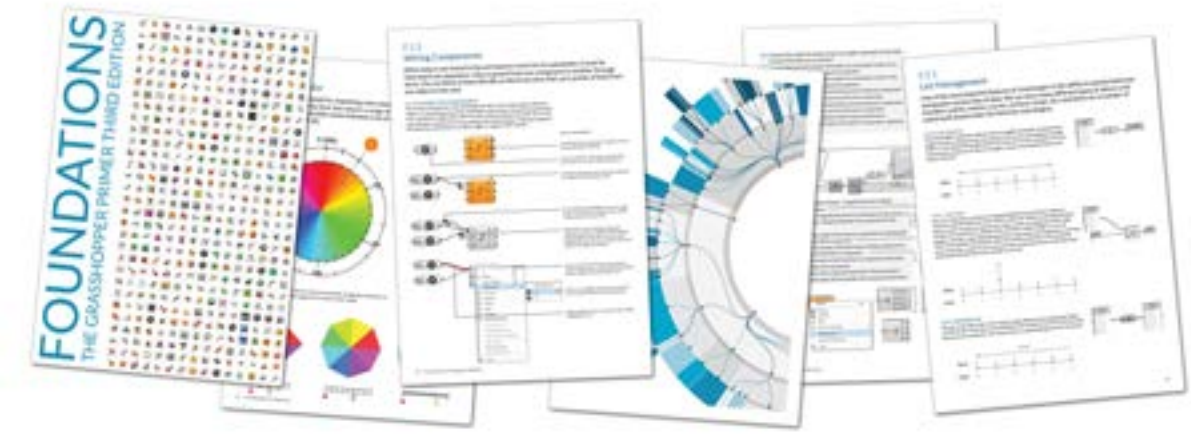
The Grasshopper Primer (DE)

Third Edition V3.3



> Grasshopper is a graphical algorithm editor tightly integrated with Rhino's 3-D modeling tools, allowing designers to build form generators from the simple to the awe-inspiring.

WELCOME

You have just opened the third edition of the Grasshopper Primer. This primer was originally written by Andrew O. Payne of Lift Architects for Rhino4 and Grasshopper version 0.6.0007 which, at the time of its release, was a giant upgrade to the already robust Grasshopper platform. We now find ourselves at another critical shift in Grasshopper development, so a much needed update to the existing primer was in order. We are thrilled to add this updated, *and now web-based*, primer to the many amazing contributions put forth by Grasshopper community members.

With an already excellent foundation from which to build, our team at Mode Lab went to work designing and developing the look and feel of the third edition. This revision provides a comprehensive guide to the most current Grasshopper build, version 0.90076, highlighting what we feel are some of the most exciting feature updates. The revised text, graphics, and working examples are intended to teach visual programming to the absolute beginner, as well as provide a quick introduction to Generative Design workflows for the seasoned veteran. It is our goal that this primer will serve as a field guide to new and existing users looking to navigate the ins and outs of using Grasshopper in their creative practice.

This primer introduces you to the fundamental concepts and essential skill-building workflows to effectively use Grasshopper. Foundations is the first volume in a forthcoming collection of Grasshopper primers. Here's what you can expect to learn from the primer:

- **Introduction** - What is Grasshopper and how is it being used?
- **Hello Grasshopper** - Make your first definition
- **Anatomy of a Grasshopper Definition** - What makes up a definition?
- **Building Blocks of Algorithms** - Start simple and build complexity
- **Designing with Lists** - What's a list and how do I manage them?
- **Designing with Data Tree** - What's a data structure and what do they mean for my process?

- **Appendix** - References and Working files for continued exploration

We hope that at the very least this primer will inspire you to begin exploring the many opportunities of programming with Grasshopper. We wish you the best of luck as you embark on this journey.

THE GRASSHOPPER PRIMER PROJECT

The Grasshopper Primer is an open source project, initiated by Bob McNeel, Scott Davidson, and the Grasshopper Development team at Robert McNeel & Associates.

Mode Lab authored the Third Edition of the primer. http://modelab.is

If you would like to contribute to this project, check out the github project wiki to get started (https://github.com/modelab/grasshopper-primer/wiki).

ACKNOWLEDGEMENTS

Aspecial thanks to David Rutten for the endless inspiration and invaluable work pioneering Grasshopper. We would also like to thank Andrew O. Payne for providing the assets from which this work initiated. Lastly, many thanks to Bob McNeel and everyone at Robert McNeel & Associates for their generous support over the years.

REQUIRED SOFTWARE

Rhino5

Rhino 5.0 is the market leader in industrial design modeling software. Highly complicated shapes can be directly modeled or acquired through 3D digitizers. With its powerful NURBS based engine Rhino 5.0 can create, edit, analyze, and translate curves, surfaces, and solids. There are no limits on complexity, degree, or size.

http://www.rhino3d.com/download/rhino/5/latest

Grasshopper

For designers who are exploring new shapes using generative algorithms, Grasshopper is a graphical algorithm editor tightly integrated with Rhino's 3D modeling tools. Unlike RhinoScript or Python, Grasshopper requires no knowledge of the abstract syntax of scripting, but still allows designers to build form generators from the simple to the awe inspiring.

http://www.grasshopper3d.com/page/download-1

FORUMS

The Grasshopper forum is very active and offers a wonderful resource for posting questions/answers and finding help on just about anything. The forum has categories for general discussion, errors & bugs, samples & examples, and FAQ.

http://www.grasshopper3d.com/forum

The Common Questions section of the Grasshopper site contains answers to many questions you may have, as well as helpful links:

http://www.grasshopper3d.com/notes/index/allNotes

For more general questions pertaining to Rhino3D be sure to check out the McNeel Forum powered by Discourse.

http://discourse.mcneel.com/

LICENSING INFORMATION

Inhaltsverzeichnis

0. About
    0.1. Grasshopper - an Overview
    0.2. Grasshopper in Action
1. Foundations
    1.1. Hello Grasshopper!
        1.1.1. Installing and Launching Grasshopper
        1.1.2. The Grasshopper UI
        1.1.3. Talking to Rhino
    1.2. Anatomy of a Grasshopper Definition
        1.2.1. Grasshopper Object Types
        1.2.2. Grasshopper Component Parts
        1.2.3. Data Types
        1.2.4. Wiring Components
        1.2.5. The Grasshopper Definition
    1.3. Building Blocks of Algorithms
        1.3.1. Points Planes & Vectors
        1.3.2. Working With Attractors
        1.3.3. Mathematics, Expressions & Conditionals
        1.3.4. Domains & Color
        1.3.5. Booleans and Logical Operators
    1.4. Designing with Lists
        1.4.1. Curve Geometry
        1.4.2. What is a List?
        1.4.3. Data Stream Matching
        1.4.4. Creating Lists
        1.4.5. List Visualization
        1.4.6. List Management
        1.4.7. Working with Lists
    1.5. Designing with Data Trees
        1.5.1. Surface Geometry
        1.5.2. What is a Data Tree?
        1.5.3. Creating Data Trees
        1.5.4. Working with Data Trees
    1.6. Getting Started with Meshes
        1.6.1. What is a Mesh?
        1.6.2. Understanding Topology
        1.6.3. Creating Meshes
        1.6.4. Mesh Operations
        1.6.5. Mesh Interactions
        1.6.6. Working with Mesh Geometry
2. Extensions
    2.1. Element*
        2.1.1. Introduction
        2.1.2. Half Edge Data
        2.1.3. Components
        2.1.4. Exercise
        2.1.5. Architectural Case Study

3. Appendix
    3.1. Index
    3.2. Example Files
    3.3. Resources
    3.4. About This Primer

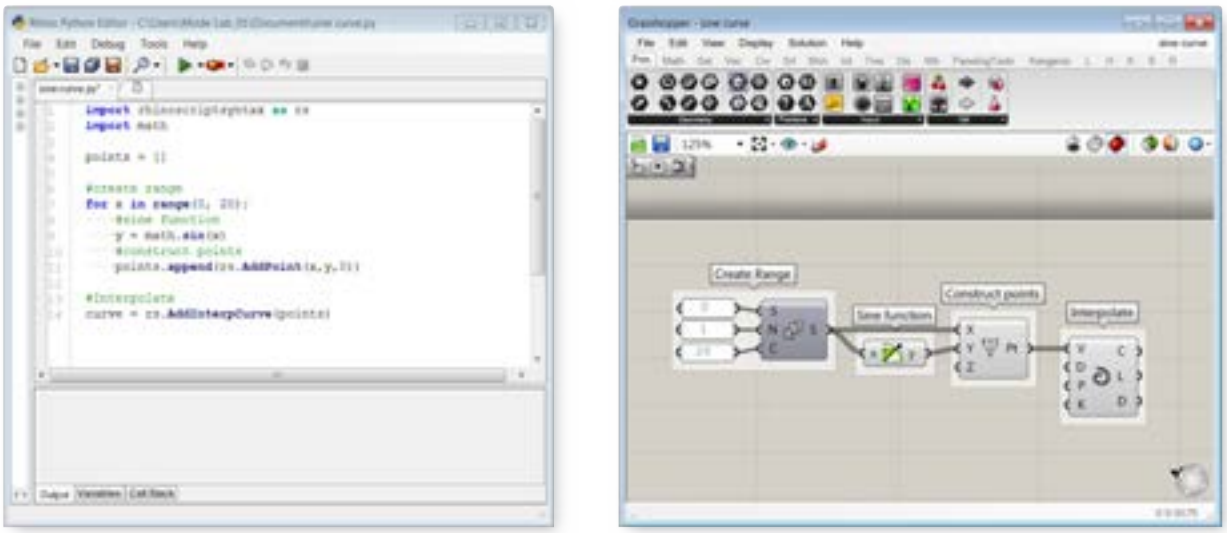
Grasshopper - Eine Uebersicht

Grasshopper ist ein visueller Programmeditor, der von David Rutten und Robert McNeel & Associates entwickelt wurde. Als Plug-In fuer Rhino3D, ist Grasshopper integriert in eine robuste und vielseitige Modellierungsumgebung. Diese wird von kreativen Profis in verschiedenen Bereichen, wie z.B. Architektur, Ingenieurwissenschaften und Produktdesign genutzt. Zusammen mit Rhino bietet Grasshopper uns die Moeglichkeit praezise parametrische Beziehungen in Modellen zu definieren, die Faehigkeit generative Entwurfsablaeufe zu erkunden und eine Plattform um uebergeordnete Programmierlogik zu entwickeln – und all das in einer intuitiven, graphischen Oberflaeche.

Die Urspruenge von Grasshopper koennen bis zur Funktionalitaet des "Record History" Knopfes in Rhino3d Version 4 zurueckverfolgt werden. Dieses spezifische Merkmal erlaubte es Anwendern Modellierungsverfahren waehrend des Ablaufes implizit im Hintergrund zu speichern. Falls Du aus vier Kurven mit der eingeschalteten Aufzeichnungsfunktion eine Loftflaeche erstellt hattest, konntest Du die Kontrollpunkte der Kurven bearbeiten und die Geometrie der Flaeche wurde automatisch aktualisiert. Im Jahr 2008 hat David folgende Frage gestellt: “Was wuerde geschehen, wenn man mehr explizite Kontrolle ueber die Aufzeichnungsfunktion haette?”. Dies war der Geburtsmoment fuer den Vorlaeufer von Grasshopper - Explicit History. Diese Funktion erlaubte es die gespeicherten Arbeitsablaeufe im Detail zu bearbeiten und befaehigte den Anwender logische Sequenzen zu entwickeln, welche ueber die bestehenden Faehigkeiten von Rhino3Ds eingebaute Funktionalitaet hinausreichen. Sechs Jahre spaeter ist Grasshopper nun eine robuster, visueller Programmiereditor, der mit verschiedenen extern entwickelten Plug-Ins erweitert werden kann. Ausserdem wurden dadurch Arbeitsablaeufe von Profis in verschiedenen Industrien grundsaetzlich veraendert und eine aktive, globale Gemeinschaft der Anwender beguenstigt.

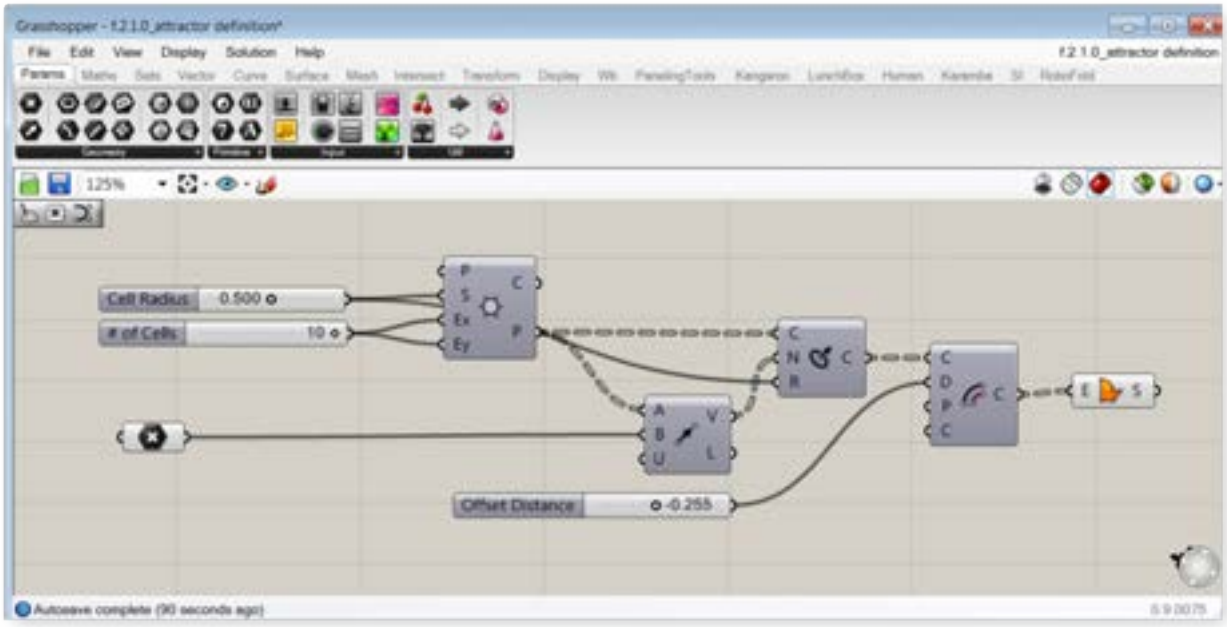
Dieser Primer legt seinen Fokus sowohl auf die Grundlagen um Kernwissen anzubieten, das Du brauchst um in eine regelmaessige Nutzung von Grasshopper einzutauchen, als auch auf einige Sprungbretter fuer die naechsten Schritte in Deiner eigenen kreativen Praxis. Bevor wir in die Beschreibungen, Diagramme und Beispiele einsteigen, welche in der Folge dargeboten werden, sollten wir darueber sprechen, was visuellen Programmierung ist, wie die Grasshopper Benutzeroberflaeche aufgebaut ist, welche grundlegenden Begriffe Grasshopper nutzt und welche "live" Eigenschaften die Rueckkopplung des Ansichtsfensters bzgl. des Benutzererlebnisses hat.

Visueller Programmierung ist ein Paradigma der Computerprogrammierung innerhalb dessen der Benutzer logische Elemente graphisch statt textbasiert manipuliert. Einige der bekanntesten text-basierten Programmiersprachen, wie C#, Visual Basic oder Processing – und naeher an Rhinoceros – Python und Rhinoscript, verlangen von un, dass wir Code schreiben, der an sprachspezifische Syntax gebunden ist. > Im Unterschied dazu erlaubt uns visuelles programmieren funktionale Bloecke in einer Sequenz von Aktionen miteinander zu verbinden. Dabei ist die einizig benoetigte “Syntax” dass die Eingabeparameter Daten des entsprechenden Typs bekommen, und idealerweise, dass das Skript auf das beabsichtigte Resultat hin organisiert ist. Mehr dazu in den Abschnitten ueber die Abgleichung des Datenflusses und das Entwerfen mit Baumstrukturen. Diese Charakteristik des visuellen programmierens senkt die Einstiegsbarriere, welche ueblicherweise bei dem Erlernen einer neuen Sprache - auch bei einer gesprochenen - vorgefunden wird. In den Vordergrund zu stellen ist auch die graphische Benutzeroberflaeche, die Grasshopper fuer Designer in einem vertrauten Umfeld verortet.



> Dieses Bild zeigt die Entwicklung eine gezeichneten Sinuskurve in Python und Grsshopper.

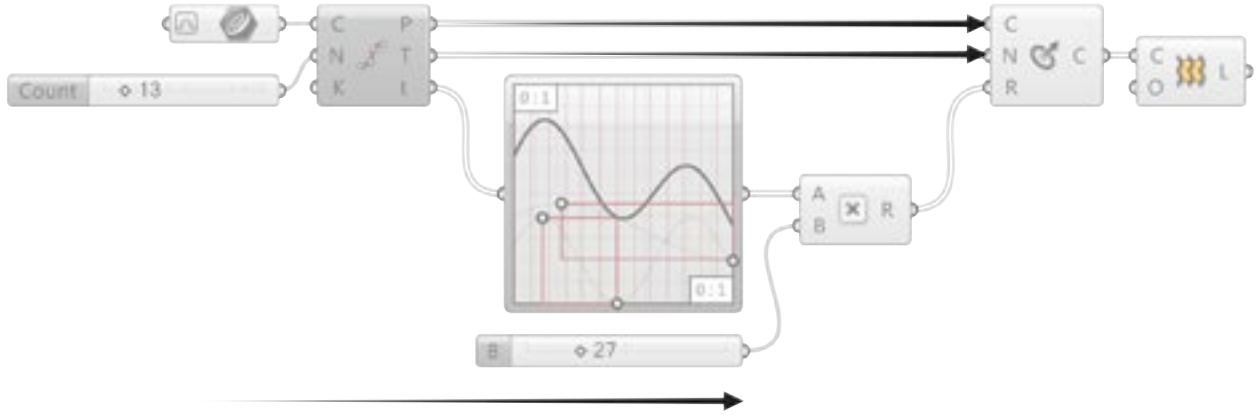
Um bei Grasshopper einzusteigen und seine Faehigkeiten bezueglich visuellen programmierens nutzen zu koennen, muessen wir das Programm von der Grasshopper3D.com Webseite laden und auf unserem Computer installieren. Einmal installiert, koennen wir das das Plug-In oeffnen, indem wir “Grasshopper” in die Rhino Befehlszeile eingeben. Das erste Mal wenn wir diese Eingabe in einer neuen Rhinositzung machen, werden wir die Grasshopper Ladeanzeige sehen, bevor wir das Fenster des Grasshoppereditors sehen. Wir koenen nun funktionale Bloecke – “Komponenten” genannt- auf der “Leinwand” einfuegen, mit Kabeln verbinden und die gesamte “Definition” im .ghx Dateiformat speichern.



> Eine Grasshopperdefinition, bestehend aus Komponenten die mit Kabeln auf der Leinwand verbunden sind.

Sobald wir begonnen haben eine Grasshopperdefinition zu entwickeln und Schieberegler in unserer Leinwand erstellt haben um Geometrien zu kontrollieren, koennen wir unsere natuerliche Intuition nutzen um Verbindungen zwischen den Komponenten zu verstehen, da die Ergebnisse direkt im Rhinosansichtsfenster sichtbar werden. Die Verbindung zwischen Grasshopper und Rhino ist grundsaetzlich in Echtzeit – wenn wir einen Schieberegler veraendern, wird die Folge dieser Handlung im Rahmen der Definition ausgewertet und die Loesung neu berechnet, bevor das Ansichtsfenster aktualisiert wird. Beim Einstieg in Grasshopper ist es von Vorteil, dass die

Vorschau der Geometrie, die wir sehen eine leichtgewichtige Darstellung der Loesung ist und diese automatisch aktualisiert wird. Es ist jetzt wichtig zu wissen, dass sobald Deine Definitionen komplexer werden und auf geschickte Art und Weise den Datenfluss regeln, das Rhinoansichtsfenster den Status der Rechnerloesung wiederspiegeln, um ungewollte Kopfschmerzen zu vermeiden.



> Programmfluss von links nach rechts.

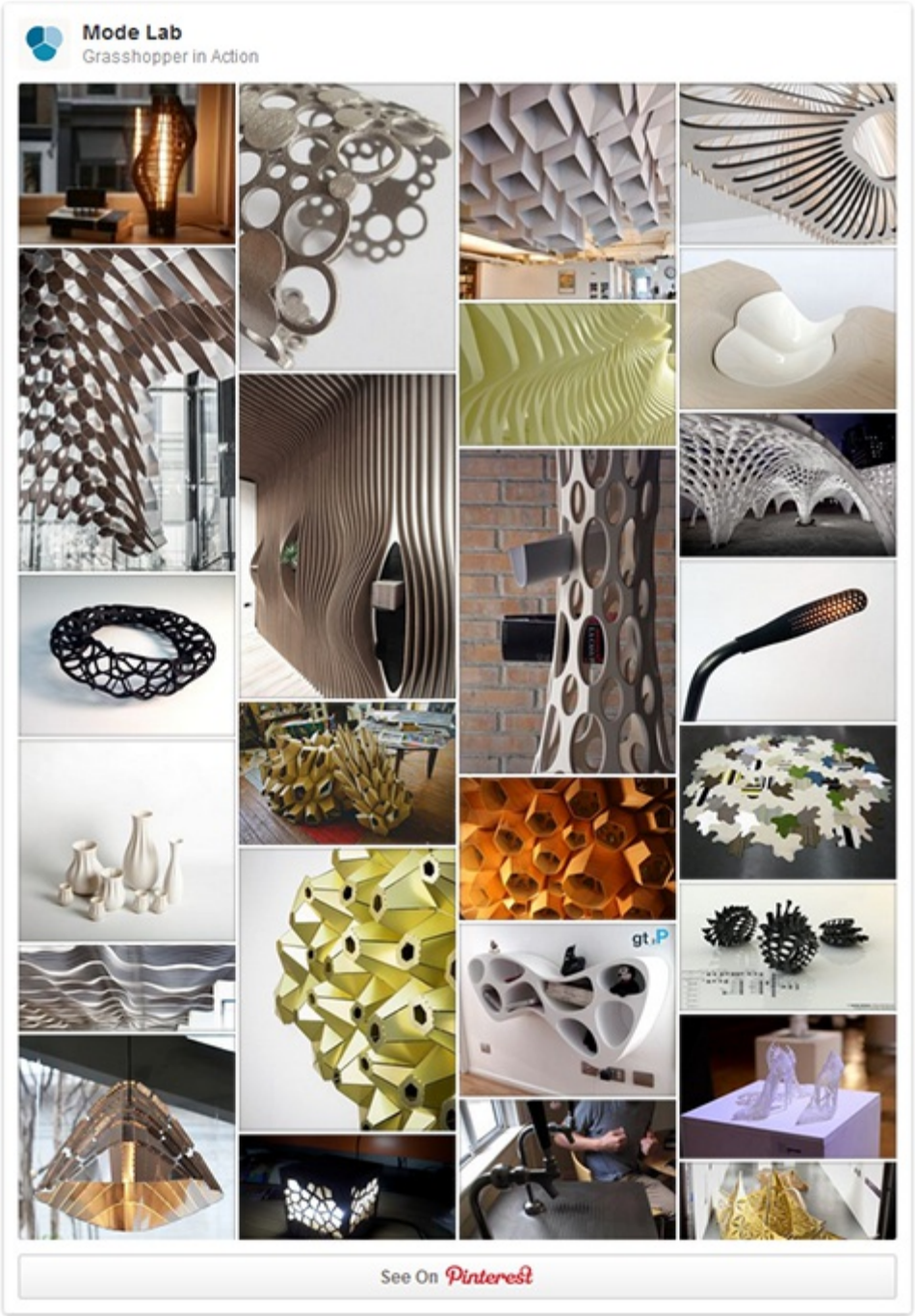
MERKPUNKTE

- Grasshopper ist ein grafischer Algorithmeneditor, der in das Rhino3D Modellierwerkzeug integriert ist.
- Algorithmen sind schrittweise Prozeduren, die entworfen wurden um eine bestimmte Operation durchzufuehren.
- Du benutzt Grasshopper um Algorithmen zu entwerfen, die dann Aufgaben in Rhino3D automatisieren.
- Ein einfacher Weg um einzusteigen, wenn Du Dir nicht sicher bist, wie eine bestimmte Operation in Grasshopper ausgefuehrt werden kann, waere es den Algorithmus haendisch und schrittweise mit Rhinobefehlen zu erarbeiten.

Sobald Du erstmals beginnst Grasshopper zu erkunden oder Deine Faehigkeiten weiter entwickelst, bist Du ein Teil der weltweiten Grasshoppergemeinschaft. Diese Gemeinschaft besteht aus aktiven Mitgliedern aus vielen verschiedenen Anwendungsgebieten und vielfaeltigen Erfahrungsniveaus. Das Forum auf Grasshopper3D.com ist eine nuetzliche Quelle um Fragen zu stellen, Ergebnisse zu teilen und Wissen zu sammeln. Diese Gemeinschaft hat uns waehrend des Schreibens des Primers sehr geholfen und wir haben uns daran erfreut, wie sich Grasshopper ueber die Jahre entwickelt hat. Herzlich Willkommen!

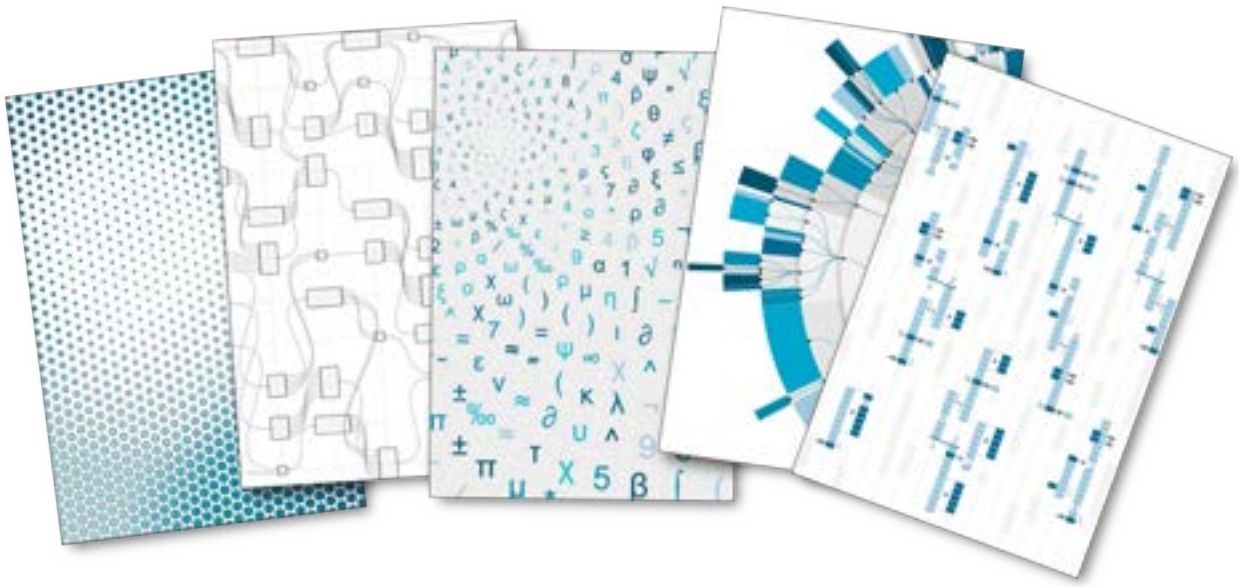
Grasshopper in Aktion

Folge der Grasshopper in Aktion Pinnwand auf Pinterest.



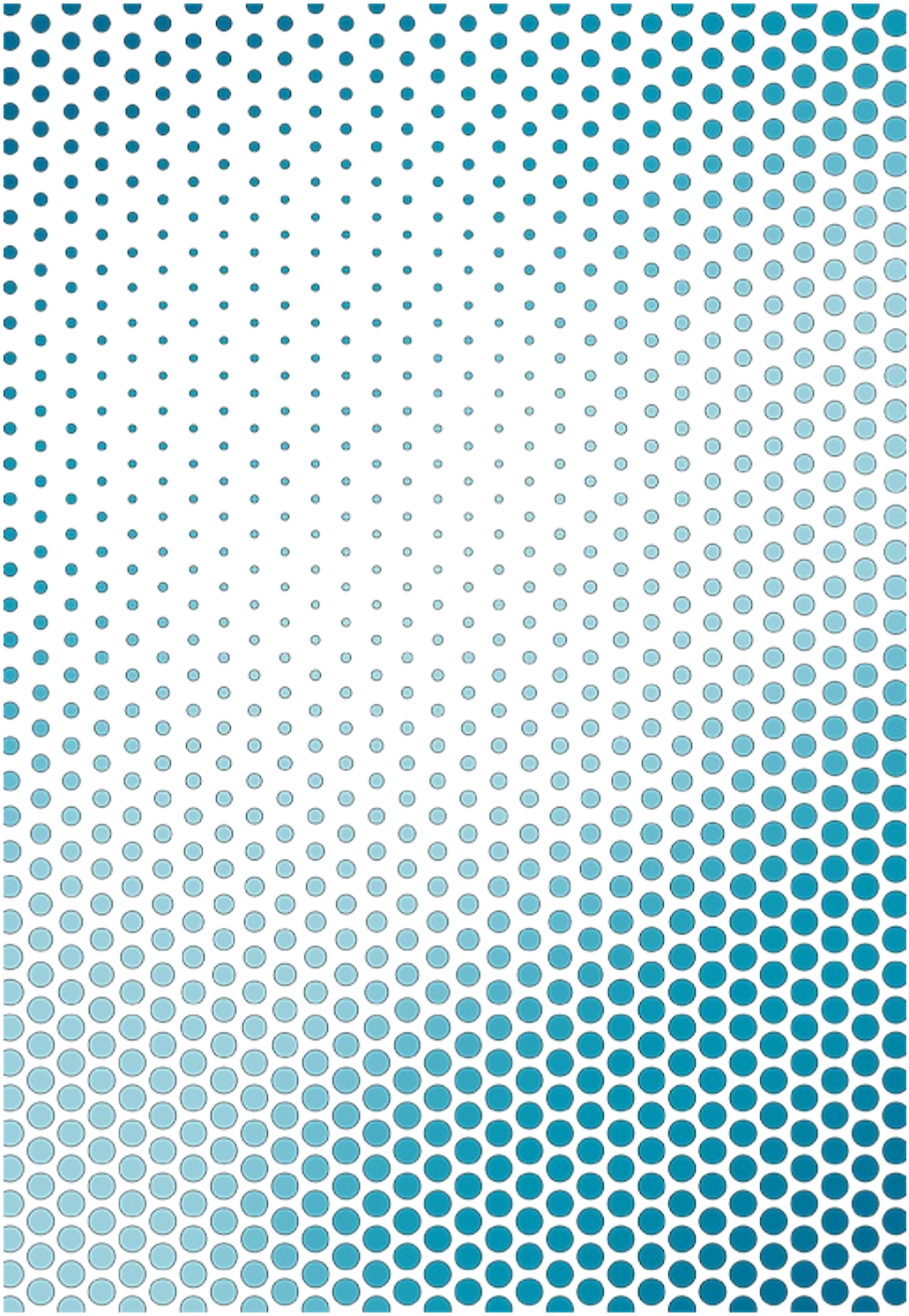
1. FUNDAMENTE

Ein starkes Fundament ist fuer die Ewigkeit gebaut. Dieser Teil des Primers fuehrt Schluesselkonzepte und -eigenschaften des parametrischen Modellierens mit Grasshopper ein.



1.1. HELLO GRASSHOPPER

Grasshopper is a graphical algorithm editor that is integrated with Rhino3D's modeling tools. You use Grasshopper to design algorithms that then automate tasks in Rhino3D.



1.1.1. INSTALLING AND LAUNCHING GRASSHOPPER

The Grasshopper plugin is updated frequently so be sure to update to the latest build.

Note that there is currently no version of Grasshopper for Mac.

1.1.1.1. DOWNLOADING

To download the Grasshopper plug-in, visit the Grasshopper web site. Click on the Download tab at the top of the page, and when prompted on the next screen, enter your email address. Now, right click on the download link, and choose Save Target As from the menu. Select a location on your hard drive (note: the file cannot be loaded over a network connection, so the file must be saved locally to your computer's hard drive) and save the executable file to that address.

> Download Grasshopper from the grasshopper3d.com website.

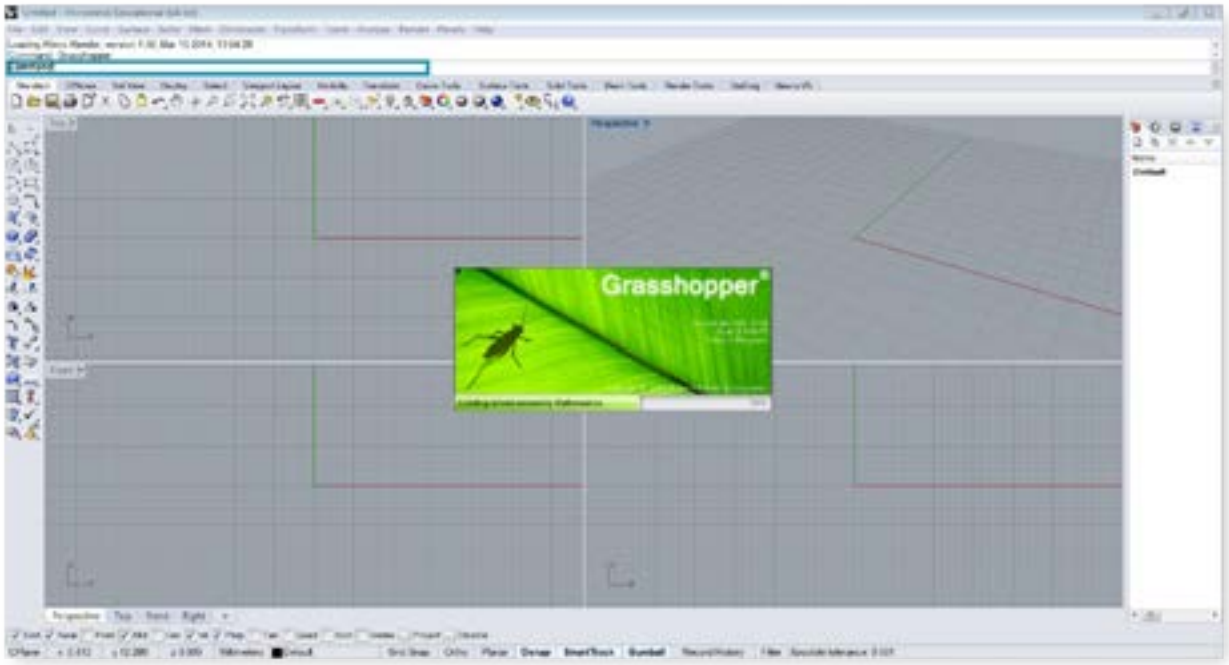
1.1.1.2. INSTALLING

Select Run from the download dialog box follow the installer instructions. (note: you must have Rhino 5 already installed on your computer for the plug-in to install properly).

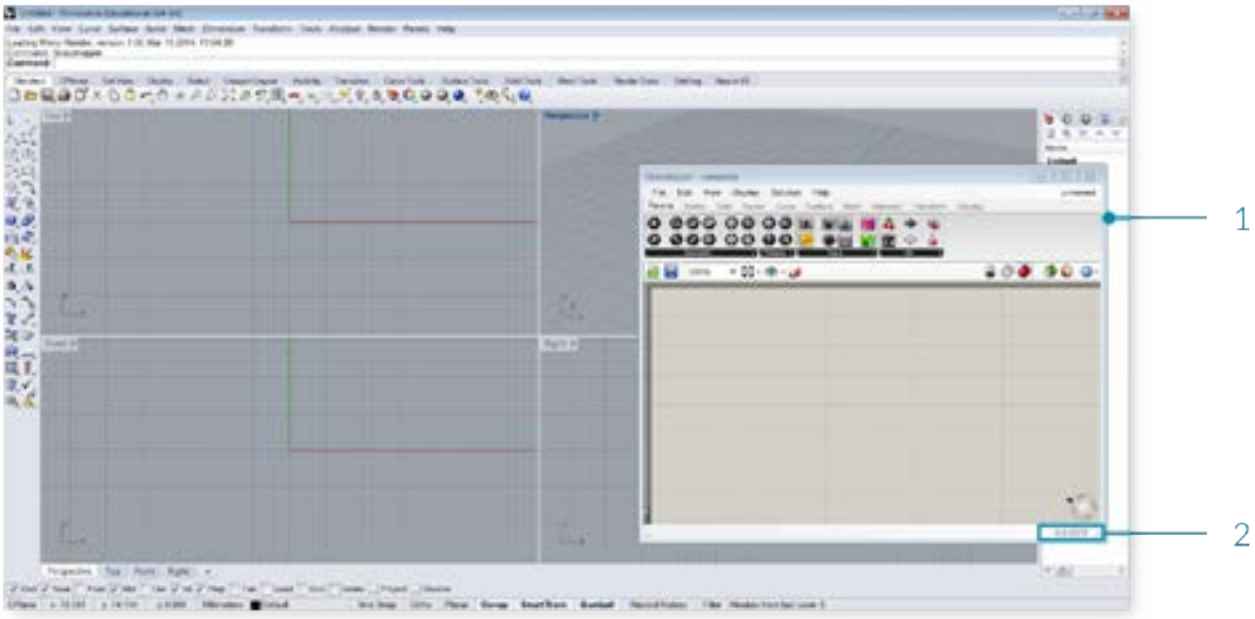
> Follow the steps in the Installation wizard.

1.1.1.3. LAUNCHING

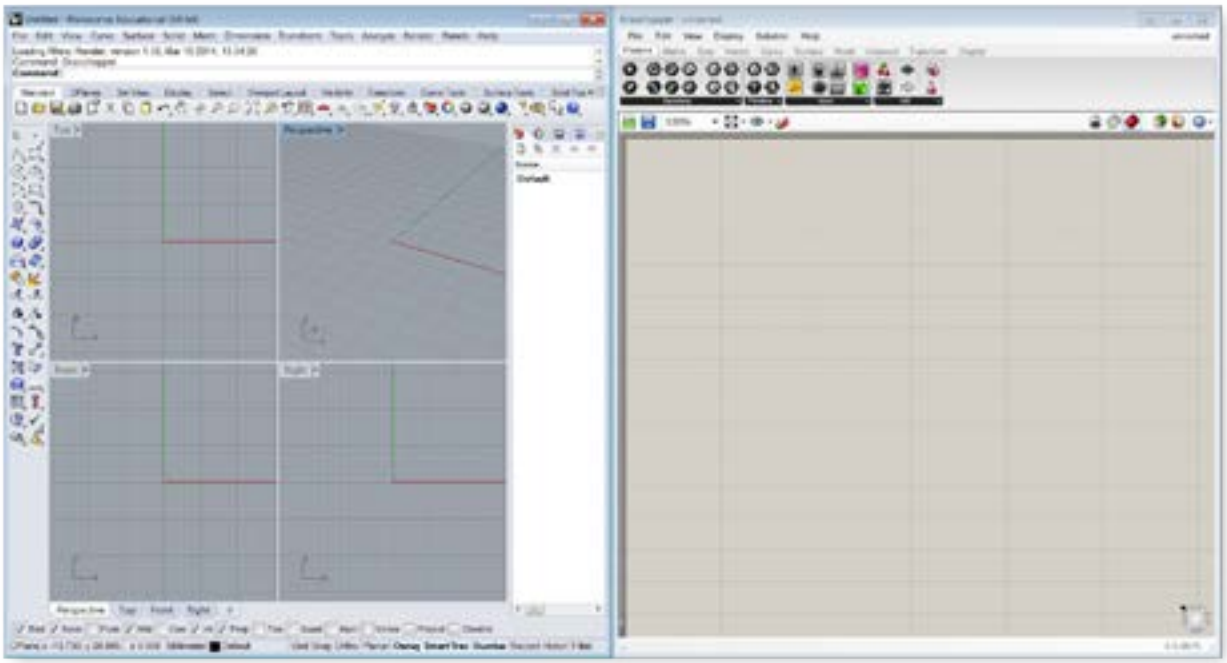
To Launch Grasshopper, type Grasshopper into the Rhino Command line. When you launch Grasshopper, the first thing you will see is a new window floating in front of Rhino. Within this window you can create node-based programs to automate various types of functionality in Rhino. Best practice is to arrange the windows so that they do not overlap and Grasshopper does not obstruct the Rhino viewports.



> Type “Grasshopper” into the Rhino command line to launch the Grasshopper plugin.



1. The Grasshopper window floats on top of the Rhino viewports.
2. Grasshopper displays the version number at the bottom of the window.



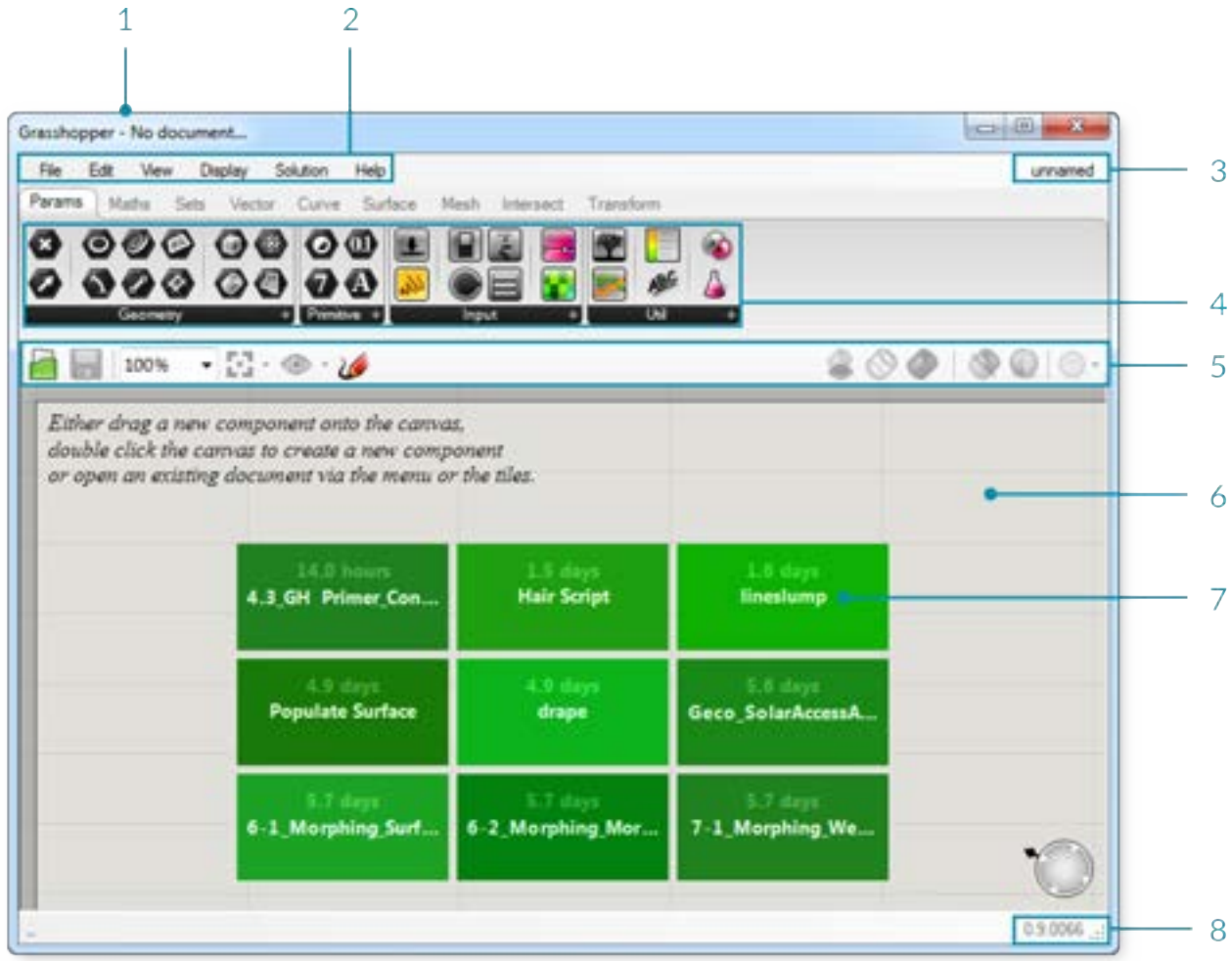
Split the screen so that Grasshopper does not obstruct the Rhino Viewports. You can do this by dragging each window to opposite sides of the screen, or by holding the Wondows key and pressing the left or right arrows.

1.1.2. THE GRASSHOPPER UI

Grasshopper's visual "plug-and-play" style gives designers the ability to combine creative problem solving with novel rule systems through the use of a fluid graphical interface.

Let's start by exploring Grasshopper's user interface UI. Grasshopper is a visual programming application where you are able to create programs, called definitions or documents, by dragging components onto the main editing window (called the canvas). The outputs to these components are connected to the inputs of subsequent components — creating a graph of information which can be read from left to right. Let's get started with the basics.

Assuming you've already installed the Grasshopper plugin (see F.0.0), type the word "Grasshopper" in the Rhino command prompt to display the Grasshopper Editor. The Grasshopper interface contains a number of elements, most of which will be very familiar to Rhino users. Let's look at a few features of the interface.



1. Windows title bar.
2. Main menu bar.
3. File browser control.
4. Component palettes.
5. Canvas toolbar.
6. Canvas.
7. This area, indicated by a grid of rectangular boxes, provides an interface with which to open recently accessed file. The 3x3 menu shows the files most recently accessed (chronologically) and will display a red rectangular box if the file cannot be found (which can occur if you move a file to a new folder or delete it).

8. The status bar tells you what version of Grasshopper is currently installed on your machine. If a newer version is available, a pop-up menu will appear in your tray providing instructions on how to download the latest version.

1.1.2.1. THE WINDOWS TITLE BAR

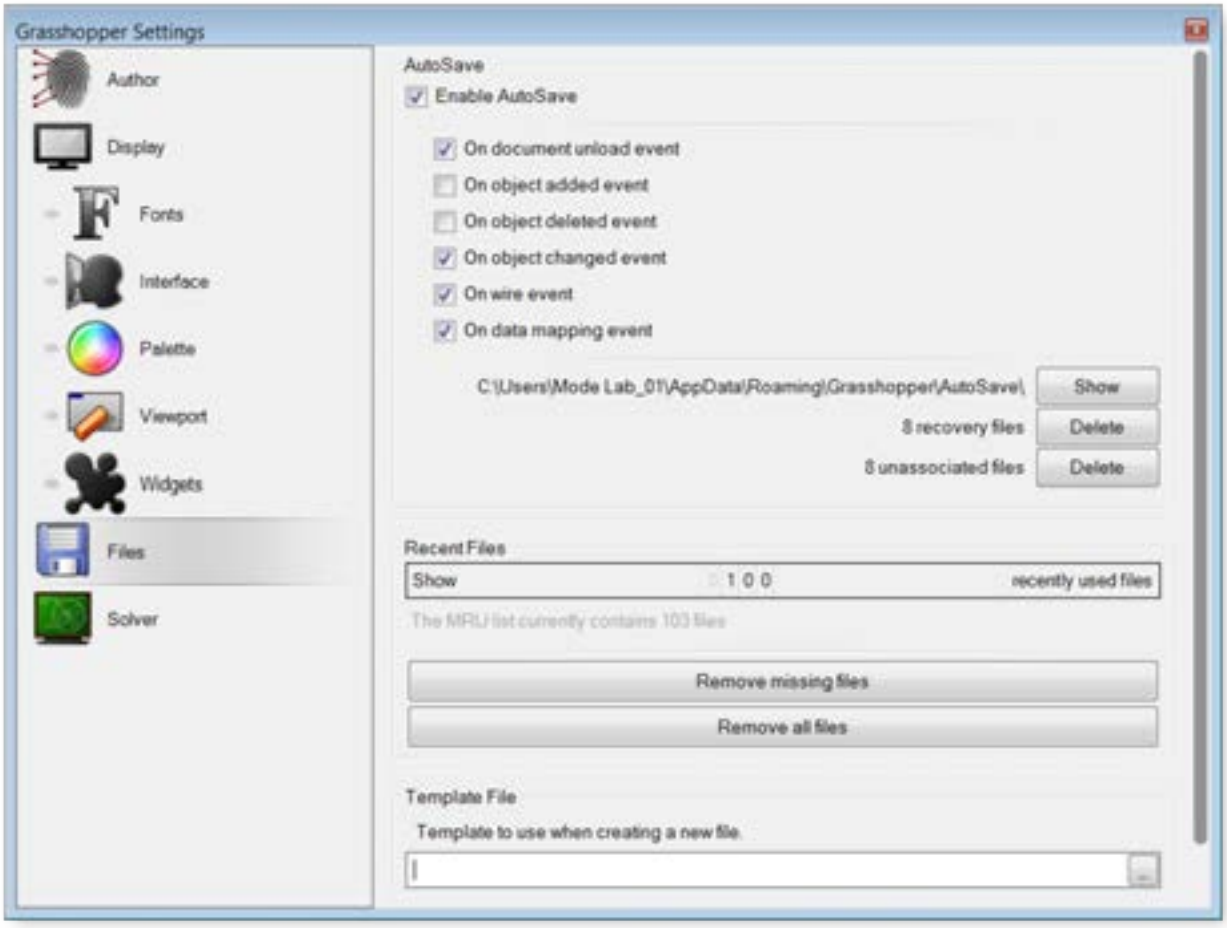
The Editor Window title bar behaves differently from most other dialogs in Microsoft Windows. If the window is not minimized or maximized, double clicking the title bar will collapse the dialog into a minimized bar on your screen. This is a great way to switch between the plug-in and Rhino because it minimizes the Editor without moving it to the bottom of the screen or behind other windows. Note that if you close the Editor, the Grasshopper geometry preview in the Rhino viewport will disappear, but the file won't actually be closed. The next time you run the "Grasshopper" command in the Rhino dialog box, the window will come back in the same state with the same files loaded. This is because once it is launched from the command prompt, your session of Grasshopper stays active until that instance of Rhino is closed.

1.1.2.2. MAIN MENU BAR

The title bar is similar to typical Windows menus, except for the file browser control on the right (see next section). The File menu provides typical functions (eg. New File, Open, Save, etc.) in addition to a few utility tools which let you export images of your current Grasshopper document (see Export Quick Image and Export Hi-Res Image). You can control different aspects of the user interface using the View and Display menus, while the Solution menu lets you manage different attributes about how the solver computes the graph solution.

It is worth noting that many application settings can be controlled through the Preferences dialog box found under the File menu. The Author section allows you to set personal metadata which will be stored with each Grasshopper document while the Display section gives you fine grain control over the look and feel of the interface. The Files section lets you specify things like how often and where to store automatically saved file (in case the application is inadvertently closed or crashes). Finally, the Solver section lets you manage core and third-party plugins which can extend functionality.

> Note: Be careful when using shortcuts since they are handled by the active window which could either be Rhino, the Grasshopper canvas or any other window inside Rhino. It is quite easy to use a shortcut command, only to realize that you had the wrong active window selected and accidentally invoked the wrong command.



> The Preferences dialog allows you to set many of Grasshopper's application settings.

1.1.2.3. FILE BROWSER CONTROL

The File Browser allows you to quickly switch between different loaded files by selecting them through this drop-down list. Accessing your open files through the File Browser drop-down list enables you to quickly copy and paste items from open definitions. Just click on the active file name in the browser control window and a cascading list of all open files will be displayed (along with a small thumbnail picture of each open definition) for easy access. You can also hit Alt+Tab to quickly switch between any open Grasshopper documents.

Of course, you can go through the standard Open File dialog to load any Grasshopper document, although you can also drag and drop any Grasshopper file onto the canvas to load a particular definition.

> Grasshopper is a plug-in that works "on-top" of Rhino and as such has its own file types. The default file type is a binary data file, saved with an extension of .gh. The other file type is known as a Grasshopper XML file, which uses the extension .ghx. The XML (Extensible Markup Language) file type uses tags to define objects and object attributes (much like an .HTML document) but uses custom tags to define objects and the data within each object. Because XML files are formatted as textdocuments, you could open up any Grasshopper XML file in a text editor like NotePad to see the coding that is going on behind the scenes.

Grasshopper has several different methods by which it can open a file, and you will need to specify which option you would like to use when using this method.

Open File: As the name suggests, this file option will simply open any definition that you drag and drop onto the canvas.

Insert File: You can use this option to insert an existing file into the current document as loose components.

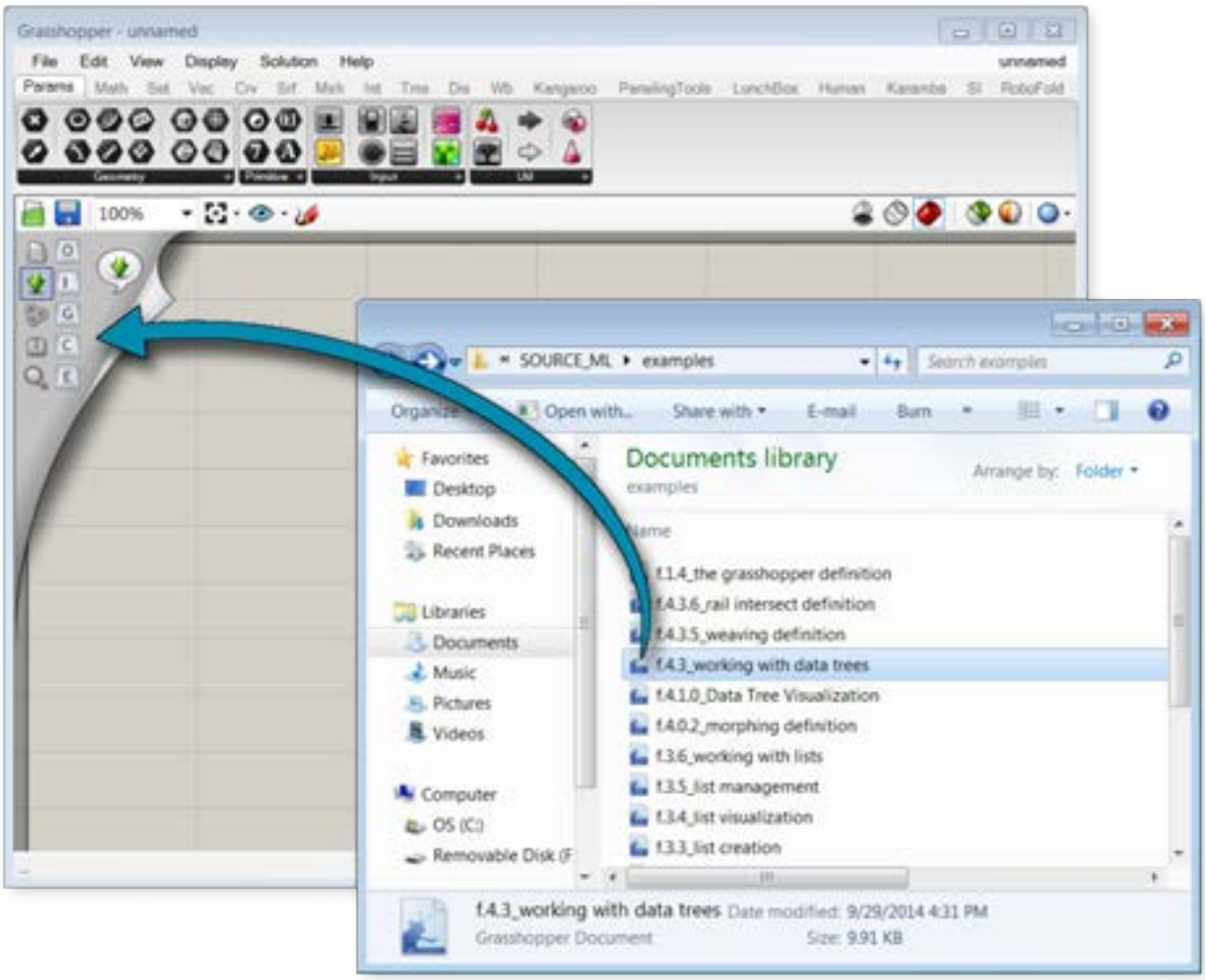
Group File: This method will insert a file into an existing document, but will group all of the objects together.

Cluster File: Similar to the group function, this method will insert a file into an existing document, but will create a cluster object for the entire group of objects.

Examine File: Allows you to open a file in a locked state, meaning you can look around a particular file but you can't make any changes to the definition.

Grasshopper also has an Autosave feature which will be triggered periodically based on specific user actions. Alist of Autosave preferences can be found under the File menu on the Main Menu Bar. When the active instance of Rhino is closed, a pop-up dialog box will appear asking whether or not you want to save any Grasshopper files that were open when Rhino was shut down.

> Autosave only works if the file has already been saved at least once.

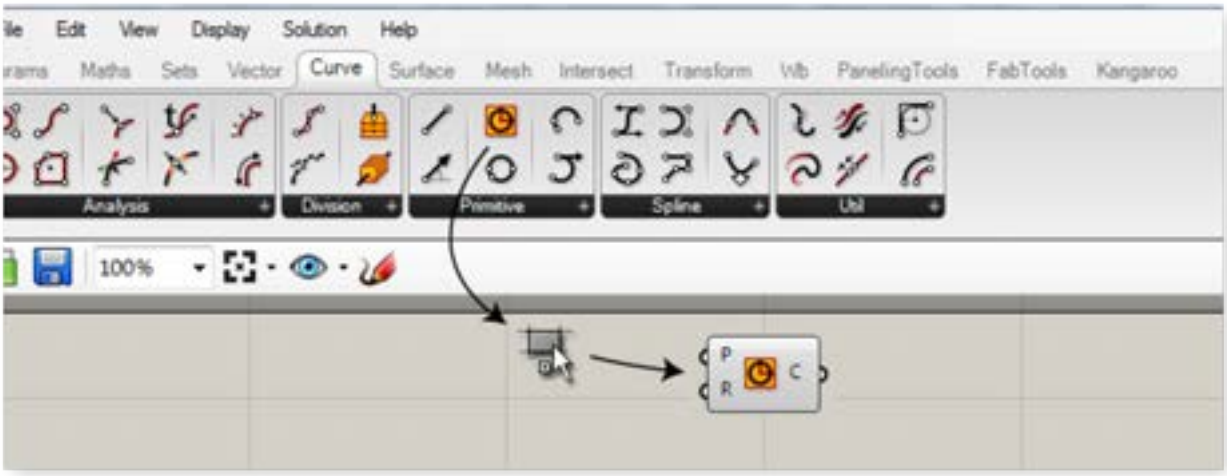


> Drag and Drop Files onto the Canvas.

1.1.2.4. COMPONENT PALETTES

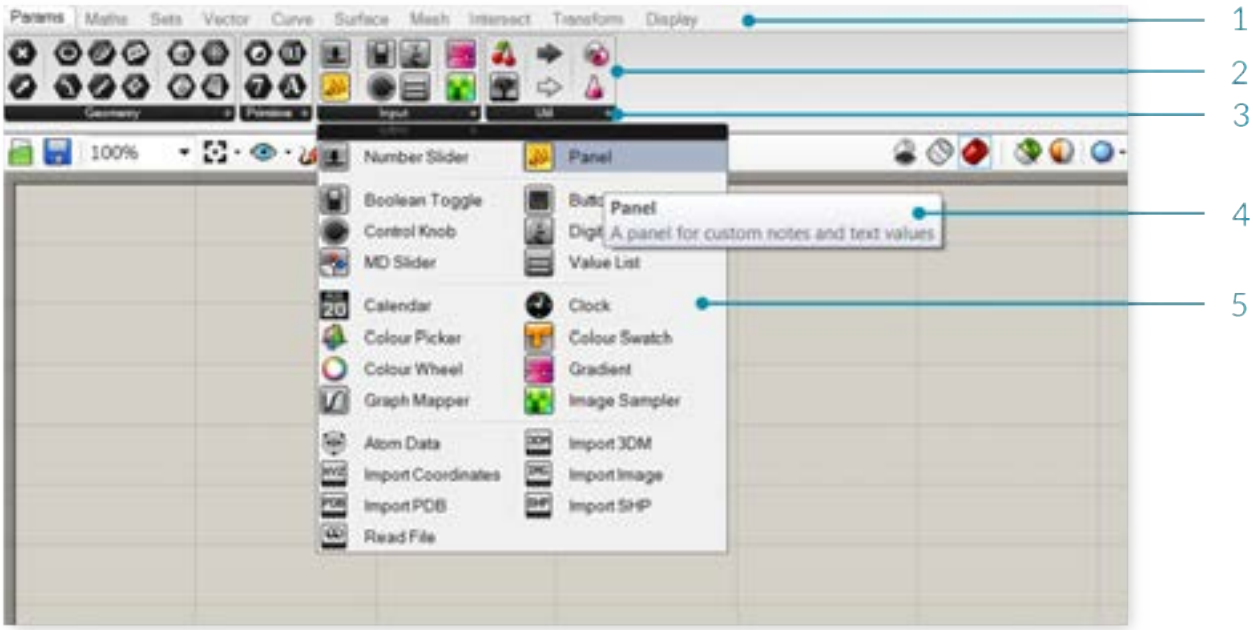
This area organizes components into categories and sub-categories. Categories are displayed as tabs, and subcategories are displayed as drop-down panels. All components belong to a certain category. These categories have been labeled to help you find the specific component that you are looking for (e.g. "Params" for all primitive

data types or “Curves” for all curve related tools). To add a component to the canvas, you can either click on the objects in the drop-down menu or you can drag the component directly from the menu onto the canvas.



> Drag + Drop a component from the palette to add a component to the canvas.

Since there can be many more components in each sub-category than will fit into the palette, a limited number of icons are displayed on each panel. The height of the component palette and the width of the Grasshopper window can be adjusted to display more or fewer components per sub-category. To see a menu of all of the components in a given sub-category, simply click on the black bar at the bottom of each sub-category panel. This will open a dropdown menu which provides access to all components in that sub-category.



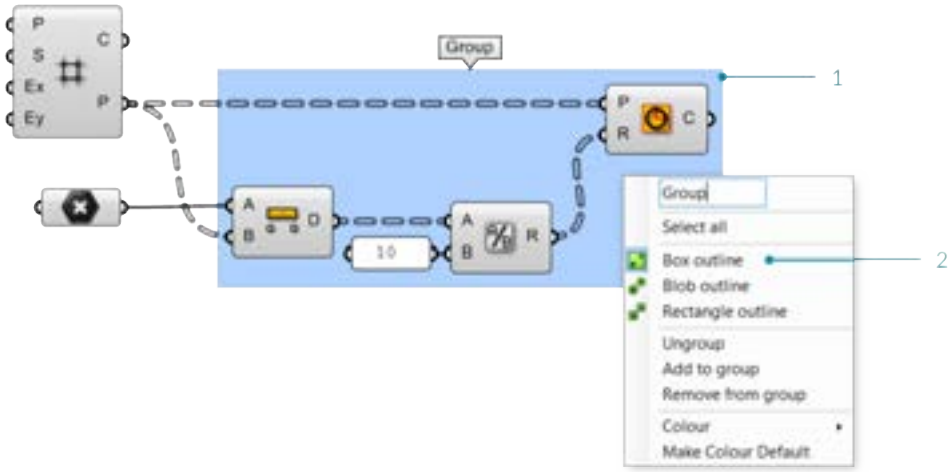
> 1. Catgory tab
> 2. Sub-category panel.
> 3. Click the black bar to open the sub-category panel menu.
> 4. Hover your mouse over a component for a short description.
> 5. Drop-down menu.

1.1.2.5. THE CANVAS

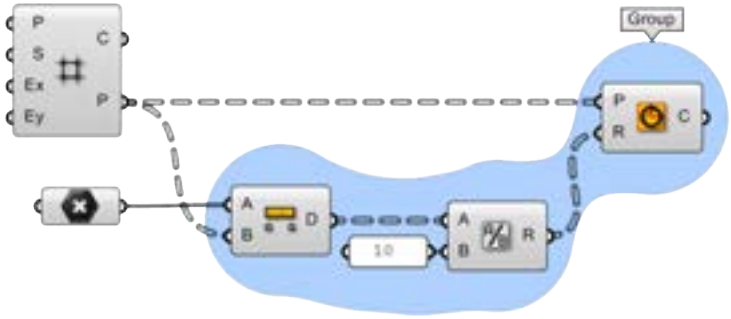
The canvas is the primary workspace for creating Grasshopper definitions. It is here where you interact with the elements of your visual program. You can start working in the canvas by placing components and connecting wires.

1.1.2.6. GROUPING

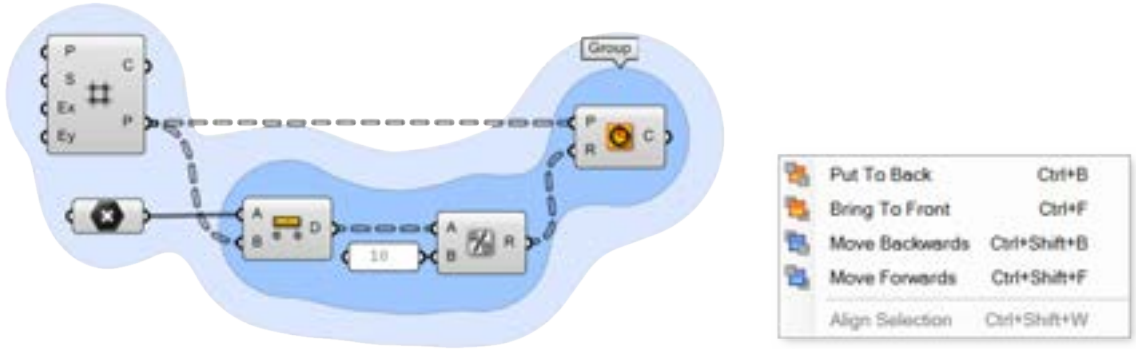
Grouping components together on the canvas can be especially useful for readability and comprehensibility. Grouping allows you the ability to quickly select and move multiple components around the canvas. You can create a group by typing Ctrl+G with the desired components selected. An alternate method can be found by using the “Group Selection” button under the Edit Menu on the Main Menu Bar. Custom parameters for group color, transparency, name, and outline type can be defined by right-clicking on any group object.



1. Agroup of components delineated by the Box Outline profile.
2. Right-click anywhere on the group to edit the name and appearance of the group.



You can also define a group using a meta-ball algorithm by using the Blob Outline profile.



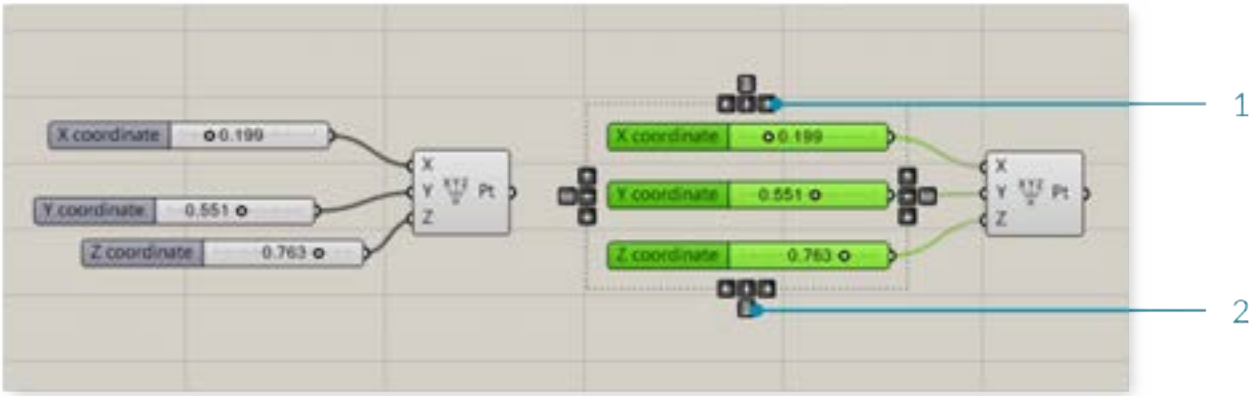
Two groups are nested inside one another. The color (light blue) has been changed on the outer group to help visually identify one group from the other. Groups are drawn “behind” the components within them and, in cases such as this, there is a depth order to the two groups. To change this, go to Edit > Arrange in the

> main menu bar.

1.1.1.7. WIDGETS

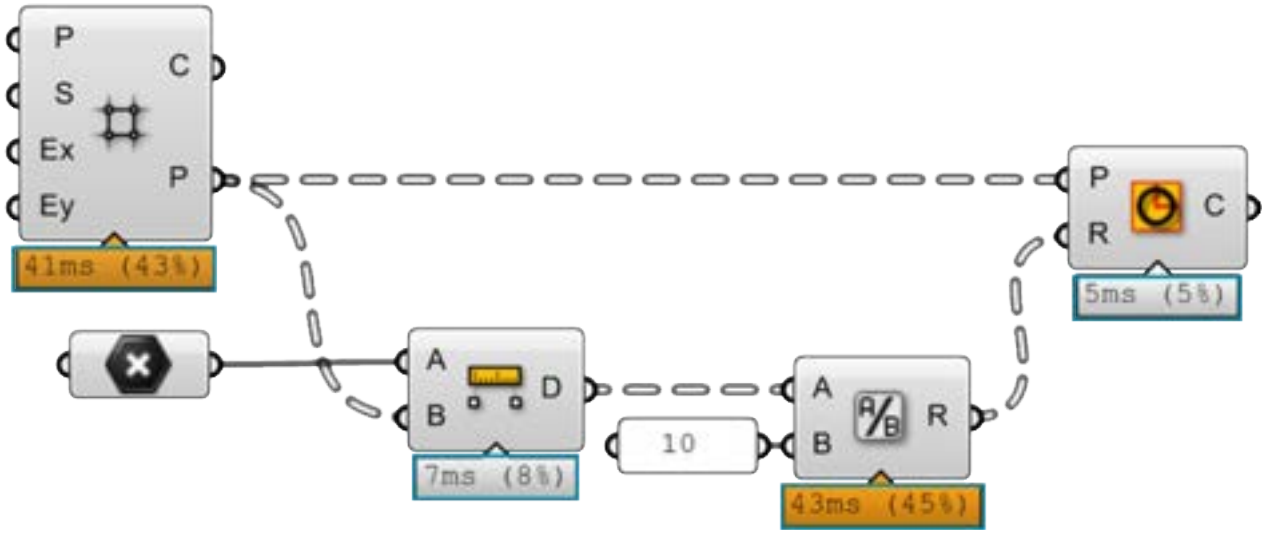
There are a few widgets that are available in Grasshopper that can help you perform useful actions. You can toggle any of these widgets on/off under the Display menu of the Main Menu bar. Below we'll look at a few of the most frequently used widgets.

The Align Widget One useful UI widget which can help you keep your canvas clean is the Align widget. You can access the Align widget by selecting multiple components at the same time and clicking on one of the options found in the dashed outline that surrounds your selected components. You can align left, vertical center, right, or top, horizontal center, bottom, or distribute components equally through this interface. When first starting out, you may find that these tools sometimes get in the way (it is possible to make the mistake of collapsing several components on top of each other). However, with a little practice these tools can be invaluable as you begin to structure graphs which are readable and comprehensible.



> 1. Align right.
> 2. Distribute vertically.

The Profiler Widget The profiler lists worst-case runtimes for parameters and components, allowing you to track down bottlenecks in networks and to compare different components in terms of performance. Note that this widget is turned off by default.



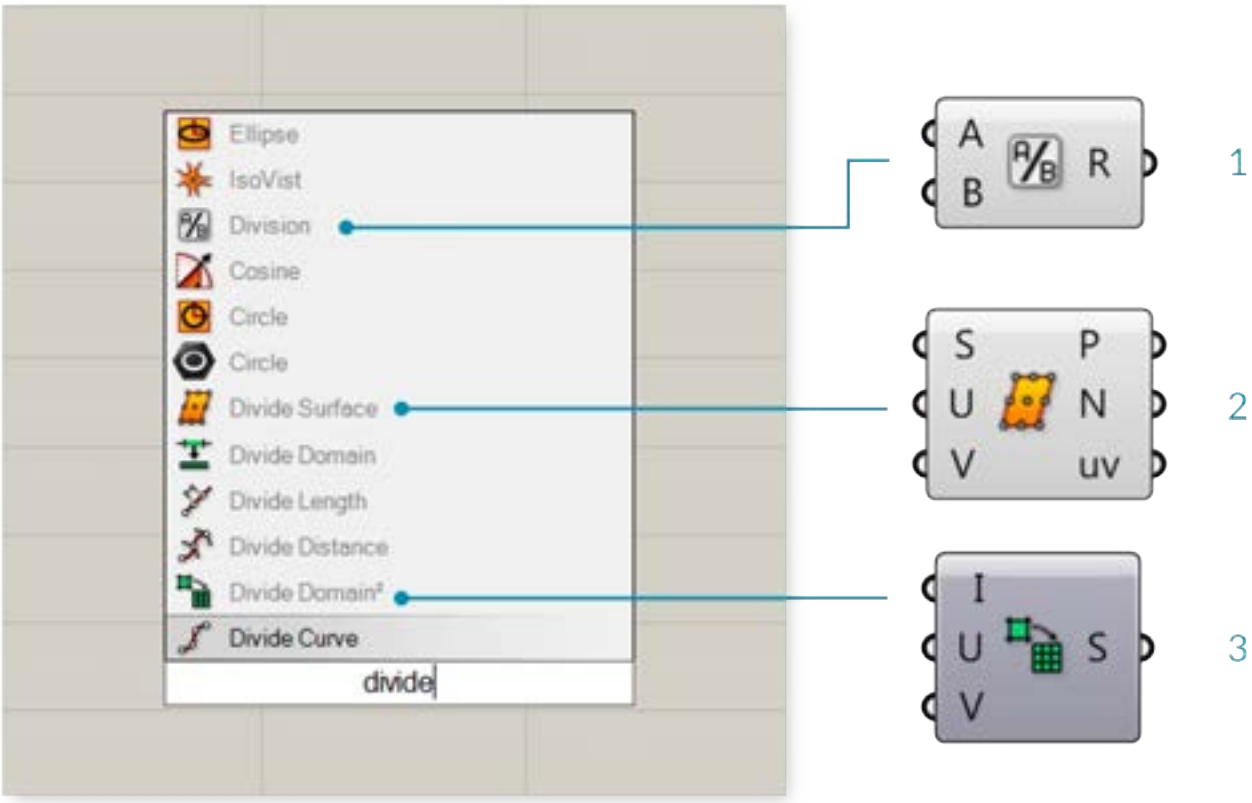
> The Profiler widget gives you visual feedback as to which components in your definition could be causing longer computational times.

