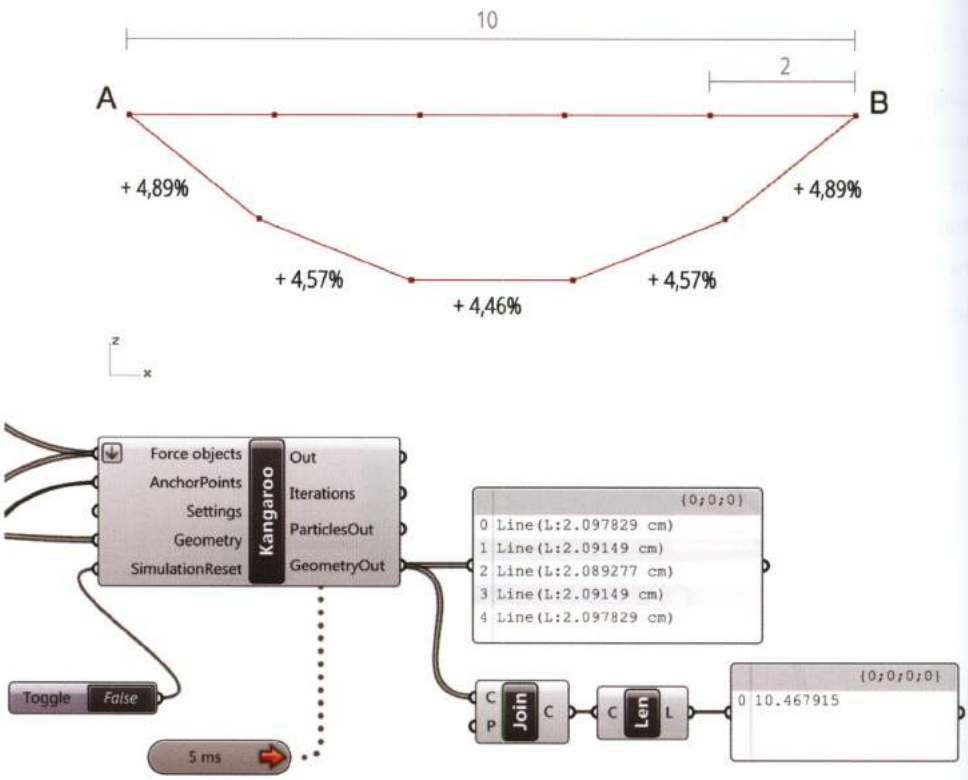


FIGURE 9.11
The deformation causes an uneven extension of the springs with a higher extension of anchored springs.

Kangaroo's elastic behavior follows Hooke's law which states: *displacements or size of the deformation of a body (treated as a spring) is directly proportional to the deforming force or load.* Under these conditions

the body returns to its original shape and size when the loads are removed. Mathematically, Hooke's law is formulated by the expression:

$$F = k \cdot X \quad [1]$$

where:

- **F:** is the applied force, commonly expressed in Newtons (N);
- **k:** is a positive constant called the *Stiffness*. The value of *k* depends on material and cross sectional geometric properties of the elastic body. The constant *k* is commonly expressed in N/cm;
- **X** is the *change in length* or deformation of the body (spring), commonly expressed in cm.

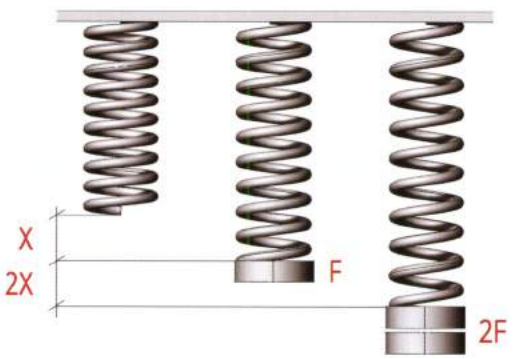


FIGURE 9.12

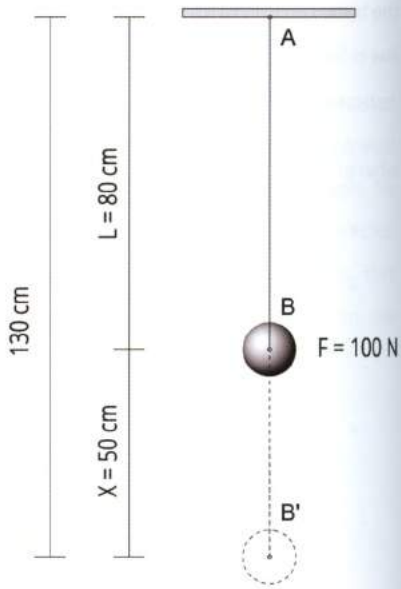
Hooke's law: the springs extension is proportional to the force.

Hooke's law is embedded in Kangaroo's *Springs From Line* component.

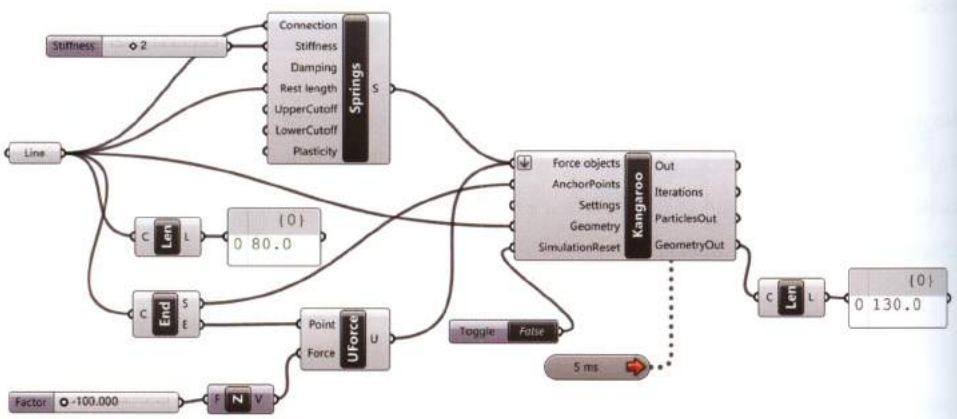
To illustrate how the component *Springs From Line* operates a 100 N weight is applied to an anchored cable with an initial length of 80 cm and a *stiffness* of 2 N/cm. In accordance with Hooke's law, the change in length of the cable is:

$$X = F / k \rightarrow X = 100 / 2 = 50 \text{ cm}$$

FIGURE 9.13
Deformation of a suspended cable according to the Hooke's law.



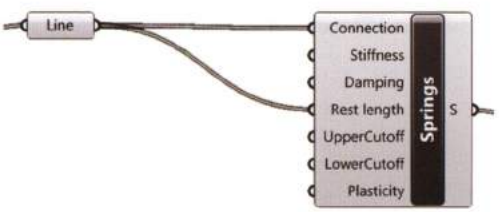
If the load is removed, the cable will return to its start length (80 cm). The length a cable reaches when loads are removed is called the ***Rest Length***, which not always coincides with the start length (as explained later). The time the cable takes to reach an equilibrium state depends on the *Stiffness* as well as the *Damping Constant* which is related to friction caused by drag. The greater the *Damping Constant* the lower the deformation velocity. The *Damping Constant* does not affect the change in lenght but only the deformation velocity.



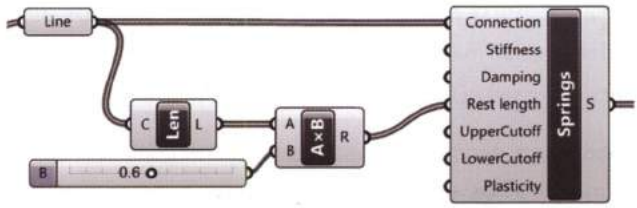
The relative algorithm in Grasshopper considers the cable as a single spring; the calculated results of the spring's deformation match Hooke's law. The *Springs From Line* component embeds the physical characteristics discussed in the previous example, as well as other important parameters including:

- **Connection**: springs are linear elastic connections. The Connection-input requires *lines*, any other geometry will yield null results. Every line's initial length is called the *Start Length*.
- **Stiffness:** according to the Hooke's law, the Stiffness-input, sets the springs' stiffness or k value. The higher the k value, the lower the deformation. Stiffness is determined by material properties as well as the area of spring's cross-section.
- **Damping:** the Damping-input influences the deformation velocity, with no influence on the change in length. By default Damping is set to 10.
- **Rest Length:** the length that a spring endeavors to reach once the loads are removed. The Rest length-input is essential to simulate behaviors of different materials. Three cases can be distinguished:

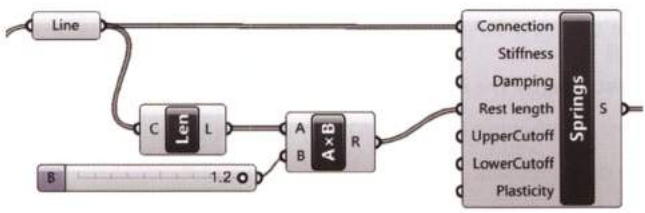
    1. *Rest Length = Start Length*. This condition mimics perfectly elastic behavior and is achieved by connecting the springs (Lines) to the Rest length-input. All simulations demonstrated so far have followed this condition.



    2. *Rest Length < Start Length*. This condition imitates the effect of pre-tensioning the springs which **shortens** their length. *Rest Lengths* which are less than the *Start Length* simulates tensile materials that attempt to minimize their length or area. This condition is accomplished in Kangaroo by using the component *Length* to measure the initial springs' length then multiplying it by a **Rest Length Factor** ranging between 0 and 1.



3. *Rest Length* > *Start Length*. This condition replicates the relaxation or lengthening of springs. This condition is accomplished in Kangaroo by using the component *Length* to measure the initial springs' length then multiplying it by a *Rest Length Factor* ranging between 1 and N.



- **Upper/Lower cutoff**: sets limits for the springs to operate, below or above respectively. By default the *Upper/Lower cutoffs* are set to 0.
- **Plasticity:** the maximum elastic deformation, as compared to the rest length.

9.5 Catenary simulation

A catenary curve is formed by a perfectly flexible, uniformly dense, and inextensible cable suspended from two end points. The equation of a catenary curve is:

$$y = a \cdot \cosh(x/a) \quad [2]$$

The distance from the x-axis to the point on the curve with a tangent line slope equal to 0 is expressed as variable *a*.

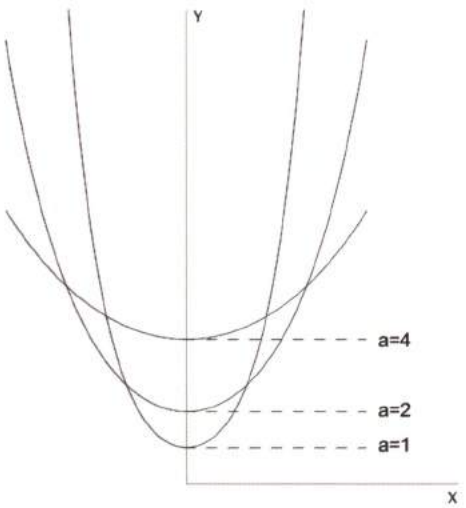


FIGURE 9.14
Catenary graph for different values of *a*.

The curve can be calculated by the *Evaluate (Fx)* component, as showed below, with *a* set to 2.

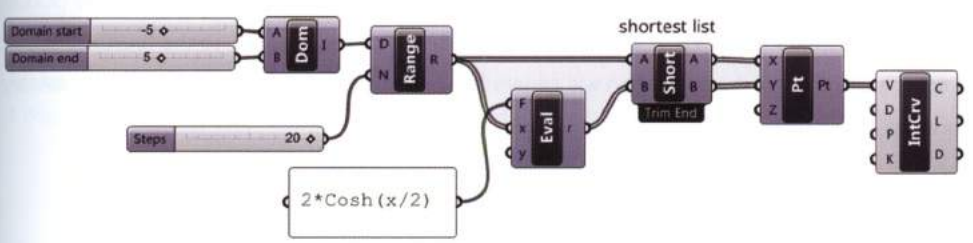
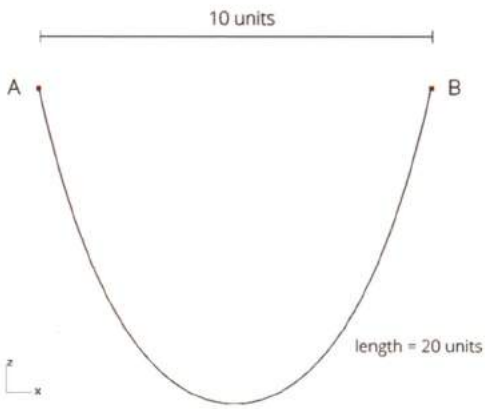


FIGURE 9.15
Algorithm based on the catenary equation.

A catenary curve can also be drawn by the component *Catenary* (Curve > Spline), which embeds the equation of a catenary curve [2]. The *Catenary* component requires as inputs: start (A) and end (B) points of the catenary curve, the length (L) of the curve, and the gravity direction (G).



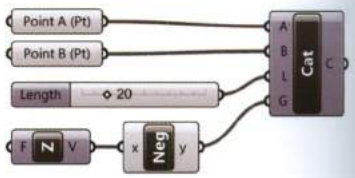


FIGURE 9.16
The catenary algorithm and generated curve.

Catenary curves can also be simulated using *particle-spring systems*. The catenary definition is: "*a curve formed by a perfectly flexible, uniformly dense, and inextensible cable suspended from its endpoints*". Therefore, the curve must comply with four conditions:

1. To be suspended by its end points;
2. To be perfectly flexible;
3. To be uniformly dense;
4. To be inextensible.

These conditions are not entirely met by the deformable linear-curve discussed previously (9.3). In fact, even if a high *stiffness* value is set, the springs will not be inextensible, so the fourth condition is not met. As a result the simulation will generate a curve that is slightly different from a catenary curve, as illustrated in the following image.

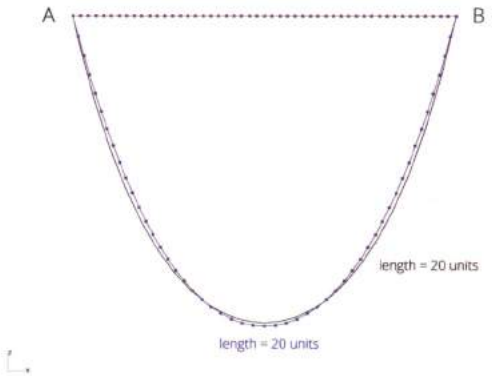


FIGURE 9.17
A line with applied gravity loads at the particles, will output a curve that is slightly different from an catenary curve.

To more closely approximate a catenary curve an arc can be used as the starting geometry, described using the component *Arc 3Pt,* through a set of three points. Then the measured arc-length is set equal to the length of the desired catenary curve.

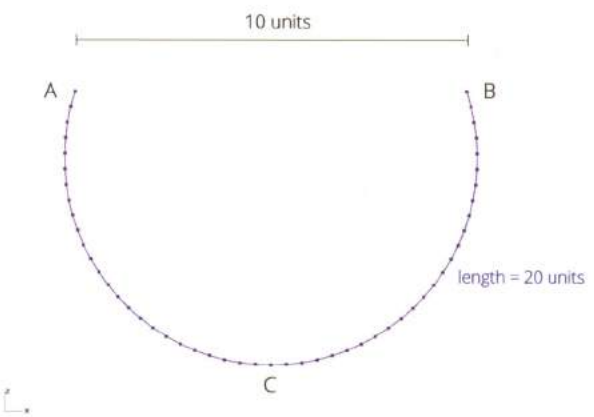
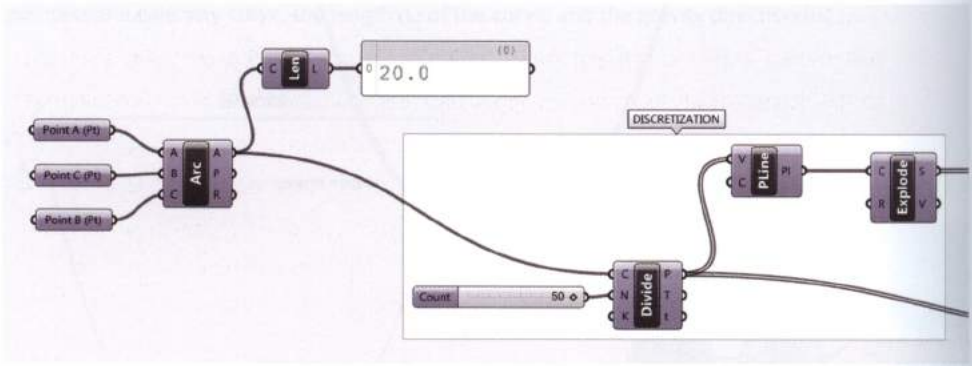
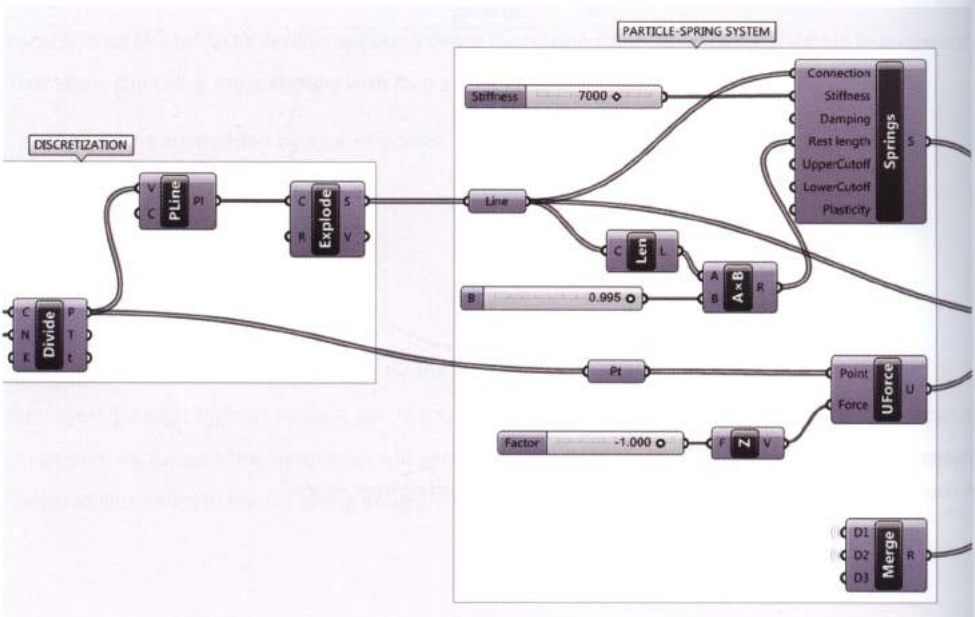


FIGURE 9.18
A close approximation of a catenary curve can be obtained starting from an arc.

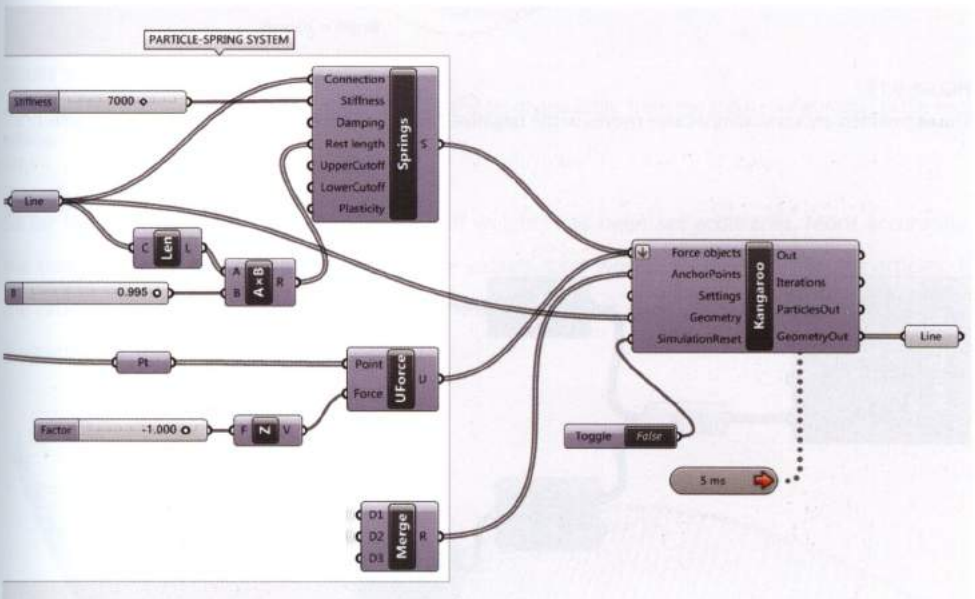
- **Initial geometry**: is an arc with a length 20 units drawn through three points: A, B and C.



- **Discretization**: the arc is then divided using the component *Divide Curve* with the N-input set to 50, yielding 51 equidistant points. A polyline is created through the resulting points and exploded into segments by the component *Shatter* in order to return 50 lines. The lines are then converted into springs, each initially measuring 20/50 = 0.4 units.



- **Particle-spring system**: the output of the *Explode* component is connected to the Connection-input of the *Springs From Line* component, after passing through a *Line* container component. To achieve *inextensibility* of the cable: the rest length is set such that the *rest length* < *start length* using a multiplier factor of 0.995, and a Stiffness-input set to 7000 units. Gravity loads are applied to each particle using the component *Unary Force* (Forces > Kangaroo) in the Z direction with a magnitude of -1, the resulting vectors are connected to the Force objects-input of the component *Kangaroo*. Set points A and B, are combined into a single list using the *Merge* component and connected to the AnchorPoints-input of the component *Kangaroo*.



When the simulation is initiated the segmented arc moves in the negative Z direction taking the form of a catenary cable. The calculated polyline complies with the four conditions of the catenary definition. The segments change minimally in length from their start length (0.4 units) after the simulation.

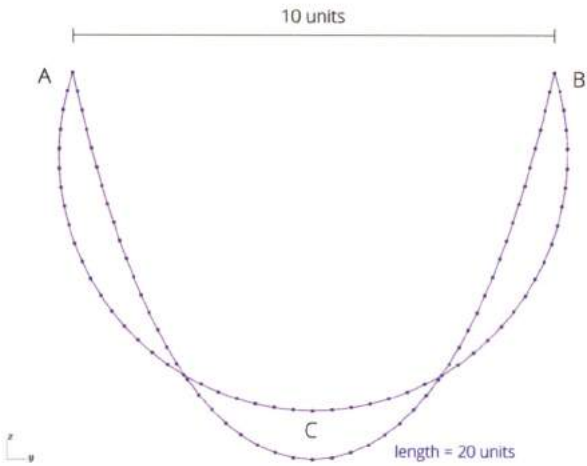


FIGURE 9.19

The segmented arc simulating a cable moves in the negative Z direction taking the form of a catenary cable.

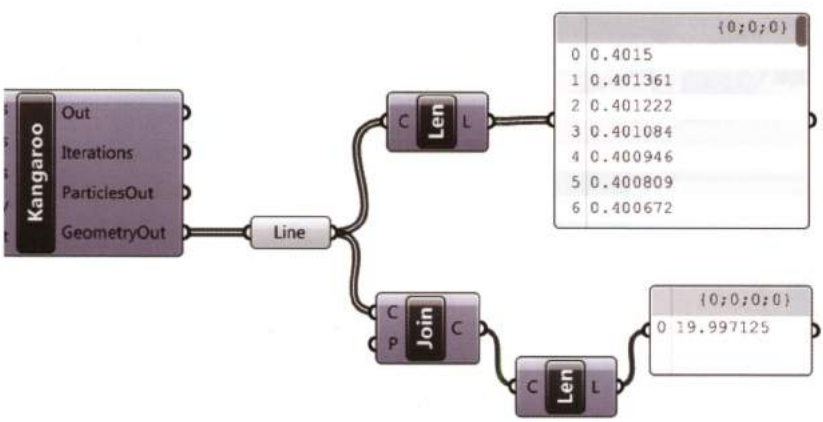


FIGURE 9.20

The segments minimal change in length from their start length (0.4 units) after the simulation.

FIGURE 9.21
The image shows the sequence of positions taken by the arc-shape cable, from the initial configuration to the final catenary form.

So far the *Unary Force* value (representing self weight) has been set arbitrarily. More accurately, the *Unary Force* should be set by dividing the cable's total weight by the number of particles. If the catenary has a weight of 10N and is divided by 51 particles each unary force vector will have a magnitude of 10/51 = 0.196N.

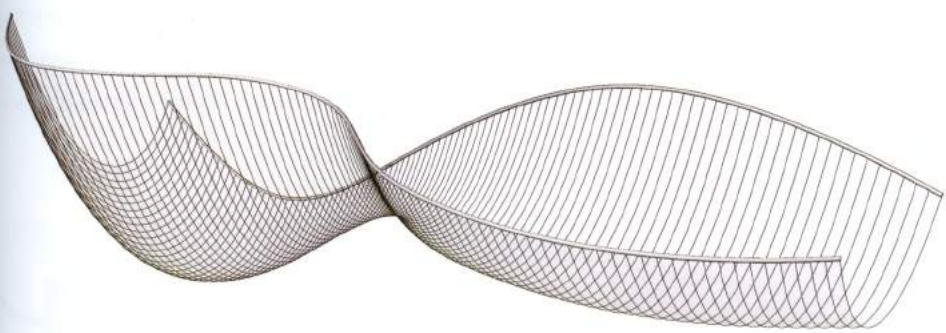


FIGURE 9.22
The image shows a set of catenaries whose start-points lie on two freeform frames.

9.6 Membrane simulation

To simulate membranes or other sheet materials such as fabrics, a grid of springs is defined. Grids can be established using numerous strategies; the most commonly used technique is to convert a NURBS surface to a mesh, then extract the edges and vertices that will become the springs and particles of a particle-spring system.

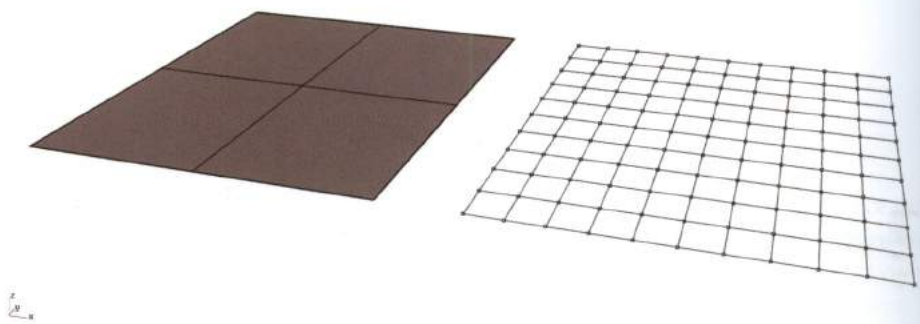
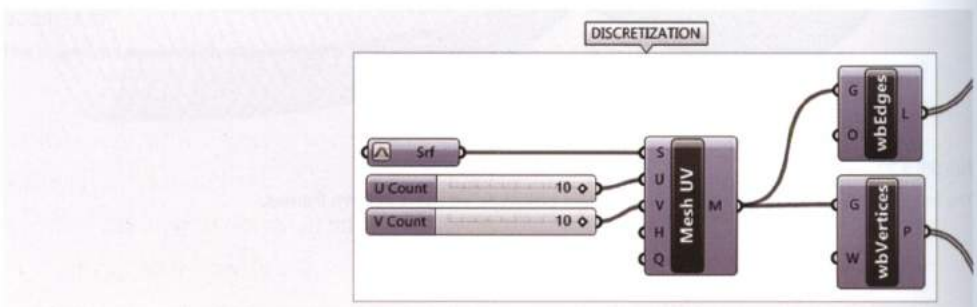
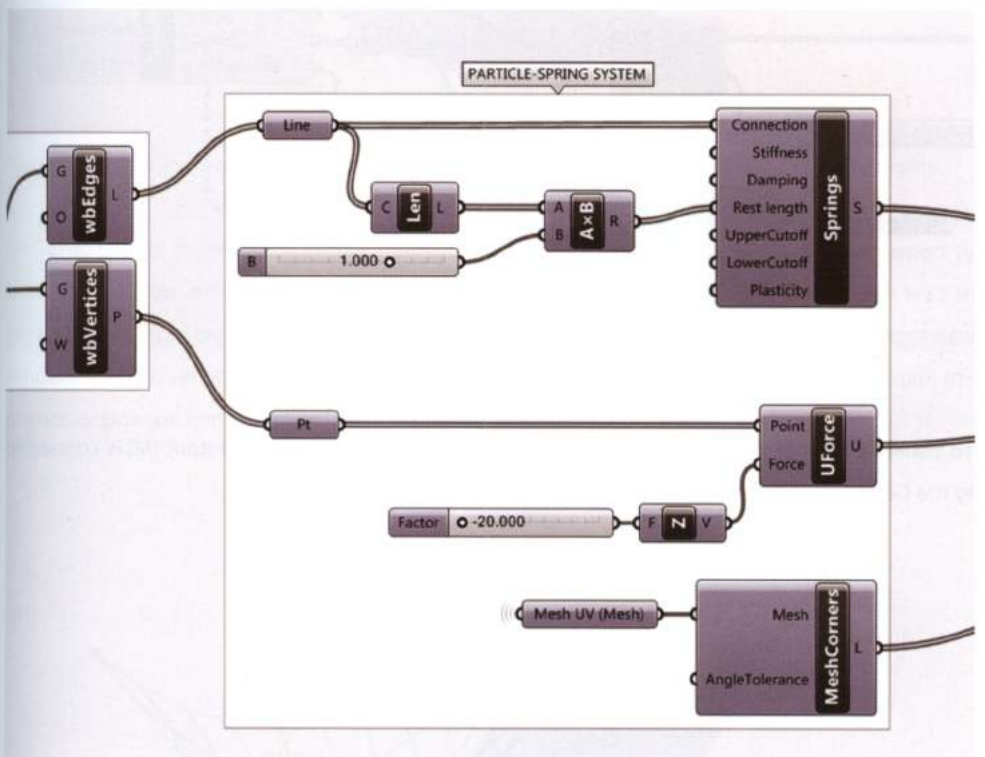


FIGURE 9.23
A grid of springs can be obtained by extracting vertices and edges from a mesh.

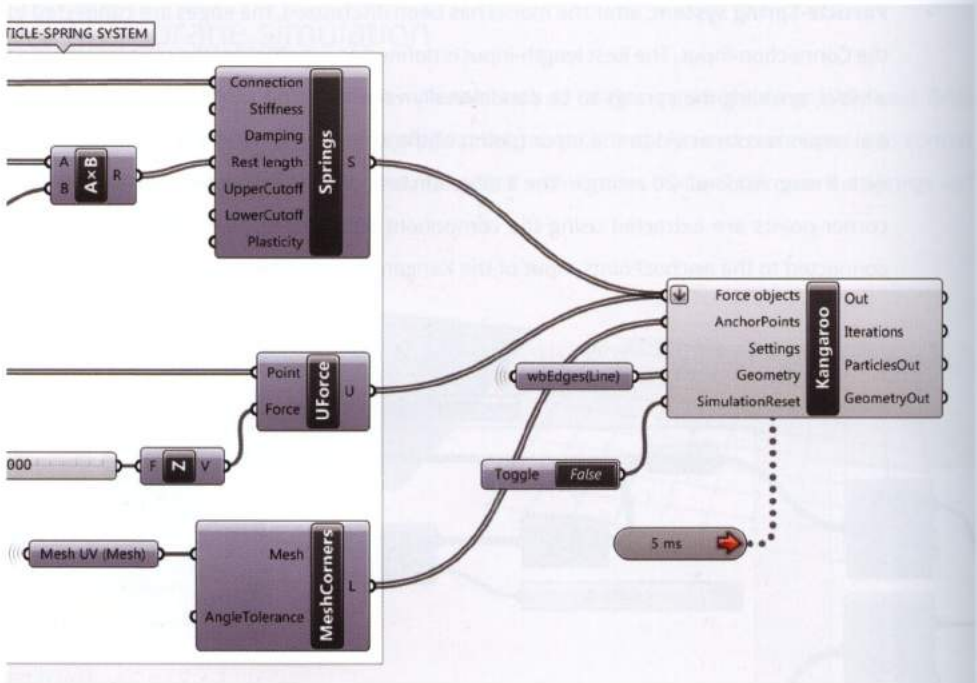
- **Discretization**: The behavior of a rectangular flat membrane anchored at four corners points and subjected to gravity loads can be simulated using particle-spring systems. To construct the discretized model: first, a set NURBS surface is converted into a mesh using the component *Mesh Surface* (Mesh > Util), then the components *wbEdges* (Weaverbird > Extract) and *wbVertices* (Weaverbird > Extract) are used to extract the mesh edges and vertices respectively.



- **Particle-Spring system**: after the model has been discretized, the edges are connected to the Connection-input. The Rest length-input is defined as a multiple of the start length using a *slider*, enabling the springs to be conditionally varied. The output (P) of the component *wbVertices* is connected to the input (point) of the component *Unary Force;* a force vector with a magnitude of -20 acting in the Z direction is applied to each particle. Next, the four corner-points are extracted using the component *MeshCorners* (Kangaroo > Utility) and connected to the AnchorPoints-input of the Kangaroo component.



Lastly the output (springs) of the component *Springs From Line* and the output (U) of the component *Unary Force* are connected to the Force objects-input of Kangaroo, in flatten mode. The *wbEdges* component output (L) is connected to the Geometry-input of the Kangaroo component to define the geometry for the simulation to output. Double-clicking the Boolean toggle switching from True to False initiates the membrane simulation. The membrane, anchored at its corners, is deformed by the *Unary Force* vectors and resisted by the springs stiffness value.



To visualize the mesh faces instead of edges the *Mesh Surface* component output (M) is connected to the Geometry-input of Kangaroo.