The Markov Widget This widget uses Markov chains to 'predict' which component you may want to use next based on your behavior in the past. A Markov chain is a process that consists of a finite number of states (or levels) and some known probabilities. It can take some time for this widget to become accustomed to a particular user, but over time you should begin to notice that this widget will begin to suggest components that you may want to use next. The Markov Widget can suggest up to five possible components depending on your recent activity. You can right-click on the Markov widget (the default location is the bottom left-hand corner of the canvas) to dock it into one of the other corners of the canvas or to hide it completely.

1.1.2.8. USING THE SEARCH FEATURE

Although a lot of thought has gone into the placement of each component on the component panel to make it intuitive for new users, people sometimes find it difficult to locate a specific component that might be buried deep inside one of the category panels. Fortunately, you can also find components by name, by double-clicking on any empty space on the canvas. This will invoke a pop-up search box. Simply type in the name of the component you are looking for and you will see a list of parameters or components that match your request.

> Double-click anywhere on the canvas to invoke a key word search for a particular component found in the Component Panels.



> Asearch for "divide" lists a variety of components.
>
> 1. Division operator component.
> 2. Divide Surface component.

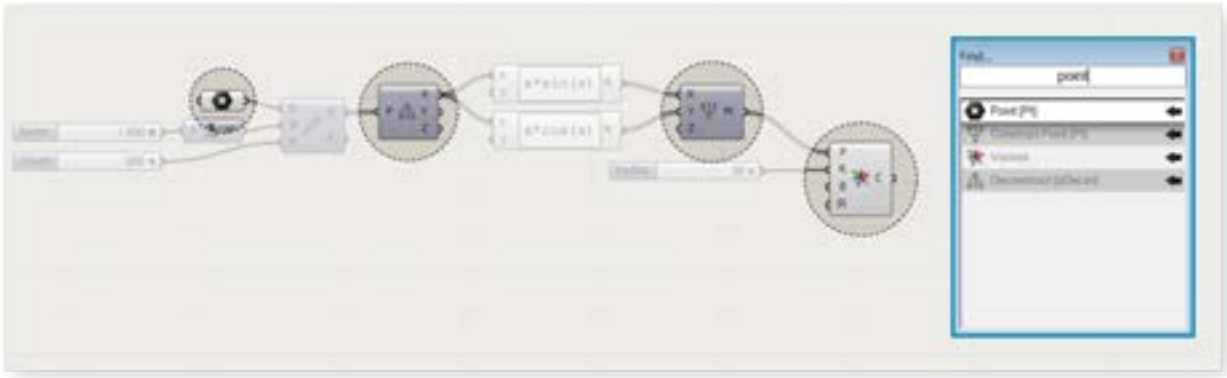
> 3. Divide Domain2 component.

1.1.2.9. THE FIND FEATURE

There are literally hundreds (if not thousands) of Grasshopper components which are available to you and it can be daunting as a beginner to know where to look to find a specific component within the Component Palettes. The quick solution is to double-click anywhere on the canvas to launch a search query for the component you are looking for. However, what if we need to find a particular component already placed on our canvas? No need to worry. By right-clicking anywhere on the canvas or pressing the F3 key, you can invoke the Find feature. Start by typing in the name of the component that you are looking for.

The Find feature employs the use of some very sophisticated algorithms which search not only for any instances of a component's name within a definition (a component's name is the title of the component found under the Component Panel which we as users cannot change), but also any unique signatures which we may have designated for a particular component (known as nicknames). The Find feature can also search for any component type on the canvas or search through text panel, scribble, and group content. Once the Find feature has found a match, it will automatically grey out the rest of the definition and draw a dashed line around the highlighted component. If multiple matches are found, a list of components matching your search query will be displayed in the Find dialog box and hovering over an item in the list will turn that particular component on the canvas green.

> By right-clicking anywhere on the canvas or pressing the F3 key, you can invoke the Find feature. Start by typing in the name of the component that you are looking for.



> The Find feature can be quite helpful to locate a particular component on the canvas. Right-click anywhere on the canvas to launch the Find dialog box.

> A small arrow will also be displayed next to each item in the list which points to its corresponding component on the canvas. Try moving the Find dialog box around on the canvas and watch the arrows rotate in to keep track of their components. Clicking on the Find result will try to place the component (on the canvas) next to the Find dialog box.

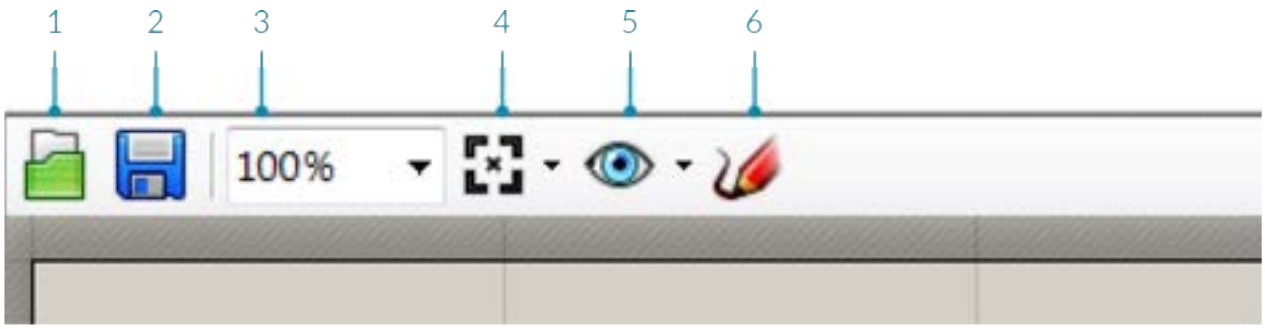
1.1.2.10. USING THE RADIAL MENU

As you become more proficient in using the Grasshopper interface, you'll begin to find ways to expedite your workflow. Using shortcuts is one way to do this, however there is another feature which can allow you to quickly access a number of useful tools – the radial UI menu. You can invoke the radial menu by hitting the space bar (while your mouse is over the canvas or a component) or by clicking your middle mouse button. The radial menu will enable different tools depending on whether you invoke the menu by clicking directly on top of a component, or just anywhere on the canvas. In the image below, you see the radial menu has more features available when clicking on top of a selected component versus just clicking anywhere else on the canvas. This menu can dramatically increase the speed at which you create Grasshopper documents.

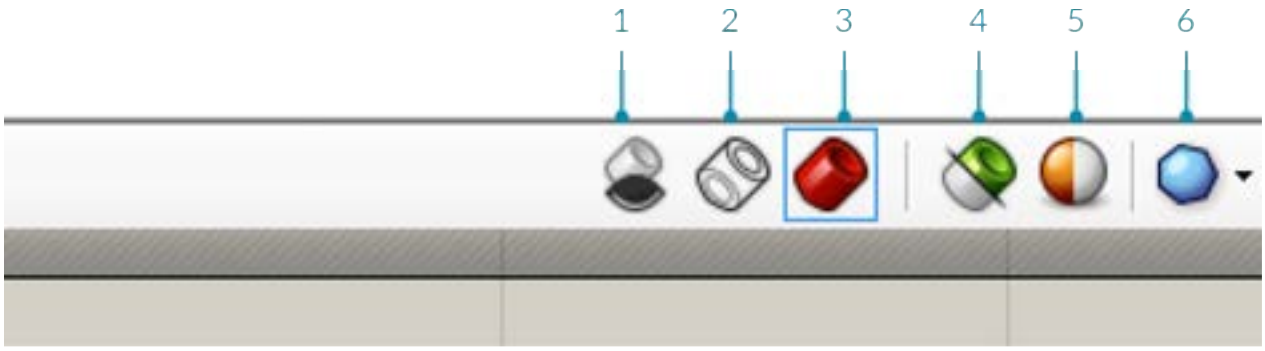
> The Radial UI menu allows you to quickly access frequently used menu items.

1.1.2.11. THE CANVAS TOOLBAR

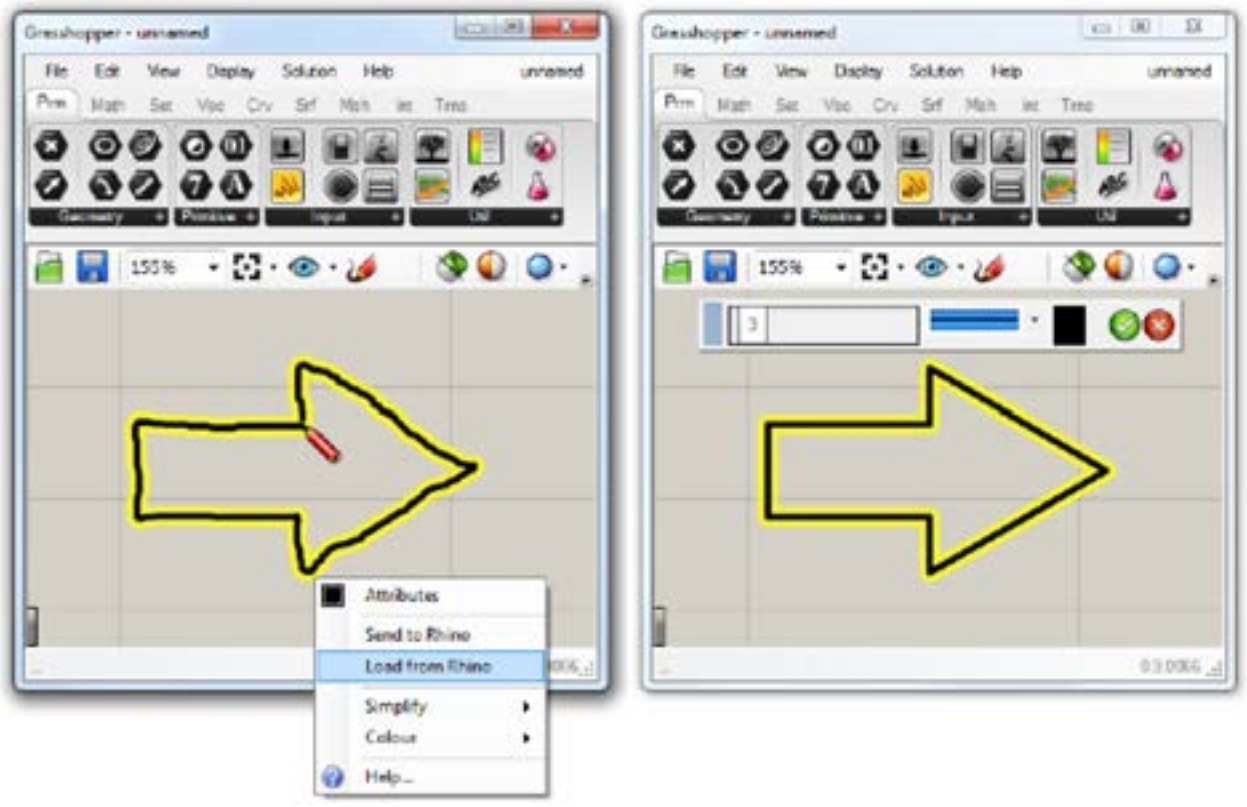
The canvas toolbar provides quick access to a number of frequently used Grasshopper features. All of the tools are available through the menu as well, and you can hide the toolbar if you like. The toolbar can be re-enabled from the View tab on the Main Menu Bar.



1. **Open File:** Ashortcut to open a Grasshopper File.
2. **Save File:** Ashortcut to save the currentGrasshopper File.
3. **Zoom Defaults**: Default zoom settings that allow you to zoom in or out of your canvas at predefined intervals.
4. **Zoom Extents:** Zoom to the extents of your definition. Click on the arrow next to the Zoom Extents icon to select one of the sub-menu items to zoom to a particular region within your definition.
5. **Named Views:** This feature exposes a menu allowing you to store or recall any view area in your definition.
6. **The Sketch Tool:** The sketch tool works similarly to the pencil tool set found in Adobe Photoshop with a few added features.

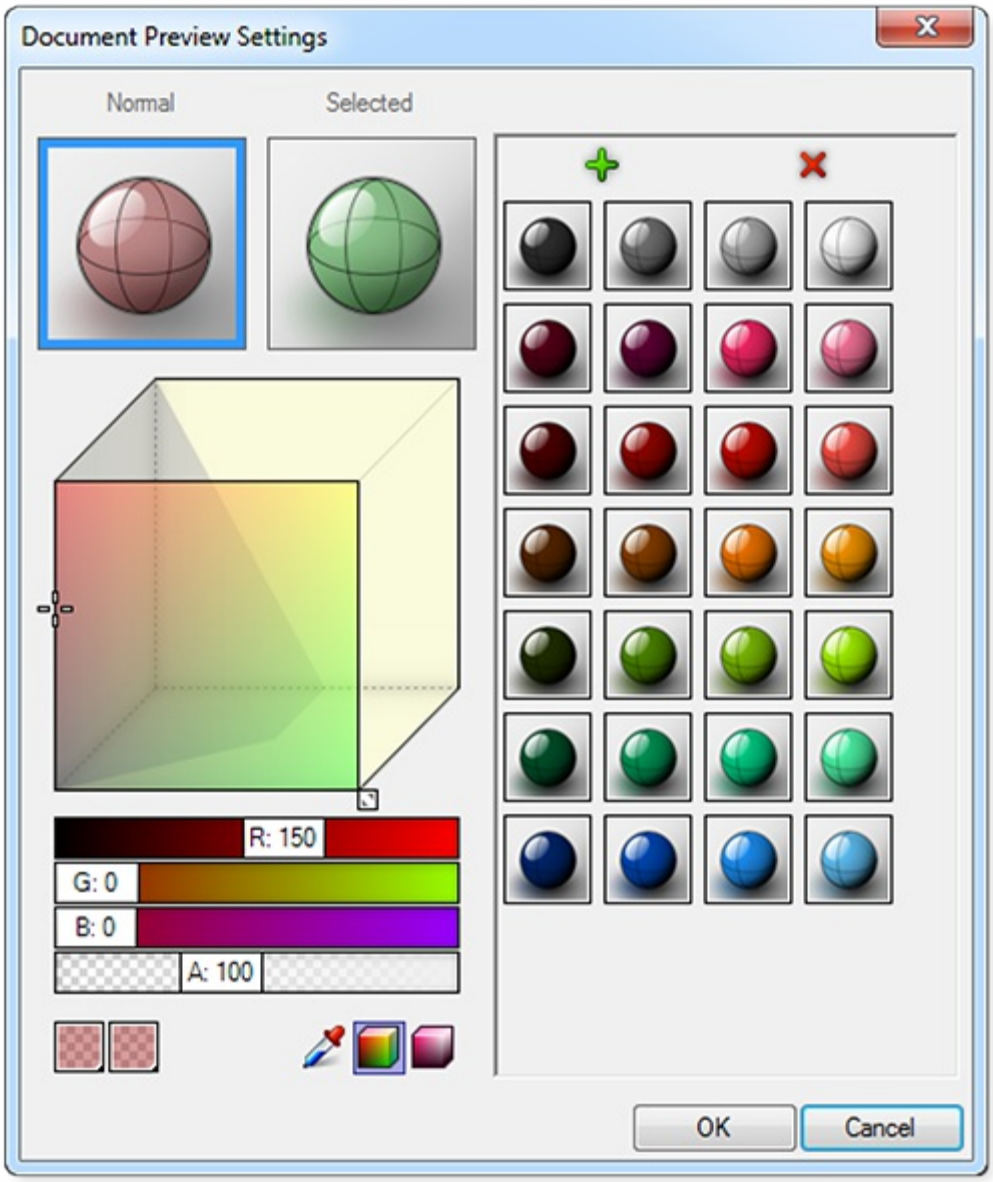


1. **Preview Settings:** If a Grasshopper component generates some form of geometry, then a preview of this geometry will be visible in the viewport by default. You can disable the preview on a perobject basis by right-clicking each component and de-activating the preview feature, or globally change the preview state by using one of these three buttons.
2. Wire-frame preview.
3. Turn off preview.
4. Shaded preview (default).
5. **Preview Selected Objects:** With this button toggled, Grasshopper will only display geometry that is part of selected components, even if those components have a preview=off state.
6. **Document Preview Settings:** Grasshopper has a default color scheme for selected (semi-transparent green) and unselected (semi-transparent red) geometry. It is possible to override this color scheme with the Document Preview Settings dialog.
7. **Preview Mesh Quality:** For optimization purposes, these settings allow you to control the quality of the mesh/surface display of the geometry rendered in Rhino. Higher quality settings will cause longer calculation times, whereas lower settings will display less accurate preview geometry. It should be noted that the geometry still maintains a high-degree of resolution when baked into the Rhino document – these settings merely effect the display performance and quality.



The sketch tool allows changes to the line weight, line type, and color. By right-clicking on the selected sketch

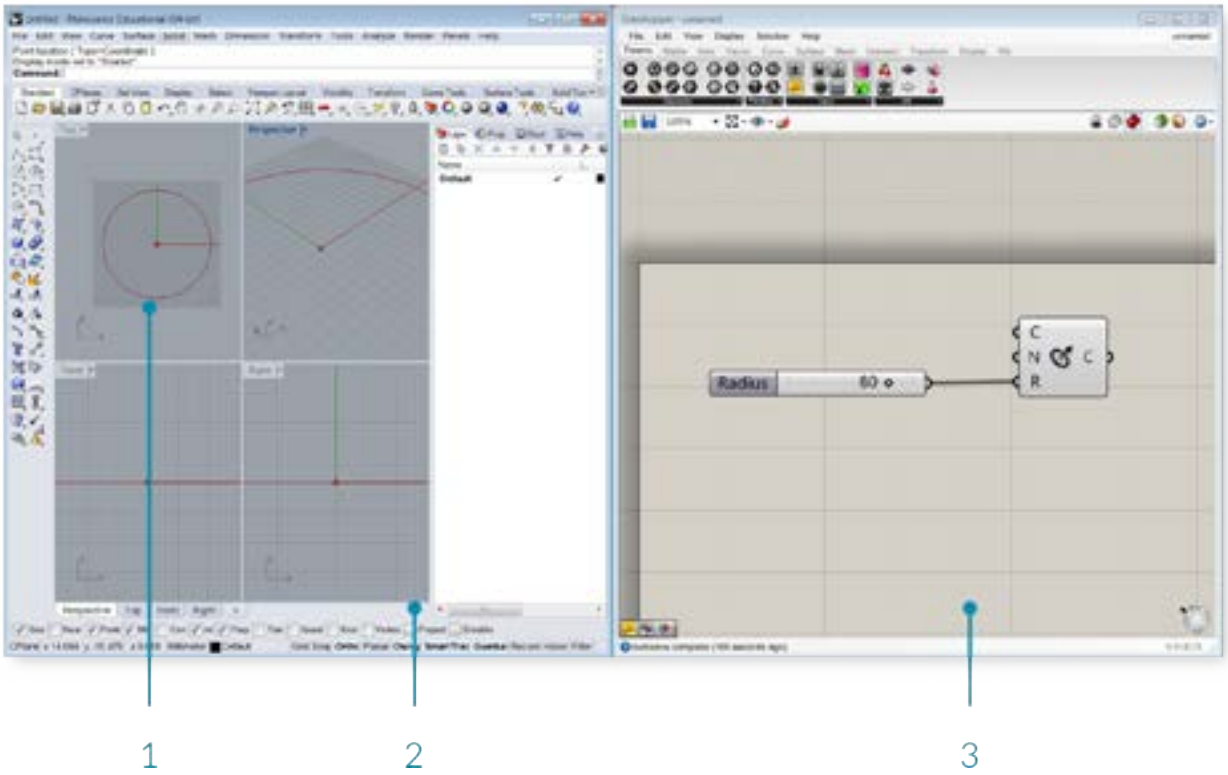
> object you can choose to simplify your line to create a smoother effect. Right-click on your sketch object and select “Load from Rhino”. When prompted, select any 2D shape in your Rhino scene. Once you have selected your referenced shape, hit Enter, and your previous sketch line will be reconfigured to your Rhino reference shape.
>
> Note: Your sketch object may have moved from its original location once you have loaded a shape from Rhino. Grasshopper places your sketch object relative to the origin of the canvas (upper left hand corner) and the world xy plane origin in Rhino.



> Grasshopper has a default color scheme for selected (semi-transparent green) and unselected (semi-transparent red) geometry. It is possible to override this color scheme with the Document Preview Settings dialog.

1.1.3. TALKING TO RHINO

Unlike a Rhino document, a Grasshopper definition does not contain any actual objectsor geometry. Instead, a Grasshopper definition represents a set of rules & instructions for how Rhino can automate tasks.

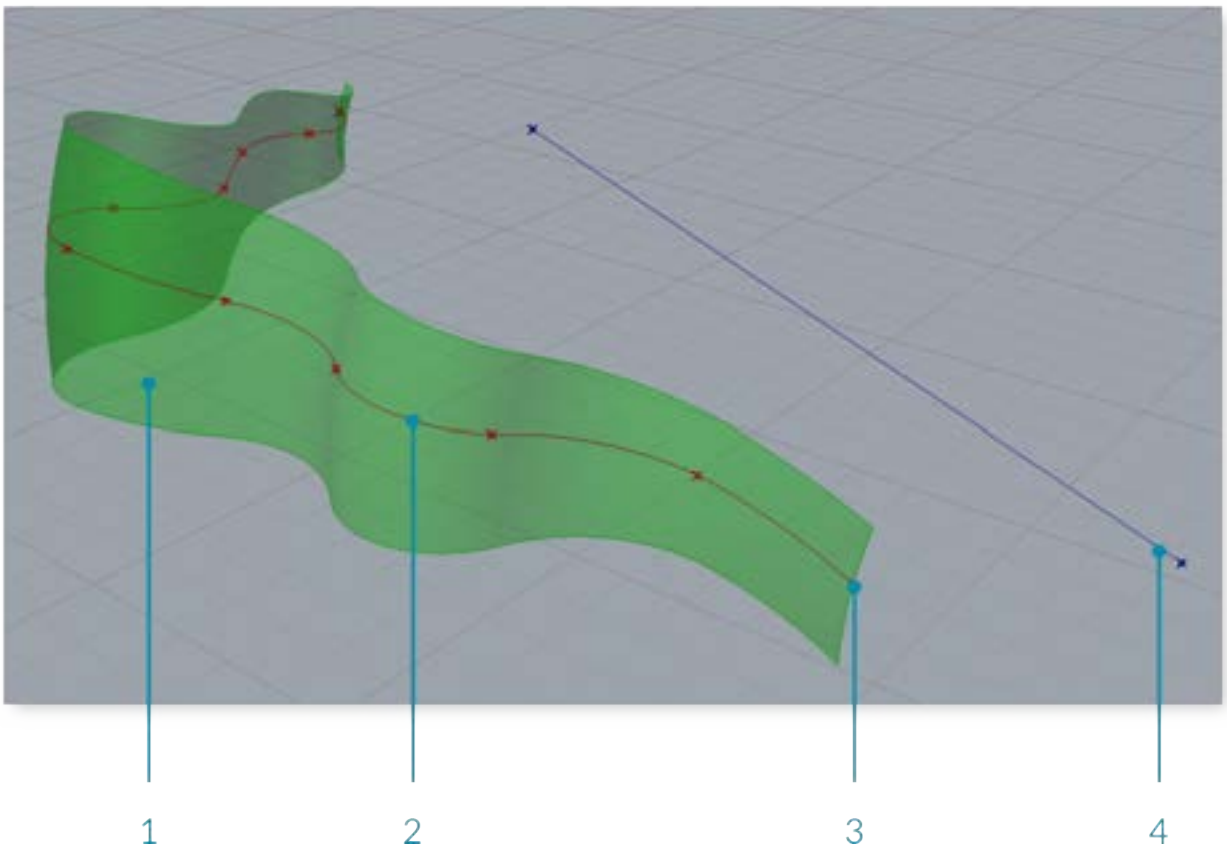


1. Grasshopper preview geometry.
2. Rhino viewports.
3. Grasshopper Application window.

1.1.3.1. VIEWPORT FEEDBACK

All geometry that is generated using the various Grasshopper components will show up (by default) in the Rhino viewport. This preview is just an Open GL approximation of the actual geometry, and as such you will not be able to select the geometry in the Rhino viewport (you must first bake it into the scene). You can turn the geometry preview on/off by right-clicking on a component and selecting the Preview toggle. The geometry in the viewport is color coded to provide visual feedback. The image below outlines the default color scheme.

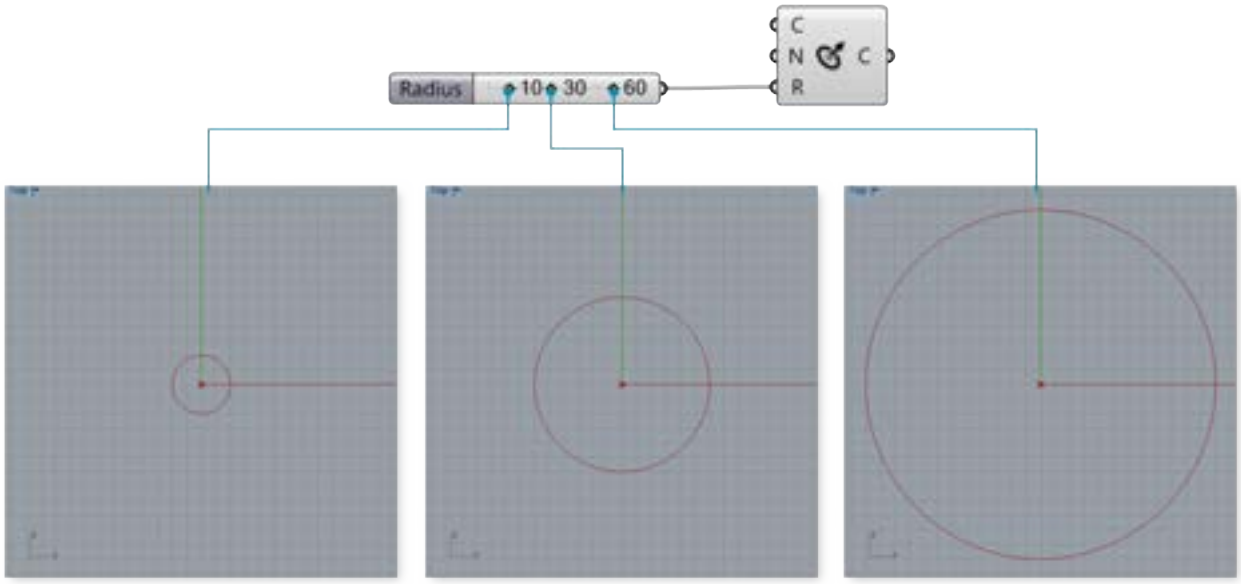
> Note: This is the default color scheme, which can be modified using the Document Preview Settings tool on the canvas toolbar.



1. Green geometry in the viewport belongs to a component which is currently selected.
2. Red geometry in the viewport belongs to a component which is currently unselected.
3. Point geometry is drawn as a cross rather than a rectangle to distinguish it from other Rhino point objects.
4. Blue feedback means you are currently making a selection in the Rhino Viewport.

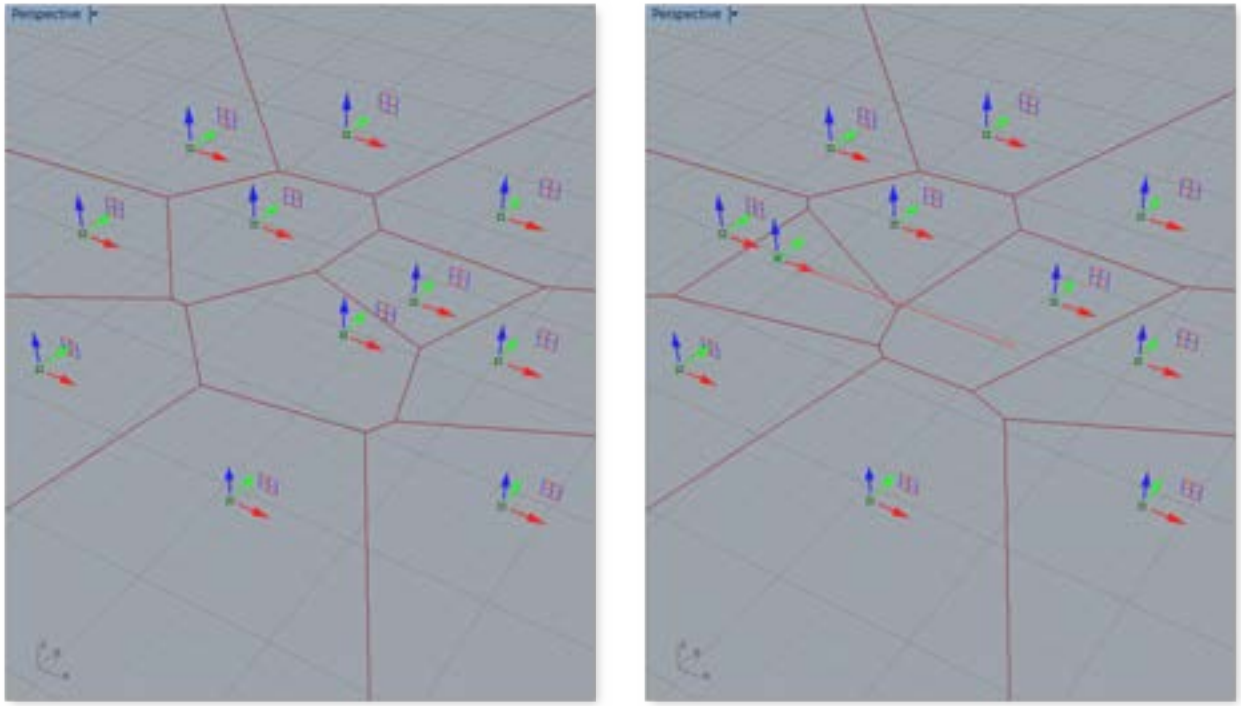
1.1.3.2. LIVE WIRES

Grasshopper is a dynamic environment. Changes that are made are live and their preview display is updated in the Rhino viewport.



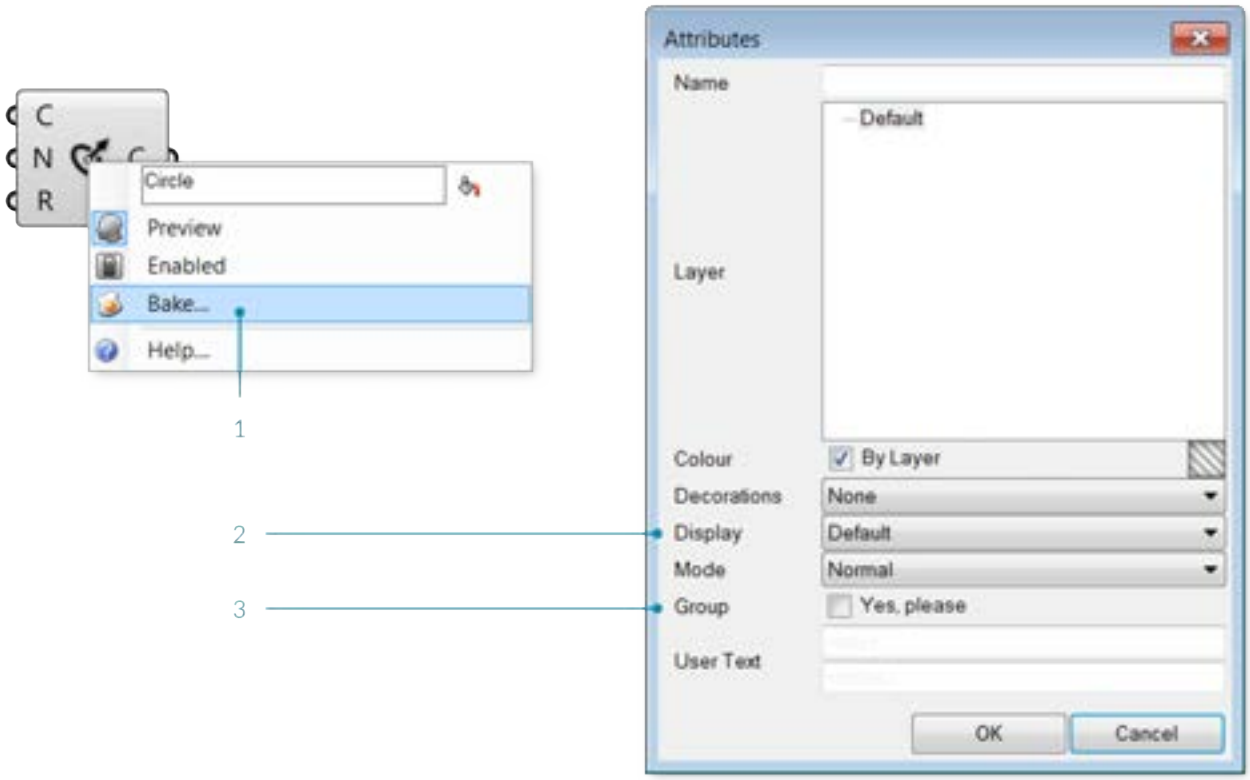
1.1.3.3. GUMBALL WIDGET

When storing geometry as internalized in a Grasshopper parameter, the gumball allows you to interface with that geometry in the Rhino viewport. This update is live and updates will occur as you manipulate the gumball. In contrast, geometry referenced from Rhino directly will continue to exist in the Rhino document and updates from Grasshopper will happen only after any changes occur (as opposed to during).



1.1.3.4. BAKING GEOMETRY

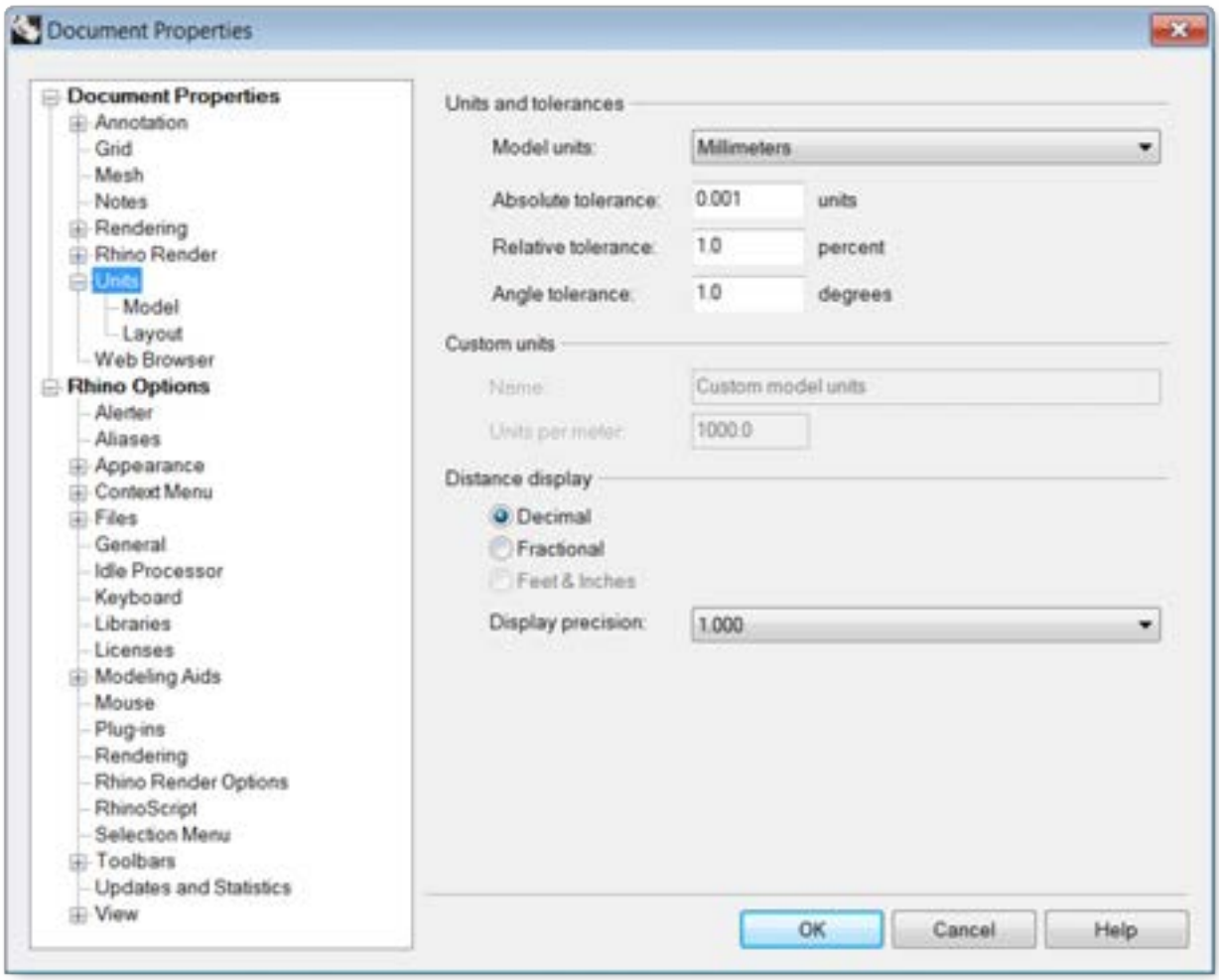
In order to work with (select, edit, transform, etc.) geometry in Rhino that was created in Grasshopper, you must “bake” it. Baking instantiates new geometry into the Rhino document based on the current state of the Grasshopper graph. It will no longer be responsive to the changes in your definition.



1. Bake by right-clicking a component and selecting Bake.
2. Adialog will appear that allows you to select onto which Rhino layer the geometry will bake.
3. Grouping your baked geometry is a convenient way to manage the instantiated Rhino geometry, particularly if you are creating many objects with Grasshopper.

1.1.3.5. UNITS & TOLERANCES

Grasshopper inherits units and tolerances from Rhino. To change the units, type Document Properties in the Rhino command line to access the Document Properties menu. Select Units to change the units and tolerances.



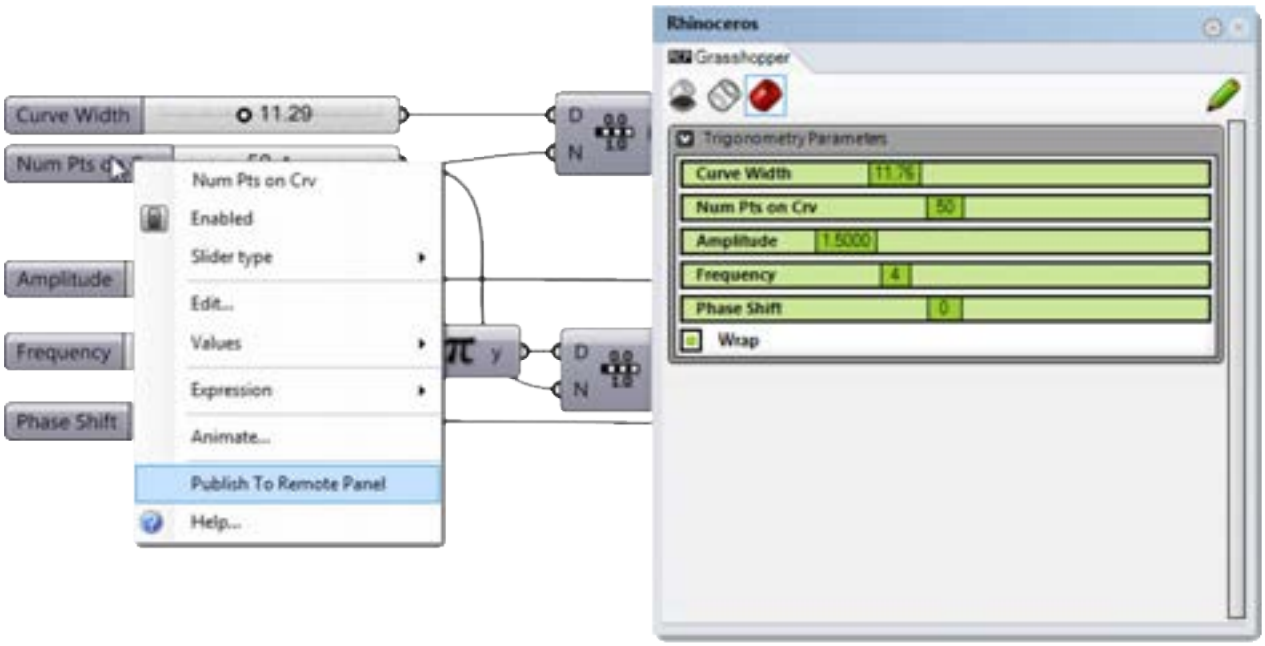
> Change the units and tolerances in the Rhino Document Properties menu.

1.1.3.6. REMOTE CONTROL PANEL

Once you get the hang of it, Grasshopper is an incredibly powerful and flexible tool which allows you to explore design iterations using a graphic interface. However, if you're working with a single screen then you may have already noticed that the Grasshopper editor takes up a lot of screen real-estate. Other than constantly zooming in and out and moving windows around your screen, there really isn't an elegant solution to this problem. That is… until the release of the Remote Control Panel!

The Remote Control Panel (RCP) provides a minimal interface to control your definition without taking up a substantial portion of your screen. The RCP can be instantiated by clicking on the toggle under the View menu of the Main Menu bar. By default, the RCP is blank — meaning it doesn't contain any information about your current Grasshopper document. To populate the RCP with UI elements like sliders, toggles, and buttons, simply right click on the element and click Publish To Remote Panel. This will create a new group and create a synchronized UI element in the RCP. Changing the value of the element in the RCP will also update the value in the graph, as well as modify any geometry in the viewport which might be dependant on this parameter. You can publish multiple elements and populate a complete interface which can be used to control your file without having the clutter of the visual graph showing up on top of the Rhino viewport.

> Note: The RCP will inherit the UI elements name and use it as the label. It is good practice to update your sliders and toggles with comprehensible and meaningful names. This will translate directly to your RCP making it easier to use.



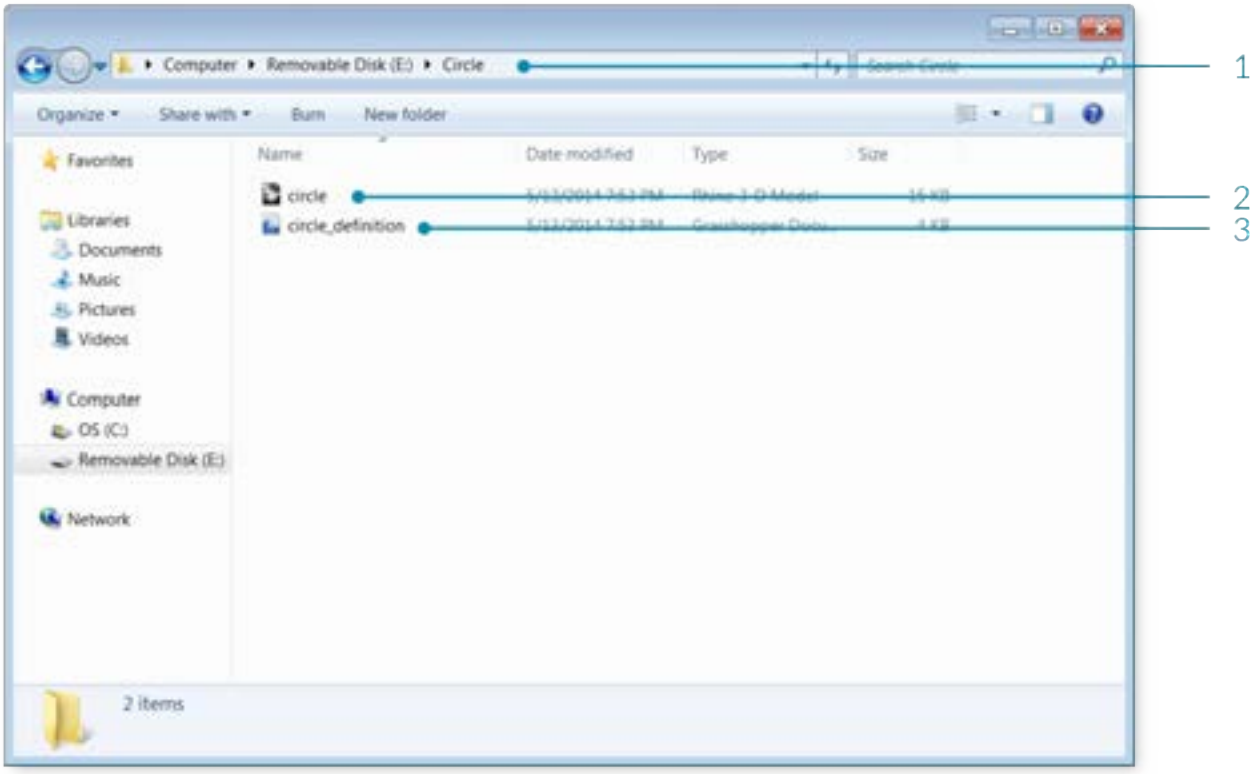
> In order to get a UI element (eg. slider, toggle, button, etc.) to show up in the Remote Control Panel, we have to first publish it.

The RCP UI can also be customized – allowing you to control where objects appear in the interface, the names and colors of different groups. To modify the layout of the RCP you first have to switch from Working Mode (the default RCP view) to Edit Mode. You can enter the Editing Mode by clicking on the green pencil in the upper right hand corner of the RCP. Once in Editing Mode, you can create new UI groups, rearrange elements within groups, add labels, change colors and more. To delete a UI element, simply drag the element outside the border of the RCP. You cannot change the individual values of the parameters if you are in Editing Mode. Instead, you will have to click on the green pencil icon to switch back to the standard Working Mode.

> *The Remote Control Panel has two modes: Edit Mode (left) which allows you to reorganize the look and feel of the RCP, and Working Mode where you can modify the actual values of the UI elements.*
>
> The Remote Control Panel in Edit Mode has an orange background.

1.1.3.7. FILE MANAGEMENT

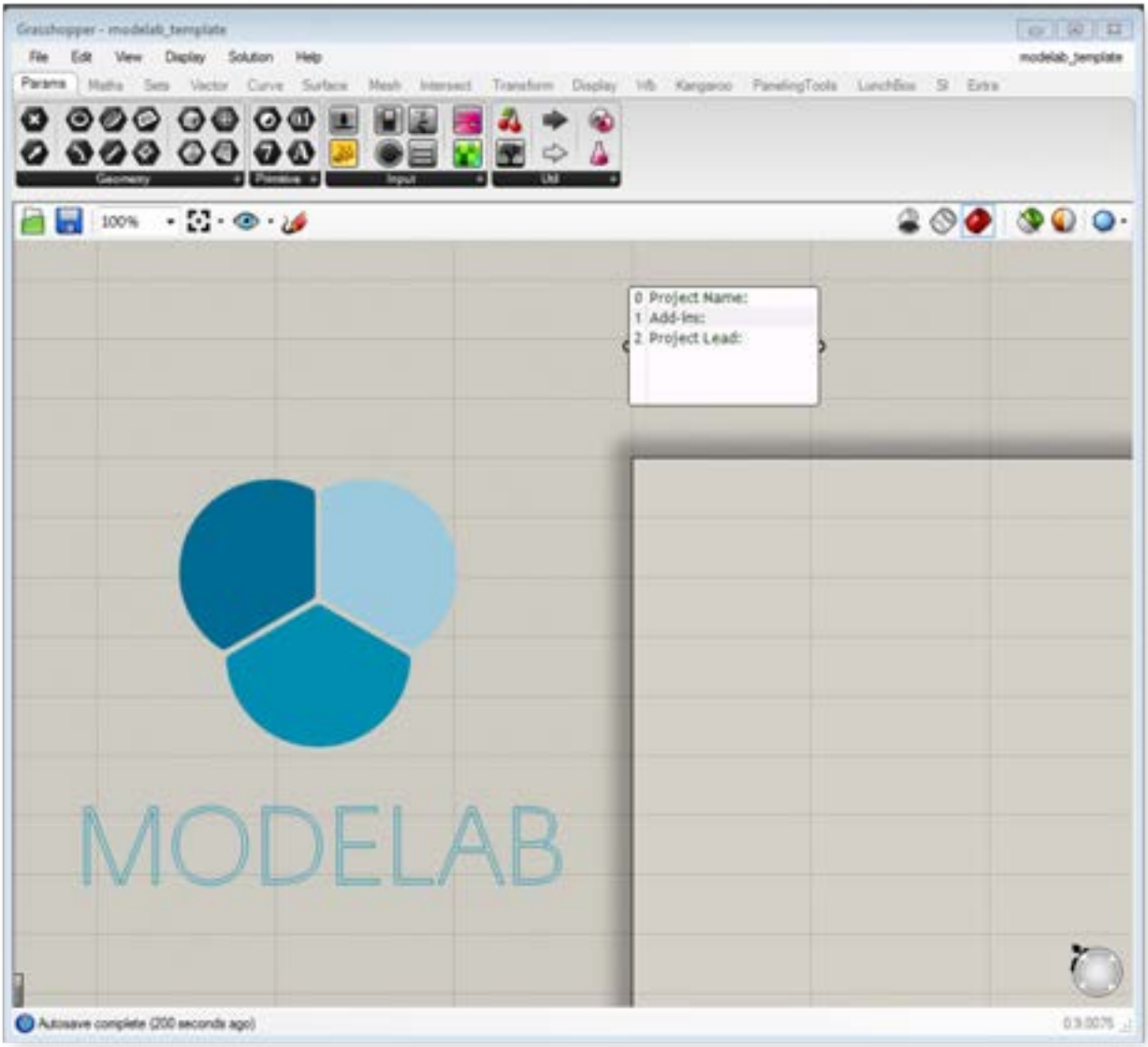
If your Grasshopper file references geometry from Rhino, you must open that same file for the definition to work. Keep your files organized by storing the Grasshopper and Rhino files in the same folder, and giving them related names.



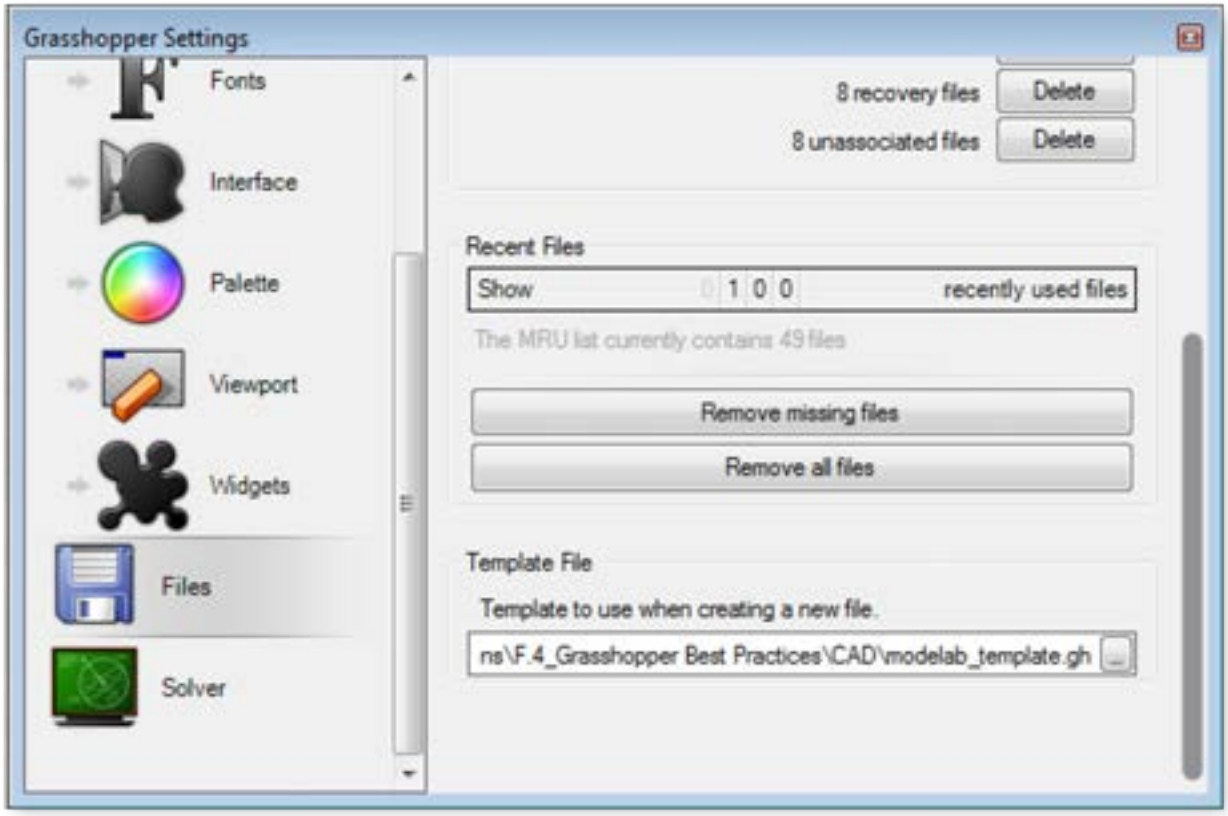
1. Project Folder.
2. Rhino file.
3. Grasshopper file.

1.1.3.8. TEMPLATES

Creating and specifiying a template file in your Grasshopper preferences is convenient way to set up every new Grasshopper definition you create. The template can include Grasshopper components as well as panels and sketch objects for labeling.



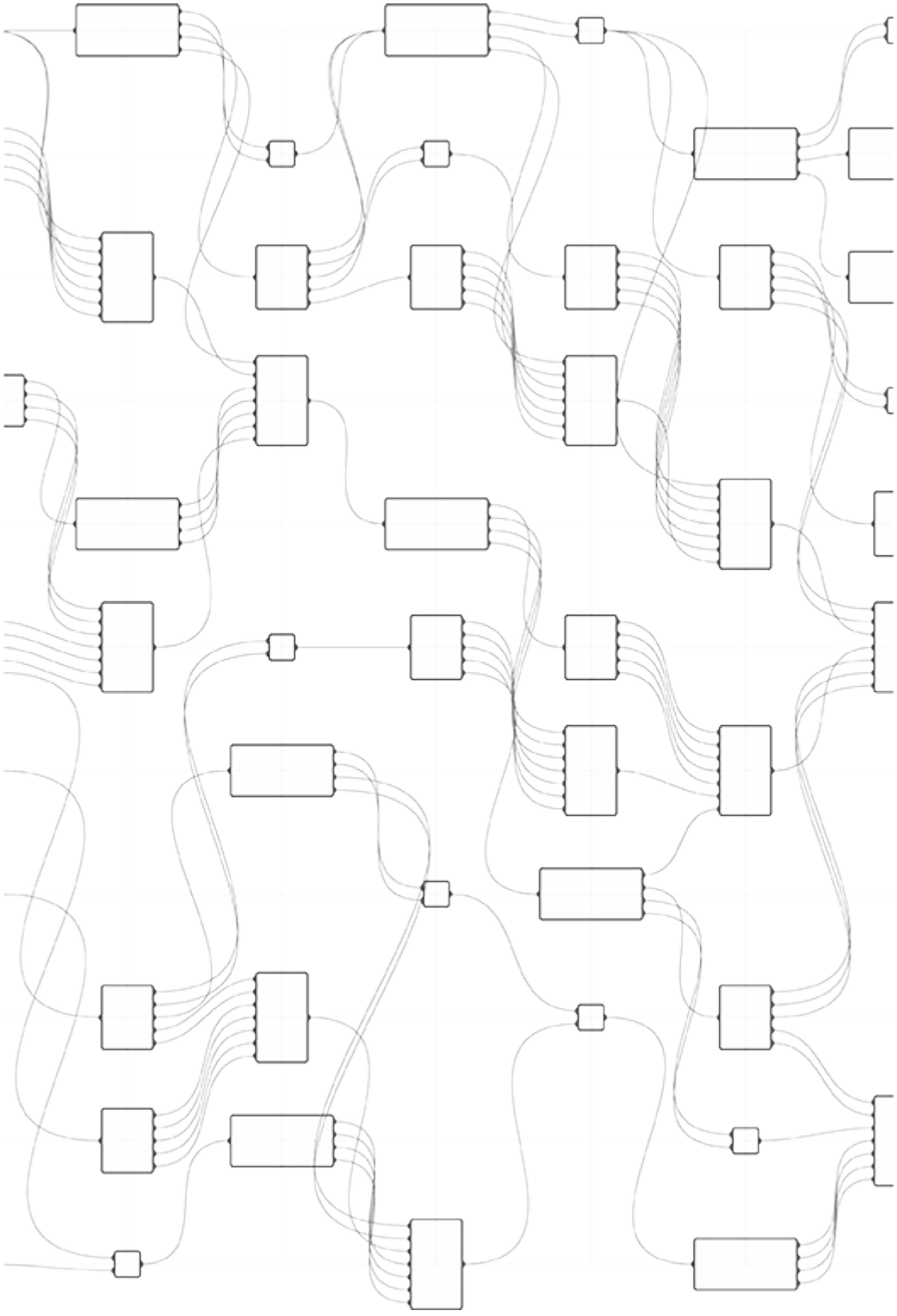
> Create a template file and save it



1. In File/Preferences, load the file you just created under Template File. Your template will now be used each time you create a new file.

1.2. ANATOMY OF A GRASSHOPPER DEFINTION

Grasshopper allows you to create visual programs called definitions. These definitions are made up of nodes connected by wires. The following chapter introduces Grasshopper objects and how to interact with them to start building definitions.



1.2.1. GRASSHOPPER OBJEKTTYPEN

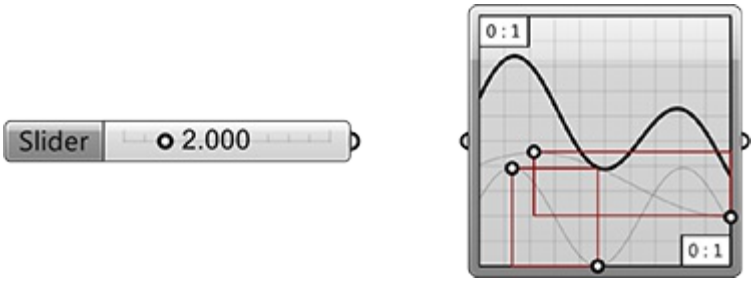
Grasshopper besteht aus zwei primaeren Typen von Benutzerobjekten: Parameter und Komponenten. Parameter speichern Daten, wobei Komponenten Aktionen ausfuehren, welche wiederum Daten erzeugen. Der einfachste Weg, um Grasshopper zu verstehen ist, sich daran zu erinnern, dass wir Daten nutzen werden, um die Eingabe von Aktionen zu definieren (was neue Daten erzeugt, die wir im weiteren Verlauf nutzen koennen).

1.2.1.1. PARAMETER

Parameter speichern die Daten – Zahlen, Farben, Geometrien , u.a. – die wir durch den Graph unserer Definition senden. Parameter sind Containerobjekte, welche gewoehnlich als kleine rechteckige Kaestchen mit einer einzelnen Eingabe und Ausgabe angezeigt werden. Wir koennen Parameter auch an der Form des Symbols erkennen, da alle Parameterobjekte eine sechseckige Einfassung haben.

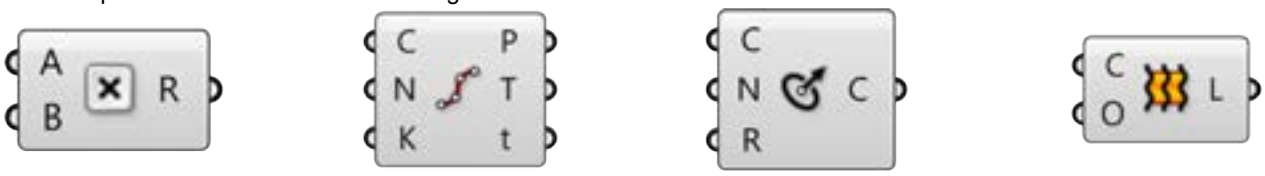
Geometrieparameter koennen Geometrien aus Rhino referenzieren oder Geometrien von anderen Komponenten aufnehmen. Punkt- und Kurvenobjekte sind beide Geometrieparameter.

Eingabeparameter sind dynamische Schnittstellenobjekte, die es ermoeglichen direkt mit der Definition zu interagieren. Die Schieberegler und der Graphmapper sind jeweils Eingabeparameter.



1.2.1.2. KOMPONENTEN

Komponenten fuehren Aktionen auf Basis der Eingaben aus, welche sie erhalten. Es gibt viele verschiedene Typen von Komponenten fuer verschiedene Aufgaben.



1 2 3 4

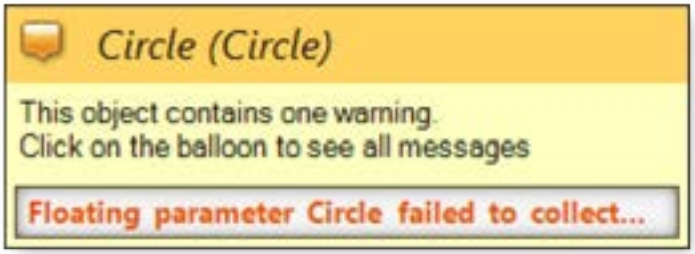
1. Die Multiplikations-Komponente ist ein Operator, der das Produkt zweier Zahlen berechnet.
2. Die Divide Curve-Komponente arbeitet mit Geometrien. Sie teilt eine Kurve in gleich lange Segmente.

1. Die Circle CNR-Komponente konstruiert eine Kreisgeometrie von den Eingaben, dem Mittelpunkt, Normalenvektor und Radius.
2. Die Loft-Komponente konstruiert eine Loftflaeche von zwei Kurven.

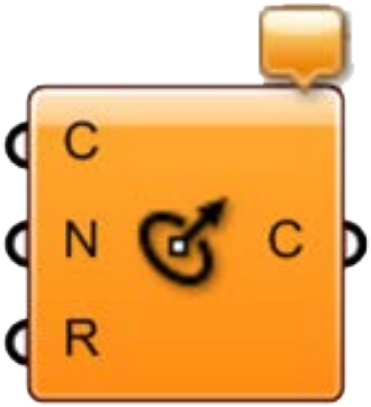
1.2.1.3. OBJEKTFARBEN

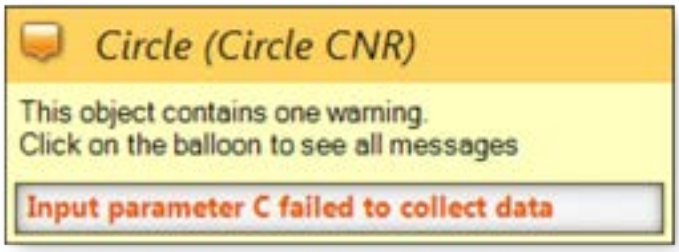
Wir koennen einige Informationen ueber die Objekte erhalten, wenn wir ihre Farben betrachten. Schauen wir uns die grundsaetzliche Farbcodierung von Grasshopper gemeinsam an. Ein Parameter, der keine Warnungen oder

Fehlermeldungen enthaelt, wird in hellgrau angezeigt. Diese Objektfarbe zeigt an, dass dieser Parameter einwandfrei arbeitet. Ein Parameter, der Warnungen enthaelt, wird als orangenes Kaestchen angezeigt. Objekte, die keine Daten als Eingabe erhalten, sind verdaechtig, weil sie nicht zur Grasshopperdefinition beitragen. Deshalb werden alle Parameter, wenn sie frisch hinzugefuegt werden, erst einmal orange dargestellt, um anzuzeigen, dass sie keine Daten enthalten und keinen Einfluss auf das Ergebnis der Definition haben. Standardmaessig erhalten orange angezeigte Parameter und Komponenten einen kleinen Ballon an der rechten, oberen Ecke des Objekts. Sobald du mit der Maus ueber den Ballon faehrst, wird er Informationen darueber anzeigen, warum die Warnung aufgetreten ist. Sobald ein Parameter Daten enthaelt oder definiert, wird er in grau angezeigt und der Ballon verschwindet.



Eine Komponente ist immer ein staerker eingebundenes Objekt, weshalb wir seine Eingabe und Ausgabe bewusst verstehen und anschliessend koordinieren muessen. Wie bei den Parametern wird eine Komponente, die Warnungen enthaelt, orange dargestellt. Behalte im Hinterkopf, dass Warnungen nicht gezwungenermassen schlecht sind, sondern dass Grasshopper dich mit ihnen lediglich auf ein potenzielles Problem in der Definition hinweisen moechte.





Eine Komponente, die weder Warnungen noch Fehlermeldungen enthaelt, wird in hellgrau angezeigt.

Eine Komponente, deren Vorschau deaktiviert wurde, wird in einem etwas dunkleren Grauton dargestellt. Es gibt zwei Moeglichkeiten, um die Vorschau einer Komponente zu deaktivieren. Zuerst einmal kannst du einfach einen Rechtsklick auf der Komponente ausfuehren und den Vorschauschalter umschalten. Um die Vorschau verschiedener Komponenten gleichzeitig auszuschalten, musst du die gewuenschten Komponenten auswaehlen und dann den entsprechenden Schalter (Mann mit verbundenen Augen) im Dropdownmenue auswaehlen, nachdem du irgendwo auf der Leinwand einen rechten Mausklick ausgefuehrt hast.

Eine deaktivierte Komponente wird in einem stumpfen Grauton dargestellt. Um eine Komponente zu deaktivieren, kannst du einen Rechtsklick auf einer Komponente machen und den Disable-Schalter betaetigen, oder die gewuenschten Komponenten auswaehlen und nach einem Rechtsklick auf der Leinwand die Option Disable auswaehlen. Deaktivierte Komponenten senden keine Daten mehr an nachgelagerte Komponenten.

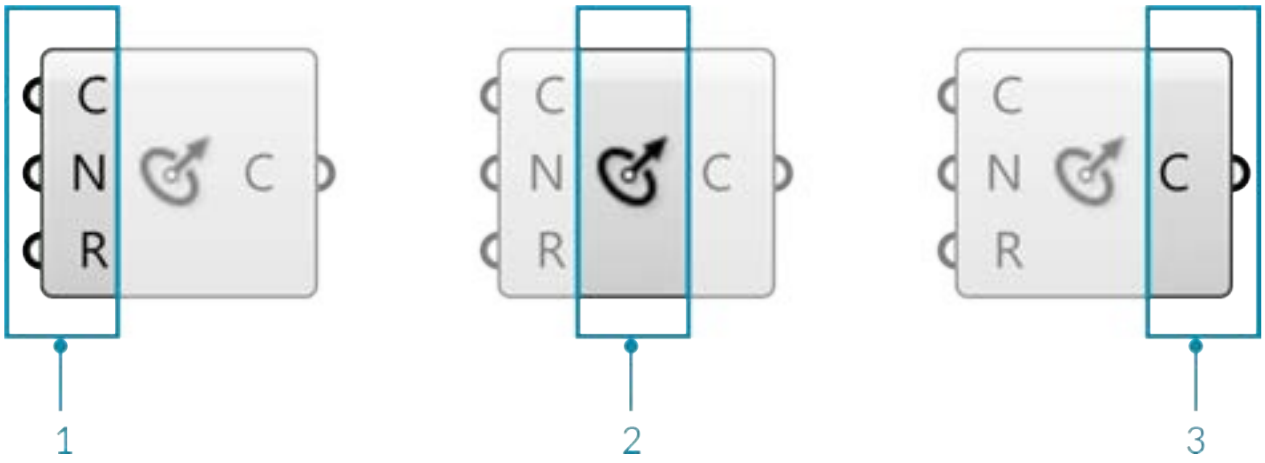
Eine ausgewaehlte Komponente wird in hellgruen angezeigt. Falls die Komponente Geometrien in der Rhinoszene

erzeugt hat, werden diese ebenfalls gruen, um dir eine optische Rueckmeldung zu geben. Eine Komponente, die mindestens eine Fehlermeldung enthaelt, wird rot dargestellt. Die Fehlermeldung kann durch die Komponente selbst oder ihre Eingabe und Ausgabe ausgeloest werden.

1. Ein Parameter ohne Warnungen und Fehlermeldungen
2. Ein Parameter mit Warnungen
3. Eine Komponente mit Warnungen
4. Eine Komponente ohne Warnungen und Fehlermeldungen
5. Eine Komponente mit deaktivierter Vorschau
6. Eine deaktivierte Komponente
7. Eine ausgewaehlte Komponente
8. Eine Komponente mit einer Fehlermeldung

1.2.2. GRASSHOPPER COMPONENT PARTS

Components are the objects you place on the canvas and connect together with Wiresto form a visual program. Components can represent Rhino Geometry or operationslike Math Functions. Components have inputs and outputs.



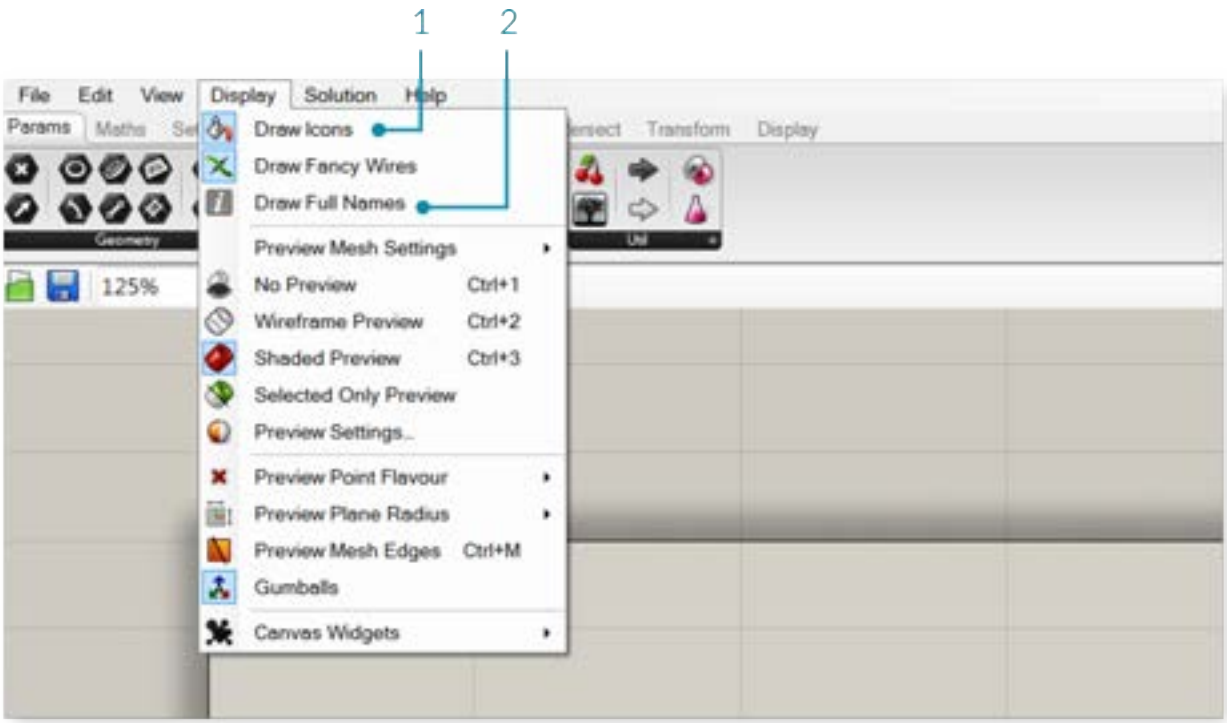
1. The three input parameters of the Circle CNR component.
2. The Circle CNR component area.
3. The output parameter of the Circle CNR component.

Acomponent requires data in order to perform its actions, and it usually comes up with a result. That is why most components have a set of nested parameters, referred to as Inputs and Outputs, respectively. Input parameters are positioned along the left side, output parameters along the right side.

There are a few Grasshopper components that have inputs but no outputs, or vice versa. When a component doesn't have inputs or outputs, it will have a jagged edge.

1.2.2.1. LABEL VS ICON DISPLAY

Every Grasshopper object has a unique icon. These icons are displayed in the center area of the object and correspond to the icons displayed in the component palettes. Objects can also be displayed with text labels. To switch between icon and label display, Select "Draw Icons" from the display menu. You can also select "Draw Full Names" to display the full name of each object as well as its inputs and outputs.

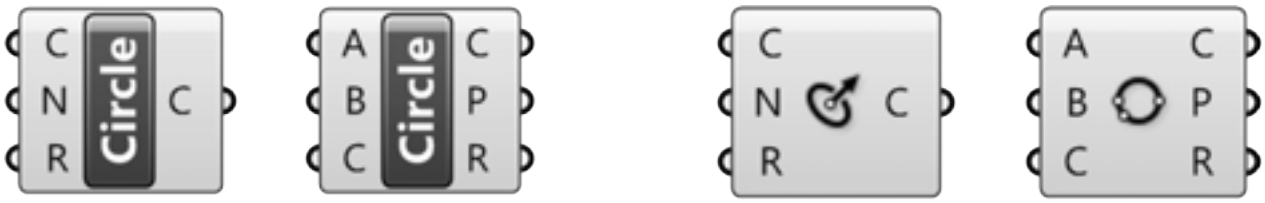


1. Switch between Icon and Label display.
2. Display the full name of the component and its inputs and outputs

1 2 3

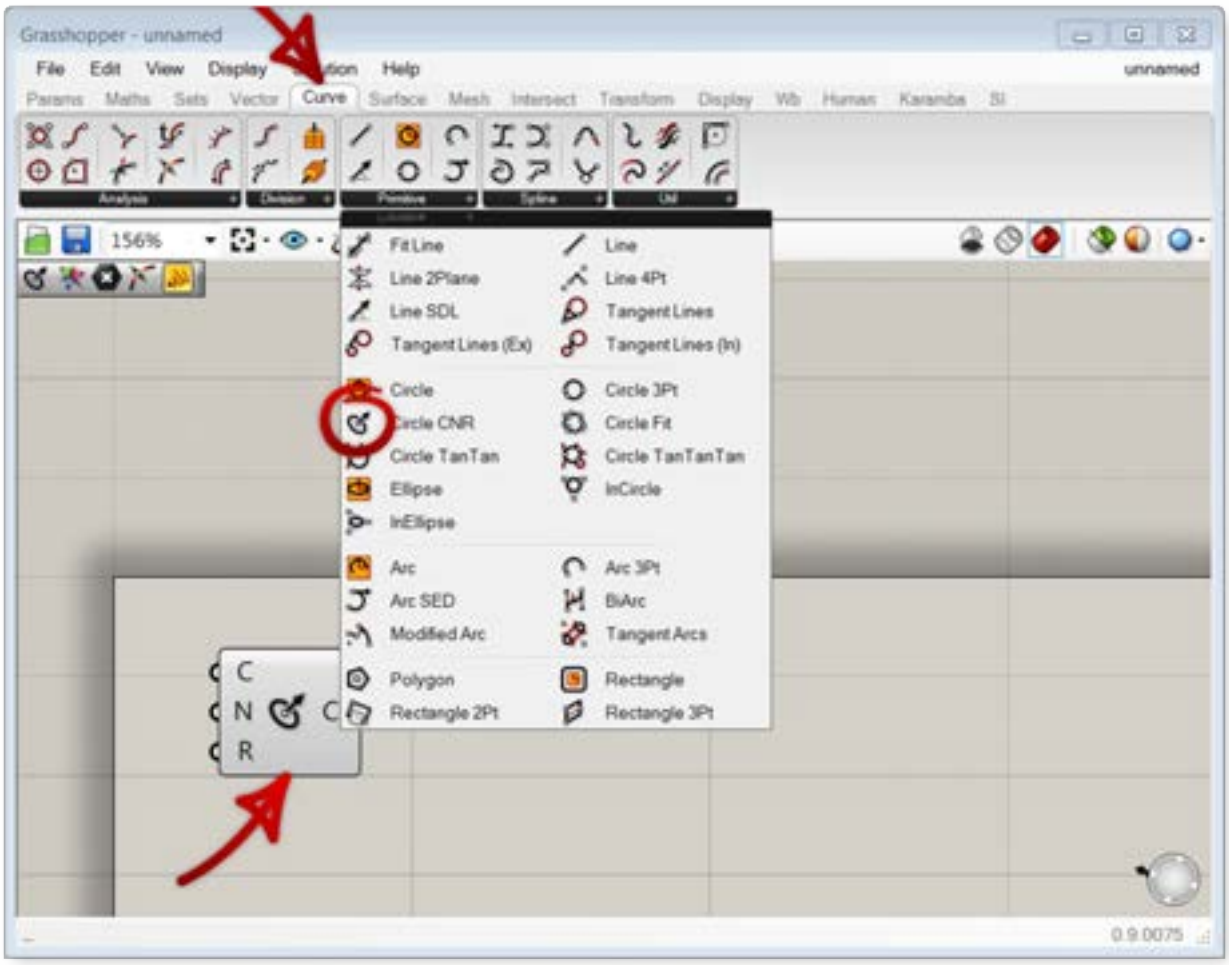
1. The Circle CNR component in Label Display
2. The Circle CNR component in Icon Display
3. The Circle CNR component with full names displayed

We reccommend using icon display to familiarize yourself with the component icons so you can quickly locate them in the palettes. This will also enable you to understand definitions at a glance. Text labels can be confusing because different components may share the same label.