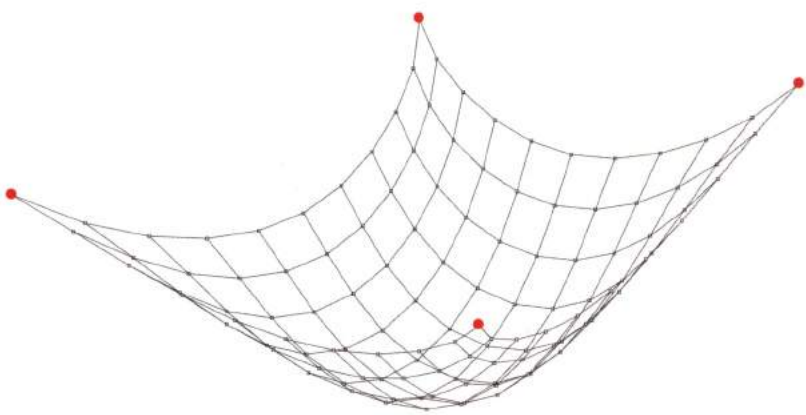


FIGURE 9.24
The membrane, anchored at its corners, is deformed by the *Unary Force* vectors and resisted by the springs stiffness value.

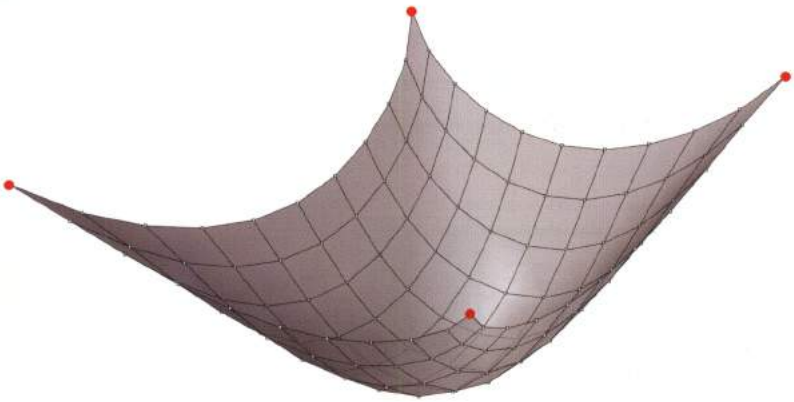


FIGURE 9.25
Mesh faces can be visualized instead of edges by connecting the *Mesh Surface* output to the Kangaroo engine.

Similar to cables, membranes anchor points can be set from Rhino then adjusted with respect to their XYZ position after the simulation is started. Manual interaction with the model can lead to errors if the points are not returned to their original position before stopping and starting a new simulation. If the simulation is run without returning the anchor points to their original position the physics engine will not recognize how restrain the model

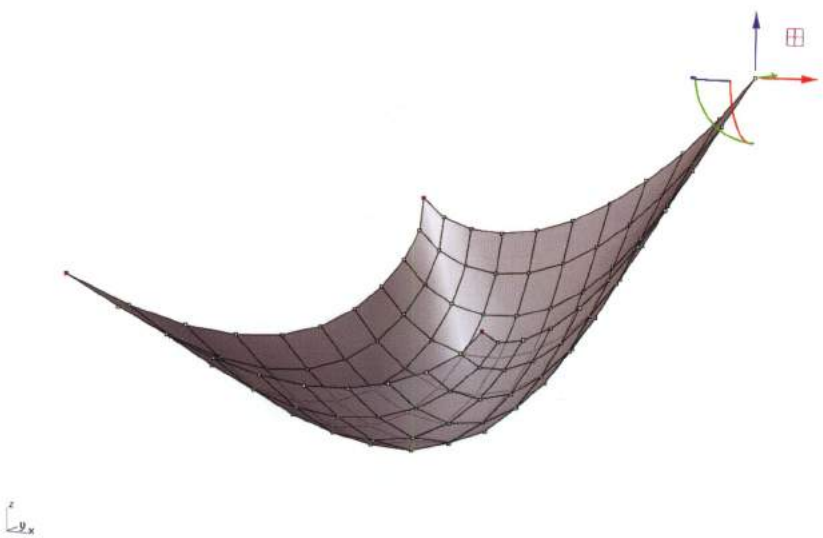
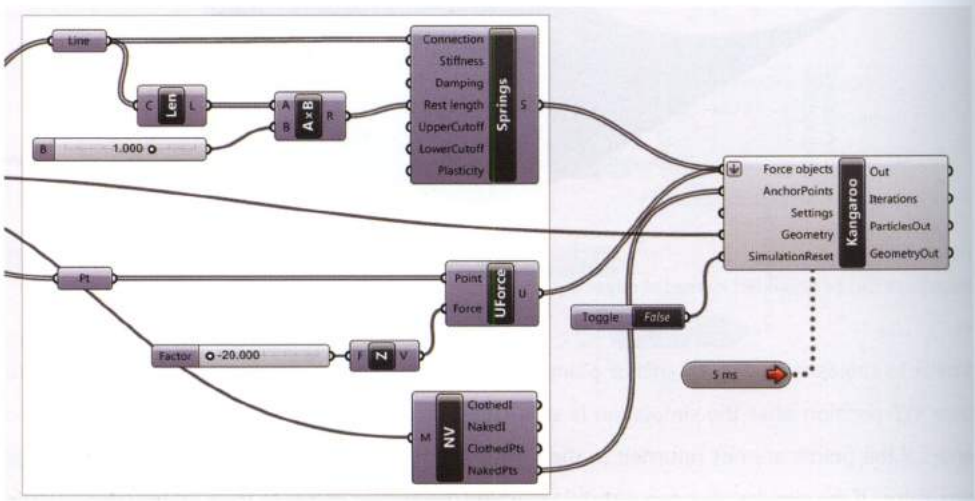


FIGURE 9.26
Membranes anchor points can be manually manipulated in Rhino.

Of course a membrane can be "multi-anchored" provided that anchor points are positioned on particles. Membranes can also be anchored along their edges, by setting the anchor points as the meshes naked vertices. The component *Naked Vertices* (Kangaroo > Utility) extracts the naked edge vertices, i.e. vertices not bordered by faces from a given mesh.



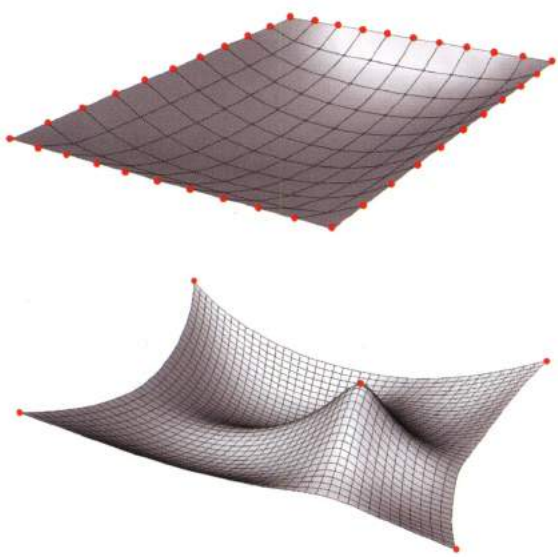


FIGURE 9.27

Top: membrane with naked vertices as anchor points. Bottom: membrane with corner and internal anchor points.

The membrane's area can be minimized by setting the rest length factor to 0. With a rest length set to 0 the membrane behaves like a tensioned-film simulating a soap film.

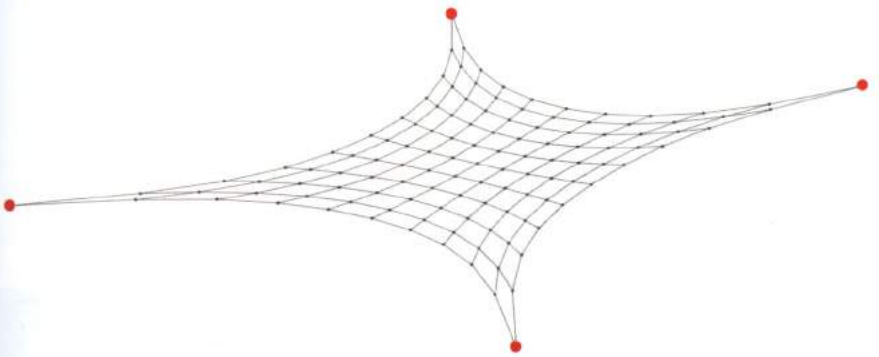


FIGURE 9.28
A membrane with Rest Length Factor equal to 0.

The membrane so far studied behaves like a **cable net**. To simulate a more rigid membrane, **diagonals can be added which prevent the mesh-faces from deforming into diamond shapes**. To computationally mimic the behavior of sheet materials two *Springs from Line* components are used to separate the mesh-edges and the diagonals.

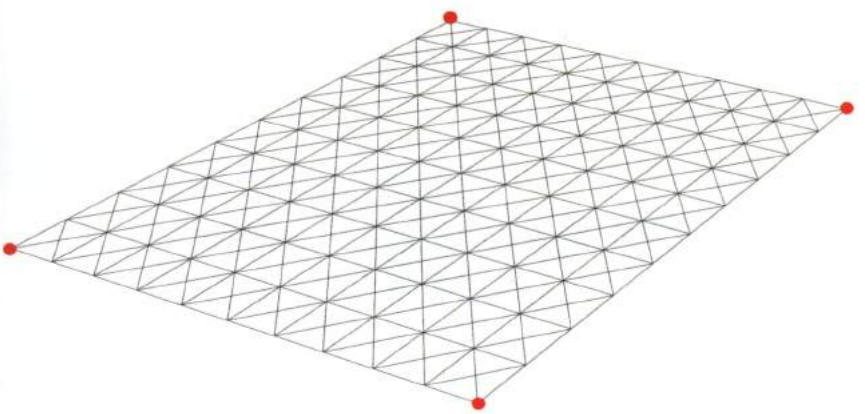


FIGURE 9.29
Diagonals prevent the mesh-faces from deforming into diamond shapes.

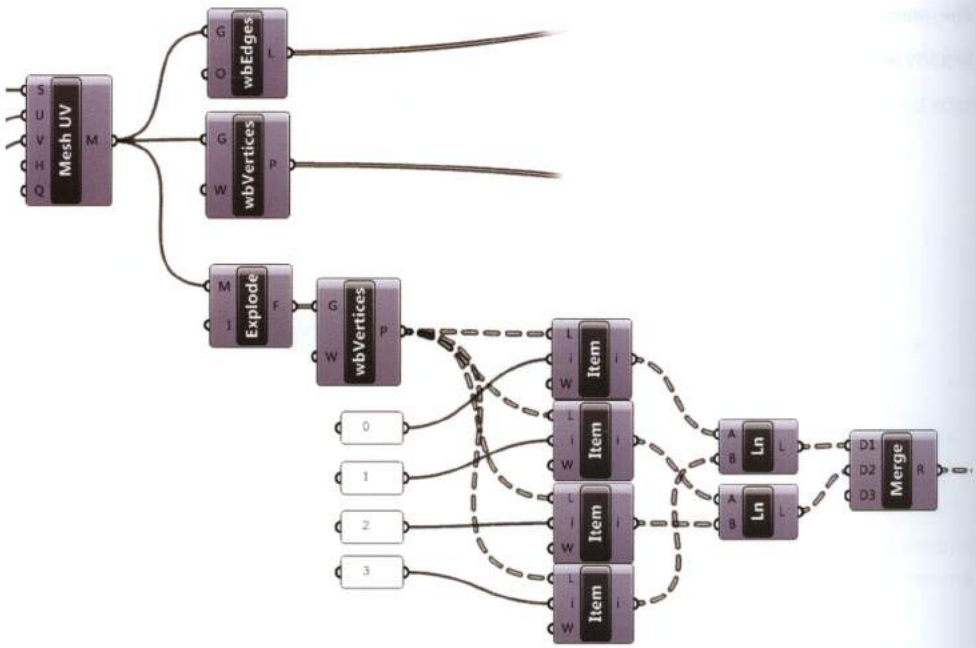
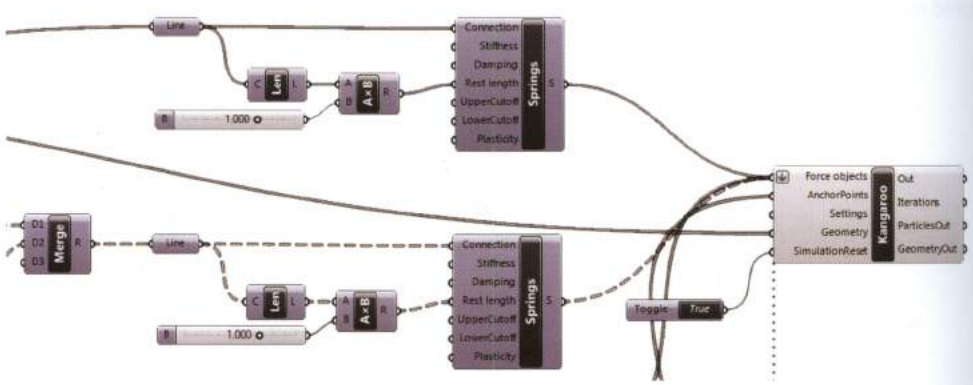
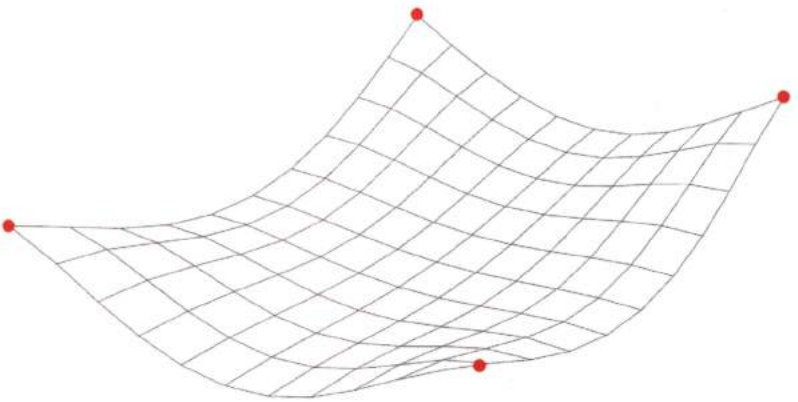


FIGURE 9.30

Two *Springs from Line* components are used to separate the mesh-edges and the diagonals.



Once the simulation is started the more rigid configuration no longer behaves like a cable-net, but instead like a sheet material. If the *Rest Length Factor* is reduced the membrane will form origami-like creases.



Additionally, different diagonals configurations can be tested which yield varying results. For example, if one diagonal is set per quad (image A), asymmetrical behavior can be achieved by setting different *Rest Length* values for the edges. The component *WarpWeft* (Kangaroo > Mesh) can also be used to separate the edges of a mesh according to *warp* and *weft* directions (C).

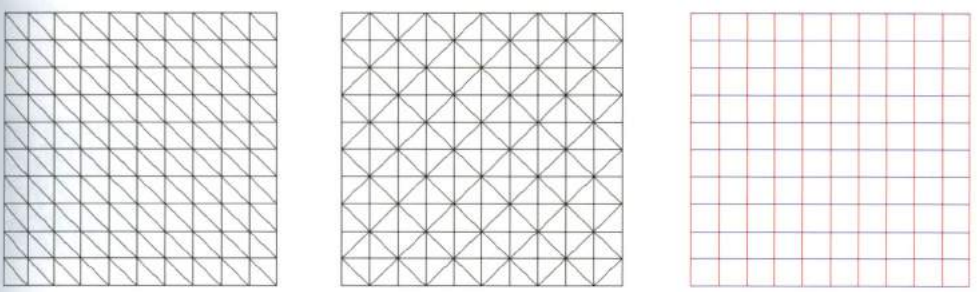


FIGURE 9.31
Different mesh configurations can be set to simulate specific behaviors.

9.6.1 Practical Exercise: multi-supported membrane

The Bach Chamber Music Hall by Zaha Hadid Architects, composed of a continuous ribbon of stretched fabric that intertwines to envelop the performers and the audience, is the focus of the following practical exercise.

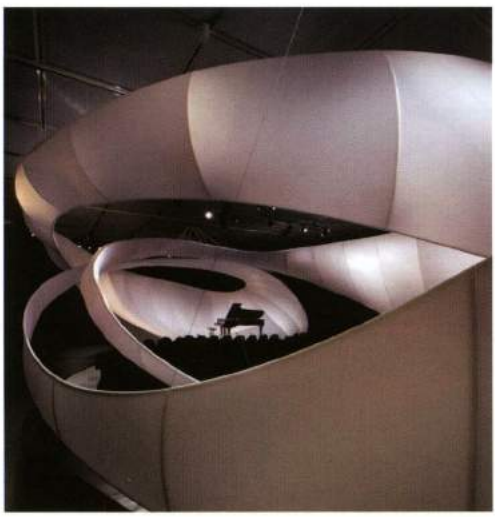


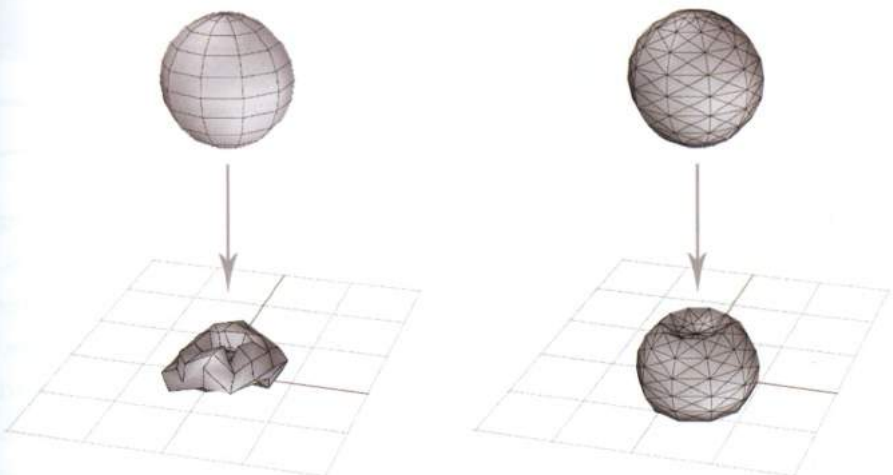
FIGURE 9.32
The *Bach Chamber Music Hall* by Zaha Hadid Architects. Manchester, UK (2009). Image courtesy of Zaha Hadid Architects. Image copyright by Luke Hayes.

Instead of studying the entire ribbon, a section is extracted and studied in a simplified model. Specifically, the model investigates the behavior of a membrane stretched on a truss support-structure composed of 2 horizontal free-form beams and 11 vertical variable profile arc-shape beams.

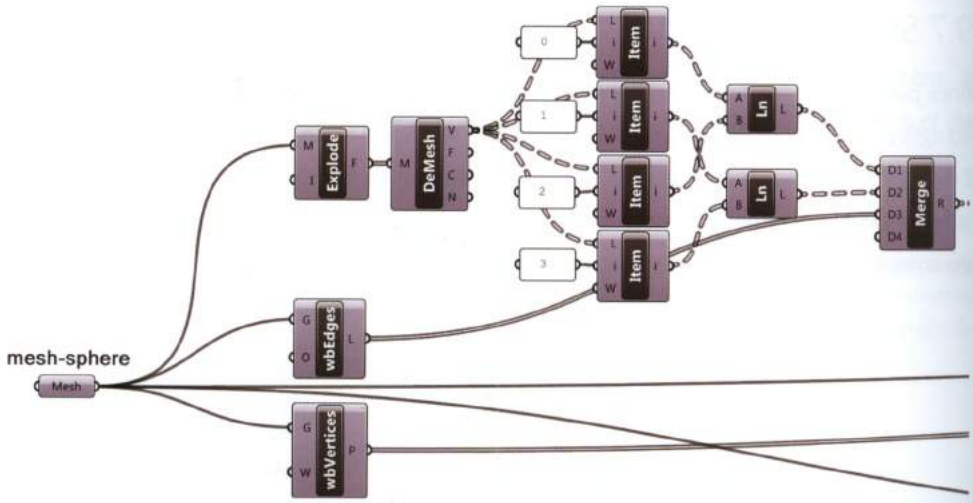
The complete exercise can be found using the following QR code. 4

9.7 Shell behavior

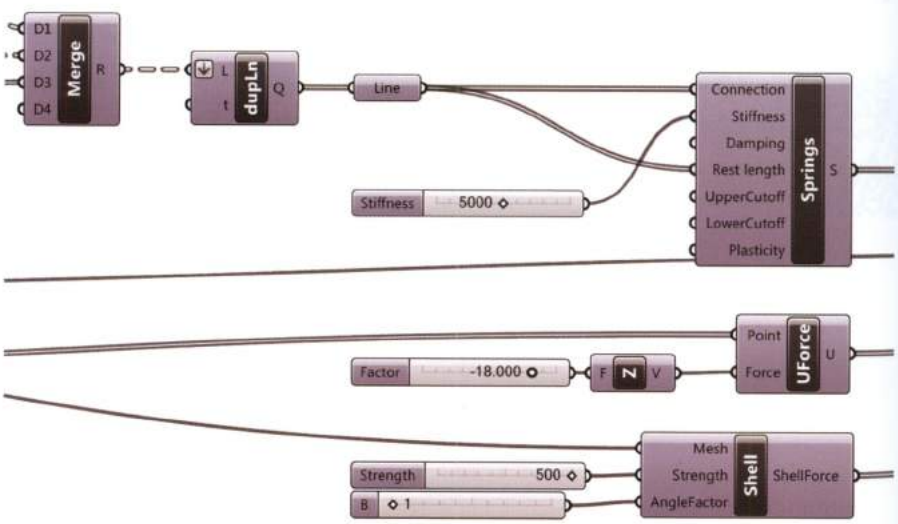
Since particles behave like spherical hinges, without moment capacity, a discretized model cannot act rigidly without additional restraint. For example, a simulated mesh-sphere moving in the negative Z direction under the action of gravity will crumple (even with reinforcing mesh diagonals and a high stiffness value) when the sphere reaches the XY plane or the simulated floor. To prevent crumpling the component *Shell* (Kangaroo > Utility) can be utilized.

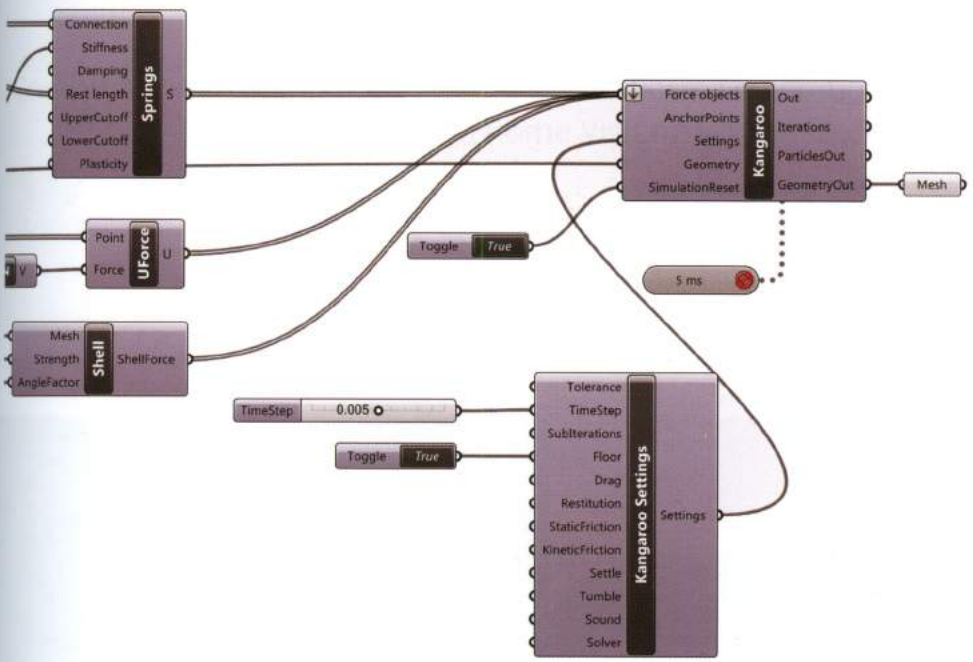


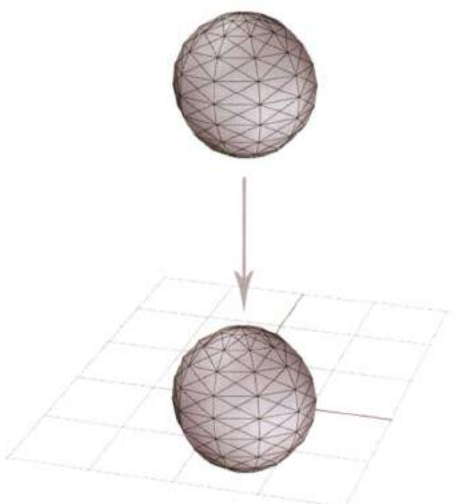
- **Discretization**: The mesh edges and vertices are extracted from a mesh sphere set from Rhino. The points (or particles) are connected to the *Unary Force* component, and the lines (edges and diagonals) are converted into springs. The diagonals are defined by connecting the opposite vertices of each face using the *Mesh Explode* and *Deconstruct Mesh* components.



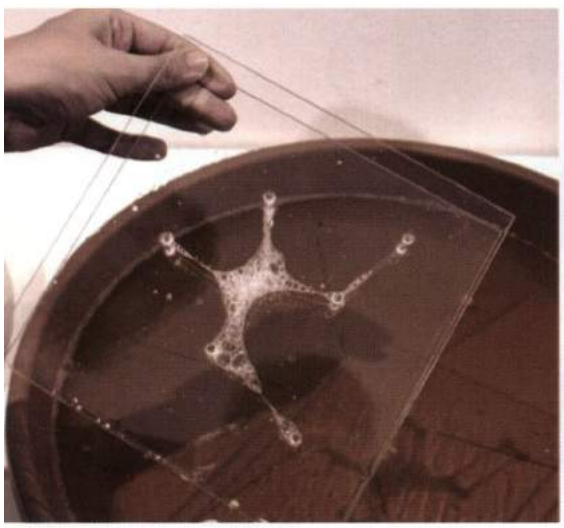
- **Particle-spring system**: The edges and diagonals are merged into a single list using the *Merge* component. Since the mesh-sphere has triangular faces around the poles, the edges and diagonals overlap. This condition would impact the simulation. The component *removeDuplicateLines* (Kangaroo > Utility) removes coincident lines within a tolerance. The output (Q) of the *removeDuplicateLines* component is connected to the inputs Connection-input and Rest length-input of the *Springs* component, with the stiffness set to 5000 units. Lastly, the *Shell* component returns a shell force from the input mesh with an input strength set to 500 and an input angle factor set to 1.

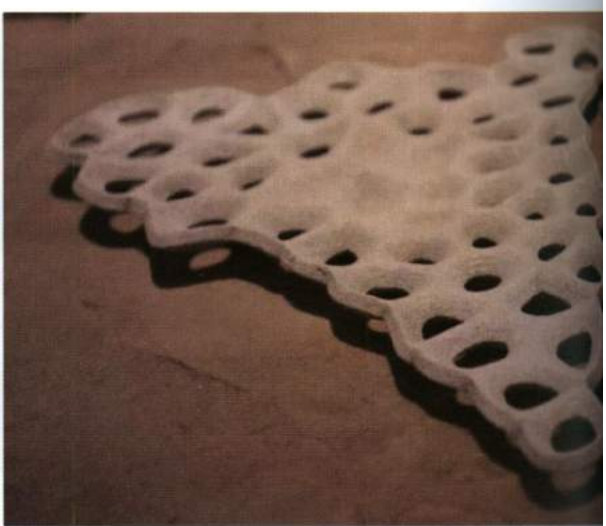






The *Shell* component is useful when simulating materials with bending resistance, such as steel or rubber sheets.





'Synopsis' by Faisal Al Barazi, Michela Falcone, Myrto Grigori and Vittorio Paris, one of the projects of the AA Rome Visiting School "Form As Unknown (X)".

Form as Unknown

Computational Methodology and Material Form Generation in the AA Rome Visiting School Workshops.

Lawrence Friesen, Lorenzo Vianello

"In nature, shape is cheaper than material."

(Lightness. Beukers, Van Hinte)

Material is computational. It is composed of effects and relations, contained in behavior. Each assembly of material system employs a series of characteristics of affections that relate one to another through a system of events to behave in a particular way. The built world around us is composed of relational systems that materially respond to the environment from which architects and designers seek to rationalize and interpret in form. Evolving over centuries, the organization and structure of our built environment is explicit in the urbanized development.

Cities that have resulted in the slow migration and flow of people in response to changes in technology and economies of social organization are generated as a system of relational material events and experiences. The rationalization of these experiences constructs an ordered and predictable form that is recognizable as the built environment; it is generated from a system of economies, and needs which determine the form of the built environment.

> *We live in a world populated by structures - a complex mixture of geological, biological, social, and linguistic constructions that are nothing but accumulations of materials shaped and hardened by history (Manual De Landa. A Thousand Years of Non Linear History. 1997 Zone Books).*

In architecture we are concerned about material that defines space and informs experience. The form of any structure is the result and negotiation of material constraints and the forces that act upon it to produce form. The form making properties are part of the component of behavior embedded in material. Behavior, material and architecture have the expressed relation to predict form, structure and space.

As part of the **AA Rome Visiting School Workshops - Form as Unknown (X) and Form as (Dynamic) Unknown** - the exploration of material through behavior affects the way we see the development of form and structures. Behavior generating form from the relationship of material performance as exemplified in the work most famously of Antoni Gaudì, Frei Otto, Sergio Musmeci and many others, is revealed through the interpretation of material to become a related system of derivation. Working through the experimentation with materials and utilizing the observations and discussions from the famous trio, the Rome workshops sought ways to use this form making methodology to experiment with the translation of a material events to active agents in parametric terms. The material properties themselves are an indication of form making potential, but are indicative in form making and require a translation in scale and properties. The event in form generating becomes a material guide to the understanding a method of interpretation that is material based. The material experimentation indicates a specific effect of shape making that is translatable and predictable based on interpretation. Form generation indicates the clues that can be computed to reveal a way of seeing or understanding the form potential, but through a force generation rather than a purist geometric topology.

Material as Information

Form finding as the method of developing the rational concept of form and structure is predicated on the understanding of material as information. Otto's research in the performance of fabric and soap film materials in the context of the development of the methodology of light weight structures were crafted from observable qualities in natural structures and material behavior. Understanding the natural material organization has bred a new understanding of material for the development of new architectural form. The structures that have been built in response to that exploration are many. The knowledge of these structures has influenced the methods of construction, the use of materials and the way we fabricate. With the more recent advent of parametric design strategies, originally with MAYA and GC and now with Grasshopper, designers have been responsive to the understanding of a new language of form. Although initially providing carefully ordered interpretation of geometries that are artfully formed, they often were void of information that would reflect the making or indeed the environment in which it was placed. They appeared as unbuildable objects that had no relation materially or dynamically to the forces that would engage material in form and structure is often left to the engineer to "work out" the solution independent of form.

The wealth of algorithmic tools that have directly interfaced with the software, in particular Grasshopper for its open sourced approach to development, is now playing a larger role in the development of architecture that is informed from the environment and by building strategies,

there is now a greater potential of interpreting the building information and environmental data in a series of code driven formalizing strategies. If we consider the vastness of data that is collected today from the sources of research and raw data from the number of sensors that register behavior, like mobile phones, personal devices, card data readers and other devices, there is an enormous wealth of data that may influence the way we see and understand cities in which we live, and provide a system of information based support to affect the way make architecture informed by data and responsive to change.

Responsive Design Strategy

Generative design methods and scripting strategies provide architecture the capacity of dealing with large amounts of data and information affecting the composition of urban design and architecture. It also holds a tremendous methodology for the understanding of responsive and mapping capability that informs design to work toward a responsive design strategy.

In traditional forms of design points of analysis are fixed in the development of a design strategy. The information and decisions that make up the process are viewed as a snapshot in time or as an aggregation of moments to a fixed picture while still in the information generation stage of design development. Changes or fluctuations in time are flattened or discarded, subtleties and variation are rationalized, in favor of the generic form.

An algorithmic approach leaves open the fixed point to be variable and responsive, creating a higher order responsive network. As referenced in Autopoiesis of Architecture (Schumacher)

> *The development of computation and scripting techniques was established through a process of development to enrich formal information in the metric diagram. The metric diagram is constrained, and defined by limits without variable quantities. Bringing an order of variability to the metric diagram transforms to a parametric diagram (Schumacher).*

Luigi Moretti first discusses the concept of the parametric diagram in architecture when he adopts the mathematical term Parametric into an architectural lexicon in the 1940's and explores the relational systems that are prevalent in the form of stadia built up of parameters affecting form. Forms become defined by the "forces" acting on them. These forces are observable quantities that help to define a relational system where he alludes to generation of information to transform a formal system. This was of course a highly mathematical approach that was accomplished before the use of computers and software to aid in the data management to control variables, but the form generated is unique and descriptive of the system that produced it.

What happens when we design through a process that utilizes information to affect a system to

become generative? A system that is reflective of the material forces that define form, generates a computational methodology to engage the otherwise 'happy accident' and gives over the form making to the computation of events in space. This methodology employs systems to control the condition and context of an event but allows form to be driven from the event. This performative approach to material form-making, driven from events and forces, offers a further dimension in the direction of computational design strategies. Responsive components and building interfaces may relate the environmental data to reflect the potential of structures changing qualities or transforming uses, allowing for a greater study of effectiveness utilization of spaces to enrich urban spaces.

Limits of Material in Computation

The idea that our world is linked materially in a system of relations is well discussed in Manuel de Landa's book One Thousand Years of Non-Linear History. In computation the effects that drive the forming of architecture is the understanding of data to inform human scale, movement and experience and emotional responses to define an architecture of experience.

The potential of architectural space to be derived through computation techniques through the arrangement and organization of material, offers an opportunity for new language of space to arise. Often though new forms continue to emerge, but are all too close to the original typology from rejecting the influence of material (De Landa). The material computation drives the principles of making and understanding the responses and the structural diagram. Material computation has long been used in engineering and mathematics as models of behavior. Hooke[1] in 1675, used the hanging chain model to decipher the formula for the curvature of a catenary and was used on the unique triple dome structure of St Paul's Cathedral, London and of course Antoni Gaudì with his lengths of strings and lead weights, famously set out the agenda for understanding the way a form will perform in space. Their interpretation of form then translated to the material form. That translation clearly defined by the interaction of material in space. Architecture was once dependent on the understanding of performative design through material to define form.

1. Robert Hooke's hanging chain. Robert Hooke (1635-1703) described the relationship between a hanging chain, which forms a catenary in tension under its own weight, and an arch, which stands in compression.