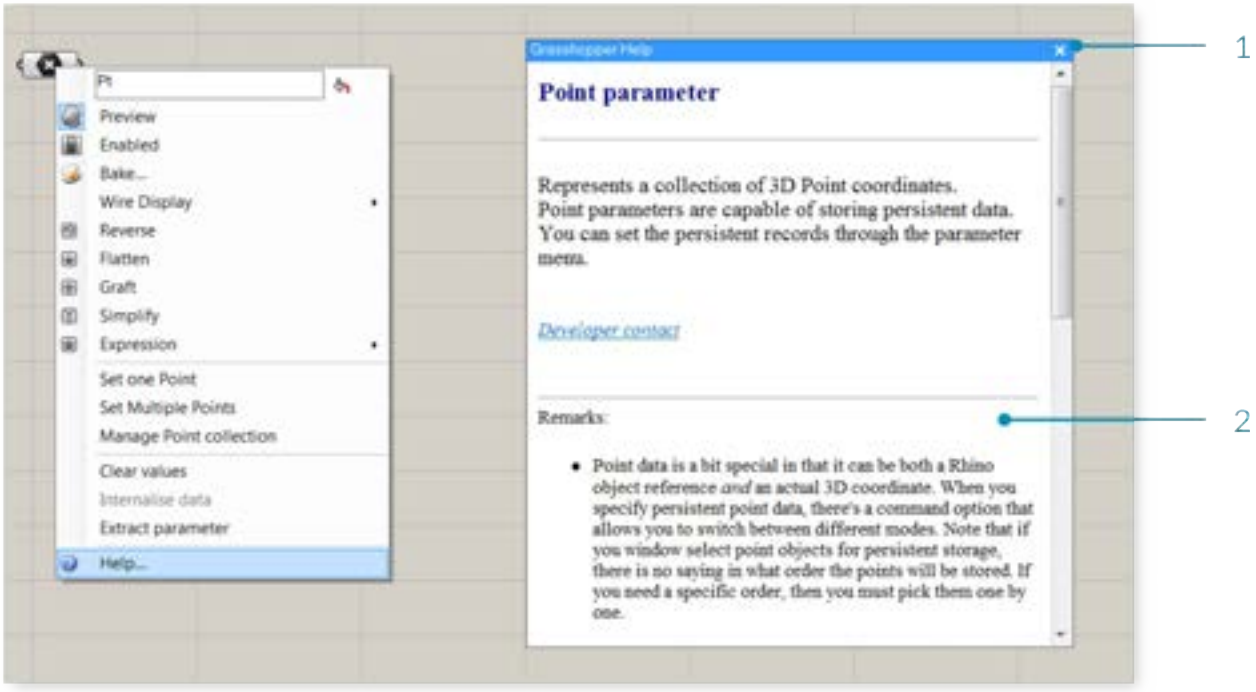
Circle CNR and Circle 3pt have the same label, but different icons.

One feature that can help you familiarize yourself with the location of components in the palettes is holding down Ctrl + Alt and clicking on an existing component on the canvas. This will reveal its location in the palette.



1.2.2.2. COMPONENT HELP

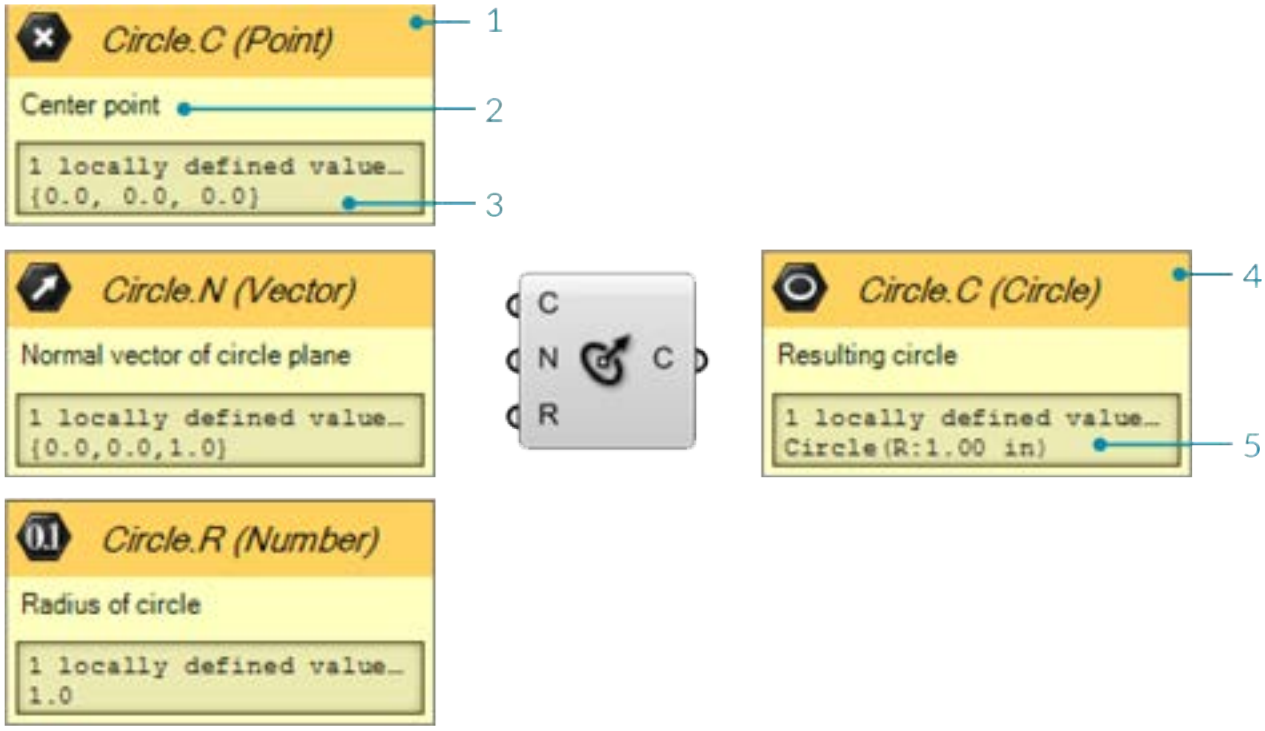
Right clicking an object and selecting “Help” from the drop-down menu will open a Grasshopper help window. The help window contains a more detailled description of the object, a list of inputs and outputs, as well as remarks.



1. Grasshopper help window for the Point parameter
2. The remarks in the help window give additional insight about the point parameter.

1.2.2.3. TOOL TIPS

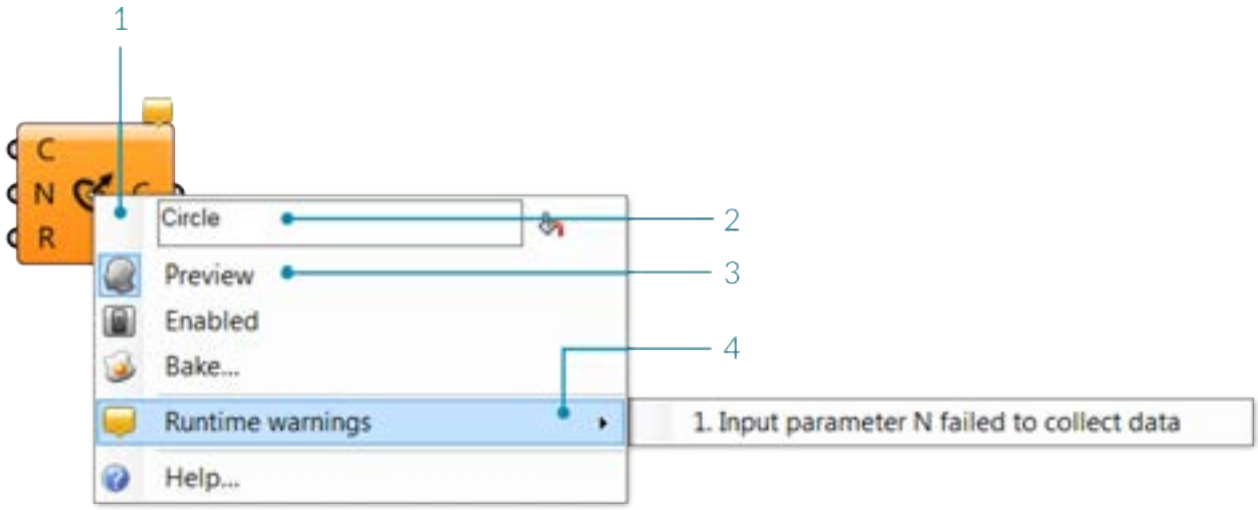
Component inputs are expecting to receive certain types of data, for example a Component might indicate that you should connect a point or plane to its input. When you hover your mouse over the individual parts of a Component object, you'll see different tooltips that indicate the particular type of the sub-object currently under the mouse. Tooltips are quite informative since they tell you both the type and the data of individual parameters.



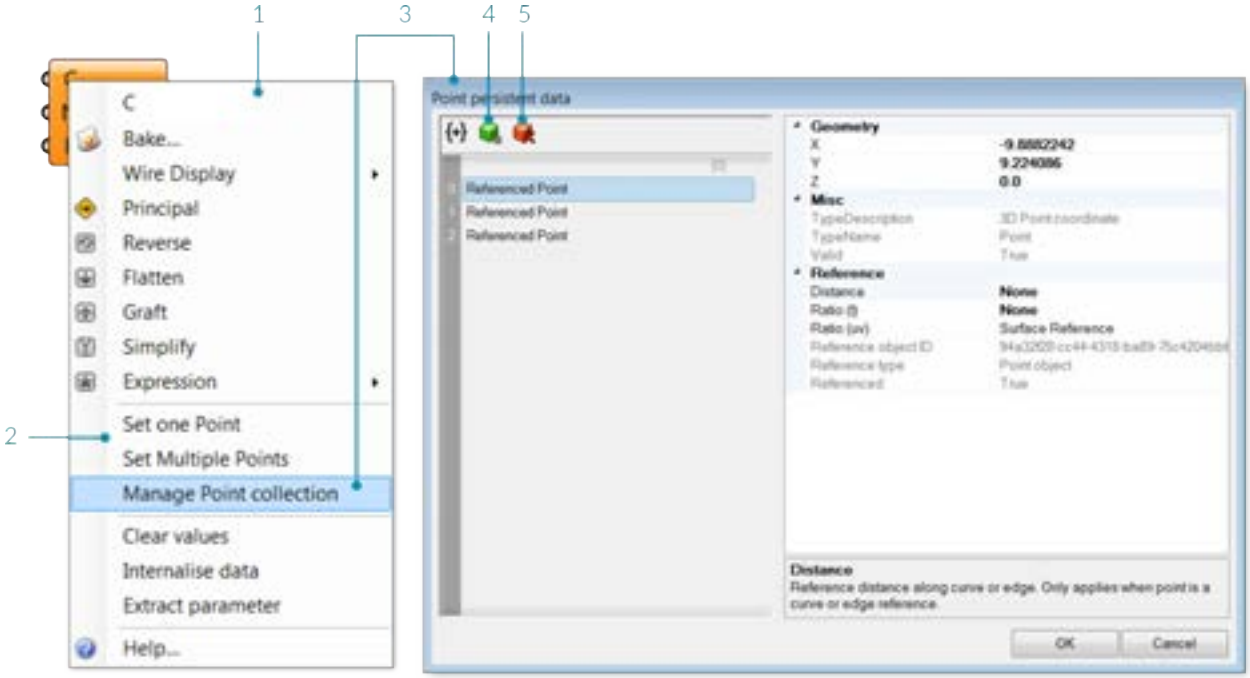
1. Header of the tooltip shows the icon for the input type, the name of the Component, the label for the input, and the input type again in text format.
2. The flain language description of what the input is for the Component.
3. Any values defined for the input - either locally or from its connected wire.
4. The header of the output tooltip provides the same detail os for inputs, but for the corresponding output.
5. The result of the component's action.

1.2.2.4. CONTEXT POPUP MENUS

All objects on the Canvas have their own context menus that expose their settings and details. You can access this context menu by right-clicking on the center area of each component. Inputs and outputs each have their own context menus which can be accessed by right-clicking them.



1. Component context menu.
2. Editable text field that lists the name of the object.
3. Preview flag - indicates whether or not the geometry produced by this object will be visible in the Rhino viewports. Switching off preview will speed up both the Rhino viewport frame-rate and the time taken for a solution.
4. Runting warnings - lists warnings that are hindering the functioning of the component.

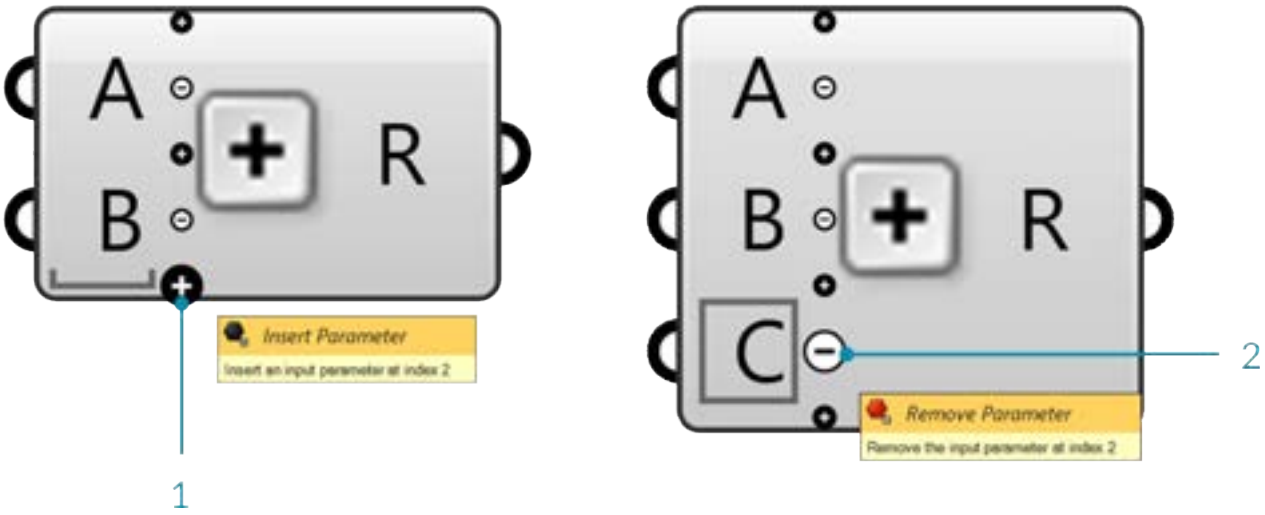


1. C input context menu.
2. Set one or multiple points - allows you to select reference geometry in the Rhino viewport.
3. Manage Point collection - opens a dialog that allows you to add or remove points from the point collection and view information about each point.
4. Add item to collection.
5. Delete selection.

1.2.2.5. ZOOMABLE USER INTERFACE

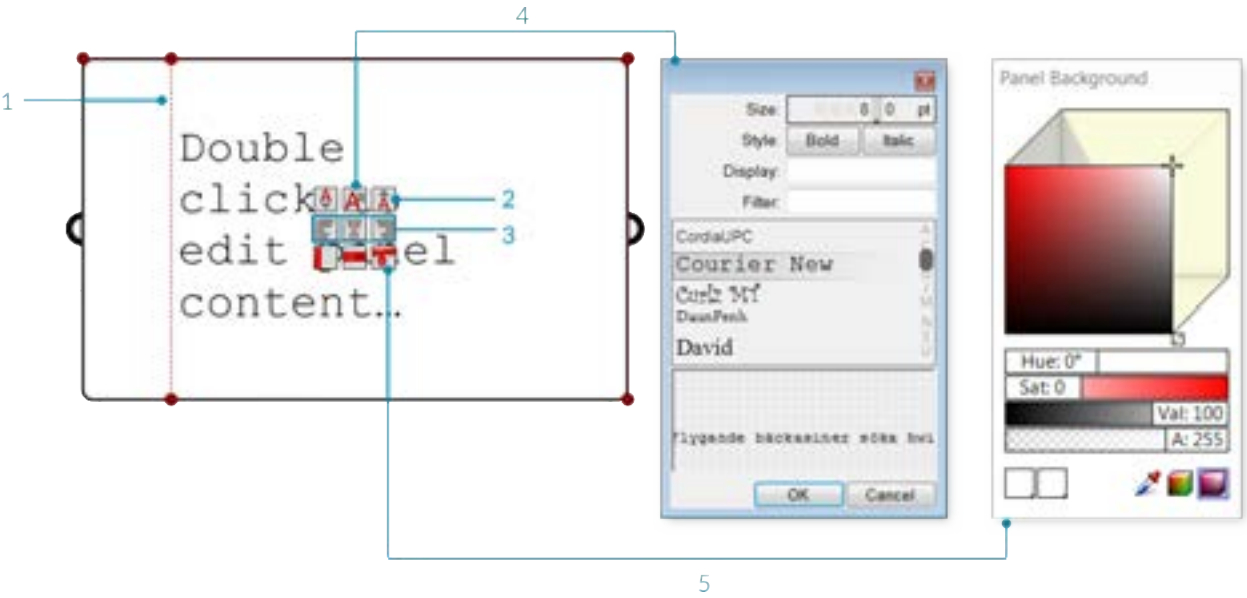
Some components can be modified to increase the number of inputs or outputs through the Zoomable User Interface (ZUI). By zooming in on the component on the canvas, an additional set of options will appear which

allows you add or remove Inputs or Outputs to that component. The Addition component allows you to add inputs, representing additional items for the addition operation.



1. Click the + sign to add an Input.
2. Click the - sign to remove an Input.

The panel component also has a zoomable user interface. APanel is like a Post-It™ sticker. It allows you to add little remarks or explanations to a Document. You can change the text through the menu or by double-clicking the panel surface. Panels can also receive and display data from elsewhere. If you plug an output into a Panel, you can see the contents of that parameter in real-time. All data in Grasshopper can be viewed in this way. When you zoom in on a panel, a menu appears allowing you to change the background, font, and other attributes. These options are also available when you right-click the panel



1. Drag grips to adjust panel margins.
2. Increase or reduce the font size of the panel content.
3. Change the alignment of panel content.
4. Select a font for ponel conent.
5. Select a color for the panel background. You can set a new default color for your panels by right clicking the panel and selecting "Set Defaut Color".

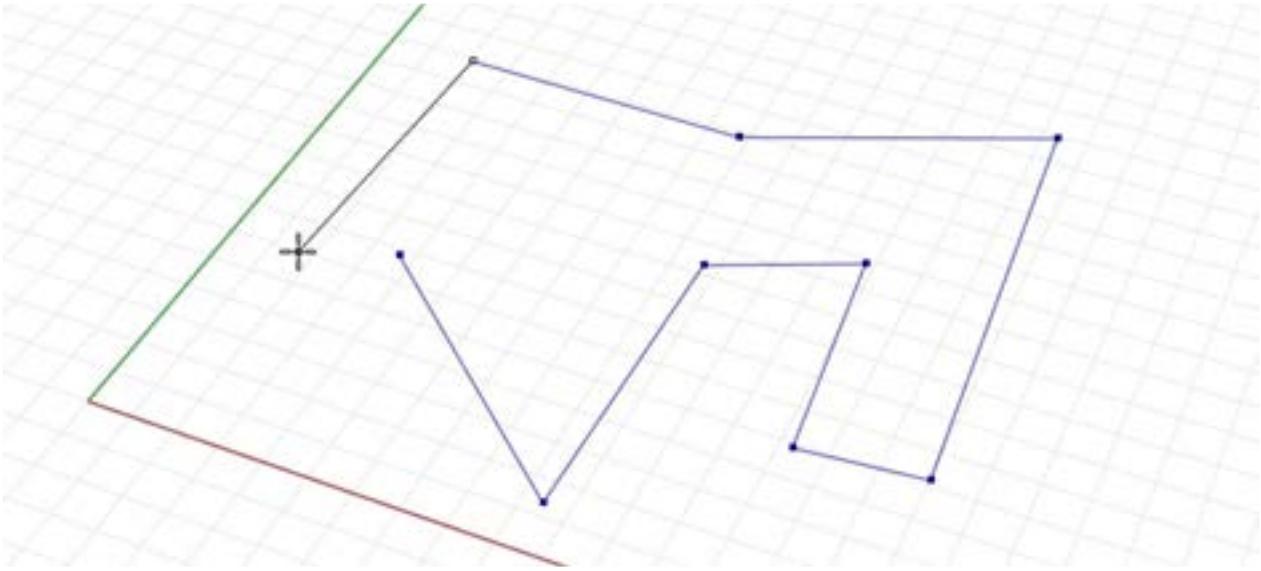
1.2.3. DATA TYPES

Most parameters can store two different kinds of data: Volatile and Persistent. Volatile data is inherited from one or more sources and is destroyed (i.e. recollected) whenever a new solution starts. Persistent data is data which has been specifically set by the user.

1.2.3.1. PERSISTENT DATA

Persistent data is accessed through the menu, and depending on the kind of parameter has a different manager. A Point parameter for example allows you to set one or more points through its menu. But, let's back up a few steps and see how a Point Parameter behaves.

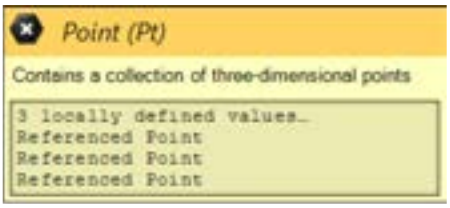
When you drag and drop a Point Parameter from the Params/Geometry Panel onto the canvas, the Parameter is orange, indicating it generated a warning. It's nothing serious, the warning is simply there to inform you that the parameter is empty (it contains no persistent records and it failed to collect volatile data) and thus has no effect on the outcome of the solution. The context menu of the Parameter offers two ways of setting persistent data: single and multiple. Right click on the parameter to set Multiple Points. Once you click on either of these menu items, the Grasshopper window will disappear and you will be asked to pick a point in one of the Rhino viewports.



Once you have defined all the points you can press Enter and they will become part of the Parameters persistent data record. This means the Parameter is now no longer empty and it turns from orange to grey. (Notice that the information balloon in the upper right corner also disappears as there are no more warnings). At this point you can use the points stored in this Parameter for any subsequent input in your definition.

1

2



3

1. The parameter is orange, indicating it contains no persistent records (and it failed to collect volatile data) and thus has no effect on the outcome of the solution. Right click on any parameter to set its persistent data.
2. Once the parameter contains some persistent data, the component will turn from orange to grey.

3. The tooltip for the point parameter shows the persistent data (a collection of referenced points) that is stored.

1.2.3.2. VOLATILE DATA

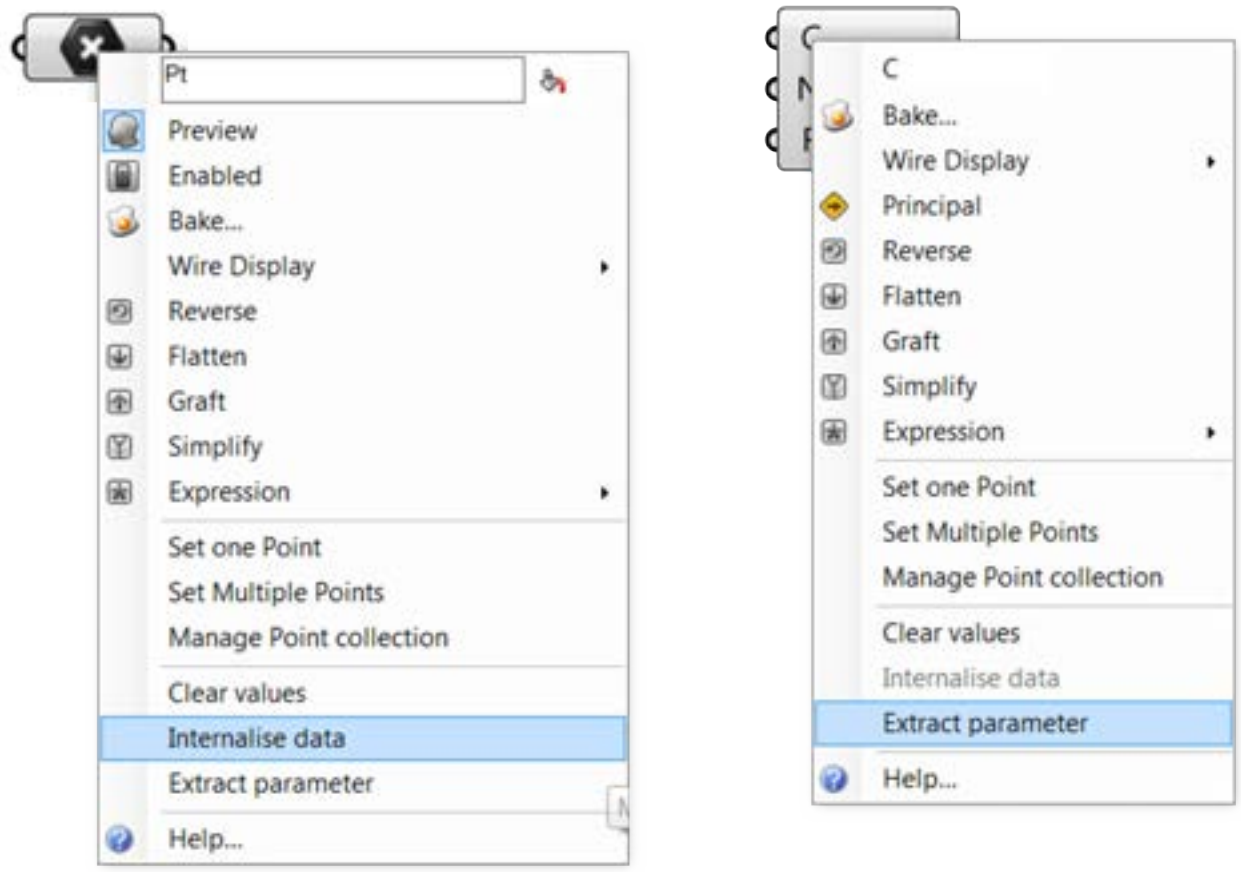
Volatile data, as the name suggests, is not permanent and will be destroyed each time the solution is expired. However, this will often trigger an event to rebuild the solution and update the scene. Generally speaking, most of the data generated 'on the fly' is considered volatile.

As previously stated, Grasshopper data is stored in Parameters (either in Volatile or Persistent form) and is used in various Components. When data is not stored in the permanent record set of a Parameter, it must be inherited from elsewhere. Every Parameter (except output parameters) defines where it gets its data from and most Parameters are not very particular. You can plug a number Parameter (which just means that it is a decimal number) into an integer source and it will take care of the conversion.

You can change the way data is inherited and stored in the context menu of a parameter or component input. To change store referenced Rhino geometry in the grasshopper definition itself, right click a parameter and select Internalise data from the menu. This is useful if you want your grasshopper definition to be independent from a Rhino file.

You can also Internalise data in a component input. Once you select Internalise data in the menu, any wires will disconnect from that input. The data has been changed from volatile to persistent, and will no longer update.

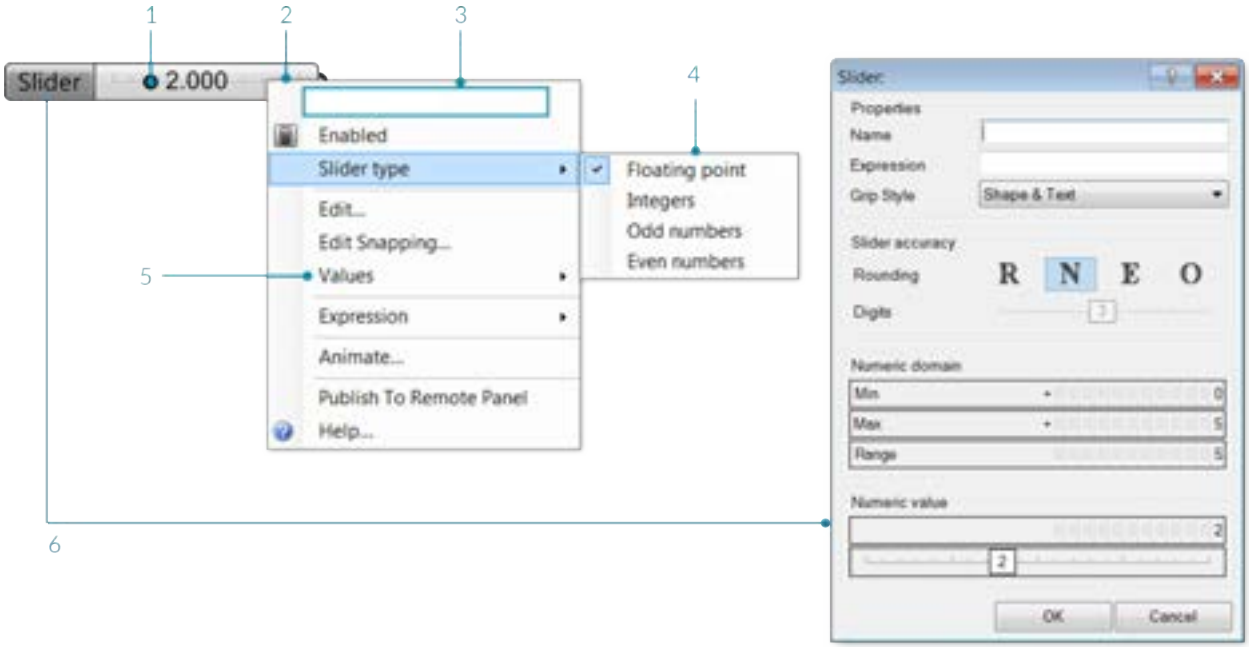
If you want the data to become volatile again, simply reconnect the wires to the input and the values will automatically be replaced. You can also right click the input and select Extract parameter. Grasshopper will create a parameter connected to the input with a wire that contains the data.



1.2.3.3. INPUT PARAMETERS

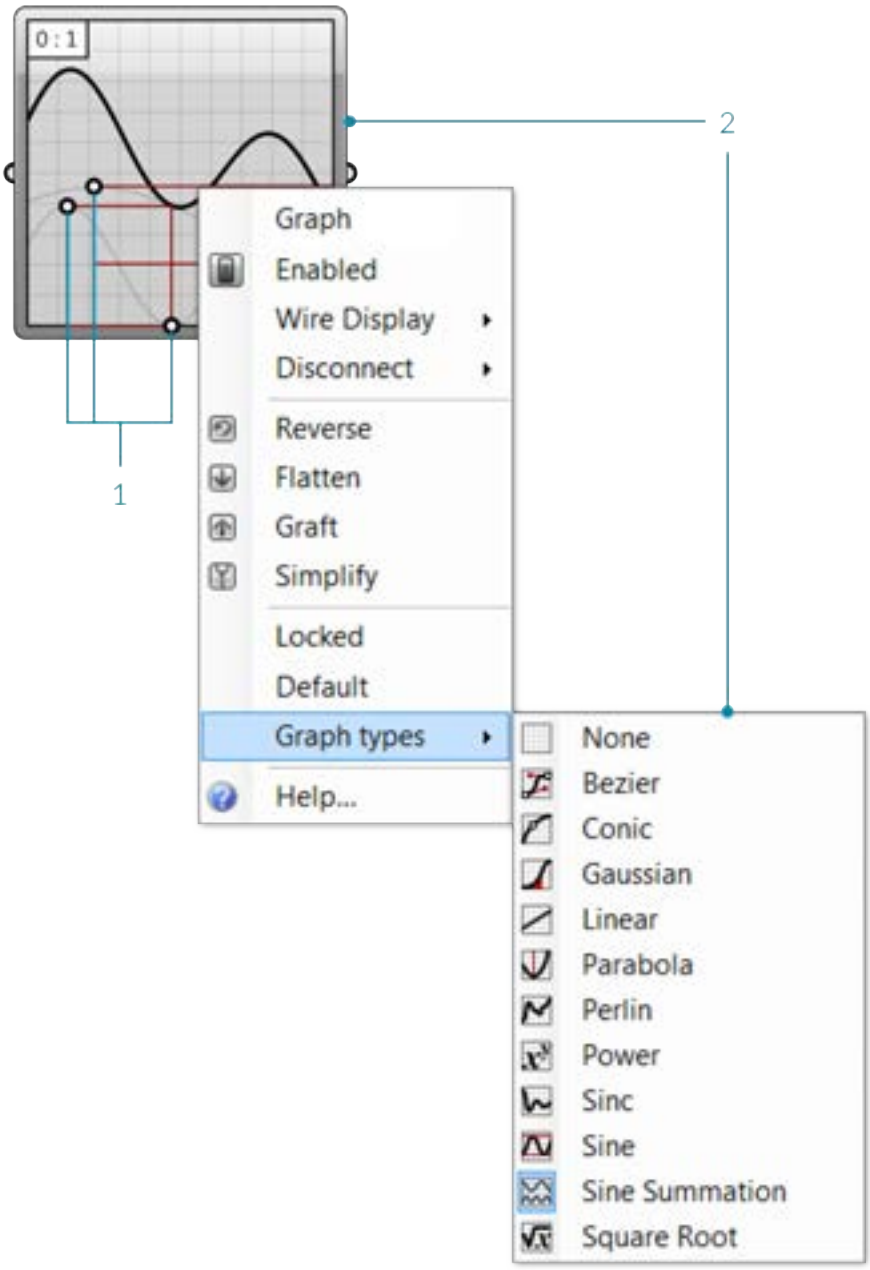
Grasshopper has a variety of Parameters that offer you the ability to interface with the data that is begin supplied to Component inputs and thereby control for changing the result of your definition. Because they Parameters that change with our input, they generate Volatile Data.

Number Slider The number slider is the most important and widely used Input Parameter. It allows us to output a value between two given extremes by interacting with its grip with our mouse. Sliders can be used to specify a value and see the change to our deifnition that comes with adjusting the grip, but a slider should also be thought of as the means to identify successful ranges of our definition.

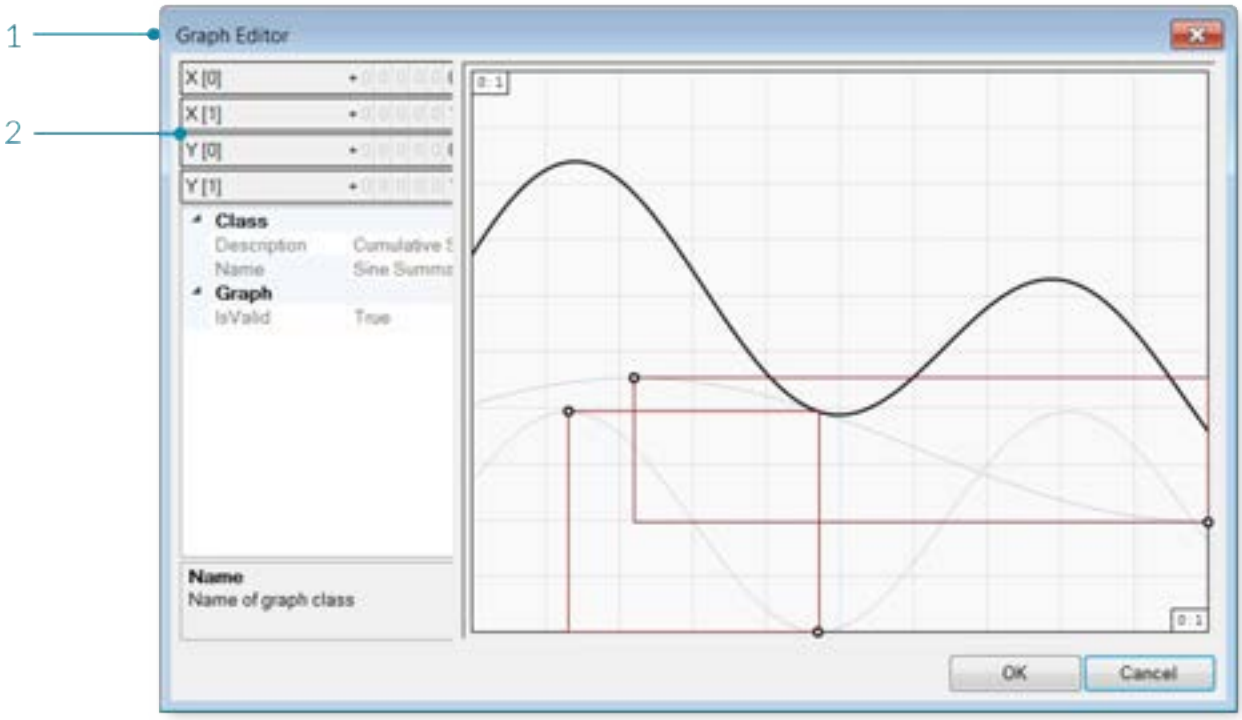


1. Drag the slider grip to change the value - each time you do this, Grasshopper will recompute the solution.
2. Right click the slider component to change the name, type, and values.
3. Editable text field for the slider name.
4. Select the type of number for the slider to use.
5. Edite the range of values.
6. Double click the name portion of the slider component to open the Slider Editor.

Graph mapper The Graph Mapper is a two-dimensional interface with which we can modify numerical values by plotting the input along the Graph's X Axis and outputting the corresponding value along the Y Axis at the X value intersection of the Graph. It is extremely useful for modulating a set of values within an institutive, grip-based interface.

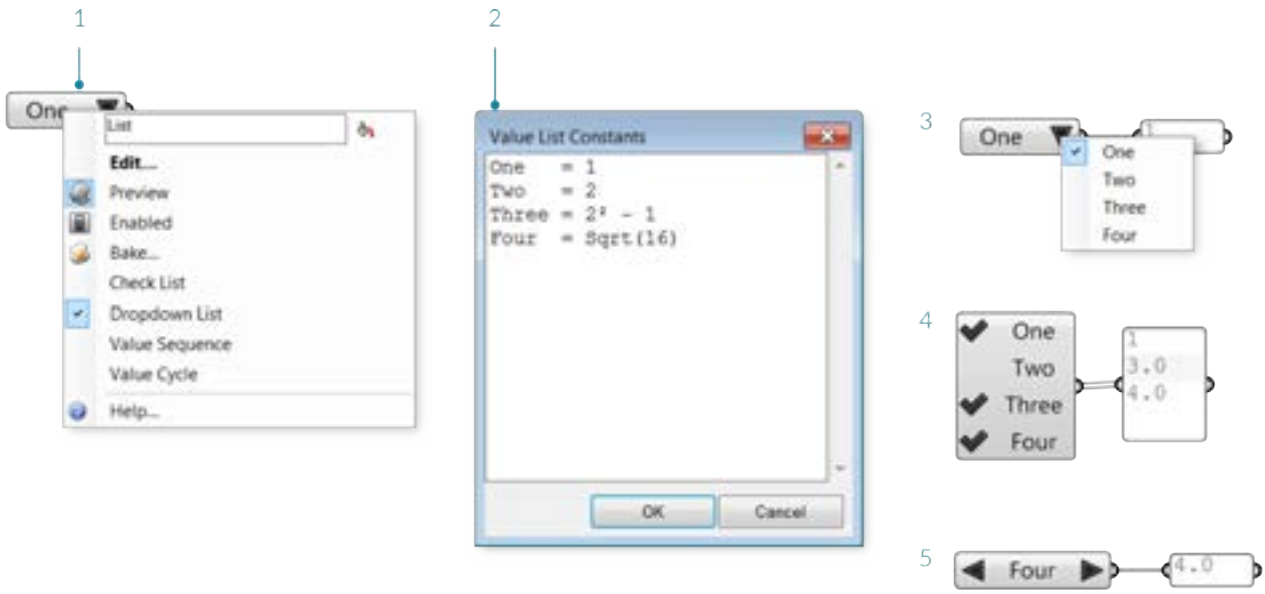


1. Move the grips to edit the graph - each time you do this, Grasshopper will recompute the solution.
2. Right click the graph mapper compenent to select the graph type.



1. Double click the graph mapper to open the Graph Editor.
2. Change the x and y domains.

Value List The Value List stores a collection of values with corresponding list of Labels associated by way of an equal sign. It is particularly useful when you want to have a few options, labeled with meaning, that can supply specific output values.



1. Right click the Value List component and select an option from the menu.
2. Double click the Value List component to open the editor and add or change values.
3. In Dropdown List mode, click the arrow to select one of the values. The solution will recompute each time you change the value.
4. In Check List mode, click next to each value to check it. The component will output all the values that are checked.
5. In Value Sequence and Value Cycle modes, click the left and right facing arrows to cycle through the values.

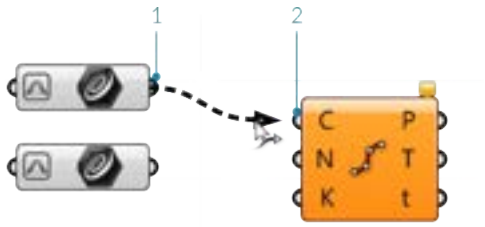
1.2.4. WIRING COMPONENTS

When data is not stored in the permanent record set of a parameter, it must be inherited from elsewhere. Data is passed from one component to another through wires. You can think of them literally as electrical wires that carry pulses of data from one object to the next.

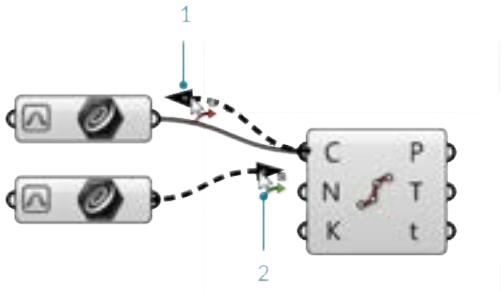
1.2.4.1. CONNECTION MANAGEMENT

To connect components, click and drag near the circle on the output side of an object. Aconnecting wire will be attached to the mouse. Once the mouse hovers over a potential target input, the wire will connect and become solid. This is not a permanent connection until you release the mouse button. It doesn't matter if we make the connections in a 'left to right' or 'right to left' manner.

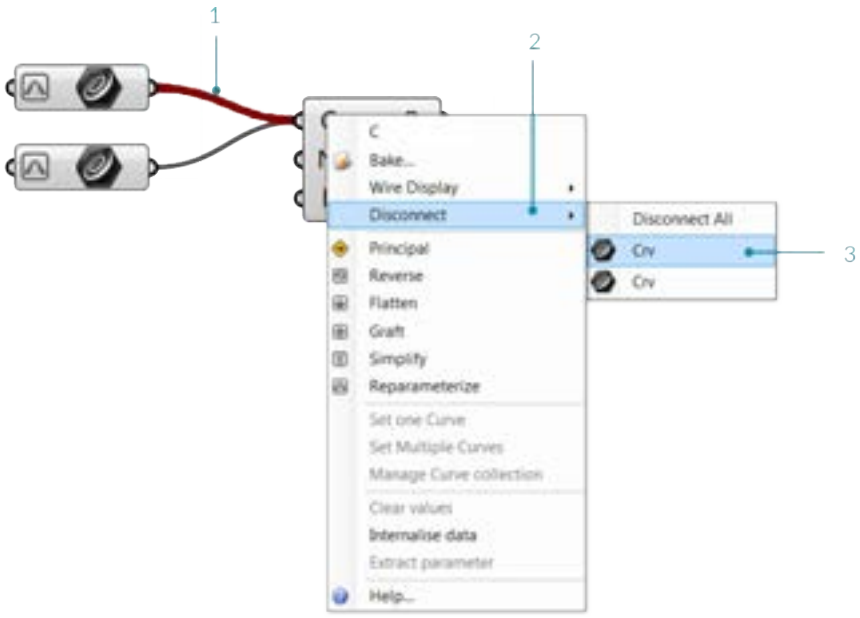
1. The Divide Curve component - divides a curve into equal length segments.
2. Curve parameter - right click and select Set One Curve to reference Rhino Geometry.



Left click and drag the wire from the output (1.) of one object to the input (2.) of another.



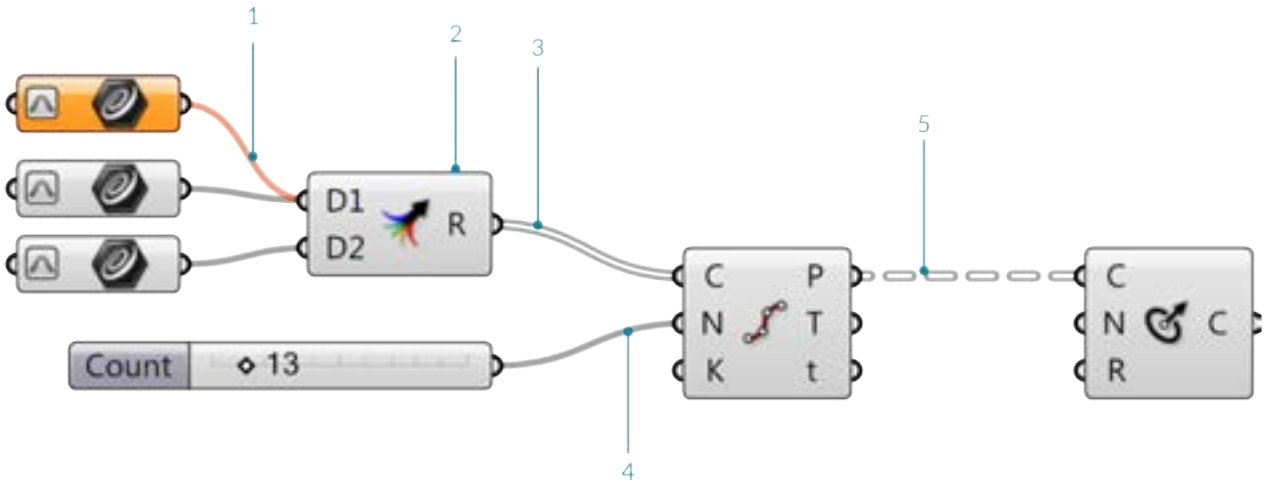
1. If you hold down CONTROL, the cursor will become red, and the targeted source will be removed from the source list.
2. By default, a new connection will erase existing connections. Hold the SHIFT button while dragging connection wires to difne multiple sources. The cursor will turn green to indicate the addition behavior.



1. You can also disconnect wires through the context popup menu - right click the grip of the input or output and select disconnect.
2. If there are multiple connections, select the one you want to disconnect from the list.
3. When you hover over an item, the wire will be highlighted in red.

1.2.4.2. FANCY WIRES

Wires represent the connections as well as the flow of data within the graph in our definition. Grasshopper can also give us visual clues as to what is flowing through the wires. The default setting for these so-called “fancy wires” is off, so you have to enable them before you can view the different types of line types for the connection wires. To do this, simply click on the Display Tab on the Main Menu Bar and select the button labeled “Draw Fancy Wires.” Fancy wires can tell you a lot of information about what type of information is flowing from one component to another.



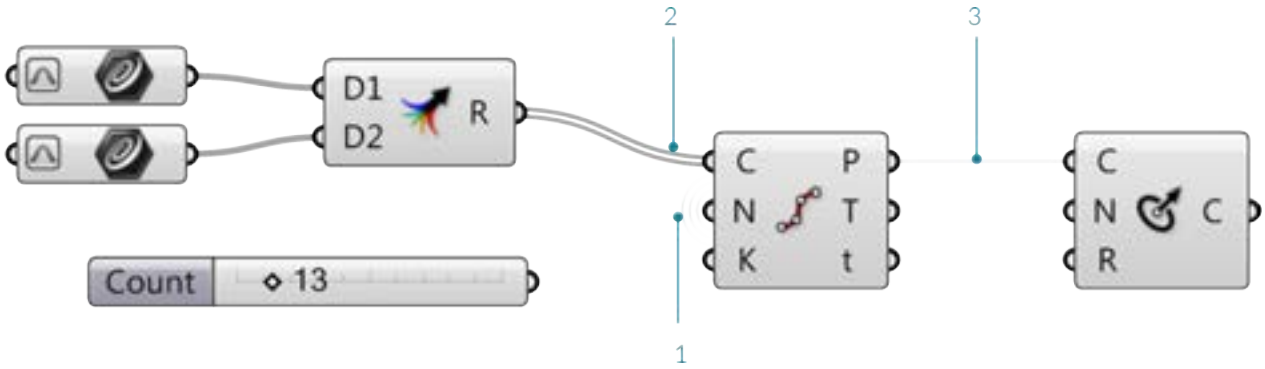
1. Empty Item – An orange wire type indicates that no information has been transferred. This parameter has generated a warning message because it contains no data, and thus no information is being sent across the wire.
2. The Merge component is an alternative to conecting more than one source to a single input.
3. List – If the information flowing out of a component contains a list of information, the wire type will be shown as a grey double line.
4. Single Item – The data flowing out of any parameter that contains a single item will be shown with a solid grey line.

5. Tree – Information transferred between components which contain a data structure will be shown in a grey double-line-dash wire type.

1.2.4.3. WIRE DISPLAY

If you have spent any great deal of time working on a single Grasshopper definition, you may have realized that the canvas can get cluttered with a nest of wires quite quickly. Fortunately, we have the ability to manage the wire displays for each input of a component.

There are three wire displays: Default Display, Faint Display, and Hidden Display. To change the wire display, simply right-click on any input on a component and select one of the views available under the Wire Display pop out menu.



1. Hidden Display – When hidden display is selected, the wire will be completely 'invisible'. The data is transferred 'wirelessly' from the source to the input parameter. If you select the source or target component, a green wire will appear to show you which components are connected to each other. Once you deselect the component, the wire will disappear.
2. Default Display – The default wire display will draw all connections (if fancy wires is turned on).
3. Faint Display – The faint wire display will draw the wire connection as a very thin, semi-transparent line. Faint and Hidden wire displays can be very helpful if you have many source wires coming into a single input.

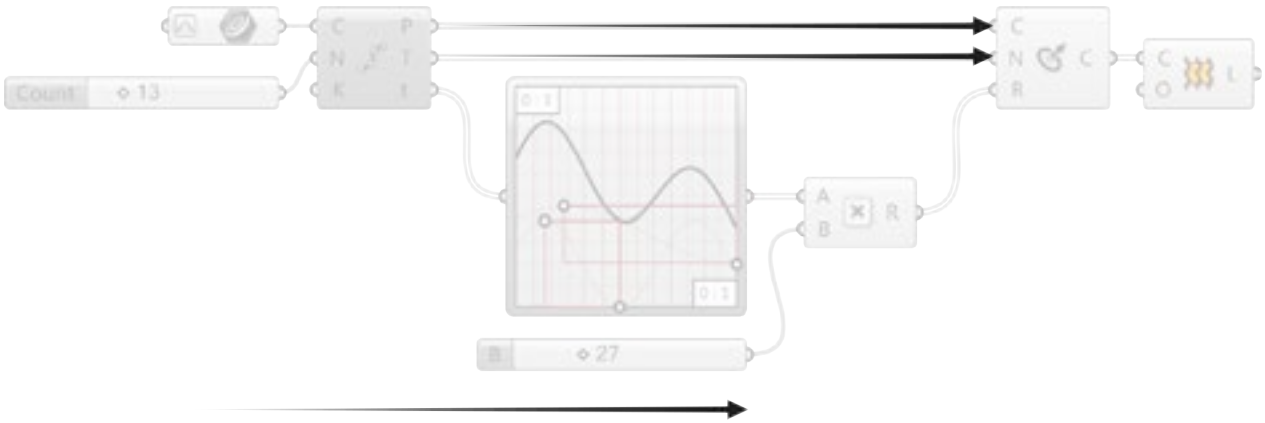
1.2.5. THE GRASSHOPPER DEFINITION

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Grasshopper Definitions have a Program Flow that represents where to start program execution, what to do in the middle and how to know when program execution is complete.

1.2.5.1. PROGRAM FLOW

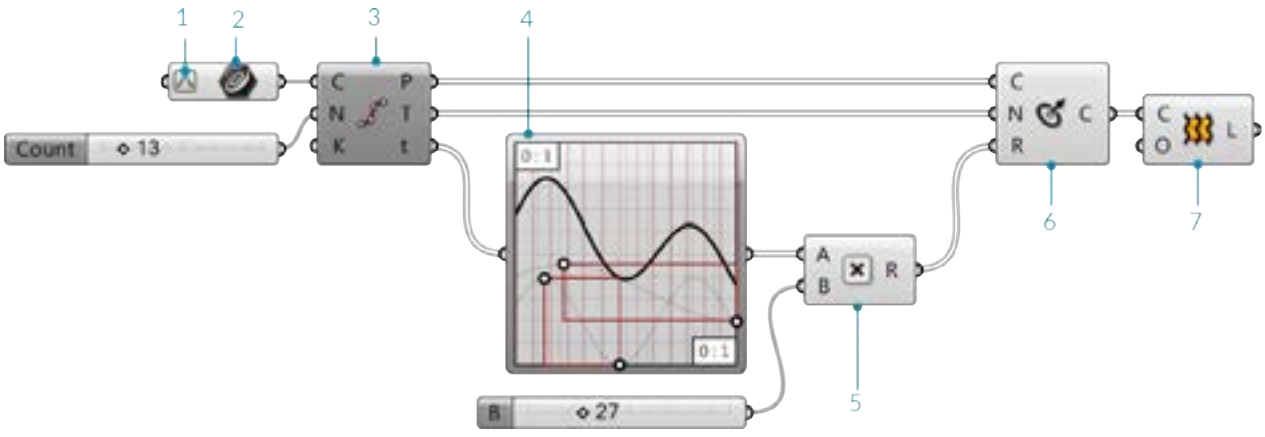
Grasshopper visual programs are executed from left to right. Reading the graph relative to the wired connections from upstream to downstream provides understanding about the Program Flow.



> Directionality of data is left to right.

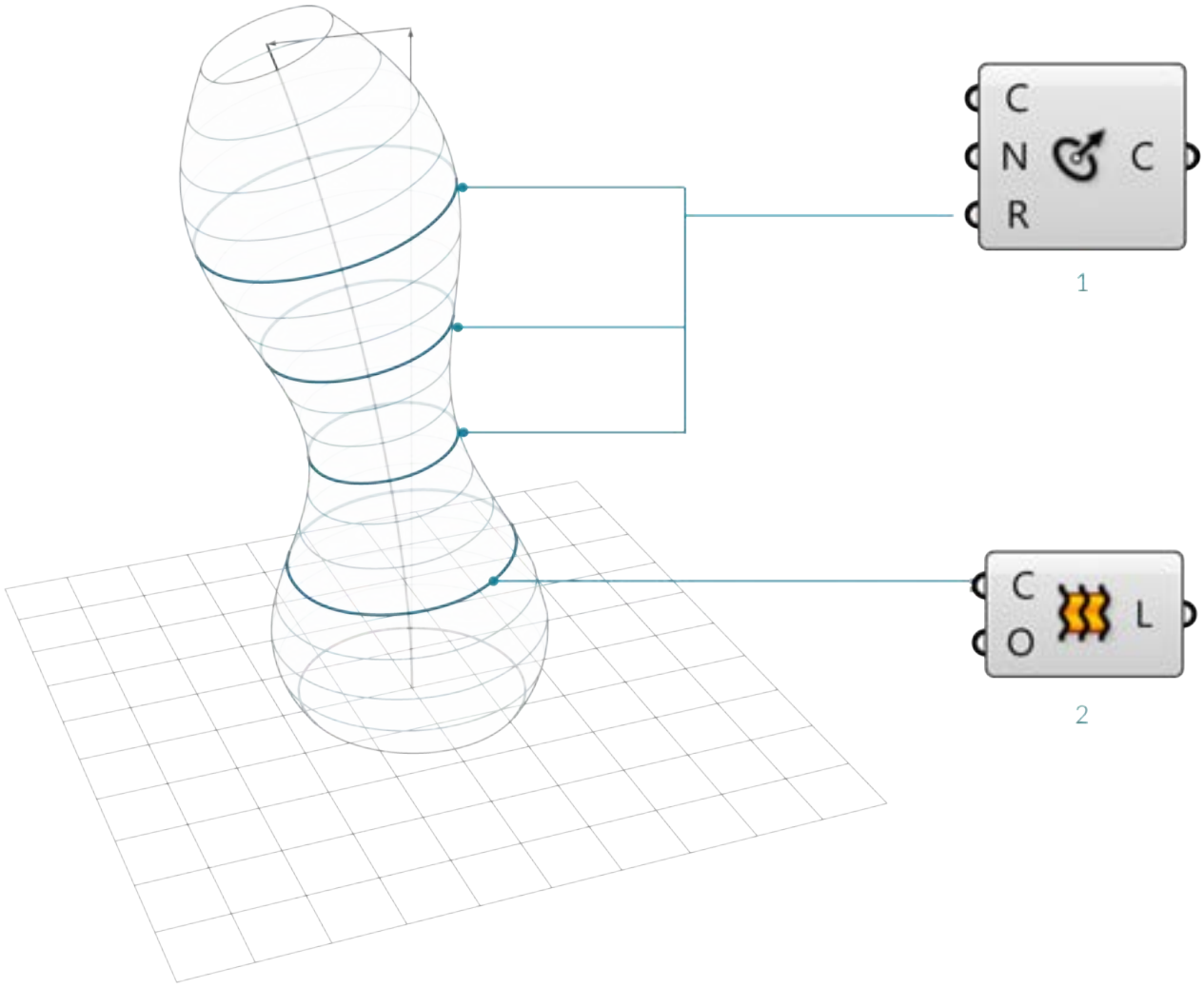
1.2.5.2. THE LOGICAL PATH

All of the objects and the wires connecting the objects represent the logical graph of our program. This graph reveals the flow of data, dependencies of any input to its wiired output. Any time our graph changes, sometimes referrred to as being “dirtied,” every downstream connection and object will be updated.



> 1. Reparameterize the curve domain between 0.0 and 1.0.
> 2. Reference a curve from Rhino.
> 3. Divide the curve into 13 equal segments.
> 4. Run the parameter values at each curve division point through the graph.
> 5. Multiply each value by 27.
> 6. Draw a circle at each division point along th ecurve, normal to the tangent vector at each pont, with a radius defined by the parameter values (t) modified by the graph mapper and multiplied by 27.

7. Loft a surface between the circles



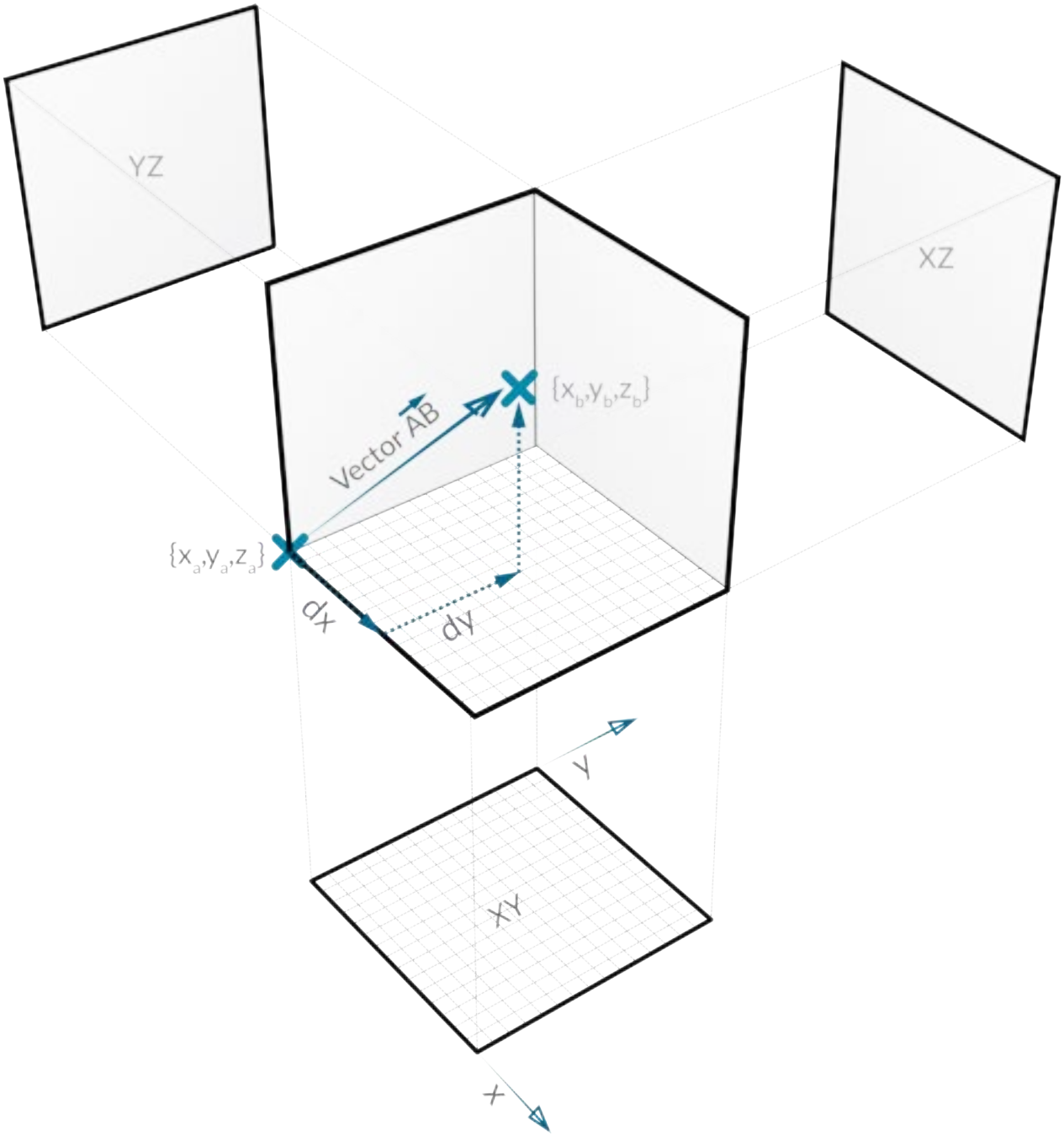
1. Variable circle radius.
2. Loft between circles.

1.3. Building Blocks of Algorithms

This chapter will introduce you to basic geometric and mathematical concepts and how they are implemented and manipulated in Grasshopper.

1.3.1. Points, Planes & Vectors

Everything begins with points. A point is nothing more than one or more values called coordinates. The number of coordinate values corresponds with the number of dimensions of the space in which it resides. Points, planes, and vectors are the base for creating and transforming geometry in Grasshopper.

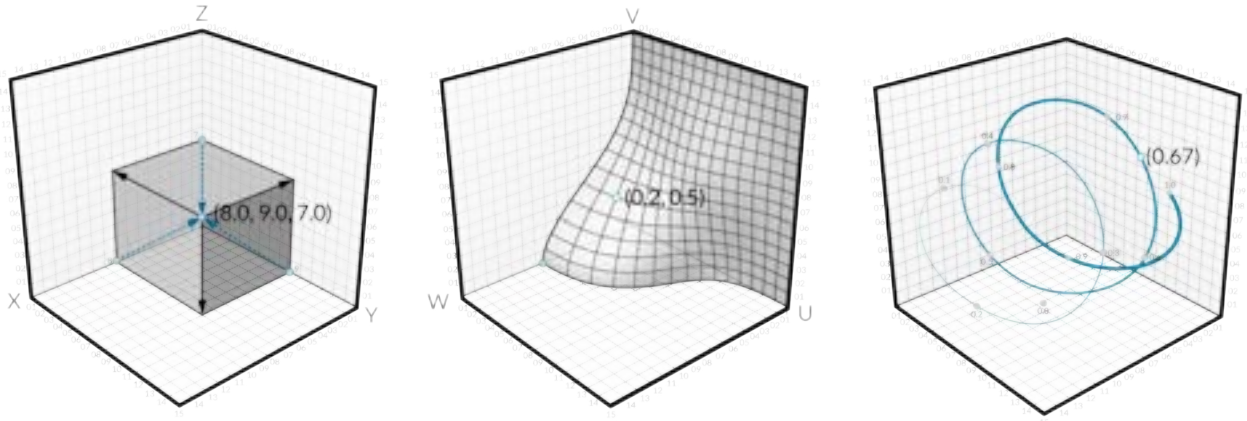


1.3.1.1 POINTS

Points in 3D space have three coordinates, usually referred to as [x,y,z]. Points in 2D space have only two coordinates which are either called [x,y] or [u,v] depending on what kind of two dimensional space we're talking about. 2D parameter space is bound to a finite surface. It is still continuous, I.e. hypothetically there are an infinite amount of points on the surface, but the maximum distance between any of these points is very much limited. 2D parameter coordinates are only valid if they do not exceed a certain range. In the example drawing, the range has been set between 0.0 and 1.0 for both [u] and [v] directions, but it could be any finite domain. A point with

coordinates [1.5, 0.6] would be somewhere outside the surface and thus invalid.

Since the surface which defines this particular parameter space resides in regular 3D world space, we can always translate a parametric coordinate into a 3D world coordinate. The point [0.2, 0.5] on the surface for example is the same as point [1.8, 2.0, 4.1] in world coordinates. Once we transform or deform the surface, the 3D coordinates which correspond with [0.2, 0.5] will change.

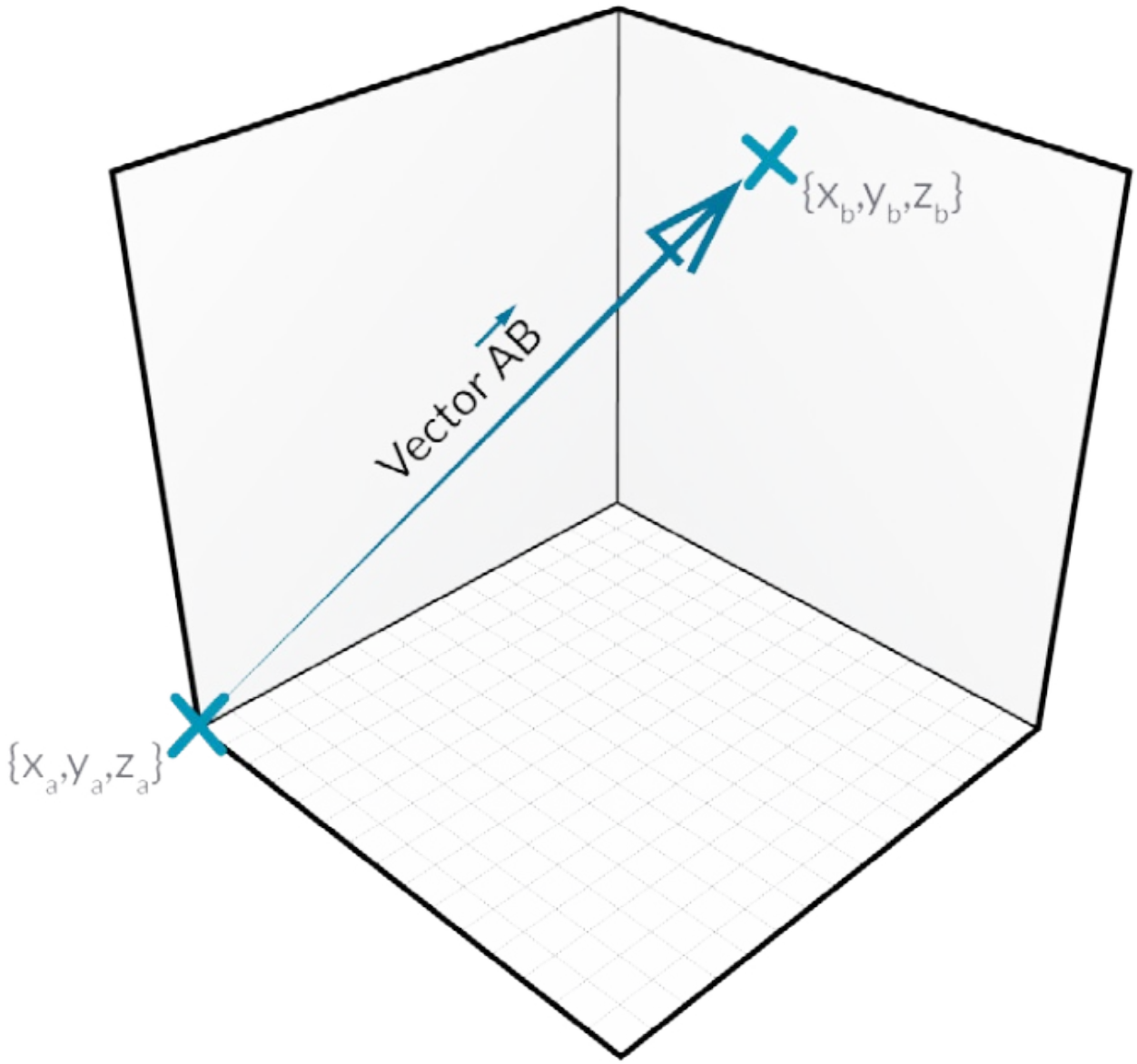


If this is a hard concept to grasp, it might help to think of yourself and your position in space. We tend to use local coordinate systems to describe our whereabouts; “I’m sitting in the third seat on the seventh row in the movie theatre”, “I’m in the back seat”. If the car you’re in is on the road, your position in global coordinates is changing all the time, even though you remain in the same back seat ‘coordinate’.

1.3.1.2. VECTORS

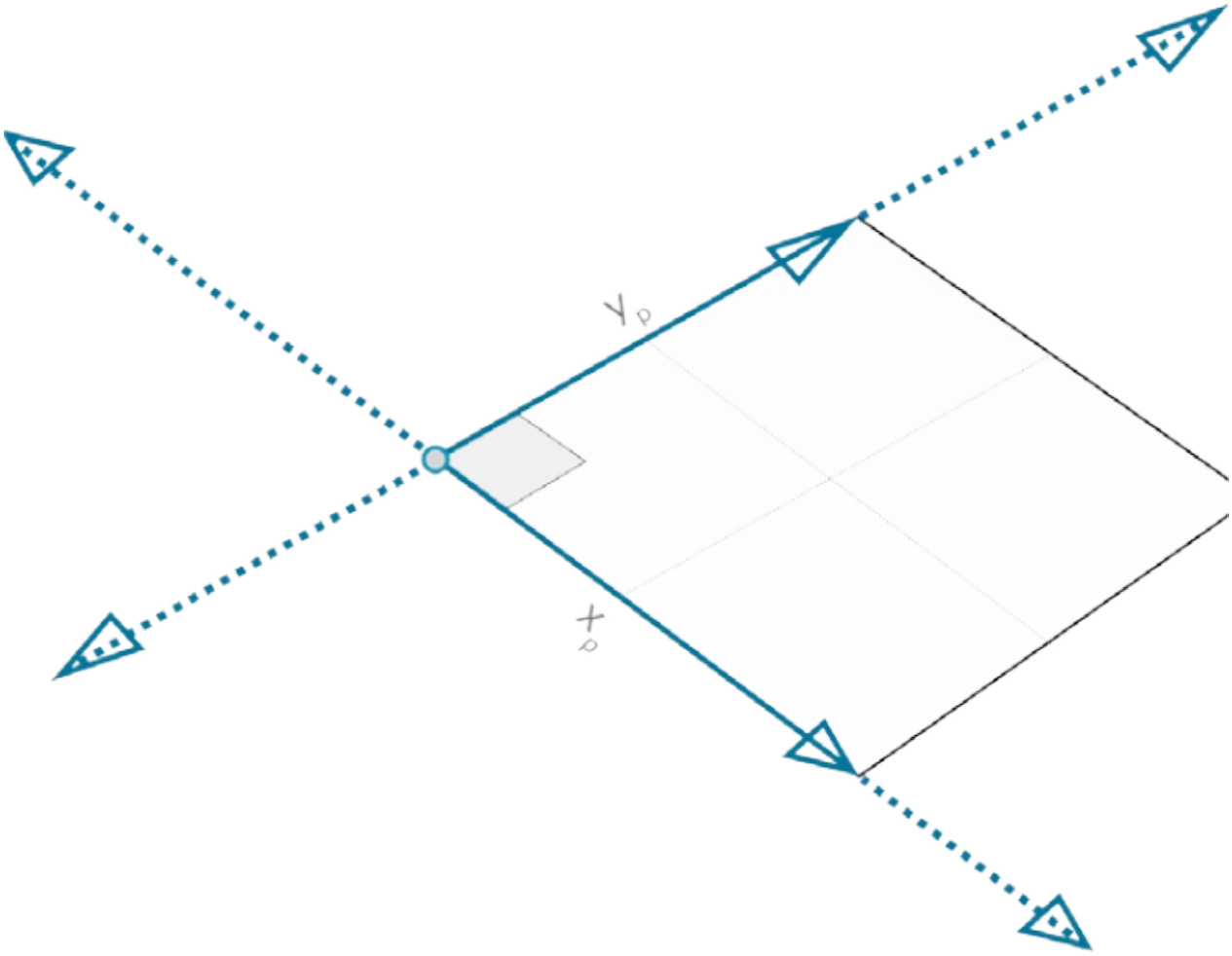
Avector is a geometric quantity describing Direction and Magnitude. Vectors are abstract; ie. they represent a quantity, not a geometrical element.