Form as Unknown / AA Rome Visiting School Workshops
Computation and Making

The premise of the research of the AA Rome Visiting School Workshops was the investigation of computational form through material and performance. Basing the program on Musmeci's definition of form as unknown was part of a larger discussion that outlines that actual form as part of a development through behavior. Indeed computational design strategies are applied not just to structures, but to the order and organization of architecture through the understanding of material and behavior. In the study example of the Basento River Viaduct, in Potenza, the design was envisaged as the result of a stressed skin stretching between support points and the bridge deck. The undulating surface creates a unique spatial quality in the bridge that could be seen as a secondary route. The material exploration that developed this concept was of an elastic surface stretching between points to the degree of force to realize this form. Each of the anchors and fix points represents the force through the surface to find the equilibrium in the form. The final form is translated to a compressive system when it is constructed in concrete. The geometry however is similar and follows the material computation of the material stretched in the model. In this computation approach the form is the unknown of an equation where only the conditions and behavior are given.

Material/Immaterial

Computational material informs all aspects of the development of material formation from environment to behavior. Suspending the idea that a specific material has a particular presence to investigate the relative essence of material in the area of affecting geometry. Deleuze wrote of Spinoza in describing the unique formation of the material world as composed of a single substance delineated by essences and affections (Deleuze on Spinoza). Through this simplicity it is conceived as one substance in many forms. Forms are realized through forces and interactions and ruptures through the substance. This represents an analogy of material in computation. Materials respond to forces and are characteristic by their nature 'pushing' back or resistance to those forces. In these investigations it is conceiving of material not as a specific composition, but by its behavior. A behavior of responses that can be decoded or represented computationally in a model. The decoding informs a process of computationally assembling a strategy in code that follows, mimics, or resembles the material responses.

Computation/Parametric Strategies (Unmaking, Making, Synthesis)

The workshop were set out as an investigation of a material phenomena where the particular responses are recorded and analyzed into the specific event in the material property. The process of analysis is formulated around three categories of investigation:

1) Unmaking or decoding the material phenomena,
2) Making assembling a code of responses which mimic the force lead responses
3) Synthesis, the design of parameters shaping a design response.

Unmaking - investigations of behavior exploring the nature of material in form. In order to understand the system of logic that records the force as a form generator. Unmaking refers to the decoding of a material phenomena, a way of transcribing the observable changes of form from forces acting through the material. In the case of Musmeci and the material based experiments relating to the bridge, the decoding is related to the understanding of the minimal surface geometry and the catenary surfaces which mimic the form of the stretched material. The reference material describes a specific geometry through an event. The event in this case is the forces that pull the material into a state of equilibrium negotiating the forces through the surface geometry. The decoding of the geometric response becomes the tool to investigate responses in a developable model geometry that is translatable.

Making - developing a code that mimics the forces and begins to identify specific form generating systems that will affect a design strategy. Making refers to the computation or parametric strategy in design for understanding the assessment and analysis, and making a proposition based on the observable quality to guide or inform the design process. Investigating the information available to make a proposal defining the constraints of the form generating strategy. In this sense it is interesting to draw a comparison to the mimetic strategies through observation that informed Da Vinci's research in physical phenomena, guiding the design of machines and devices, learned from the environment, observation of forces and defining forms. The design of machines were based on the observation and decoding of the affecting elements. The design proposal utilizes the coding strategy interpreted from the observable elements of the dynamic of form to manipulate form and derive design solutions.

Synthesis - is a translation of the observed to form the parametric and performative components of the design strategy. In shaping the design through information, forces become active elements informing design decisions in a synthesis of form. The design agenda becomes dynamic and responsive to data forces and the potential of shaping of a new architecture. This system is an approach that may employ the same tools that are traditionally the hold of engineers to verify a static design, to one that is used to inform. A system that gives architects and form makers the

power to make architecture that performs and is responsive and is founded on the dynamics of form making. Dynamic analysis informs design strategy where the understanding of forces works interactively to bring the form making potential to designers to inform the making of architecture, and generate a more responsive, efficient, and intelligent building culture.

A Responsive Agenda for Architecture/Urban Form

Recent trends in Architecture have moved away from the understanding of form and performance and the connection to material in design and making. The preoccupation of architects to reference and image design strategy has pushed the study of architecture to the realm of ideological false hood that is removed from material and materiality. An image is not form and is removed from the material sources that inform the design decision. Materiality has to be more than a reference for a mapping code to produce renders in hyper reality. Architects must move beyond the image making and engaging with materiality and the informing of material from our data rich world if it is to be relevant in the world that is reshaping at the rate it is currently. Architecture must respond to the development of technology that informs, and design must respond and be dynamically engaged with the information that shapes our world. In a world of increased urbanization at unprecedented rates, cities will increase at levels unlike ever before. Over the next half century, the world will add approximately one new city of a million every 5 days (Fragkias).

Dynamic responsive technique informing design where forces are informing the shape of our built environment and playing a more active role in the shaping of cities and urban form, offers the potential to design a more appropriate level of urbanization that is informed in design from behavioral and environmental data making changes to form through generated responses. Material changes and perception of spaces and development of new form that is responsive to dynamic forces and social forces that shape our material cities and environment. Not simply buildings but entire cities giving a design agenda that architects must employ a responsive technique to design methods to inform the design and to build responsibly to meet the challenges of the changing natural and urban environment. Through the investigation and analysis of behavior, a performative agenda of real time informing material strategies and responsive environments, architects may gain a skill that is perceptibly lost. As technology in parallel industry grows and techniques that are employed in manufacturing that engage robotics and responsive materials, optimization of form, to create change while responding to changing environmental data, the building design industry must learn from these approaches to use techniques that inform the unknown. The investigation of material and materiality is key to the forming of a behavioral responsive architecture as form making and building.

References:

A. Beukers, E. Van Hinte, 1998, *Lightness - The Inevitable Renaissance of Minimum Energy Structures*, 010 publishers, Rotterdam.

M. De Landa, 1997, *1000 Years of Nonlinear History*, Zone Books, New York.

P. Schumacher, 2010, *The Autopoiesis of Architecture*, vol. 1, John Wiley & Sons Ltd., London.

G. Deleuze, 1990, *Expressionism in Philosophy: Spinoza, trans*, Martin Joughin, Zone Books.

M. Fragkias, 2012, *The Rise and Rise of Urban Expansion*, IGBP Global Change Magazine Issue 78, March 2012.

Web references:

P. Block, M. DeJong, J. Ochsendorf, As Hangs the Flexible Line: Equilibrium of Masonry Arches, http://web.mit.edu/masonry/papers/block_dejong_ochs_NNJ.pdf
Building Technology Program MIT, Cambridge MA 02139 USA.

Lawrence Friesen studied Architecture and Environmental design at Dalhousie University, Canada, and worked at a number of architectural practices before beginning the Design Geometry modeling group at Buro Happold Engineers in London. In the past 15 years he has been involved in a number of complex projects whose innovative realization entailed digital method of building and fabrication. At the GenGeo, architectural structures are designed, optimized structurally and realized through a digital design process to engage with simulation software. Forms are translated into a material design process that is true to the intent of the design but have a spatial and material that enhances the design realization.

Formerly he was a studio technical tutor at the AA graduate masters program Design Research Lab (DRL) and technical tutor to AA intermediate Unit 10. Currently teaching at the AA Visiting Schools in Rome, Italy, and is Associate Lecturer Unit lead at Oxford Brookes University, he is also a practice principle for GenGeo consulting for architects and designers and is a design lead for design projects.

Lorenzo Vianello graduated in architecture in Italy in 2005. From 2005 to 2009 and from 2011 to 2013 he collaborated with several firms, including OMA, Studio Fuksas, UNStudio and Foster+Partners. From 2009 to 2011 he deepened his research, attending the Design Research Laboratory program at the Architectural Association. Here he was guided by tutor Patrik Schumacher, partner at Zaha Hadid Architects, during the development of his final thesis. The theme of the thesis was a proto-tower, system of algorithms that generates projects of high-rises with optimized properties for different environments and functional programs. Lorenzo currently is based in London and works as a Design Tutor at the Oxford Brookes University, as an AA Rome Visiting School program co-director at the Architectural Association and as a freelance architect.

The Protohouse project was developed by Softkill Design in the Architectural Association School's Design Research Lab within the 'behavioral matter' studio of Robert Stuart-Smith. Image courtesy of Softkill Design. http://www.softkilldesign.com/.

10_evolutive structures

topology optimization

"I don't want to undress architecture. I want to enrich it and add layers to it. Basically like in a Gothic cathedral, where the ornament and the structure form an alliance".

Cecil Balmond

Contemporary architecture practices have enabled a clear distinction between the role of the architect and the role of the engineer. In this paradigm, the architect is concerned with the buildings shape and functionality while the engineer is concerned with mathematical "firmitas." Technological developments and contemporaneous architectonic research have attempted to reintegrate the two fields by nesting shape and statics, as well as creativity and structural calculation.

The study of *optimization* integrates design and structural decisions, to create optimal solutions within set parameters. These solutions are often structural: and aim to reduce the amount material, span further, and leverage structural latencies. Optimization can also be applied to building program, function and form.

The two leading domains of optimization are: shape optimization and topological optimization. Both forms of optimization attempt to reach an optimal solution with respect to a set of parameters that define a fitness function.

Both approaches share the same procedural logic:

- An initial geometry is defined;
- Boundary conditions are defined that control the optimization;
- Definition of one or more fitness functions.

Even though the fundamental process is the same for both of these techniques, there are conceptual and algorithmic differences.

10.1 Shape Optimization

Shape optimization is the process of calculating a geometry that minimizes the assigned fitness function within the boundary conditions.
The elements of a geometric optimization are:

- Fitness function, that is commonly defined as a minimum research problem;
- Variable definition;
- Supports setting;
- Definition of a domain for the elaboration of a solution.

The first three elements are decisions that set the desired outcome path for the optimization problem. The optimization process can be summarized in 5 steps as illustrated below. Once the analysis is complete and the results have been evaluated, the algorithm seeks for an optimal solution. The result of this process will be a modified initial geometry that maintains the initial topology.

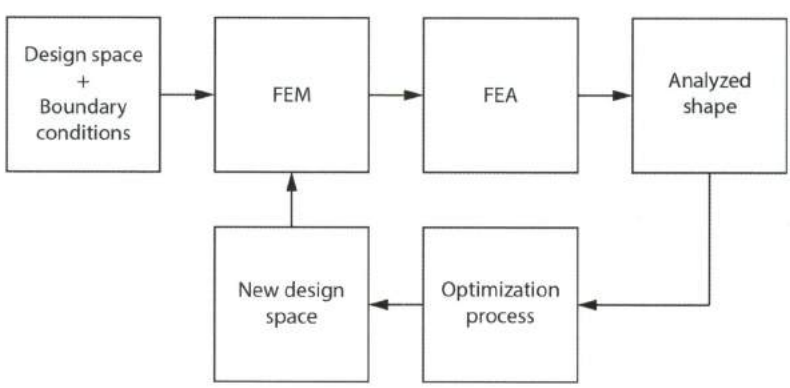


FIGURE 10.1
Shape optimization flowchart.

For example, this approach can be used to return an optimized cross section within the define domain of a cantilever.

The cantilever is composed by one fixed extreme named as A and one free extreme named as B. The boundary conditions are:

- Fixed connection at point A, no restraints at B;
- Point load at point B;
- Defined material properties and a list of possible cross sections: UNI5397-78 and UNI5398-78 standards.

The algorithm seeks to minimize the displacement of point (B). The calculations are performed by a genetic solver that analyzes all the possible solutions and discards solutions that do not efficiently meet the demands of the fitness function. Solvers will be further discussed in section 10.6.

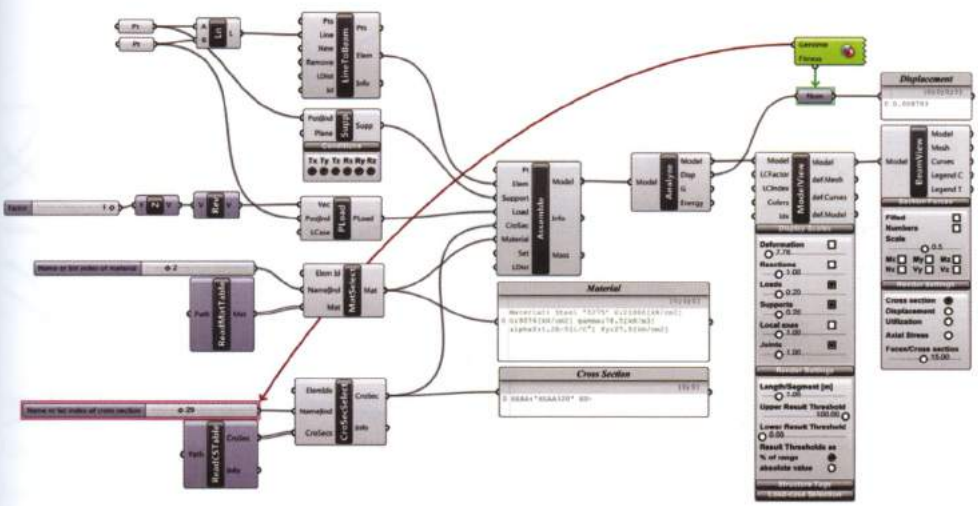


FIGURE 10.2

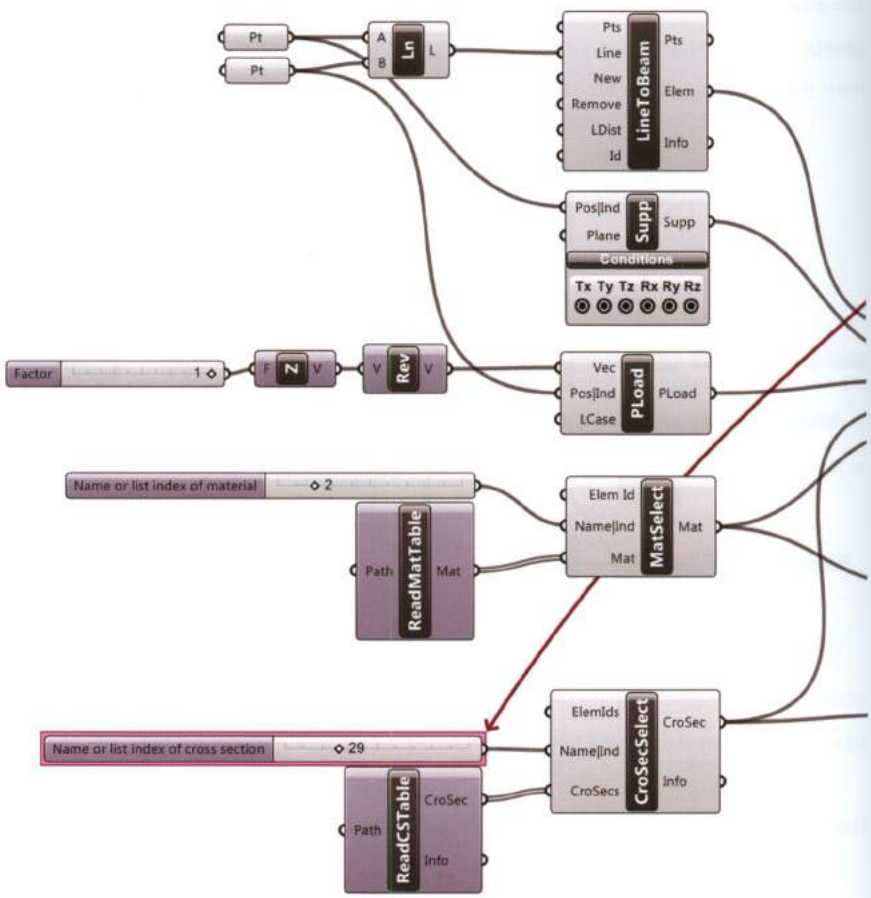
Algorithm definition of the shape optimization.

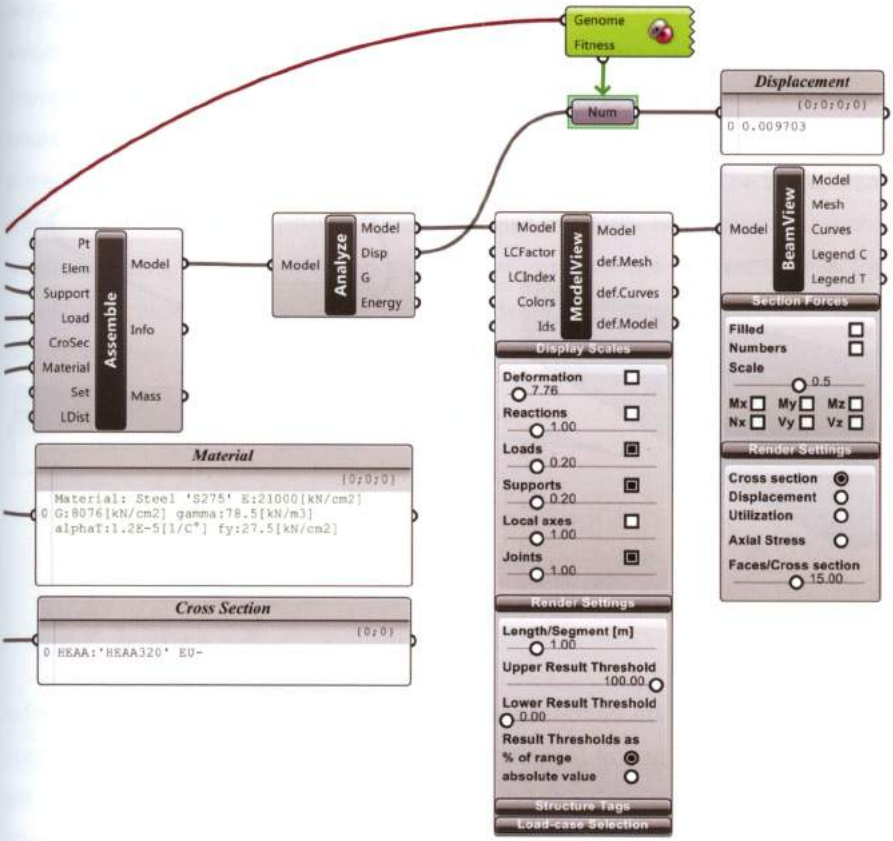
The definition illustrated in figure 10.2 can be read as a sequence of four steps:

1. Modeling of the static scheme and the definition of the boundary conditions;
2. Elaboration of the finite element model;
3. Model analysis;
4. Visualization and analysis of the results.

After the analysis is completed and the results are evaluated by the designer, the optimization process can start defining an additional step.

The first step in the structural optimization definition is constructed as follows. Two points set from Rhino are used to define a line in Grasshopper. The line defines the longitudinal axis of the beam on which the analysis is performed The end points (A) and (B) are respectively used to define the location of fixed supports and a concentrated load described by a vector in the Unit Z direction with a magnitude set by a slider. Finally, the material properties and the list of cross sections to analyze are set.





The material and cross section properties are set using the *Karamba*[28] material and cross section libraries respectively. In this instance S275 steel and IPE and HE sections are selected. The list of cross sections represents the population of solutions in which the genetic solver will look for the best one. Each cross section is analyzed under the described boundary conditions using finite element analysis; the section that allows the minimum displacement at point (B) is the optimal solution from the list. Once the process is completed it is possible to visualize all the solutions and the gradual selection of the optimum cross section.

NOTE 28

Karamba is a Grasshopper plug-in used to analyze structures such as: spatial trusses, frames or shell structures. It is developed by Clemens Presinger in cooperation with Bollinger-Grohmann-Schneider ZT GmbH Vienna. For further information visit www.karamba3d.com.

It is necessary to note that the solver selects the optimum section from a short list of possible solutions in a very well known problem; experience demonstrates that, dealing with complex problems, often the solution provided by the solver is not "the best one" but is closer to it.

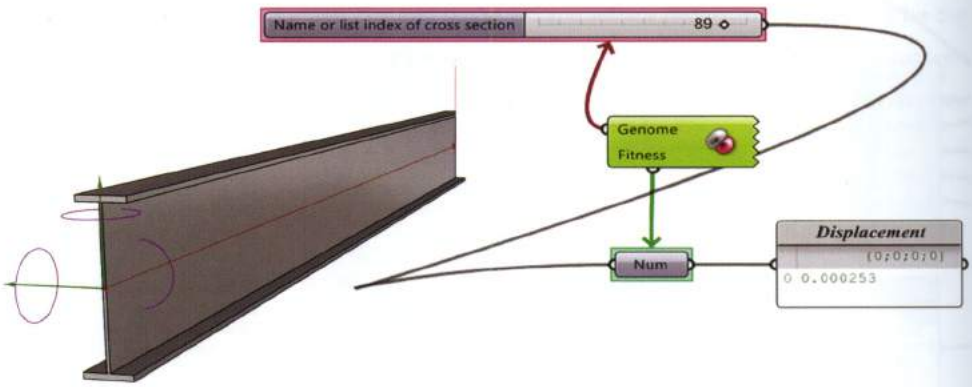


FIGURE 10.3
Optimized cross section.

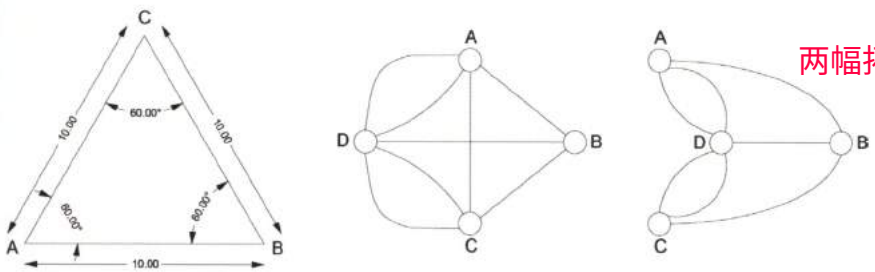
The optimization calculations are performed by Galapagos (see 10.6), a genetic solver within Grasshopper. Research in the field of computational generative algorithms started in the 1960's with the publication "On the organization of the intellect"[29] published by Lawrence G. Fogel. The popularity of evolutionary calculation in the IT world came with the book "The Blind Watchmaker"[30] written by Richard Dawkins and the following applications have been made possible thanks to the improvement in computers.

NOTE 29
L.G. Fogel, "On the Organization of Intellect", PhD Thesis, UCLA, 1964.
NOTE 30
R. Dawkins, *The Blind Watchmaker* (New York: W. W. Norton & Company,1986).

10.2 Topology

Topology is the study of the relationship between geometric parts undergoing deformation.

Unlike plane geometry, topological analysis does not require metrical or angular measurements; instead the study of topology is based on the comparison of figures. The figure below shows a comparison between an equilateral triangle defined metrically and angularly, and the topology graph illustrating the "Konigsberg bridge problem"[31] as theorized by Euler in 1736. Even though the two illustrations are different the two graphs describe the same problem, since the position of objects and their dimensions do not affect the system topology.



两幅拓扑图不同吧?

FIGURE 10.4
Comparison between the classical geometric definition of a shape and the topological approach to a problem using graphs.

The main points of topology are:

- The principle of continuity due to strain, even when losing its metrical and projective characteristics, there will be a biunique correspondence among the points of the original figure and those of the transformed one.
- The principle of homomorphism states that strains, causing penetrations or the overlapping of material are not admissible after the deformation.

Topology principles can be visualized by the transformation of a square. A homomorphism map can be created from the square to the circle; thus it is possible to carry out a transformation leading from one figure to the other without overlapping or cuts.

Homeomorphism between two figures is regulated by a parameter called *genus*. Genus is a topological

NOTE 31
L. Euler, "Solution problematis ad geometriam situs pertinentis", Commentarii academiae scientiarum Petropolitanae 8, 1741, pp. 128-140.

invariant that can be mathematically demonstrated which asserts that two homomophic figures must have the same genus value (g). The (g) value of a figure is equivalent to the number of holes within the geometry or, more exactly, is the greater number of non-intersetting simple closed curves drawable on a surface without splitting the surface itself in two separated parts. From a morphological perspective the presence of the same number of holes in two figures, or a total absence of them, leads to homomorphism. Thus, when the sphere is defined by a genus (g) with a value of 0 and when the torus is defined by a genus (g) with a value of 1, they are not homomorphic and it is impossible for one to become the other and vice versa. So the homomorphism settles the transformation possibilities among figures.

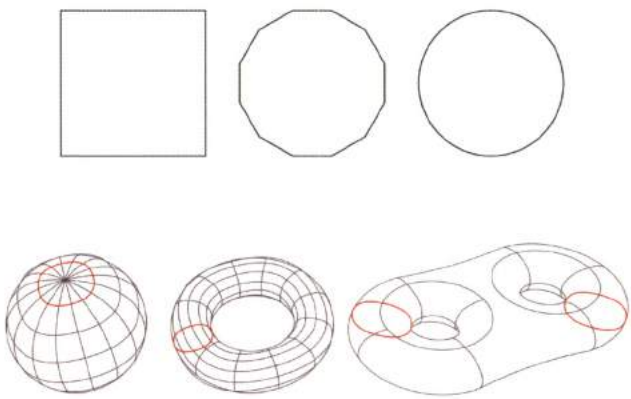


FIGURE 10.5
Representation of continuity and homomorphism principles.

10.3 Topology optimization

Mathematical topology principles are used to study the topological optimization of form. Research in the field of structural optimization has brought teams of engineers, mathematicians and later on, architects, to investigate topological optimization to more efficiently use material.

Topological optimization began conceptually in the 19th century, however, the results were not validated till the end of the 1980's with the improvement of digital calculation. The first functioning algorithm was presented in 1992 by Xie and Steven under the name of Evolutionary Structural Optimization (ESO).

The main characteristics of the algorithm are:

- Possibility to change the structural topology exceeding shape optimization;
- Evolution-based procedure for the development of an optimization process.

The process follows four main steps:

1. Definition of the initial data: geometry, supports, loads and constitutive material;
2. Definition of optimization parameters;
3. Finite Element Analysis (FEA);
4. Structural optimization.

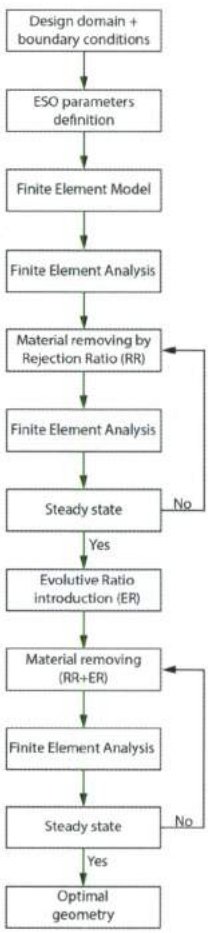


FIGURE 10.6

Evolutionary structural optimization flowchart.

The first step in an ESO process is to set the input data, then allow the ESO to run subtracting material. The initial geometry definition requires a larger design space compared to the final result of the optimization process. Contextually, the position and type of the supports, the loads and the material properties will be set so that the strain in each part of the structure can be calculated by the Finite Element Analysis.

The second step in the ESO process determines the parameters for the optimization process and the discretization of the initial design space into finite elements (FE). Once the structural FE analysis is concluded the distribution of the strains can be visualized within the cross sectional area clearly displaying areas of high and low stress.

These two steps are applied to a simple bi-dimensional example of a beam supported at each end and loaded with a concentrated load applied to the lower edge of the beam[32].

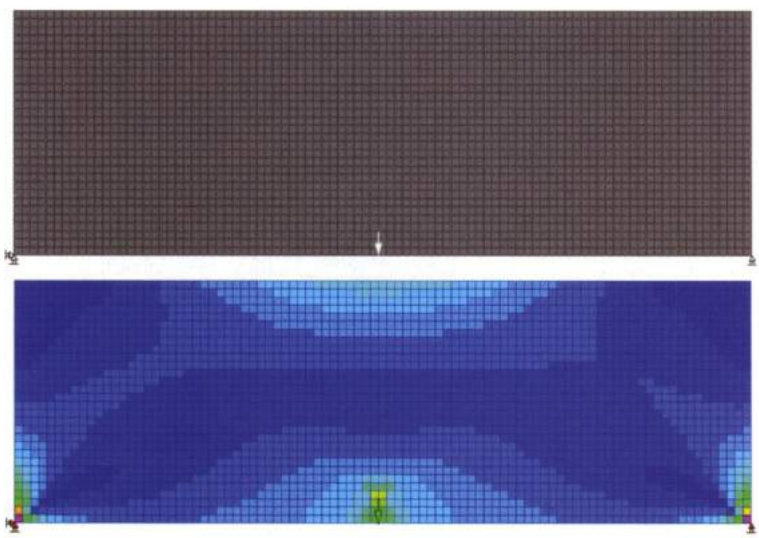


FIGURE 10.7
Finite element model and finite element analysis.

Once the calculation is complete it is then possible to discard inefficient material using Rejection Criterion (RC) defined by the Von Mises stresses, and the Rejection Ratio (RR).

NOTE 32
Example realized using BESO2D, software developed by Prof. Mike Xie and introduced in the book *Evolutionary Topology Optimization of Continuum Structures: Methods and Applications*, Huang, X. and Xie, Y.M. Chichester, England, John Wiley & Sons, Ltd, 2010.

The RR is a percentage value used by the algorithm to identify FE discretized cells were material must be rejected. If the RR is set up to 50%, we must reject all of the finite elements where the RC reaches values lower than the 50% of the maximum admissible value.

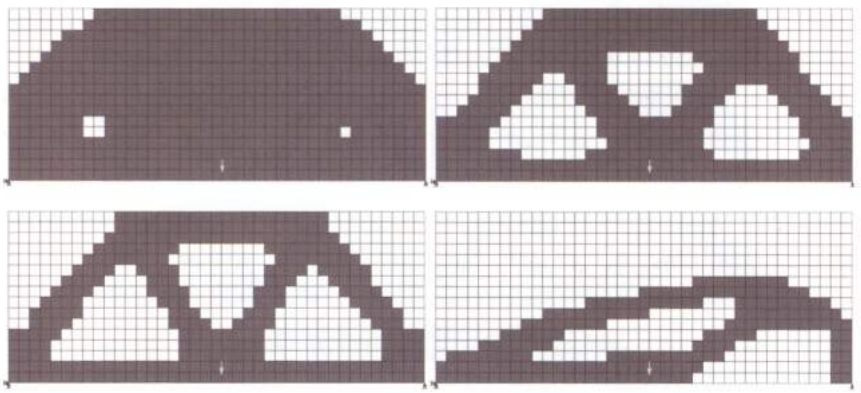


FIGURE 10.8
Different rejection ratios applied on a MBB beam optimization. microbubble beams

The correct calibration of the RR value is essential to the accuracy of the optimization. The above image shows the same optimization example, with different RR values applied.
By examining the results, high RR values (i.e 90%) lead to an extremely limited subtraction of material, Medium RR values (i.e. 60% and 50%) show results that are compatible with the expectations provided by experience, and Low RR values (i.e. 30%) lead to an excessive rejection of material; in this case a non valid static solution is produced.
Once the first step of material removal is concluded and a stable configuration for the beam is achieved the beam is further optimized using an Evolutionary Rate (ER), which is added to the RR. The removal of inefficient material has changed the distribution of the strains inside the structure consequently the stress concentrations inside the material which survived the evolution process have increased. It is desirable to let every element work at the maximum of its mechanical potential by including a parameter to limit the possible tension inside a determined maximum value.

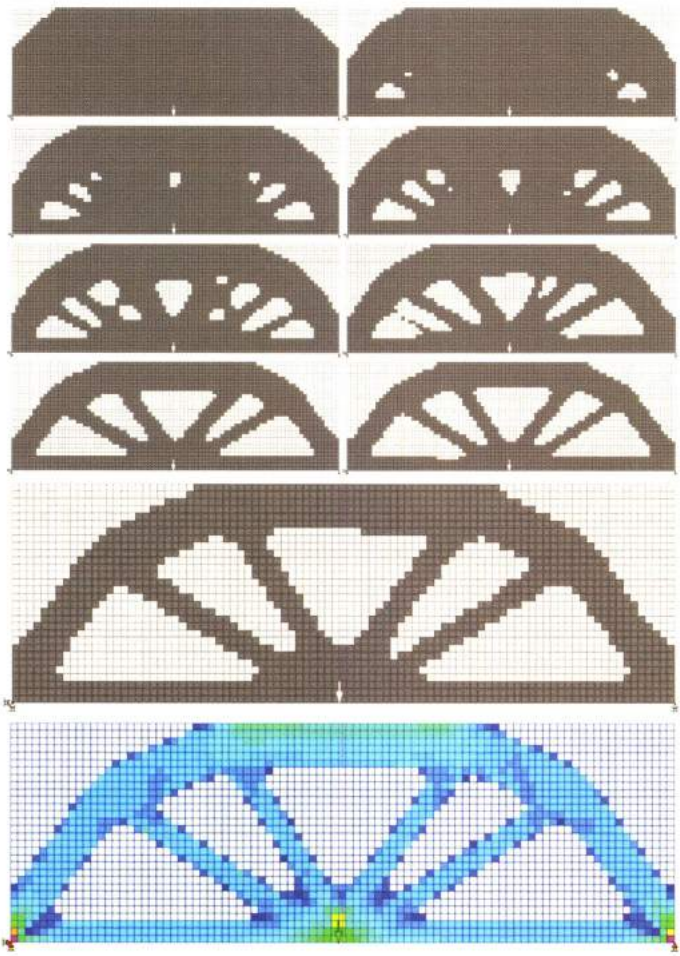


FIGURE 10.9

MBB beam optimization and Finite element analysis of the optimized beam.

Evolutionary Structural Optimization

Extended ESO (XESO), suggested by Sasaki[33], is based on the concept that organisms underwent environmental adaptations by rejecting the unnecessary and strengthening the necessary. The *Banyan Tree* epitomizes this approach. The tree has a set of solid aerial roots that by extending to the ground support the main trunk. This structure is only made of what is functionally necessary without wasting material.

NOTE 33

C. Cui, H. Ohmori and M. Sasaki, "Computational morphogenesis of 3d structures by extended eso method," *Journal of the international assocuation for shell and spatial structures* IASS, 4 (1), (2003): 51–61.

Starting from this example Sasaki expounded that ESO, in its original configuration, is limited by two factors: the mono-directionality of the evolutionary process; which allows the discarding of only inefficient material, and the lack of control of the optimization.