Vectors are indistinguishable from points. That is, they are both lists of three numbers so there’s absolutely no way of telling whether a certain list represents a point or a vector. There is a practical difference though; points are absolute, vectors are relative. When we treat a list of three doubles as a point it represents a certain coordinate in space, when we treat it as a vector it represents a certain direction. Avector is an arrow in space which always starts at the world origin (0.0, 0.0, 0.0) and ends at the specified coordinate.



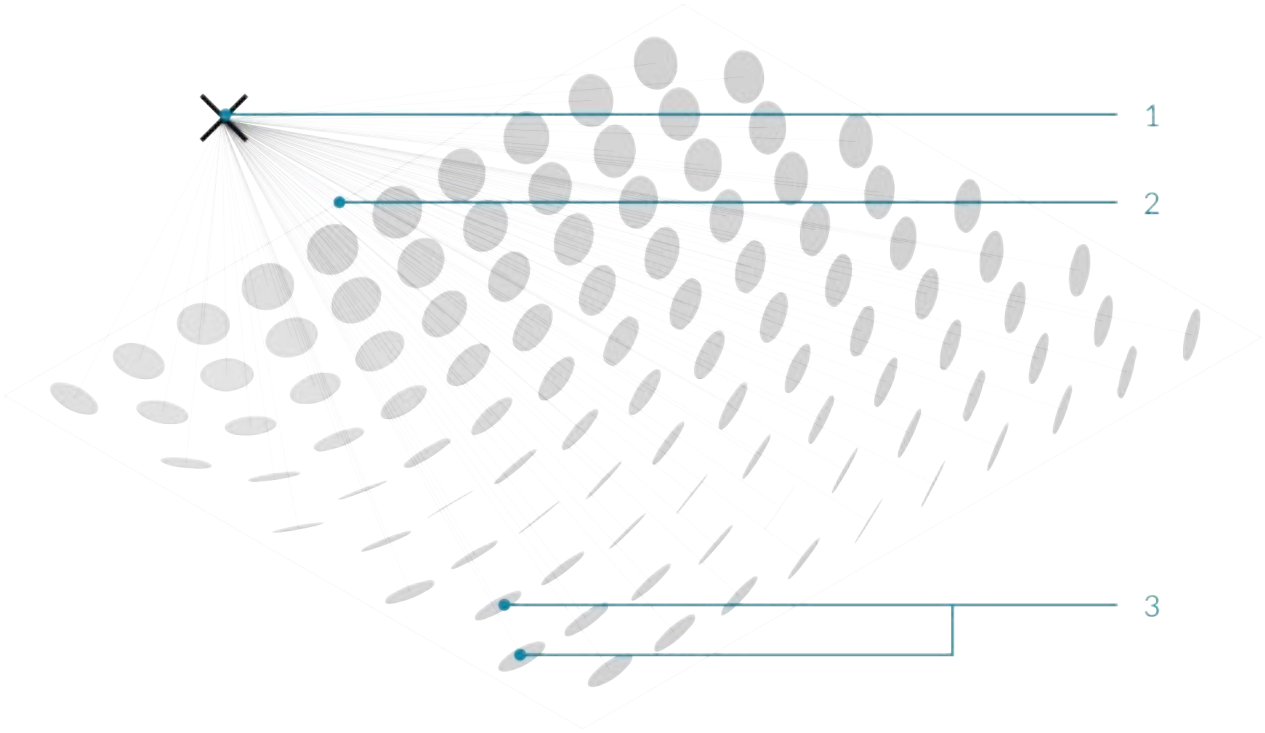
1.3.1.3. PLANES

Planes are “Flat” and extend infinitely in two directions, defining a local coordinate system. Planes are not genuine objects in Rhino, they are used to define a coordinate system in 3D world space. In fact, it’s best to think of planes as vectors, they are merely mathematical constructs.


y p
x p

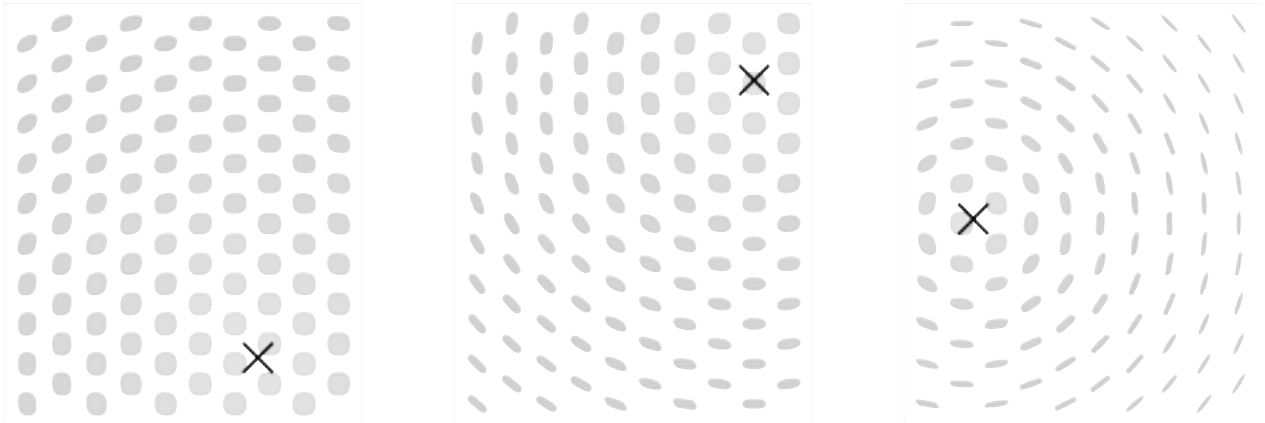
1.3.2. Working with Attractors

Attractors are points that act like virtual magnets - either attracting or repelling other objects. In Grasshopper, any geometry referenced from Rhino or created withinGrasshopper can be used as an attractor. Attractors can influence any number of parameters of surrounding objects including scale, rotation, color, and position. These parameters are changed based on their relationship to the attractor geometry.



1. Attractor point
2. Vectors
3. Circles orient towards attractor based on their normals

In the image above, vectors are drawn between an attractor point and the center point of each circle. These vectors are used to define the orientation of the circles so they are always facing the attractor point. This same attractor could be used to change other parameters of the circles. For example, circles that are closest to the attractor could be scaled larger by using the length of each vector to scale the radius of each circle.

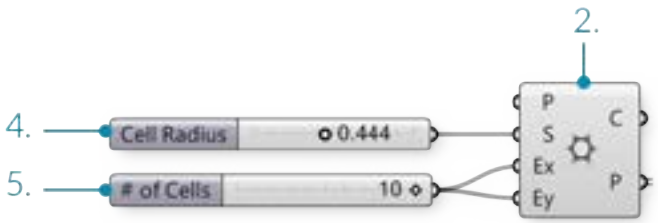


1.3.2.1. ATTRACTOR DEFINITION

Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

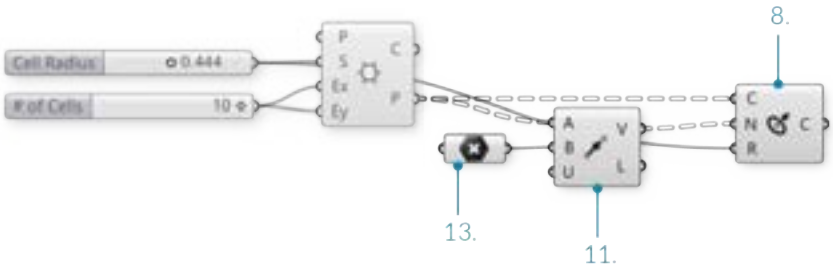
In this example, we will use an attractor point to orient a grid of circles, based on the vectors between the center points of the circles and the attractor point. Each circle will orient such that it is normal to (facing) the attractor point.

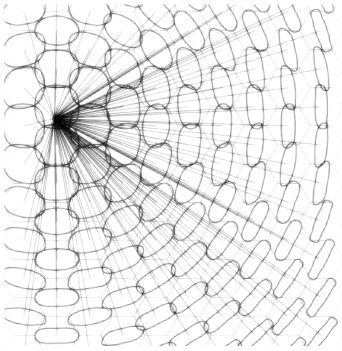
01.	Type Ctrl+N in Grasshopper to start a new definition	
02.	**Vector/Grid/Hexagonal** - Drag and drop the **Hexagonal Grid** component onto the canvas	
03.	**Params/Input/Slider** - Drag and drop two **Numeric Sliders** on the canvas	
04.	Double-click on the first **Numeric Sliders** and set the following: Name: Cell Radius Rounding: Floating Point Lower Limit: 0.000 Upper Limit: 1.000 Value: 0.500	
05.	Double-click on the second **Numeric Sliders** and set the following: Name: # of Cells Rounding: Integers Lower Limit: 0 Upper Limit: 10 Value: 10	
06.	Connect the **Number Slider** (Cell Radius) to the Size (S) input of the Hexagon Grid component	
07.	Connect the **Number Slider** (# of Cells) to the Extent X (Ex) input and the Extent Y (Ey) input of the Hexagon Grid component	



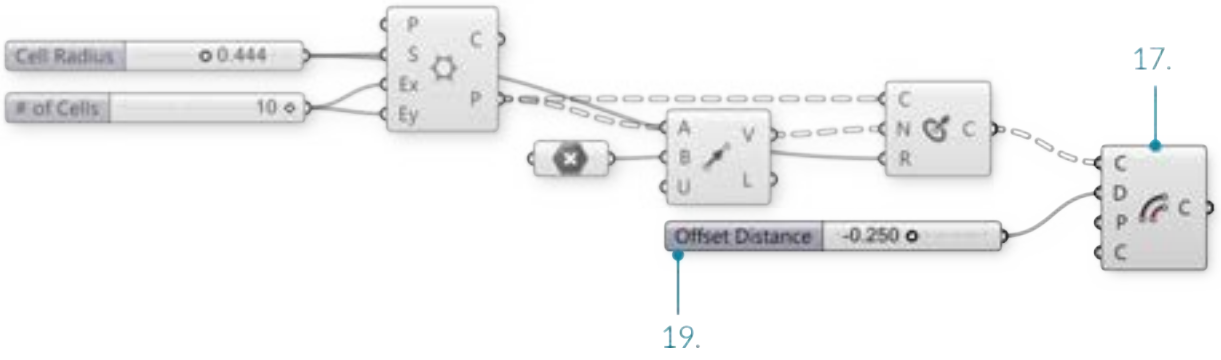
08.	**Curve/Primitive/Circle CNR** - Drag and drop a **Circle CNR** component onto the canvas	
09.	Connect the Points (P) output of the **Hexagon Grid** to the Center (C) input of the **Circle CNR** component	
10.	Connect the **Number Slider** (Cell Radius) to the Radius (R) input of the **Circle CNR** component.	

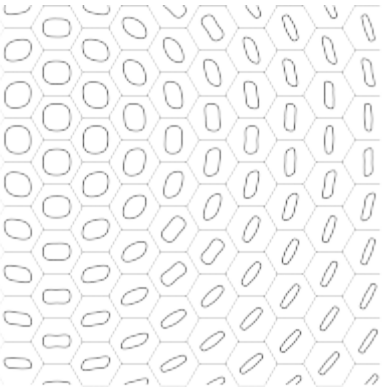
11.	**Vector/Vector/Vector 2Pt** - Drag and Drop the **Vector 2Pt** component onto the canvas	
12.	Connect the Points output (P) of the **Hexagonal Grid** component to the Base Point (A) input of the **Vector 2Pt** component.	
13.	**Params/Geometry/Point** – Drag and Drop the **Point** component onto the canvas	
14.	Right-Click the **Point** component and select set one point. In the model space select where you would like the attractor point to be	
15.	Connect the **Point** component to the Tip Point (B) input of the **Vector 2Pt** component	
16.	Connect the Vector (V) output of the **Vector 2Pt** to the Normal (N) input of the **Circle CNR** component.	



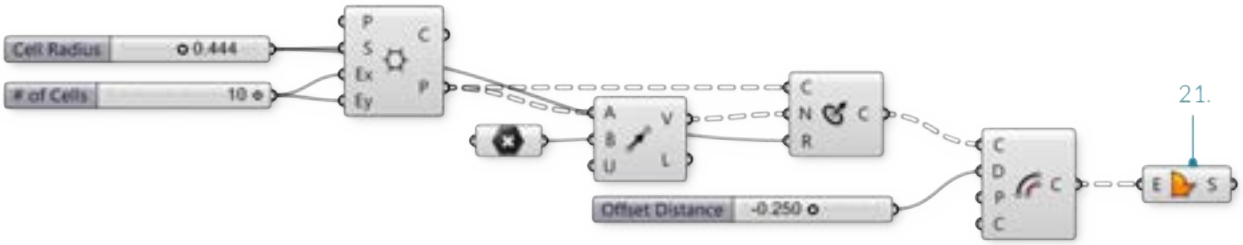


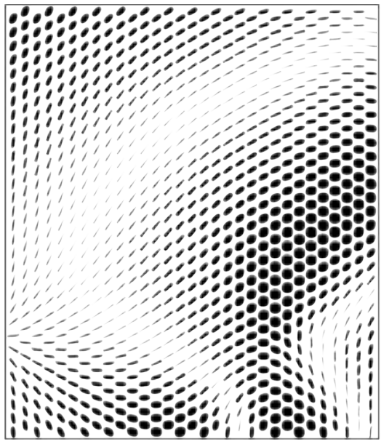
17.	**Curve/Util/Offset** – Drag and Drop the **Offset Component** onto the canvas.	
18.	**Params/Input/Slider** - Drag and drop a **Numeric Slider** on the canvas	
19.	Double-click on the Number Slider and set the following: Name: Offset Distance Rounding: Floating Point Lower Limit: - 0.500 Upper Limit: 0.500 Value: -0.250	
20.	Connect the **Number Slider** (Offset Distance) to the Distance (D) input of the **Offset** component	

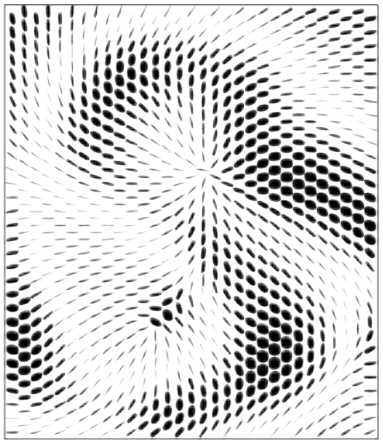


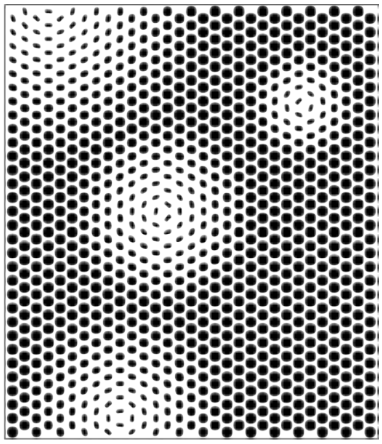


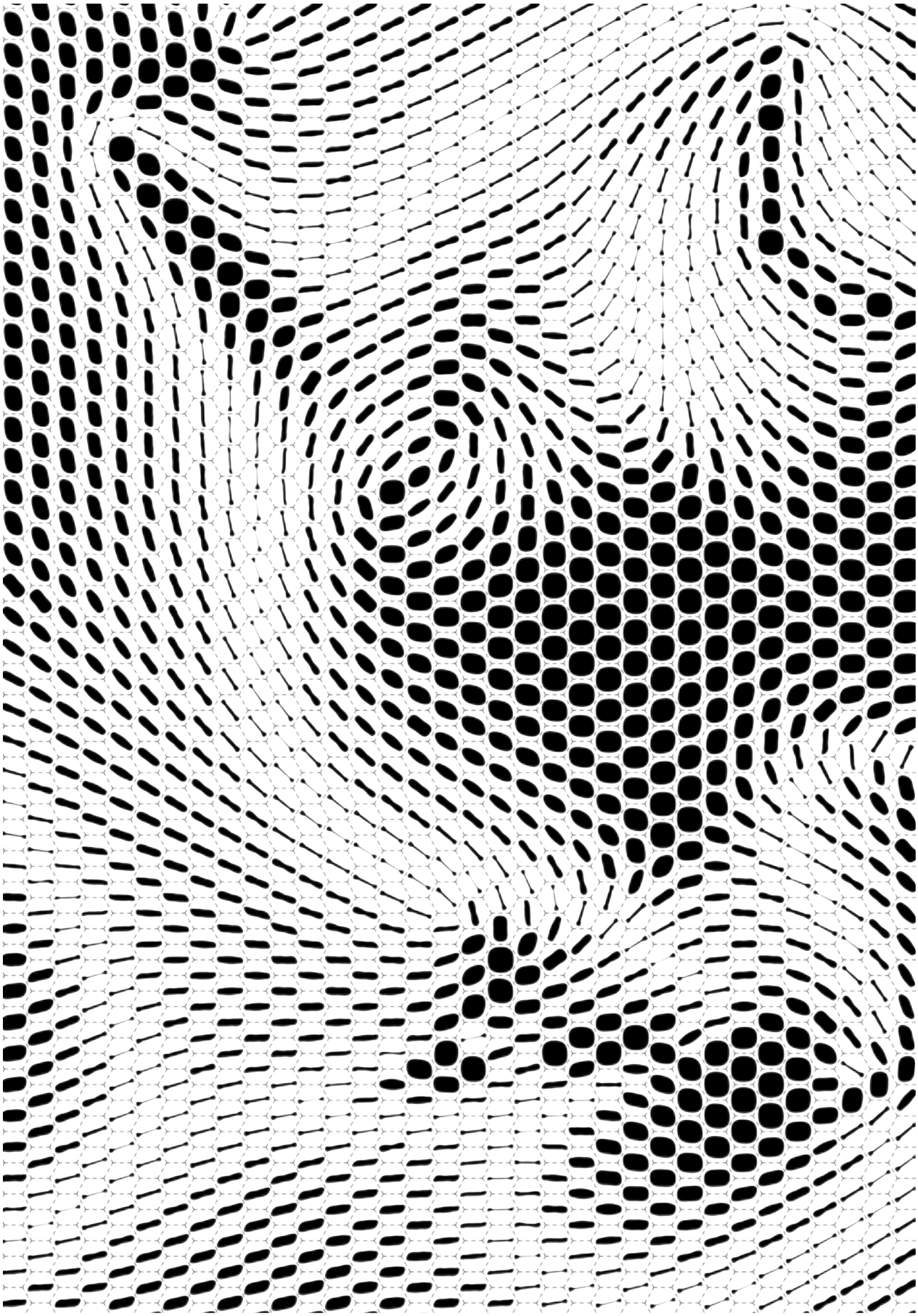
21.	**Surface/Freeform/Boundary Surfaces** – Drag and drop **Boundary Surfaces** on to the canvas	
22.	Connect the Curves (C) output of the **Offset** component to the Edges (E) input of the **Boundary Surfaces**	











1.3.3. Mathematics, Expressions & Conditionals

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Knowing how to work with numeric information is an essential skill to master as you learn to use Grasshopper. Grasshopper contains many components to perform mathematical operations, evaluate conditions and manipulate sets of numbers.

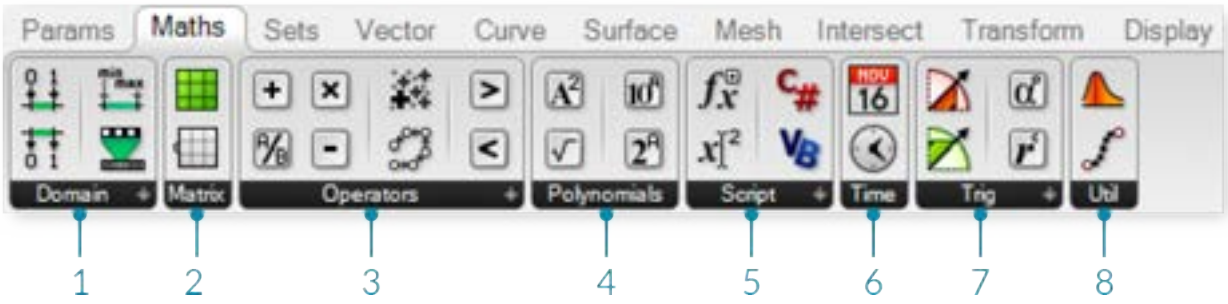
In mathematics, numbers are organized by sets and there are two that you are probably familiar with:

Integer Numbers: [..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...]
Real Numbers: [8, ..., -4.8, -3.6, -2.4, -1.2, 0.0, 1.234, e, 3.0, 4.0, ..., 8]

While there are other types of number sets, these two interest us the most because Grasshopper uses these extensively. Although there are limitations in representing these sets exactly in a digital environment, we can approximate them to a high level of precision. In addition, it should be understood that the distinction between Integral types (integers) and Floating types (real numbers) corresponds to the distinction between discrete and continuous domains. In this chapter, we're going to explore different methods for working with and evaluating various sets of numbers.

1.3.3.1. THE MATH TAB

Most of the components that deal with mathematical operations and functions can be found under the following sub-categories of the Math tab:



1. Domains are used to define a range of values (formerly known as intervals) between two numbers. The components under the Domain tab allow you to create or decompose different domain types.
2. In mathematics, a matrix is an array of numbers organized in rows and columns. This subcategory contains a series of utility tools to construct and modify matrices.
3. Operators are used to perform mathematical operations such as Addition, Subtraction, Multiplication, etc. Conditional operators allow you to determine whether a set of numbers are larger than, less than, or similar to another set of numbers.
4. Polynomials are one of the most important concepts in algebra and throughout mathematics and science. You can use the components found in this subcategory to compute factorials, logarithms, or to raise a number to the nth power.
5. The script subcategory contains single and multi-variable expressions as well as the VB.NET and C# scripting components.
6. These components allow you to solve trigonometric functions such as Sine,Cosine, Tangent, etc.
7. The time subcategory has a number of components which allow you to construct instances of dates and times.
8. The utility subcategory is a 'grab bag' of useful components that canbe used in various mathematical equations. Check here if you're trying find the maximum or minimum values between two lists of

> numbers; or average a group of numbers.

1.3.3.2. OPERATORS

As was previously mentioned, Operators are a set of components that use algebraic functions with two numeric input values, which result in one output value.

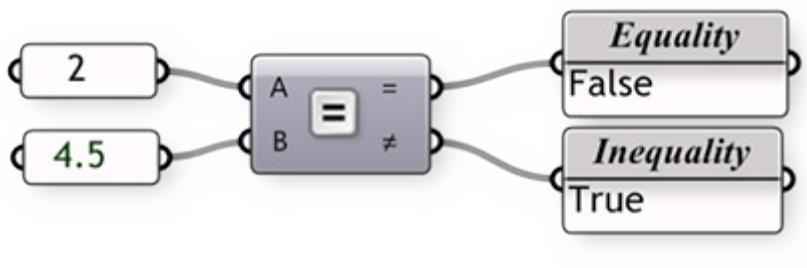
Most of the time, you will use the Math Operators to perform arithmetical actions on a set of numbers. However, these operators can also be used on various data types, including points and vectors.

1.3.3.3. CONDITIONAL OPERATORS

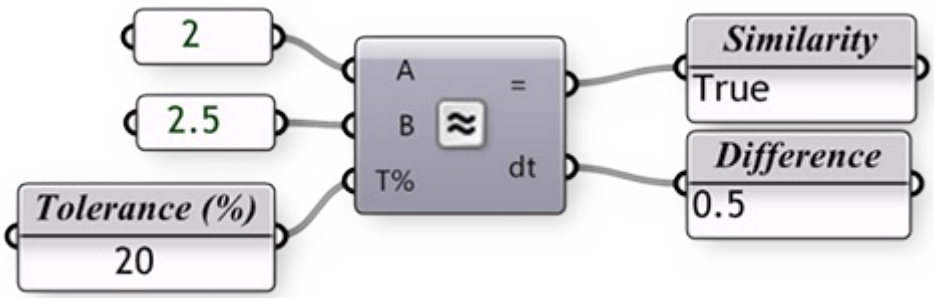
Almost every programming language has a method for evaluating conditional statements. In most cases the programmer creates a piece of code to ask a simple question of “what if.” What if the area of a floor outline exceeds the programmatic requirements? Or, what if the curvature of my roof exceeds a realistic amount? These are important questions that represent a higher level of abstract thought. Computer programs have the ability to analyze “what if” questions and take actions depending on the answer to that question. Let’s take a look at a very simple conditional statement that a program might interpret: If the object is a curve, delete it. The piece of code first looks at an object and determines a single boolean value for whether or not it is a curve. There is no middle ground. The

boolean value is True if the object is a curve, or False if the object is not a curve. The second part of the statement performs an action dependent on the outcome of the conditional statement; in this case, if the object is a curve then delete it. This conditional statement is called an If statement. There are four conditional operators (found under the Math/ Operators subcategory) that evaluate a condition and return a boolean value.

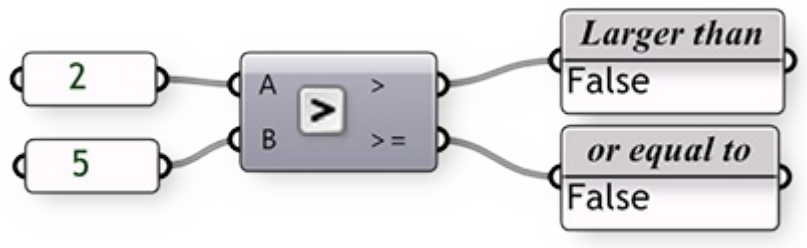
The Equality component takes two lists and compares the first item of List Aand compares it to the first item of List B. If the two values are the same, then a True boolean value is created; conversely if the two values are not equal, then a False boolean value is created. The component cycles through the lists according to the set data matching algorithm (default is set to Longest List). There are two outputs for this component. The first returns a list of boolean values that shows which of the values in the list were equal to one another. The second output returns a list that shows which values were not equal to one another - or a list that is inverted from the first output.



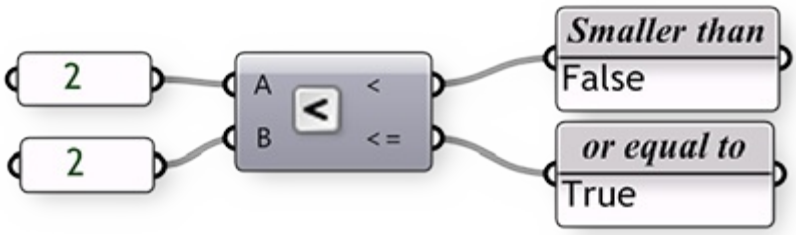
The Similarity component evaluates two lists of data and tests for similarity between two numbers. It is almost identical to the way the Equality component compares the two lists, with one exception: it has a percentage input that defines the ratio of list Athat list B is allowed to deviate before inequality is assumed. The Similarity component also has an output that determines the absolute value distance between the two input lists.



The Larger Than component will take two lists of data and determine if the first item of List Ais greater than the first item of List B. The two outputs allow you to determine if you would like to evaluate the two lists according to a greater than (>) or greater than and equal to (>=) condition.



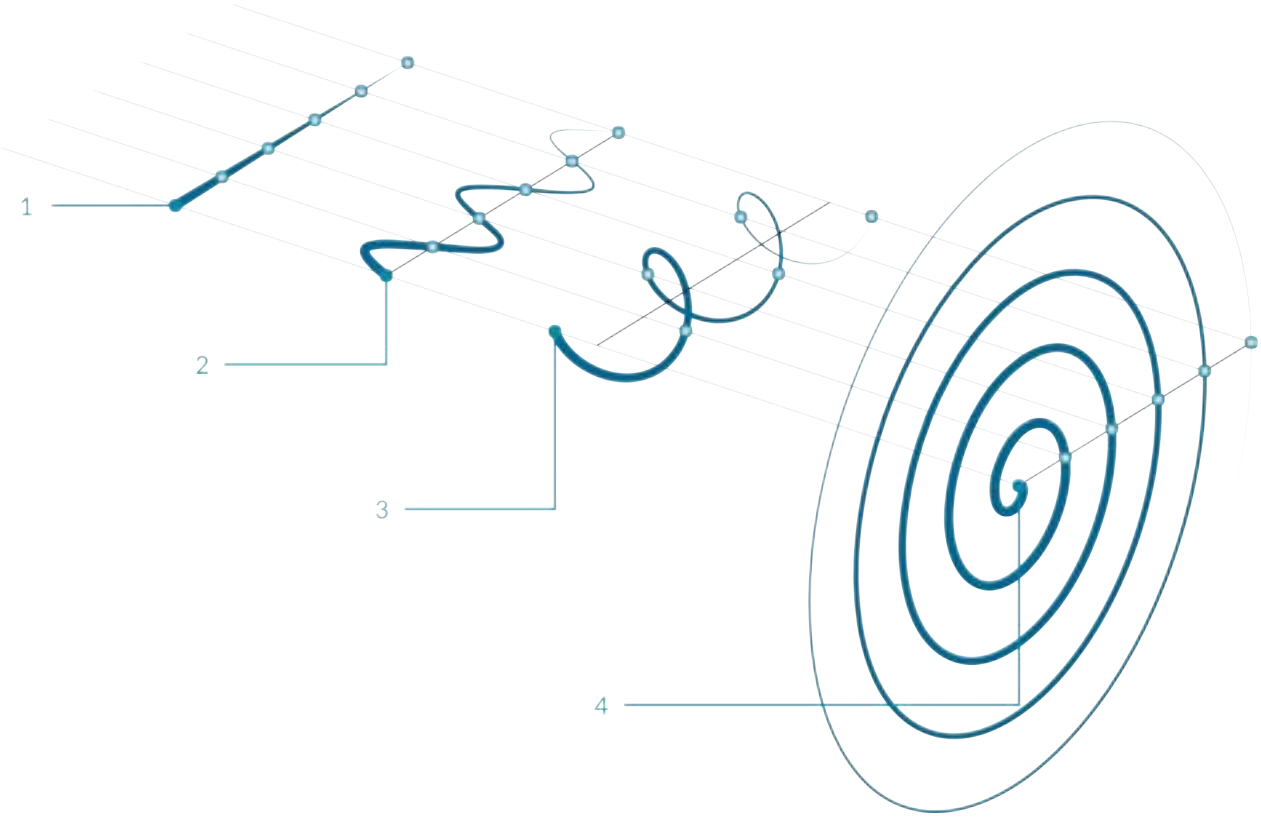
The Smaller Than component performs the opposite action of the Larger Than component. The Smaller Than component determines if list Ais less than list B and returns a list of boolean values. Similarly, the two outputs let you determine if you would like to evaluate each list according to a less than (<) or less than and equal to (<=) condition.



1.3.3.4. TRIGONOMETRY COMPONENTS

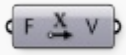
> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

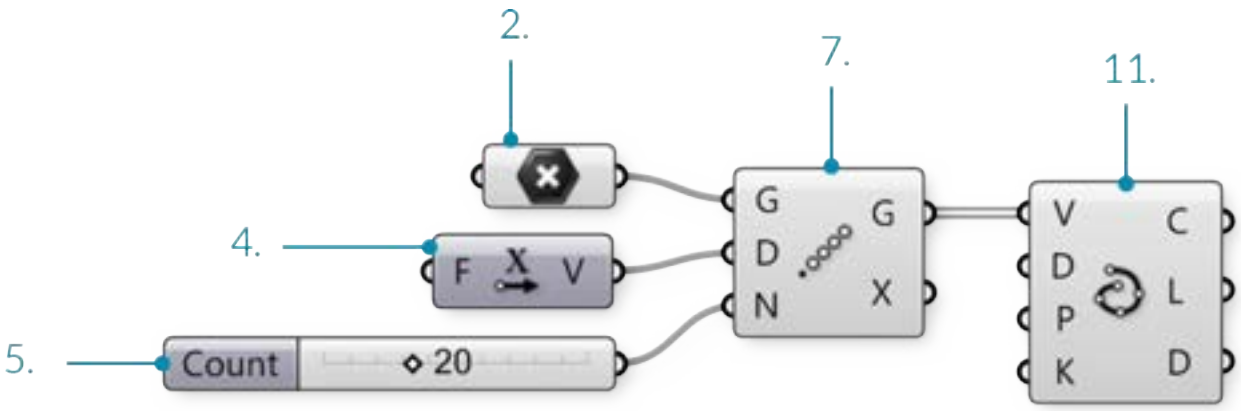
We have already shown that we can use an Expression (or Evaluate) component to evaluate conditional statements as well as compute algebraic equations. However, there other ways to calculate simple expressions using a few of the built in Trigonometry functions. We can use these functions to define periodic phenomena like sinusoidal wave forms such as ocean waves, sound waves, and light waves.



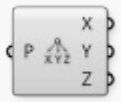
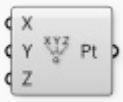
1. Line
   `y(t) = 0`
2. Sine Curve
   `y(t) = sin(t)`
3. Helix
   `x(t) = cos(t)`
   `y(t) = sin(t)`
   `z(t) = b(t)`
4. Spiral
   `x(t) = t*cos(t)`
   `y(t) = t*cos(t)`

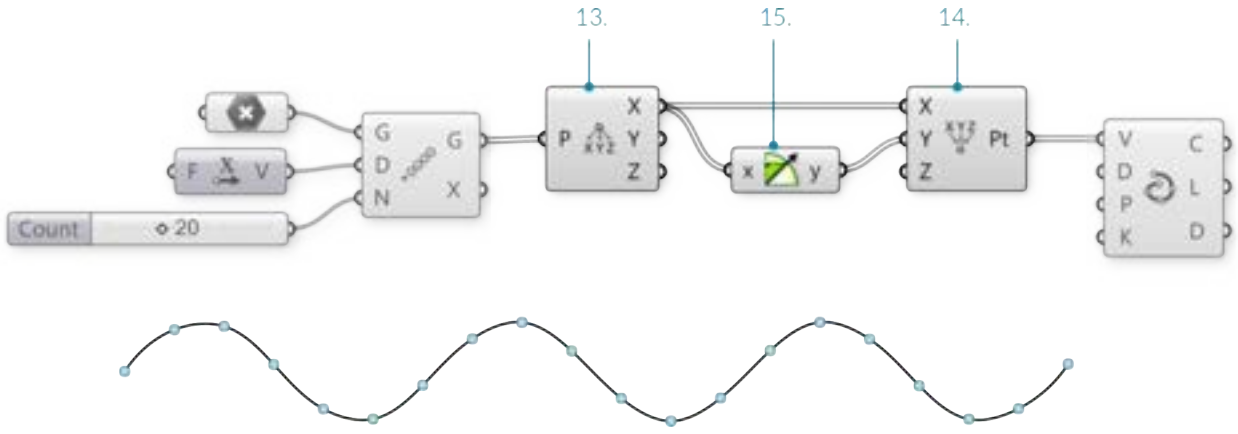
In this example, we will use Grasshopper to construct various trigonometric curves using trigonometry function components found in the Math tab:

01.	Type Ctrl+N (in Grasshopper) to start a new definition	
02.	**Params/Geometry/Point** – Drag and drop a **Point** parameter onto the canvas	
03.	Right click the **Point** parameter and click Set One Point – select a point in the Rhino viewport	
04.	**Vector/Vector/Unit X** – Drag and drop the **Unit X** component to the canvas	
05.	**Params/Input/Number Slider** – Drag and drop the **Number Slider** component onto the canvas	
06.	Double-click on the **Number Slider** and set the following: Rounding: Integer Lower Limit: 10 Upper Limit: 40 Value: 20	
07.	**Transform/Array/Linear Array** – Drag and drop the **Linear Array** component onto the canvas	
08.	Connect the output of the **Point** parameter to the Geometry (G) input of the **Linear Array** component	
09.	Connect the Unit Vector (V) output of the **Unit X** component to the Direction (D) input of the **Linear Array** component > You should see a line of 20 points along the x axis in Rhino. Adjust the slider to change the number of points in the array.	
10.	Connect the **Number Slider** output to the Count (N) input of the **Linear Array** Component	
11.	**Curve/Spline/Interpolate** – Drag and drop the **Interpolate Curve** component to the canvas	
12.	Connect the Geometry (G) output of the **Linear Array** component to the Vertices (V) input of the **Interpolate Curve** component	



> We have just created a line by connecting an array of points with a curve. Let's try using some of Grasshopper's Trigonometry components to alter this curve:

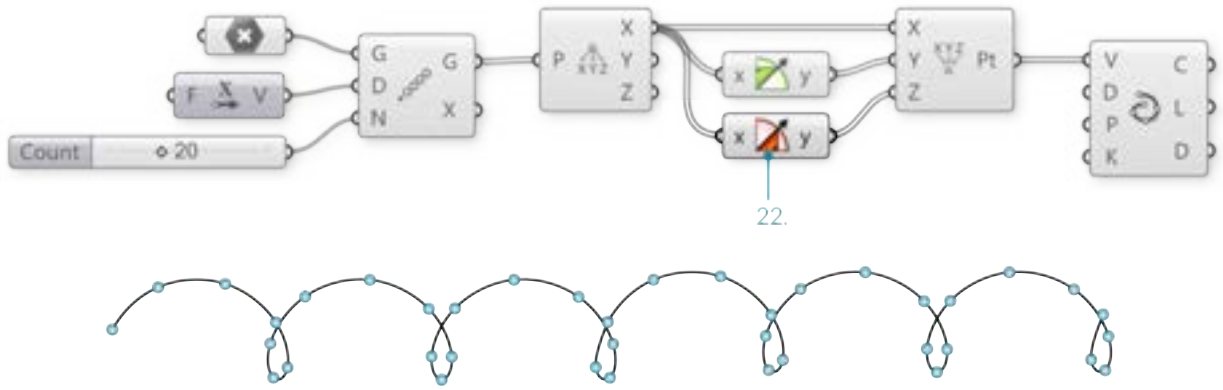
13.	**Vector/Point/Deconstruct** – Drag and drop a **Deconstruct** component onto the canvas	
14.	**Vector/Point/Construct Point** - Drag and drop a **Construct Point** component onto the canvas	
15.	**Maths/Trig/Sine** - Drag and drop a **Sine** component onto the canvas	
16.	Disconnect the wire from the Vertices (V) input of the **Interpolate Curve** component. > You can disconnect wires by holding down control and dragging, or by right-clicking the input and selecting Disconnect	
17.	Connect the Geometry (G) output of the **Linear Array** component to the Point (P) input of the **Deconstruct** component	
18.	Connect the Point X (X) output of the **Deconstruct** component to the X coordinate (X) input of the **Construct Point** Component	
19.	Connect a second wire from the Point X (X) output of the **Deconstruct** Component to the Value (x) input of the **Sine** component	
20.	Connect the Result (y) output of the **Sine** component to the Y coordinate (Y) input of the **Construct Point** component > We have now reconstructed our points with the same X values, modifying the Y values with a sine curve.	
21.	Connect the Point (Pt) output of the **Construct Point** component to the Vertices (V) input of the **Interpolate** component	



> You should now see a sine wave curve along the X axis in Rhino

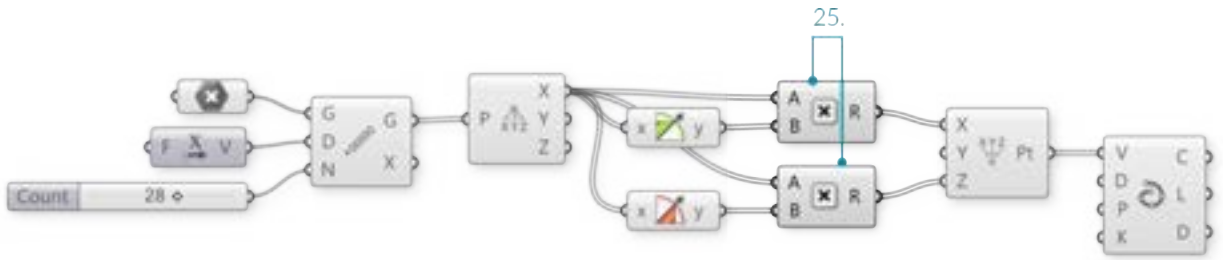
22.	**Maths/Trig/Cosine** – Drag and drop a **Cosine** component to the canvas	
23.	Connect a third wire from the Point X (X) output of the **Deconstruct** Component to the Value (x) input of the **Cosine** component	

24.	Connect the Result (y) output of the **Cosine** component to the Z coordinate (Z) input of the **Construct Point** component	

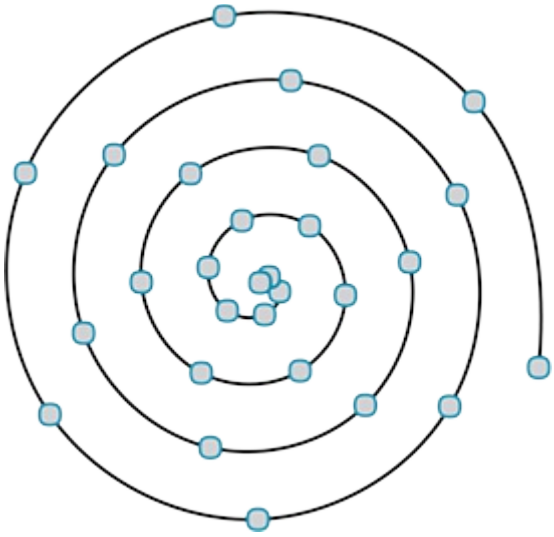


> We have now created a 3D helix

25.	**Maths/Operators/Multiplication** – Drag and drop two **Multiplication** components onto the canvas	
26.	Connect wires from the Point X (X) output of the **Deconstruct** component to the (A) input of each **Multiplication** component	
27.	Connect the Result (y) output of the **Sine** component to the (B) input of the first **Multiplication** component	
28.	Connect the Result (y) output of the **Cosine** component to the (B) input of the second **Multiplication** component	
29.	Disconnect the wire from the Y Coordinate (Y) input of the **Construct Point** component	
30.	Connect the Result (R) output of the first **Multiplication** component to the X Coordinate (X) input of the **Construct Point** component	
31.	Connect the Result (R) output of the second **Multiplication** component to the Z Coordinate (Z) input of the **Construct Point** component	



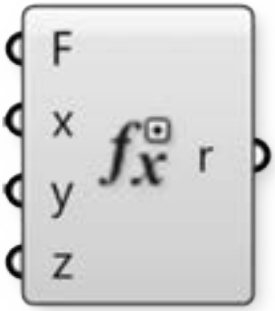
> You should now see a spiral curve



1.3.3.5. EXPRESSIONS

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

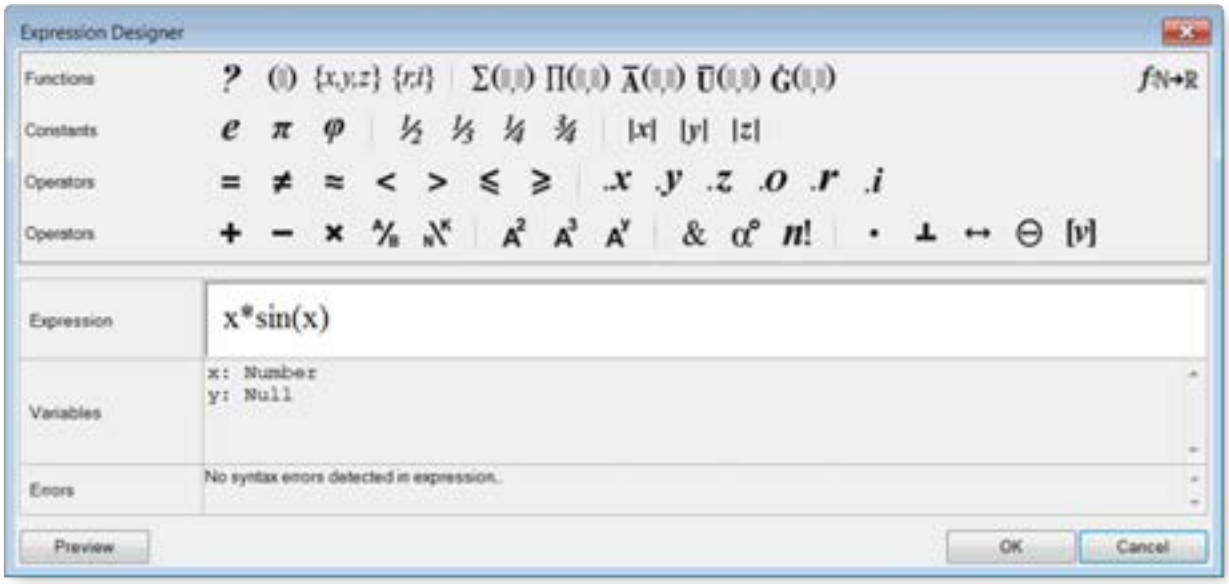
The Expression component (and its brother the Evaluate component) are very flexible tools; that is to say that they can be used for a variety of different applications. We can use an Expression (or Evaluate component) to solve mathematical algorithms and return numeric data as the output.



In the following example, we will look at mathematical spirals found in nature and how we can use a few Functions components to create similar patterns in Grasshopper. We will build on our trigonometric curves definition as a starting point.

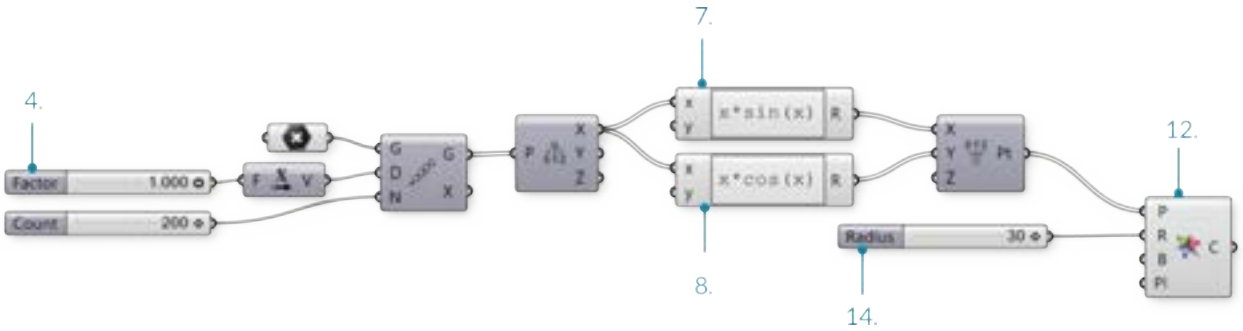
01.	Open your Trigonometric curves Grasshopper definition from the previous example	
02.	Delete the **Sine**, **Cosine**, **Multiplication**, and **Interpolate** components	
03.	**Params/Input/Number Slider** – Drag and drop a Number Slider onto the canvas	
04.	Double-click on the **Number Slider** and set the following: Rounding: Float Lower Limit: 0.000 Upper Limit: 1.000 Value: 1.000	
05.	Connect the **Number Slider** to the Factor (F) input of the Unit X component. This slider allows you to adjust the distance between the points in the array.	
06.	**Maths/Script/Expression** – Drag two **Expression** components onto the canvas	

07.	Double-click the first **Expression** component to open the Expression Editor and change the expression to: `x*sin(x)`	
08.	Double-click the second **Expression** component to open the Expression Editor and change the expression to: `x*cos(x)`	

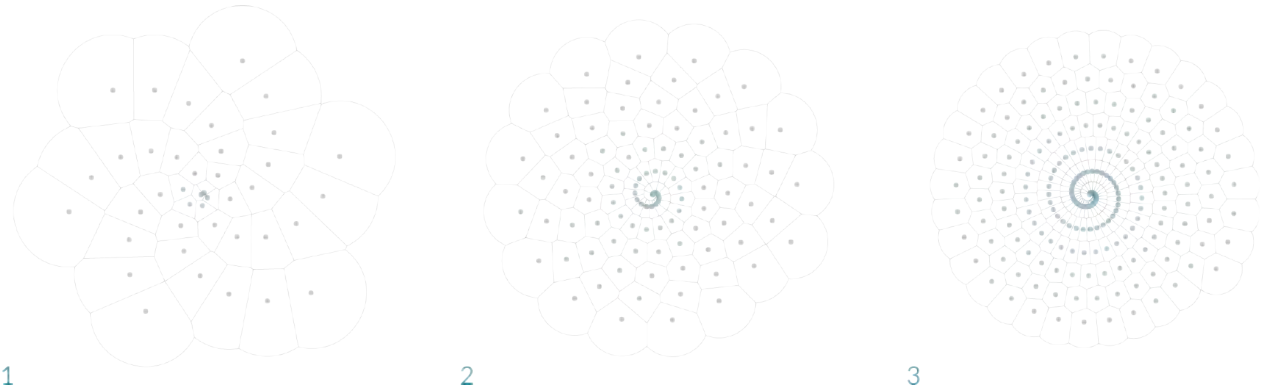


> Double click the Expression component to open the Grasshopper Expression Editor

09.	Connect two wires from the Point X (X) output of the **Deconstruct** component to the Variable x (x) input of each **Expression** component	
10.	Connect the Result (R) output of the first **Expression** component to the X coordinate (X) input of the **Construct Point** component	
11.	Connect the Result (R) output of the second **Expression** component to the Y coordinate (Y) input of the **Construct Point** component > We have replaced the Trigonometry functions and multiplication operators with the expression components for a more efficient definition.	
12.	**Mesh/Triangulation/Voronoi** – Drag and drop the **Voronoi** component onto the canvas	
13.	**Params/Input/Number Slider** – Drag and drop a **Number Slider** onto the canvas	
14.	Double-click on the **Number Slider** and set the following: Rounding: Integer Lower Limit: 1 Upper Limit: 30 Value: 30	
15.	Connect the **Number Slider** to the Radius (R) input of the **Voronoi** component	
16.	Connect the Point (Pt) output of the **Construct Point** component to the Points (P) input of the **Voronoi** component	

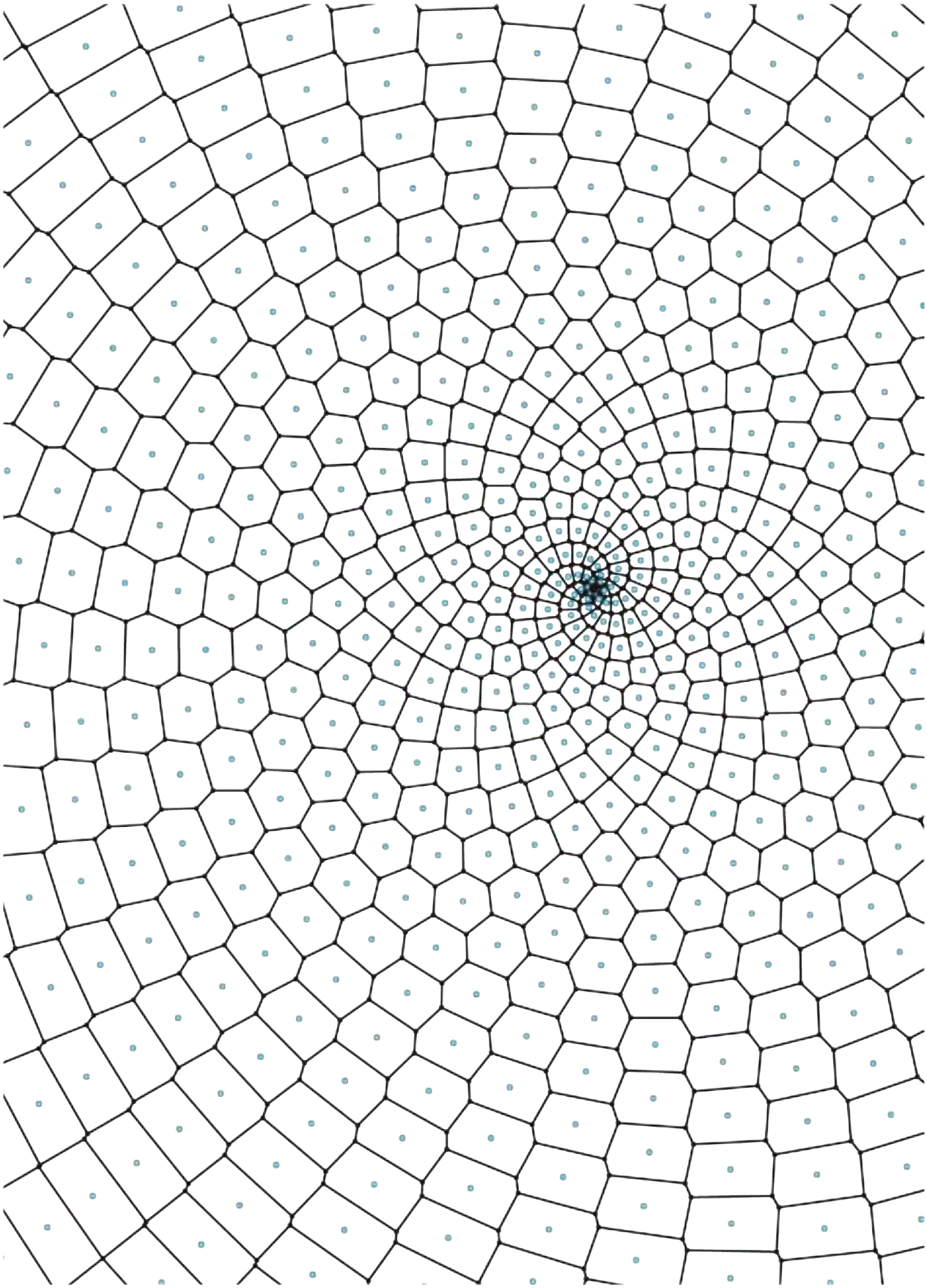


You can create different Voronoi patterns by manipulating the Factor, Count, and Radius sliders. Below are three examples:



1 2 3

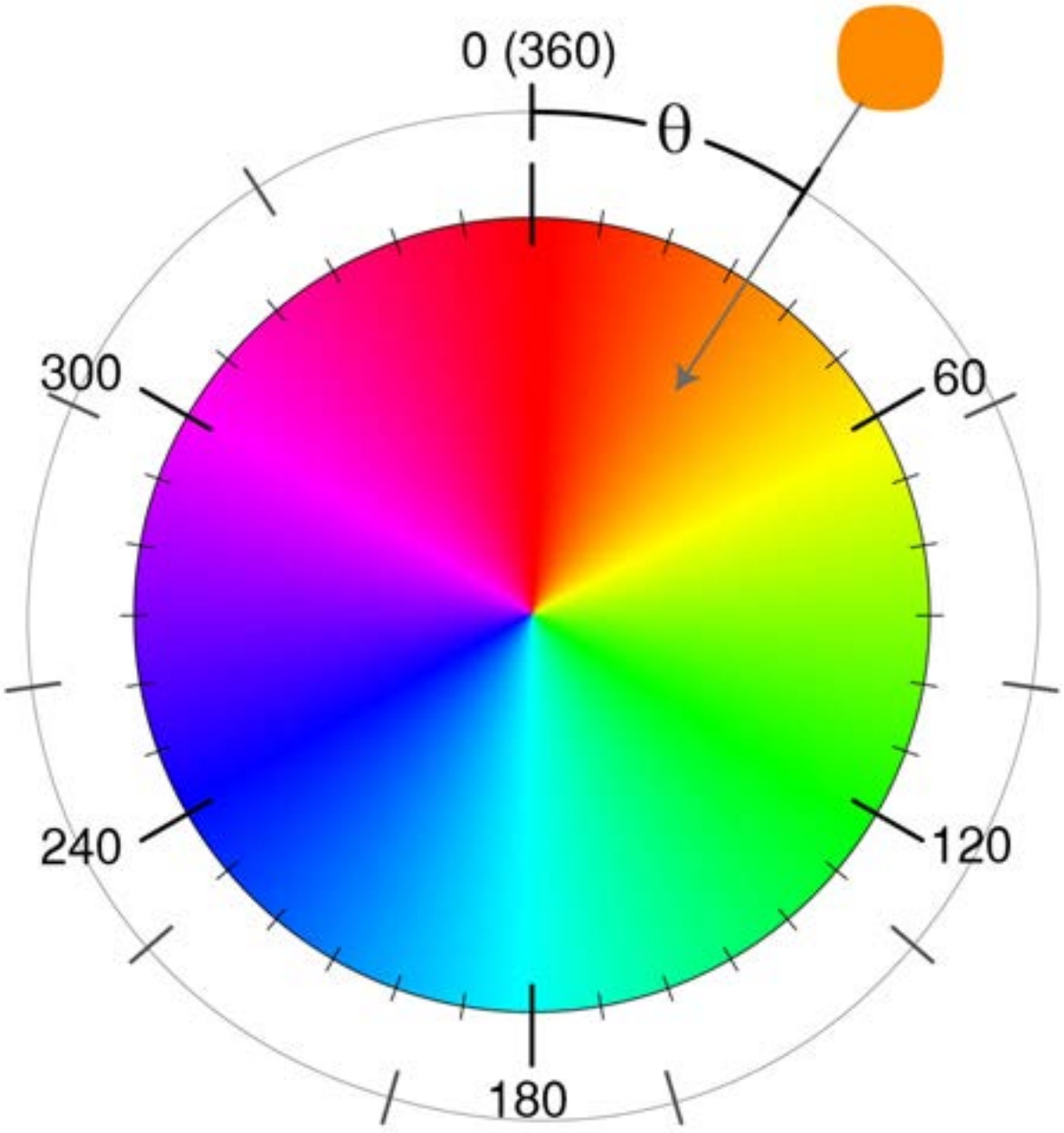
1. Factor = 1.000, Radius = 15
2. Factor = 0.400, Radius = 10
3. Factor = 0.200, Radius = 7



1.3.4. Domains & Color

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

The color wheel is a model for organizing colors based on their hue. In Grasshopper, colors can be defined by their hue value in a range of 0.0 to 1.0. Domains are used to define a range of all possible values between a set of numbers between a lower limit(A) and an upper limit (B).

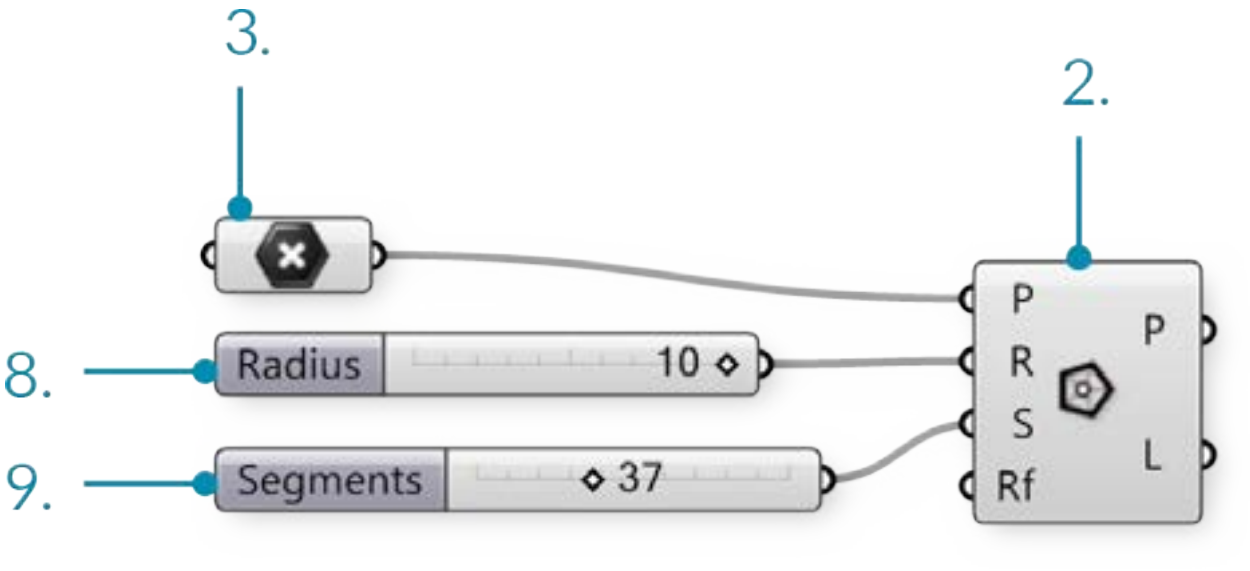


> In the color wheel, hue corresponds to the angle. Grasshopper has taken this 0-360 domainand remapped it between zero and one.

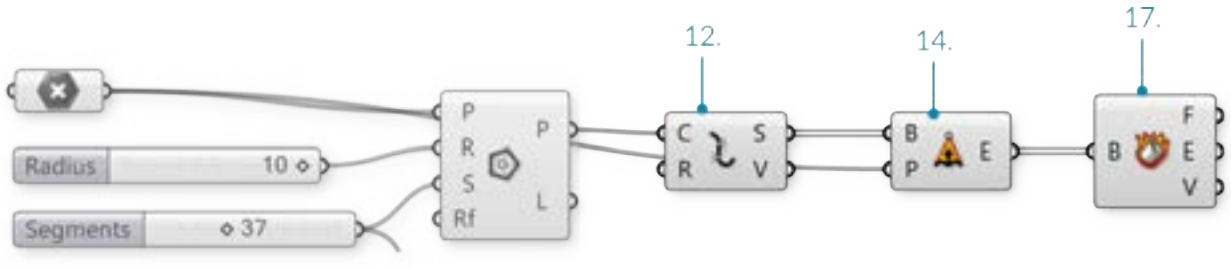
By dividing the Hue domain (0.0 to 1.0) by the number of segments desired, we can assign a hue value to each segment to create a color wheel.

In this example, we will use Grasshopper's domain and color components to create a color wheel with a variable amount of segments.

01.	Type Ctrl+N (in Grasshopper) to start a new definition	
02.	**Curve/Primitive/Polygon** – Drag and drop a **Polygon** component onto the canvas	
03.	**Params/Geometry/Point** – Drag and drop a **Point** Parameter onto the canvas	
04.	Right-Click on the **Point** Component and select set one point	
05.	Set a point in the model space.	
06.	Connect the **Point** Parameter (Base Point) to the Plane (P) input of the **Polygon** component	
07.	**Params/Input/Number Sliders** – Drag and drop two **Number Sliders** onto the canvas	
08.	Double-click on the first **Number Sliders** and set the following: Rounding: Integers Lower Limit: 0 Upper Limit: 10 Value: 10	
09.	Double-click on the second **Number Sliders** and set the following: Rounding: Integers Lower Limit: 0 Upper Limit: 100 Value: 37	
10.	Connect the **Number Slider** (Radius) to the Radius (R) input of the **Polygon** component When you connect a number slider to a component in will automatically change its name to the name of input that it is connecting to.	
11.	Connect the **Number Slider** (Segments) to the Segments (S) input of the **Polygon** component	

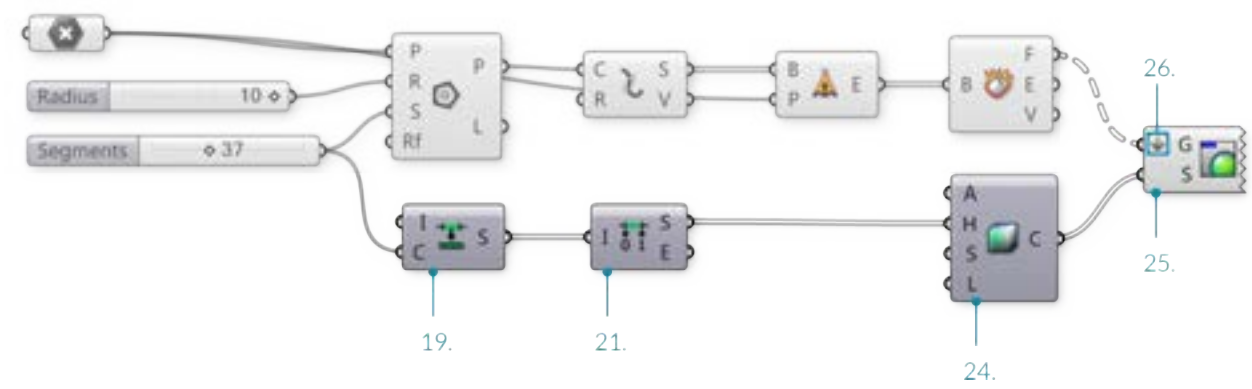


12.	**Curve/Util/Explode** – Drag and drop an **Explode** component onto the canvas.	
13.	Connect the Polygon (P) output of the **Polygon** component to the Curve (C) input of the **Explode** component	
14.	**Surface/Freeform/Extrude Point** – Drag and drop the **Extrude Point** component onto the canvas	
15.	Connect the Segments (S) output of the **Explode** component to the Base (B) input of the **Extrude Point**	
16.	Connect the **Point** Parameter (Base Point) to the Extrusion Tip (P) of the **Extrude Point** component	
17.	**Surface/Analysis/Deconstruct Brep** – Drag and drop the **Deconstruct Brep** component on to the canvas	
18.	Connect the Extrusion (E) output of the **Extrude Point** component to the **Deconstruct Brep** (B) component	

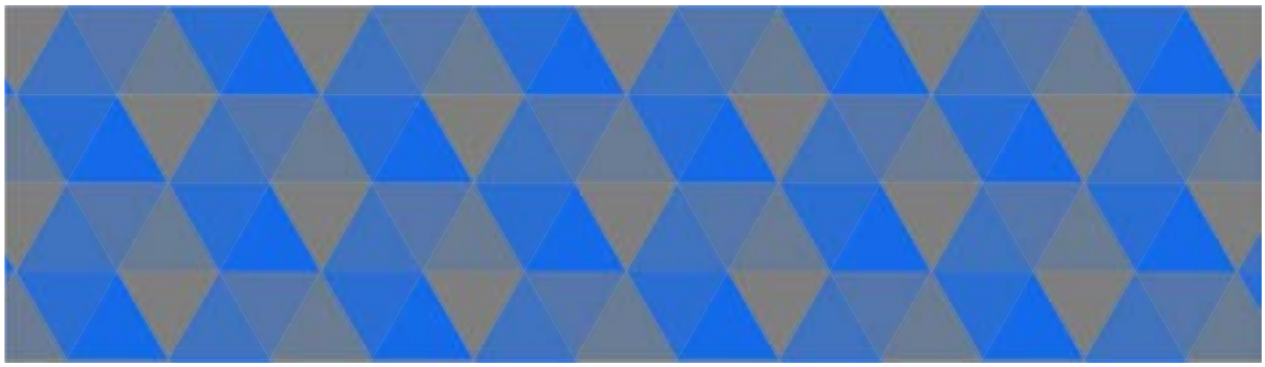


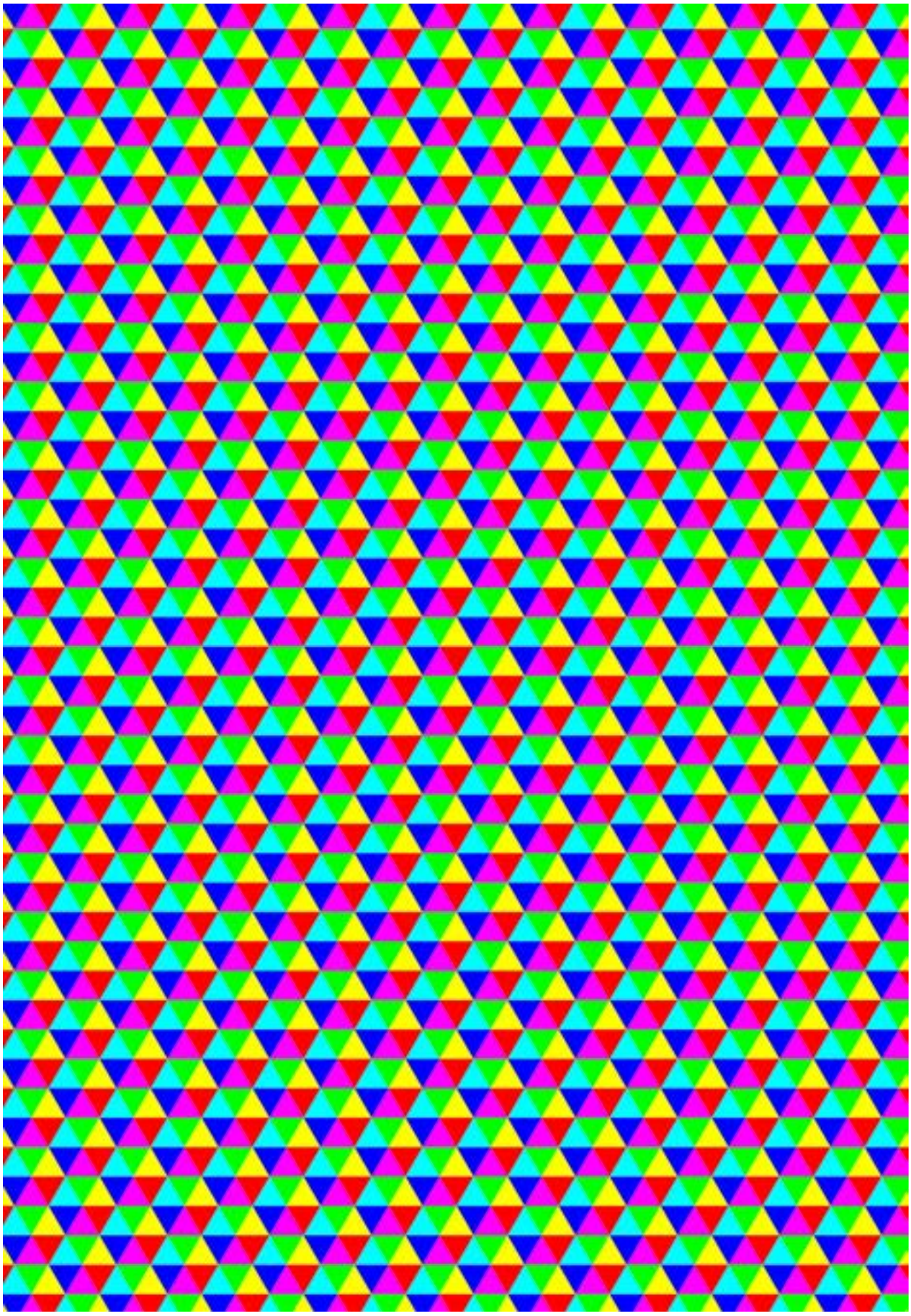
19.	**Maths/Domain/Divide Domain** – Drag and drop the **Divide Domain** component The Base Domain (I) is automatically set between 0.0-1.0 which is what we need for this exercise	
20.	Connect the **Number Slider** (Segments) to the Count (C) input of the **Divide Domain** component	

21.	**Math/Domain/Deconstruct Domain** – Drag and drop the **Deconstruct Domain** component	
22.	Connect the Segments (S) output of the **Divide Domain** component to the Domain (I) input of the **Deconstruct Domain** component	
23.	**Display/Colour/Colour HSL** – Drag and drop the **Colour HSL** component	
24.	Connect the Start (S) output of the **Deconstruct Domain** component to the Hue (H) input of the **Colour HSL** components	
25.	**Display/Preview/Custom Preview** – Drag and drop the **Custom Preview** component	
26.	Right click on the Geometry (G) input of the **Custom Preview** component and select Flatten See 1-4 Designing with Data Trees for details about flattening	
27.	Connect the Faces (F) output of the **Deconstruct Brep** component to the Geometry (G) input of the **Custom Preview** component	
28.	Connect the Colour (C) output of the **Colour HSL** component to the Shade (S) input of the **Custom Preview** component	



For different color effects, try connecting the Deconstruct Domain component to the saturation (S) or Luminance (L) inputs of the Colour HSL component.





1.3.5. Booleans & Logical Operators

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

1.3.5.1. BOOLEANS

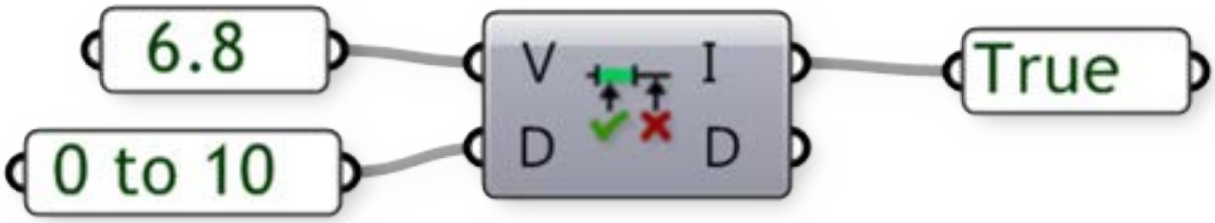
Numeric variables can store a whole range of different numbers. Boolean variables can only store two values referred to as Yes or No, True or False, 1 or 0. Obviously we never use booleans to perform calculations because of their limited range. We use booleans to evaluate conditions.

> Boolean Parameter

In Grasshopper, booleans can be used in several ways. The boolean parameter is a container for one or multiple boolean values, while the Boolean Toggle allows you to quickly change between single true and false values as inputs.

> Boolean Toggle - double click the boolean value to toggle between true and false

Grasshopper also has objects that test conditions and output boolean values. For example, the Includes component allows you to test a numeric value to see if it is included in a domain.



> The Includes component is testing whether the number 6.8 is included in the domain from 0 to 10. It returns a boolean value of True.

1.3.5.2. LOGICAL OPERATORS

Logical operators mostly work on booleans and they are indeed very logical. As you will remember, booleans can only have two values. Boolean mathematics were developed by George Boole (1815-1864) and today they are at the very core of the entire digital industry. Boolean algebra provides us with tools to analyze, compare and describe sets of data. Although Boole originally defined six boolean operators we will only discuss three of them:

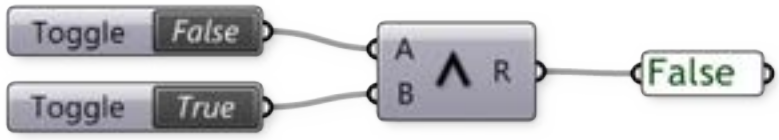
1. Not
2. And
3. Or

The Not operator is a bit of an oddity among operators, because it doesn't require two values. Instead, it simply inverts the one on the right. Imagine we have a script which checks for the existence of a bunch of Block definitions in Rhino. If a block definition does not exist, we want to inform the user and abort the script.

The Grasshopper Not operator (gate)

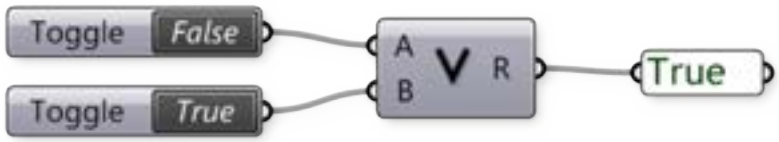
And and Or take two arguments on either side. The And operator requires both of them to be True in order for it to evaluate to True. The Or operator is more than happy with a single True value.