To fix the limits, two innovations were introduced:

- Control strategy of optimization using *contour lines*;
- Bi-directional evolution.

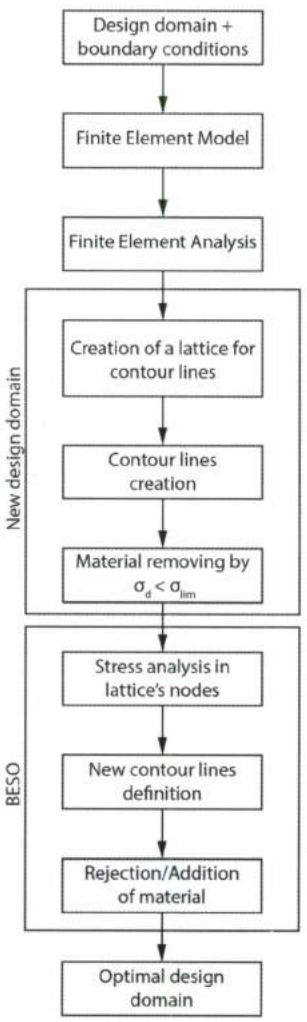


FIGURE 10.10

Extended evolutive structural optimization flowchart.

The XESO traces a set of lines describing the stress status of the object indicating portions of the surfaces with:

a) Lower stresses compared to the limit value;

b) The same stress compared to the limit value;

c) Higher stresses compared to the limit value.

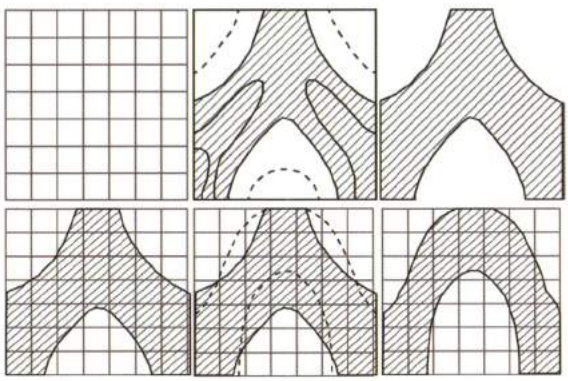


FIGURE 10.11
Contour lines application and material removal scheme.

The XESO approach speeds up the optimization initial steps because the material that exists satisfies the Von Mises condition $\sigma_d < \sigma_{lim}$; material that does not satisfy this condition is discarded. Once the initially optimized design space is set; material is added and removed material and subsequently the stresses are calculated at the overlap between the contour lines and the initial grid. Then, new contour lines are traced to depict where to remove or add material. Contour lines inside the material are equated to material to be removed, while contour lines outside of the material are equated to material to be added.

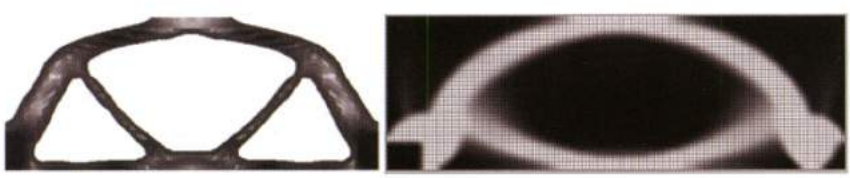


FIGURE 10.12
Comparison between MBB-beam optimized with ESO method and XESO method. Cui, C., Ohmori, H., & Sasaki, M. (2003). Computational morphogenesis of 3d structures by extended eso method. Journal of the international association for shell and spatial structures IASS, 4 (1), 51-61.

10.4 Works

10.4.1 Akutagawa office building

The Akutagawa office building was designed and built by Hiroshi Ohmori in 2005 using the XESO method. The building is within a larger commercial area close to Takatsaki station on a parcel square-shaped 6 x 10m. The XESO method was applied to the northern, southern and western facades, while the eastern facade and the attics were excluded from the optimization process. In the finite element model the specific weight of the structural elements in vertical direction, and the horizontal forces simulating a telluric action were applied. The topology of the three facades evolved as material was removed from the areas undergoing less stress and added in areas of higher stress.

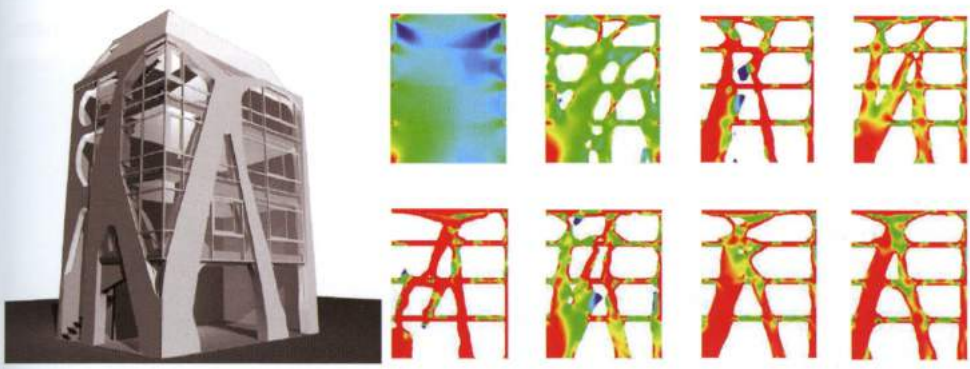


FIGURE 10.13
Akutagawa office building – model and finite element analysis of optimization steps. Ohmori, H. "Computational morphogenesis, Its current state and possibilities for the future" – IASS-IACM 2008 Cornell University, Ithaca, NY, USA 28-31 May 2008.

10.4.2 TAV station in Florence

Arata Isozaki and Mutsuro Sasaki TAV station proposal in Florence used the same algorithm that had been applied to the Akuttagawa building. The building: 400m long, 40m wide and 20 m high, houses the stations program within a single superstructure. During the optimization process, the structure was modified from the initial shape of a bridge with simple frames on the extremities to a more organic shape. The XESO process was illustrated by a set of schemes describing the evolution of the structure. During the development phase Sasaki's team was supported by Buro Happold – an international engineering firm located in London – to define the structural sequence. The structure is composed of steel trunks, pre-stressed beams and steel and concrete slabs. Isozaki, Sasaki and Buro

Happold used the same approach for several competitions until they realized the Qatar National Convention Center.

10.4.3 Qatar Education City

The XESO method used in the TAV station competition was later used to design the Qatar National Convention Center in Doha. In this case, the Japanese architect Arata Isozaki adopted the XESO method to design a structure 25 m long, 30 m wide and 20 m high volume with a roof plane. The main loads in the XESO calculations were imposed by the self weight of the structure.

The project is an archetype of maximum efficiency and the minimum use of material. The design process was a synthesis between architects, engineers and machines. Relying on the Florence TAV experience, the designers began with a more refined model to speed up the entire process. This choice allowed a considerable reduction in time-schedules concerning the analysis and optimization phase.

FIGURE 10.14
The Qatar National Convention Center. Image by UNCTAD.

10.4.4 Radiolaria

Equally imperative as the application of XESO optimization method is the necessity to define an evolutionary building technology.

The process of building optimized structures, have stretched the conventions of construction. For instance, to build the Akutagawa Office Building molds were taken from the shipbuilding industry. For the Qatar Education City, Buro Happold used an elaborate system of prefabricated modules that were built in situ nullifying Sasaki and Isazaki's flux structure idea.

Currently the evolutionary building technology 3D printing is being investigated. For example the work of Architect Andrea Morgante was realized by Engineer Enrico Dini using the D-Shape printer. The project was based on a micro-organism, the Radiolaria, and was printed monolithically without formwork directly from the static approved CAD model.

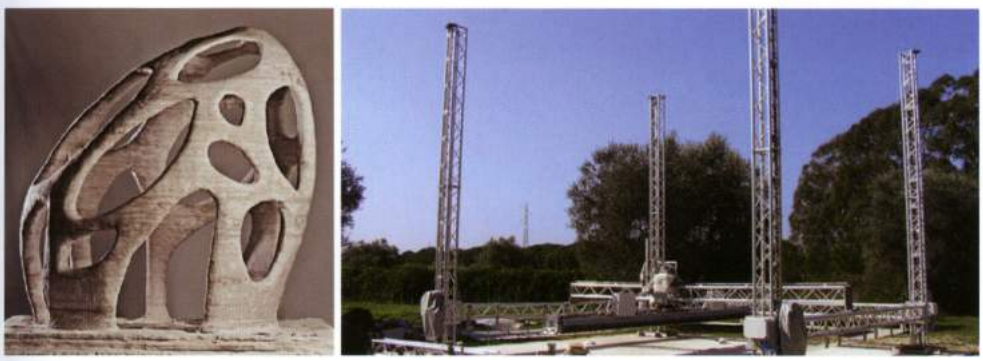


FIGURE 10.15
Radiolaria printed model and picture of the D-Shape3D printer. Image courtesy of D-Shape.

10.5 Examples

In 1904 Anthony Michell[34] economized structure by topological optimizing a beam to support itself and an applied load while minimizing weight and material. The Michell Truss consists of a planar beam designed to support a single load placed on one extremity having, on the opposite extremity, a circular support area.

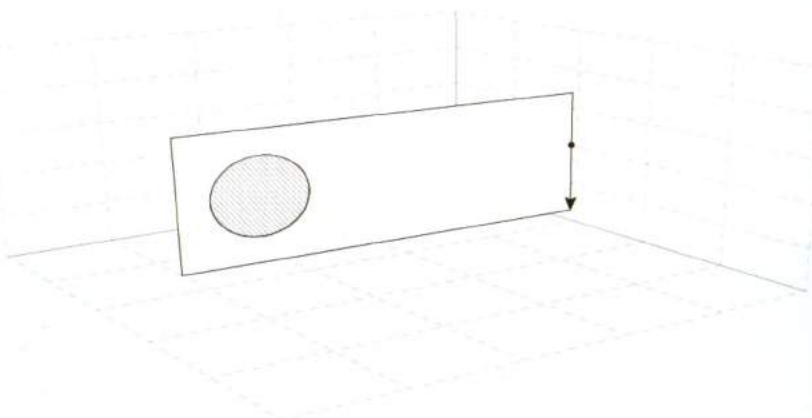


FIGURE 10.16
Michell truss 3D model.

Michell Truss problem can be calculated in Millipede[35] using the components for 2D structural optimization.

The workflow is composed of four steps:

1. Definition of the design space and of the boundary conditions: initial geometry, supports and loads;
2. Creation of the finite element model;
3. Topological optimization;
4. Visualization and analysis of the results.

NOTE 34
A.G.M Michell, "The limits of economy of material in frame-structures," *Philosophical Magazine*, Vol. 8 (47), (1904): 589–597.
NOTE 35
Millipede is a Grasshopper plug-in developed by Kaijima Sawako and Michalatos Panagiotis that allows FEM analysis, form finding and topology optimization (www.sawapan.eu)

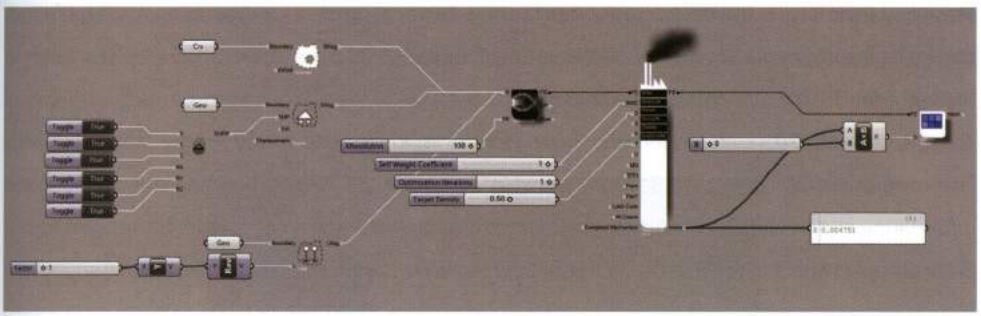
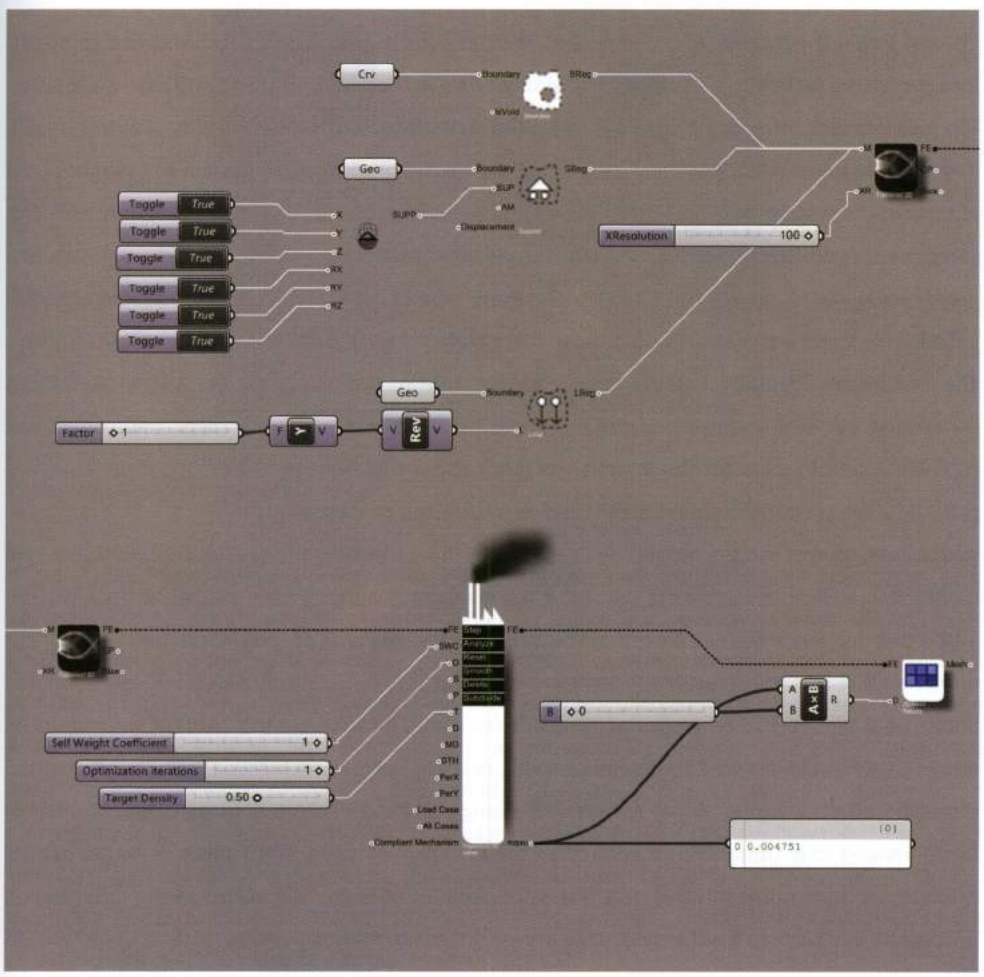


FIGURE 10.17
Algorithmic definition of the Michell truss.



The initial geometry is the design space where the optimal geometric configuration must be found, and in the examples is a rectangle realized in Grasshopper or set from Rhinoceros, which represents the input for the component *2DBoundaryRegion*.

The components *2DSupportRegion* and *Stock Support Type* are used to model the static supports. The component *2DSupportRegion* requires input geometry ideally representing the support and information concerning fixed displacements and rotations. The region that must be supported is a circle placed inside the beam at the center of the median axis of the rectangle. The type of support in this case is fixed so displacements and rotations are fixed in each direction. Restraints can be released or engaged by introducing a *Boolean toggle* to each of the *StockSupportType* component inputs, and toggling True and False respectively. Since this is a bi-dimensional problem it is unnecessary to support the displacements in the z axis and the rotations in the x and y axis.

The last step is the creation of a load region. The component *2DBoundaryLoad* requires a geometric input which is a rectangle placed at the extremity acting in the Unit Y direction with a negative sense. The load intensity has no relationship with the result of the topological optimization. For this reason, it is possible to set a fictitious load intensity of L=-1N/mq or to rely only on the self weight of the structure. The system is assembled by the component which creates the finite element model. This component Topostruct2DModel also has a second input that determines the model resolution. The preset value XR=12, proves to be too low for every type of application, and it is recommended to increase the value with the consideration that the higher the value the longer the evaluation time.

The component *Topostruct2DSolver* represents the core of the plug in and is where calculations concerning the FEM analysis and the topological optimization are performed. This component contains a set of inputs for the insertion of data, and unlike classical Grasshopper components, provides a set of internal tools that can be directly selected by the designer.

In this case, four of the ten inputs are used: the FE-input, receives the previously elaborated FEM model. It is advisable to connect this input last because it automatically starts the analysis. The SWC-input deals with the self weight of the structure and requires an integer numeric value; if the numeric value is 0 the structures weight will be none, if it is 1 the self weight will be computed once. The O-input sets the number of iterations calculated by solver; in this case the value must be numeric and can be entered by a number slider, practice suggests to follow a O=1 value to control the entire process, higher values will require a longer calculation time because of the number of iterations and may cause loss of control on the evolution process. The T-input or target density is expressed by a numeric value and states the amount of material involved in the optimization process (values close to 0 will provide an extreme discard of material, values close to 1 will result in a inadequate decrease of material).

Once the FE-input has been connected, the FE analysis will automatically start. The direction of the topological optimization process is controlled by the STEP tool embedded in the solver, which starts a single iteration. Further iterations will produce an optimal configuration.

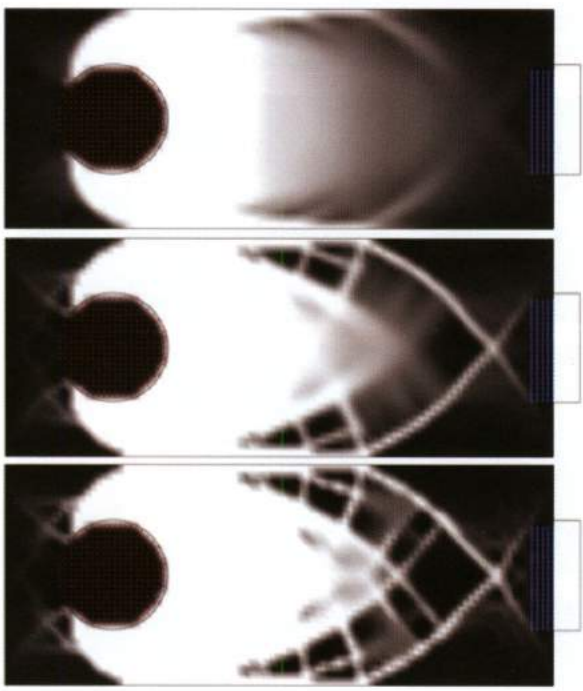


FIGURE 10.18
Iterations of the optimization process.

It is possible to visualize real time results by linking the solver with a visualization component *2DMeshResult* to graphically illustrate information about the stresses inside the member (VonMisesStresses), the flux of the main stresses (Principal tension) and displacements (Deflection). The *Topostruct2Dsolver* provides two outputs: the FE-output, used to visualize the results and the maxu-output that retrieves the maximum displacement value. The last one can be used to visualize the physical deformation of the object. Generally the maxu-value is so low, compared with the dimensions of the analyzed object, that requires an increasing factor to show how the deformation works. In order to boost the magnitude of this value is sufficient to multiply it by an integer number through a *Multiplication* component. Note that this enhancement affects the visualization of the deformed object and not the real displacement.

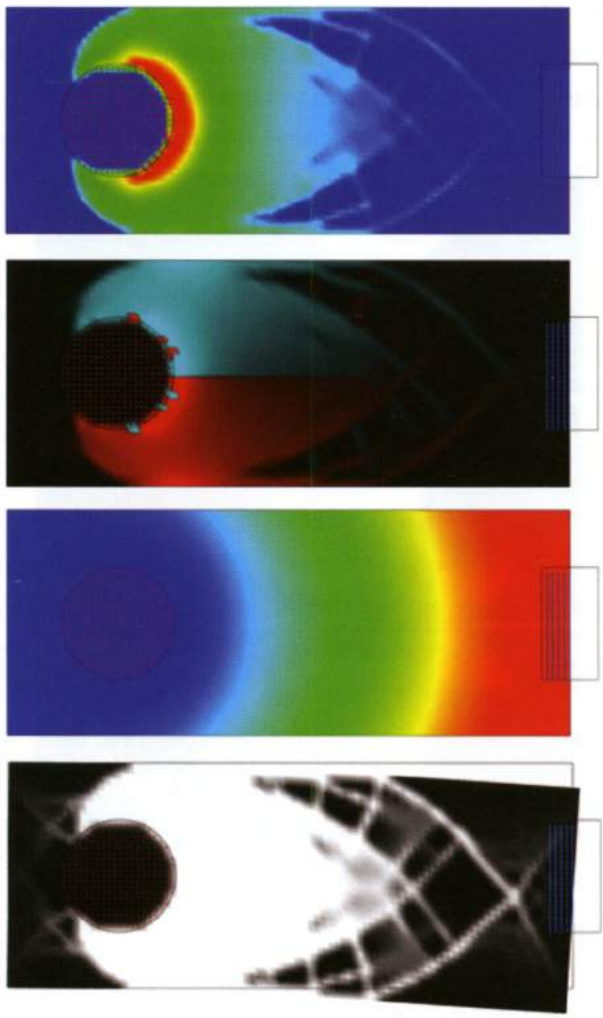


FIGURE 10.19
Michell truss's Von mises stress, principal strain and deflections.

Another common example of bi-dimensional topology optimization is the MBB-beam under self weight.

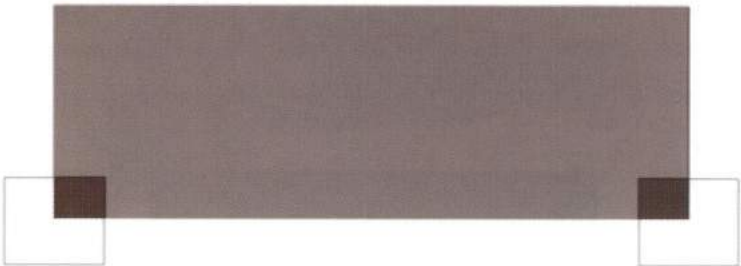


FIGURE 10.20
MBB beam scheme.

The initial model is elementary; a rectangular beam with two supports without any external applied loads.

The optimization strategy is the same as in the previous case except for two differences: the absence of agent loads, and the addition of material self weight from the reinforced concrete; during the creation of the FE model. The definition is as follows:

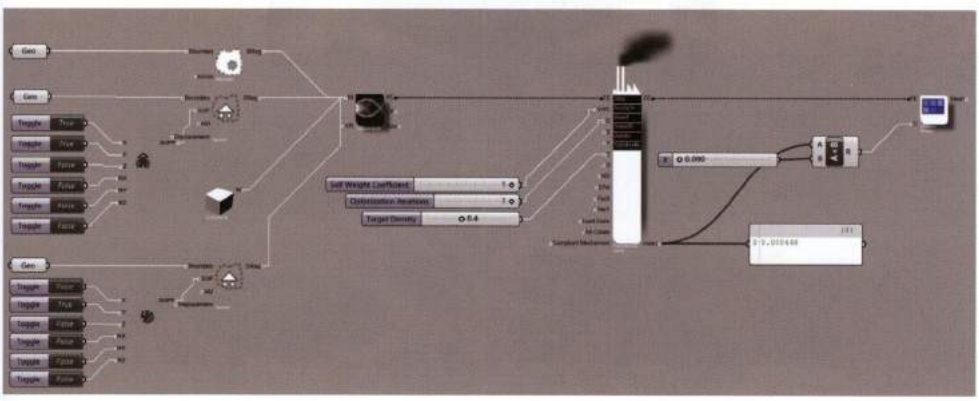


FIGURE 10.21
MBB beam algorithmic definition. The following QR code opens an enlarged image of the algorithm.

In this case there are two regions at each support. The first resists displacements in x and y direction while in the second resists displacements only in the y direction.

Material properties can be set from the *Stock* section in Millipede. In this case reinforced concrete has been chosen.

5

The remaining part of the definition is the same as the Michell Truss example. The solver calculates the maximum value of the structural displacement. This numeric value, multiplied to increase intensity, can be used to visualize the deformed configuration of the object.

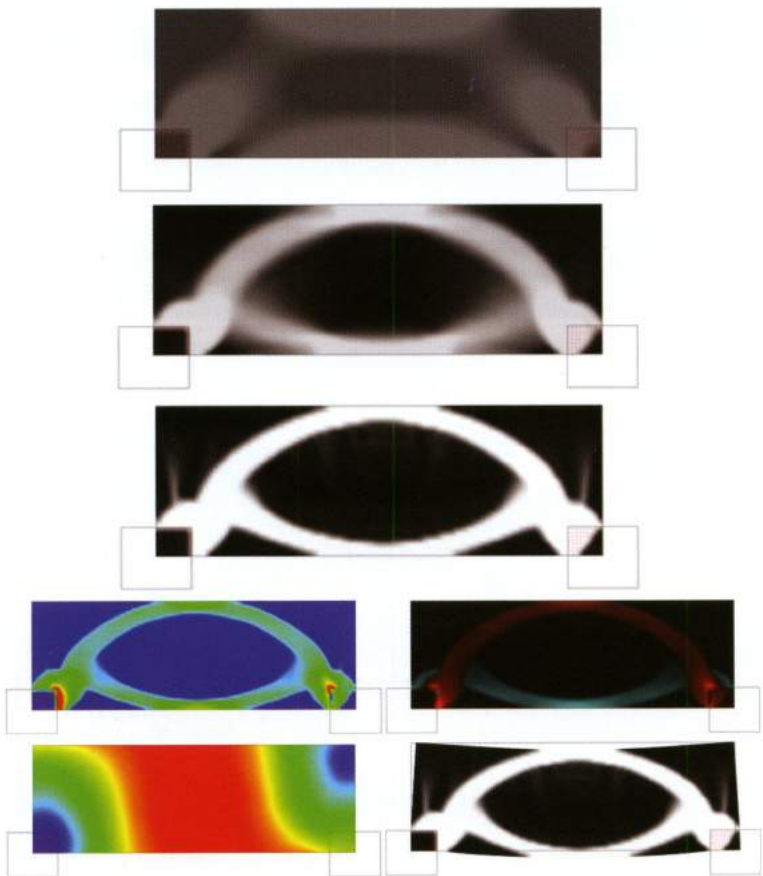


FIGURE 10.22
MBB beam optimization process and images of strain and deformation behavior of the structure.

The third application investigates topological optimization applied to a three- dimensional bridge structure. The schematic model is a tri-dimensional rectangular volume with two supports and without any external applied loads.

The optimization strategy is the same as in the previous case, there are no agent loads, and the material self weight from the reinforced concrete will be considered during the creation of the FE model. However in this instance, the design space and the supports are three-dimensional.

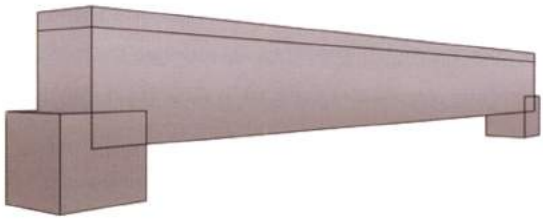


FIGURE 10.23
Bridge scheme.

Evolution optimization relies on the known data of a starting point to generate a variable output. In this case it is necessary to maintain the framework so a further geometric element is required to identify the volume which will not undergo the evolution process. The element that will not undergo optimization is a parallelepiped having the same length of the former beam and a arbitrary height. The parallelepiped will be inset within the design space framework. This new element will represent the input data (3DDensityRegion), a region were everything inside of it does not undergo optimization (non-design space).

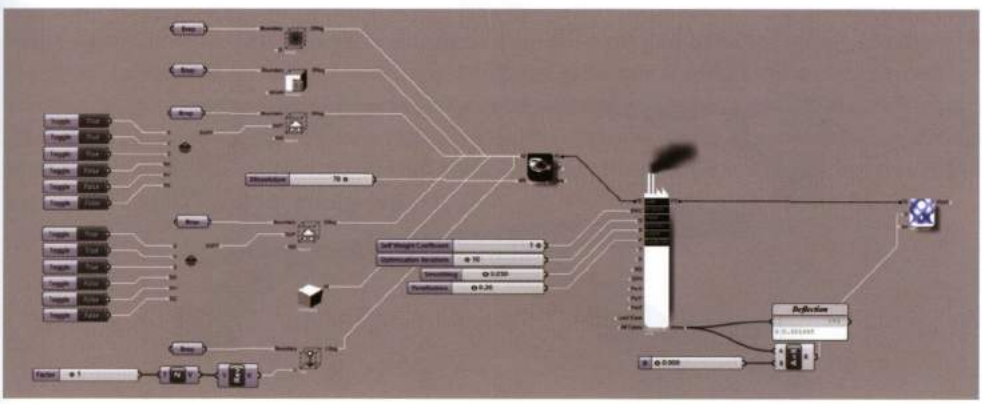


FIGURE 10.24
Algorithmic definition for the bridge optimization. The QR code opens an enlarged image of the algorithm.

6

All the components used in this definition are the exact three-dimensional transposition of those described in the two previous examples. The inputs that define the initial conditions, design space, non-design space, supports and constitutive material, are used to create the FEM and thus connected to the solver.

At the end of the analysis and at the end of the evolution optimization, the results can be visualized by using the component *3DisoMesh*, which: renders the geometry, displays the internal stresses and the maximum displacements of the structure.

By observing the results of the Extended Evolution Structural Optimization (XESO), on which Millipede is based, the behavior can be understood.

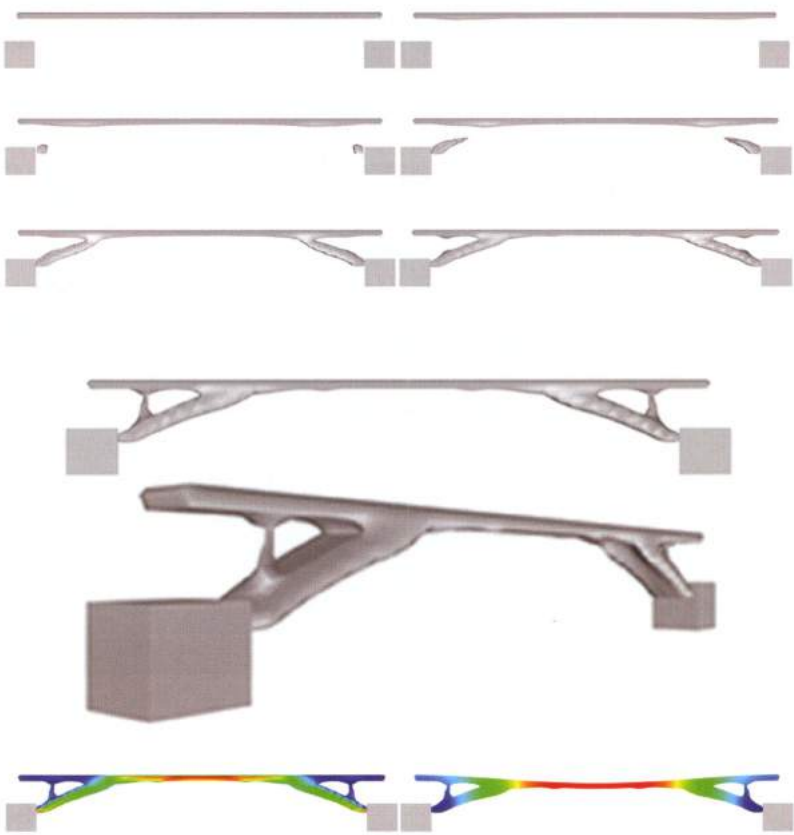


FIGURE 10.25
Optimization iterations and stress and deflections visualization.

Starting from a framework which remains unaltered during the optimization process, the XESO does not only remove the material with low stress levels, it also adds material where required. For instance material is added at the restraints, and at the connection between the extremity and the framework.

The examples analyzed represent a first approach to topological optimization. The simplicity by which it is possible to obtain valid architectonic results is not equated to the mathematical complexity required to calculate a solution. It is highly recommended to face the problem under a theoretical point of view and be able to manage the variables during the calculations phase.

The evolution algorithms can also lead to improvements in other applications. Since XESO is versatile it is possible to use the technique not only for research on structural optimization but also for solutions to other problems that can benefit from constant improvement algorithms.

This chapter is part of the Ph.D. thesis developed by Davide Lombardi at School of Advanced Studies "Gabriele d'Annunzio" (Pescara, Italy) under the supervision of prof. arch. Livio Sacchi, teacher at Università degli studi "Gabriele d'Annunzio" and prof. arch. Alberto Pugnale, lecturer at University of Melbourne (Australia) about evolutive optimization techniques in architecture.

Davide Lombardi (1985) architect, graduated in Architecture (Master of Sciences) with honors at Università degli Studi "Gabriele d'Annunzio" Chieti-Pescara. In 2012 started his PhD course at School of Advanced Studies "Gabriele d'Annunzio" and he is currently developing his thesis under the supervision of Alberto Pugnale, lecturer in architectural design at University of Melbourne. His interest in the field of computational design, focused on structural design and optimization/analysis, led him to start a personal training following high level workshops with international trainers as the AA Rome Visiting School. He taught at "Dynamic Morphologies" workshop organized in Rome by the Oxford Brookes University. He collaborated with architecture firms in Pescara and Rome working on international projects in Saudi Arabia and Ethiopia.

10.6 Optimization: finding solutions with Grasshopper

Optimization solvers enable users to solve real-world problems by finding the best solution among feasible alternatives. There are many optimization solvers available. Fundamentally, users must recognize the appropriate solver for a particular problem. Solvers can be grouped into two categories:

1. **Exact solvers**: find the *best* solution and yield the same result each time;
2. **Heuristic solvers**: find an approximate solution when no exact solution can be found. The resulting output can vary from one solution to the next.

The first category includes *linear solvers*, and the second category includes *evolutionary solvers*. Evolutionary solvers apply the principles of natural evolution to the problem of finding an optimal solution. This section will introduce two Grasshopper optimization solvers: *Galapagos* and *Goat*. *Galapagos* is a built-in Grasshopper solver developed by David Rutten and *Goat* is a plug-in for Grasshopper developed by Simon Flöry[36]. The following text will discuss how to use these solvers as a tool for optimization from a designer's point of view.

10.6.1 Optimization problems

A problem can be described by a model composed of two elements:

1. An *objective function* that we want to *minimize* or *maximize*;
2. A set of *variables* which affect the value of the objective function. Variables are usually defined by a range, with a start and end value.

For example, a basic problem is to find a point on a curve that is closest to a given external point A. In this case the *objective* is the *d(t)* function that describes all possible distances between the points on the curve and the external point A. The only *variable* is the parameter (t) ranging between 0 and 1. *The optimal solution* for this problem is the value of (t) that *minimizes* the d(t) function.

NOTE 36

Simon Flöry received his PhD in mathematics from Vienna University of Technology in 2010. For many years, he has been developing and consulting software solutions in geometry processing and architectural geometry. His ongoing research focuses on effectively processing and optimizing geometric data for geodesy, medical applications and facade engineering. http://www.rechenraum.com/en/

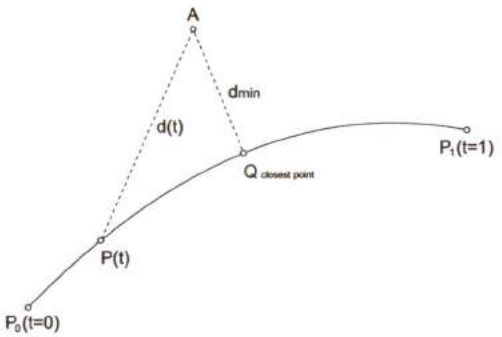
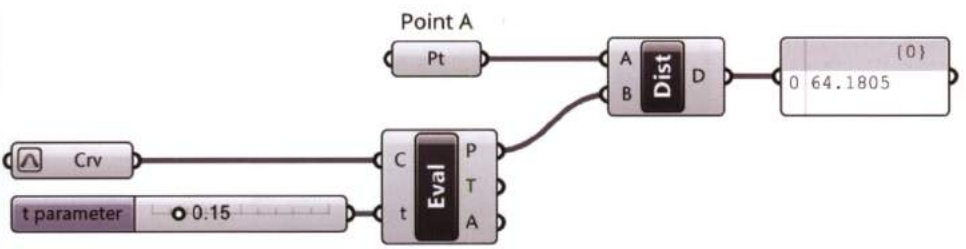


FIGURE 10.26
The objective function d(t) and the parameter (t) ranging between 0 and 1.

10.6.2 Exact solvers. Goat plug-in

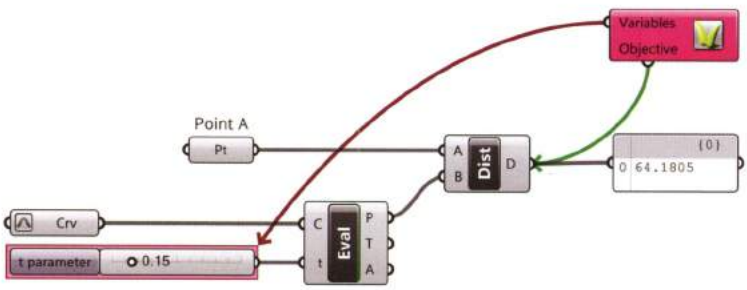
Optimization problems can be translated into Grasshopper definitions; the following sequence calculates the distance between point A and a generic point P on a reparameterized curve. The position of point P with respect to the curve is controlled by the input (t) of the component *Evaluate Curve*.



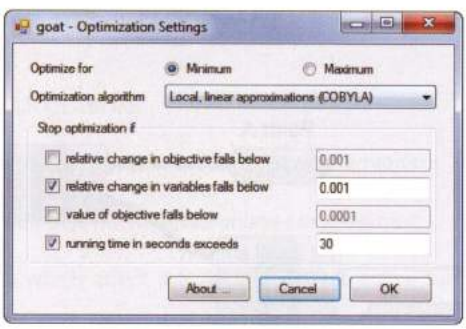
By moving the *slider* we can read the distance value on the *Panel* and we might "manually" try to find the *t* value related to the minimum distance *d*. Of course this method is inconvenient and will not generate an exact value without significant efforts. In order to find the optimal solution the *Goat* solver can be used. After installation, *Goat* can be found in *Special* > *Util*; the component has two slots for outgoing connections instead of the ingoing connections of standard components.

1. The *Variables* slot can maintain one or more *Number Sliders*. In Grasshopper the variables related to the optimization problem are **always Number Sliders**;

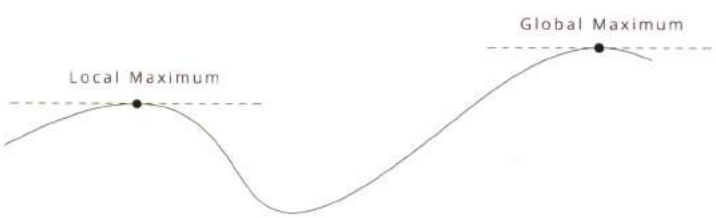
2. The *Objective* slot is always connected to a *numerical output* to minimize or maximize. The *Objective* input has **a single value** connected to the slot.



Double-clicking the *Goat* component opens the contextual settings window which can be used to set several parameters. The first option is used to minimize (*Minimum*) or maximize (*Maximum*) the *objective* function; in the example, the distance component output (D) is optimized for the minimum output.



The second option allows to select among two type of optimization based on *Local* or *Global* algorithms. Roughly speaking, *Local* algorithms attempt to find exact solutions by inspecting values close to the starting value of the variable, while *Global* algorithms inspect all possible values. For very complex problems it is often useful to perform a *Global* algorithm and then a *Local* optimization.

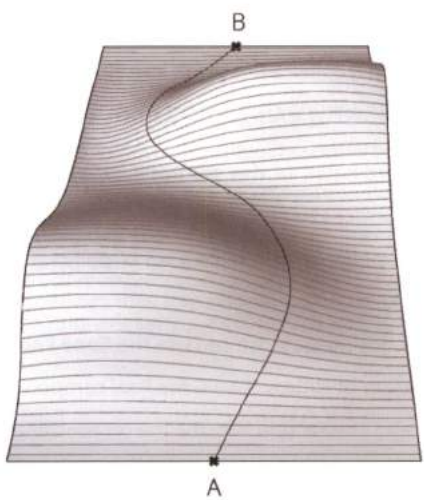


Lastly, clicking *OK* will initiate the optimization procedure. The slider will move guided by *Goat* until the optimum value (minimum distance) is reached. When the procedure ends two values can be read: the (t) parameter on the number slider, and the minimized distance output (D), stored in a panel.

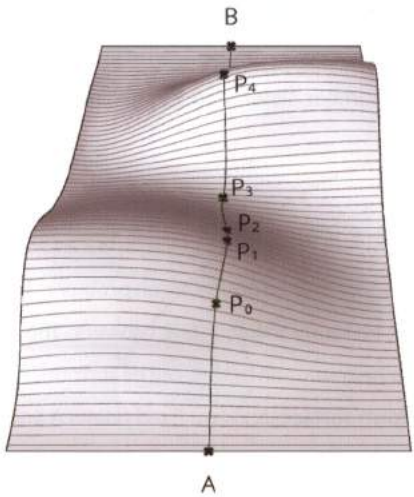
10.6.3 Evolutionary solvers. Galapagos

Heuristic evolutionary solvers are used when optimization problems have a large number of variables and an optimal solution cannot be found through exact solvers. The evolutionary solver embedded in Grasshopper, *Galapagos*, is a heuristic solver that will be introduced in the following example.

Given a reparameterized freeform surface, and two fixed points A and B, located on opposite edges of the surface, the *Galapagos* solver can be used to calculate the shortest path on the surface that connects the two points.

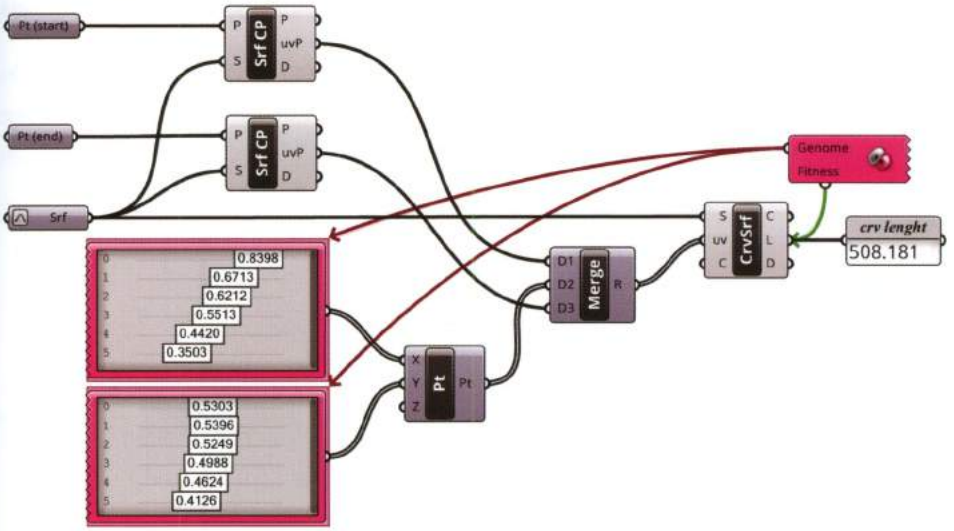


The first step in the shortest distance definition is to describe a *path curve*. The path curve is defined as a *flexible* interpolated curve that is coincident to the surface with fixed ends A and B. The curve geometry is controlled by the *uv* coordinates of a set of interpolation points $P_i (P_0, P_1, P_n)$. By changing the coordinates of the interpolation points, different curve lengths will be output. The heuristic solver is used to find the interpolation points coordinates that correspond to the curve with the minimum length, among all feasible alternatives.



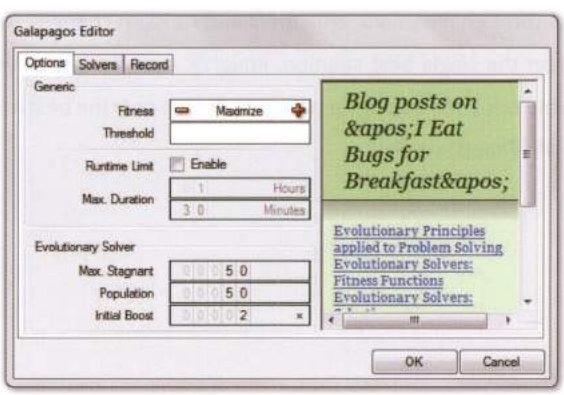
The *path curve* is created using the component *Curve on Surface* (Curve > Spline). The *Curve on Surface* component creates an interpolated curve on a surface input (S), through an arbitrary set of points input (uv) expressed in the LCS. In this instance, the *Curve on Surface* component requires three sets of *uv coordinates*, which are collected by a *Merge* component, whose inputs are:

- D1: are the fixed coordinates of the start-point (A) converted to the Local Coordinate System through the *Surface CP* component (see 3.7.2).
- D2: are the (uv) coordinates of an arbitrary set of points (P_i) on the surface. To define the points the *Construct Point* component is used with coordinates set through sliders ranging between 0 and 1. For this purpose we can use the ***Gene Pool*** (Params > Util), a component that embeds a collection of sliders. Two *Gene Pool* components are used, the first one for the (u) coordinates and the second one for the (v) coordinates. The number of sliders or *Gene Count* as well as the sliders range and significant digits can be set by double-clicking the *Gene Pool* component. The larger the number of sliders, the higher the accuracy of the resulting curve.
- D3: are the fixed coordinates of the end-point (B) converted to the Local Coordinate System through the *Surface CP* component.

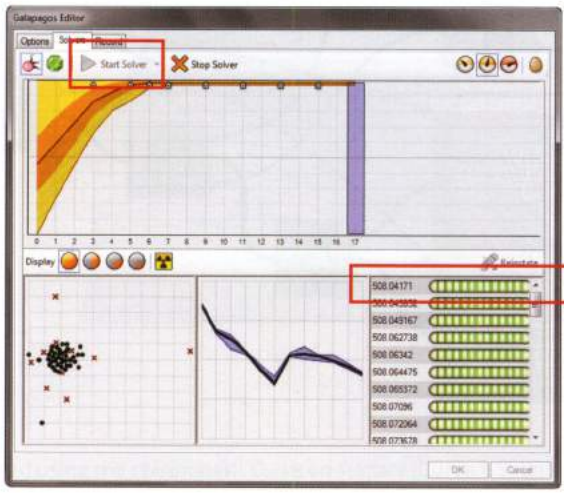


Since the *Curve on Surface* output (L) returns the curve's length value as the (uv) coordinates change position, a fixed element and a variable element exist respectively, that can be used to define an optimization procedure for an evolutionary solver. For this optimization *Galapagos* (Params > Util) is used. Similar to *Goat*, *Galapagos* has two ports for outgoing connections: *Genome* which corresponds to *Variables* in *Goat*, and *Fitness* which corresponds to *Objective* in *Goat*. The *Genome* port is satisfied by the arbitrary set of sliders embedded within the two *Gene Pool* components. The *Fitness* port is connected to the L-output of the *Curve on Surface* component.

Double-clicking the *Galapagos* component opens a contextual menu, where the *Fitness* can be set to be minimized or maximized.



Since, the shortest length is desired the *Fitness* option is set to *Minimize*. To perform the optimization the *Solvers* tab is opened and the *Start Solver* button is selected.

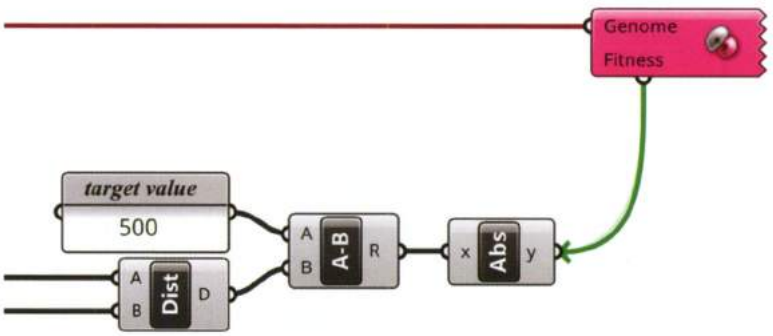


The optimization process runs until an optimum is found, which can take minutes or hours depending on the complexity of the solution. Galapagos can be stopped by selecting the button *Stop Solver* if the first value displayed on the bottom-right window (V1 in the previous image) becomes stable, tending to a specific number. In this example, the optimization returns a set of sliders representing (uv) coordinates of the interpolation points that correspond to the minimal path curve. The shortest curve is returned by the C-output of the *Curve on Surface* component.

Evolutionary solvers apply principles of evolution to problem solving, and are actually based on *population of candidate solutions*. The candidates attempt to reach better solutions with better genetic characteristics through *mutations*, *crossovers* and random changes. This is in contrast with exact solvers that target the single best solution. Roughly, evolutionary solvers perform a *natural selection* among possible solutions through a process in which only the best members of population (according to the specific *Fitness*) survive.

10.6.4 Target values

Both exact and evolutionary solvers operate by minimizing or maximizing a *Fitness* function. In some instances a target value for the *Fitness* function is desired; in these cases the absolute value difference between the *Fitness* function and the target value are minimized and maximized.



In the above definition the *Fitness* is defined as the distance between two points. In this instance, the target value is 500, the absolute value of the distance minus the target value is minimized using the *Fitness* function set to minimize.

Crassula "Buddha's Temple", photographed at the Huntington gardens, Los Angeles.

11_environmental analysis

"Don't fight forces, use them".

R. Buckminster Fuller

Several software applications are available in the field of environmental design; most however, are back-end analysis and give small contributions in exploring holistic design solutions. Due to difficulties in making changes during advanced stages of modelling/drawing designers often rely on personal judgment deduced from experience, over analysis. Thereby, narrowing the possibilities to explore a large set of environmental-informed designs.

The combined use of environmental and parametric packages can guide the generation of form from the initial sketches to the final design. This chapter introduces plug-in software used to bridge between parametric modelling and environmental analysis. The software find forms through real-time analysis based on insolation, lighting etc.. These techniques, supported by a deep understanding of data can lead to environmental-conscious designs.

11.1 Tools

The Grasshopper ecosystem provides users with several plug-ins for environmental analysis; some plug-ins perform complete environmental analysis, while others bridge between specific environmental analysis software. Among the latter is the plug-in **GECO**, which directly links between Grasshopper and one of the most powerful analysis software: **Autodesk® Ecotect® Analysis**, developed in 1996 by Andrew Marsh and subsequently acquired by Autodesk in 2008. This chapter will demonstrate the use of the plug-in GECO to solve environmental analysis problems.

11.2 GECO and Ecotect

GECO, developed by [uto], an Innsbruck-based design team and research collective founded by Ursula Frick and Thomas Grabner[37], contains a set of components for Grasshopper that enables data to directly flow between Grasshopper and Ecotect, making environmental optimization possible.

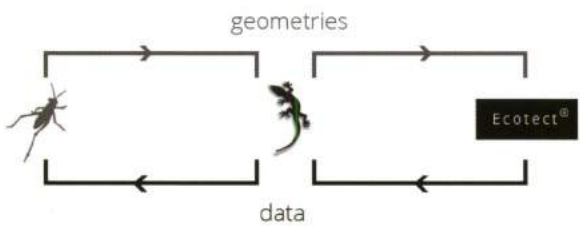


FIGURE 11.1
The GECO – Grasshopper – Ecotect Workflow.

The Ecotect interface is both graphically intuitive and analytically comprehensive. The software has the ability to organize data related to thermal performance, solar radiation, daylighting, shadows, and acoustical analysis. Ecotect also features a complete 3D modelling environment, however is not conducive to creating complex forms. GECO fills the void between shape modelling and analysis, allowing users

NOTE 37
Ursula Frick and Thomas Grabner are the founding directors of [uto]. Their research focuses, along with the generation and analysis of complex geometries, on the development of formal strategies, data driven design approaches and generative structures, implementing and evaluating the physical matter of a design. They have led studios and run workshops worldwide. They are also recognised for their development of digital tools of digital tools, namely PhysX, Geco, Flowlines, MeshPaint and MeshEdit, which seek to improve workflow in a parametric environment. Both graduated with distinction from the University of Innsbruck, with degrees in Architecture.

to parametrically model geometries in Grasshopper and export them to Ecotect generating real-time feedback loops used to inform and optimize the geometries of facades, covers, buildings, etc.

In summary, GECO's components facilitate:

- The exporting of points or geometries from Grasshopper to Ecotect;
- Setting analysis grids;
- Performing daylighting calculations (*EcoLightCal*);
- Calculating solar radiation (*EcoSolCal*);
- Importing feedbacks from Ecotect to Grasshopper;
- Importing Grasshopper solar diagrams calculated by Ecotect.

GECO component *EcoLua* enables users to interact directly with the Ecotect scripting language *Lua*.

11.2.1 Interface

Three software's are required to be installed to use GECO: Ecotect, GECO itself and Mesh Edit (see 6.3.3). Ecotect is a commercial software available as a 30-day trial or 3-years student license available at www.autodesk.com. GECO and Mesh Edit are both available as a free download from www.food4rhino.com. Once installed, a new panel "GECO" will appear in the Extra tab.

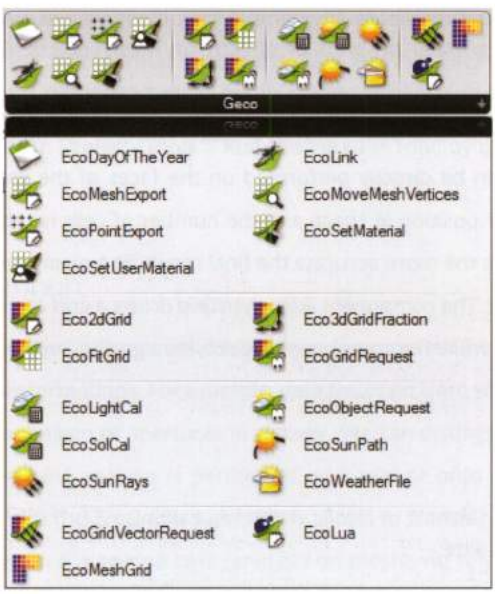


FIGURE 11.2
The GECO panel, hosted within the Extra tab.

GECO's components can be grouped into four categories:

- Link and export;
- Analysis grids;
- Calculations;
- Feedbacks.

Once a GECO component is place on the canvas, the component's name changes from what is displayed in the GECO panel; for example once added on the canvas: *EcoLink* becomes *Link Ecotect*, *EcoSolCal* becomes *Insolation Calculation*.

11.2.2 Link and export

Two components are used to link and export: first, the component *EcoLink* (*Link Ecotect*) is a switch that establishes a connection between Ecotect and Grasshopper, and second, the component *EcoMeshExport*, which exports objects modelled in Rhino or generated in Grasshopper into Ecotect, in order to perform analysis. Similar to other analysis software **Ecotect works exclusively on mesh geometries**.

11.2.3 Analysis grid and mesh-analysis

Ecotect performs analysis either on grids or directly on mesh-objects. **Grids** are a set of points within a space that can be imagined as *sensors* where Ecotect generates results; for example, lighting analysis is performed at several points within a space not only on objects. In other cases, such as insolation, analysis can be directly performed on the faces of the mesh-geometry. If analysis grids are necessary, their position in space and the number of cells is required to be defined. The larger the number of cells the more accurate the final result. The main components to set grids are *2dAnalysisGrid* and *FitGrid*. The component *2dAnalysisGrid* draws a grid according to a spatial domain and to a number of cells, while the component *FitGrid* sets a grid according to the dimensions of the current object in Ecotect.

11.2.4 Calculations

GECO's components calculate:

- **Solar diagrams** by using the two components: *EcoSunPath* and *EcoSunRays* (see 11.4);
- **Insolation** by acting on a mesh or a grid using the component *InsolationCalculations* which

returns the following values (see 11.6):

1. Incident solar radiation;
2. Absorbed and transmitted solar radiation;
3. Sky factor and photosynthetically active radiation (PAR);
4. Shading, overshadowing, and sunlight hours.

- **Lighting comfort,** by using the component *LightingCalculations* the following can be returned (see 11.8):
  1. Daylight factor;
  2. Daylight levels;
  3. Sky component.

WEATHER FILES

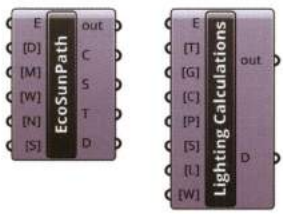
To execute environmental calculations, valid climate data is required to be set for the **geographic location** that hosts the object to analyze. Geographic data is embedded in specific files called *Weather Files*. A **Weather File** contains the climatic data for a location such as: solar radiation, prevailing winds, temperatures, rainfall, humidity etc. Ecotect has Weather Files embedded in the software for major world cities; other locations can be download directly from the U.S. Department of Energy (http://apps1.eere.energy.gov/buildings/energyplus/weatherdata_about.cfm). The USDOE website provides files in the .EPW format file (Energy Plus Weather File) which cannot be read by Ecotect which requires .WEA files (Weather Data File). The .EPW file can be converted into a .WEA file using the *Weather Manager* in Ecotect (Tools > Run the WeatherTool) by opening the .EPW file and saving it as a .WEA file.

11.2.5 Feedback / import

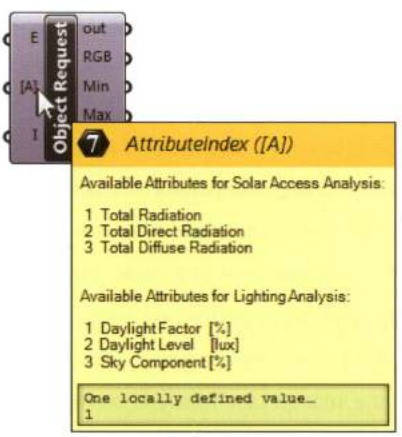
Once the analysis is performed in Ecotect, the results can be imported into Grasshopper to control and inform geometric transformations. For example, data returned from solar radiation analysis can be used to control the dimension of apertures in a cover. We can distinguish between two import modes depending whether the analysis is performed on a grid or onto a mesh. In the first case (analysis on grid) we will use the *EcoGridRequest* which allows to transfer into Grasshopper all the data calculated by Ecotect. In the second case (analysis on mesh), we rely on the *EcoObjectRequest* component able to import calculations made directly on mesh like radiation or lighting.

11.3 About GECO's components

Most GECO components have a similar structure facilitating **cascading connections**. To better understand this concept, two components, *EcoSunPath* (used to draw solar diagrams) and *LightingCalculations* (used to calculate lighting comfort) will be examined.

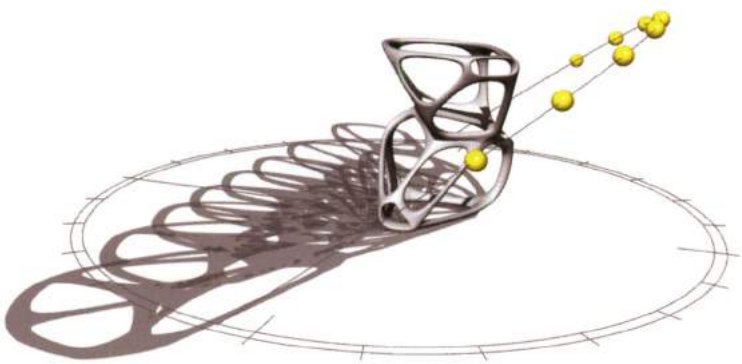


Both components have an input (E) that is satisfied by a Boolean statement which can be provided by a *Boolean Toggle*. A *Boolean Toggle* set to True will execute the specific calculation instructed by the component in Ecotect. All GECO components have: an output (out) which displays information received from Ecotect, and an output (D) which returns a Boolean value False if Ecotect is still running or True if the Ecotect calculation is complete. The output (D) and the input (E) can be used to establish a cascade connection between several components such that the Boolean value from the output (D) will activate the second component input (E). Similar to Grasshopper standard components, hovering the mouse over an input or output will provide specific information.

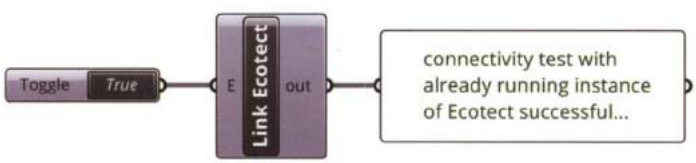


11.4 Solar diagram and shadows

Using the Components *EcoSunPath* and *EcoSunRays* a solar path diagram and the solar ray vector respectively, can be generated for a specific location and time. The solar ray vector can be used to draw the projected shadows of an arbitrary object onto a plane, as illustrated in the following image.



The necessary part of any GECO definintion is to establish a link between Ecotect and Grasshopper using the component *Link Ecotect*. The link can be established or disconnected using a *Boolean Toggle* connected to the input (E). The output (out) generates a text log which displays the result of the connectivity test. *EcoLink* is always an isolated component that is not connected to other components except the *Panel*.



Occasionally the component *EcoLink* (*Link Ecotect*) could be in a warning state, displayed by an orange component colour, meaning the connection with Ecotect has failed. If this happens the connection can be re-established by switching the *Boolean Toggle* from True to False and again from False to True. If *EcoLink* is in an error state, displayed by a red component colour, Ecotect or GECO are not installed properly.

SUN PATH

Once the connection with Ecotect is established the component *EcoSunPath* can extract sun path data for a specific day input [D], month input [M] and location as defined by a Weather File input [W]. The input [D] must be fed by a slider ranging between 1 and 31 and input [M] by a slider ranging between 1 and 12 and input [W] by a Weather File. Weather Files can be loaded using the component *File Path* (Params > Primitive) – renamed as *weather file* in the following images – by right-clicking the component and selecting the option *Set One File Path* from the contextual menu. The .WEA files for specific locations can be found in the default folder: *c:\programs(x86)\Autodesk\Ecotect Analysis\ Weather Data.* If the E-input is set to True the analysis starts and a sun path diagram will appear in Rhino. The component *EcoSunPath* returns as its output two curves representing the *sundial* output (C) and the sun path output (S). The output (T) is a number that indicates the *sunrise and sunset*. The scale of sun path and sundial curves can be controlled by the [S]-input.

SOLAR RAY

Once the sun path is defined for a specific day, the component *EcoSunRays* returns a solar ray for a specific time. The inputs [D] and [M] are required to be connected to the same sliders used for the component *EcoSunPath.* The input [T] specifies the time and is connected to a slider ranging from sunrise to sunset (e.g. from 8.00 to 18.00). The V-output of the component *EcoSunRays* is the solar ray vector, which can be displayed using the component *Vector Display* (Display > Vector). The P-output is the vector's start-point.

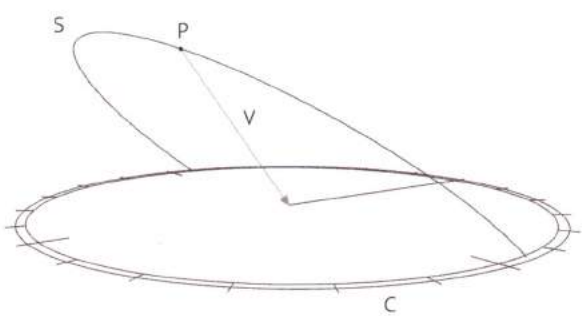


FIGURE 11.3
EcoSunPath and *EcoSunRays* draw in Rhino the sun path curve (S), the sundial (C) and the solar ray vector (V) for a specific place and time.

SHADOWS

The data so far calculated can be used to draw the shadows of an arbitrary mesh-object onto an arbitrary plane. The component *MeshShadow* (Mesh>Util) calculates the cast shadows of a mesh

geometry input (M) from a solar ray vector input (L) satisfied by the V-output of *EcoSunRays* component.

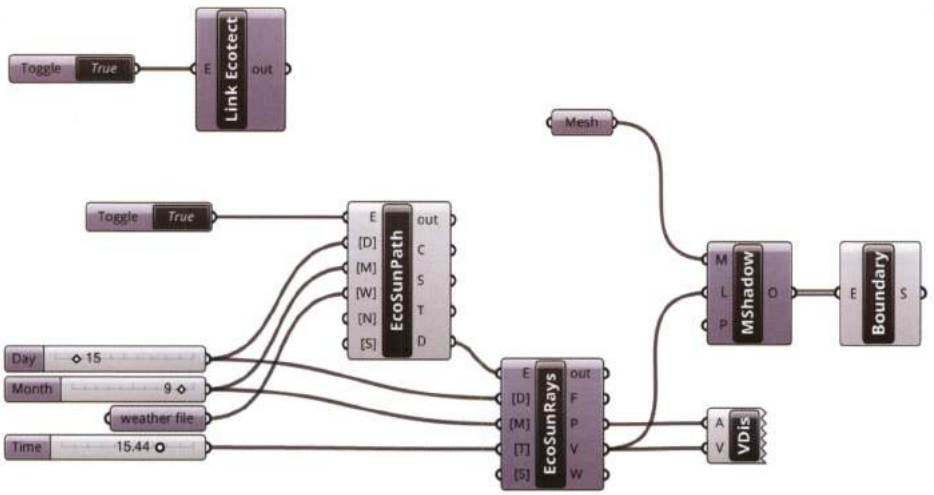
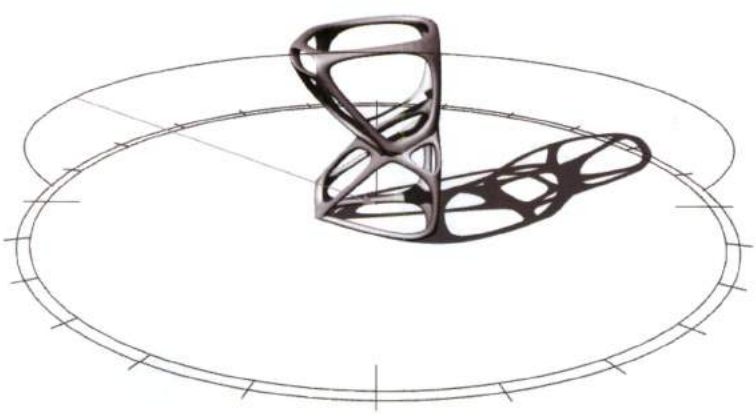


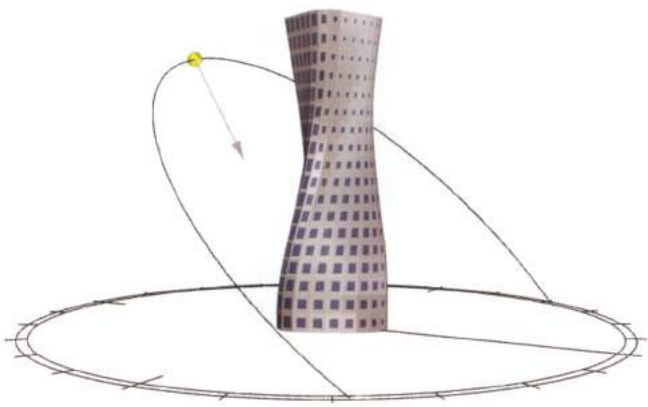
FIGURE 11.4
The complete algorithm that calculates sun path, solar ray and shadows for an arbitrary mesh geometry.

The output (O) stores the shadow contour data as projected on an arbitrary plane input (P). To better visualize the shadows a planar surface can be created using the component *Boundary Surfaces* (Surface > Freeform).

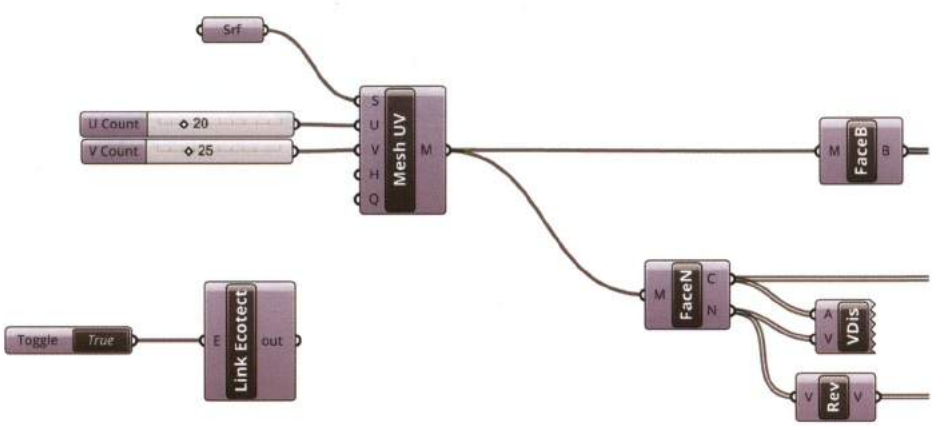


11.4.1 Responsive skin

Environmental data calculated in Ecotect filtered through GECO can be useful in the conceptual stages of design. For instance, the solar ray vector can be used to control a facade's window dimensions, by comparing the angles between the solar ray vector and the surface's normals to inform a geometric transformation.