As you can see, the problem with Logical operators is not the theory, it's what happens when you need a lot of them to evaluate something. Stringing them together quickly results in convoluted code; not to mention operator precedence problems.



A	B	Result
True	True	True
True	False	False
False	True	False
False	False	False

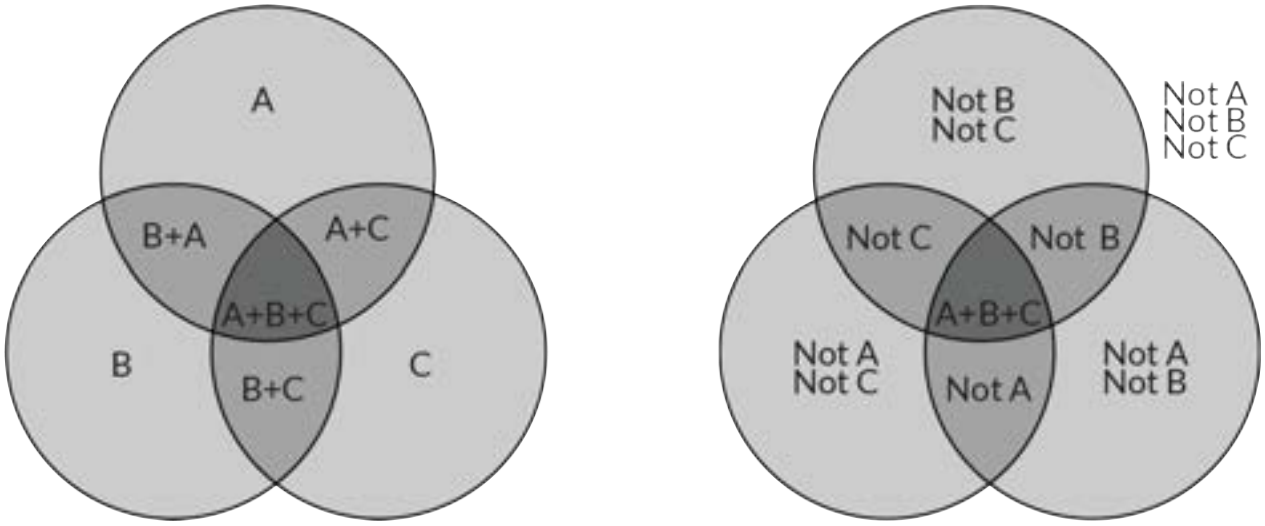
The Grasshopper And operator (gate)



A	B	Result
True	True	True
True	False	True
False	True	True
False	False	False

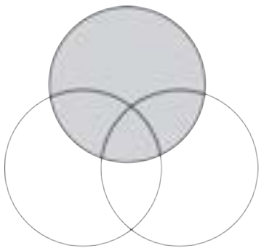
The Grasshopper Or operator (gate)

Agood way to exercise your own boolean logic is to use Venn diagrams. A Venn diagram is a graphical representation of boolean sets, where every region contains a (sub)set of values that share a common property. The most famous one is the three-circle diagram:

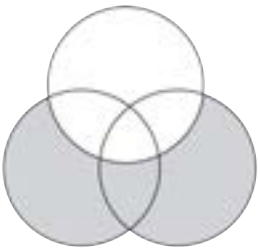


Every circular region contains all values that belong to a set; the top circle for example marks off set {A}. Every value

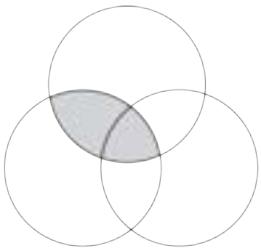
inside that circle evaluates True for {A} and every value not in that circle evaluates False for {A}. By coloring the regions we can mimic boolean evaluation in programming code:



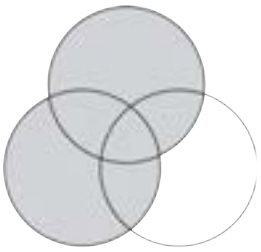
A



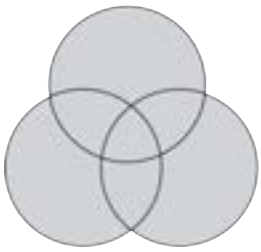
Not A



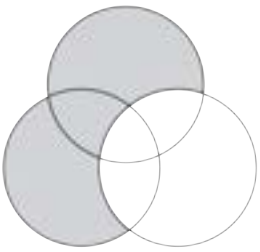
A And B



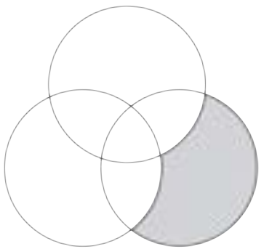
A Or B



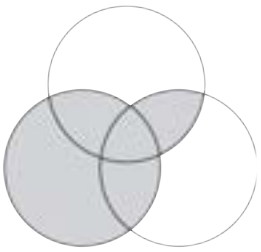
A Or B Or C



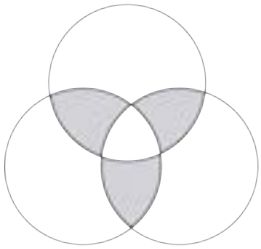
(A Or B) and Not C



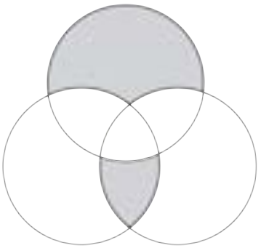
C And Not A and Not B



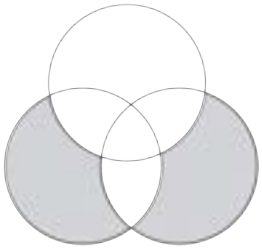
B Or (C And A)



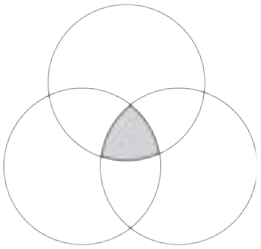
(A And B) Or
(B And C) Or (A And C)



((B And C) and Not A) Or
(A and Not B and Not C)



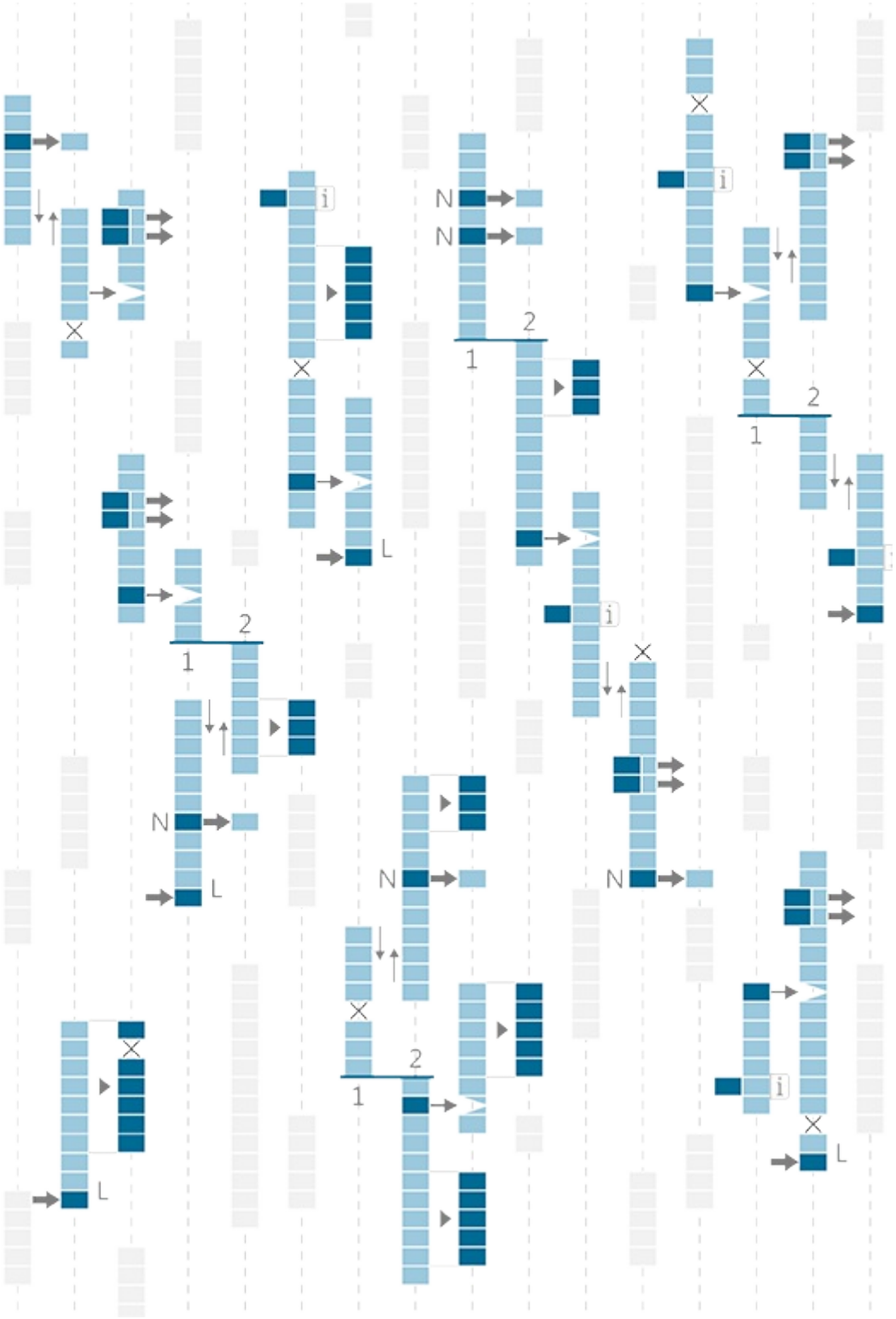
(B and Not C) Or
(C and Not B)



A And B And C

1.4. Designing with Lists

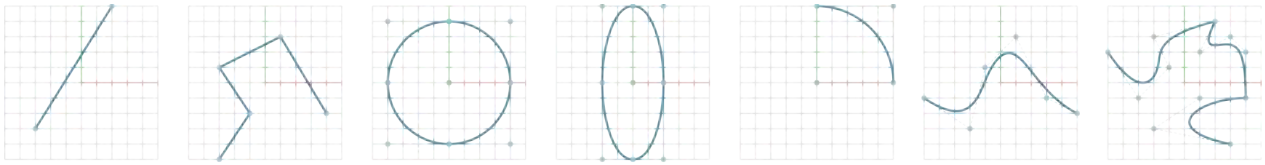
One of the most powerful features of Grasshopper is the ability to quickly build and manipulate lists of data. This chapter will explain how to create, manipulate, and visualize list data.



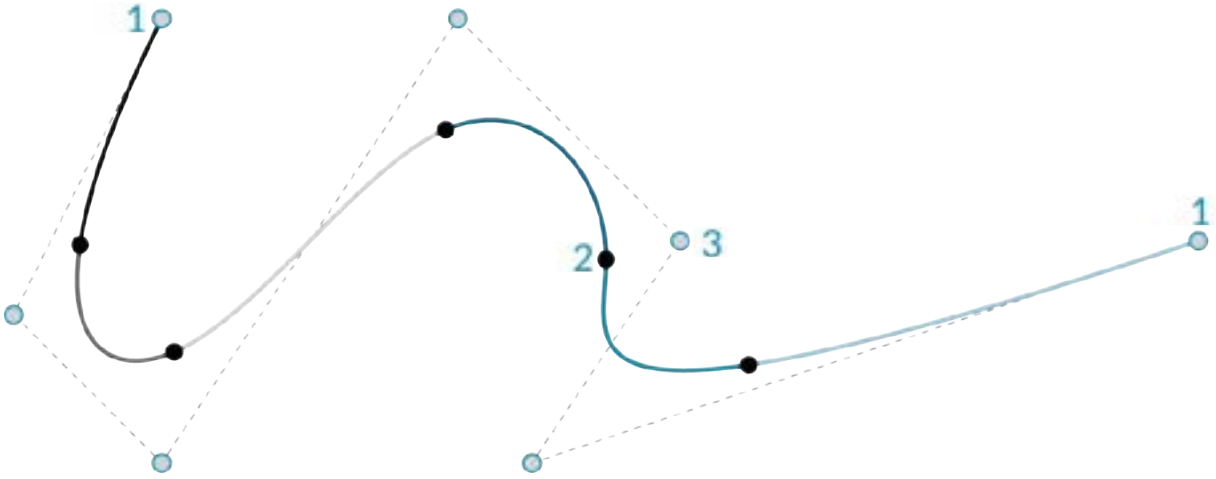
1.4.1. CURVE GEOMETRY

NURBS (non-uniform rational B-splines) are mathematical representations that can accurately model any shape from a simple 2D line, circle, arc, or box to the most complex 3D free-form organic surface or solid. Because of their flexibility and accuracy, NURBS models can be used in any process from illustration and animation to manufacturing.

Since curves are geometric objects, they possess a number of properties or characteristics which can be used to describe or analyze them. For example, every curve has a starting coordinate and every curve has an ending coordinate. When the distance between these two coordinates is zero, the curve is closed. Also, every curve has a number of control-points, if all these points are located in the same plane, the curve as a whole is planar. Some properties apply to the curve as a whole, while others only apply to specific points on the curve. For example, planarity is a global property while tangent vectors are a local property. Also, some properties only apply to some curve types. So far we've discussed some of Grasshopper's Primitive Curve Components such as: lines, circles, ellipses, and arcs.



1. Line
2. Polyline
3. Circle
4. Ellipse
5. Arc
6. NURBS Curve
7. Polycurve



1. End Point
2. Edit Point
3. Control Point

1.4.1.1. NURBS CURVES

Degree: The degree is a positive whole number. This number is usually 1, 2, 3 or 5, but can be any positive whole number. The degree of the curve determines the range of influence the control points have on a curve; where the

higher the degree, the larger the range. NURBS lines and polylines are usually degree 1, NURBS circles are degree 2, and most free-form curves are degree 3 or 5.

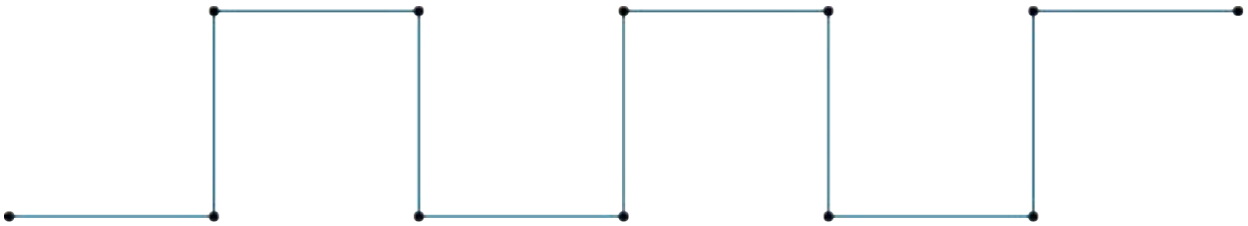
Control Points: The control points are a list of at least degree+1 points. One of the easiest ways to change the shape of a NURBS curve is to move its control points.

Weight: Control points have an associated number called a weight . Weights are usually positive numbers. When a curve's control points all have the same weight (usually 1), the curve is called non-rational, otherwise the curve is called rational. Most NURBS curves are non-rational. A few NURBS curves, such as circles and ellipses, are always rational.

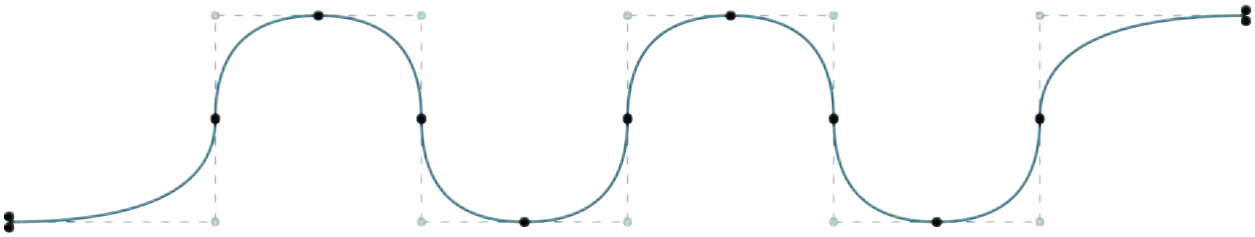
Knots: Knots are a list of (degree+N-1) numbers, where N is the number of control points.

Edit Points: Points on the curve evaluated at knot averages. Edit points are like control points except they are always located on the curve and moving one edit point generally changes the shape of the entire curve (moving one control point only changes the shape of the curve locally). Edit points are useful when you need a point on the interior of a curve to pass exactly through a certain location.

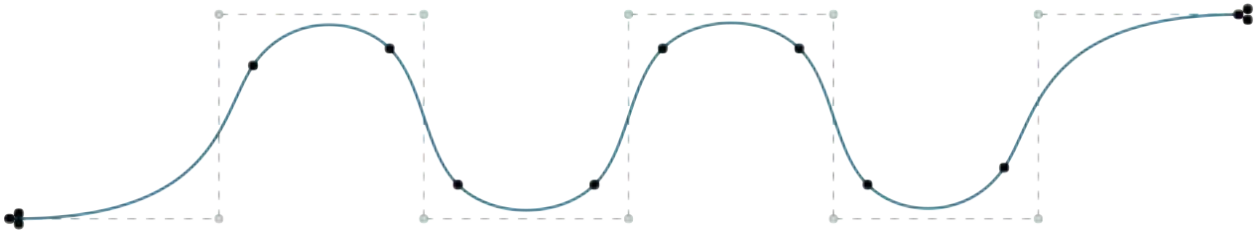
NURBS curve knots as a result of varying degree:



> AD^1 NURBS curve behaves the same as a polyline. AD^1 curve has a knot for every control point.



> D^2 NURBS curves are typically only used to approximate arcs and circles. The spline intersects with the control polygon halfway each segment.



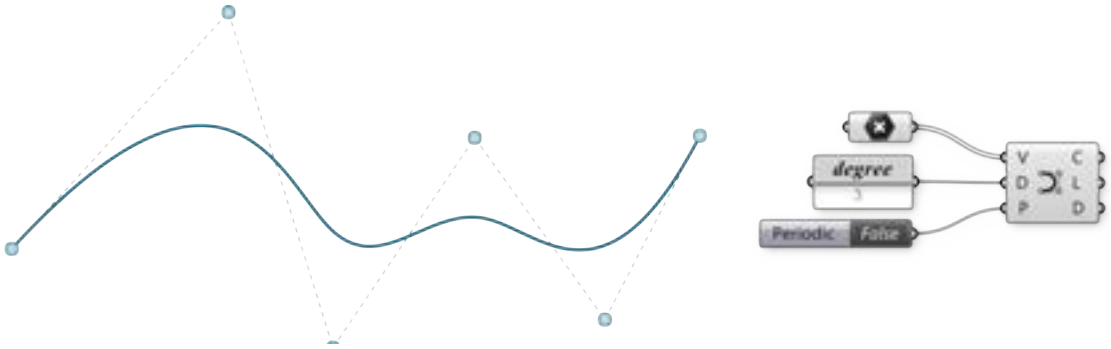
> D^3 is the most common type of NURBS curve and is the default in Rhino. You are probably very familiar with the visual progression of the spline, even though the knots appear to be in odd locations.

1.4.1.2. GRASSHOPPER SPLINE COMPONENTS

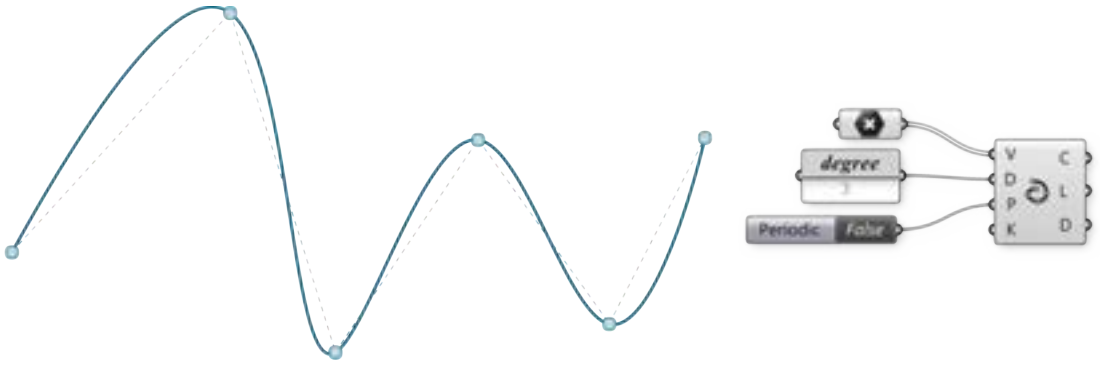
> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Grasshopper has a set of tools to express Rhino's more advanced curve types like nurbs curves and poly curves. These tools can be found in the Curve/Splines tab.

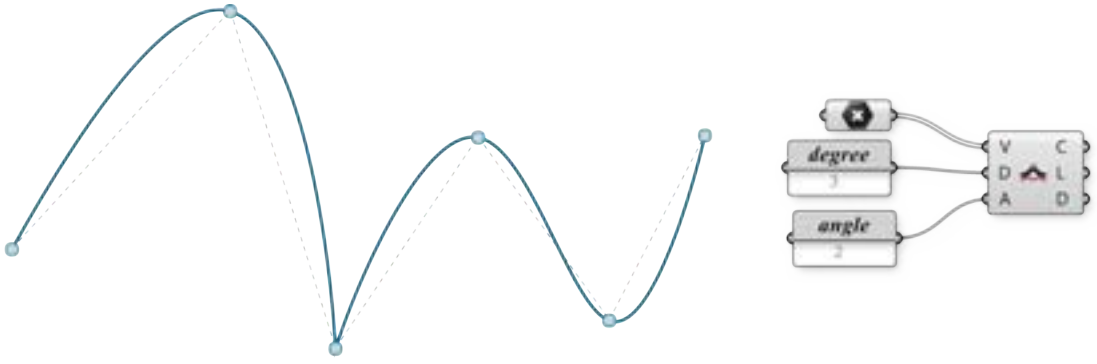
Nurbs Curve (Curve/Spline/Nurbs curve): The Nurbs Curve component constructs a NURBS curve from control points. The V input defines these points, which can be described implicitly by selecting points from within the Rhino scene, or by inheriting volatile data from other components. The Nurbs Curve-D input sets the degree of the curve.



Interpolate Curve (Curve/Spline/Interpolate): Interpolated curves behave slightly differently than NURBS curves. The V-input is for the component is similar to the NURBS component, in that it asks for a specific set of points to create the curve. However, with the Interpolated Curve method, the resultant curve will actually pass through these points, regardless of the curve degree. In the NURBS curve component, we could only achieve this when the curve degree was set to one. Also, like the NURBS curve component, the D input defines the degree of the resultant curve. However, with this method, it only takes odd numbered values for the degree input. Again, the P-input determines if the curve is Periodic. You will begin to see a bit of a pattern in the outputs for many of the curve components, in that, the C, L, and D outputs generally specify the resultant curve, the length, and the curve domain respectively.

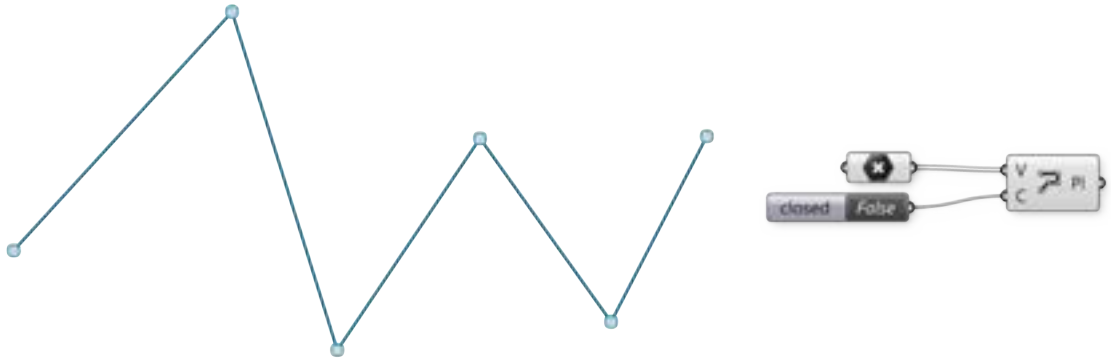


Kinky Curve (Curve/Spline/Kinky Curve): The kinky curve component allows you the ability to control a specific angle threshold, A, where the curve will transition from a kinked line, to a smooth, interpolated curve. It should be noted that the A-input requires an input in radians.



Polyline (Curve/Spline/Polyline): Apolyline is a collection of line segments connecting two or more points, the

resultant line will always pass through its control points; similar to an Interpolated Curve. Like the curve types mentioned above, the V-input of the Polyline component specifies a set of points that will define the boundaries of each line segment that make up the polyline. The C-input of the component defines whether or not the polyline is an open or closed curve. If the first point location does not coincide with the last point location, a line segment will be created to close the loop. The output for the Polyline component is different than that of the previous examples, in that the only resultant is the curve itself.

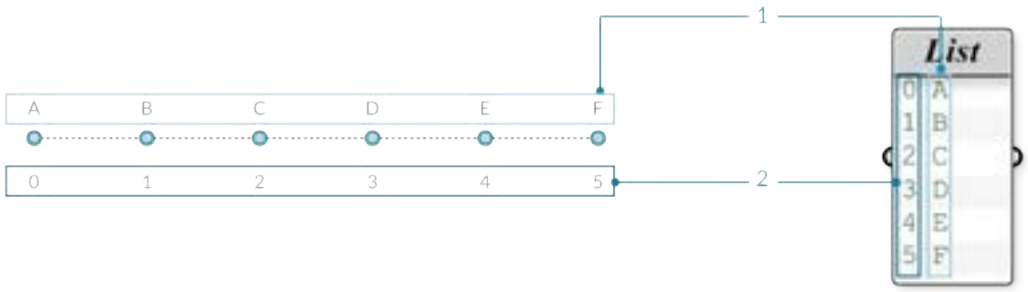


1.4.2. What is a List?

It's helpful to think of Grasshopper in terms of flow, since the graphical interface is designed to have data flow into and out of specific types of components. However, it is the data that define the information flowing in and out of the components. Understanding how to manipulate list data is critical to understanding the Grasshopper plug-in.

Grasshopper generally has two types of data: persistent and volatile. Even though the data types have different characteristics, typically Grasshopper stores this data in an array, a list of variables.

When storing data in a list, it's helpful to know the position of each item in that list so that we can begin to access or manipulate certain items. The position of an item in the list is called its index number.

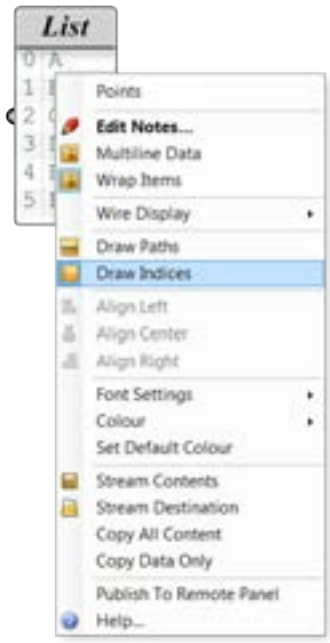


1. List Item
2. Index

The only thing that might seem odd at first is that the first index number of a list is always 0; not 1. So, when we talk about the first item of a list, we actually mean the item that corresponds to index number 0.

For example, if we were to count the number of fingers we have on our right hand, chances are that you would have counted from 1 to 5. However, if this list has been stored in an array, then our list would have counted from 0 to 4. Note, that we still have 5 items in the list; it's just that the array is using a zero-based counting system. The items being stored in the list don't just have to be numbers. They can be any data type that Grasshopper supports, such as points, curves, surfaces, meshes, etc.

Often times the easiest way to take a look at the type of data stored in a list is to connect a Text Panel (Params/Input/Panel) to the output of a particular component. By default, the Text Panel automatically shows all index numbers to the left side of the panel and displays the data items on the right side of the panel. The index numbers will become a crucial element when we begin working with our lists. You can turn the index numbers on and off by right-clicking on the Text Panel and clicking on the "Draw Indices" item in the sub-menu. For now, let's leave the entry numbers turned on for all of our text panels.

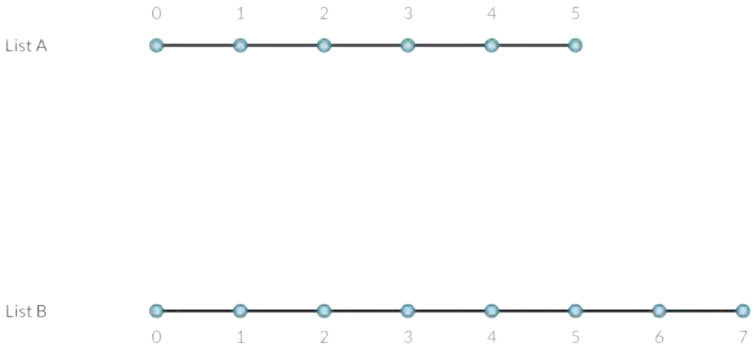

List
0 A
1
2
3
4
5
Points
Edit Notes...
Multiline Data
Wrap Items
Wire Display
Draw Paths
Draw Indices
Align Left
Align Center
Align Right
Font Settings
Colour
Set Default Colour
Stream Contents
Stream Destination
Copy All Content
Copy Data Only
Publish To Remote Panel
Help...

1.4.3. Data Stream Matching

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

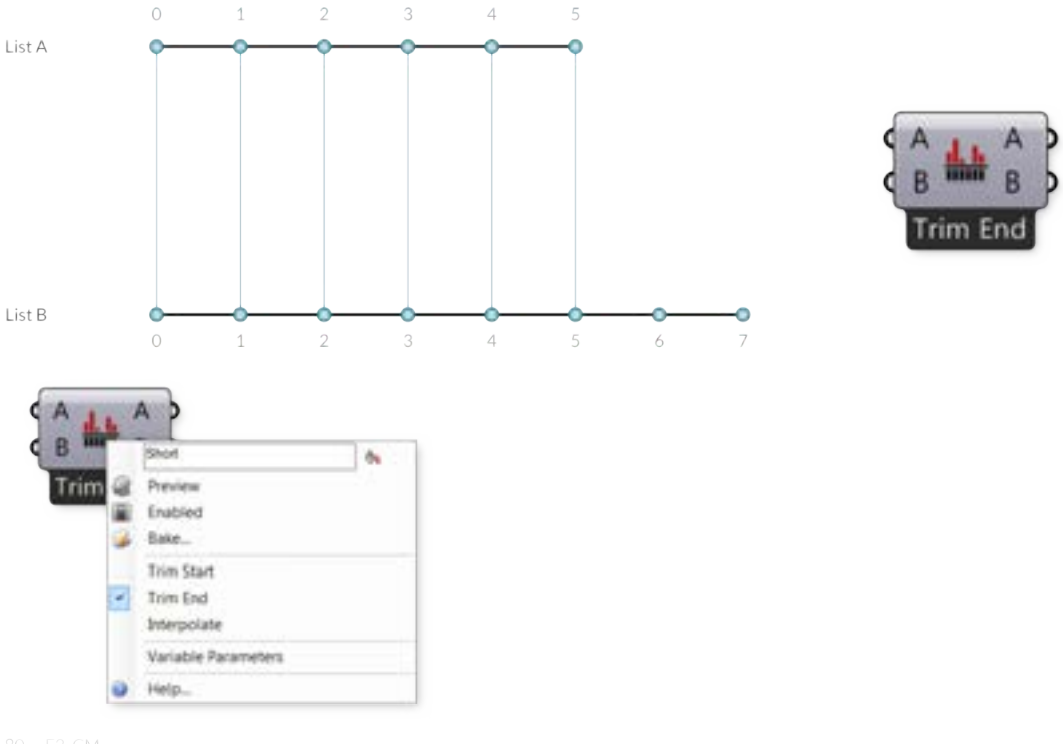
Data matching is a problem without a clean solution. It occurs when a component has access to differently sized inputs. Changing the data matching algorithm can lead to vastly different results.

Imagine a component which creates line segments between points. It will have two input parameters which both supply point coordinates (List A and List B):



As you can see there are different ways in which we can draw lines between these sets of points. New to Grasshopper 0.9 are three components for data matching, found under the Sets/List panel: Shortest List, Longest List, and Cross Reference. These new components allow for greater flexibility within the three basic data matching algorithms. Right clicking each component allows you to select a data matching option from the menu.

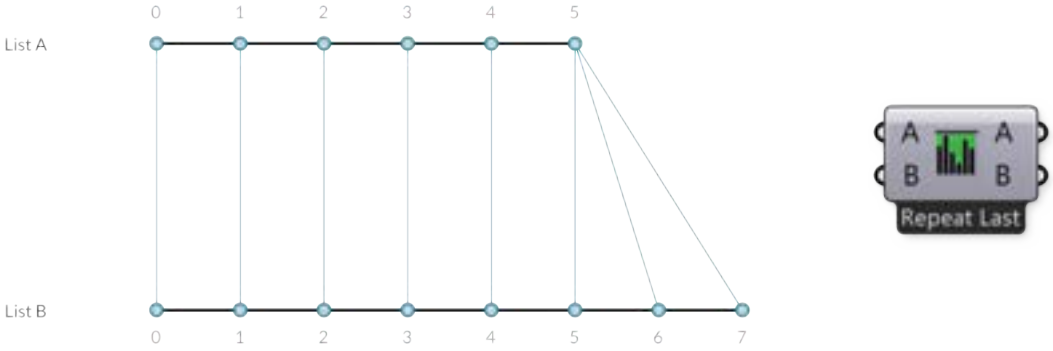
The simplest way is to connect the inputs one-on-one until one of the streams runs dry. This is called the **"Shortest List"** algorithm:



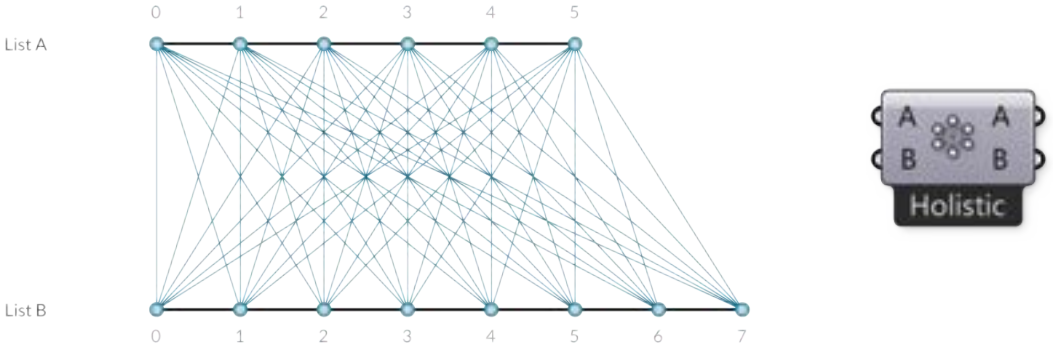
> Select a matching algorithm option from the component menu by right-clicking the component.

The **"Longest List"** algorithm keeps connecting inputs until all streams run dry. This is the default behavior for

components:

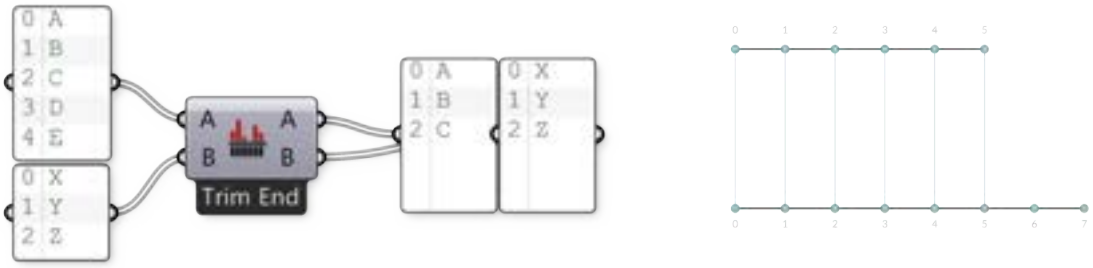


Finally, the **“Cross Reference”** method makes all possible connections:

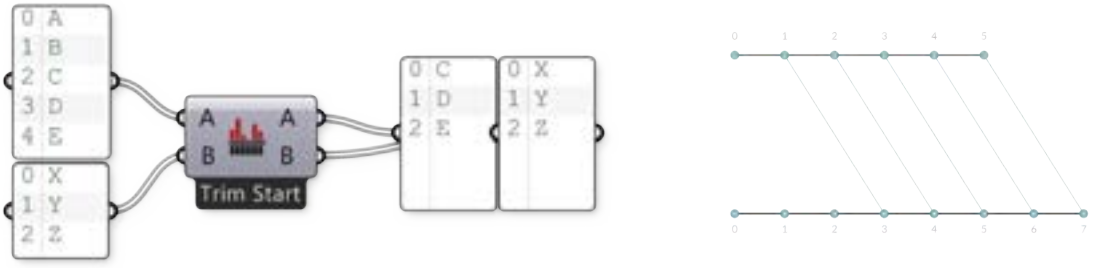


This is potentially dangerous since the amount of output can be humongous. The problem becomes more intricate as more input parameters are involved and when the volatile data inheritance starts to multiply data, but the logic remains the same.

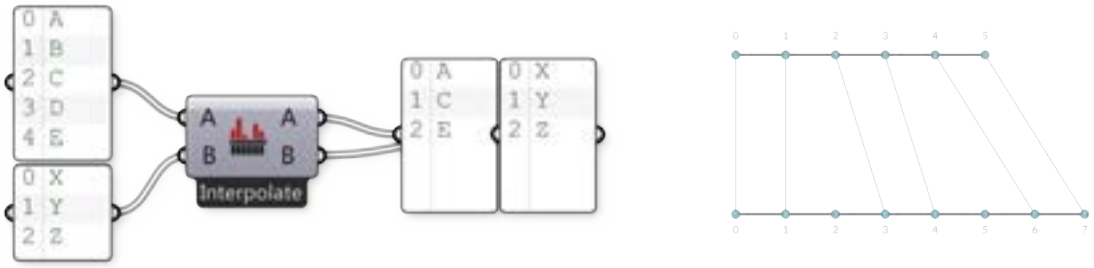
Let’s look more closely at the Shortest List component:



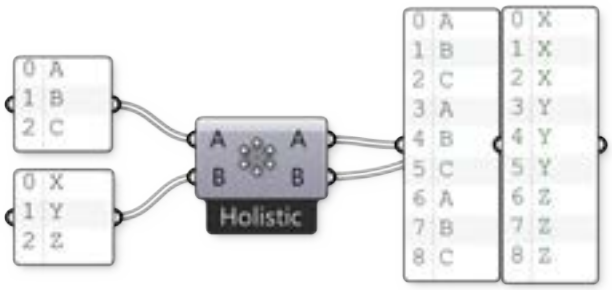
Here we have two input lists {A,B,C,D,E} and {X,Y,Z}. Using the Trim End option, the last two items in the first list are disregarded., so that the lists are of equal length.



Using the Trim Start option, the first two items in the first list are disregarded, so that the lists are of equal length.

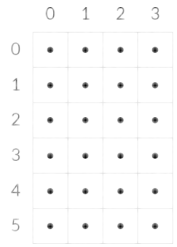


The Interpolate option skips the second and fourth items in the first list. Now let's look at the Cross Reference component:

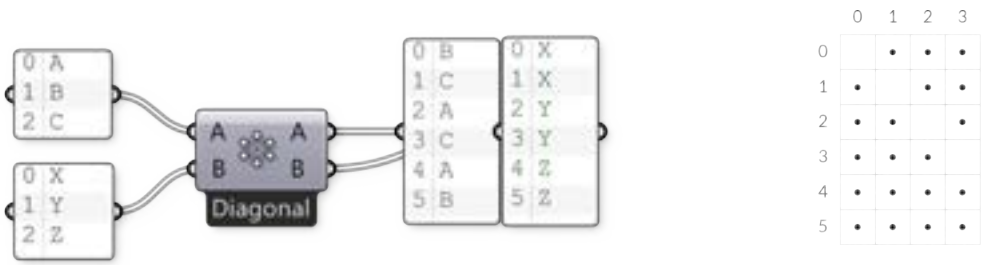


Here we have two input lists {A,B,C} and {X,Y,Z}. Normally Grasshopper would iterate over these lists and only consider the combinations {A,X}, {B,Y} and {C,Z}. There are however six more combinations that are not typically considered, to wit: {A,Y}, {A,Z}, {B,X}, {B,Z}, {C,X} and {C,Y}. As you can see the output of the Cross Reference component is such that all nine permutations are indeed present.

We can denote the behaviour of data cross referencing using a table. The rows represent the first list of items, the columns the second. If we create all possible permutations, the table will have a dot in every single cell, as every cell represents a unique combination of two source list indices.



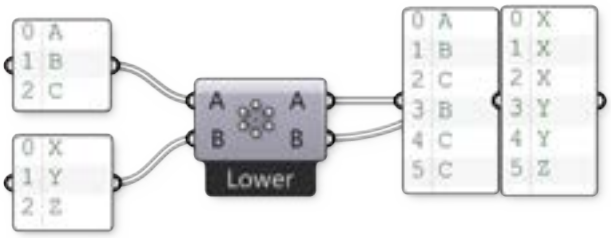
Sometimes however you don't want all possible permutations. Sometimes you wish to exclude certain areas because they would result in meaningless or invalid computations. Acommon exclusion principle is to ignore all cells that are on the diagonal of the table. The image above shows a 'holistic' matching, whereas the 'diagonal' option (available from the Cross Reference]component menu has gaps for {0,0}, {1,1}, {2,2} and {3,3}. If we apply this to our {A,B,C}, {X,Y,Z} example, we should expect to not see the combinations for {A,X}, {B,Y} and {C,Z}:

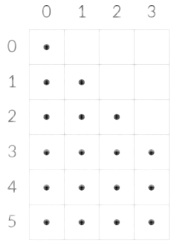


The rule that is applied to 'diagonal' matching is: "Skip all permutations where all items have the same list index". 'Coincident' matching is the same as 'diagonal' matching in the case of two input lists, but the rule is subtly

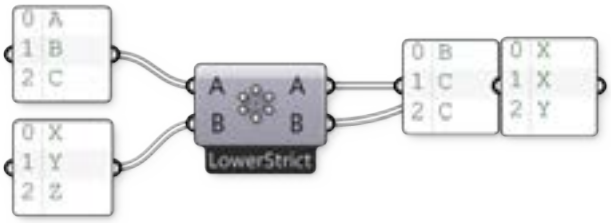
different: “Skip all permutations where any two items have the same list index”.

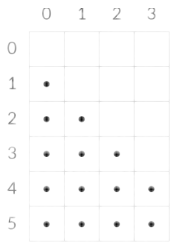
The four remaining matching algorithms are all variations on the same theme. ‘Lower triangle’ matching applies the rule: “Skip all permutations where the index of an item is less than the index of the item in the next list”, resulting in an empty triangle but with items on the diagonal.





‘Lower triangle (strict)’ matching goes one step further and also eliminates the items on the diagonal:





‘Upper Triangle’ and ‘Upper Triangle (strict)’ are mirror images of the previous two algorithms, resulting in empty triangles on the other side of the diagonal line.

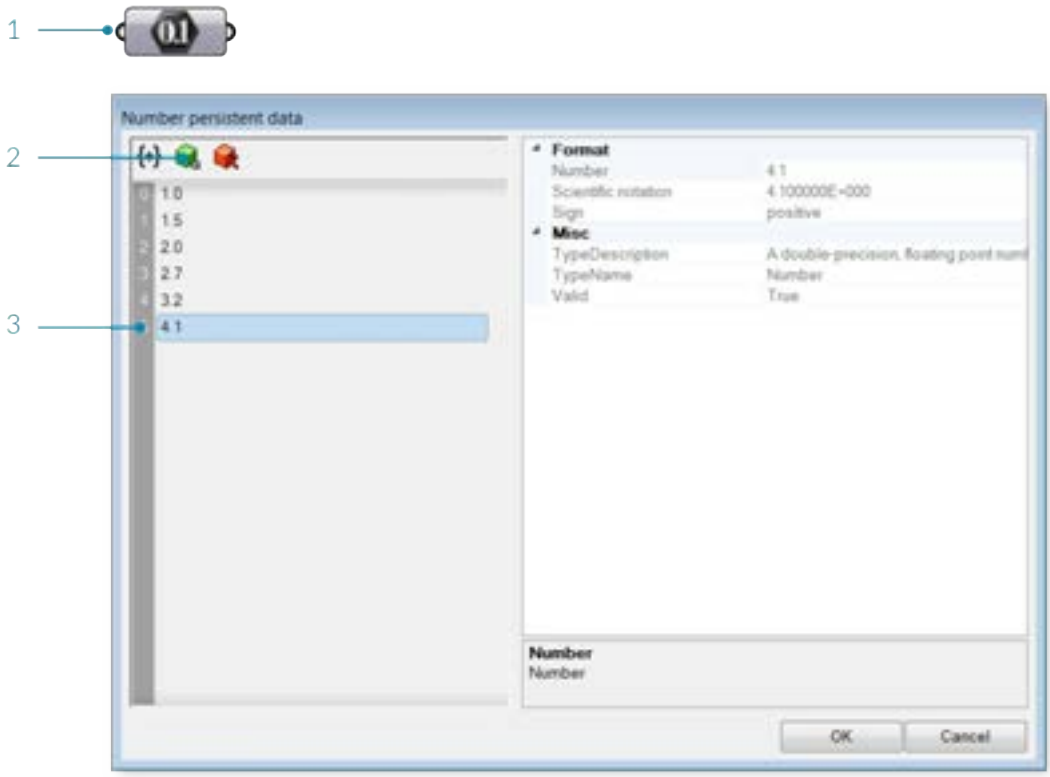
1.4.4. Creating Lists

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

There are many different ways to generate lists in Grasshopper. Below, we'll look at a few different methods for generating lists and then look at how the data can be used to convey information in the viewport via a visualization.

1.4.4.1. MANUAL LIST CREATION

Perhaps the easiest way to create a list (and one of the most over-looked methods) is to manually type in a list of values into a parameter. Using this method puts added responsibility on the user because this method relies on direct user input (ie. persistent data) for the list creation. In order to change the list values, the user has to manually type in each individual valuewhich can be difficult if the list has many entries. There are several ways to manually create a list. One way is to use a Number paramter. Right click the Number parameter and select "Manage Number Collection."



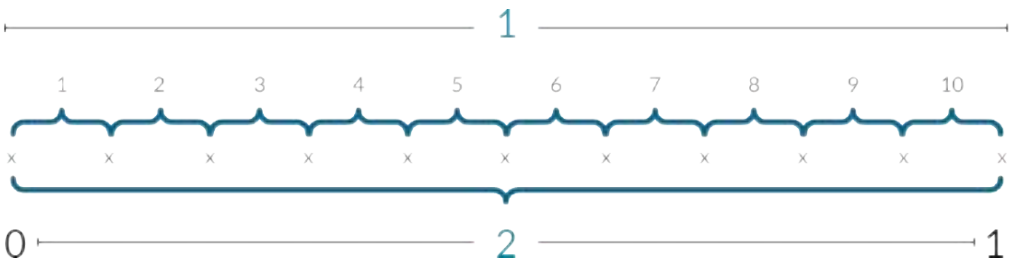
> 1. Right click the number component to open the Number collection Manager.
> 2. Click the Add Item icon to add a number to the list.
> 3. Double click the number to change its value.

Another method is to manually enter the list items into a panel. Make sure that "Multiline Data" is deselected.

1.4.4.2. RANGE

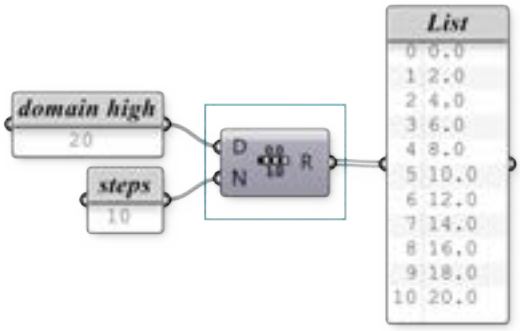
The Range component, found under Sets/Sequence/Range, creates a list of evenly spaced numbers between a low and a high value called the Domain. A domain (also sometimes referred to as an interval) is every possible number between two numeric extremes.

ARange component divides a numeric domain into even segments and returns a list of values.



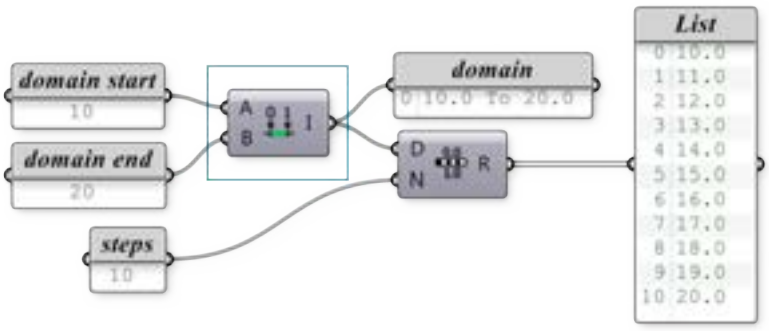
1. Number of Steps = 10
2. Domain goes from 0 to 1
3. Total number of points = 11

In the example below, the numeric domain has been defined as every possible number between 0 and 20. The Range component takes that domain and divides it up by the number of steps (in this case 10). So, we have 10 even spaced segments. The Range component returns a list of values. Because it keeps the first and the last values in the list, the output of a Range component is always one more than the number of steps. In the example above, we created 10 steps, so the Range component returns 11 values.



Create a list using the Range component by specifying a Domain and number of steps.

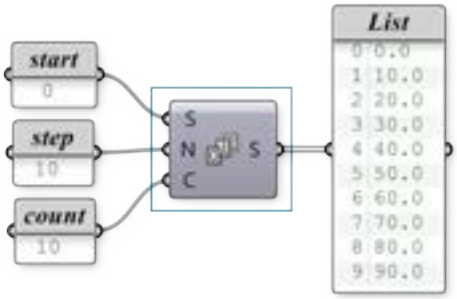
You may have noticed something a little quirky about the setup we just made. We know that a domain is always defined by two values (a high and low value). Yet, in our definition we simply connected a single value to the domain input. In order to avoid errors, Grasshopper makes an assumption that you are trying to define a domain between zero and some other number (our slider value). In order to create a range between two numbers that doesn't start at zero, we must use the Construct Domain component to specify the domain.



> To create a Range from a domain that does not start at zero, use the Construct Domain component.

1.4.4.3. SERIES

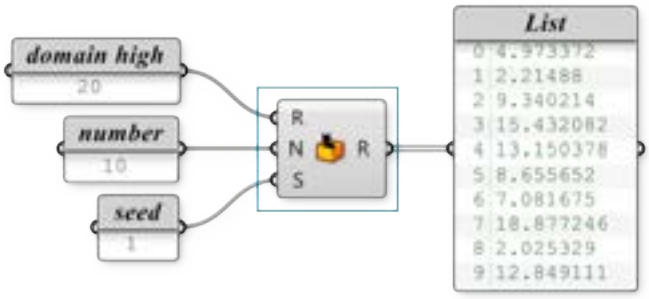
The Series component is similar to the Range component, in that, it also creates a list of numbers. However a Series component is different because it creates a set of discreet numbers based on a start value, step size, and the number of values in the series.



> The Series component creates a list based on a start value, step value, and the number of values in the list.

1.4.4.4. RANDOM

The Random Component (Sets/Sequence/Random) can be used to generate a list of pseudo random numbers. They are referred to as “pseudo” random because the number sequence is unique but stable for each seed value. Thus, you can generate an entirely new set of random numbers by changing the seed value (S-input). The domain, as in the previous example, is a defined interval between two numeric extremes.

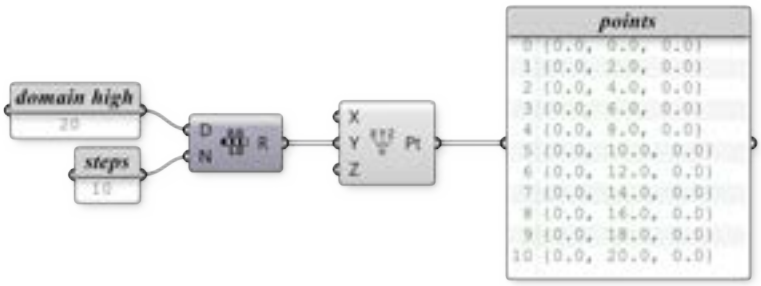


1.4.5. List Visualization

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Understanding lists in Grasshopper can be difficult without being able to see the data flowing from one component to the next. There are several ways to visualize lists that can help to understand and manipulate data.

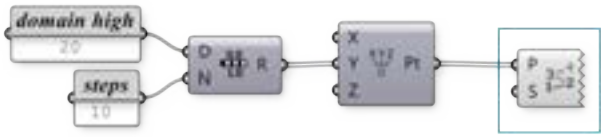
There are many different ways to visualize a list of data. The most common way is to create some geometry with the list of data. By connecting the R output of the Range component to the Y input of the Construct Point component, we can see an array of points in the Y direction.



Lets look at some components that can help us understand the data.

1.4.5.1. THE POINT LIST COMPONENT

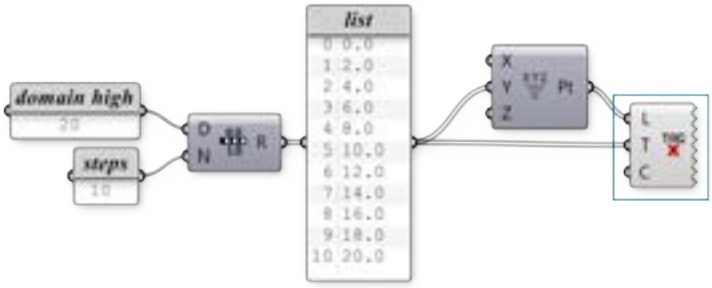
The Point List component is an extremely useful tool for visualizing the order of a set of points in a list. Essentially, the Point List component places the index item number next to the point geometry in the viewport. You can also specify whether or not you want to draw the number tags, the connection lines, or the size of the text tags.



> You can visualize the order of a set of points using the Point List component.

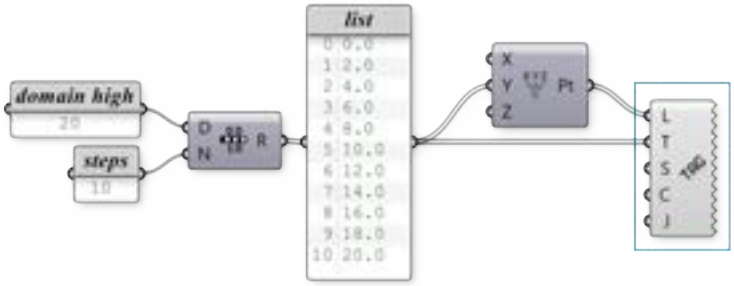
1.4.5.2. TEXT TAGS

The text tag component allows you to draw little strings (a string is a set of ASCII characters) in the viewport as feedback items. Text and location are specified as input parameters. When text tags are baked into the scene, they turn into Text Dots. The other interesting thing about Text Tags is that they are viewport independent - meaning the tags always face the camera (including perspective views) and they always remain the same size on the screen regardless of your zoom settings.



> You can visualize any string information in the viewport using the Text Tag component. In this setup, we have decided to display the value of each point on top of each point locaiton. We could have assigned any text to display.

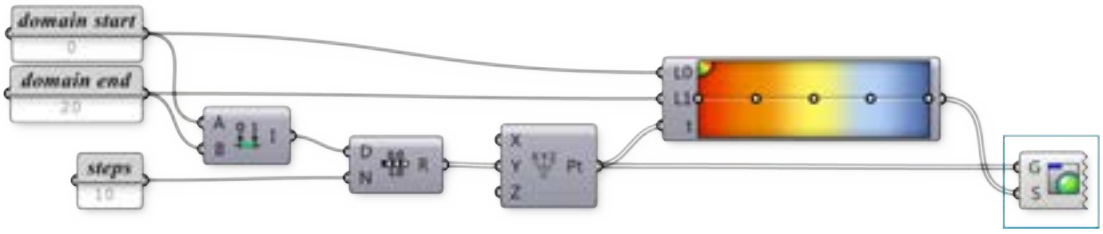
The Text Tag 3d component works very similarly to the Text Tag component. They differ, in that, when Text Tag 3d objects are baked into the scene, they become Text objects in Rhino. The scale of the Text Tag 3d font can also be controlled via an input (which is inaccessible in the Text Tag component).



> You can use a Text Tag 3d component to visualize information like a Text object in Rhino.

1.4.5.3. COLOR

One of the other things we can do to visualize the list data is to assign color to the geometry. Grasshopper has limited ‘rendering’ capabilities, but we can control simple Open GL settings like color, specular color, transparency, etc. The L0 value represents the low end (left side) of the gradient, whereas the L1 value represents the upper end (right side). These values correspond to the start and end of our domain. The t-values are the elements in the list that will get mapped somewhere within the L0 and L1 range. The output of the gradient is a list of RGB color values which correspond to each point in our list. Right-click on the Gradient to set one of the gradient presets, or define your own using the color node points.



1. Points
2. Point list
3. Text Tag
4. Text Tag 3D
5. Custom color preview

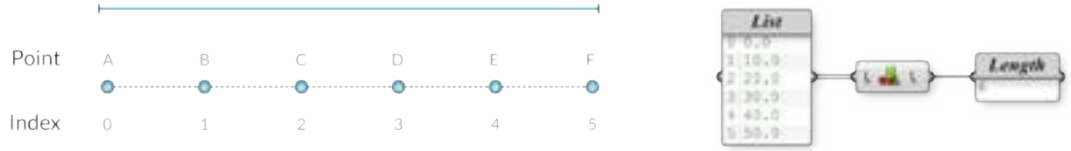
1.4.6. List Management

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

One of the most powerful features of Grasshopper is the ability to quickly build and manipulate various lists of data. We can store many different types of data in a list (numbers, points, vectors, curves, surfaces, breps, etc.) and there are a number of useful tools found under the Sets/List subcategory.

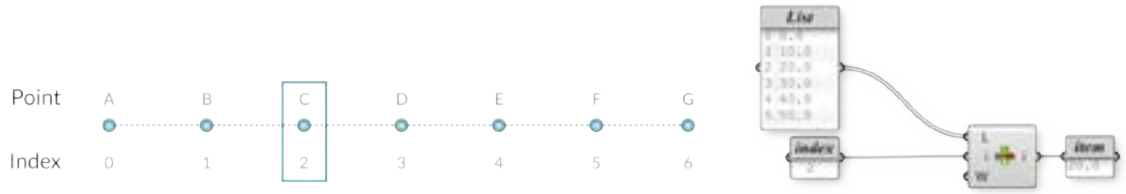
1.4.6.1. LIST LENGTH

The List Length component (Sets/List/List Length) essentially measures the length of the List. Because our lists always start at zero, the highest possible index in a list equals the length of the list minus one. In this example, we have connected our base List to the List Length-L input, showing that there are 6 values in the list.



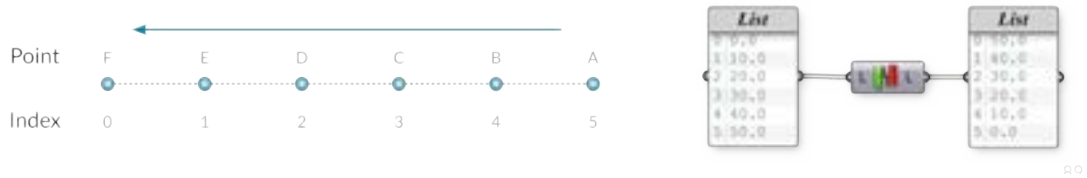
1.4.6.2. LIST ITEM

Our List is fed into a List Item component (Sets/List/List Item) in order to retrieve a specific data item from within a data set. When accessing individual items in a list, we have to specify the i-input; which corresponds to the index number we would like to retrieve. We can feed a single integer or a list of integers into the i-input depending on how many items we would like to retrieve.The L-input defines the base list which we will be analyzing. In this example, we have set the i-input to 2 so the List Item component returns the data item associated with the 3rd entry number in our list.



1.4.6.3. REVERSE LIST

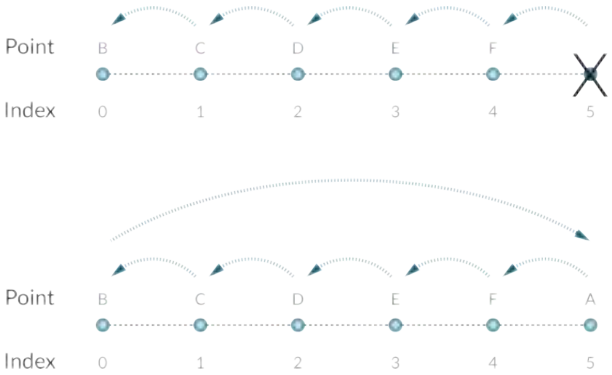
We can invert the order of our list by using a Reverse List component (Sets/List/Reverse). If we input an ascending list of numbers from 0.0 to 50.0 into the Reverse List component; the output returns a descending list from 50.0 to 0.0.

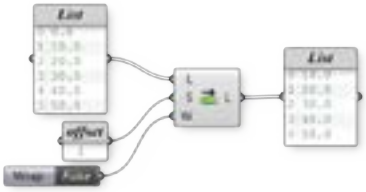


1.4.6.4. SHIFT LIST

The Shift List component (Sets/Sequence/Shift List) will either move the list up or down a number of increments depending on the value of the shift offset. We have connected the List output into the Shift-L input, while also connecting a number to the Shift-S input. If we set the offset to -1, all values of the list will move down by one entry

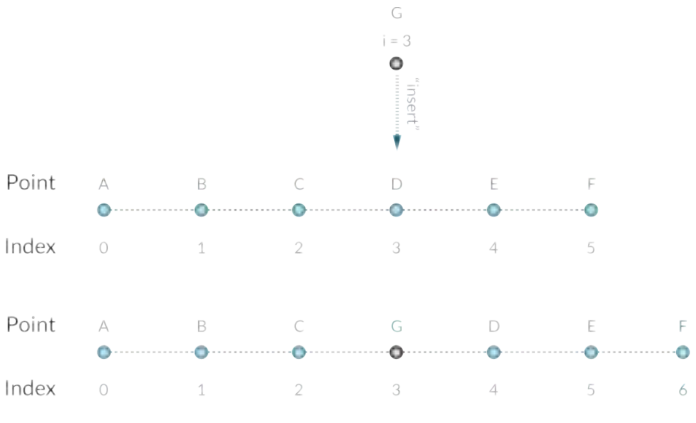
number. Likewise, if we change the offset to +1, all values of the list will move up by one entry number. If Wrap input equals True, then items that fall off the ends are re-appended to the beginning or end of the list. In this example, we have a shift offset value set to +1, so that our list moves up by one entry number. Now, we have a decision to make on how we would like to treat the first value. If we set the Wrap value to False, the first entry will be shifted up and out of the list, essentially removing this value from the data set (so, the list length is one less than it was before). However, if we set the wrap value to True, the first entry will be moved to the bottom of the list

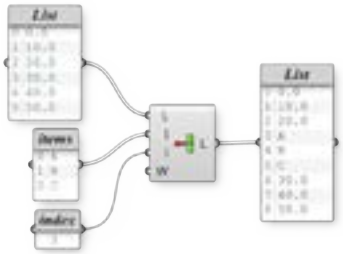




1.4.6.5. INSERT ITEMS

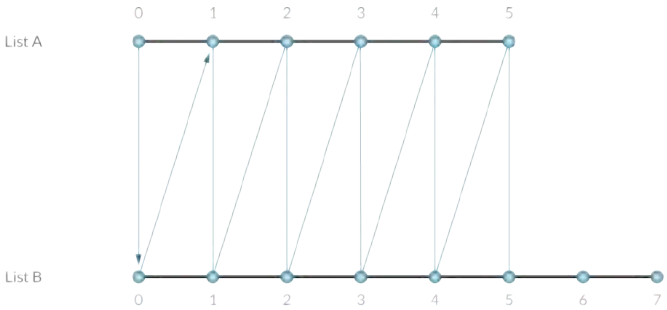
The Insert Items component (Sets/Lists/Insert Items) enables you to insert a collection of items into a list. In order for this to work properly, you need to know the items you want to insert and the index position for each new item. In the example below, we will insert the letters A, B, and C into index position three.

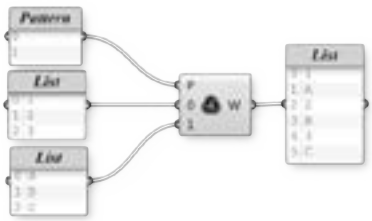


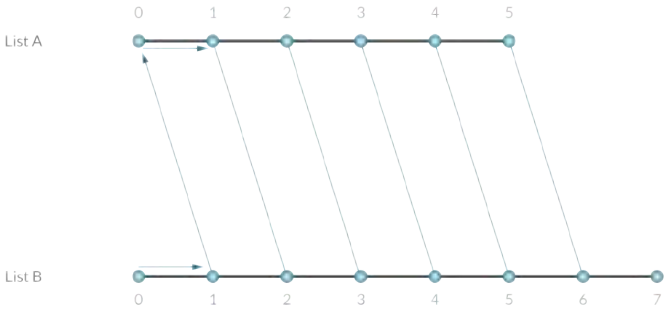


1.4.6.6. WEAVE

The Weave component (Sets/Lists/Weave) merges two or more lists together based on a specified weave pattern (P input).When the pattern and the streams do not match perfectly, this component can either insert nulls into the output streams or it can ignore streams which have already been depleted.

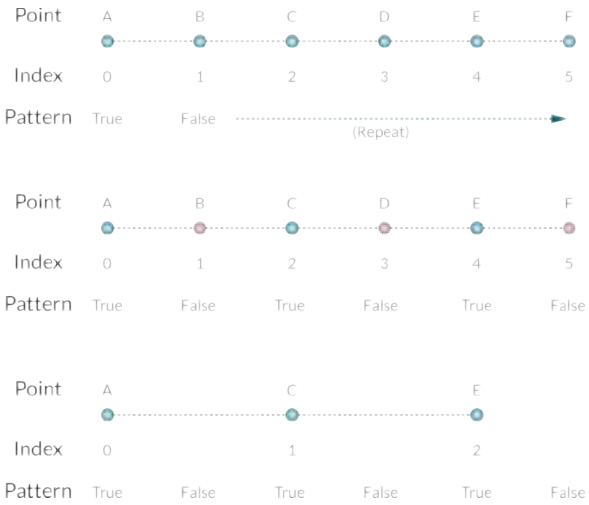






1.4.6.7. CULL PATTERN

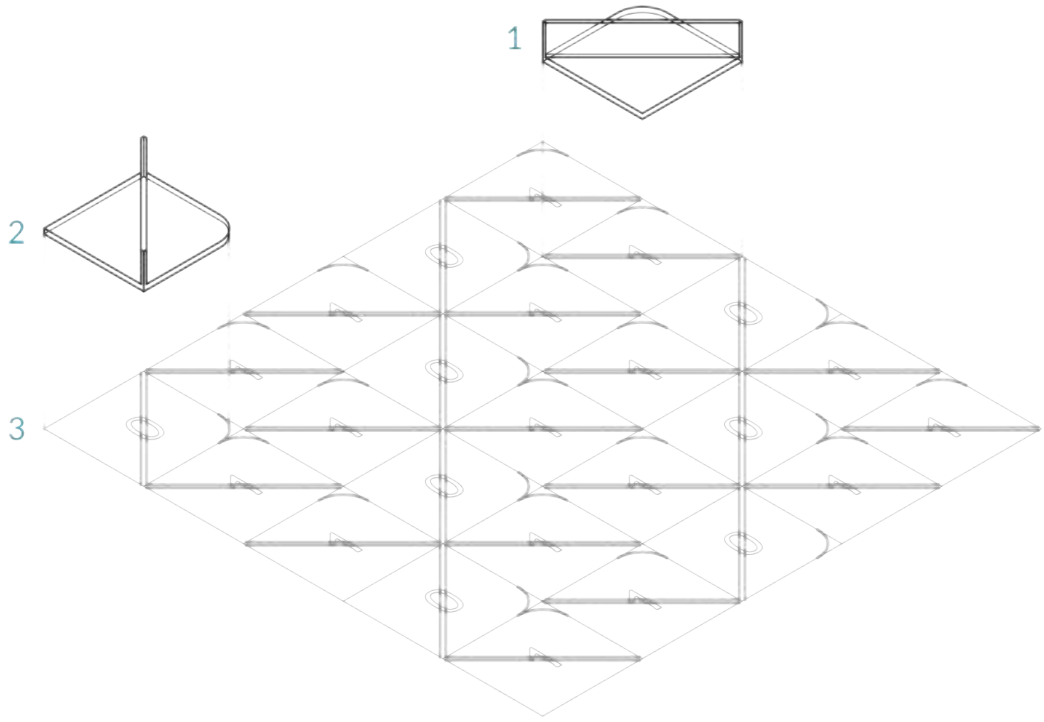
The Cull component (Sets/Sequence/Cull Pattern) removes elements in a list using a repeating bit mask. The bit mask is defined as a list of Boolean (true or false) values. The bit mask is repeated until all elements in the data list have been evaluated.



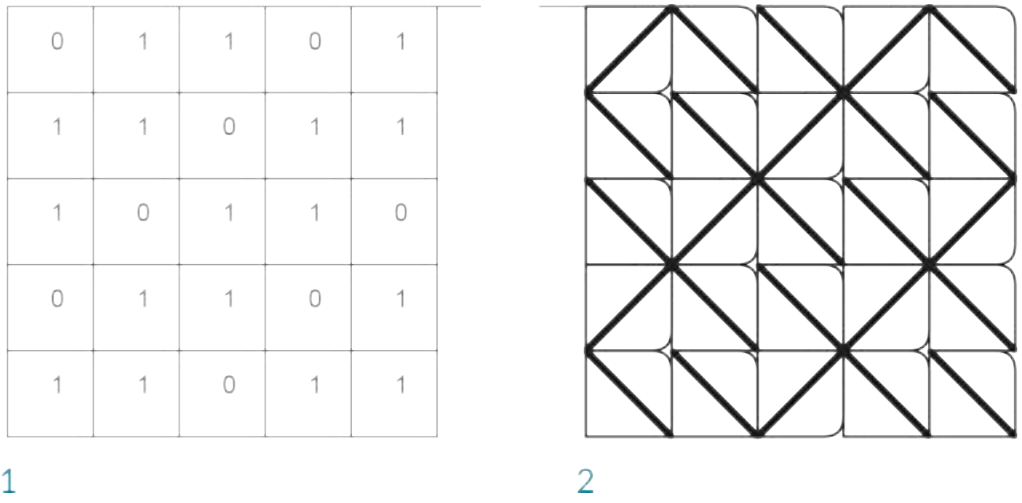
1.4.7. WORKING WITH LISTS

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Lets take a look at an example using the components from the previous section. In this example, we are creating a tile pattern by mapping geometry to a rectangular grid. The pattern is created by using the List Item component to retrieve the desired tile from a list of geometry.

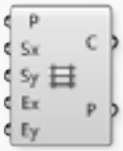


1. Geometry corresponding to index 1
2. Geometry corresponding to index 0
3. Rectangular grid

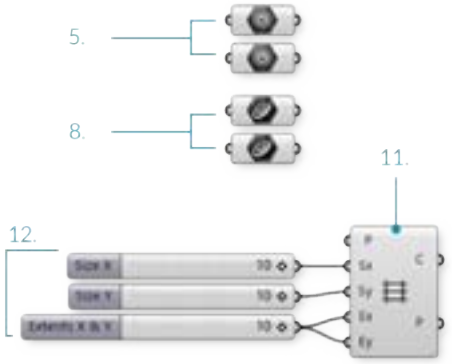


1. Mapping pattern
2. Mapped geometry

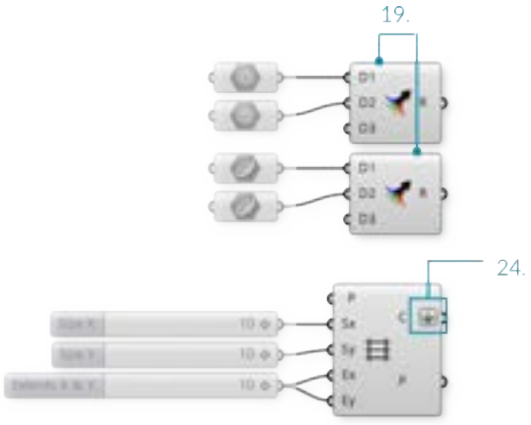
01.	Start a Rhinoceros File.	
02.	Create two equally sized squares.	

03.	Create different geometries in each square. In the example shown above, we created a simple surface with a tab. The tab is filleted to demonstrate the orientation and the base is filleted to distinguish the two geometries.	
04.	Start a new definition, type Ctrl+N (in Grasshopper).	
05.	**Params/Geometry/Geometry** – Drag and drop two **Geometry** parameters onto the canvas.	
06.	Right-Click the first **Geometry** Parameter and select set one Geometry. Set the first Geometry that you are referencing.	
07.	Right-Click the second **Geometry** Parameter and select set one Geometry. Set the second Geometry that you are referencing. It is possible to reference multiple geometries in a single parameter, but for simplicity were are using two separate parameter components.	
08.	**Params/Geometry/Curve** – Drag and drop two **Curve** parameters onto the canvas.	
09.	Right-Click the first **Curve** Parameter and select set one Curve. Set the first square that you are referencing.	
10.	Right-Click the second **Curve** Parameter and select set one Curve. Set the second square that you are referencing. Be sure that the geometry and the square that you are referencing correspond.	
11.	**Vector/Grid/Rectangular** – Drag and drop a **Rectangular Grid** component onto the canvas.	
12.	**Params/Input/Slider** - Drag and drop three **Number Sliders** on the canvas.	
13.	Double-click on the first **Number Slider** and set the following: Rounding: Integers Lower Limit: 0 Upper Limit: 10 Value: 10	
14.	Double-click on the second **Number Slider** and set the following: Rounding: Integers Lower Limit: 0 Upper Limit: 10 Value: 10	
15.	Double-click on the third **Number Slider** and set the following: Name: Extents X & Y Rounding: Integers Lower Limit: 0 Upper Limit: 10 Value: 10	
16.	Connect the first **Number Slider** to the Size X (Sx) input of the **Rectangular Grid** component.	