The responsive skin algorithm starts by defining a single freeform surface and then converting it to a mesh using the component *Mesh Surface*. The defined number of U and V subdivisions becomes the dimensions of the facade panel matrix. As second step the normal vectors at each mesh-face center are calculated, using the component *Face Normals* (Mesh > Analysis). Then, we extract the mesh edges using the component *Face Boundaries* (Mesh > Analysis) outputting a set of closed polyline curves.



After the normal vectors are calculated it is crucial to verify the normal vectors direction using the component *Vector Display*. The vectors must have the same sense as the solar ray vectors, thus the vectors must be directed towards the inside of the surface. In the example, the normal vectors are directed toward the outside of the surface and are required to be inverted using the component *Reverse* (Vector > Vector). After the vectors have the correct sense, the component *EcoLink* (*Link Ecotect*) can be used to establish a connection between Ecotect and Grasshopper.

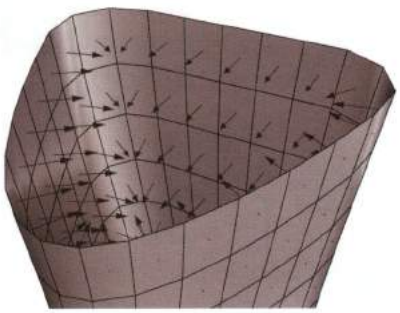
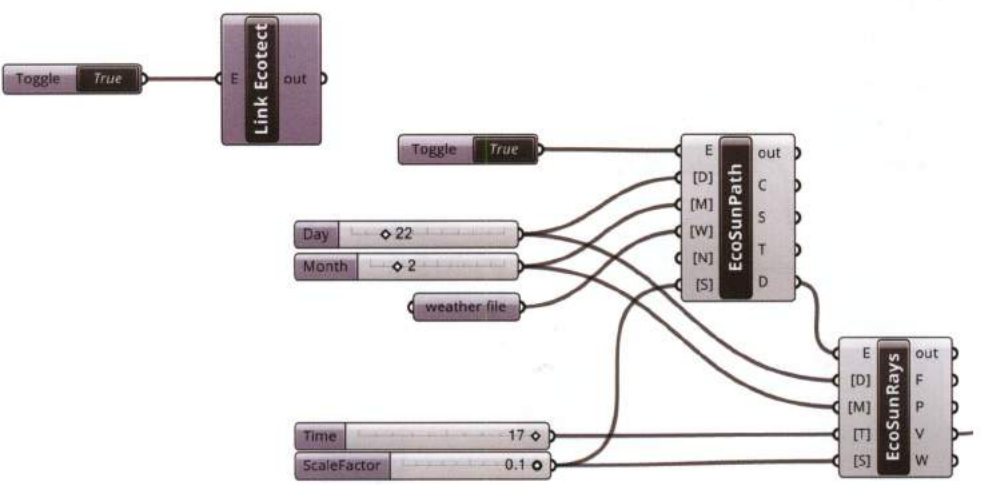


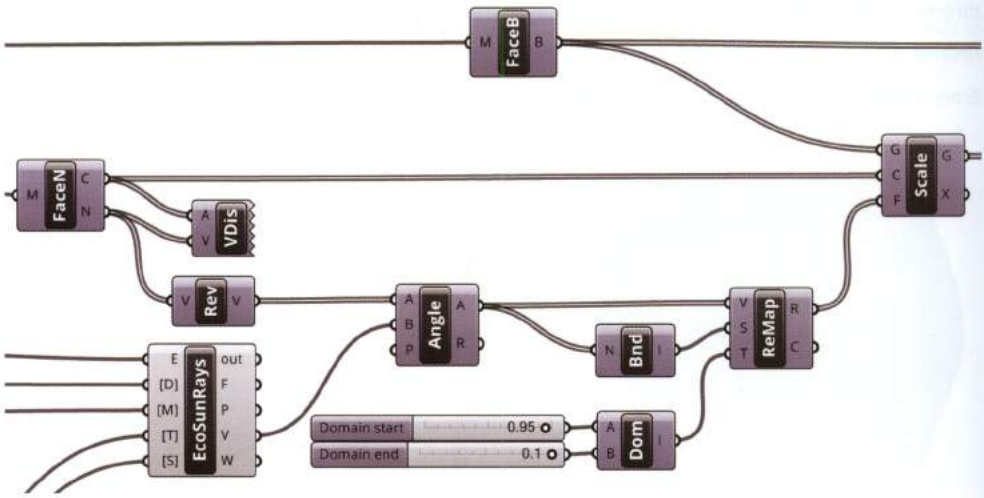
FIGURE 11.5
Faces' normals must be oriented according to the solar rays, meaning they must be directed toward the inside of the geometry to analyze.

The second step is to generate the solar diagram and the solar ray vector using the components *EcoSunPath* and *EcoSunRays* respectively for a specific day and time.

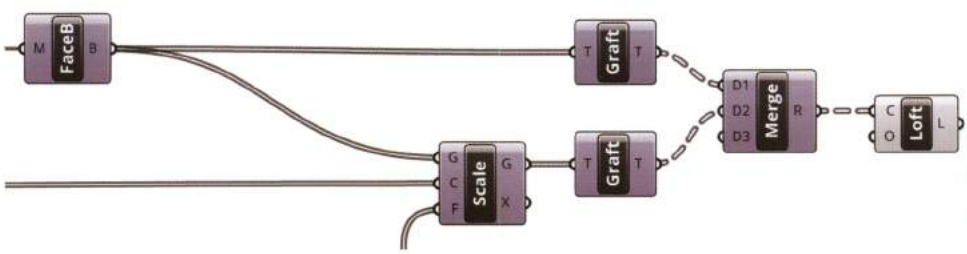


The component *Angle* (Vector > Vector) is used to measure the angle in radians between the solar ray vector and faces' normals. To vary the size of the apertures the mesh edges are scaled using the component *Scale*; such that, the lower the angle value the lower the scale factor. In order to inwardly

scale the mesh edges the scale factor must be within the domain (0,1]. As a result, the component *remap* (see 2.5) is used to remap the angle values to a target domain.



Lastly, the variable holed skin is generated by a *Loft* between each original edge item to the respective scaled edge item. As explained in chapter 5, the two data flows are required to be *grafted* before they are merged. After the skin is defined, modifying the [T]-input of *the component EcoSunRays* will yield varying aperture dimensions.



11.5 Exporting geometries and importing data

As discussed in the previous section, Grasshopper can calulate the shadow of a mesh modelled in Rhino using sun path data from Ecotect, through GECO. Some types of analysis, for instance lighting and insolation, require that mesh geometries be exported from Grasshopper to Ecotect to perform analysis then the results are returned into Grasshopper.

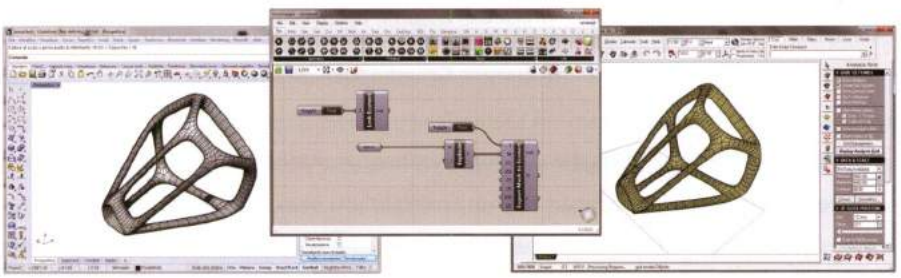
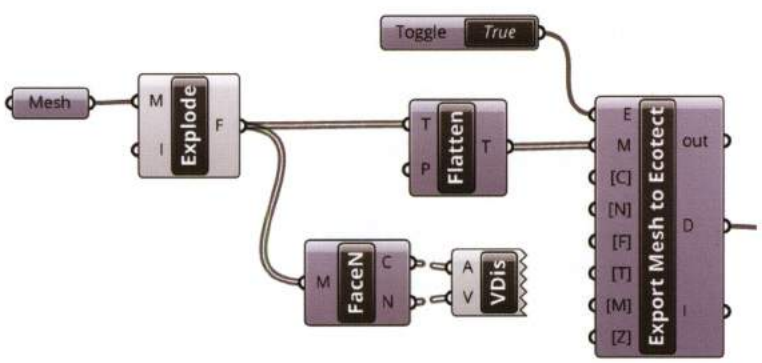


FIGURE 11.6
A mesh geometry modeled in Rhino and exported into Ecotect using GECO as a bridge.

11.5.1 Exporting geometries from Grasshopper to Ecotect

The main export component is *EcoMeshExport* (*Export Mesh to Ecotect*).



When exporting mesh geometries, some important *guidelines* should be followed:

- **Exploding meshes**: to improve compatibility meshes must be exploded using the component *Mesh Explode* (Mesh > Analysis);

- **Checking faces' normal vectors**: normal vectors must be directed towards the outside. The direction of normals' can be visualized using the components *Face Normals* and *Vector Display*;
- **Flattening data**: the output (F) of component *Mesh Explode* must be in *Flatten* mode before connecting it to the M-input of the component *Export Mesh to Ecotect*. Otherwise, Ecotect will import just the faces hosted within the last branch.

The E-input set to True exports the mesh to Ecotect. It is important to point out that the export procedure is controlled by the N-input which can assume three values:

- if **[N] = 0** GECO exports the current Grasshopper meshes and deletes any geometry present in Ecotect;
- if **[N] = 1** GECO exports the current Grasshopper meshes and deletes just the meshes exported by the latest *Export Mesh to Ecotect* (mesh selected in Ecotect);
- if **[N] = 2** GECO exports the current Grasshopper Meshes and maintains all geometry present in Ecotect (this option may lead to overlapping errors).

The remaining inputs are:

- **[C]**, is the scale factor of the exported mesh. Since Ecotect usually operates in millimeters we can set the scale factor to 0, the default value, if the Rhino environment is also set to millimeters. Otherwise, if the Rhino environment is set to meters we must set [C] = 1000;
- **[F]** is an input related to the analysis grids. Through a *Boolean Toggle* set to True we can fit the analysis grid to the exported mesh;
- **[T]** and **[M]** set an element type and a material to a mesh respectively;

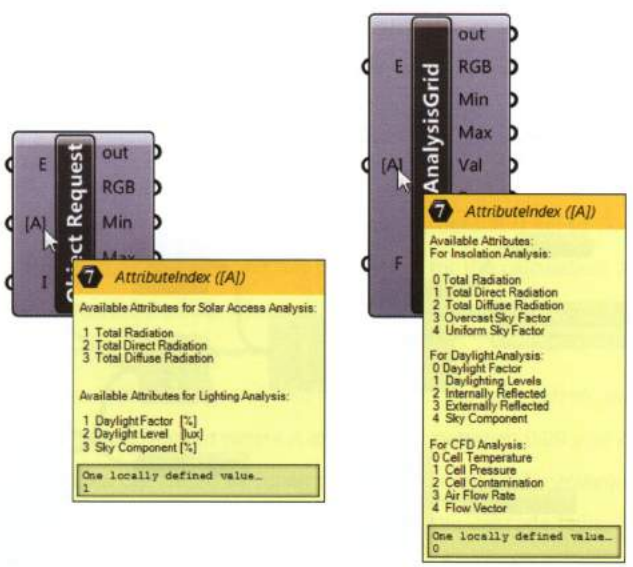
- **[Z]** defines the Ecotect's **zone**. Zones are defined in Ecotect as homogenous enclosed areas or volumes and often but not always coincide with "rooms". Zones are managed similar to CAD-layers, names can be acquired by typing text into a *Panel* connected to the input [Z]. If multiple *Export Mesh to Ecotect* components are used, the same input [Z] is required in order to export different meshes into the same zone.

11.5.2 Importing data from Ecotect

There are two methods to import data from Ecotect into Grasshopper; the selected method depends on whether the analysis is performed on grids or on meshes.

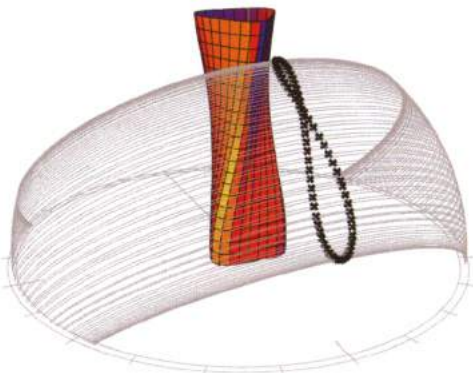
The component *EcoObjectRequest* (*Object Request*) is used for meshes. The component returns data calculated at center of each mesh face and it requires as I-input the indices of mesh faces. The I-output of *EcoMeshExport* provides the indices. The component *EcoObjectRequest* has two main outputs: (Val) which provides numeric values resuting from the specific analysis and (RGB) which assigns a color gradient to a mesh according to the numeric values. The component *EcoGridRequest* (*ImportAnalysisGrid*) is used to import grids and data performed on grids.

Both components have an input [A] that allows to select specific results (attributes) from the entire set of data returned by an analysis. For example to select the attribute *Total Radiation the* [A]-input must be set to 0. The available attributes can be read on the help menu which appears hovering the mouse over the input.

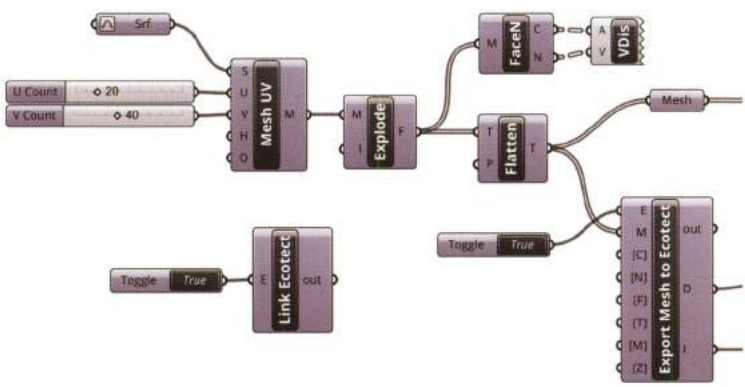


11.6 Insolation analysis

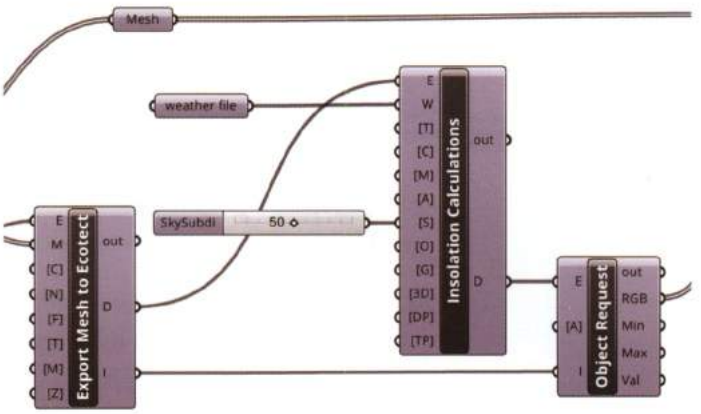
One of most important parameters of *passive energy efficiency* is **insolation**, or the energy received on a given surface during a given time measured in *watt* per *square meter* (Wh/m²). To calculate insolation, Ecotect uses the **Lambert's cosine law**.



Buildings' geometry and materials should be defined to maximize insolation during the winter and minimize it during the summer. Energy optimization usually focuses on the concept of *passive solar gain*. Passive solar gain is defined as the increase in temperature within a space due to the solar radiation. Insolation is also pivotal in defining the correct position and dimensions of solar collectors and photovoltaic cells. In order to calculate insolation on an arbitrary given mesh, the component *EcoSolCal (Insolation Calculations)* is used. The definition is composed of several steps: define the mesh-geometry to analyze, run Ecotect by *EcoLink* (*Link Ecotect*), and export the mesh using the export guidelines explained in (11.5).



The insolation calculation performed by the component *EcoSolCal (Insolation Calculations)* is cascade-connected (see 11.3) to the component *EcoMeshExport* (*Export Mesh to Ecotect*).

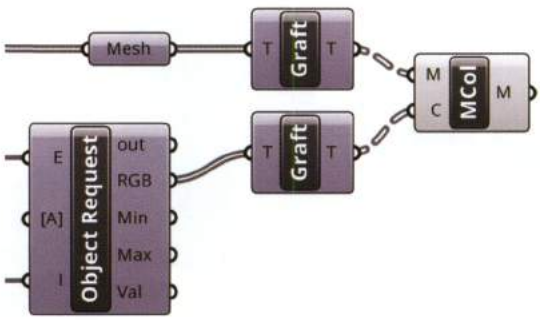


The main inputs are:

- The **Weather File [W]** as explained in 11.2.4.
- The **Terrain Types [T]**: 0 for a location exposed to the wind, 1 for rural settings, 2 for suburban settings and 3 for dense urban settings.
- The type of **Calculation**s to perform **[C]**: 1 for incident solar radiation, 2 for solar absorption-transmission, 3 for sky factor and photosynthetically active radiation.
- The **Sky Subdivision** is a technique that Ecotect uses to subdivide the sky dome in order to perform calculations. The smaller the value, the higher the accuracy. The input [S] must be fed by a number ranging between 1 and 15; [S] can also be set to 50 to run a fast calculation.
- **[G]** Switches analysis between Objects and Grid. By default is set to Object.
- **[DP]** Determines the start and end day of the year for the calculation, and is set using the component *Construct Domain* where A and B range between 1 and 365 (Julian date). By default the domain is [1,365].
- **[TP]** Determines the starting and ending time for the calculation, and is set using the component *Construct Domain* where A and B range between 0.00 and 23.99.

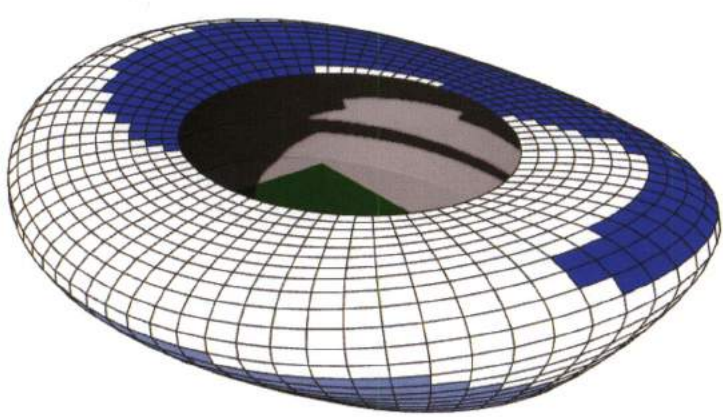
In order to import the results calculated by Ecotect, the component *EcoObjectRequest* (*Object Request*) is connected to the *Insolation Calculation* component using the cascade logic. The input (I) or the faces' indices is connected to the output (I) of the component *EcoMeshExport*.

As discussed earlier, the component *EcoObjectRequest* has two main outputs: *(RGB)* and *(Val)*. (RGB) is useful to generate a colour gradient illustration which is helpful in the concpetual stage of design. To display the colour gradient the component *Mesh Colours* (Mesh > Primitive) is connected to the output *(RGB)* of the component *EcoObjectRequest*. The numeric values are returned by the output (Val).



11.6.1 Practical Exercise: Positioning photovoltaic panels

The following exercise demonstrates how insolation analysis can guide the positioning of photovoltaic panels on a cover. The algorithm divides the cover into a set of panels and identifies the panels where insolation is maximized. The complete exercise can be acessed using the QR code. 7



11.7 Analysis Grids

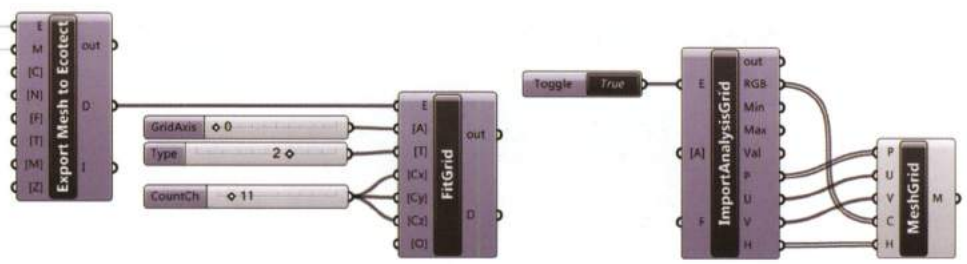
Many calculations, in particular those related to lighting, are performed on analysis grids. As we already explained, grids are set of points which can be considered sensors placed in the space. Grids can be set according to the following methods:

- Grids can be fitted to the extents of selected objects in Ecotect, by the component *EcoFitGrid* (*FitGrid*);
- Grids can be defined using the component *Eco2DGrid* (*2dAnalysisGrid*) which sets the dimensions and the position of grids.

In both cases, two additional components are required to be connected: *EcoGridRequest* (*ImportAnalysisGrid*), which imports analysis values and *EcoMeshGrid* (*MeshGrid*), which draws the grid in Rhino.

FIT GRID

In order to adapt a grid to a mesh geometry exported into Ecotect, the *EcoFitGrid* component is connected to *EcoMeshExport* using the cascade logic.



EcoFitGrid requires:

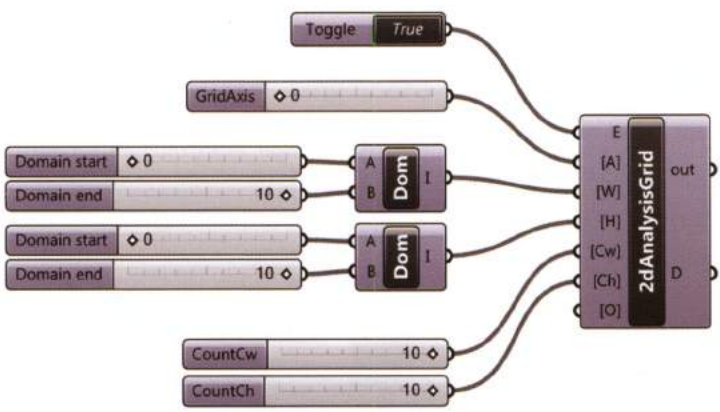
- The reference plane [A] can be defined by a *Number Slider* set to: 0 (XY plan), 1 (YZ plane) or 2 (XZ plane);
- The type of fit [T] which specifies whether the grid will be created *within* the mesh geometry (slider set to 0), *around* the geometry (slider set to 1), *adapted* to the actual shape (slider set to 2) or *larger* than the object (slider set to 3).

The three inputs [Cx], [Cy], [Cz] control the count of faces in three main directions. Input (O) defines the offset from the reference plane.

Once a grid analysis is completed, the results can be imported using the component *EcoGridRequest* (*ImportAnalysisGrid*). The input [A] sets selected attributes for the specific analysis. For instance, for a daylight analysis (see 11.8) the *daylight factor* will be returned when A=0 and the *daylighting levels* will be returned when A=1. Other attributes can be set specifying numeric values as specified in the contextual menu. Lastly, the component *EcoMeshGrid* (*MeshGrid*) generates a colored mesh according to the analysis values.

2D ANALYSIS GRID

The component *Eco2DGrid* (*2dAnalysisGrid*) is useful when manually defining a grid's characteristics; in particular: the plane [A], the domain width [W] and height [H] and the count of faces in W and H directions.



It is recommended that grids are created such that domains in W and H directions are both positive or negative. In other words the grid must lie entirely in one of four quadrants of Rhino's space.

11.8 Light Control

Light distribution plays a large role in a building's energy use as well as visual comfort. As a result, the optimization of light distribution is a key strategy in passive design. The component *EcoLightCal* (*LightingCalulations*) analyzes illumination levels and distribution of light using the parameters:

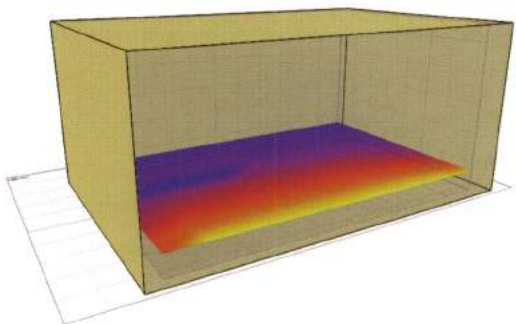
- Daylight Factors;
- Daylighting Levels;
- Internally Reflected;
- Externally Reflected;
- Sky Component.

Light distribution calculations requires the light levels outside the buildings, including the direct sunlight and the indirect illumination of the sky. The ***sky illuminance*** value (lux), is derived from statistical analysis of dynamic outdoor sky illuminance levels. The following image shows design sky illuminance levels for different latitudes. Several tools available online can be used to determine recommended Lux levels for particular locations.

FIGURE 11.7
Sky illuminance (in lux) according to latitude.

11.8.1 Visual comfort of a room

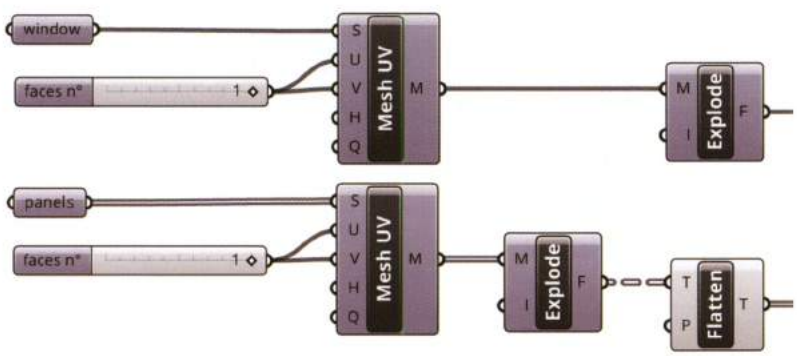
This example will cover a visual comfort analysis of a prismatic room with an east-facing glass wall.



SETTING THE GEOMETRY

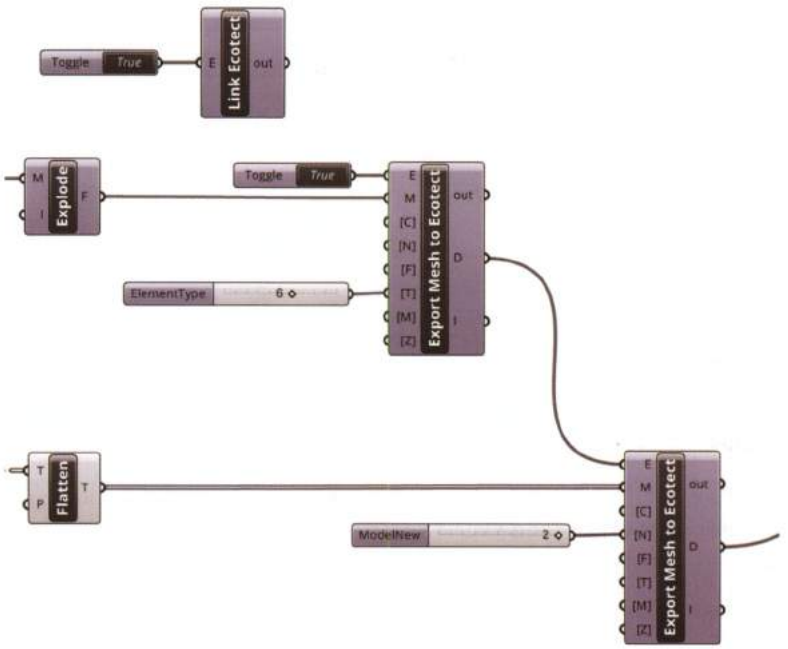
The first step in the definition is to define a box in Rhino or Grasshopper with the dimensions: 12000 millimeters length, 6000 millimeters width and 6000 millimeters high. Subsequently, the box is exploded into individual faces, the east face is collected by a *Surface* container component renamed as *window* and the remaininig faces are collected by a *Surface* container component renamed as *panels*.

The second step is to convert the two set of surfaces into two sets of meshes, so different materials can be assigned to the window and panels. Each surface is converted into mesh using just one subdivision for U and V directions, to output one face for each mesh. Lastly, the two sets of meshes are exploded using the component *Mesh Explode,* and the *panels* data is then flattened.



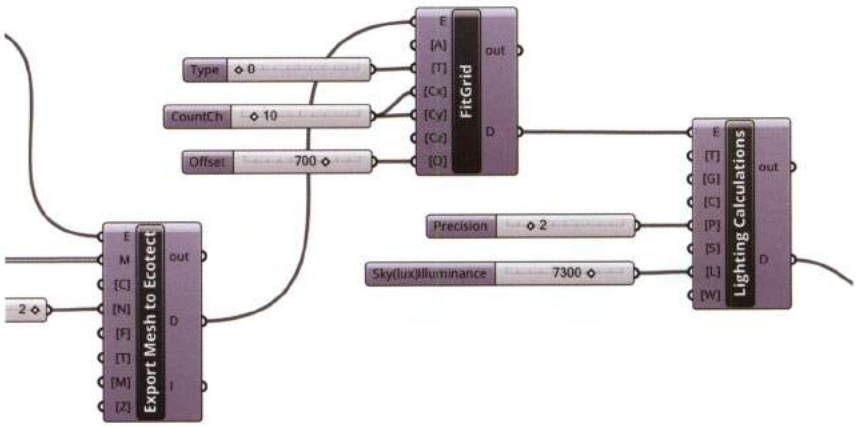
EXPORTING MESHES INTO ECOTECT

Meshes are exported using two *EcoMeshExport* components. In order to assign the *window* type to the east-facing meshes we have to set the [T]-input to 6. The *panels* meshes are exported with the default [T] value (7).



SETTING THE ANALYSIS GRID

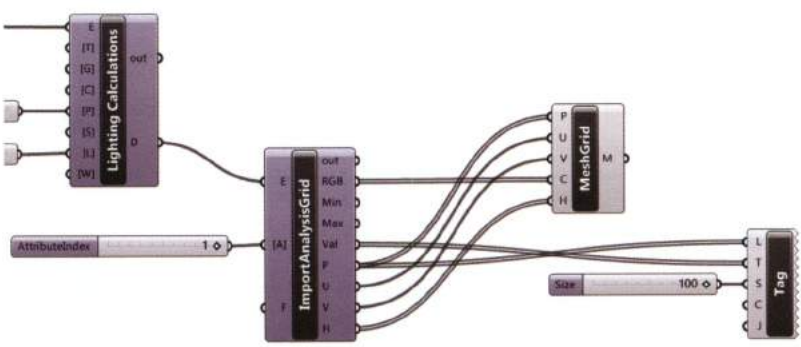
In order to define the analysis grid the component *EcoFitGrid* is used, setting the inputs such that: the grid will be parallel to the XY plane (input [A] set to 0), with an offset of 700mm (input [O]). The grid will be created within the object by specifying [T]=0) and that the cells count is 10 in [Cx] and [Cy] directions. The input [Cz] is irrelevant in this case since the analysis will be performed on the 2D grid of points at 700mm from the reference plane.



LIGHTING ANALYSIS

Lighting analysis is performed by the component *EcoLightCal* (*Lighting Calculations*), in the example the precision input [P] is set to 2 (High Precision) and the *sky illuminance* (see figure 11.6) input [L] is set to 7300 lux. The input [T] specifies whether the calculation is performed over the analysis grid (0, default value) or over point objects (1). Input [C] sets the available lighting calculations.

In order to import results the component *EcoGridRequest* (*ImportAnalysisGrid*) is used, while the component *EcoMeshGrid* (*MeshGrid*) generates a colored mesh according to the analysis values. As discussed in 11.7, the component *EcoGridRequest* (*ImportAnalysisGrid*) enables selected attributes to be set for the specific analysis through the A-input.

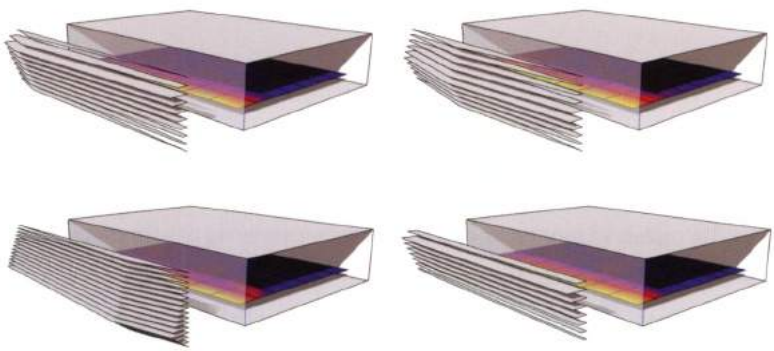


The calculated values can be directly displayed on the grid using the *Text Tag 3D* component (Display > Dimensions), feeding the L-input with the P-output (grid's points) of *EcoGridRequest* (*ImportAnalysisGrid*). The T-input needs the calculated values and (S) the size of text.

11.8.2 Practical Exercise Galapagos + GECO, visual comfort optimization

The following exercise demonstrates how to optimize – using Galapagos – the visual comfort of a space by modifying number and shape of a set of sun shading devices. The complete exercise can be acessed using the QR code.

[8]



This chapter is part of a research developed by **Maurizio Degni**, a specialist in the field of energy and environmental analysis related to complex systems with a particular focus on parametric and optimization strategies. Maurizio Degni worked for several offices in Italy such as Studio Kami and J.M. Schivo where he collaborated on many international competitions, projects and research activities. With Arturo Tedeschi he designed the NU:S Installation within the Cloister of Bramante (Rome) and he had a main role in the computational design of the NU:S Parametric Shoes an avant-garde project that matched advanced design techniques and prototyping technologies. From 2012 he is a tutors assistant for the AA Rome Visiting School.