17.	Connect the second **Number Slider** to the Size Y (Sy) input of the **Rectangular Grid** component.	
18.	Connect the third **Number Slider** to the Extent X (Ex) input and the Extent Y (Ey) input of the **Rectangular Grid** component.	

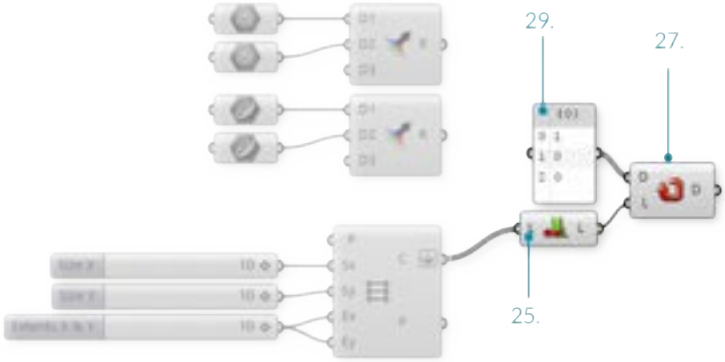


19.	**Sets/Tree/Merge** – Drag and drop two **Merge** components onto the canvas.	
20.	Connect the first **Geometry** parameter to Data Stream 1 (D1) input of the first **Merge** component.	
21.	Connect the second **Geometry** parameter to Data Stream 2 (D2) input of the first **Merge** component.	
22.	Connect the first **Curve** parameter to Data Stream 1 (D1) input of the second **Merge** component.	
23.	Connect the second **Curve** parameter to Data Stream 1 (D2) input of the second **Merge** component.	
24.	Right-click the Cells (C) output of the **Rectangular Grid** component and select Flatten.	

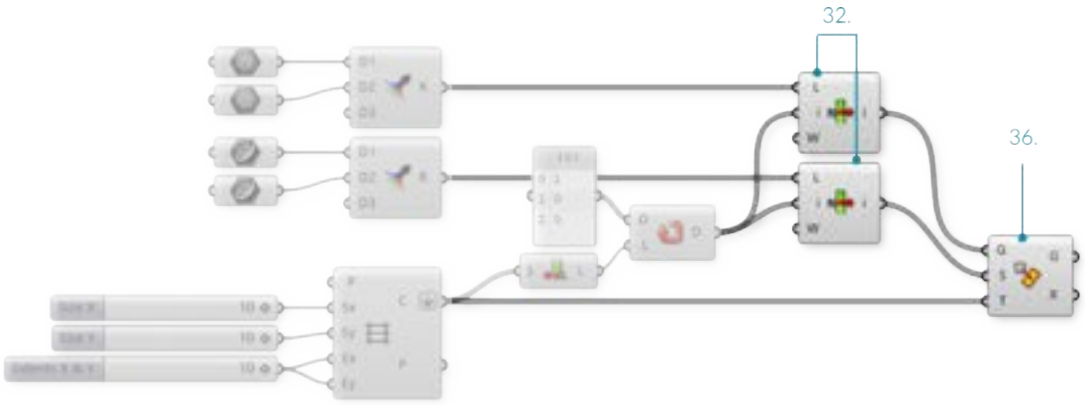


25.	**Sets/List/List Length** – Drag and drop a **List Length** component onto the canvas.	
26.	Connect the Cells (C) output of the **Rectangular Grid** component to the List (L) input of the **List Length** component.	
27.	**Sets/Sequence/Repeat Data** – Drag and drop a **Repeat Data** component onto the canvas.	

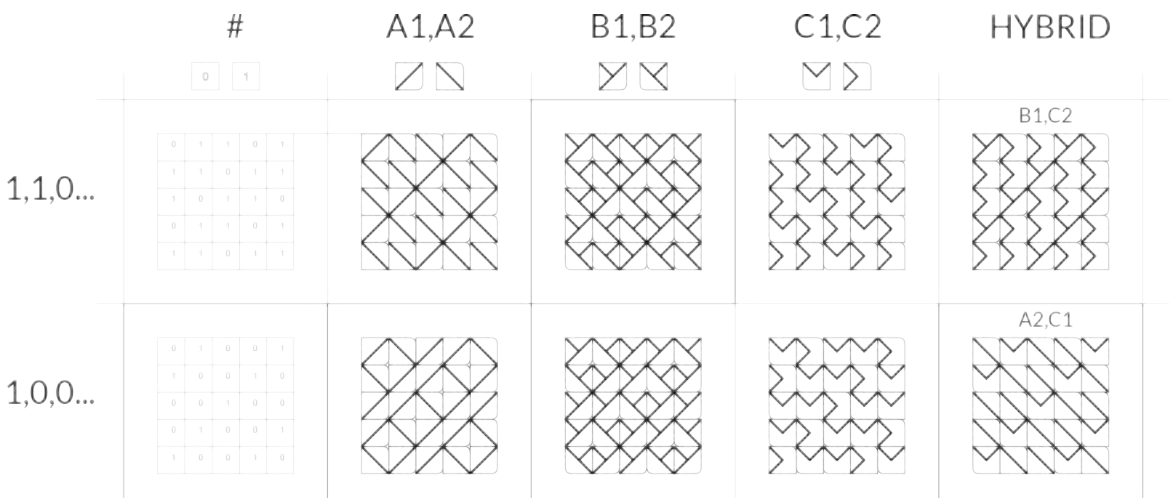
28.	Connect the Length (L) output of the **List Length** component to the Length (L) input of the **Repeat Data** component.	
29.	**Params/Input/Panel** – Drag and drop a **Panel** onto the canvas.	
30.	Double-click the **Panel**. Deselect multiline data, wrap items, and special codes. Enter the following: 1 0 0 > This is the pattern in which the geometries are being distributed. 0 is calling out the first referenced Geometry and 1 is calling out the second referenced Geometry. Changing the number sequence will change the pattern, as will changing the extents of the grid.	
31.	Connect the **Panel** to the Data (D) input of the **Repeat Data** component.	

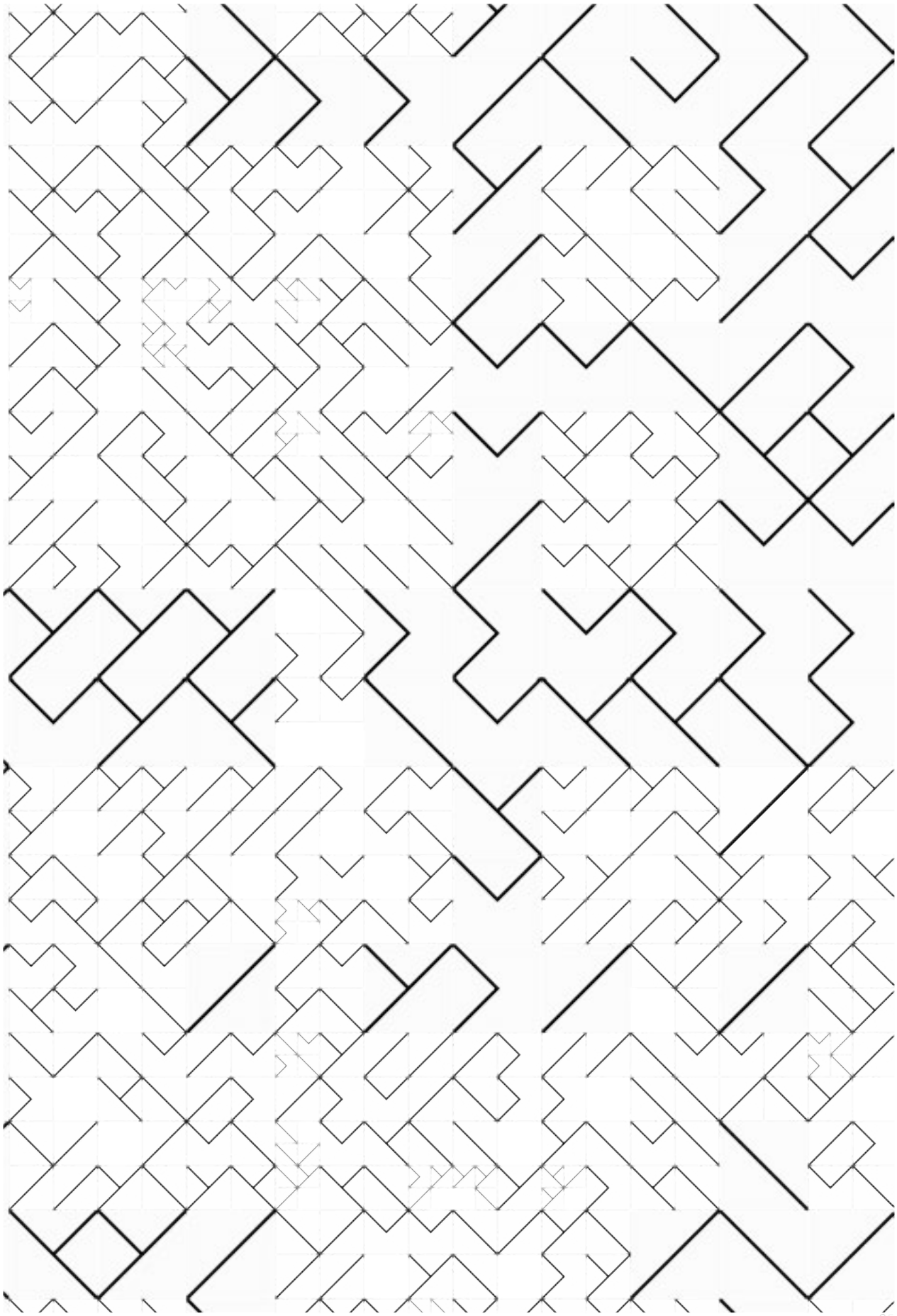


32.	**Sets/List/List Item** – Drag and drop two **List Item** components.	
33.	Connect the Result (R) output of the first **Merge** component to the List (L) input of the first **List Item**component.	
34.	Connect the Result (R) output of the second **Merge** component to the List (L) input of the second **List Item** component.	
35.	Connect the Data (D) output of the **Repeat Data** component to the Index (i) input of the first and second **List Item** components.	
36.	**Transform/Affine/Rectangle Mapping** – Drag and Drop the **Rectangle Mapping** component onto the canvas.	
37.	Connect the Cells (C) output of the **Rectangular Grid** component to the Target (T) input of the **Rectangular Mapping** component.	
38.	Connect the items (I) output of the first **List Item** component to the Geometry (G) input of the **Rectangular Mapping** component.	
39.	Connect the items (I) output of the second **List Item** component to the Source (S) input of the **Rectangular Mapping** component.	



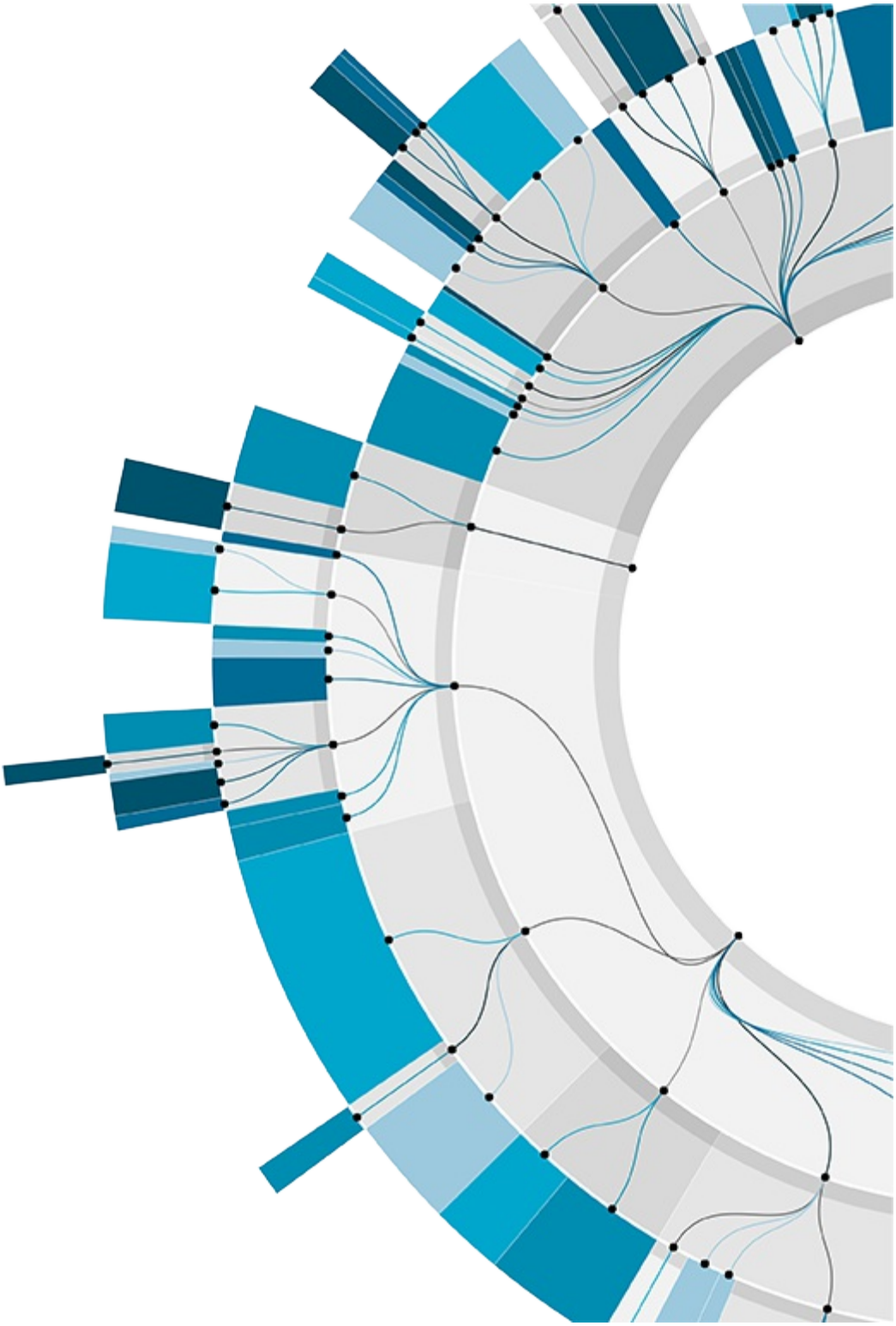
Changing the input geometry and the pattern will change the final tile pattern.





1.5. DESIGNING WITH DATA TREES

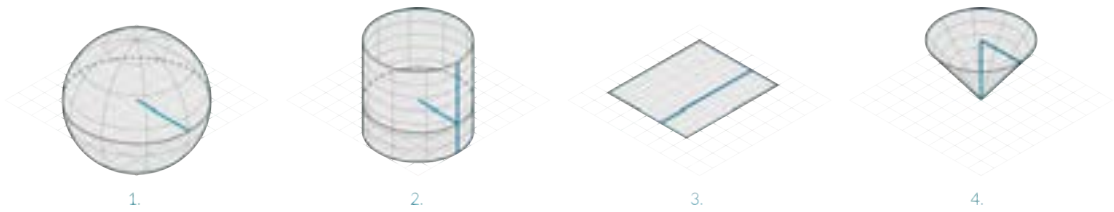
As your definitions increase in complexity, the amount of data flowing through also increases. In order to effectively use Grasshopper, it is important to understand how large quantities of data are stored, accessed, and manipulated.



1.5.1. Surface Geometry

NURBS (non-uniform rational B-splines) are mathematical representations that can accurately model any shape from a simple 2D line, circle, arc, or box to the most complex 3D free-form organic surface or solid. Because of their flexibility and accuracy, NURBS models can be used in any process from illustration and animation to manufacturing.

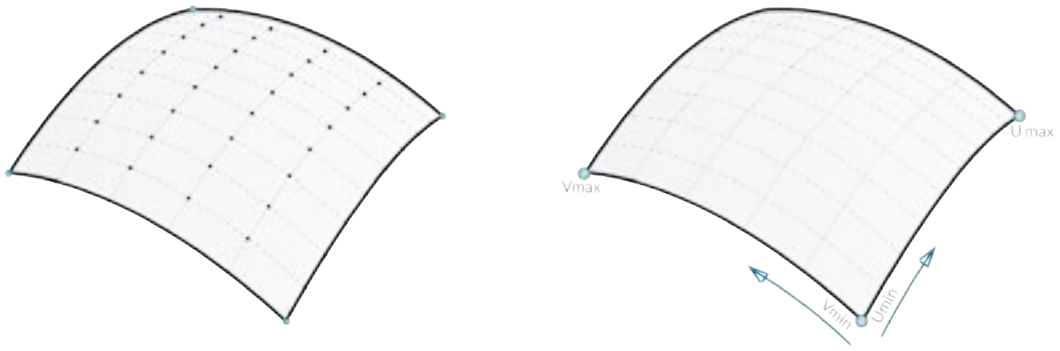
Apart from a few primitive surface types such as spheres, cones, planes and cylinders, Rhino supports three kinds of freeform surface types, the most useful of which is the NURBS surface. Similar to curves, all possible surface shapes can be represented by a NURBS surface, and this is the default fall-back in Rhino. It is also by far the most useful surface definition and the one we will be focusing on.



1. Sphere Primitive [plane, radius]
2. Cylinder Primitive [plane, radius, height]
3. Plane Primitive [plane, width, height]
4. Cone Primitive [plane, radius, height]

1.5.1.1. NURBS SURFACES

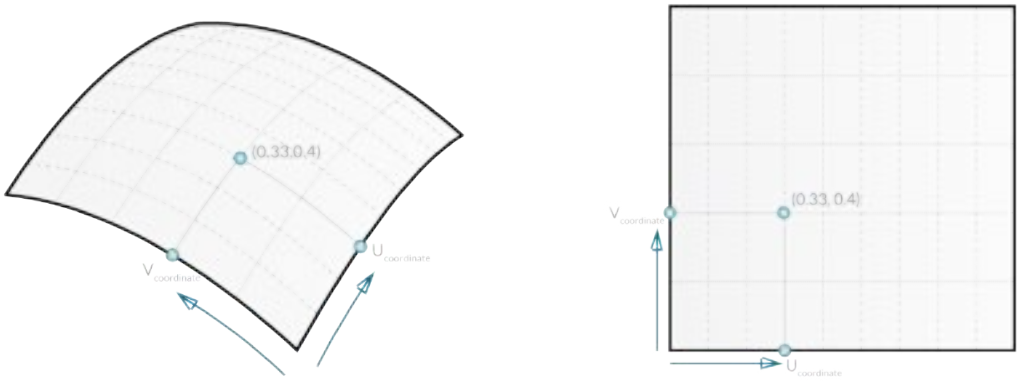
NURBS surfaces are very similar to NURBS curves. The same algorithms are used to calculate shape, normals, tangents, curvatures and other properties, but there are some distinct differences. For example, curves have tangent vectors and normal planes, whereas surfaces have normal vectors and tangent planes.This means that curves lack orientation while surfaces lack direction. In the case of NURBS surfaces, there are in fact two directions implied by the geometry, because NURBS surfaces are rectangular grids of {u} and {v} curves. And even though these directions are often arbitrary, we end up using them anyway because they make life so much easier for us.



You can think of NURBS surfaces as a grid of NURBS curves that go in two directions. The shape of a NURBS surface is defined by a number of control points and the degree of that surface in the u and v directions. NURBS surfaces are efficient for storing and representing free-form surfaces with a high degree of accuracy.

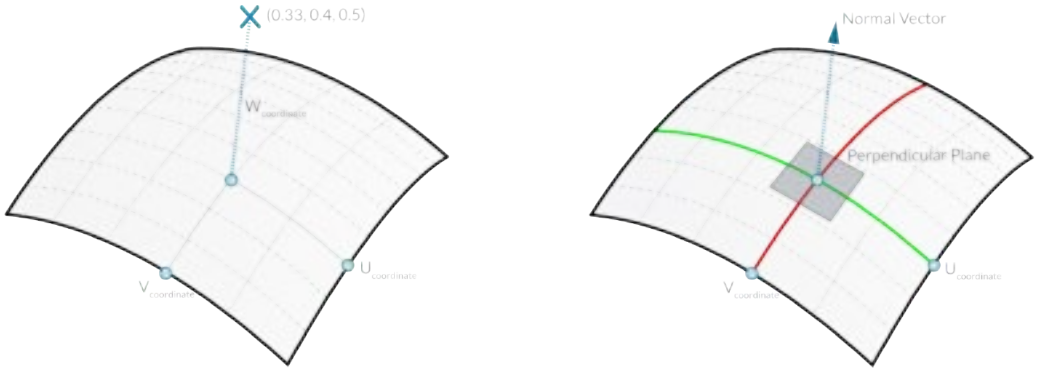
Surface Domain Asurface domain is defined as the range of (u,v) parameters that evaluate into a 3-D point on that surface. The domain in each dimension (u or v) is usually described as two real numbers (u_min to u_max) and (v_min to v_max) Changing a surface domain is referred to as reparameterizing the surface.

In Grasshopper, it is often useful to reparameterize NURBS surfaces so that the u and v domains both range from 0 to 1. This allows us to easily evaluate and operate on the surface.



> Evaluating parameters at equal intervals in the 2-D parameter rectangle does not necessarily translate into equal intervals in 3-D space.

Surface evaluation Evaluating a surface at a parameter that is within the surface domain results in a point that is on the surface. Keep in mind that the middle of the domain (mid-u, mid-v) might not necessarily evaluate to the middle point of the 3D surface. Also, evaluating u- and v-values that are outside the surface domain will not give a useful result.

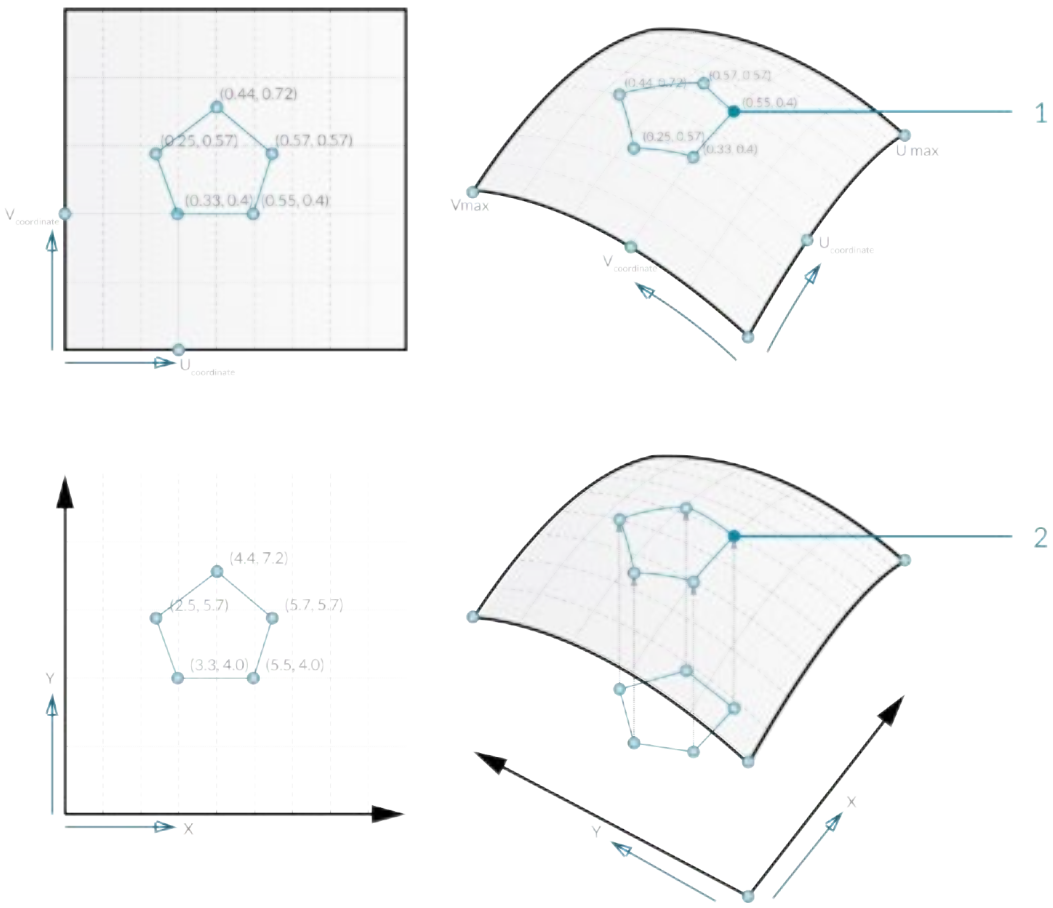


Normal Vectors and Tangent Planes The tangent plane to a surface at a given point is the plane that touches the surface at that point. The z-direction of the tangent plane represents the normal direction of the surface at that point.

Grasshopper handles NURBS surfaces similarly to the way that Rhino does because it is built on the same core of operations needed to generate the surface. However, because Grasshopper is displaying the surface on top of the Rhino viewport (which is why you can't really select any of the geometry created through Grasshopper in the viewport until you bake the results into the scene) some of the mesh settings are slightly lower in order to keep the speed of the Grasshopper results fairly high. You may notice some faceting in your surface meshes, but this is to be expected and is only a result of Grasshopper's drawing settings. Any baked geometry will still use the higher mesh settings.

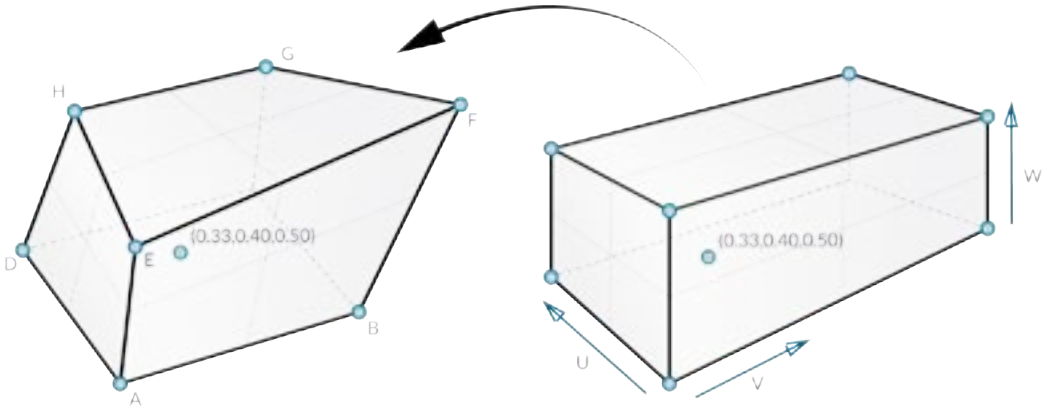
1.5.1.2. PROJECTING SURFACES

In the previous section, we explained that NURBS surfaces contain their own coordinate space desfined by u and v domains. This means that two dimensional geometry that is defined by x and y coordinates can be mapped onto the uv space of a surface. The geometry will stretch and change in response to the curvature of the surface. This is different from simply projecting 2d geometry onto a surface, where vectors are drawn from the 2d geometry in a specified direction until they intersect with the surface.



> You can think of projecting as geometry casting a shadow onto a surface, and mapping as geometry being stretched over a surface.
>
> 1. Mapped geometry defined by uv coordinates
> 2. Projecting geometry onto a surface

Just as 2d geometry can be projected onto the uv space of a surface, 3d geometry that is contained by a box can be mapped to a corresponding twisted box on a surface patch. This operation is called box morphing and is useful for populating curved surfaces with three dimensional geometric components.

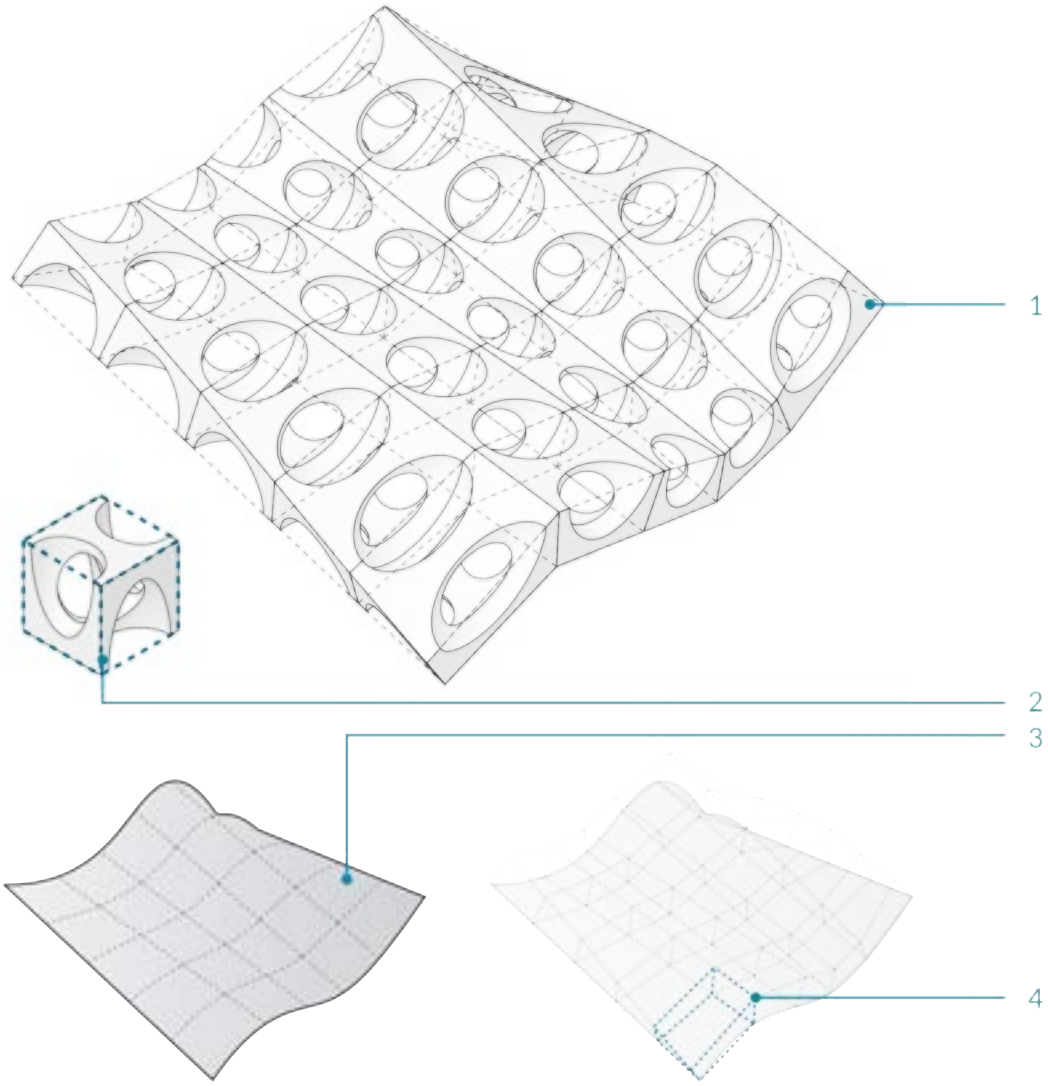


To array twisted boxes on a surface, the surface domain must be divided to create a grid of surface patches. The twisted boxes are created by drawing normal vectors at the corners of each patch to the desired height and creating a box defined by the end points of those vectors and the corner points of the patch.

1.5.1.3. MORPHING DEFINITION

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

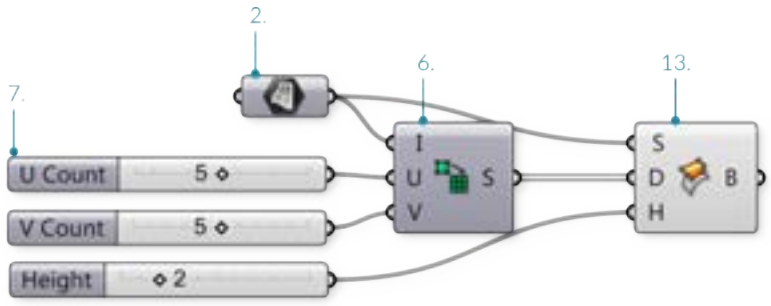
In this example, we will use the box morph component to populate a NURBS surface with a geometric component.



1. NURBS surface populated with component.
2. Original component in reference box.
3. Surface divided into patches.
4. Twisted boxes arrayed on surface.

01.	Start a new definition, type Ctrl+N (in Grasshopper)	
02.	**Params/Geometry/Surface** – Drag and drop a **Surface** parameter onto the canvas This is the surface that we will populate with geometric components.	
03.	**Params/Geometry/Geometry** – Drag a **Geometry** parameter to the canvas This is the component that will be arrayed over the surface.	
04.	Right click the **Surface** Parameter and select “Set One Surface” – select a surface to reference in the Rhino viewport	
05.	Right click the **Geometry** parameter and select “Set One Geometry” – select the your Rhino geometry	

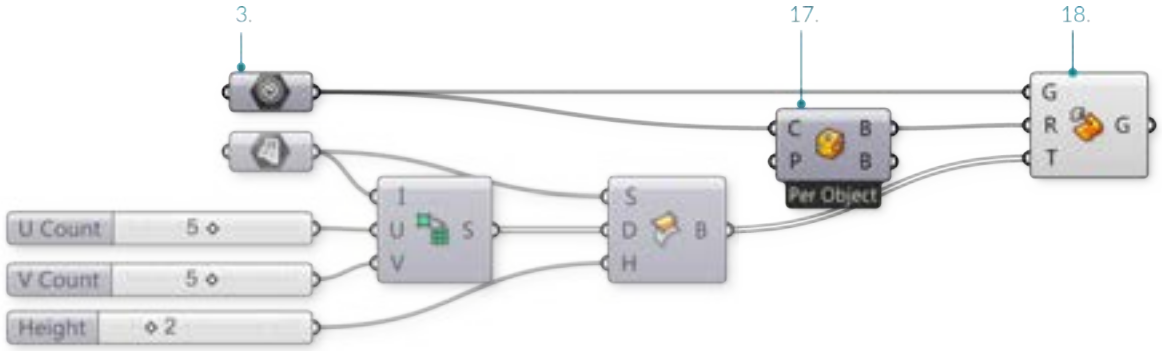
06.	**Maths/Domain/Divide Domain2** – Drag and drop the **Divide Domain2** component onto the canvas	
07.	**Params/Input/Number Slider** – Drag three **Number Sliders** onto the canvas	
08.	Double click the first **Number Slider** and set the following: Rounding: Integer Lower Limit: 0 Upper Limit: 10 Value: 5	
09.	Set the same values on the second and third **Number Sliders**	
10.	Connect the output of the **Surface** parameter to the Domain (I) input of the **Divide Domain2** component	
11.	Connect the first **Number Slider** to the U Count (U) input of the **Divide Domain2** component	
12.	Connect the second **Number Slider** to the V Count (V) input of the **Divide Domain2** component	
13.	**Transform/Morph/Surface Box** – Drag the **Surface Box** component to the canvas	
14.	Connect the output of the **Surface** parameter to the Surface (S) input of the **Surface Box** component	
15.	Connect the Segements (S) output of the **Divide Domain2** component to the Domain (D) input of the **Surface Box** component	



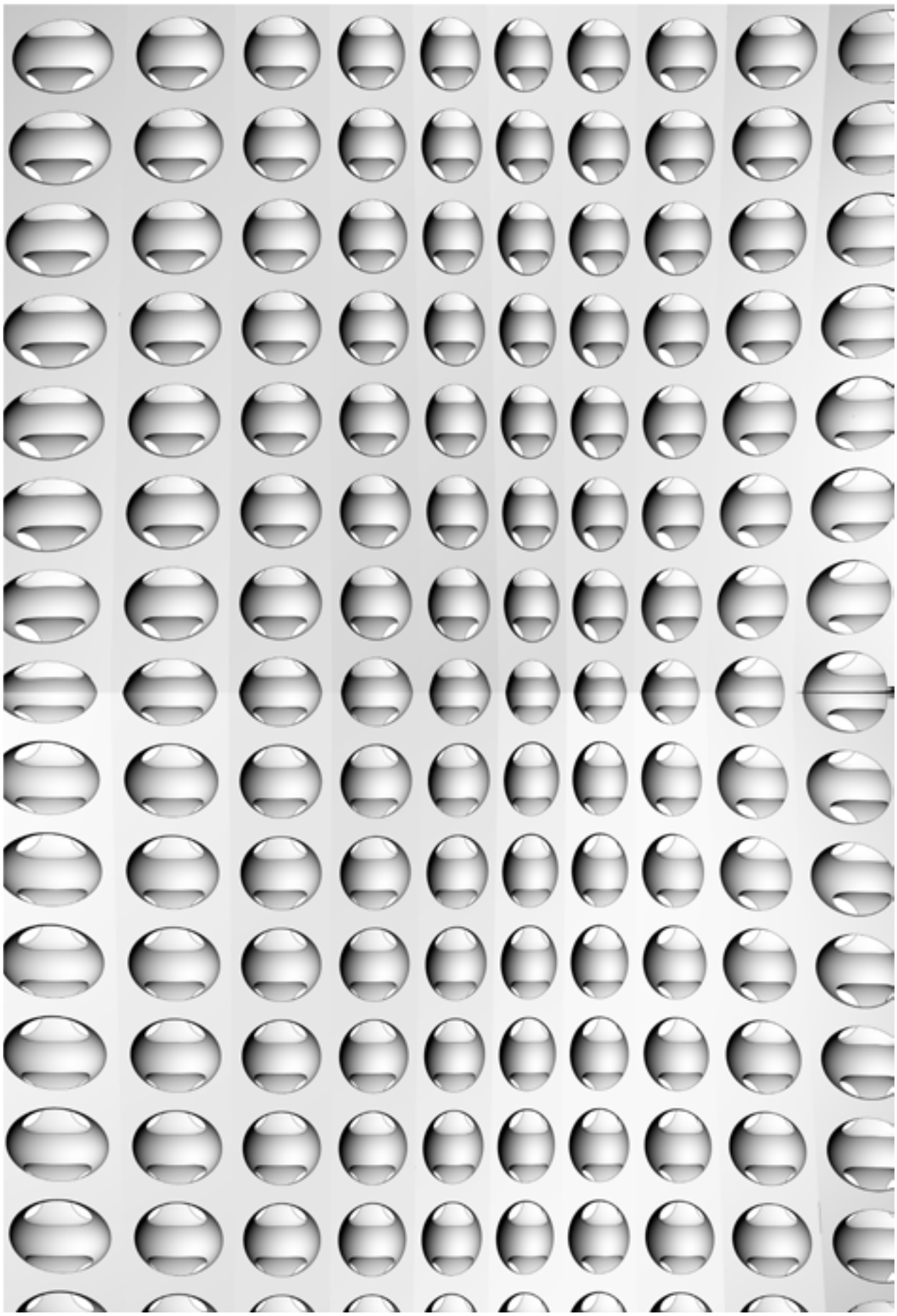
> You should see a grid of twisted boxes populating your referenced surface. Change the U and V count sliders to change the number of boxes, and use the height slider to adjust their height.

16.	Connect the third **Number Slider** to the Height (H) input of the **Surface Box** component	
17.	**Surface/Primitive/Bounding Box** – Drag a **Bounding Box** component to the canvas	
18.	**Transform/Morph/Box Morph** – Drag and drop the **Box Morph** component onto the canvas	
19.	Connect the output of the **Geometry** parameter to the Content (C) input of the **Bounding Box** component	

20.	Connect the output of the **Geometry** parameter to the Geometry (G) input of the **Box Morph** component	
21.	Connect the Box (B) output of the **Bounding Box** component to the Reference (R) input of the **Box Morph** component	
22.	Connect the Twisted Box (B) output of the **Surface Box** component to the Target (T) input of the **Box Morph** component	



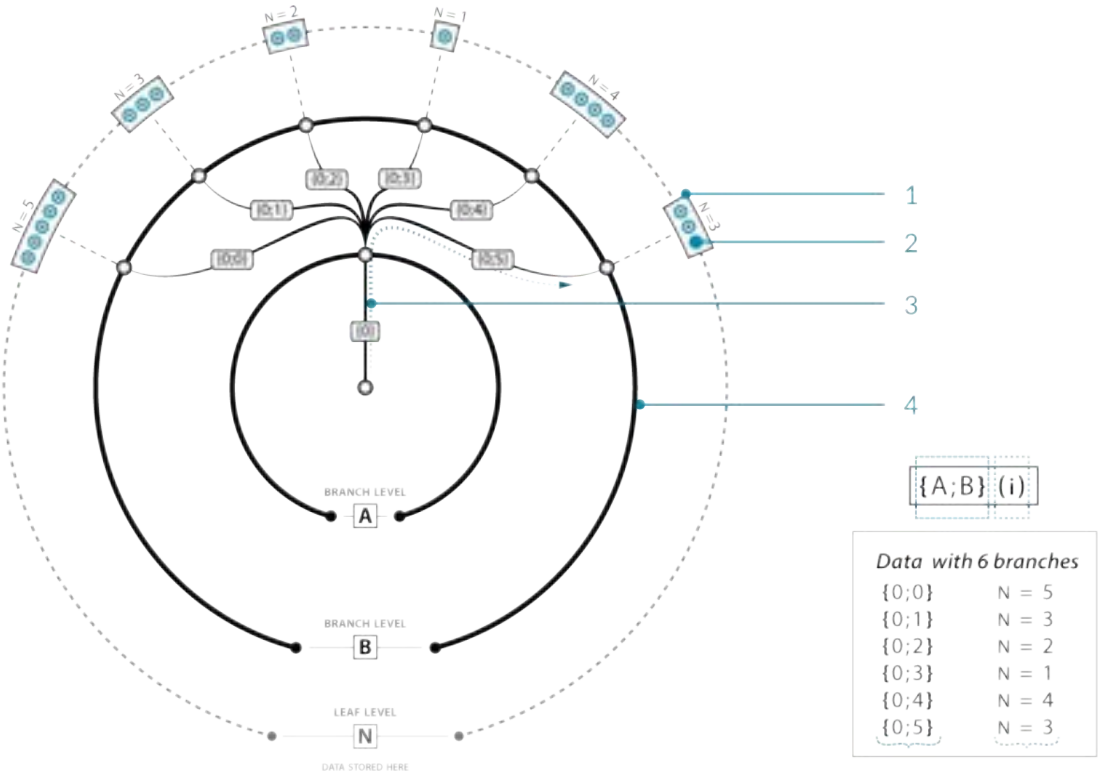
> You should now see your geometry populating your surface.



1.5.2. What is a Data Tree?

A Data Tree is a hierarchical structure for storing data in nested lists. Data trees are created when a grasshopper component is structured to take in a data set and output multiple sets of data. Grasshopper handles this new data by nesting it in the form of sub-lists. These nested sub-lists work in the same way as folder structures on your computer in that accessing indexed items require moving through paths that are informed by their generation of parent lists and their own sub-index.

It's possible to have multiple lists of data inside a single parameter. Since multiple lists are available, there needs to be a way to identify each individual list. A Data Tree is essentially a list of lists, or sometimes a list of lists of lists (and so on).



In the image above, there is a single master branch (you could call this a trunk, but since it's possible to have multiple master branches, it might be a bit of a misnomer) at path {0}. This path contains no data, but does have 6 sub-branches. Each of these sub-branches inherit the index of the parent branch {0} and add their own sub-index (0, 1, 2, 3, 4, and 5 respectively). It would be wrong to call this an "index", because that implies just a single number. It is probably better to refer to this as a "path", since it resembles a folder-structure on the disk. At each of these sub-branches, we encounter some data. Each data item is thus part of one (and only one) branch in the tree, and each item has an index that specifies its location within the branch. Each branch has a path that specifies its location within the tree.

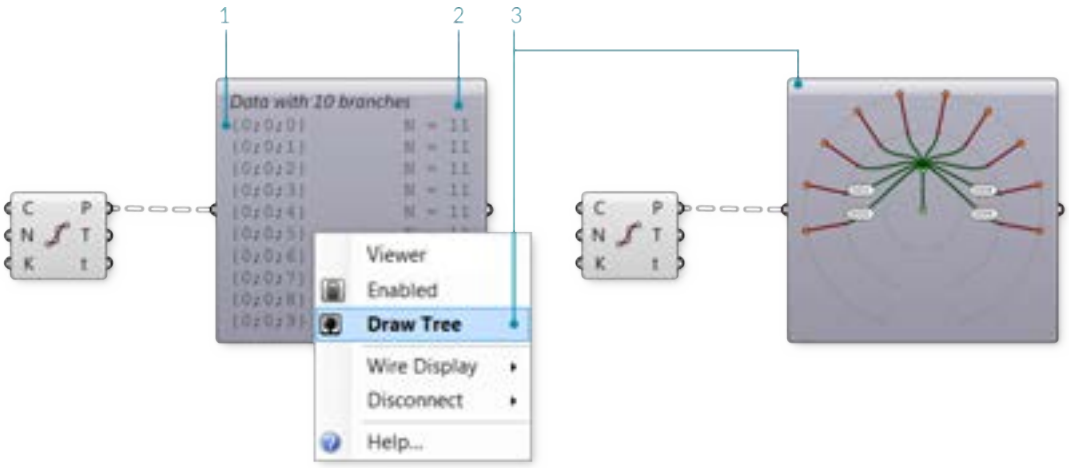
The image below illustrates the difference between a list and a data tree. On the left, an array of four columns of six points each is all contained in one list. The first column numbered 0-5, the second 6-11, and so on. On the right is the same array of points contained in a data tree. The data tree is a list of four columns, and each column is a list of six points. The index of each point is (column number, row number). This is a much more useful way of organizing this data, because you can easily access and operate on all the points in a given row or column, delete every second row of points, connect alternating points, etc.

1.5.2.1. DATA TREE VISUALIZATION

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

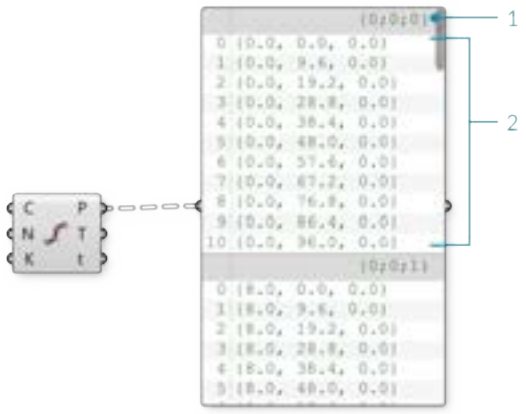
Due to their complexity, Data Trees can be difficult to understand. Grasshopper has several tools to help visualize and understand the data stored in a tree.

The Param Viewer The Param Viewer (Params/Util/Param Viewer) allows you to visualize data in text form and as a tree. Connect any output containing data to the input of the Param Viewer. To show the tree, right-click the Param Viewer and select “draw tree.” In this example, the Param Viewer is connected to the Points (P) output of a Divide Curve component that divided 10 curves into 10 segements each. The ten branches correspond to the ten curves, each containing a list of 11 points which are the division points of the curve.



> 1. Path of each list
> 2. Number of items in each list
> 3. Select "Draw Tree" to display the data tree

If we connect a panel to the same output, it displays ten lists of 11 items each. You can see that each item is a point defined by three coordinates. The path is displayed at the top of each list, and corresponds to the paths listed in the Param Viewer.

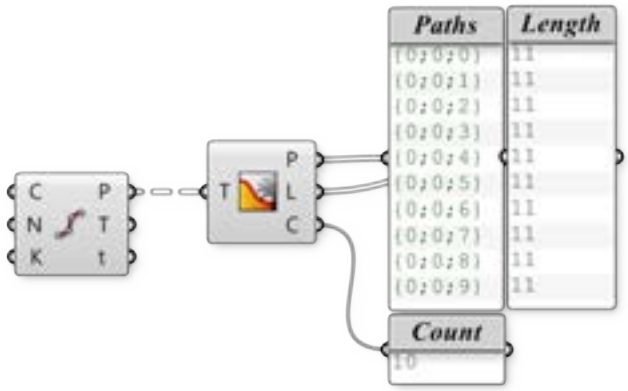


1. Path
2. List of 11 items

Tree Statistics The Tree Statistics component (Sets/Tree/Tree Statistics) Returns some statistics of the Data Tree including:

- P - All the paths of the tree
- L - The length of each branch in the tree
- C - Number of paths and branches in the tree

If we connect the Points output of the same Divide Curve component, we can display the paths, lengths, and the count in panels. This component is helpful because it separates the statistics into three outputs, allowing you to view only the one that is relevant.



Both the Param Viewer and the Tree Statistics component are helpful for visualizing changes in the structure of the Data Tree. In the next section, we will look at some operations that can be performed to change this structure.

1.5.3. Creating Data Trees

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Grasshopper contains tools for changing the structure of a data tree. Theese tools can help you access specific data within a tree, and change the way it is stored, ordered, and identified.

Let's look at some data tree manipulations and visualize how they affect the tree.

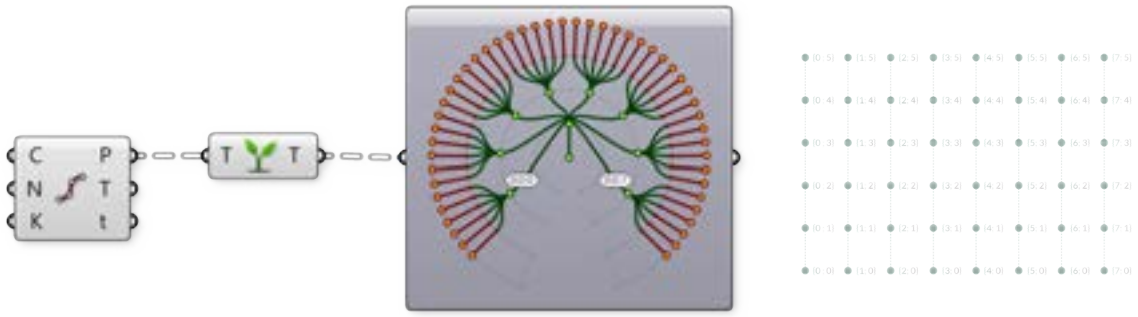
1.5.3.1. FLATTEN

Flattening removes all levels of a Data Tree, resulting in a single List. Using the Flatten component (Sets/Tree/Flatten) on the P output of our Divide Curve component, we can use the Param Viewer to visualize the new data structure.

> In the Param Viewer, we can see that we now only have 1 branch containing a list of 48 points.

1.5.3.2. GRAFT TREE

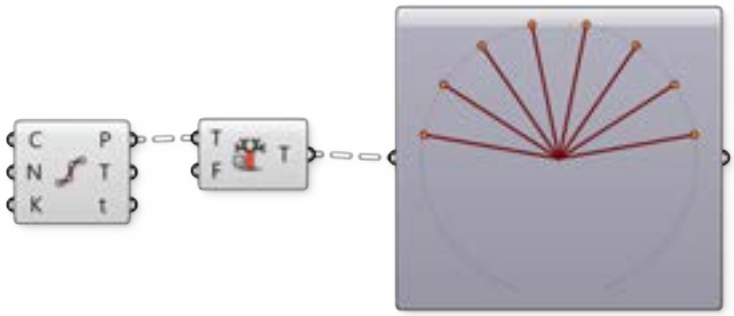
Grafting creates a new Branch for every Data Item. If we run the data through the Graft Tree component (Sets/Tree/Graft Tree), each division point now has its own individual branch, rather than sharing a branch with the other division points on the same curve.



> In the Param Viewer, we can see that what was data with 8 branches of 6 items each, we now have 8 branches with 6 sub-branches containing 1 item each.

1.5.3.3. SIMPLIFY TREE

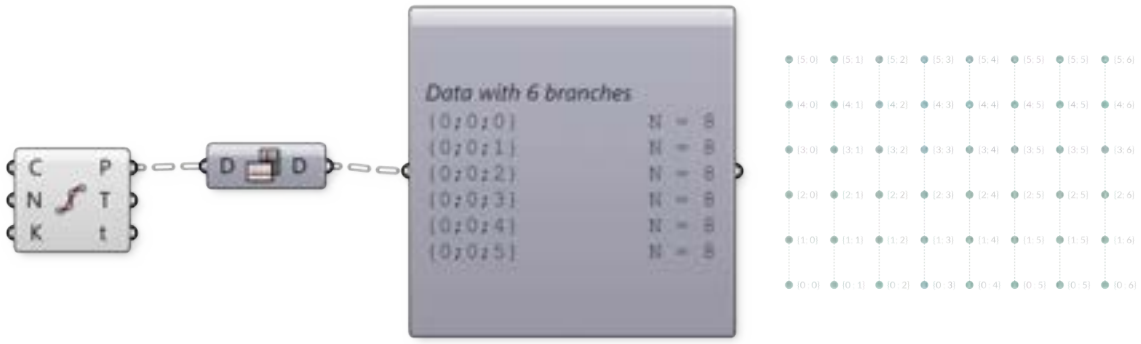
Simplify removes overlapping Branches in a Data Tree. If we run the data through the Simplify Tree component (Sets/Tree/Simplify Tree), the first branch, containing no data, has been removed.



> In the Param Viewer, we still have 8 branches of 6 items each, but the first branch has been removed.

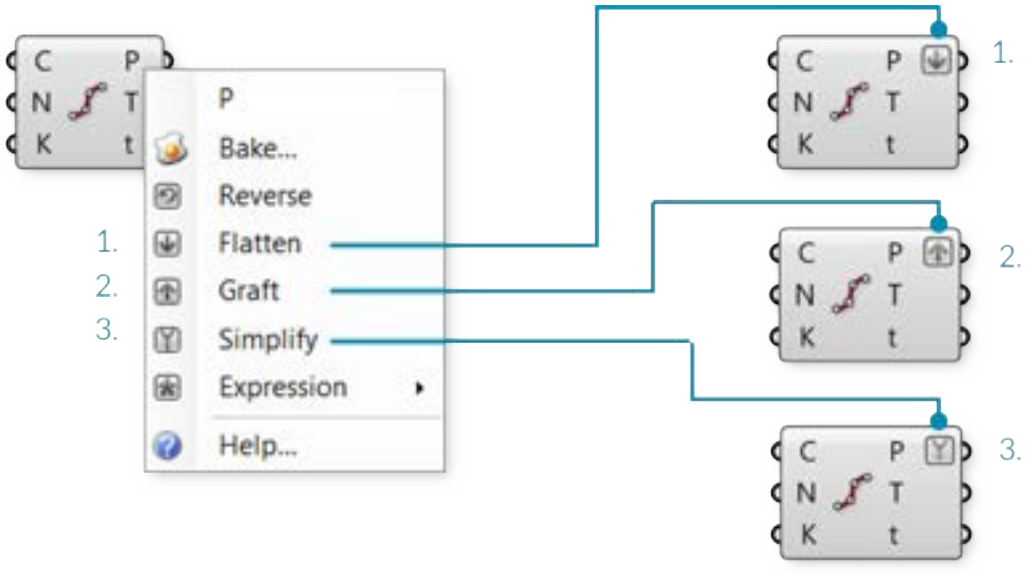
1.5.3.4. FLIP MATRIX

The Flip Matrix component (Sets/Tree/Flip Matrix) Swaps the “Rows” and “Columns” of a Data Tree with two Path Indices.



> In the Param Viewer, we can see that what was data with 8 branches of 6 items each, we now have 6 branches with 8 items each.

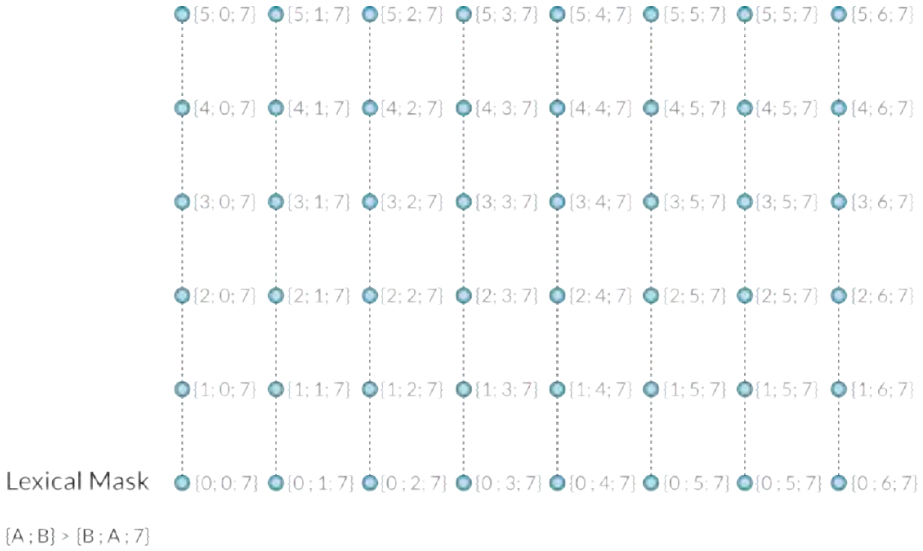
The Flatten, Graft, and Simplify operations can be applied to the component input or output itself, rather than feeding the data through a separate component. Just right-click the desired input or output and select Flatten, Graft, or Simplify from the menu. The component will display an icon to indicate that the tree is being modified. Keep in mind Grasshopper’s program flow. If you flatten a component input, the data will be flattened before the component operation is performed. If you flatten a component output, the data will be flattened after the component performs its action.

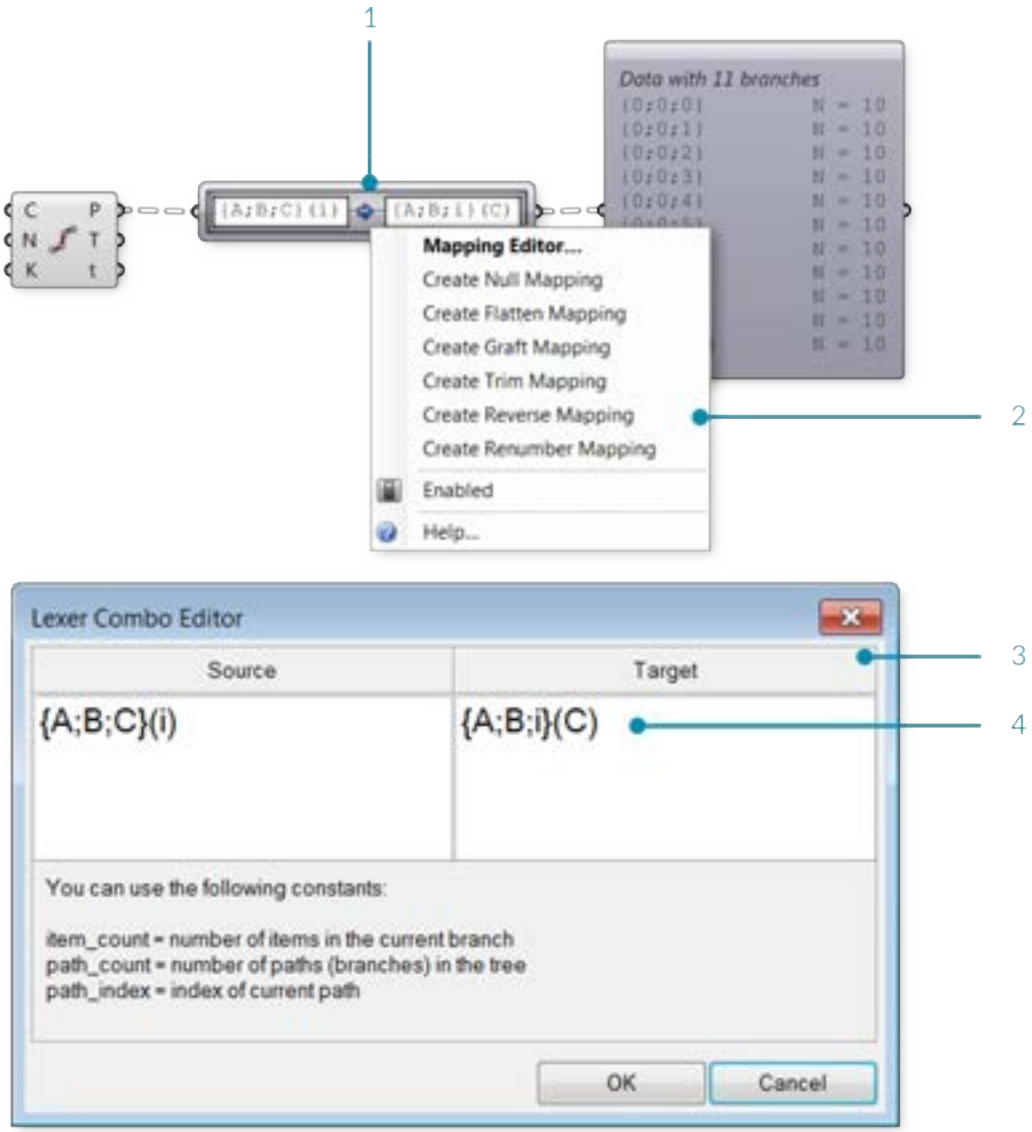


1. Flattened output P
2. Grafted output P
3. Simplified output P

1.5.3.5. THE PATH MAPPER

The Path Mapper component (Sets/Tree/Path Mapper) allows you to perform lexical operations on data trees. Lexical operations are logical mappings between data paths and indices which are defined by textual (lexical) masks and patterns.



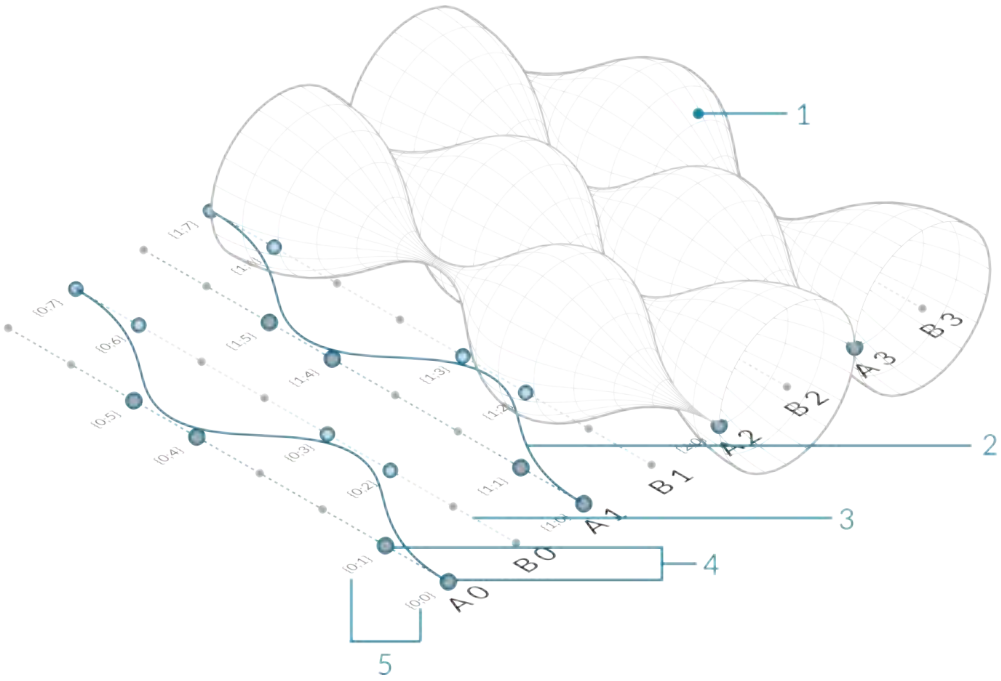


1. The Path Mapper component
2. Right-click the Path Mapper component and select a predefined mapping option from the menu, or open the mapping editor
3. The Mapping Editor
4. You can modify a data tree by re-mapping the path index and the desired branch

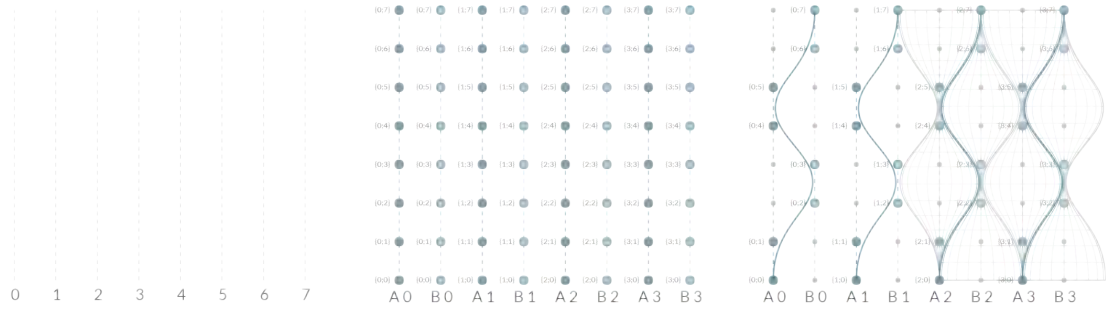
1.5.3.6. WEAVING DEFINITION

Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

In this example, we will manipulate lists and data trees to weave lists of points, define a pattern, and create surface geometry.



1. Revolved NURBS surface
2. NURBS curve
3. Curve array
4. Division points
5. Paths (indices) of points

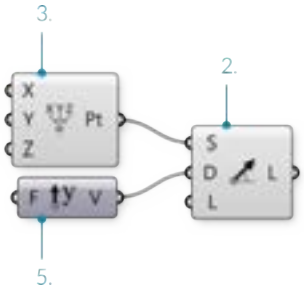


1. 2. 3.

1. Array curves
2. Dispatch curves into lists Aand B, divide curves
3. Cull poits, weave, and revolve

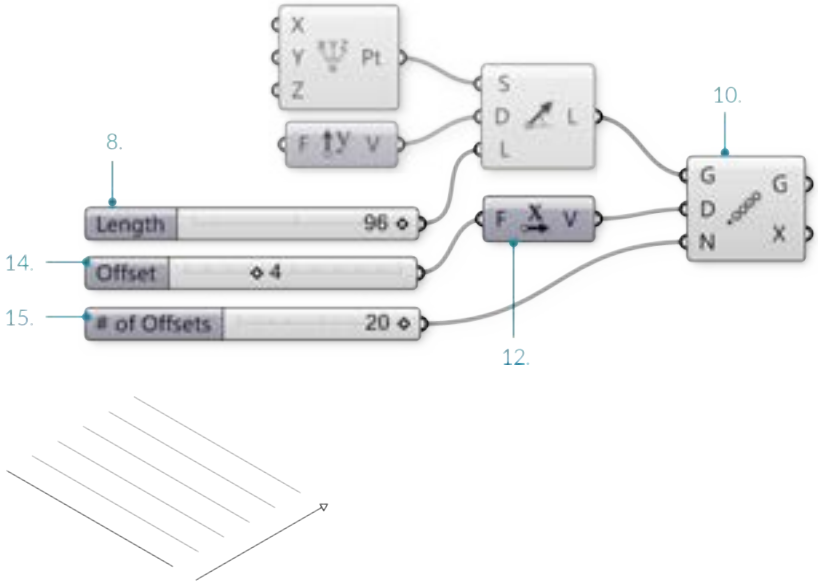
01.	Start a new definition, type Ctrl+N (in Grasshopper)	
02.	**Curve/Primitive/Line SDL** – Drag and drop the **Line SDL** component onto the canvas	
03.	**Vector/Point/Construct Point** – Drag and drop the **Construct Point** component onto the canvas	
04.	Connect the Point (Pt) output of the **Construct Point** component to the Start (S) Input of the **Line SDL** component	
05.	**Vector/Vector/Unit Y** – Drag and drop the vector **Unit Y** component onto the canvas The factor of Unit Vector components is 1.0 by default.	

05.	The factor of Unit Vector components is 1.0 by default.	
06.	Connect the **Unit Y** component to the Direction (D) input of the **Line SDL** component	



07.	**Params/Input/Number Slider** – Drag and drop the **Number Slider** component onto the canvas	
08.	Double-click on the **Number Slider** and set the following: Name: Length Rounding: Integer Lower Limit: 0 Upper Limit: 96 Value: 96	
09.	Connect the **Number Slider** to the Length (L) input of the **Line SDL** component	
10.	**Transform/Array/Linear Array** – Drag and drop the **Linear Array** component onto the canvas	
11.	Connect the Line (L) output of the **Line SDL** component to the Geometry (G) input of the **Linear Array** component	
12.	**Vector/Vector/Unit X** – Drag and drop the vector **Unit X** component onto the canvas	
13.	**Params/Input/Number Slider** – Drag and drop two **Number Slider** components onto the canvas	
14.	Double-click on the first **Number Slider** and set the following: Name: Offset Distance Rounding: Integer Lower Limit: 1 Upper Limit: 10 Value: 4	
15.	Double-click on the second **Number Slider** and set the following: Name: # of Offsets Rounding: Even Lower Limit: 2 Upper Limit: 20 Value: 20	
16.	Connect the **Number Slider** (Offset Distance) to the Factor (F) input of the **Unit X** component	
17.	Connect the Vector (V) output of the **Unit X** component to the Direction (D) input of the **Linear Array** component	

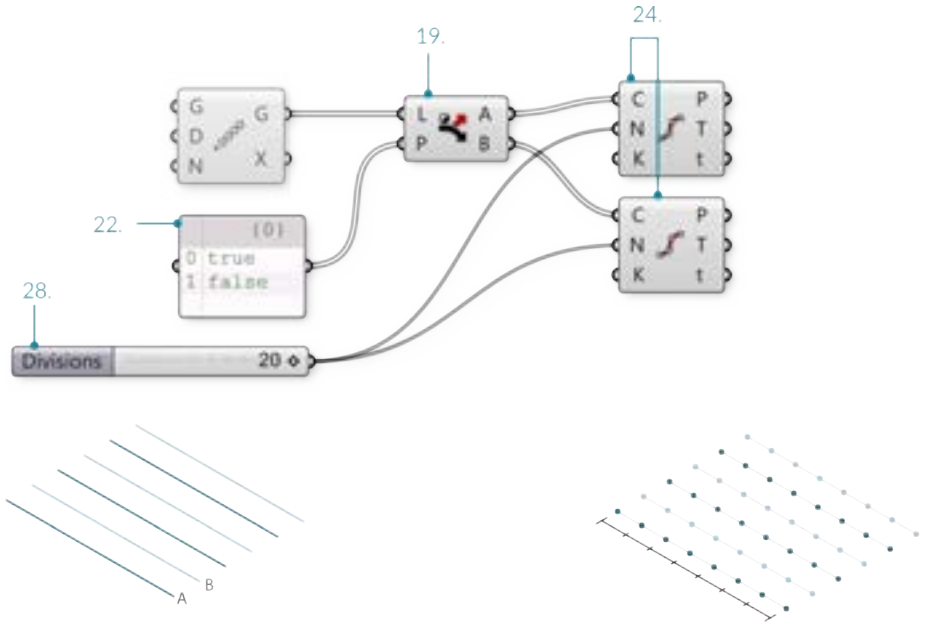
18.	Connect the **Number Slider** (# of Offsets) to the Count (N) input of the **Linear Array** component	



> You should now see an array of lines in the Rhino viewport. The three sliders allow you to change the length of the lines, their distance from each other, and the number of lines in the array.

19.	**Sets/Lists/Dispatch** – Drag and drop the **Dispatch** component onto the canvas	
20.	Connect the Geometry (G) output of the **Linear Array** component to the List (L) input of the **Dispatch** component	
21.	**Params/Input/Panel** – Drag and drop the **Panel** component onto the canvas	
22.	Double-click the **Panel**, deselect Multiline Data, Wrap Items and Special Codes, and enter the following: true false	
23.	Connect the **Panel** to the Pattern (P) input of the **Dispatch** component	
24.	**Curve/Division/Divide Curve** – Drag and drop two **Divide Curve** components onto the canvas	
25.	Connect the List A (A) output of the **Dispatch** component to the Curve (C) input of the first **Divide Curve** component	
26.	Connect the List B (B) output of the **Dispatch** component to the Curve (C) input of the second **Divide Curve** component	
27.	**Params/Input/Number Slider** – Drag and drop the **Number Slider** component onto the canvas	
28.	Double-click on the **Number Slider** and set the following: Name: Divisions Rounding: Integer Lower Limit: 0 Upper Limit: 20 Value: 20	

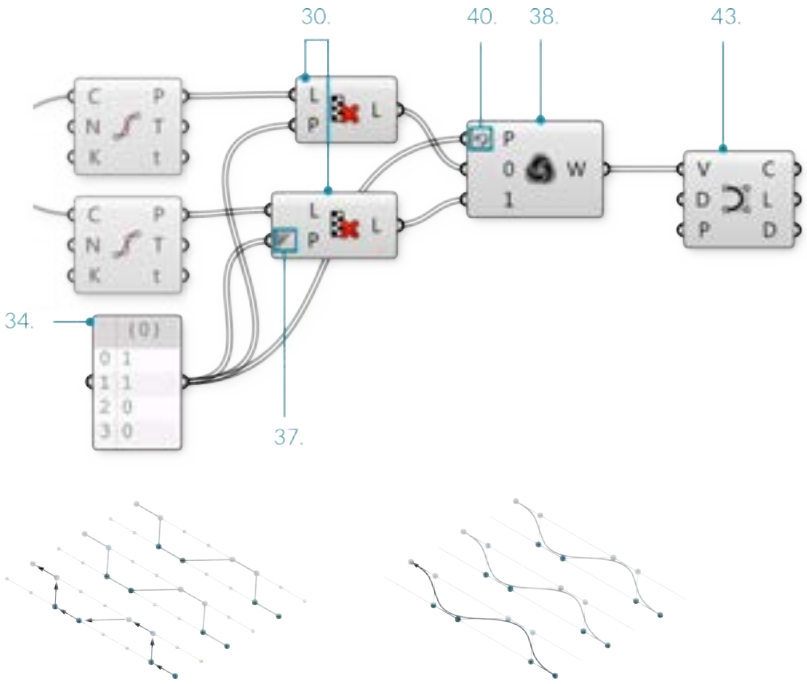
29.	Connect the **Number Slider** (Divisions) to the Count (N) input of both **Divide Curve** components.	



1. The Dispatch component sends every second curve in the array to a separate list.
2. The Divide Curve component divides the curves into the number of segments specified by the slider. Adjust the slider to change the number of points.

30.	**Sets/Sequence/Cull Pattern** – Drag and drop two **Cull Pattern** components onto the canvas	
31.	Connect the Points (P) output of the first **Divide Curve** component to the List (L) input of the first **Cull Pattern** component	
32.	Connect the Points (P) output of the second **Divide Curve** component to the List (L) input of the second **Cull Pattern** component	
33.	**Params/Input/Panel** – Drag and drop a second **Panel** component onto the canvas	
34.	Double-click the second **Panel** and deselect: Multiline Data, Wrap Items, and Special Codes. Then enter the following: 1 1 0 0 > We are using 1 and 0 in place of true and false. These are the two syntaxes that Grasshopper accepts for boolean values.	
35.	Connect the second **Panel** to the Pattern (P) input of the first **Cull Pattern** component	
36.	Connect the second **Panel** to the Pattern (P) input of the second **ull Pattern** component	
37.	Right-click on the Pattern (P) input of the second **Cull Pattern** component and select Invert > This will invert the **Cull Pattern**, a useful trick to keep definitions short.	

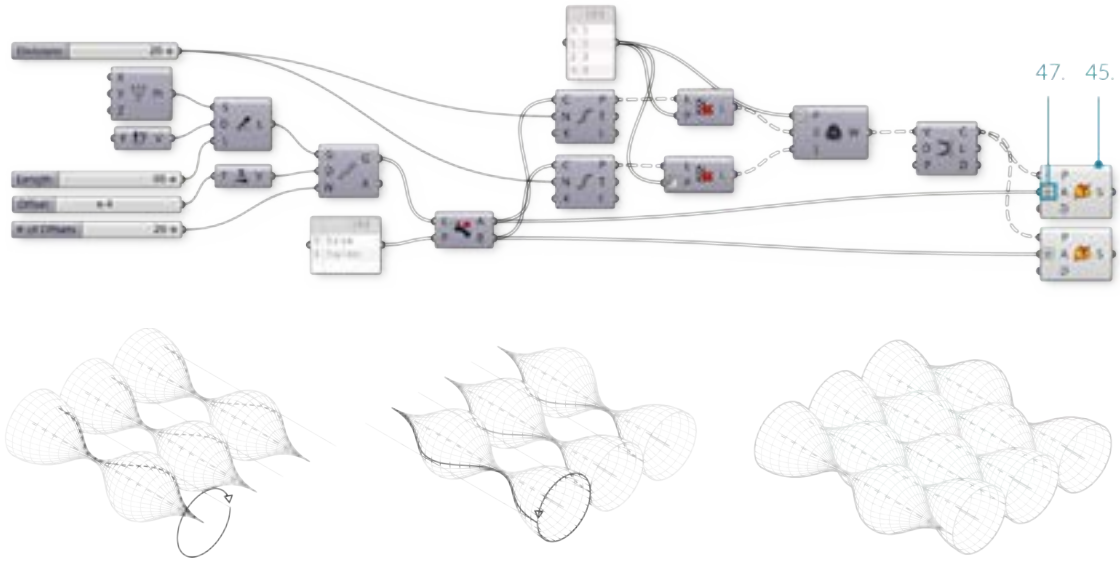
38.	**Sets/List/Weave** – Drag and drop the **Weave** component onto the canvas	
39.	Connect the second **Panel** to the Pattern (P) input of the **Weave** component	
40.	Right-click the Pattern (P) input of the **Weave** component and select reverse	
41.	Connect the List (L) output of the first **Cull Pattern** component to the Stream 0 (0) input of the **Weave** component	
42.	Connect the List (L) output of the second **Cull Pattern** component to the Stream 0 (0) input of the **Weave** component	
43.	**Curve/Spline/Nurbs Curve** – Drag and drop the **Nurbs Curve** component onto the canvas	
44.	Connect the Weave (W) output of the **Weave** component to the Vertices (V) input of the **Nurbs Curve** component.	

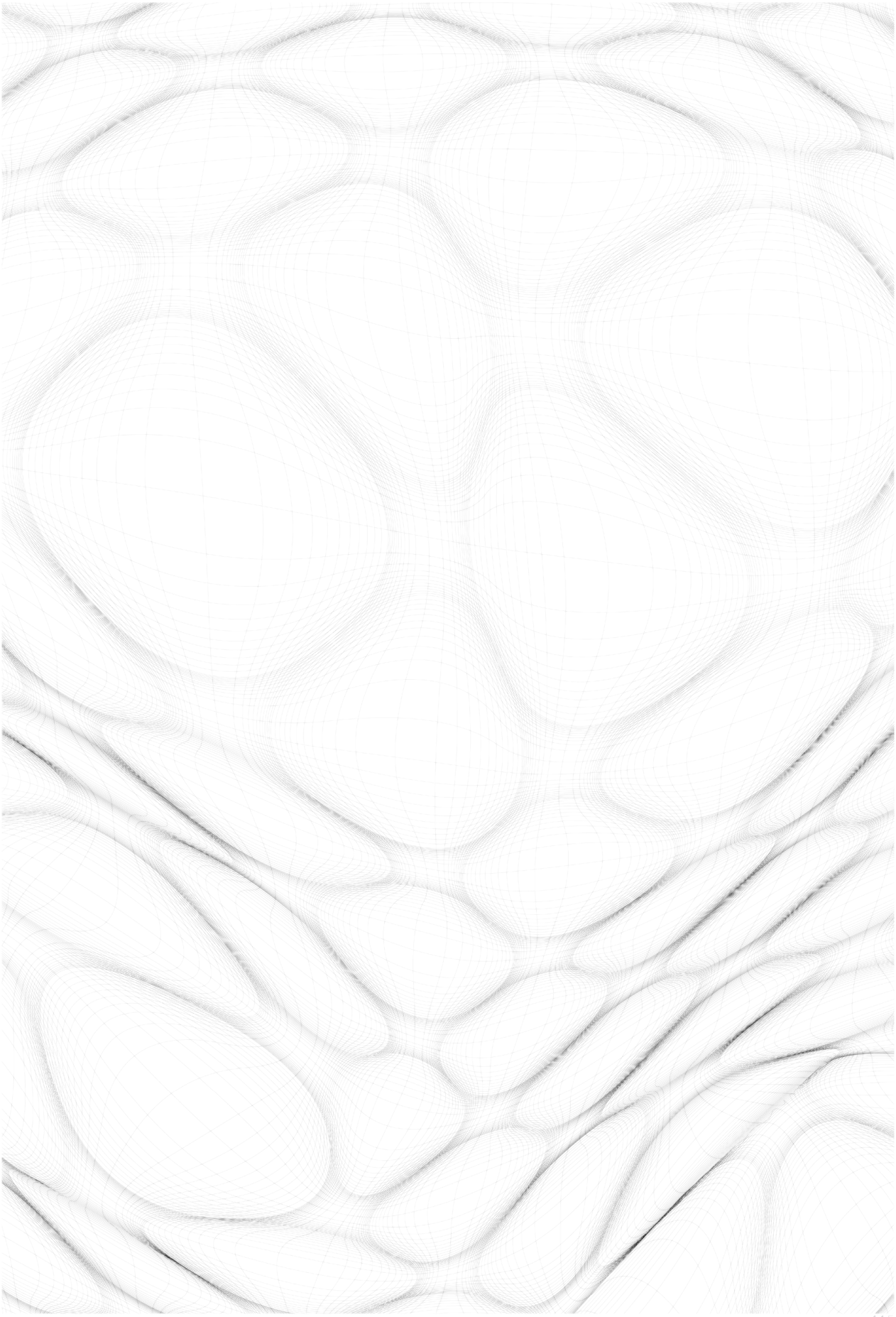


1. The cull patterns remove alternating points from each list.
2. The Weave component collects data from the point lists according to a custom pattern. This data is fed into the interpolate component to create curves.

45.	**Surface/Freeform/Revolution** – Drag and drop two **Revolution** components onto the canvas	
46.	Connect the Curve output of the **Nurbs Curve** component to the Profile Curve (P) input of both **Revolution** components.	
47.	Right Click on Axis (A) input of both **Revolution** components and select Graft.	
48.	Connect the List A(A) output of the **Dispatch** component to the Axis (A) input of the first **Revolution** component	
	Connect the List B (B) output of the **Dispatch** component to the Axis (A) input of the	

49.	second **Revolution** component Select all the components except the two Revolution components and turn the preview off - it is helpful to turn previews off as you build the definition to focus on the most recent geometry	

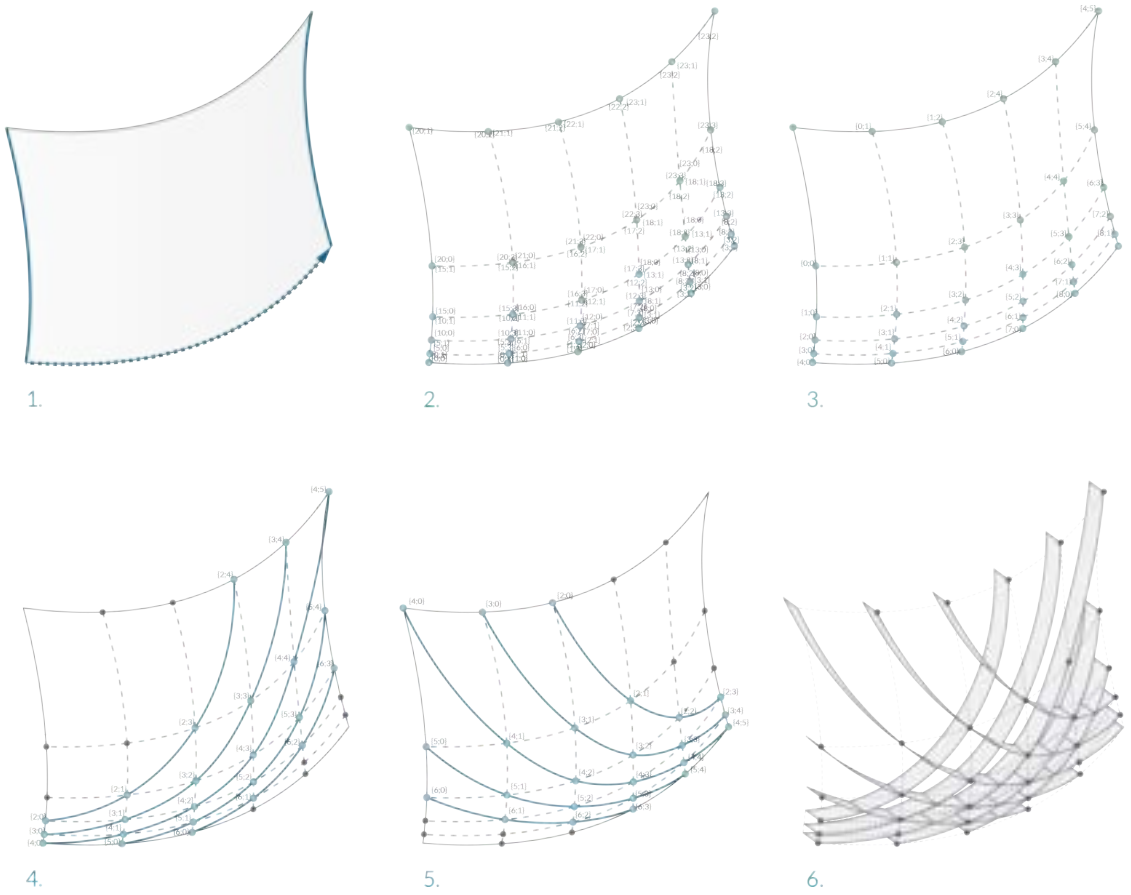




1.5.4. Working with Data Trees

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

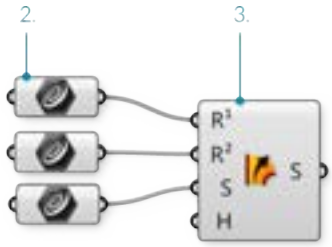
In this example, we will use some of Grasshopper's tools for manipulating data trees to retreive, reorganize, and interpolate the desired points contained in a data tree and create a lattice of intersecting fins.



1. Sweep with two rails to create a NURBS surface.
2. Divide the surface into variable sized segments, extract vertices. Data comprised of one list with four items in each segment.
3. Flip the matrix to change the data structure. Data comprised of four lists, each containing a single corner point of each segment.
4. Explode the tree to connect corner points and draw diagonal lines across each segement.
5. Prune the tree to cull branches containing insufficient points to construct a degree 3 NURBS curve and interpolate points.
6. Extrude the curves to create intersecting fins.

01.	Start a new definition, type Ctrl+N (in Grasshopper)	
02.	**Params/Geometry/Curve** – Drag and drop three **curve** parameters onto the canvas	
03.	**Surface/Freeform/Sweep2** – Drag a **Sweep2** component onto the canvas	
	Right-click the first **Curve** parameter and select "Set one curve." Select the first rail	

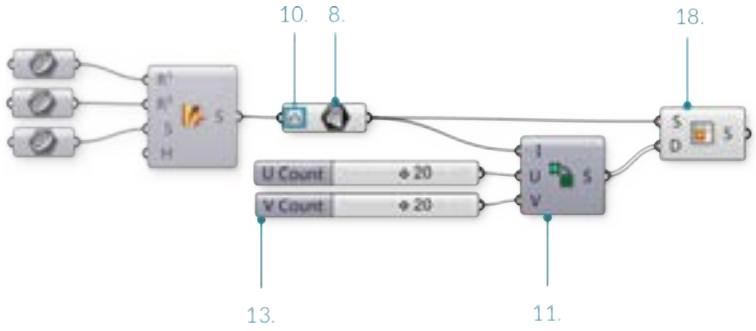
	curve in the Rhino viewport	
05.	Right-click the second **Curve** parameter and select “Set one curve.” Select the second rail curve in the Rhino viewport	
06.	Right-click the third **Curve** parameter and select “Set one curve.” Select the section curve in the Rhino viewport	
07.	Connect the outputs of the **Curve** parameters to the Rail 1 (R1), Rail 2 (R2), and Sections (S) inputs of the **Sweep2** respectively	



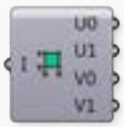
> We have just created a NURBS surface

08.	**Params/Geometry/Surface** – drag a **Surface** parameter to the canvas	
09.	Connect the Brep (S) output of the **Sweep2** component to the input of the **Surface** parameter	
10.	Right-click the **Surface** parameter and select “Reparameterize”. > In this step, we re-mapped the u and v domains of the surface between 0 and 1. This will make future operations possible.	
11.	**Maths/Domain/Divide Domain2** – drag and drop a **Divide Domain2** component onto the canvas	
12.	**Params/Input/Number Slider** – drag two **Number Sliders** onto the canvas	
13.	Double click the first **Number Sliders** and set the following: Rounding: Integer Lower Limit: 1 Upper Limit: 40 Value: 20	
14.	Set the same values on the second **Number Sliders**	
15.	Connect the output of the reparameterized **Surface** parameter to the Domain (I) input of the **Divide Domain2** component	
16.	Connect the first **Number Sliders** to the U Count (U) input of the **Divide Domain2** component	
17.	Connect the second **Number Sliders** to the V Count (V) input of the **Divide Domain2** component	
18.	**Surface/Util/Isotrim** – Drag and drop the **Isotrim** component onto the canvas	
19.	Connect the Segments (S) output of the **Divide Domain2** component to the Domain (D) input of the **Isotrim** component	

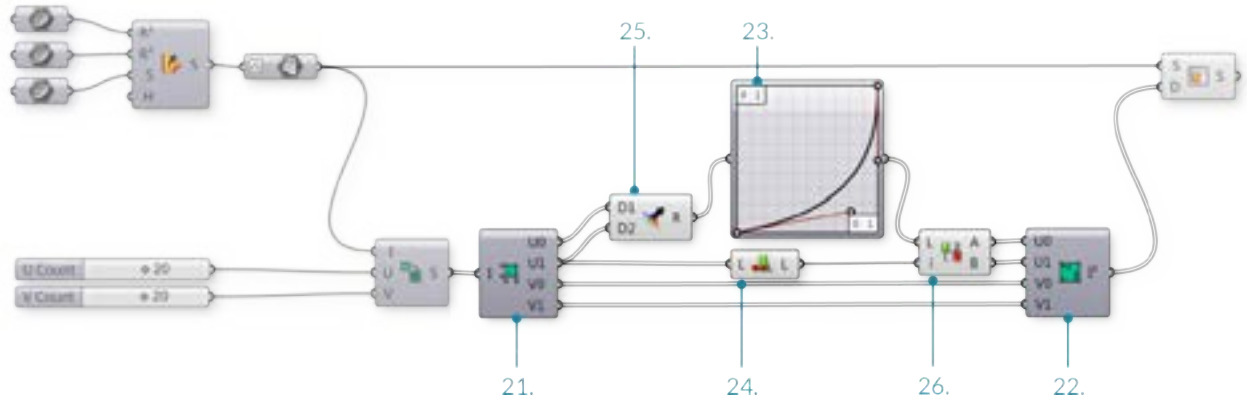
20.	Connect the output of the **Surface** parameter to the Surface (S) input of the **Isotrim** component	



> We have now divided out surface into smaller, equally sized, surfaces. Adjust the U and V Count sliders to change the number of divisions. Lets add a Graph Mapper to give the segments variable size.

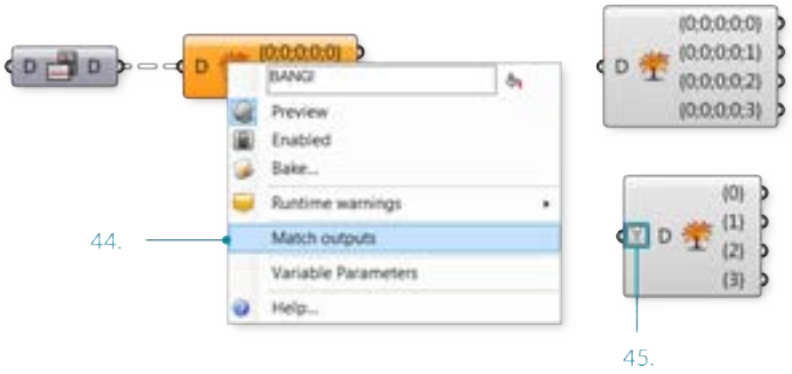
21.	**Maths/Domain/Deconstruct Domain2** – Drag a **Deconstruct Domain2** component onto the canvas	
22.	**Maths/Domain/Construct Domain2** – Drag a **Construct Domain2** component to the canvas	
23.	**Params/Input/Graph Mapper** – Drag a **Graph Mapper** to the canvas	
24.	**Sets/List/List Length** – Drag a **List Length** component to the canvas	
25.	**Sets/Tree/Merge** – Drag a **Merge** component to the canvas	
26.	**Sets/List/Split List** – Drag a **Split List** component to the canvas > The Merge and Split components are used here so that the same Graph Mapper could be used for both the U min and U max values.	
27.	Connect the U min (U0) and U max (U1) outputs of the **Deconstruct Domain2** component to the Data 1 (D1) and Data 2 (D2) inputs of the **Merge** component	
28.	Connect the Result (R) output of the **Merge** component to the input of the **Graph Mapper**	
29.	Right-click the **Graph Mapper** and select “Bezier” under “Graph Types”	
30.	Connect a second wire from the U max (U1) output of the **Deconstruct Domain2** component to the List (L) input of the **List Length** component	
31.	Connect the **Graph Mapper** output to the List (L) input of the **Split List**	
32.	Connect the Length (L) output of the **List Length** component to the Index (i) input of the **Split List** component	
33.	Connect the List A (A) output of the **Split List** component to the U min (U0) input of the **Construct Domain2** component	

34.	Connect the List B (B) output of the **Split List** component to the U max (U1) input of the **Construct Domain2** component	
35.	Connect the V min (V0) output of the **Deconstruct Domain2** component to the V min (V1) input of the **Construct Domain2** component	
36.	Connect the V max (V1) output of the **Deconstruct Domain2** component to the V max (V1) input of the **Construct Domain2** component	
37.	Connect the 2D Domain (I2) output of the **Construct Domain2** component to the Domain (D) input of the **Isotrim** component, replacing the existing connection	



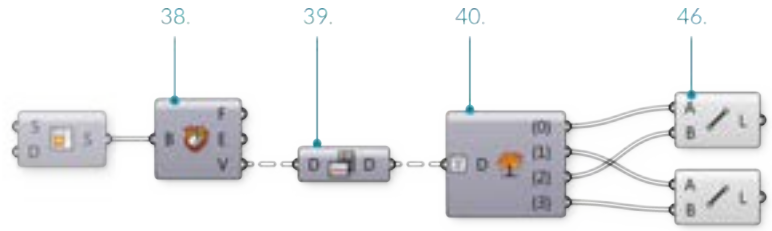
> We have just deconstructed the domains of each surface segment, remapped the U values using a Graph Mapper, and reconstructed the domains. Adjust the grips of the Graph Mapper to change the distribution of the surface segments. Let's use Data Trees to manipulate the surface divisions.

38.	**Surface/Analysis/Deconstruct Brep** – Drag the **Deconstruct Brep** component onto the canvas	
39.	**Sets/Tree/Flip Matrix** – Drag the **Flip Matrix** component to the canvas	
40.	**Sets/Tree/Explode Tree** – Drag the **Explode Tree** component to the canvas	
41.	Connect the Surface (S) output of the **Isotrim** component to the Brep (B) input of the **Deconstruct Brep** component > The Deconstruct Brep component deconstructs a Brep into Faces, Edges, and Vertices. This is helpful if you want to operate on a specific constituent of the surface.	
42.	Connect the Vertices (V) output of the **Deconstruct Brep** component to the Data (D) input of the **Flip Matrix** component > We just changed the Data tree structure from one list of four vertices that define each surface, to four lists, each containing one vertex of each surface.	
43.	Connect the Data (D) output of the **Flip Matrix** component to the Data (D) input of the **Explode Tree** component	
44.	Right-click the **Explode Tree** component and select "Match Outputs"	
45.	Right-click the Data (D) input of the **Explode Tree** component and select simplify	



> Each output of the Explode Tree component contains a list of one vertex of each surface. In other words, one list with all the top right corners, one list with all the bottom right corners, one list of top left corners, and one list of bottom left corners.

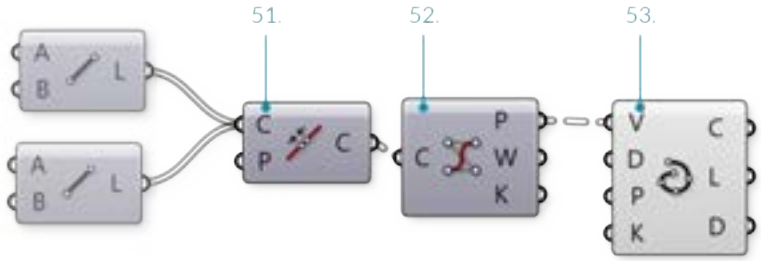
46.	**Curve/Primitive/Line** – Drag and drop two **Line** components onto the canvas	
47.	Connect the Branch 0 {0} output of the **Explode Tree** component to the Start Point (A) input of the first **Line** component	
48.	Connect the Branch 1 {1} output of the **Explode Tree** component to the Start Point (A) input of the second **Line** component	
49.	Connect the Branch 2 {2} output of the **Explode Tree** component to the End Point (B) input of the first **Line** component	
50.	Connect the Branch 3 {3} output of the **Explode Tree** component to the End Point (B) input of the second **Line** component	



> We have now connected the corner points of each surface diagonally with lines.

51.	**Curve/Util/Join Curves** – Drag and drop the **Join Curves** component to the canvas	
52.	**Curve/Analysis/Control Points** – Drag a **Control Points** component onto the canvas	
53.	**Curve/Spline/Interpolate** – Drag and drop the **Interpolate** component onto the canvas	
54.	Connect the Line (L) outputs of each **Line** component to the Curves (C) input of the **Join Curves** component Hold down the Shift key to connect multiple wires to a single input	

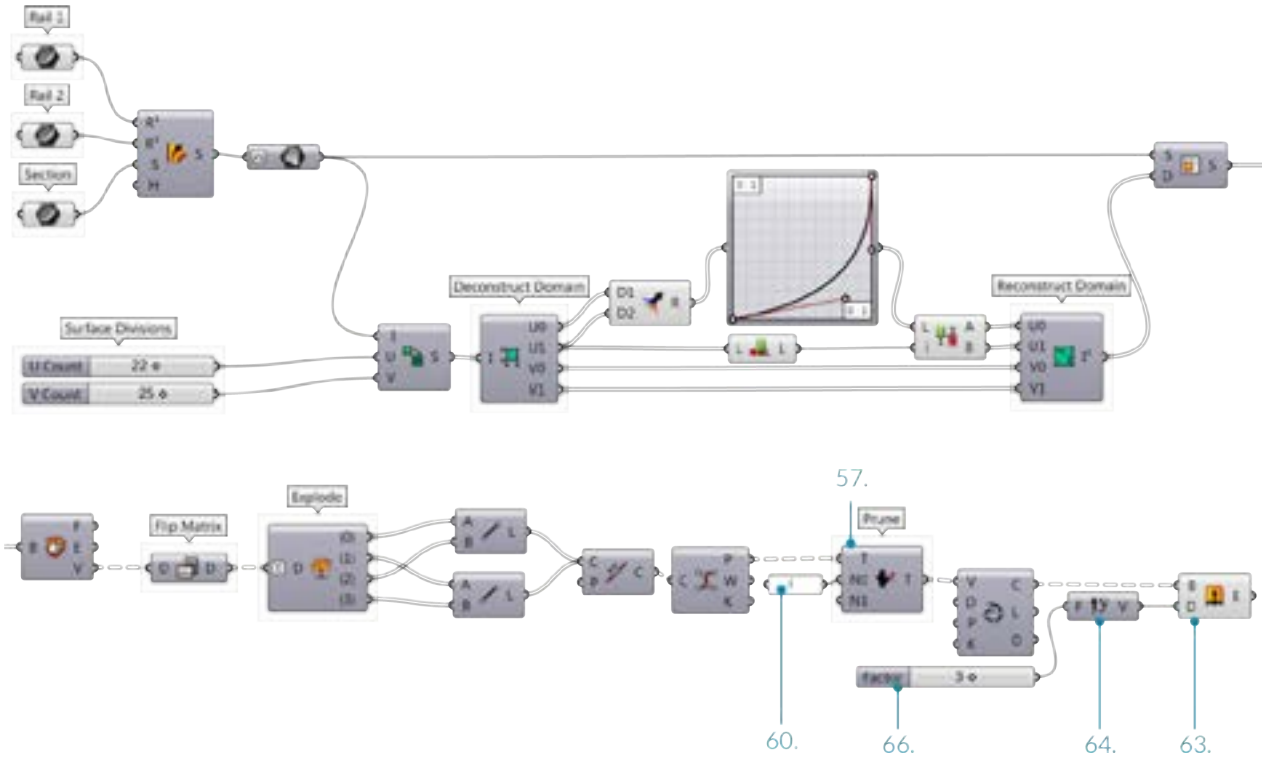
55.	Connect the Curves (C) output of the **Join Curves** component to the Curve (C) input of the **Control Points** component	
56.	Connect the Points (P) output of the **Control Points** component to the Vertices (V) input of the **Interpolate** component	



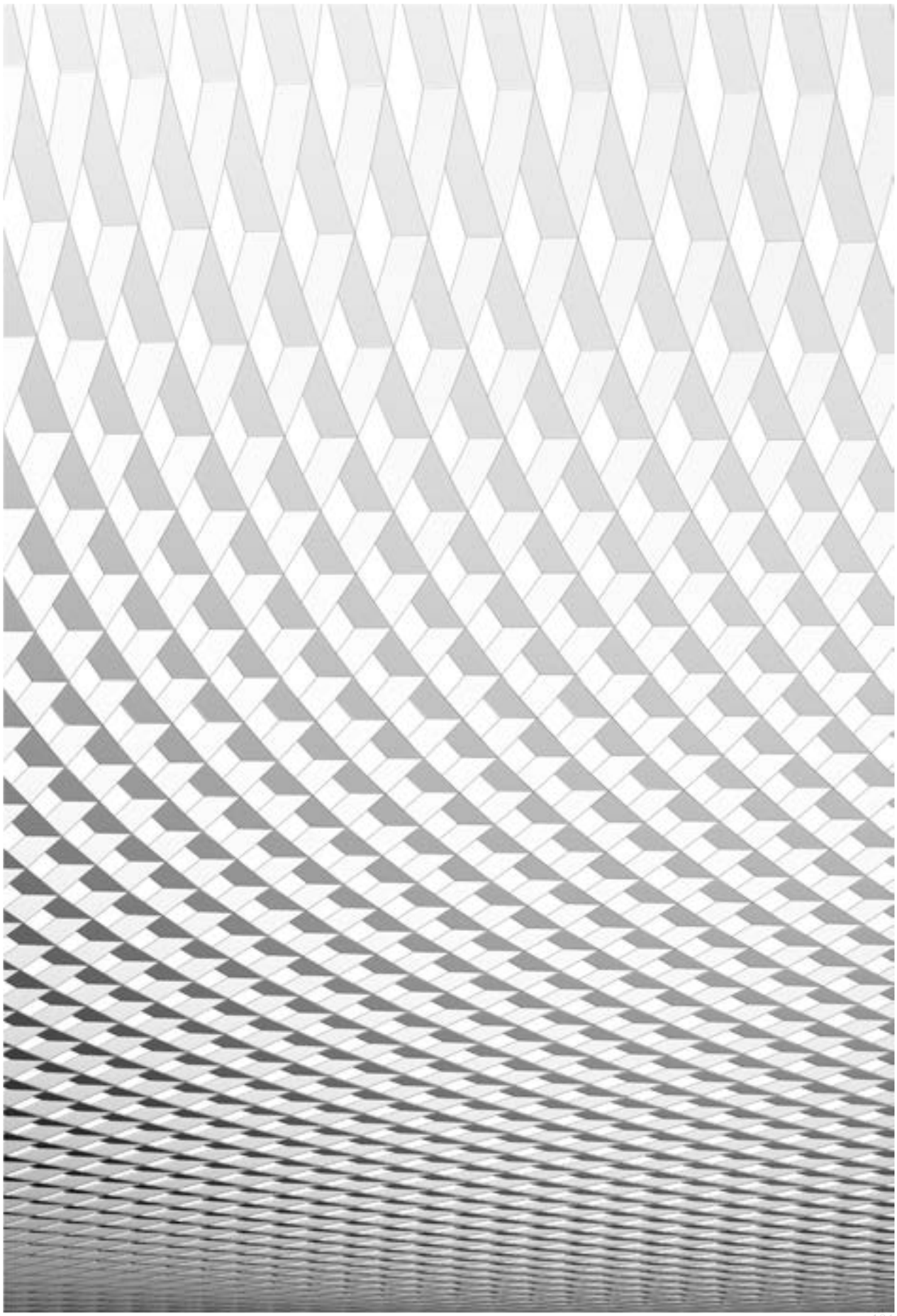
> We have now joined our lines into polylines and reconstructed them as NURBS curves by interpolating their control points. In the Rhino viewport, you might notice that the shorter curves are still straight lines. This is because you cannot make a degree three NURBS curve with fewer than four control points. Let's manipulate the data tree to eliminate lists of control points with less than four items.

57.	**Sets/Tree/Prune Tree** – Drag and drop the **Prune Tree** component onto the canvas	
58.	**Params/Input/Panel** – Drag a Panel onto the canvas	
59.	Connect the Points (P) output of the **Control Points** component to the Tree (T) input of the **Prune Tree** component > If you connect one Param Viewer to the Points (P) output of the Control Points component, and another to the Tree (T) output of the Prune Tree component, you can see that the number of branches has been reduced.	
60.	Double click the **Panel** and enter 4.	
61.	Connect the output of the **Panel** to the Minimum (N0) input of the **Prune Tree** component	
62.	Connect the Tree (T) output of the **Prune Tree** component to the Vertices (V) input of the **Interpolate** component	
63.	**Surface/Freeform/Extrude** – Drag and drop the **Extrude** component onto the canvas	
64.	**Vector/Vector/Unit Y** – Drag a **Unit Y** component onto the canvas *You may need to use a Unit X vector, depending on the orientation of your referenced geometry in Rhino*	
65.	**Params/Input/Number Slider** – Drag a **Number Slider** onto the canvas	
66.	Double click the **Number Slider** and set the following: Rounding: Integer Lower Limit: 1 Upper Limit: 5 Value: 3	

67.	Connect the Curve (C) output of the **Interpolate** component to the Base (B) input of the **Extrude** component	
68.	Connect the **Number Slider** output to the Factor (F) input of the **Unit Y** component	
69.	Connect the Unit Vector (V) output of the **Unit Y** component to the Direction (D) input of the **Extrude** component	

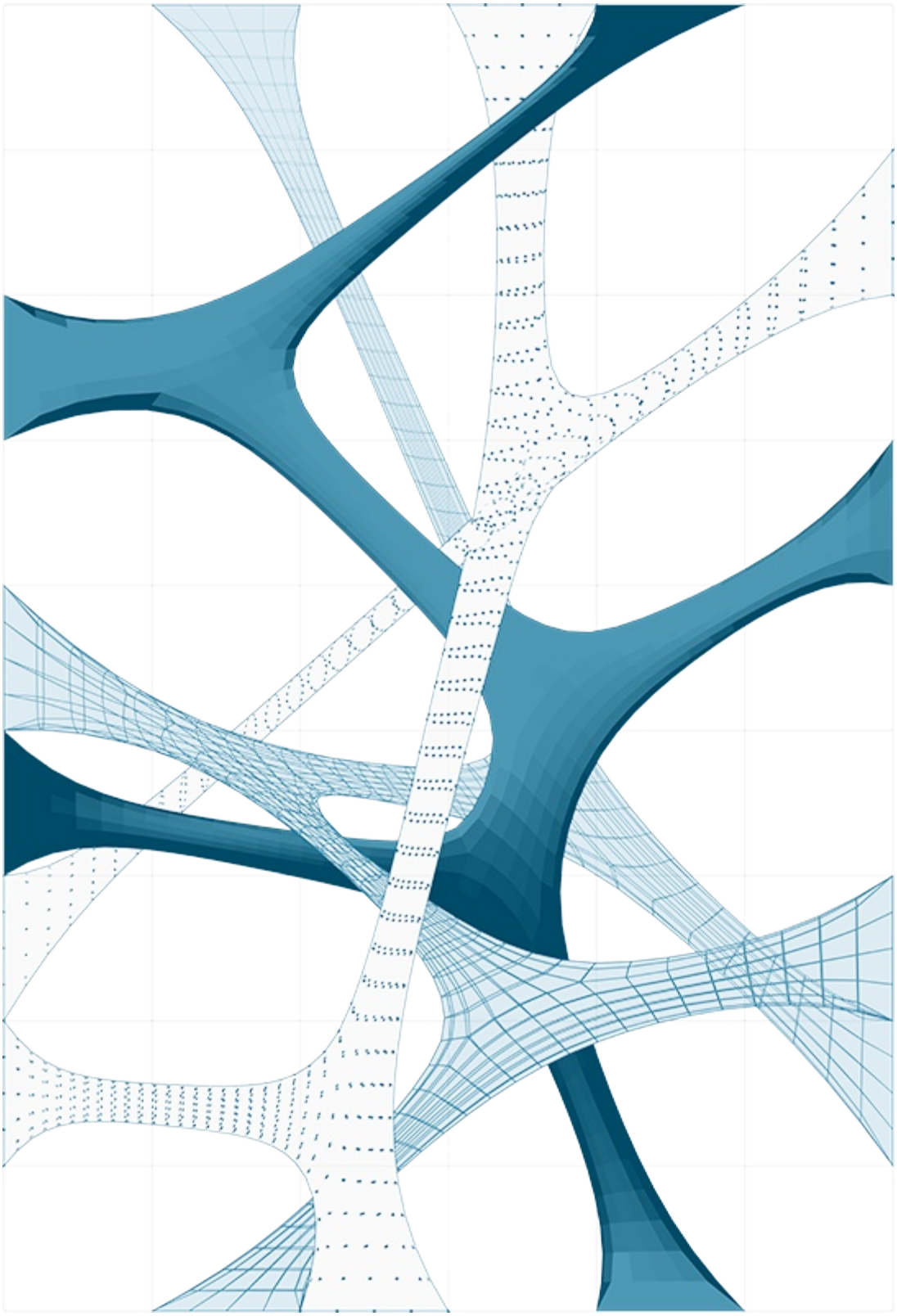


> You should now see a diagonal grid of strips or fins in the Rhino Viewport. Adjust the Factor slider to chnage the depth of the fins



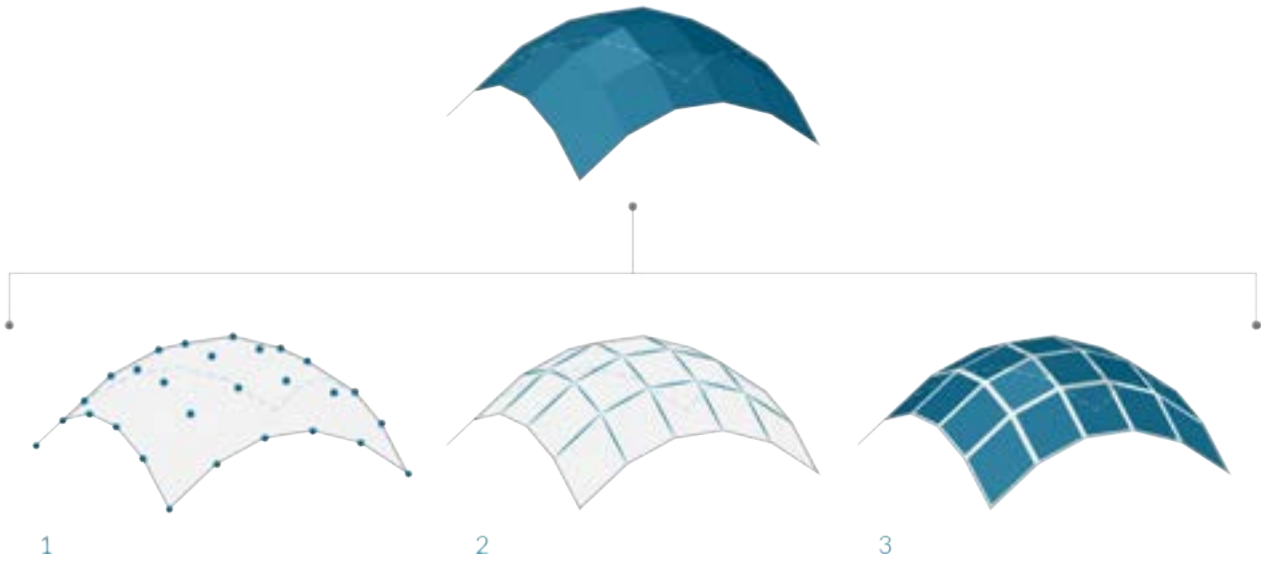
1.6. Getting Started with Meshes

In the field of computational modeling, meshes are one of the most pervasive forms of representing 3D geometry. Mesh geometry can be a light-weight and flexible alternative to working with NURBS, and are used in everything from rendering and visualizations to digital fabrication and 3D printing. This chapter will provide an introduction to how mesh geometry is handled in Grasshopper.



1.6.1 What is a Mesh?

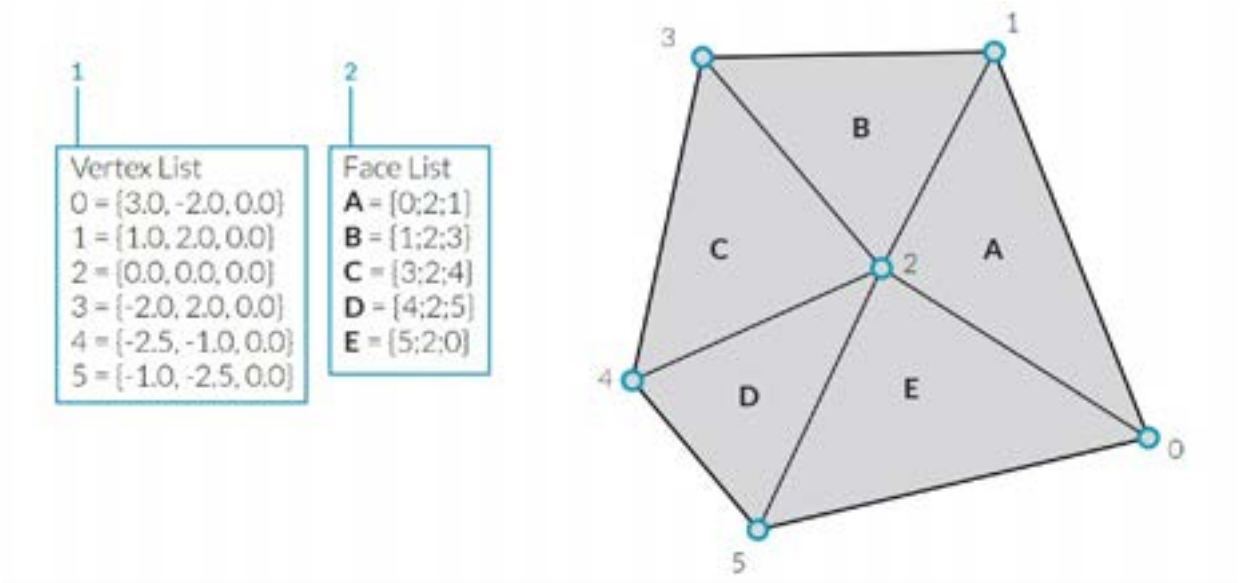
A Mesh is a collection of quadrilaterals and triangles that represents a surface or solid geometry. This section discusses the structure of a mesh object, which includes vertices, edges, and faces, as well as additional mesh properties such as colors and normals.



1. Mesh vertices
2. Mesh edges
3. Mesh faces

1.6.1.1 Basic Anatomy of a Mesh

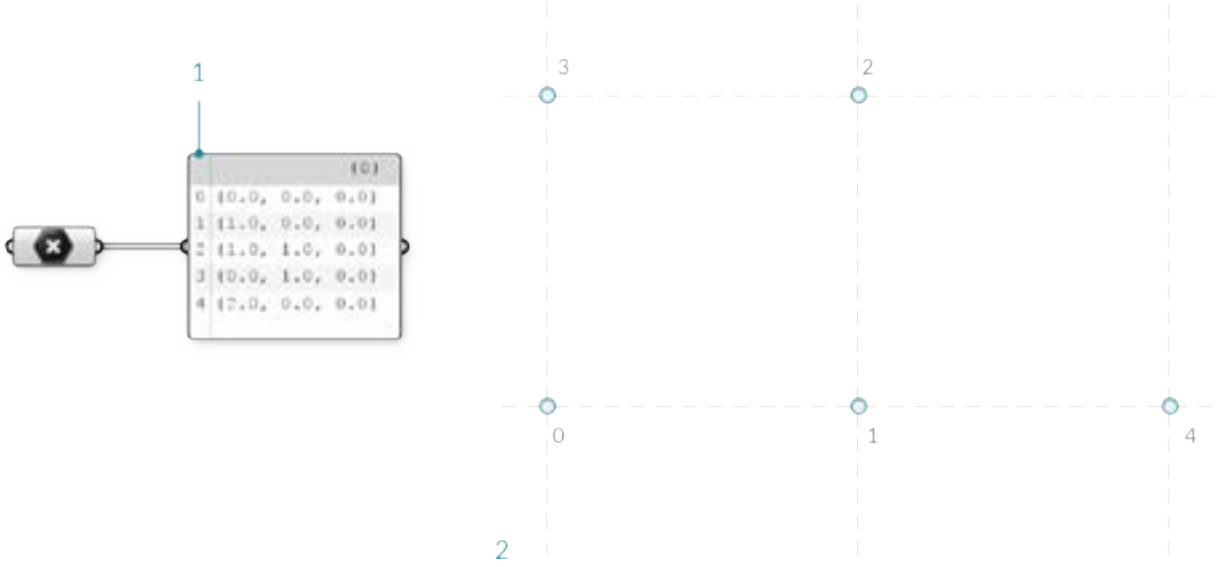
Grasshopper defines meshes using a Face-Vertex data structure. At its most basic, this structure is simply a collection of points which are grouped into polygons. The points of a mesh are called *vertices*, while the polygons are called *faces*. To create a mesh we need a list of vertices and a system of grouping those vertices into faces.



1. A list of vertices.
2. Faces with groupings of vertices

Vertices

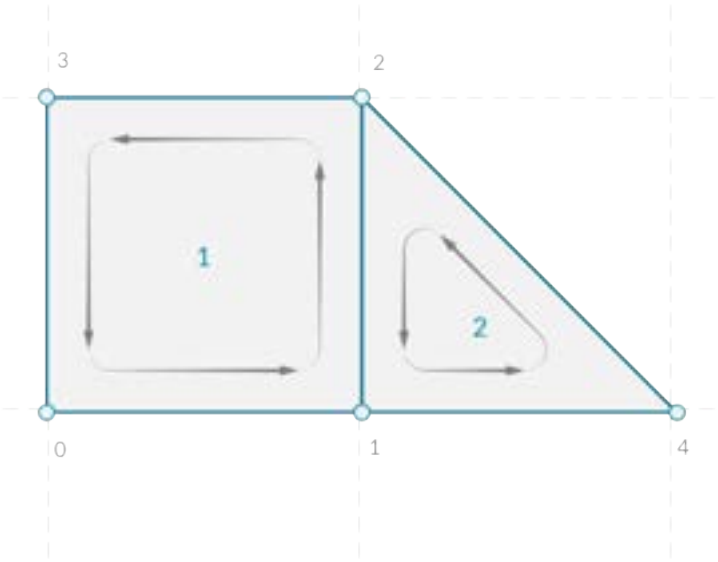
The vertices of a mesh are simply a list of points. Recall that a *list* in Grasshopper is a collection of objects. Each object in the list has an *index* which describes that objects position in a list. The index of the vertices is very important when constructing a mesh, or getting information about the structure of a mesh.



1. Alist of points. All lists in Grasshopper begin with an index of zero
2. The set of points labeled with their index

Faces

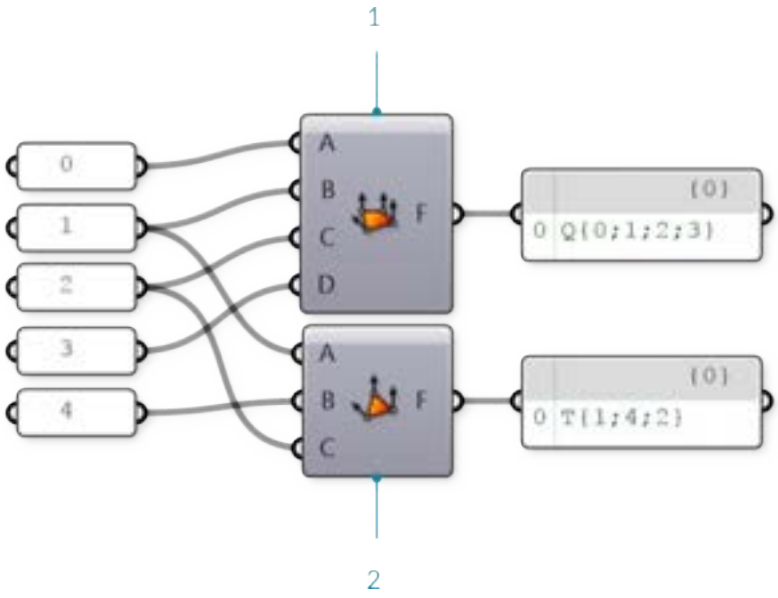
Aface is an ordered list of three or four vertices. The “surface” representation of a mesh face is therefore implied according to the position of the vertices being indexed. We already have the list of vertices that make up the mesh, so instead of providing individual points to define a face, we instead simply use the index of the vertices. This also allows us to use the same vertex in more than one face.



1. Aquad face made with indices 0, 1, 2, and 3
2. Atriangle face made with indices 1, 4, and 2

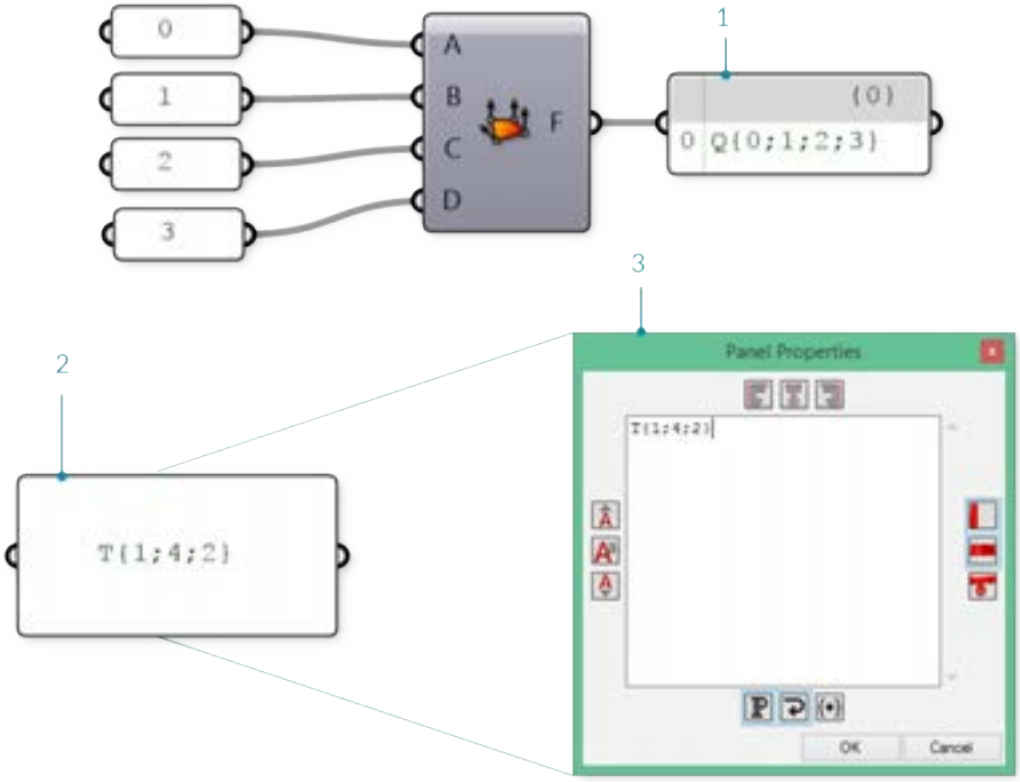
In Grasshopper, faces can be created with the **Mesh Triangle** and **Mesh Quad** components. The input for these components are integers that correspond to the index of the vertices we want to use for a face. By connecting a

Panel to the output of these components, we can see that a triangular face is represented as T{A;B;C}, and a quad face as Q{A;B;C;D}. Faces with more than 4 sides are not allowed. To make a 5-sided mesh element, the mesh must be broken into two or more faces.



1. **Mesh Quad** component with indices 0, 1, 2, and 3
2. **Mesh Triangle** component with indices 1, 4, and 2

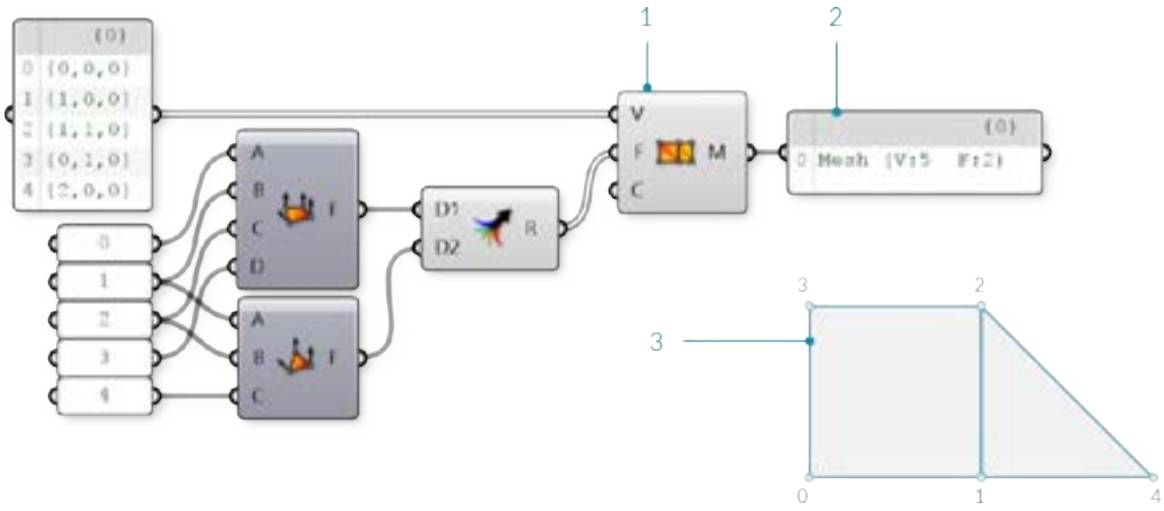
It is important to remember that these components do not result in the creation of mesh geometry, rather the output is a list of indices that define how a mesh should be constructed. By paying attention to the format of this list, we can also create a face manually by editing a **Panel** component and entering the appropriate format for either triangular or quad faces.



1. Aface created using a **Mesh Quad** component

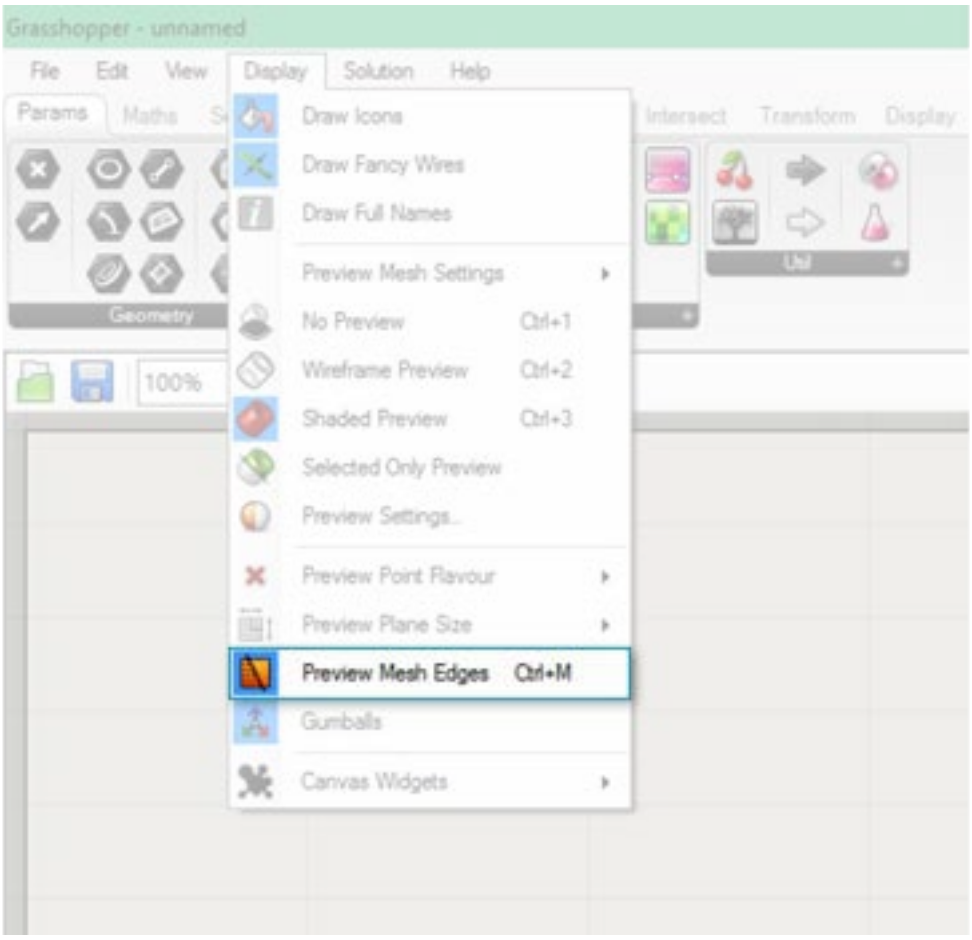
2. A face created using a **Panel**
3. A Panel Properties window is automatically opened when double-clicking a panel while zoomed out, or by right-clicking a panel and selecting "Edit Notes..."

So far we have a list of vertices and a set of face definitions, but have not yet created a mesh. In order to create a mesh, we need to connect the faces and vertices together by using the **Construct Mesh** component. We connect our list of vertices to the V input, and a merged list of faces to the F input. (The component also has room for an optional Color input, which is discussed below.) If we connect a panel to the output of the **Construct Mesh** we can see information about the number of faces and number of indices.

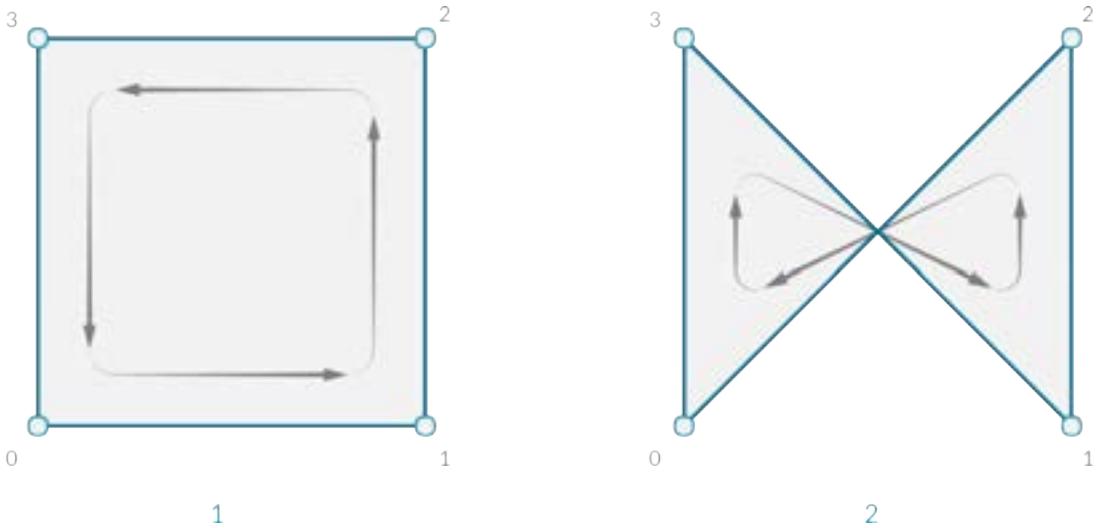


1. The **Construct Mesh** component takes a list of vertices and a list of faces as input. The Color input is optional, and is left blank for now
2. A panel shows that we have created a mesh with 5 vertices and 2 faces
3. The resulting mesh (the vertices have been labeled with their indices)

By default, Grasshopper does not preview the edges of mesh geometry. To preview the edges as well as the surfaces, you can turn on mesh edge preview by using the shortcut Ctrl-M, or by going to the Display menu and selecting 'Preview Mesh Edges'.



It is extremely important to pay attention to the order of the indices when constructing a mesh face. The face will be constructed by connecting the vertices listed in order, so the quad faces Q{0,1,2,3} and Q{1,0,2,3} are very different, despite using the same four vertices. Incorrect vertex ordering can lead to problems such as holes, non-manifold mesh geometry, or non-orientable surfaces. Such mesh geometry is usually not correctly rendered, and not able to be 3D printed. These issues are discussed in more detail in the **Understanding Topology** section.



1 2

1. Aquad face with indices 0,1,2,3
2. Aquad with indicies 0,3,1,2

1.6.1.2 Implicit Mesh Data

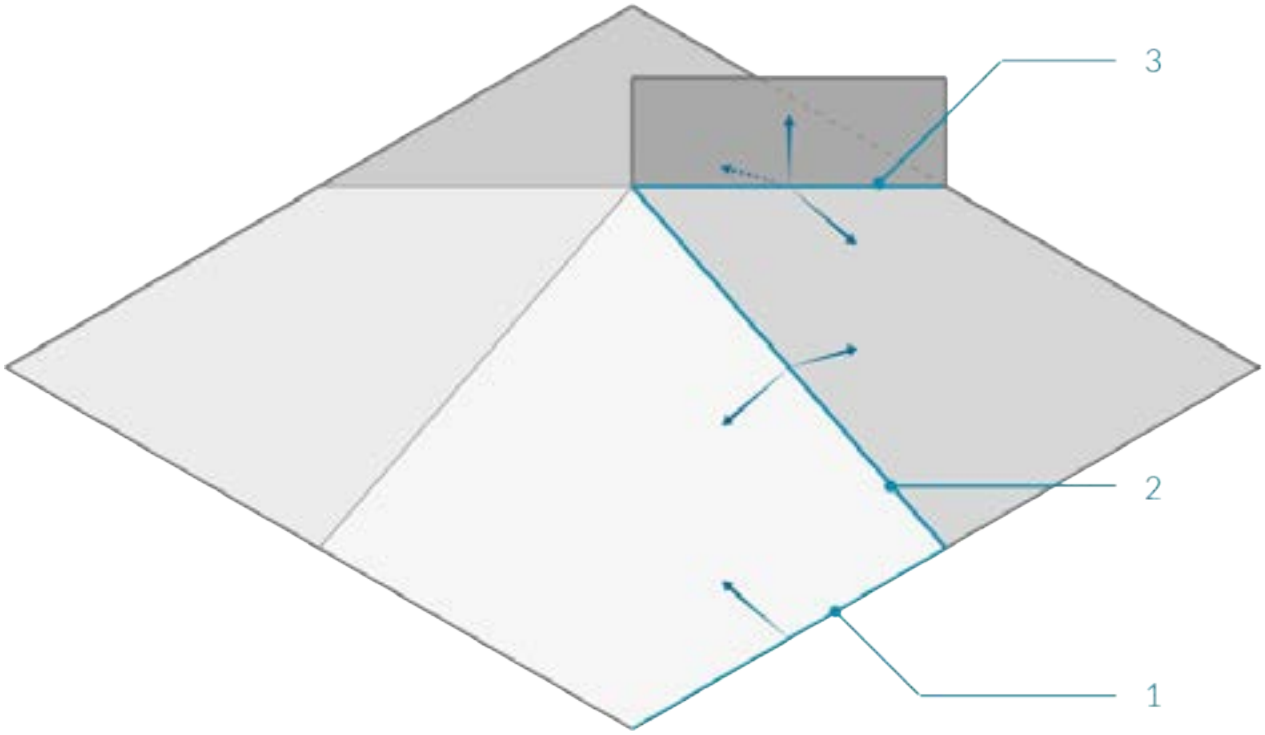
In addition to faces and vertices, there is other information about a mesh that we will want to use. In a Face-Vertex based mesh, data such as *edges* and *normals* are calculated implicitly based on the given faces and vertices. This section describes ways to query this information.

Edges

The *edges* of a mesh are lines connecting any two consecutive vertices in a face. Notice that some edges are shared between multiple faces, while other edges are only adjacent to one face. The number of faces an edge is adjacent to is called the *valence* of that edge.

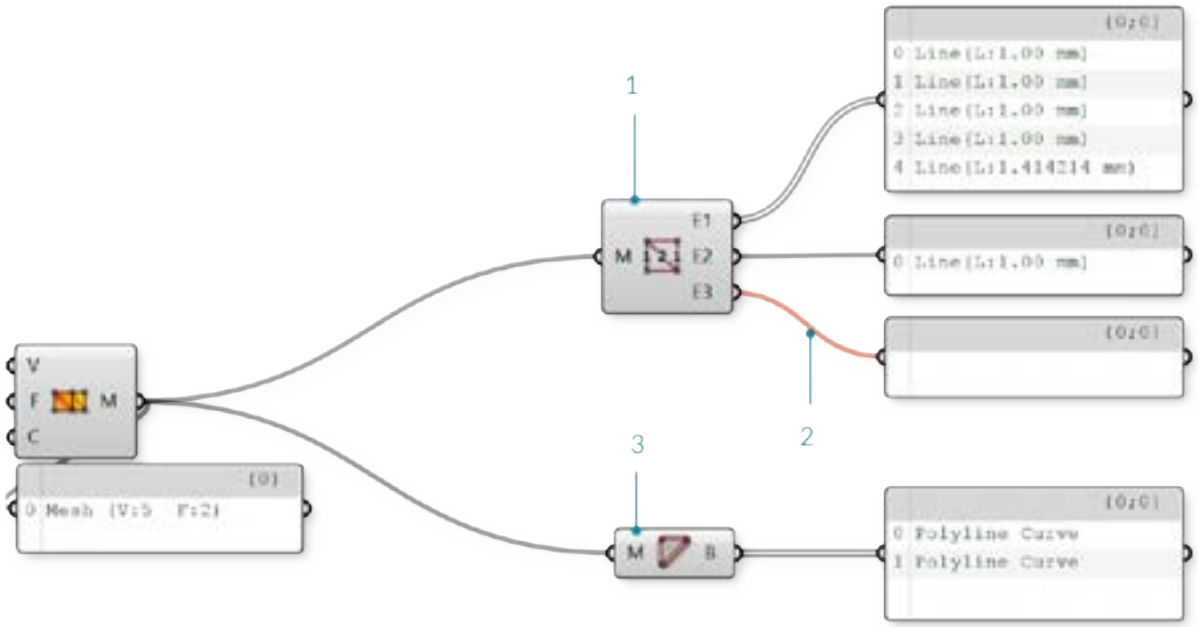
Grasshopper groups edges into three categories based on the valence:

1. E1 - 'Naked Edges' have a valence of 1. They make up the external boundary of a mesh.
2. E2 - 'Interior Edges' have a valence of 2.
3. E3 - 'Non-Manifold Edges' have a valence of 3 or greater. Meshes that contain such structure are called "Non-Manifold", and are discussed in the next section.



1. Naked edge with valence of 1
2. Interior edge with valence of 2
3. Non-manifold edge with valence of 3

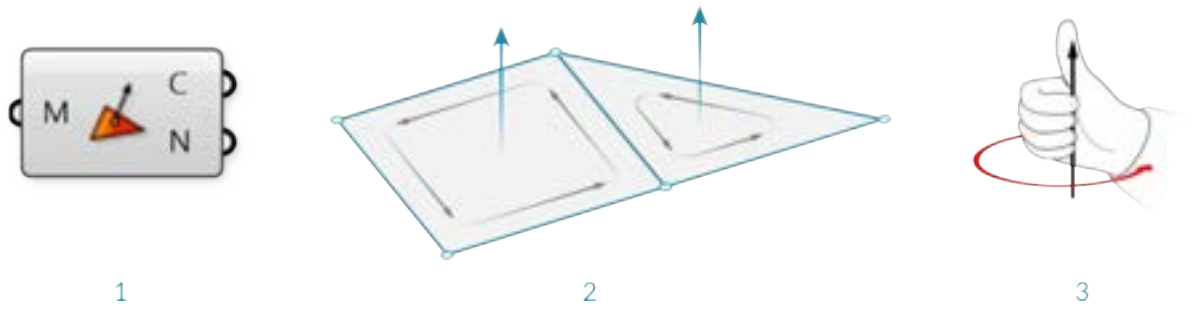
We can use the **Mesh Edges** component to get the edges of a mesh outputted according to valence. This allows us to locate edges along the boundary of a mesh, or to identify non-manifold edges. Sometimes, however, it is more useful to have the full boundary of each face. For this, we can use the **Face Boundaries** component. This will return a polyline for each face.



1. The **Mesh Edges** component outputs three sets of edges. This mesh has 5 naked edges, 1 interior edge, and zero non-manifold edges
2. The E3 output is empty, because this mesh does not have any non-manifold edges, resulting in an orange wire.
3. The **Face Boundaries** component outputs one polyline for each face

Face Normals

A *normal vector* is a vector with a magnitude of one that is perpendicular to a surface. In the case of triangular faces, we know that any three points must be planar, so the normal will be perpendicular to that plane, but how do we know which direction ('up' or 'down') the normal will be pointing? Once again, the order of the indicies is crucial here. Mesh faces in Grasshopper are defined counter-clockwise, so a face with indices {0,1,2} will be 'flipped' as compared to the indicies {1,0,2}. Another way to visualize this is to use the *Right-Hand-Rule*.



1 2 3

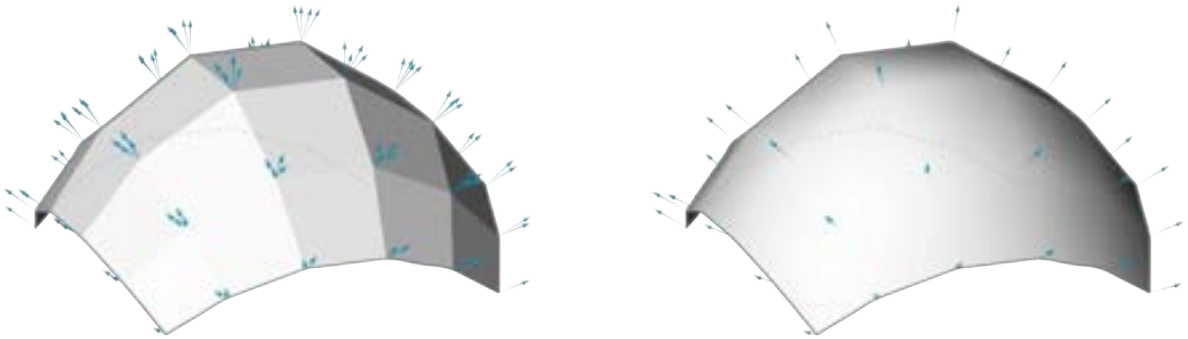
1. The **Face Normals** component will return a list of center points and normal vectors for each face
2. Face normals according to vertex sequence
3. "Right-Hand-Rule" for determining normal direction

Grasshopper also allows quad faces, in which case the 4 points will not always be planar. For these faces, the center point will be simply the average of the coordinates of the 4 vertices (in the case of a non-planar quad, note that this point is not necessarily on the mesh). To calculate the normal of a quad face, we need to first trianglulate the quad by splitting it into two planar triangles. The normal of the overall face is then the average of the two normals, weighted according to the area of the two triangles.

Vertex Normals

In addition to the face normals, it is also possible to calculate normals for each vertex of a mesh. For a vertex that is only used in a single face, the normal at the vertex will point in the same direction as the face normal. If a vertex has multiple adjacent faces, the vertex normal is calculated by taking the average of the faces.

While less intuitive than face normals, vertex normals are important for smooth visualization of meshes. You might notice that even when mesh is composed of planar faces, such a mesh can still appear smooth and rounded when shaded in Rhino. Using the vertex normals allows this smooth visualization.



1 2

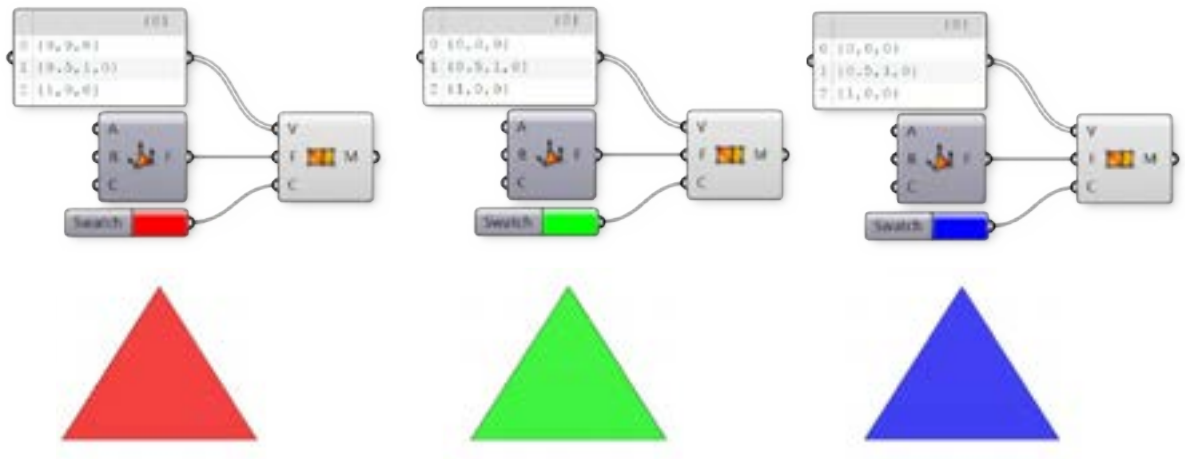
1. Normals set according to the face normal results in discrete polygonal shading
2. Adjancent face normals are averaged together to create vertex normals, resulting in smooth shading across faces

1.6.1.3 Mesh Attributes

Meshes can also be assigned additional attributes to either vertices or faces. The simplest of these is vertex color, which is described below, but other attributes exist such as texture UV coordinates. (Some programs even allow vertex normals to be assigned as attributes instead of being derived from the faces and vertices, which can provide even more flexibility in rendered surface appearance.)

Color

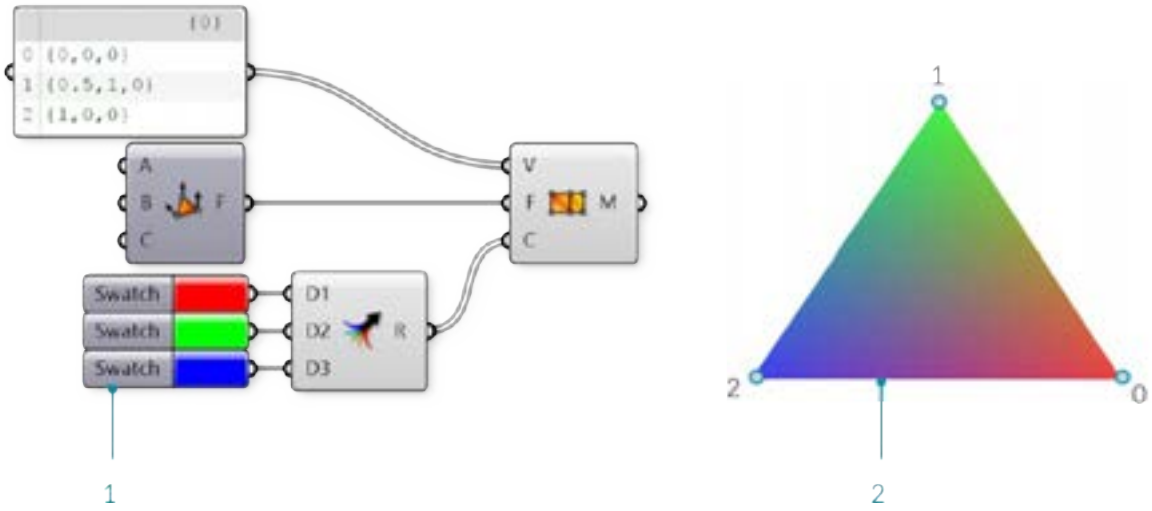
When using a **Construct Mesh** component, there is an option input for vertex color. Colors can also be assigned to an existing mesh using the **Mesh Color** component. By using a single color for a mesh, we can color the entire mesh.



Trianglular mesh objects colored with red, green, or blue

While the above examples colored the entire mesh, color data are actually assigned for each vertex. By using a list

of three colors, we can color each vertex in the triangle separately. These colors are used for visualizations, with each face rendered as an interpolation of the vertex colors. For example, the image below shows a triangular face with vertex colors of Red, Green, and Blue.