Post Digital Strategies

Pragmatic Computation in Grasshopper

Brian Vesely

makelite design studio

Data: big data, small data, ubiquitous data. Algorithms fed by data are indifferent to output. The practice of organizing data to generate geometry has been explored by design offices since the early 1980's, discussed in academic texts, and demonstrated at institutions. Beyond the commonly perceived formal scope, the advantage of parametricism is the iterative calculability to formulate pragmatic data.

Parametricism is often used and abused as a justification for design decisions without close study of the inputs; decisions are often made without integrity to produce a formal output. *Post digital strategies*[1] considers the organization of data as a tool to produce a pragmatic output. Parametricism and pragmatism are seemingly polar concepts; however, a pragmatic parametric workflow can expose latencies in even mundane problems.

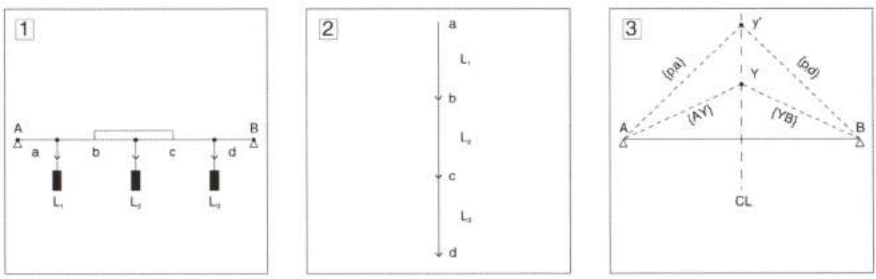
As discussed in Chapter 9, *Digital Simulation,* a hanging chain can be illustrated in Kangaroo following Hooke's Law. As an alternative, a hanging chain model can be calculated using computation graphical statics in Grasshopper. Graphical statics is an equilibrium method of visualizing forces acting on a rigid body by graphically representing forces with vectors that are proportional in magnitude and equivalent in direction to the final geometry of the body. Graphical statics is a pragmatic approach to form-finding that, when used in conjunction with optimization solvers, can produce intelligent output. Intelligent graphic statics applied to a reversed cantenary structure can be defined by following the subsequent algorithmic workflow diagrams[2]. Workflow diagrams – similar to spatial, conceptual, and structural diagrams – are an essential means to define the procedural logic of complex multi-part algorithms. Workflow figures 1-12 decompose a graphical static definition into its respective sub-parts; within each part, variables that are required to be calculated are noted. Figure

1. The graphical statics algorithmic logic presented was researched in the independent study, *Post Digital Strategies, at the Illinois School of Architecture with students Catherine Lie and Ailin Wang.*

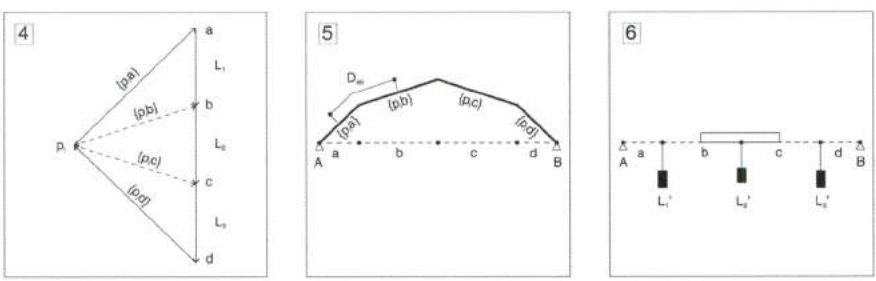
2. For additional information on this topic, see: Allen, Edward, and Wac Zalewski. *Form and Forces: Designing Efficient, Expressive Structures*. Hoboken, N.J.: John Wiley & Sons, 2010.

13 illustrates how the iterative calculability of Grasshopper can be used as optimization input. The workflow diagrams used to organize the data of a generic static definition of a reversed cantenary are illustrated and discussed in the following pragmatic example.

Loads calculated for the given span and material are converted into vectors and placed incrementally along the loading line (figure 1). The loads are transposed to the initial force diagram as the load vectors $\{L_1, L_2, L_3\}$ (figure 2). The vertical height point (y') is set at the centerline of the loading line two times the desired funicular height (Y), and tangent vectors $\{p_i a\}$ and $\{p_i d\}$ are constructed by connecting points (A) and (B) to point {y'} (figure 3).

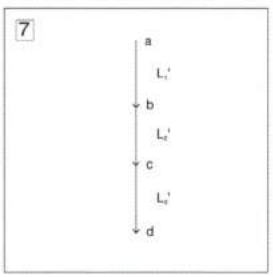


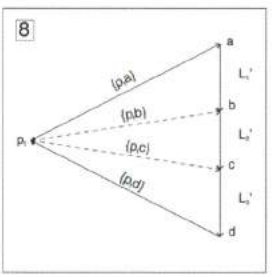
The form diagram tangent vectors $\{p_i a\}$ and $\{p_i d\}$ are translated to the load line and intersected to define the initial pole (p_i). Points (b) and (c) are connected to (p_i), defining vectors $\{p_i b\}$ and $\{p_i c\}$ (figure 4). The length of each loading segment is measured (i.e. D_{ab}) on the initial loading diagram (figure 5), and the loads are updated for each segment (figure 6).

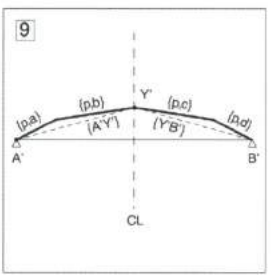


The loads are transposed to the final force diagram as the load vectors $\{L_1', L_2', L_3'\}$ (figure 7). The trial pole (p_t) is defined as a point in proximity to the final load line. Points (a), (b), (c), and (d) are connected to the point (p_t) defining vectors $\{p_t a\}$, $\{p_t b\}$, $\{p_t c\}$, and $\{p_t d\}$ (figure 8). The defined vectors

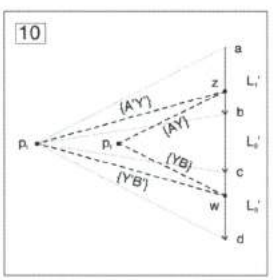
are used to construct the trial form polygon. The intersection of the geometric centerline and the trial form polygon defines point (Y'). Vectors {A'Y'} and {B'Y'} are defined by connecting points (A') and (B') to point (Y') (figure 9).

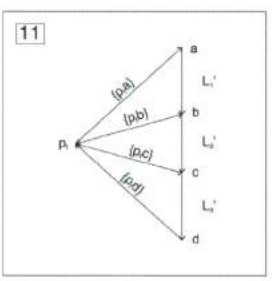






Vectors {A'Y'} and {B'Y'} are translated to originate at point (p_t) and intersected with the load line defining points (z) and (w). Vectors {AY} and {YB} are translated to originate at points (z) and (w) respectively and intersected to define point (p_f) (figure 10). Points (a), (b), (c), and (d) are connected to the point (p_f) defining vectors {p_fa}, {p_fb}, {p_fc}, {p_fd}, and the final force diagram (figure 11). The calculated vectors are translated to construct the final form diagram through point (Y) (figure 12).





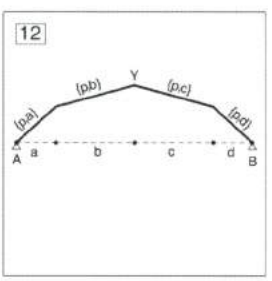
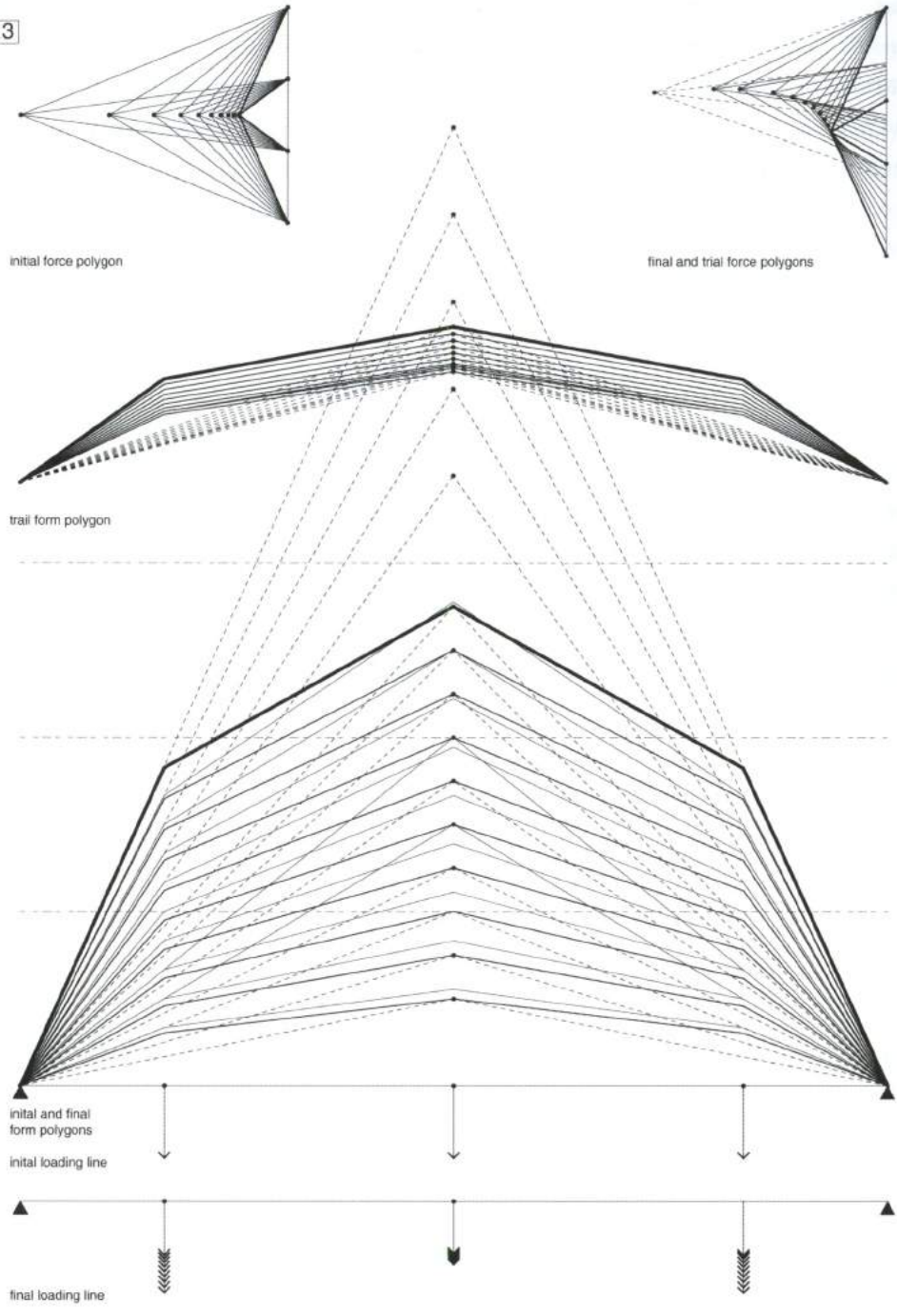


Figure 13 displays an abbreviated illustration of solutions in a composite diagram. The solver Goat can be used to minimize the *objective*, axial force, as calculated by measuring the length of the vectors respectively in the final force diagram. The calculated axial force can be used to determine the thickness of the vault using the stress equation:

F_{allow}=P/A expanded to **t**=(**P** lb)/((**12** in)*(F_{allow} lb/in^2))

13


initial force polygon
final and trial force polygons
trail form polygon
inital and final
form polygons
inital loading line
final loading line

Consideration within the definition must be given to the minimum and maximum rise-to-span ratios for issues of stability as well as the minimum thickness[3]. These considerations can be overcome by knowledgeably setting the range of the sliders as well as the conditional statements with respect to the axial force-thickness relationship. Once the generic workflow has been defined in Grasshopper, the sliders for the number of segments and the span can be expanded, the position of the restraints can be manipulated, and series components can be added to define a multitude of vaults. The complete funicular definition as well as other Grasshopper definitions and tutorials can be viewed and downloaded from [www.make-lite.com].

This pragmatic afterword is intended to amplify a point that the author Arturo Tedeschi makes throughout the text: Grasshopper, and more generically, algorithms, are tools to organize data; it is up to the user to determine the output. The text's compilation of examples should not be seen as how-to manual to complete formal exercises; instead, the knowledgeably crafted examples should be seen as a means to understand the way data is organized. In the end, understanding the way data is organized unleashes the power of Grasshopper.

Brian Vesely

Brian Vesely is the lead Designer at *makelite*. He graduated with a Masters Degree from the University of Illinois School of Architecture in 2012. As a Visiting Lecturer at the Illinois School of Architecture, he was part of the Detail and Fabrication Concentration (d+F), teaching courses in fabrication, computation, digital workflow, and design. He operates from experience in the fields of architecture, land surveying, and engineering. His work occurs at the intersection of speculative narratives and anecdotes of construction, exposing tectonic latencies. He creates fun-rigorous-smart assemblies exploring the potentials of fabrication, computation, and emerging infrastructural and ecological potentials.

3. Additional attention is required to consider all loads imposed on any structure and the results should be reviewed by a qualified engineer; this example is used to demonstrate pragmatic data logic only.

I am City, we are City

Francesco Lipari
Founding principal at OFL Architecture

When I think about the future, my mind comes back to the 80's to the dawn of the new millennium. I clearly remember that period, no doubt that it seemed like the future to me: a time when the scale between technology and humanity was perfectly balanced. The future, in fact, always coincides with something simple that simplifies our life. Today, we live in complex times which are a sort of borderline for humanity. In response to rapid changes of lifestyles and habits, humans must adapt and change their own nature, influencing the urban environment, understood as the ideal ground of sharing and exchange. Everything changes at an unprecedented speed that we are not prepared for, dragged by a two-speed rhythm. On the one hand there is *technology*, which proceeds inexorably as it had suddenly realized to be in late on original plans; on the other hand, there is *humanity*, which gathers those who are similar in feel an estrangement in which they must live, as the current stage doesn't make them protagonists. These are unprecedented times, we have never been so close and so distant at the same time. The natural, adaptive human-mechanism seems to be jammed, or simply inadequate to face the deep changes which happen cyclically at ever shorter intervals.

The shape of earth and circularity of events leads to idea that our world is finite, predetermined. Humans are naturally inclined to interpret signs, trying to anticipate changes that actually seem to have already happened. Time variations and continuous adaptations are just the effects of something which has already happened and recurs, in a new shape.

The man of today, for instance, is closer to a man who lived in fifth century than to a man who lived in nineteenth century. The latter, in turn, is similar to a man lived in roman age. In this scenario, apparently anachronistic – in which is difficult to find humanity – cities grew up. Cities are a representation of our society and, today we are losing control.

The typical model of independent and self-sufficient cities no longer exists. We are experiencing a new "ageographical" city, without precise spaces and forced to demonstrate its strength to face and bear new rhythms. Each city will become a *follower* of a bigger city within a process which is similar to the editing of an infinite book, with an infinite number of pages, hosted in a universal library. Cluster-cities will become nodes of an algorithmic-generative definition, tailored to support a global vision.

Moreover, the current technology-overflow is generating two different effects on the same actors, i.e. the human beings. By increasing the level of interaction and communication two kinds of societies

emerge. The first one is a *high-communication* society with no territorial and expressive boundaries, while the second one is a *low-communication* society, which reinterprets obsolete technologies with the dual aim to save memory and identity; such a society also uses the technology-divide as an elitist and less controllable type of communication.

All this leads to a disaggregation of consolidated models of urban management and opens the doors to the individual contribution as a model for generation and implementation. We will experience a new age where humans will be a part of a complex bio-mechanical mechanism structured and integrated with urban environment. Technology will further simplify its hardware and also the city will be smaller and smaller, personal, up to coincide with the humans themselves.

Francesco Lipari

Francesco Lipari is a Sicilian architect based in Rome. Recipients of several Prizes for Young Architects Francesco has been lecturer at the MAXXI and MACRO museum and curator of several architecture projects. He's the founding principal of OFL architecture and was formerly senior architect at the Fuksas office in Rome and Mad in Beijing. He's also founder of CityVision (http://www.cityvisionweb.com/) an innovative architecture platform with the aim of generating a dialogue between the contemporary city and its future image – http://www.oflstudio.com/.

Parametric Urbanism: a New Frontier for Smart Cities

Paolo Fusero, Lorenzo Massimiano, Arturo Tedeschi, Sara Lepidi[1]
Planum. The Journal of Urbanism, no. 27. vol. 2/2013, pp. 1-13.

[...] In only a few short years the concept of Smart Cities has evolved from an evocative representation of futuristic digital metropolises into an overblown term used to indicate all manner of virtuous processes: economic, environmental, technological, social, etc. It now appears that any human activity we wish to qualify in positive terms cannot avoid being tagged with the adjective "smart". This induces a reflection: on the one hand the fact that smart thinking applied to cities and territories is becoming "trendy" can be considered positive as it contributes to raising public awareness about such issues as environmental sustainability and technological innovation. On the other hand, the smart phenomenon induces a form of disorientation for the abusive use of the term and the consequent dilution of its importance to research. [...] From our point of view, a less explored, and thus even more interesting frontier, is that which can be defined as Parametric Urbanism. In other words, the use of parametric software in urban design, not only to three-dimensionally represent projects at the urban scale [...], but precisely as part of the processes of developing the tools of urbanism, as an instrument for assisting the planner in evaluating diversified scenarios and making informed decisions. For example, it would be interesting to understand what contribution can be made by parametric tools to the construction of effective models of compensation (options on permutations, flexible distribution, etc.), or what assistance they can bring to the rationalisation of the layout of services within a territory, based on the real needs of users [...]. Or further still, the simulation of alternative scenarios to urban transformations based on a choice of diverse building typologies or densities of inhabitation.

1. Paolo Fusero, full Professor at the G. d'Annunzio University of Chieti-Pescara, Faculty of Architecture; Lorenzo Massimiano, researcher in Urbanism at the G. d'Annunzio University of Chieti-Pescara, Faculty of Architecture; Arturo Tedeschi, independent researcher, co-director since 2012 of the Architectural Association Rome Visiting School; Sara Lepidi, Graduate in "Territorial and Environmental Planning", 2012-13, G. d'Annunzio University.

From Typological to Procedural Thinking

The use of the computer in the world of design has accelerated a direction of research culturally rooted in the avant-gardes of the 1960s. This branch recently arrived at the elaboration of theoretical apparatuses constructed around a notion that compares architecture to systems in evolution and mechanisms of self-regulation. The research focuses substantially on the pragmatic passage from the concept of the *type* to one of *process*. This involves overcoming the logic of composition to the advantage of a "neo-positivist" vision founded on a multiplicity of interconnected elements (objects, materials, data). Through a propagation of effects, the variation
of one single element can bring about a modification to an entire architectural or urban organism. Hence the final form is an output generated by a procedure, almost as if it were unknown inherent to the system. Design is thus transformed into a sort of "definition of intelligent rules".

[...] The reciprocal fecundation between architectural theories and the possibilities offered by digital technologies consented the rapid extension of the utilisation of the computer. From a simple tool of production (focused on increasing the speed of operations) it has evolved into a refined system of control that permits previously unimagined formal explorations. The introduction of complex programming techniques and parametric software offers designers unexpected possibilities, making it almost impossible to predict the effects these tools will have on design simulations.

Simplifying to a great extreme, parametric software can be considered a programming platform – working within three-dimensional CAD environments – capable of generating form through the definition of a conceptual diagram that becomes the only "drawing" developed by the designer. This diagram explicates the associative ties between a range of input data, and generates an output that is a system of dynamic and modifiable forms.

From Reactivity to Proactivity

Networks of communication, sensors and *smart objects* are able to gather consistent masses of data. This data is in turn filtered through specific software created precisely to organise this material and facilitate its comprehension. A challenge to multinational digital companies of the future will lie precisely in the development of systems with an ability to define relations between heterogeneous data and create innovative forecasting models. Models will no longer be elaborated according to statistic methods, but instead through the *real-time* evaluation of significant parameters and indicators capable of influencing the design process at the urban scale.

For example, the overlapping reading of data as information alphabetisation or the offering of on-line services and relative user feedback, may suggest the territorial decentring of services that no longer require direct relations with users. To the same degree, data related to *co-working*, when compared to

correlated parameters, may offer important indications on urban mobility and energy consumption. Or, data from external sensors used to measure air quality, solar heat gain, ventilation, acoustic pollution, etc., may indicate solutions that optimise the energy efficiency and comfort of settlements. Within scenarios of this type, parametric software may even serve as a tool for experimenting with "new models of Urban Plans". No longer comprised solely of a series of "routine" drawings produced to satisfy normative requirements, they become a dynamic three-dimensional representation. These models can be constantly updated by *smart data*, which thus assumes a "proactive" role, anticipating phenomena and future changes in order to implement rapid and opportune actions and decisions. No longer a traditional "reactive" system controlled by mechanisms of consequential decision-making, but almost a new paradigm of planning supported

by a collective intelligence that is the fruit of choices, decisions and interactions supported and guided by technology. [...] The use of parametric software may thus offer designers a very interesting tool for experimenting with new methods of designing. Projects employing parametric logics are distinguished, in their form and content, from those developed according to traditional methods. The first important difference is conceptual, as mentioned: the final result is not established by the designer *a priori*, but is the result of a process of elaborating selected *smart data*. The second difference lies in the vivacity of the system that structures it: the passage from a static to a dynamic system. The formal result is no longer the definitive crystallisation of a particular line of reasoning, but instead a "snapshot" that captures the status of a process in continuous evolution. It is generated to react to variations, autonomously adapting to stimuli it receives in accordance with the rules established by the designer during the phase of concept design. Projects thus evolve on their own, almost demonstrating a capacity for *selforganisation*[2]. Despite their adaptive capacities, it is clear (and this is directed at those sceptics already thumbing their noses at the thought of substituting the designer during the "creative" process) that parametric platforms always require an *a priori* selection of data to be processed. It is precisely through the control of *input data* that designers are able to evaluate alternative solutions, utilising a "snapshot" of a *work in progress* to satisfy desired qualitative performance values. The phase of data selection and reactive control thus represents a crucial moment within the entire process [...].

2. Brian Team Consulting, "Teoria della complessità", from http://braint.net, last view on 5th July 2013.

Tools and methods for parametric urbanism

Andrea Galli
Architect at Carlo Ratti Associati

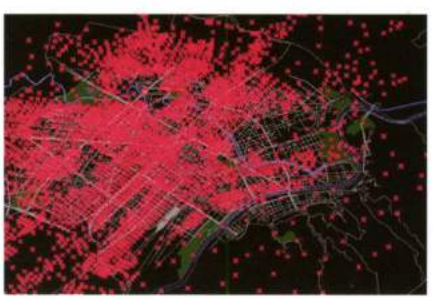
The modern city is a complex and dynamic system that, today, can benefit from the huge possibilities of technology scattered ubiquitously, changing the way the urban context talks to us, and how we live and interact within it. All this represents an incredible opportunity in planning the city of tomorrow, but it is not enough. To find the right answers to the questions that "big data" generates we can rely on the potential offered by parametric tools in elaborating – with total control– the new resources available, which would be unmanageable and redundant in traditional analysis and planning processes. In order to prepare our cities to become "Smart" it is necessary to adapt the design process that rules the change. In fact, many famous examples suggest the opposite, the parametric approach is not based on *shape*, but instead on the relationship between every elementary part of the complex system, where if an element changes, all the other elements will self-organize. This behaviour, called "adaptive" allows to design systems where – once the rules that describe the relationships between every element are set down – if the system changes unexpectedly (emergent behaviour[1]), it will transform itself according to these rules, defined as parametric variables. Parametric Urbanism allows us to understand and control the behavior of complex systems, such as our cities or parts of them, in order to plan his reaction to the real time change of the data context around it. To describe this context it is necessary to make a selection of data; this inherently represents an important design choice, because it deeply influences the final result. In addition to data typologies, the source of them is in the same way very important: data for example can be open-data released by public administrations, freely available to everyone; this data can be easily found on the Internet and embody the social and economic aim of unlocking the potential value of a huge quantity of information, usually under-used. Vector data related to a particular region of interest can be easily found on the portal *OpenStreetMap.org*[2], downloading the .osm file associated

1. Emergent behaviour: can appear when a number of simple entities operate in an environment, forming more complex behaviours as a collective.

2. Open Street Maps: is a collaborative project to create a free editable map of the world. The major driving forces behind the establishment and growth of OSM have been restrictions on use or availability of map information across much of the world and the advent of inexpensive portable satellite navigation devices.

to any geographical region. This file can be imported inside a Grasshopper algorithm using *Elk*[3] and it incorporates inside his framework metadata useful for a differentiated treatment of the geometries according to their respective tags. Some recurring tags are: *minor roads, major roads, waterways, railways, railway: station, highway: bus stop, highway: pedestrian, parking, buildings, amenity, leisure: park, leisure: garden, landuse: industrial, area, etc.*

This database can be furthermore increased with other information regarding, for example, the geographical distribution of people, the position of commercial hubs, schools, libraries, etc. These information come from the extended basin of geo referenced datasets published by public administrations and they are organized in tables or shapefile[4] which can be imported into Grasshopper, respectively through *Lunchbox* and *Finches*[5]. An exact overlapping can be accomplished using *gHowl*, thanks to the perfect correspondence with the coordinate system of the Open Street Maps geometries.

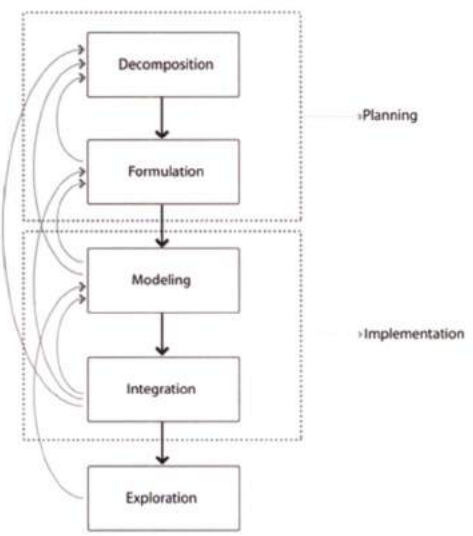


3. Elk: a plugin to generate map and topographical surfaces using open source data from OSM and Shuttle Radar Topography Mission, Timothy Logan, www.food4rhino.com/project/elk

4. Shapefile: or Esri shapefile, is a popular geospatial vector data format for geographic information system software. Shapefiles spatially describe vector features: points, lines, and polygons, representing, for example, water wells, rivers, and lakes.

5. Finches: a component developed by Nicholas De Monchaux to import, export and batch processing shapefiles.

The result is a hybrid model made of vector data and metadata that allows to exploit parametric modelling tools in association with a new informative dimension. A thinkable use of a certain model could be the visualization and the interrogation of the database, as with GIS. However it is possible to explore some more interesting direction, processing the database in order to generate an unlimited range of potential design proposal on different scales. Taking inspiration from the approach suggested by Anas Alfaris within his PhD thesis *Emergence through Conflict: The Multi-Disciplinary Design System (MDDS)*, developed at MIT under the supervision of W. J. Mitchell[6] in 2009, we can think to structure the entire design process using a dynamic and holistic framework divided in five phases according to reversible connections, but meanwhile, maintaining a robust hierarchic organization. The five phases are *decomposition, formulation, modeling, integration* and *exploration* and can be referred to different design stages: *decomposition* and *formulation* belong to the planning stage, *modeling* and *integration* belong to the implementation stage, the *exploration* represents the final verification stage where the previous phases can be validated or rejected. The *decomposition* requires to divide the global project vision into elementary units and goals. The *formulation* contemplates the specific analysis of every single goal so to independently define the processes to realize it, but also the way in which they influence each other. During *modelling* every goal is developed in a cluster in order to transform into algorithms (thanks to the parametric tools) what was defined during *formulation*. The *integration* consists in implementing the relationships between the different clusters, until now described as isolated elements. The *exploration* is the phase that allows to verify the match between results and what was expected during *formulation*. In fact, the *integration* can easily generate unexpected consequences, that will eventually require the process to be reviewed starting from *modelling*.



The described flow chart represents a structured way to manage the design process of complex entities such as our cities. Following it Grasshopper can assume the role of an extremely flexible platform to organize into a single data stream large scale data, which come from different sources, describing their relationship through mathematic rules but also counting on the huge 3D modelling possibilities offered by the Rhinoceros environment.

The opportunity of a real time comparison of many alternative scenarios, obtained by changing the mutual influence of the system parameters, is a scientific method for an organic design of our cities considering their real necessities and ensuring consistency with the project constrains and goals.

Andrea Galli

Andrea Galli, independent researcher, graduated in engineering and architecture from the Politecnico di Torino, collaborates with the office Carlo Ratti Associati of Turin from 2012.

6. William J. Mitchell: considered one of the world's leading urban theorists. Through the work of his Smart Cities research group at the MIT Media Lab, he pioneered new approaches to integrating design and technology to make cities more responsive to their citizens and more efficient in their use of resources. He likened tomorrow's cities to living organisms or very-large-scale robots, with nervous systems that enable them to sense changes in the needs of their inhabitants and external conditions, and respond to these needs. W. J. Mitchell died in 2010 at the age of 65.

Playful computation

How Grasshopper3D & its Plugins increased my creativity with five project examples

Arthur Mamou-Mani AAdip

Grasshopper3D for Rhinoceros is not just a tool, it is a platform for a community to learn and share computational design projects. It is an educational environment, which together with its numerous plugins will help you to understand mathematics, computer science, physics, fabrication and much more. It will break down barriers between design fields and soon you will be discussing vectors with a jewellery designer, topology with a wheel-chair fabricator or recursion with an engineer (true story). Grasshopper is complicated but it is also friendly and playful. Unlike most design software, it wants you to understand and learn how your tools and items can be linked together to create systems, instead of repetitive and time consuming 3D modelling (A.K.A. CAD monkeying). You will be linking components together with wires and not modelling curves and surfaces in space as you might be used to. Of course Grasshopper can simply accelerate your current workflow, but it can also transform the way you design in a much deeper way. You might have to imagine your project as a set of interconnected elements as opposed to a finished sculpted object. This will encourage you to see the reason behind forms and perhaps understanding these rules may even bring you closer to understanding how forms occur in nature. After doing a couple exercises within this book, you might see scary things like {0;0} N=1, or users around you could start asking if you have "flattened or grafted your list?" This is strange and unfamiliar territory for any early user, so ... DO NOT PANIC! We have all been through that phase of confusion. You will soon understand how the data trees will help you organise and manipulate the information going through the "fancy wires" and how you can apply successive operations to an initial input, which is the beginning of automating your tasks. Why are we all excited about this tool? It has opened up coding to visual people. It is a door to a complex world that we would have not dared to enter if Grasshopper was not here. Moreover it is a community of friendly people that you can find on the buzzing forum Grasshopper3d.com. Please note that if you ask for a new tool it is very likely that you will be asked to write it yourself and add it to the many plugins available on Food4Rhino.com. Users often write the plugins in their free time and out of interest, learning programming from nothing - so anyone can do it. This is the ultimate proof of Grasshopper's success, it grows with the users and soon you will be one of them, wanting to add your creative brick to the parametric building. Following are details of five projects I worked on and used Grasshopper within; to inspire you and show you how

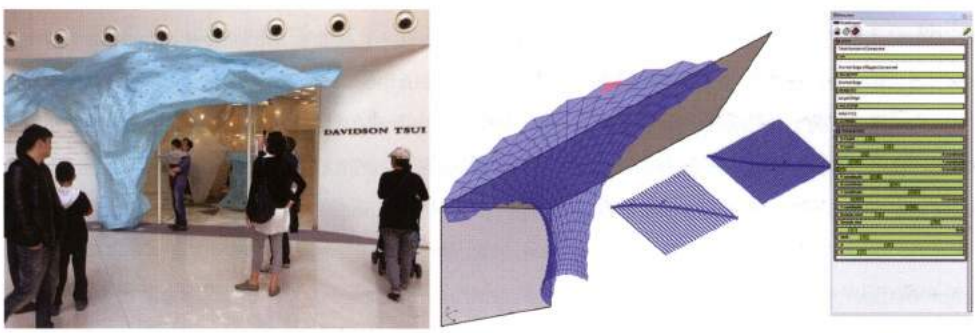
Grasshopper can be used in a practical way. Arturo Tedeschi, a teacher like myself, knows that you will probably not follow tutorials unless you are first inspired. You need to see not just the theory but how it can be applied, so let's look at these five projects of different scale and see how they benefited from the use of Grasshopper.

Project 1

RIBA Windows Project for Davidson Tsui in Xintiandi Style Shanghai[1]

with James K. Cheung of ARUP Associates

A parametric or associative approach to 3D modelling means that your model can be changed based on different input or parameters. By changing the parameters you are creating variations in the system not changing it. This makes Grasshopper3D very useful for quick changes in the geometry of a project or to produce many variations of a design system. This project illustrates how useful Grasshopper has been for these variations.

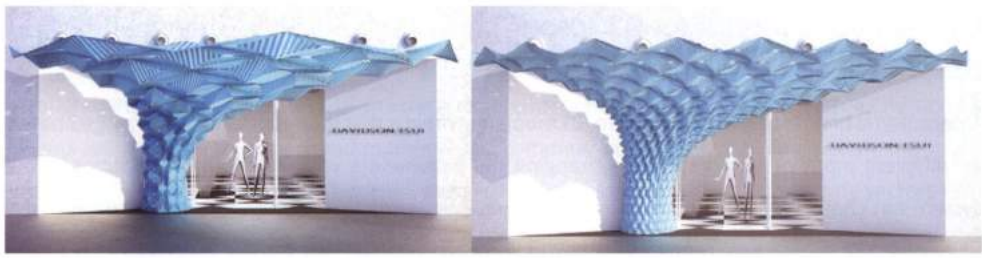


Left: the finished piece in Xintiandi Shanghai - Photo by XiaoHui Chen. Right: The parametric model for the main tree with the remote control panel for display outside the GH interface

The project's main element is a giant folded tree made of 576 laser-cut origami components. By folding the component, the sheet bends into a hyperbolic paraboloid (known as hypar). This is a ruled surface that can be described as two non-parallel lines in space connected with adjacent straight lines and forming two parabolas when cut in section. We used Grasshopper to populate the

1. The Fitting Room - More information: http://mamou-mani.com/RIBAShanghai/ - Client: Davidson Tsui and Shui-On Land - Team: Lead Architects: James K. Cheung (ARUP Associates London), Arthur Mamou-Mani (Mamou-Mani) - Architects: Suzanne Li (ARUP Associates Shanghai), Benita Tan (ARUP Associates Shanghai) Engineer: Eric Sturel (ARUP Associates Shanghai) Collaborators: Arup Associates London: Paul Jeffries, Daryl Miles, Stephen Philips – ARUP Associates Shanghai: Sunglin Tsai, Marta Colas, Milo Gu, Wonder Wu, Vicky Feng – Mamou-Mani: Laetitia Sfez – Tongji and Shanghai university: Gamzar Lee, Alexander Gösta, Tracy Zhang, Rachel Zheng, Zhang Licheng, Zhou Yejun, Liu Xun, Zheng Raven, Cao Sophie, Shi Ji, Chen Kaiyu, Xu Lei, Hu Zhixuan, Zhang Ying.

double-curved geometry of the tree with the folded hypars. To do so, we used the UV parameters of the surface but divided the resulting trimmed surfaces into sets of four in order to get mirroring components. To control the size of the panels, we changed the density of the division and used a custom "graph mapper" tool which projects values from the x axis to the y axis using a user-defined curve. This is how the gradient from small to large and from flat to folded was created.



One system with different output, two renders showing how the parametric model adapted to fit the fabrication.

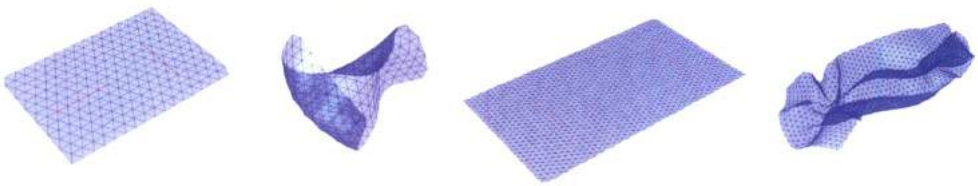
For fabrication, we were limited by the size of the machine used. At first the largest components were around 2m wide. This is possible with a large laser-cutter but in Shanghai we only had access to an A2-sized bed. Instead of drawing the whole project again we simply changed the density and the graph. The graph mapper is one of the GH components however it cannot be changed through sliders and therefore cannot be connected to the evolutionary solver "Galapagos". To make sure the longest edge would fit within an A2 laser-cut bed size we used Galapagos to change most of the sliders (UV density, graph mapper, distance of corner from original surface) until it met the required length of 594 mm.

Project 2
The Magic Garden, RIBA Regent Street Windows Project for Karen Millen[2]

The Magic garden - A 30 m long dress on Regent Street - Photo by Agnes Sanvito.

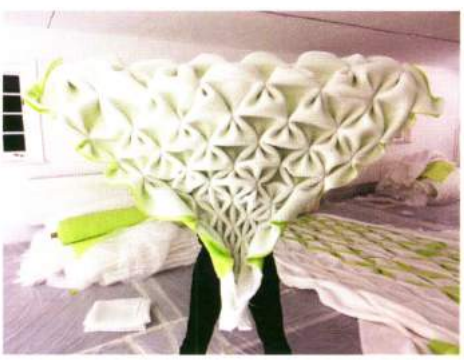
2. The Magic Garden – More information http://mamou-mani.com/KarenMillen/ – Client: Karen Millen Fashions Ltd.

The Magic Garden is a giant and architectural dress that flows along the thirty metres of Karen Millen's flagship store. The material is a cheap fabric that can be found in sport shoes called 3D Spacer Mesh. The mesh has a thickness that scatters light very well but also resists to bending. Both properties are used in this piece. The varying "smocking" pattern used all along the fabric creates different levels of strength and changes the width of the piece itself.



Pinching fabric using hinge forces in Kangaroo.

The physics engine Kangaroo (by Daniel Piker) was used to study the types of patterns that can be used. In this case the plugin "folds" the patterns from a flat sheet instead of drawing an additional geometry with extra area onto an existing surface. In a few words, Kangaroo works by plugging forces and anchor points into a physics engine. In this case we used the hinge force. Points will move in space constrained and defined by a rest angle. The anchor points can be moved to simulate someone pushing or pulling the fabric. When the anchor points move but the rest angle wants to stay at zero it creates a resistance, which is what was used here. Since we had several pinch points, we replaced anchor points by spring forces between two points to pinch together. Spring forces will move to a given rest length therefore the pinch points are linked with springs of a rest length of zero.



By changing the distance between the smocking patterns we could change the width of fabric.

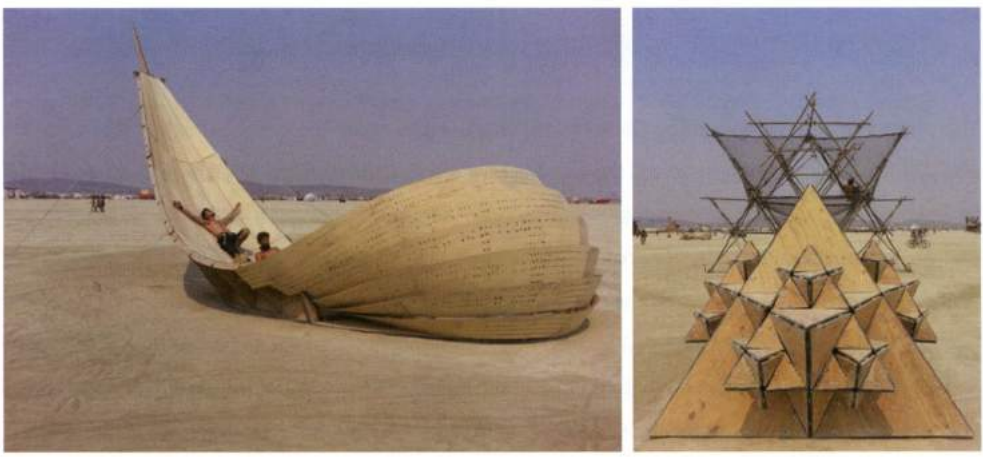
Contributors: Gemma Metheringham, Andrew Hardy Architect: Arthur Mamou-Mani Lead Collaborator: Jack Munro Collaborators: Savvas Havatzias, Madeeha Maham, Megan Sadler, Heloise Delegue, Anna Rootes, Sharon Toong, Sarah Shuttleworth, Dan Dodds, Andrei Jippa, Jessica Beagleman, Adam Holloway, Stephanie Holloway, Timothee Tan, Christina Leung, Philip Hurrel, Christopher Mount, Michael Clarke, Jacob Alsop, Nick Chung.

Project 3

Shipwreck and Fractal Cult at the Burning Man Festival[3]

with Toby Burgess and Diploma Studio 10 at the University of Westminster

These two projects were designed and built with our students at the University of Westminster. Grasshopper3D was used throughout. On Shipwreck the whole project was created from two initial NURBS curves on Rhinoceros, the top and bottom ones. The project's elegance is based on these two curves; they were therefore carefully changed in Rhinoceros while being used as input for the Grasshopper3D model. The fins flowing all along the structure were also changed within the parametric model and the whole cutting pattern changed accordingly.

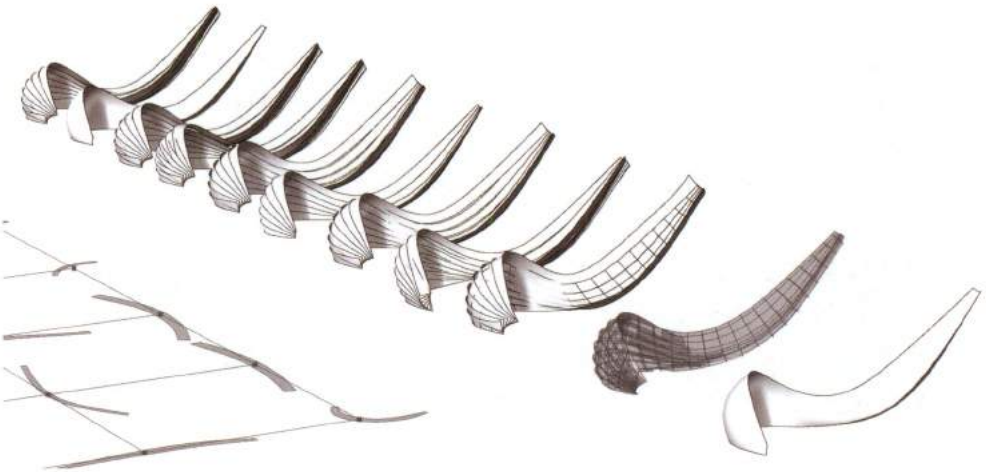


Shipwreck and Fractal Cult Designed by Georgia-Rose Collard-Watson and Thanasis Korras of Diploma Studio 10.

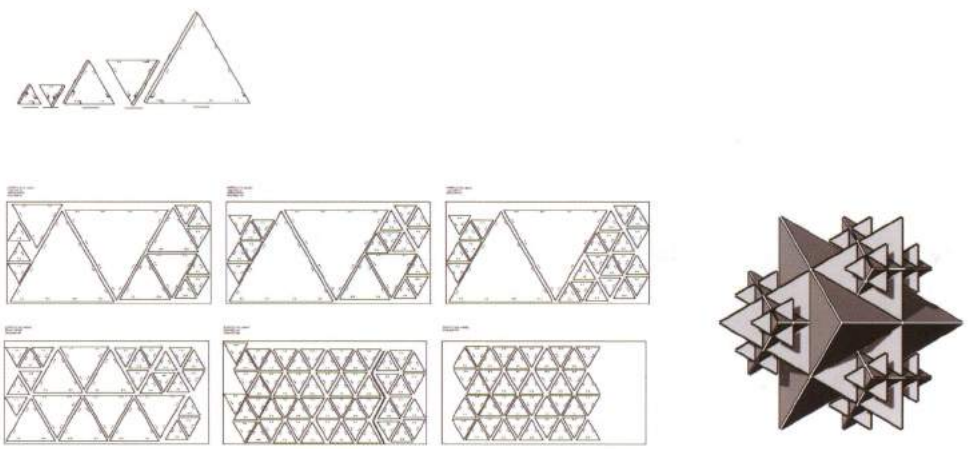
The form of Fractal Cult emerged from a three-dimensional Koch snowflake, which is well-known recursive fractal geometry. This was done using the plugin Hoopsnake, by Yannis Chatzikonstantinou, which allows the user to plug the end of a Grasshopper definition with its own beginning, hence the recursion. Once the triangles were generated we positioned hinges along the triangle edges so that as many triangles as possible could be used any way up and around. We then placed bolt holes for

3. Shipwreck and Fractal Cult at the Burning Man Festival More Information: http://wewanttolearn.wordpress.com/2013/09/12/building-fractal-cult-and-shipwreck-at-burning-man-2013/ Team: Toby Burgess and Arthur Mamou-Mani (Project Directors), Thanasis Korras (Designer of Fractal Cult), Georgia Rose Collard-Watson (Designer of Shipwreck), Jessica Beagleman (Food & Meals), Natasha Coutts (Camp and Rentals), Sarah Shuttlesworth, Andy Rixson, Luka Kreze, Tim Strnad, Philippos Philippidis, Nataly Matathias, Marina Karamali, Harikleia Karamali, Antony Joury, Emma Whitehead, , Jo Cook, Caitlin Hudson, Dan Dodds and Chris Ingram. Engineers: Ramboll Computational Design (RCD) – Stephen Melville, Harri Lewis, James Solly.

hinges with different offset according to whether the hinge is open at an obtuse or acute angle. This maintains a constant gap between the triangles throughout the whole structure. The definition then outputs 2D drawings for laser cutting.



Subtle variation of the fins in the shipwreck within the same parametric model.



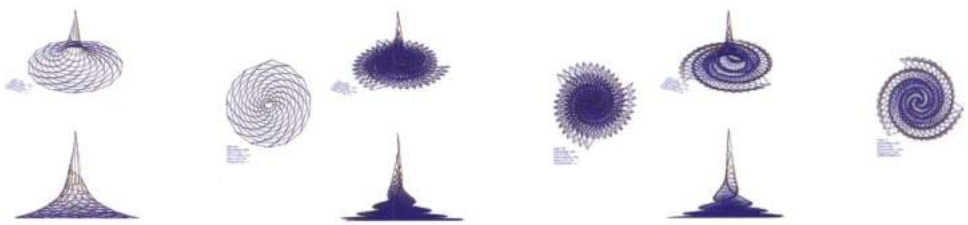
The acute and obtuse angles of the hinges defines the size of the triangles parametric model by Dan Dodds.

Project 4

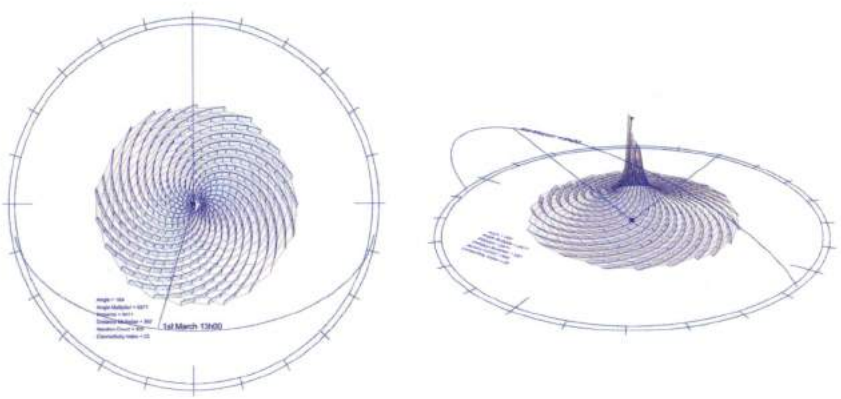
Eco-Resort in the desert of New-Mexico[4]

with Stephen Melville, Harri Lewis and Neil Clements of Ramboll RCD

Recursion is the process of repeating items in a self-similar way. For this project we moved and rotated vectors in space to form a roof that responds to sunlight and resembles a galaxy. One of the great aspects of Grasshopper3D is that you can link all kinds of tools and simulations to your parametric model directly. In this case, we linked the recursion to a sun angle that was imported to Grasshopper from Ecotect using the plugin Geco by Thomas Grabner and Ursula Frick. This allowed the roof to grow while making sure that sunlight stays out during summer and gets in during winter.



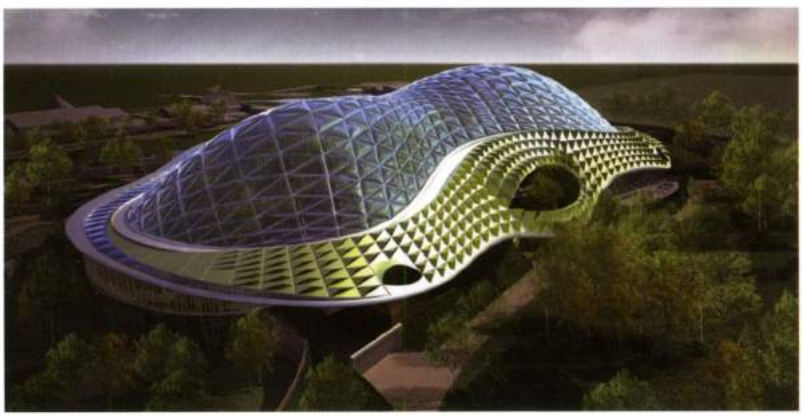
Recursive operation generating the roof mesh using Hoopsnake, slight variations in the angles and distances.

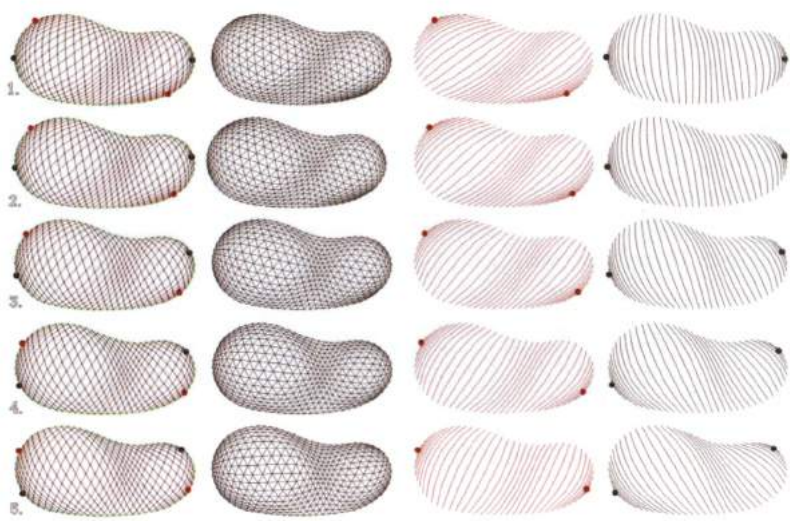


Letting light in winter and keeping it out in summer - Using Geco and Hoopsnake.

4. Eco-Resort in the desert of New-Mexico - More info:http://mamou-mani.com/ecoresort/ - Project by: Mamou-Mani Architects and Ramboll RCD.Team for Mamou-Mani: Arthur Mamou-Mani, Savvas Havatzias, Jack Munro Team for Ramboll RCD: Stephen Melville, Harri Lewis, Neil Clements (environmental engineer).

Project 5
Chester Zoo's Heart of Africa Biodome
by Proctor and Matthews Architects[5]





Chester Zoo is the first project for which I used Grasshopper. It is a 200 meters long biodome hosting animals and plants from central Africa. The directors Stephen Proctor and Andrew Matthews wanted

5. Chester Zoo by Proctor and Matthews Architect – More information: http://www.proctorandmatthews.com/project/heart-africa – Client: Zoological Gardens Chester Project Architect: Gareth Wilkins, Design Team: Stephen Proctor, Andrew Matthews, Stephanie Southward, Constance Leibrock, Benoit Sanson, Kengo Skorick, Arthur Mamou-mani, Steven Davies, Anna Marchant - Help on VB.net from Hyunbai Jun.

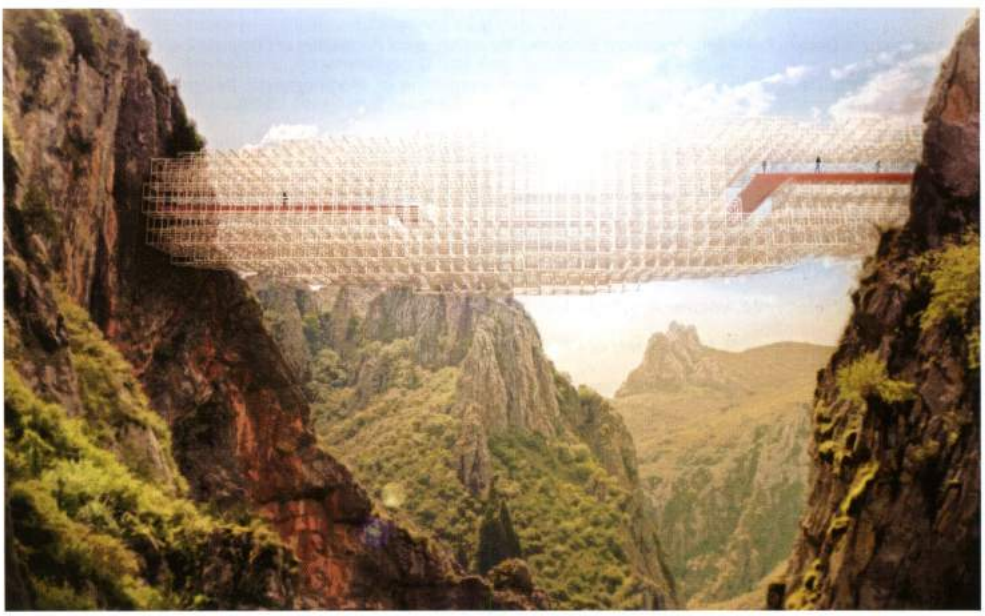
a pattern that would respond to the roof's geometry not just an orthogonal grid. We also needed to increase the greenhouse effect by making the openings very large and the structure very thin. Mark Cabrinha, on the Grasshopper3D.com forum, recommended using geodesic curves for the gridshell. These curves are the shortest path between two points on a surface. At first we populated the ring of the roof with points and connected them with these curves. Later on we generated a mesh from the intersecting points, which allowed us to organise all resulting triangles into separate branches. What worked really well is that the resulting structure was larger at the peaks and denser in the valleys. This made the structure very strong and convinced the engineers, who were advocating a more conventional truss and purlin option.

Throughout all these projects, Grasshopper3D with its plugins was used to define, generate, analyse, simulate or fabricate geometry. In all cases, it accelerated the workflow and created unique projects that would not have been created without this fertile platform. GH opens up a world of possibilities and connects designers, architects, engineers, fabricators and artists around topics that go beyond the tool itself. This book is your chance to go beyond the Voronoi, develop your left-brain and be a more complete professional. Now type "Grasshopper" in your command bar and enjoy!

Arthur Mamou-Mani is a registered architect and director of the chartered architecture and computational design practice Mamou-Mani Architects (http://mamou-mani.com). He is unit master of Diploma Studio 10 at the University of Westminster (http://WeWantToLearn.net) and advanced Grasshopper 3D tutor for Simply Rhino Ltd. (http://SimplyRhino.co.uk). Arthur has taught parametric design tools, digital fabrication as well as environmental and structural simulation at many leading academic bodies such as the Architectural Association School of Architecture and the UCL-Bartlett. He has previously worked with Atelier Jean Nouvel, Zaha Hadid Architects and Proctor and Matthews Architects for three years before setting up his own company.

The CloudBridge

"Most of the time, the fastest way to get anywhere is a straight line leading from point A to point B. But as a new conceptual project from Arturo Tedeschi architects, that's not necessarily the most efficient, or most beautiful, way to go about it. [...] CloudBridge links two mountainsides via an ethereal, cloud-like structure. Though bridges are often a visual statement of strength, CloudBridge's latticed form and non-linear path creates a super-stable bridge that appears to float between mountains, blending into its natural surroundings.



The CloudBridge, a concept developed by Arturo Tedeschi and Maurizio Degni.

The surreal concept, bolstered by extreme engineering, is a reflection of Tedeschi's work in general. "Nowadays engineering and architecture are evolving just by improving in small steps the 'state of art,' and designers are gradually losing their visionary attitude" he says. "This is also evident in technology, automotive and product design." Cloudbridge is meant to push the boundaries and be a playful look at how the trajectory and appearance of a bridge can be altered using parametric design".

(From "This cloud-Like bridge was created with an algorithm", WIRED digital magazine 10.04.2013)

References

THEORY

Barrios, C.R., *Parametric Design in Architecture: Fundamentals, Methods, Applications*, Birkhauser, 2014.

Chang, W., Vasilakos, A.V., *Molecular Computing: Towards a Novel Computing Architecture for Complex Problem Solving (Studies in Big Data)*, Springer, 2014.

Pilia, E.J., Fornasari, F., Verso, F., Boschi, L., Monopoli, D., Casolari, S., *Citymakers*, Deleyva, 2013.

AD. Architectural Design, *Computation Works: The Building of Algorithmic Thought*, Volume 222 (March/April 2013).

AD. Architectural Design, *Inside Smartgeometry: Expanding the Architectural Possibilities of Computational Design*, April 2013.

AD. Architectural Design, *Material Computation: Higher Integration in Morphogenetic Design*, Volume 216, No. 2 (March/April 2012).

Giussani, E., *Expressionist Organic Paths in Architecture: From Finsterlin to Gehry*, Lap Lambert Academic Publishing, 2012.

Pizzigoni, A., *Ingegneri e Archistar. Dialogo sul moderno costruire fra miti e mode*, Christian Marinotti Edizioni, Milano, 2011.

Coppola, C., *Attraverso l'Architettura, rappresentazione procedurale e algoritmi per il progetto generativo*, Alinea Editrice, Firenze, 2011.

DETAIL, *Analogue and Digital*, Volume 4, (July/August 2010).

Woodbury, R., Sheikholeslami, M., *Elements of Parametric Design*, Bentley Institute Press, 2010.

Schumacher, P., *The Autopoiesis of Architecture*, Voll. I and II, John Wiley & Sons, London, 2010.

Converso, S., *Il progetto digitale per la costruzione, cronache di un mutamento professionale*, Maggioli, Milano, 2009.

AD. Architectural Design, *Pattern*, Volume 79, No. 6 (November/December 2009).

AD. Architectural Design, *Digital Cities*, Volume 79, No. 4 (July/August 2009).

Otto, F., *Occupying and Connecting: Thoughts on Territories and Spheres of Influence With Particular Reference to Human Settlement*, Axel Menge, 2009.

Bechthold, M., *Innovative Surface Structures: Technology and Applications*, Taylor &Francis, 2008.

Balmond, C., *Informal*, Prestel, Munich, 2007.

Saggio, A., *Introduzione alla Rivoluzione informatica in Architettura*, Carocci, Roma, 2007.

Terzidis, K., *Algorithmic Architecture*, Architectural Press, Oxford, 2006.

AD. Architectural Design, *Techniques and Technologies in Morphogenetic Design*, Volume 76, No. 2 (March/April 2006).

Aranda, B., Lasch, C., *Tooling*, Princeton Architectural press, New York, 2006.

Coppola, C., *Computer e creatività per l'architettura, intelligenza artificiale e sistemi formali*, Alinea, Firenze, 2005.

AD. Architectural Design, *Emergence: Morphogenetic Design Strategies*, July 2004.

Koolhass, R., *Junkspace*, Quodlibet, 2001.

Thompson, D'Arcy W., *On Growth and Form*, Cambridge University Press, 1992.

ALGORITHMS

Wang, X., *Algorithmic Enhancements for Computing the Frame of a Finitely Generated Bounded Polyhedron*, Proquest, 2012.

Deb, K., *Multi-Objective Optimization using Evolutionary Algorithms*, Wiley, New York, 2001.

Goldberg, D., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison Wesley, New Jersey, 1989.

Aho, A., Ullman, J., Hopcroft, J., *Data Structures and Algorithms*, Addison Wesley, 1983.

DIGITAL FABRICATION

Jackson, P., *Cut and Fold Techniques for Pop-Up Designs*, Laurence King Publishing, London, 2014.

Jackson, P., *Cut and Fold Techniques for Promotional Materials*, Laurence King Publishing, London, 2013.

Beorkrem, C., *Material Strategies in Digital Fabrication*, Routledge, 2012.

Dunn, N., *Digital Fabrication in Architecture*, Laurence King Pub, 2012.

Nakamichi, T., Ito, C., Miyazaki, Y., *Pattern Magic: Stretch Fabrics*, Laurence King Pub, 2012.

Jackson, P., *Folding techniques for designers from sheet to form*, Laurence King Publishing, London, 2011.

Nakamichi, T., *Pattern Magic*, Laurence King Pub, 2010.

Nakamichi, T., *Pattern Magic 2*, Laurence King Pub, 2010.

Iwamoto, L., *Architectural and Material Techniques*, Princeton Architectural Press, New York, 2009.

Schodek, D., Bechthold, M., Griggs, J.K., Kao, K., Steinberg, M., *Digital Design and Manufacturing: CAD/CAM applications in architecture and design*, Wiley, 2004.

Wolff, C., *The Art of Manipulating Fabric*, Krause Pubns Inc., 1996.

FORM FINDING

Adriaenssens, S., Block, P., Veenendaal, D., Williams, C., *Shell Structures for Architecture*, Routledge, London, 2014.

Descamps, B., *Computational Design of Lightweight Structures: Form Finding and Optimization*, Wiley, 2014.

Gass, W.H., *Finding a Form: Essays*, Knopf, 2013.

Schatz, P., Carline, P., *Rhythm Research and Technology: The Invertible Cube/ Polysomatic Form-finding*, Niggli, 2013.

Knippers, J., Cremers, J., Gabler, M., Lienhard, J., *Construction Manual for Polymers + Membranes: Materials, Semi-Finished Products, Form-Finding, Design*, Birkhauser, 2012.

Pone, S., *Gridshell, i Gusci a graticcio in legno tra innovazione e sperimentazione*, Alinea, Firenze, 2012.

Kourkoutas, V., *Parametric Form Finding in Contemporary Architecture: The Simplicity Within The Complexity Of Modern Architectural Form*, Lambert, 2012.

Otto, F., Rasch, B., *Finding Form*, Deutscher Werkbund Bayern, 2006.

Guccione, M., *Il ponte e la città Sergio Musmeci a Potenza*, Gangemi, Roma, 2004.

Billington, D.P., *Thin Shell Concrete Structures*, McGraw Hill, New York, 1982.

Heyman, J., *Equilibrium of Shell Structures*, Clarendon Press, Oxford, 1977.

Kilian, A., Ochsendorf, J., "Particle-spring systems for structural form finding," *Journal of the International Association for Shell and Spatial Structures*, vol. 46, no. 148 (2005): 77-84.

SCRIPTING

Tang, M., *Parametric Building Design Using Autodesk Maya*, Routledge, 2014.

Pearson, M., *Generative art, a practical guide using processing*, Manning Publications Co., Shelter Island (NY), 2011.

Burry, M., "Scripting Cultures: Architectural Design and Programming," *AD. Architectural Design*, John Wiley & Sons, 2011.

Pisca, N., *YSYT Maya MEL Basics for Designers*, 2008.

Shiffman, D., *Learning processing*, Morgan Kaufmann, 2008.

Beri, M., *Python*, Apogeo, Milano, 2007.

ARCHITECTURAL GEOMETRY

Charleson, A., *Structure As Architecture: A Source Book for Architects and Structural Engineers*, Routledge, London, 2014.

Carrera, E., Briscetto, S., Nali, P., *Plates and Shells for Smart Structures: Classical and Advanced Theories for Modelling and Analysis*, Wiley, 2011.

Gjerde, E., *Origami Tessellations: Awe-inspiring Geometric Designs*, A.K. Peters Ltd., 2009.

Lopez, D., Ambrose, G., Fortunato, B., Ludwig, R., *The Function of Form Farshid Moussavi*, Actar, 2009.

Meredith, M., *From Control to Design: Parametric/Algorithmic Architecture*, Actar, 2008.

Pottmann, H., Asperl, A., Hofer, M., Kilian, A., *Architectural Geometry*, Bentley Institute Press, USA, 2007.

Bensøe, M., Sigmund, O., *Topology Optimization: Theory, Methods and Applications*, Springer Verlag, Berlin and Heidelberg, 2003.

Schock, H.J., *Soft Shells: Design and Technology of Tensile Architecture*, Birkhauser, 1997.

MATHEMATICS

Burry, J., Burry, M., *The New Mathematics of Architecture*, Thames & Hudson, 2012.

Rucker, R., *Infinity and the Mind: The Science and Philosophy of the Infinite*, Princeton University Press, 2004.

Frixione, M., Palladino, D., *Funzioni, Macchine, Algoritmi*, Carocci, Roma, 2004.

Saad, Y., *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, 2003.

Lorenz, E.N., *The Essence of Chaos*, University of Washington press, 1996.

MONOGRAPHS

Hiesinger, K.B., *Zaha Hadid: Form in Motion*, Yale University Press, 2011.

De Costa Meyer, E., *Frank Gehry: On Line (Princeton University Art Museum Series) (Princeton University Art Museum Monographs)*, Yale University Press, 2008.

Hadid, Z., Schumacher, P., *Latent Utopias: Experiments Within Contemporary Architecture*, Springer Verlag GmbH, NY, 2003.

Nicoletti, M., *Sergio Musmeci, organicità di forme e forze nello spazio*, Testo & Immagine, Collegno, 1999.

Faber, C., *Candela: The Shell Builder*, Reinhold Publishing Corporation, New York, 1963.

Decoded QR list

QR Code no. 1, p. 242: http://www.lepenseur.it/aad_book/chapter_05_01.jpg

QR Code no. 2, p. 304: http://www.lepenseur.it/aad_book/chapter_07_01.jpg

QR Code no. 3, p. 336: http://www.lepenseur.it/aad_book/chapter_08_01.jpg

QR Code no. 4, p. 390: http://www.lepenseur.it/aad_book/chapter_09_01.jpg

QR Code no. 5, p. 427: http://www.lepenseur.it/aad_book/chapter_10_01.jpg

QR Code no. 6, p. 429: http://www.lepenseur.it/aad_book/chapter_10_02.jpg

QR Code no. 7, p. 458: http://www.lepenseur.it/aad_book/chapter_11_01.jpg

QR Code no. 8, p. 465: http://www.lepenseur.it/aad_book/chapter_11_02.jpg

AAD_

ALGORITHMS-AIDED DESIGN

PARAMETRIC STRATEGIES USING GRASSHOPPER®

Main topics:

- Grasshopper environment and plug-in software
- Parametric modeling and advanced data management
- NURBS curves and surfaces
- Meshes and Subdivision Surfaces
- Digital fabrication techniques
- Recursions
- Form-finding strategies
- Particle-spring systems
- Topology Optimization
- Evolutionary solvers
- Environment-informed design

Algorithmic design is not simply the use of computer to design architecture and objects. Algorithms allow designers to overcome the limitations of traditional CAD software and 3D modelers, reaching a level of complexity and control which is beyond the human manual ability.

Algorithms-Aided Design presents design methods based on the use of Grasshopper®, a visual algorithm editor tightly integrated with Rhinoceros®, the 3D modeling software by McNeel & Associates allowing users to explore accurate freeform shapes. The book provides computational techniques to develop and control complex geometries, covering parametric modeling, digital fabrication techniques, form-finding strategies, environmental analysis and structural optimization. It also features case studies and contributions by researchers and designers from world's most influential universities and leading architecture firms.

Arturo Tedeschi is an architect and computational designer. Since 2004 he has complemented his professional career with an independent research on parametric design, focusing on relationships between architecture and digital design tools. He is the author of *Parametric Architecture with Grasshopper*, a best-selling book on parametric design, originally published in Italy and translated into English in 2012. He has taught at many universities, companies and institutions and from 2012 he is the co-director of the AA Rome Visiting School for the Architectural Association School (London). His work has been featured on a series of international magazines and exhibited in Rome, Milan, Venice, London and Paris. He has collaborated with leading architecture firms, including Zaha Hadid Architects, before setting up his consulting company A>T.

www.arturotedeschi.com

ISBN 978-88-95315-30-0

€ 40,00