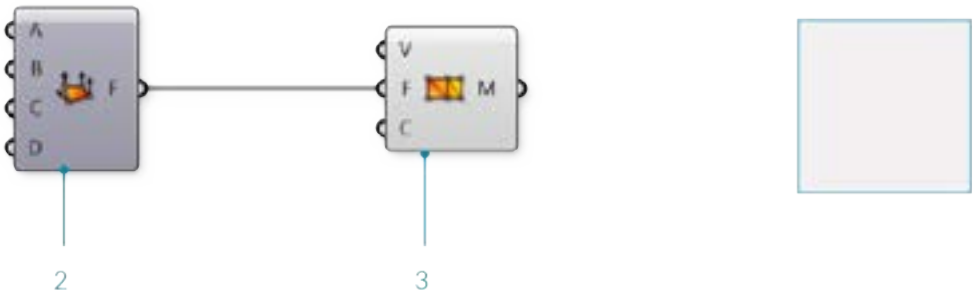


1. Red, green, and blue are assigned to the three vertices of a mesh
2. The resulting mesh interpolates the colors of the vertices

1.6.1.4 Exercise

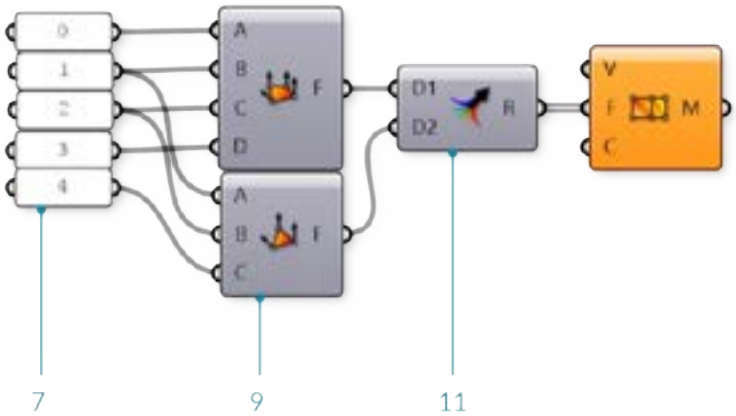
Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

01.	Start a new definition, type Ctrl-N (in Grasshopper)	
02.	**Mesh/Primitive/Mesh Quad** - Drag and drop a **Mesh Quad** component onto the canvas	
03.	**Mesh/Primitive/Construct Mesh** - Drag and drop a **Construct Mesh** component onto the canvas	
04.	Connect the Face (F) output of the **Mesh Quad** component to the Faces (F) input of the **Construct Mesh** component	



Mesh Quad and **Construct Mesh** have default values which create a single mesh face. Next, we will replace the default values with our own vertices and faces.

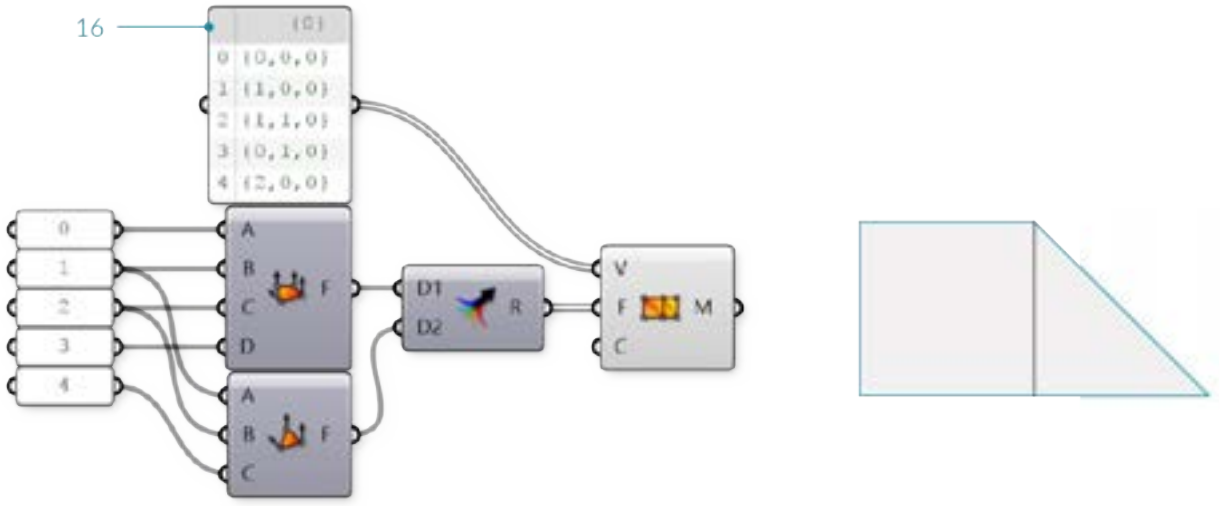
05.	**Params/Input/Panel** - Drag and drop a **Panel** component onto the canvas	
06.	Double-click the **Panel** component and set the value to '0'	
07.	**Params/Input/Panel** - Drag and drop four more **Panel** components onto the canvas and set their values to 1,2,3, and 4 > You can also copy the original **Panel** by clicking and dragging, then tapping the Alt key before releasing the click	
08.	Connect the **Panels** to the inputs of the **Mesh Quad** in the following order: 0 - A 1 - B 2 - C 3 - D	
09.	**Mesh/Primitive/Mesh Triangle** - Drag and drop a **Mesh Triangle** component onto the canvas	
10.	Connect the **Panels** to the inputs of the **Mesh Triangle** component in the following order: 1 - A 2 - B 4 - C	
11.	**Sets/Tree/Merge** - Drag and drop a **Merge** component onto the canvas	
12.	Connect the Face (F) output of the **Mesh Quad** component to the Data1 (D1) input of the **Merge** component, and the Face (F) output of the **Mesh Triangle** component to the Data2 (D2) input of the **Merge** component	
13.	Connect the Result (R) output of the **Merge** component to the Faces (F) input of the **Construct Mesh** component	



> The default Vertices (V) list of **Construct Mesh** only has 4 points, but our **Mesh Triangle** component uses an index of 4, which would correspond to the fifth point in a list. Since there are not enough vertices, the **Construct Mesh** component gives an error. To fix it, we will provide our own list of points.

14.	**Params/Input/Panel** - Drag and drop a **Panel** component onto the canvas	

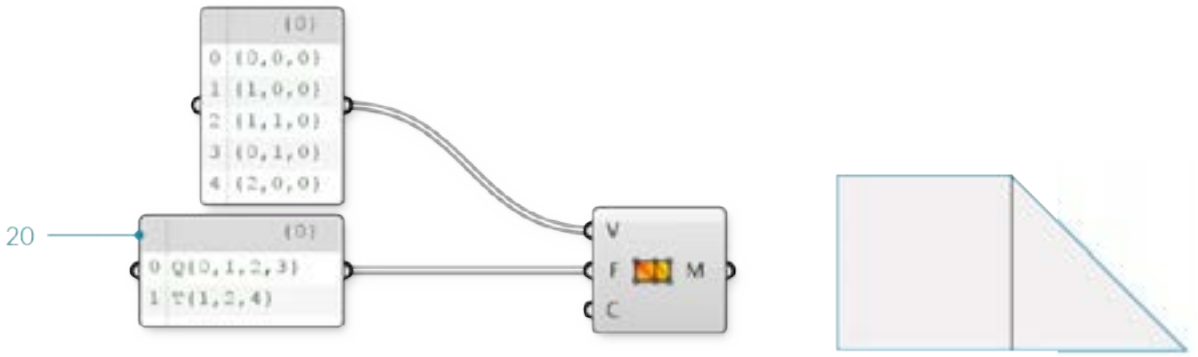
15.	Right-click the **Panel** component and de-select the 'Multiline Data' option > By default, a panel has 'Multiline Data' enabled. By disabling it, each line in the panel will be read as a separate item within a list.	
16.	Double-click the **Panel** component to edit it, and enter the following points: {0,0,0} {1,0,0} {1,1,0} {0,1,0} {2,0,0} > Make sure you use the correct notation. To define a point in a **Panel**, you have to use curly brackets: '{' and '}' with commas between the x, y, and z values	
17.	Connect the **Panel** component to the Vertices (V) input of the **Construct Mesh** component	



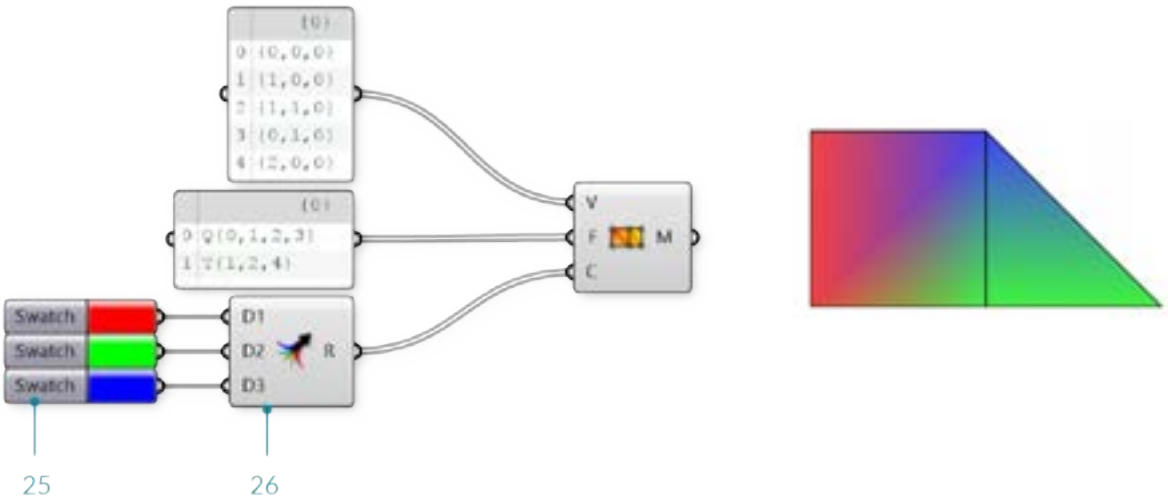
> We now have a mesh with two faces and 5 vertices.

Optionally, we can replace the **Mesh Quad** and **Mesh Triangle** components with a panel specifying the indices of the faces.

18.	**Params/Input/Panel** - Drag and drop a **Panel** component onto the canvas	
19.	Right-click the **Panel** component and deselect 'Multiline Data' > Alternatively, copy the existing **Panel** that we used for the points, which already has 'Multiline Data' disabled	
20.	Double-click the **Panel** component to edit it, and enter the following: Q{0,1,2,3} T{1,2,4}	
21.	Connect the **Panel** to the Faces (F) input of the **Construct Mesh** component	



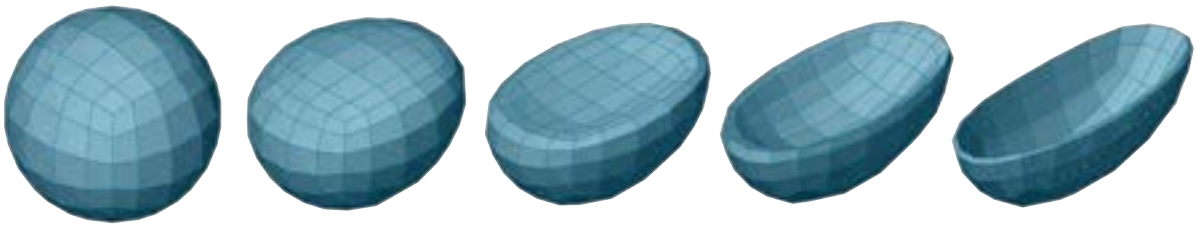
22.	**Params/Input/Colour Swatch** - Drag and drop a **Colour Swatch** component onto the canvas	
23.	Click the colored section of the component (the default is White) to open the color selection panel	
24.	Use the sliders to set the G and B values to zero. The swatch should now be Red	
25.	**Params/Input/Colour Swatch** - Drag and drop two more **Colour Swatch** components onto the canvas and set their colors to Blue and Green	
26.	**Sets/Tree/Merge** - Drag and drop a **Merge** component onto the canvas	
27.	Connect the three **Color Swatch** components into the D1, D2, and D3 inputs of the **Merge** component.	
28.	Connect the Result (R) output of the **Merge** component to the Colours (C) input of the **Construct Mesh** component	



> We have 5 vertices, but only 3 colors. Grasshopper will assign the colors in a repeating pattern, so in this cases vertices 0 and 3 will be Red, vertices 1 and 4 will be Green, and the final vertex 2 will be Blue.

1.6.2 Understanding Topology

While the vertices of a mesh contain position information, it is really the connections between the vertices that give a mesh geometry its unique structure and flexibility.

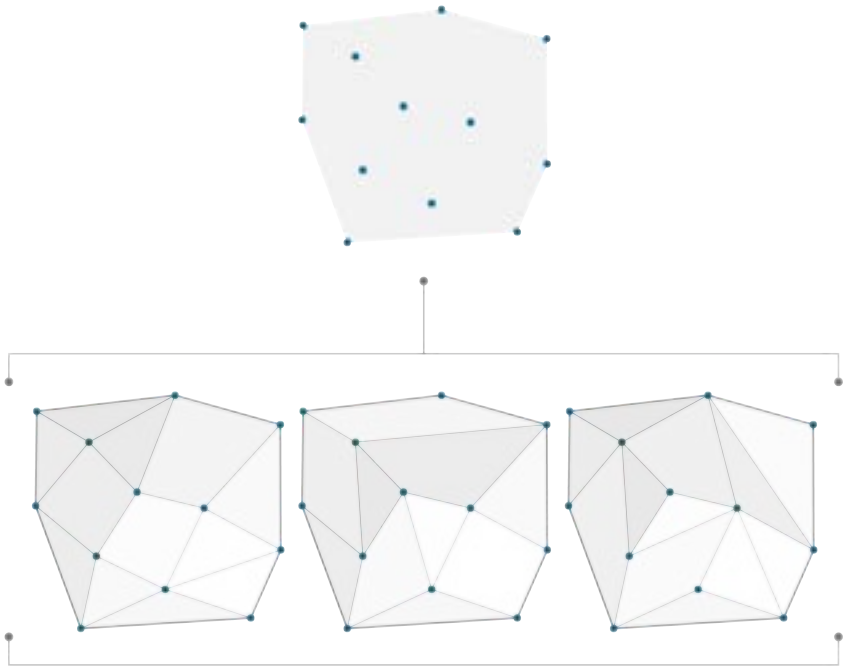


1.6.2.1 What is Topology?

Any introduction to mesh geometry would be incomplete without at least a basic introduction to topology. Because topology is concerned with the inter-relationships and properties of a set of “things” rather than the “things” themselves, it is mobilized for an enormous range of both tangible and intangible applications. In this primer, we are interested in its basic application with respect to parametric workflows that afford us the opportunity to create and control mesh geometry.

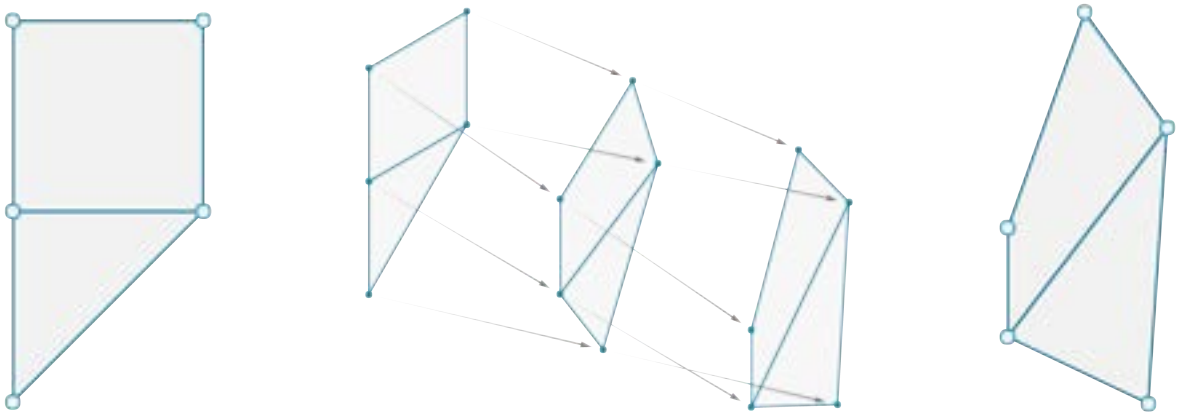
In Grasshopper, the two basic types of information required to define a mesh are geometry and connectivity; in other words, a set of points in rhino-space (vertices) and set of corresponding point-associations (faces).

Without connectivity information, a mesh is unstructured and therefore still undefined. The introduction of a set of faces is the step (or leap) that ultimately actualizes a mesh and establishes its character in terms of continuity, convergence, and connectedness; this structural network is referred to as a *topological space*.



> The same set of vertices can have different connectivity information, resulting in different topology.

Homeomorphism



> The points of a mesh can be moved without altering the connectivity information. The new mesh has the same topology as the original.

It is possible for two distinct mesh shapes to be topologically identical. All this would mean is that they are constructed out of the same number of points and that the points are structured by the same set of faces. Earlier, we established that a mesh face is only concerned with the indices of a set of points and has no interest in their actual location in rhino-space. Therefore, if the only difference between two distinct mesh shapes is the specific 3-dimentional position of the points that are used to define it, then the two meshes are considered to be “homeomorphic” (or topologically equivalent) and therefore share the same topological properties.

{A, R} {B} {C, G, I, J, L, M, N, S, U, V, W, Z}
{D, O} {E, F, T, Y} {H, K} {P, Q} {X}

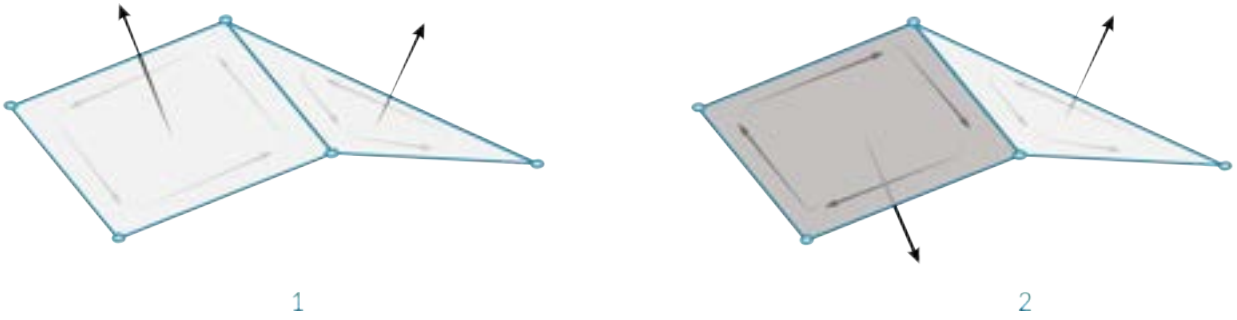
> An example of homeomorphism among letters (note that some of the above homeomorphic groups might be different depending on what font is considered)

> Atopologically equivalent mug and donut

1.6.2.2 Mesh Characteristics

Orientable

Amesh is considered *orientable* if there are well-defined sides to the mesh. An simple example of a non-orientable mesh occurs when adjacent faces have normals pointing in opposite directions. These 'flipped faces' can cause problems in visualizations and renderings, as well as manufacturing or 3D-printing.



1 2

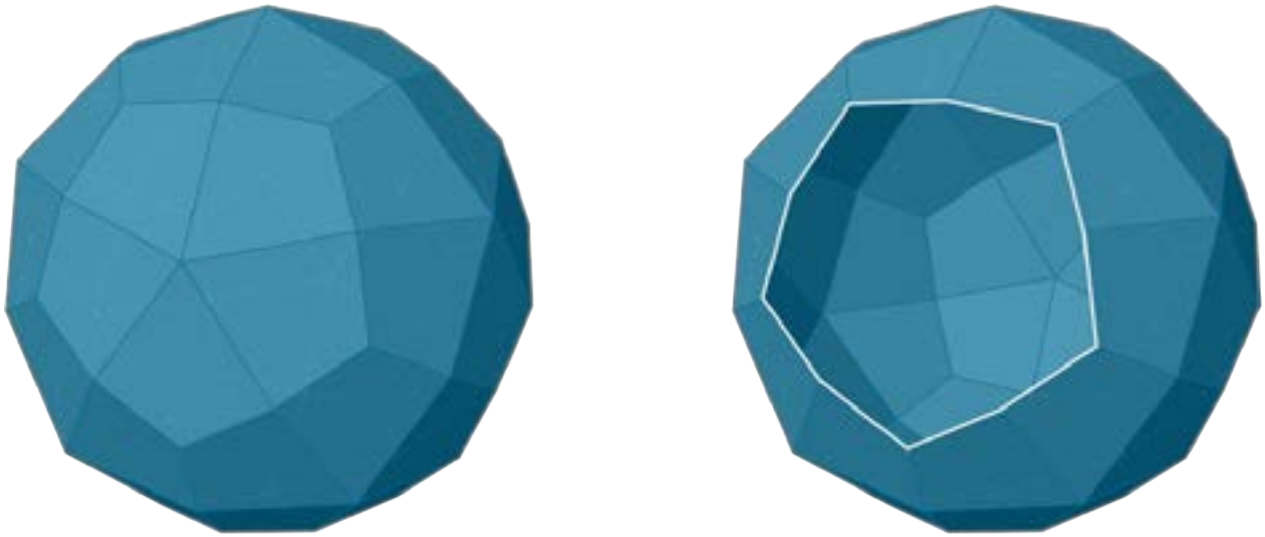
1. An orientable surface with face normals pointing in the same direction.
2. Anon-orientable surface has adjacent normals pointing in different directions.

Open vs Closed

It is often necessary to know whether a mesh is a *closed* mesh which represents a volumetric solid, or an *open* mesh that just represents a 2-dimensional surface. The difference can be imperative with respect to manufacturability. You cannot 3D print a single surface which has no thickness, but must instead thicken the mesh so that it is a solid. Solid mesh geometry is also required for Boolean operations (discussed in following section).

The **Mesh Edges** component can be used to help determine this. If none of the edges of a mesh have a valence of 1 (if the E1 output is *null*), then we know that all the edges are 'Interior Edges' and the mesh does not have an external boundary edge, and is therefore a closed mesh.

On the other hand, if there exist 'Naked Edges', then those edges must be on a boundary of the mesh, and the mesh is not closed.

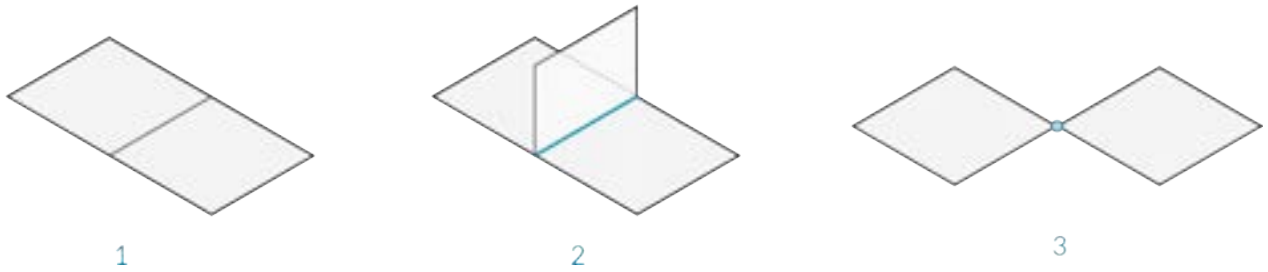


1 2

1. Aclosed mesh. All edges are adjacent to exactly two faces.
2. An open mesh. The white edges are each adjacent to only a single face.

Manifold vs Non-Manifold

Non-manifold geometry is essentially geometry that cannot exist in the "real world". This does not necessarily make it "bad geometry" but it is something to be aware of due to complications it may present for tools and operations (for example: rendering of refractive effects, fluid simulations, boolean operations, 3d printing, etc). Common conditions that result in a non-manifold mesh include: self intersection, naked edges (from holes or internal faces), disjoint topology, and overlapping/duplicate faces. Amesh can also be considered *Non-Manifold* if it includes any vertices which are shared by faces that do not share edges or any edges with a valence greater than 2, creating a junction of at least 3 faces



1. Asimple manifold mesh
2. Three faces meeting on a single edge is non-manifold, also known as a T-Junction
3. Two faces meeting at only one vertex but not sharing an edge is non-manifold

1.6.2.3 Meshes Vs NURBS

How is mesh geometry different from NURBS geometry? When might you want to use one instead of the other?

Parameterization

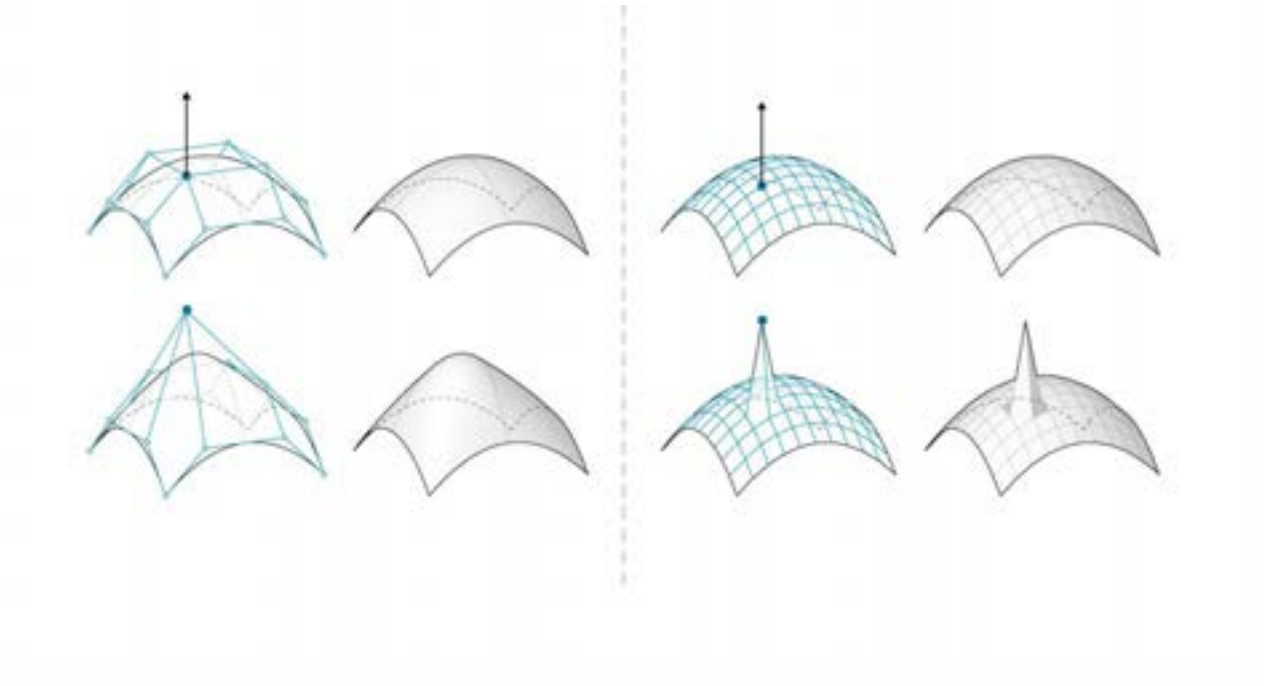
In a previous chapter, we saw that NURBS surfaces are defined by a series of NURBS curves going in two directions. These directions are labeled U and V, and allow a NURBs surface to be parameterized according to a two-dimensional surface domain. The curves themselves are stored as equations in the computer, allowing the resulting surfaces to be calculated to an arbitarily small degree of precision. It can be difficult, however, to combine multiple NURBS surfaces together. Joining two NURBS surfaces will result in a polysurface, where different sections of the geometry will have different UV parameters and curve definitions.

Meshes, on the other hand, are comprised of a discrete number of exactly defined vertices and faces. The network of vertices generally cannot be defined by simple UV coordinates, and because the faces are discrete the amount of precision is built into the mesh and can only be changed by refining the mesh and added more faces. The lack of UV coordinates, however, allows meshes the flexibility to handle more complicated geometry with a single mesh, instead of resorting to a polysurface in the case of NURBS.

> Note - While a mesh does not have implicit UV parameterization, it is sometimes useful to assign such a parameterization in order to map a texture or image file onto mesh geometry for rendering. Some modeling software therefore treats the UV coordinates of a mesh vertex as an *attribute* (like vertex color) which can be manipulated and changed. These are usually assigned and not completely defined by the mesh itself.

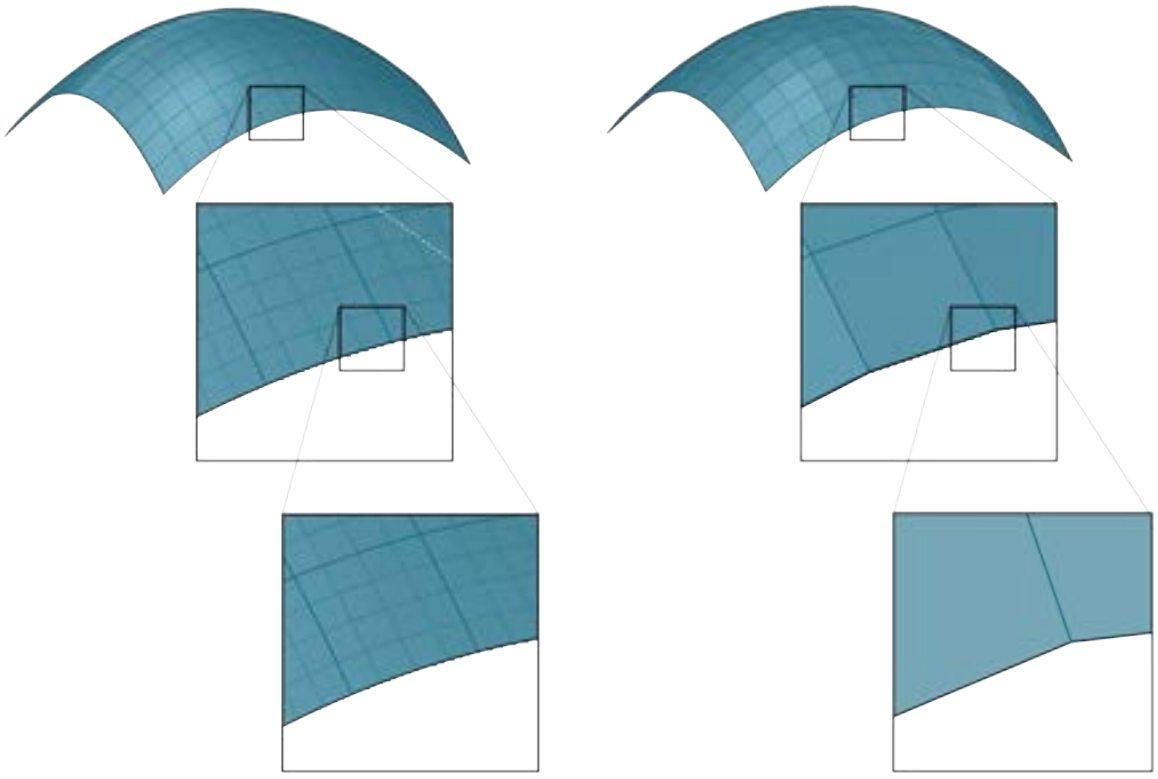
Local vs Global Influence

Another important difference is the extent to which a local change in mesh or NURBS geometry affects the entire form. Mesh geometry is completely local. Moving one vertex affects only the faces that are adjacent to that vertex. In NURBS surfaces, the extent of the influence is more complicated and depends on the degree of the surface as well as the weights and knots of the control points. In general, however, moving a single control point in a NURBS surface creates a more global change in geometry.



1. NURBS Surface - moving a control point has global influence
2. Mesh geometry - moving a vertex has local influence

One analogy that can be helpful is to compare a vector image (composed of lines and curves) with a raster image (composed of individual pixels). If you zoom into a vector image, the curves remain crisp and clear, while zooming into a raster image results in seeing individual pixels. In this analogy, NURBS surfaces can be compared to a vector image, while a mesh behaves similarly to a raster image.



> Zooming into a NURBS surface retains a smooth curve, while a mesh element has a fixed resolution

It is interesting to note that while NURBs surfaces are stored as mathematical equations, the actual visualization of these surfaces requires meshes. It is not possible for a computer to display a continuous equation. Instead, it must break that equation down into smaller parts, the result of which is that all rendering or display processing must convert NURBS to meshes. As an analogy, consider that even though we can store the equation of a line on a computer, in order to display that line, the computer must at some point convert the line into a series of discrete pixels on a screen to display.

1.6.2.4 Pros and Cons of Meshes

When we ask “What are the pros and cons of modeling with meshes?” we are really asking “What are the pros and cons of modeling with shapes that are defined solely by a set of vertices and a corresponding topological framework?” Through this method of framing the question it is easier to see how the “simplistic” nature of a mesh is the critical aspect that would make a mesh either favorable or unfavorable to model with depending on the context of its application.

Meshes can be favorable in situations where:

- There must be a dynamically updated rendering of a form that is changing in shape but not in face connectivity
- Adiscretized approximation of a curved geometry would suffice
- Alow-resolution geometry must be systematically smoothed (or articlated) using computational methods to arrive at a higher-resolution model.
- The low resolution model must be able to be to simultaneously support local, high resolution detail

Meshes can be unfavorable in situations where:

- Curvature and smoothness must be represented with a high level of precision
- True derivatives must be evaluated
- The geometry must be converted into a manufacturable solid
- The final form must be able to be easily edited manually

1.6.3 Creating Meshes

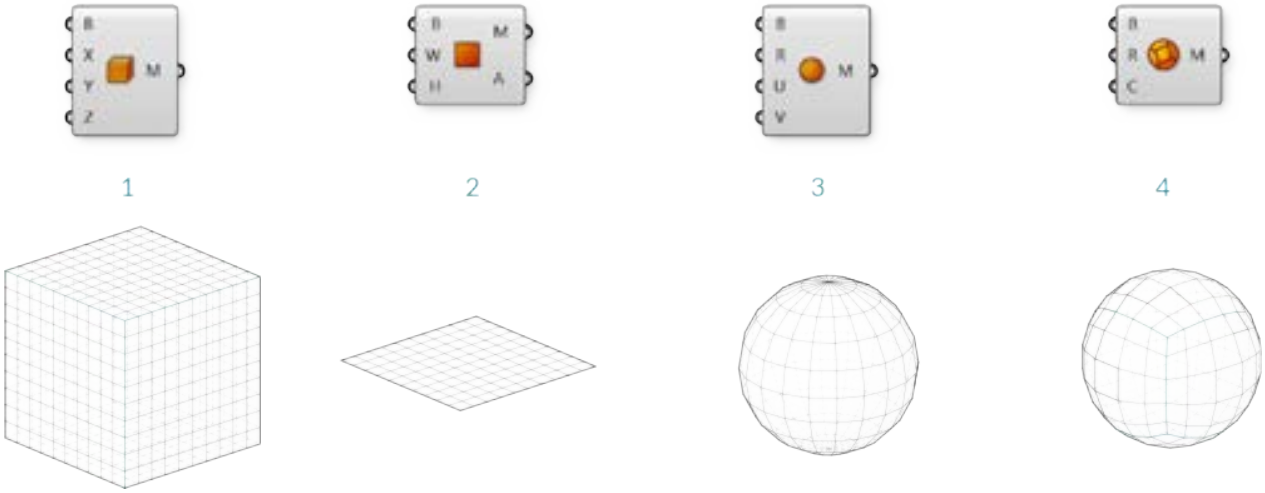
In the last section, we looked at the basic structure of meshes. In this section, we give a brief introduction to different ways of generating mesh geometry.

There are three fundamental ways of creating mesh geometry in Grasshopper:

1. Starting with a mesh primitive
2. Manually constructing a mesh from faces and vertices
3. Converting NURBS geometry into a mesh

1.6.3.1 Primitive

Grasshopper comes with a few simple mesh primitive components:



1. **Mesh Box** - This primitve requires a Box object as an input which provides the size and location, as well as X,Y, and Z values that determine how many faces to divide the box into. The six sides of a Mesh Box are *unwelded* allowing for creases. (See the following section for more information about welded meshes)
2. **Mesh Plane** - This primitive requires a Rectangle input to determine the size and location of the plane, as well as W and H values to determine the number of faces.
3. **Mesh Sphere** - This primitive requires a base plane to determine the center and orientation of the sphere, a radius for the size, and U and V values to determine the number of faces.
4. **Mesh Sphere Ex** - Also known as a Quadball, this primitive creates a sphere composed of six patches, which are subdivided according to the C input. A quadball is topologically equivalent to a cube, even though it is geometrically spherical.

1.6.3.2 Construct Mesh

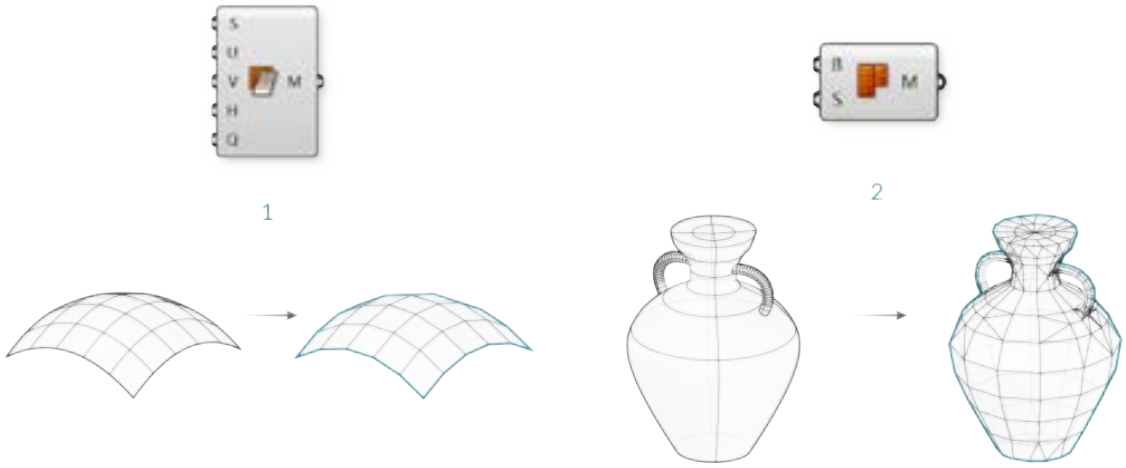
As we saw in the previous section, the **Construct Mesh** component can be used to directly create a mesh from a list of vertices and a list of faces (and an optional list of vertex colors). Constructing an entire mesh manually can be extremely tedious, so this component is more often used with an existing list of faces and vertices which have been

extracted using a **Deconstruct Mesh** component on an existing mesh.

1.6.3.3 NURBS to Mesh

Perhaps the most common method of creating a complex mesh is to generate one based off of NURBS geometry. Individual NURBS surfaces can be converted to a mesh using the **Mesh Surface** component, which simply subdivides the surface along its UV coordinates and creates quad faces. This component allows you to enter the number of U and V divisions for the resultant mesh.

More complex polysurfaces can be converted to a single mesh with the **Mesh Brep** component. This component also has an optional Settings input, which can be set by using one of the built in *Speed*, *Quality*, or *Custom* Settings components, or by right-clicking the S input and selecting "Set Mesh Options". For efficient use of meshes, it is often necessary to refine this mesh by using various strategies such as rebuilding, smoothing, or subdividing. Some of these techniques will be discussed later in this Primer.



1. **Mesh Surface** converts a NURBS surface to a mesh
2. **Mesh Brep** can convert polysurfaces and more complicated geometry into a single mesh. Adjusting the settings will allow for more or less faces, and a finer or coarser mesh.

NOTE: it is generally much easier to convert from a NURBS geometry to a mesh object rather than the other way around. While the UV coordinates of a NURBS surface are straightforward to convert to quad faces of a mesh, the opposite is not necessarily true, since a mesh might contain a combination of triangles and quads in a way that is not simple to extract a UV coordinate system out of.

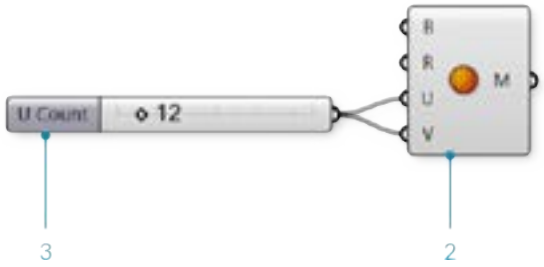
1.6.3.4 Exercise

In this exercise, we use a basic Mesh primitive, perform a transformation on the vertices, and then assign a color based on the normal vectors to approximate the rendering process.

Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

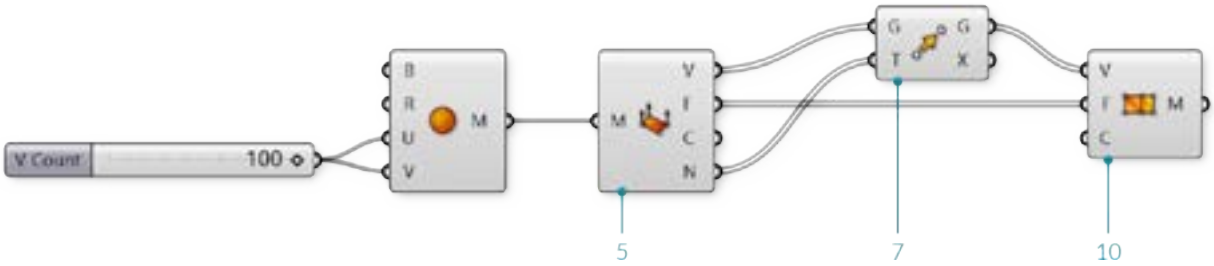
01.	Start a new definition, type Ctrl-N (in Grasshopper)	
02.	**Mesh/Primitive/Mesh Sphere** - Drag and drop a **Mesh Sphere** component onto the canvas	
03.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Rounding: Integer Lower Limit:0	

	Upper Limit: 100 Value: 10	
04.	Connect the **Number Slider** to the U Count (U) and V Count (V) inputs of the **Mesh Sphere** Component	



> Adjust the slider and notice how the resoultion of the sphere changes in the Rhino viewport. Higher numbers result in a smoother sphere, but also produce larger datasets which can require more processing time.

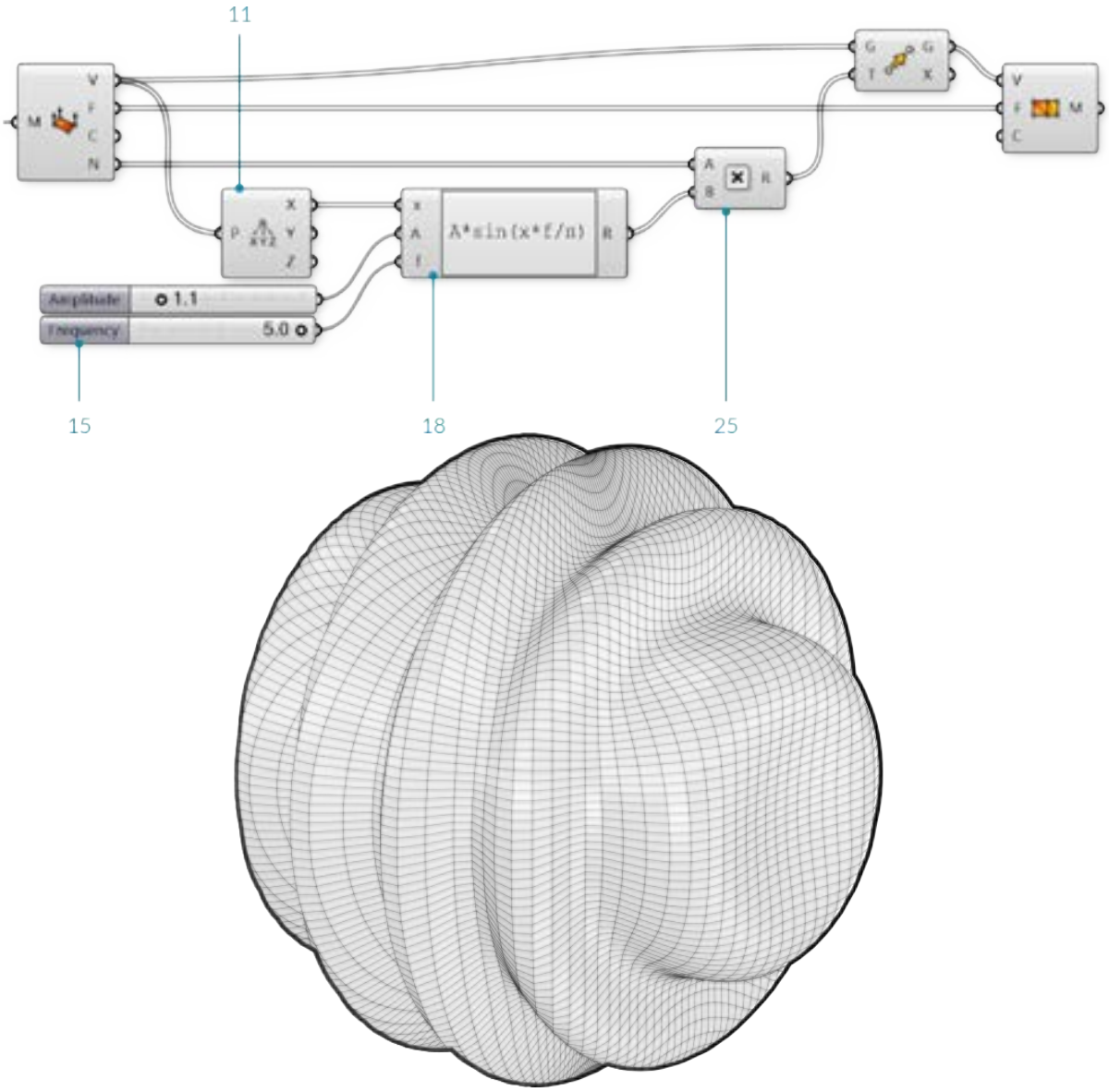
05.	**Mesh/Analysis/Deconstruct Mesh** - Drag and drop a **Deconstruct Mesh** component onto the canvas	
06.	Connect the Mesh (M) output of the **Mesh Sphere** component to the Mesh (M) input of the **Deconstruct Mesh** component	
07.	**Transform/Euclidean/Move** - Drag and drop a **Move** component onto the canvas	
08.	Connect the Vertices (V) output of the **Deconstruct Mesh** component to the Geometry (G) input of the **Move** component	
09.	Connect the Normals (N) output of the **Deconstruct Mesh** component to the Motion (T) input of the **Move** component	
10.	**Mesh/Analysis/Construct Mesh** - Drag and drop a **Construct Mesh** component onto the canvas	
11.	Connect the Geometry (G) output of the **Move** component to the Vertices (V) input of the **Construct Mesh** component	
12.	Connect the Faces (F) output of the **Deconstruct Mesh** component to the Faces (F) of the **Construct Mesh** component	



> We deconstructed the mesh to get its vertices, faces, and normals. We then simply moved each vertex according to its normal vector. Because we did not change the topology of the sphere at all, we re-used the list of faces to re-construct the new mesh. Normal vectors always have a length of one, so this ended up reconstructing a new mesh sphere with a radius of one more than the original sphere.

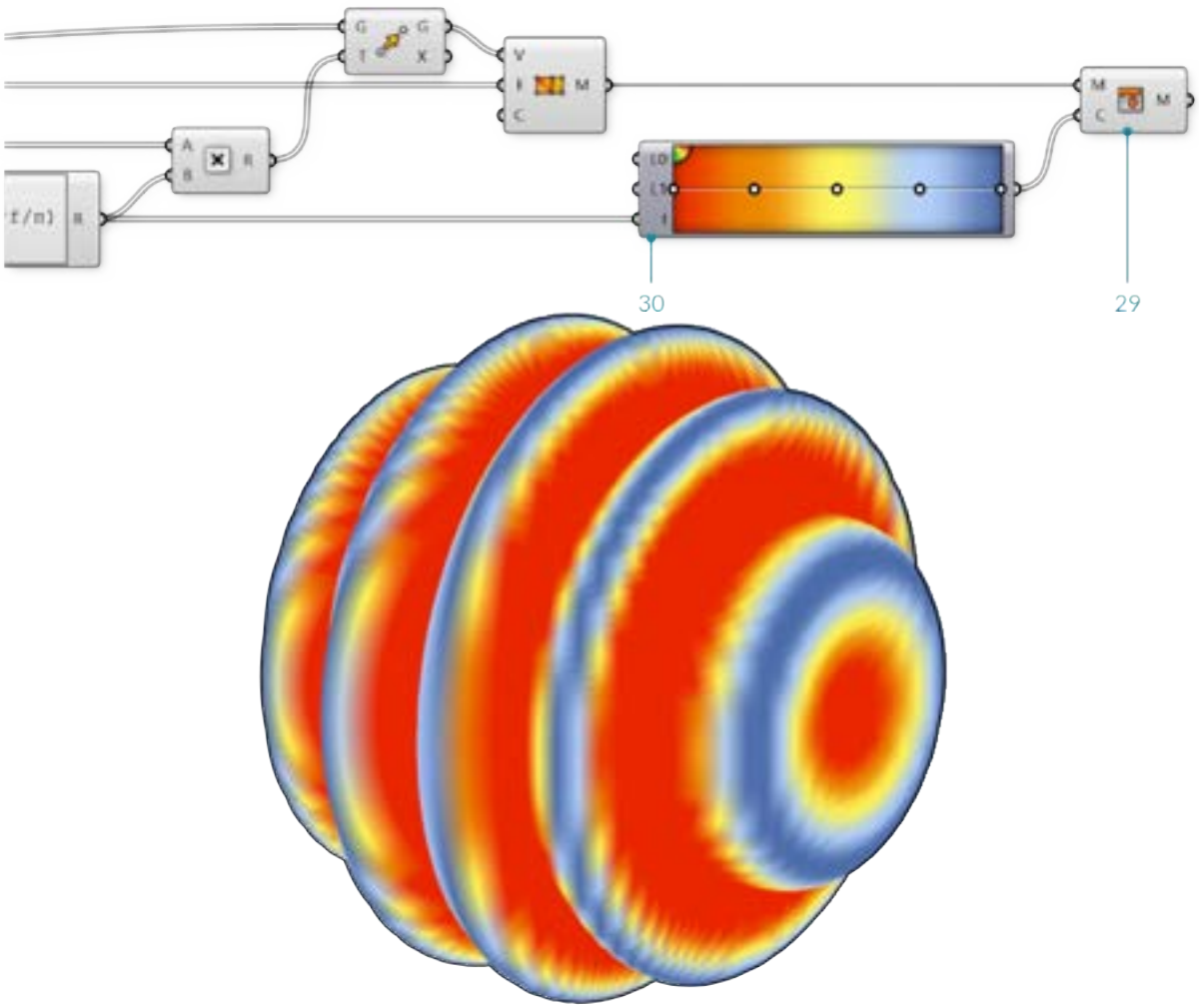
Next, we will use a sine function to manipulate the sphere in a slightly more complicated way.

13.	**Vector/Point/Deconstruct** - Drag and drop a **Deconstruct** component onto the canvas	
14.	Connect the Vertices (V) output of the **Deconstruct Mesh** component to the Point (P) input of the **Deconstruct** component	
15.	**Params/Input/Number Slider** - Drag and drop two **Number Slider** components onto the canvas	
16.	Set the values of the frist **Number Slider** to: Name: Amplitude Rounding: Float Lower Limit: 0 Upper Limit: 10	
17.	Set the values of the second **Number Slider** to: Name: Frequency Rounding: Float Lower Limit: 0 Upper Limit: 5	
18.	**Maths/Script/Expression** - Drag and drop an **Expression** component onto the canvas	
19.	Zoom in to the **Expression** component until you see the options for adding or removing input variables and click on a '+' to add a 'z' variable	
20.	Right click the 'y' input of the **Expression** component and change the text to 'A'	
21.	Right click the 'z' input of the **Expression** component and change the text to 'f'	
22.	Double click the **Expression** component to edit the expression, and enter the following: A*sin(x*f/π)	
23.	Connect the X output of the **Deconstruct** component to the 'x' input of the **Expression** component	
24.	Connect the Amplitude **Number Slider** to the Ainput, and the Frequency **Number Slider** to the 'f' input of the **Expression** component	
25.	**Maths/Operators/Multiplication** - Drag and drop a **Multiplication** component onto the canvas	
26.	Connect the Normals (N) output of the **Deconstruct Mesh** component to the Ainput of the **Multiplication** component	
27.	Connect the Result (R) output of the **Expression** component to the the B input of the **Multiplication** component	
28.	Connect the Result (R) output of the **Multiplication** component to the Motion (T) input of the **Move** component	



> Adjust the Amplitude and Frequency number sliders to see how the newly constructed mesh changes.

29.	**Mesh/Primitive/Mesh Colours** - Drag and drop a **Mesh Colours** component onto the canvas	
30.	**Params/Input/Gradient** - Drag and drop a **Gradient** component onto the canvas > You can right-click the gradient component and select "Presets" to change the color gradient. In this example, we used the Red-Yellow-Blue gradient	
31.	Connect the Result (R) output of the **Expression** component to the Parameter (t) input of the **Gradient** component	
32.	Connect the output of the **Gradient** component to the Colours (C) input of the **Mesh Colours** component	
33.	Connect the Mesh (M) output of the **Construct Mesh** component to the Mesh (M) input of the **Mesh Colours** component > In this step, we could achieve the same result by connecting the gradient directly to the Colours (C) input of the **Construct Mesh** component	

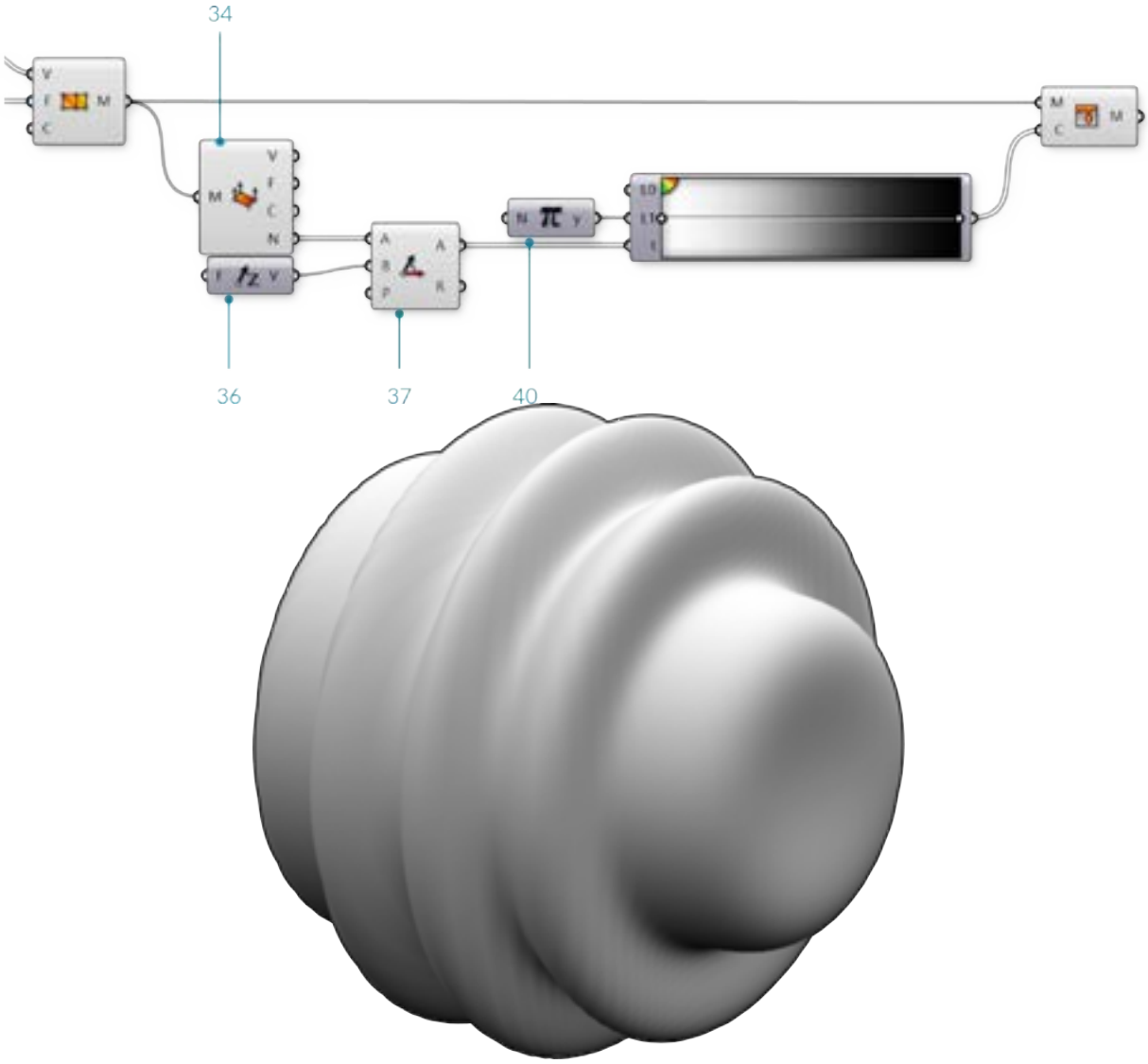


> We used the Expression results to drive both the movement of the vertices and the color of the mesh, so the color gradient in this case corresponds to the magnitude of the movement of the vertices.

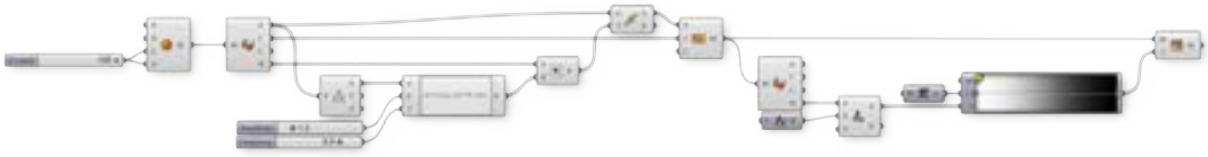
For the final portion of the exercise, we will instead use the direction of the normals relative to a 'light source' vector to simulate the basic process of rendering a mesh.

34.	**Mesh/Analysis/Deconstruct Mesh** - Drag and drop a **Deconstruct Mesh** component onto the canvas	
35.	Connect the Mesh (M) output of the **Construct Mesh** component to the Mesh (M) input of the **Deconstruct Mesh** component > While the topology of the original mesh has not changed, the normal vectors will be different, so we need to use a new **Deconstruct Mesh** to find the new normals. </blockquote	
36.	**Vector/Vector/Unit Z** - Drag and drop a **Unit X** component onto the canvas > We will use this as the direction of a light source. You can use other vectors, or reference a line from Rhino to make this more dynamic	
37.	**Vector/Vector/Angle** - Drag and drop an **Angle** component onto the canvas	
38.	Connect the Normals (N) output of the **Deconstruct Mesh** component to the Ainput of the **Angle** component	

39.	Connect the output of the **Unit Z** component to the B input of the **Angle** component	
40.	**Maths/Util/Pi** - Drag and drop a **Pi** component onto the canvas	
41.	Connect the **Pi** component to the Upper Limit (L1) input of the **Gradient** component	
42.	Connect the Angle (A) output of the **Angle** component to the Parameter (t) input of the **Gradient** component	



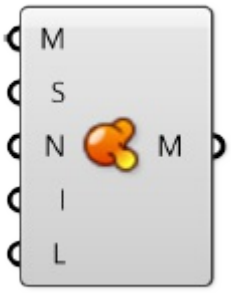
> We used the white-to-black preset for our gradient. This sets the mesh color according to the angle between the normal and the light source, with normals that are directly facing the light source to black and the normals facing away from the source to white (To be a little more accurate, you can reverse the gradient by adjusting the handles). The actual process of rendering a mesh is much more complicated than this, obviously, but this is the basic process of creating light and shadow on a rendered object.



1.6.4 Mesh Operations

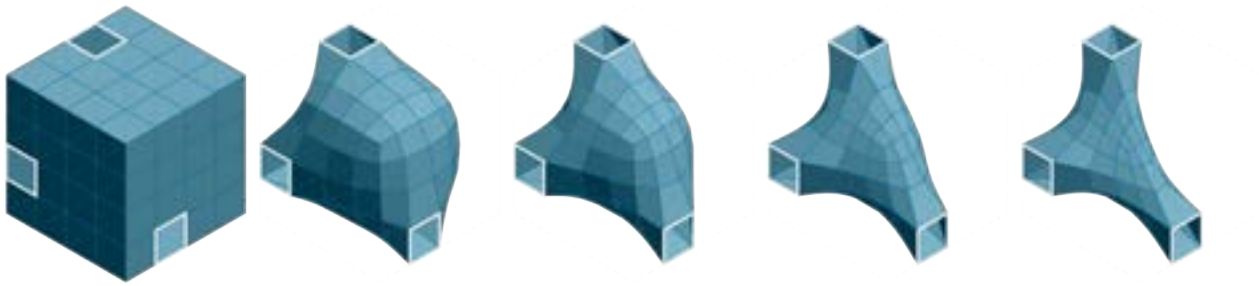
In the last section, we looked at the basic structure of a mesh. In this section, we will look at ways to manipulate mesh geometry.

1.6.4.1 Smooth



Smoother meshes can sometimes be achieved by simply increasing the number of faces in a process called *subdivision*. This can often lead to extremely large datasets which take a long time to calculate, and requires additional add-ons to Grasshopper that are not built-in. In these situations, the **Smooth** component can be used as an alternative to make meshes less jagged or faceted, without increasing vertex and face count or changing the topology. The *strength*, *number of iterations*, and displacement *limit* can all be used to adjust how much smoothing occurs.

Attaching a boolean value to the input N provides option to skip naked vertices. A vertex is naked if it is connected to a naked edge, meaning the vertex is on the boundary of an open mesh. By toggling this option, you maintain the exterior boundary of a mesh while smoothing the interior edges.

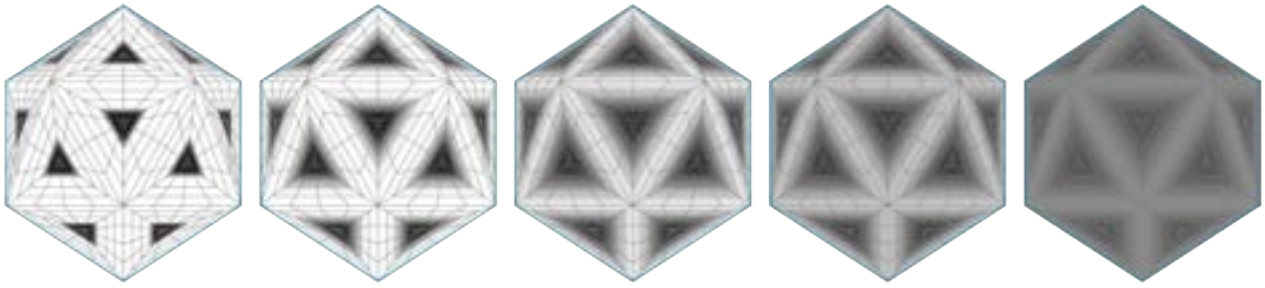


1. Initial box mesh with 3 faces removed
2. Smoothing after 2 iterations
3. 6 iterations
4. 25 iterations
5. 50 iterations

1.6.4.2 Blur

The **Blur** component acts in a similar way as smooth, except it only affect the vertex colors. It can also be used to reduce the jagged appearance of colored meshes, although to a lesser extent since it does not change any

geometry.

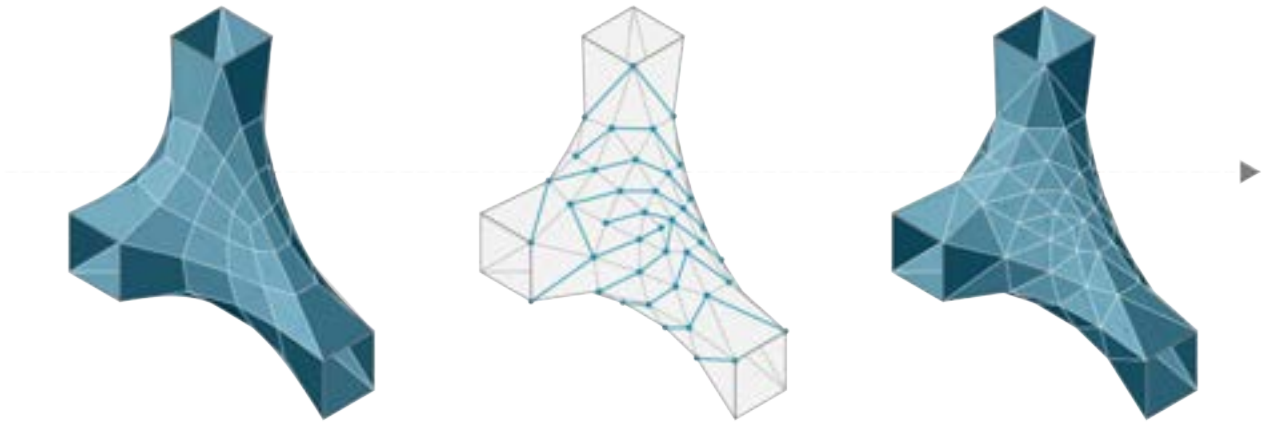


1 2 3 4 5

1. Initial mesh
2. Blur after 1 iterations
3. 6 iterations
4. 12 iterations
5. 20 iterations

1.6.4.3 Triangulate

In order to ensure each face is planar, or to export a mesh to a different software that might not allow quad faces, it is sometimes necessary to triangulate a mesh. Using the **Triangulate** component, each quad face is replaced with two triangle faces. Grasshopper always uses the shortest diagonal of the face to create a new edge.

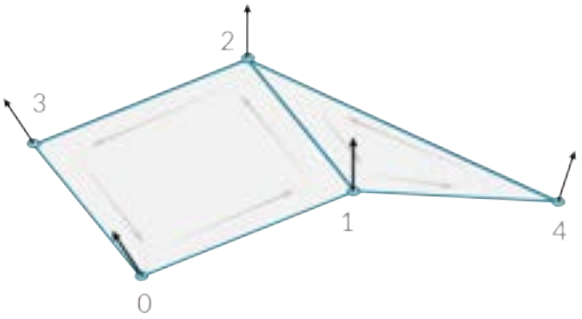


1 2 3

1. Original quad mesh
2. Added edges according to shortest distance across quads
3. Triangulated resultant mesh

1.6.4.4 Weld

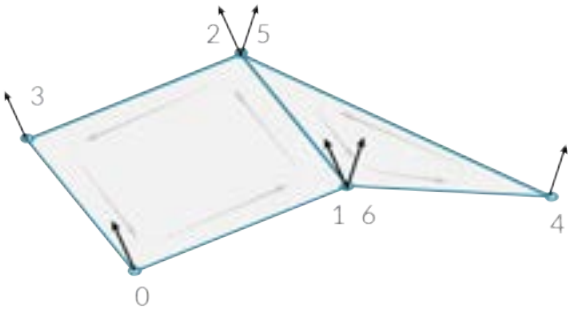
In the last section, we noticed that a single vertex can be shared by adjacent faces and the normal for that vertex is calculated as the average of the adjacent faces, allowing a smoother visualization. However, it is sometimes desireable to have a sharp crease or seam where one face does not smoothly transition to the next by way of the vertex normals. For this situation it is necessary for each face to have its own vertex with its own normal. The list of vertices would contain at least two points that have the same coordinates, but different indices.



1

Vertex List
0 = {0.0, 0.0, 0.0}
1 = {1.0, 0.0, 1.0}
2 = {1.0, 1.0, 1.0}
3 = {0.0, 1.0, 0.0}
4 = {2.0, 0.0, -1.0}

Face List
Q {0, 1, 2, 3}
T {1, 4, 2}



2

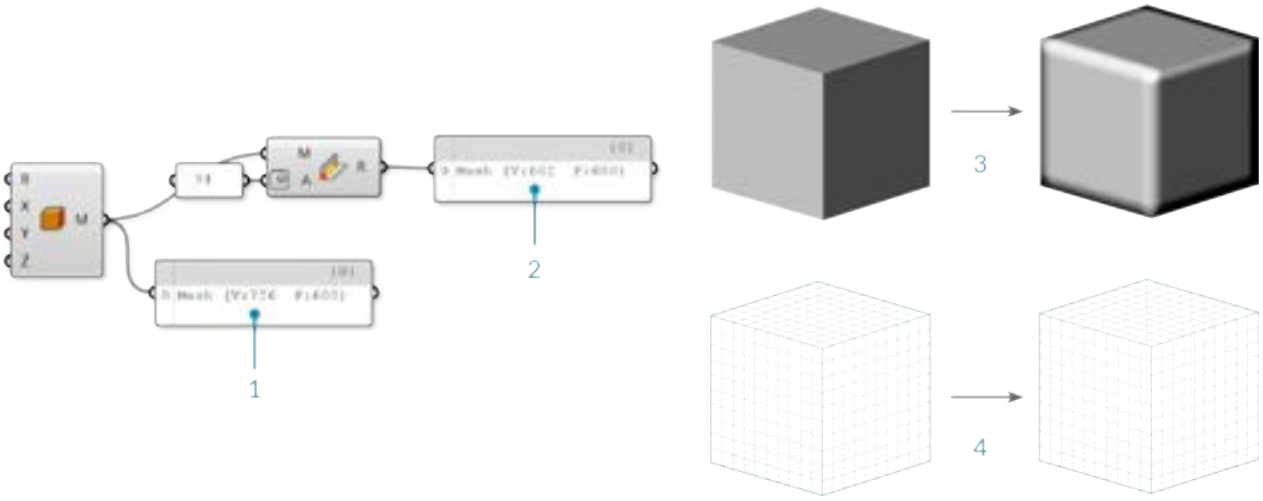
Vertex List
0 = {0.0, 0.0, 0.0}
1 = {1.0, 0.0, 1.0}
2 = {1.0, 1.0, 1.0}
3 = {0.0, 1.0, 0.0}
4 = {2.0, 0.0, -1.0}
5 = {1.0, 1.0, 1.0}
6 = {1.0, 0.0, 1.0}

Face List
Q {0, 1, 2, 3}
T {4, 5, 6}

1. Welded Faces - Both faces share vertices 1 and 2. The vertex normals at these vertices are the average of the face vertices.
2. Unwelded Faces - Duplicate vertices are added to the list. The faces do not share any vertex indices. Vertices 1 and 6, and vertices 2 and 5 have identical coordinates, but are separate vertices. They each have their own vertex normal

The process of taking two vertices that are in the same position and combining them into a single vertex is called *welding*, while *unwelding* takes a single vertex and splits it into multiple vertices.

The **Weld** component uses a threshold angle as input. Any two adjacent faces with an angle less than the threshold angle will be welded together, resulting in common vertices with a normal that is the average of the adjacent faces. **Unweld** works in the opposite manner, where adjancent faces with an angle greater than the given threshold will be unwelded, and their shared vertices will be duplicated.



1. The default Box Mesh has 726 vertices. The mesh is creased at the corners of the box, where vertices

> are doubled up.
> 2. If the mesh is welded with an angle greater than 90 degrees, the resulting mesh faces are welded together, and the number of vertices has decreased to 602 while the number of faces has remained the same.
> 3. Looking at the previewed geometry, you can also notice that the rendered welded mesh has smoothed corners.
> 4. Unlike the Smooth component which changes the mesh geometry, this mesh only appears smoother due to the vertex normal's role in rendering and shading. The actual positions of the vertices remain unchanged.

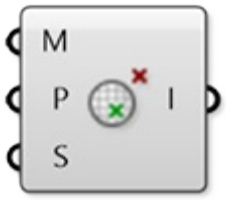
In the above image, we used the angle 91 degrees because we know that the sides of a square will be at 90 degree angles. To completely weld an entire mesh you should use an angle of 180 degrees.

1.6.5 Mesh Interactions

This section looks at ways in which Mesh Objects can interact with other objects, such as evaluating nearest points or combining multiple meshes together.

1.6.5.1 Mesh Geometry and Points

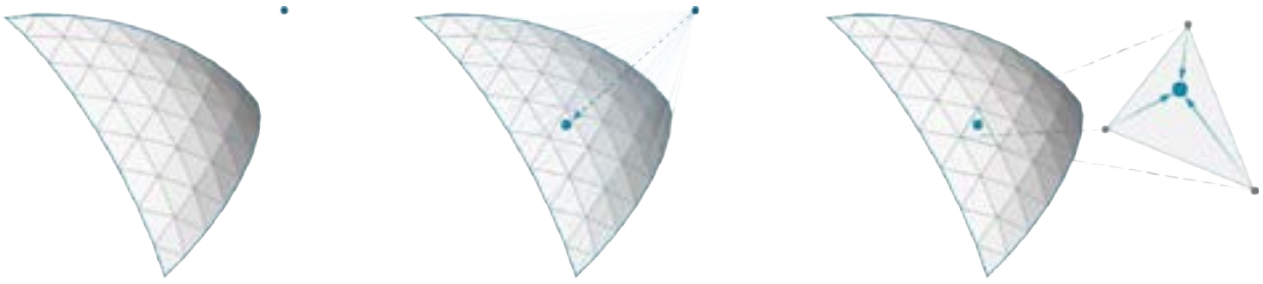
Inclusion



This component tests to determine whether a given point is inside a mesh solid or not. This only works with closed meshes.

Mesh Closest Point

This component will calculate the position on a mesh that is closest to a given point. This component outputs three pieces of data: the coordinates of the calculated point on the mesh, the index of the face which contains that point, and the mesh parameter. This parameter is extremely useful in conjunction with the **Mesh Eval** component discussed below.



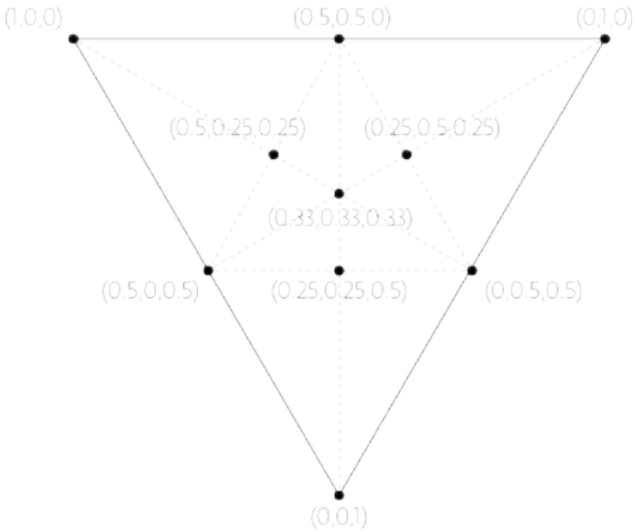
1 2 3

1. Given a point in space, We want to find the closet point on the mesh
2. The face that contains the closest point is identified
3. The parameters of the closest point on the face are calculated

For those users interested in a little more detail about how a mesh is parameterized, we can take a closer look at how a mesh parameter is structured. You can see this structure by attaching a panel to the parameter output of a **Mesh Closest Point** component. The mesh parameter has the form: N[A,B,C,D]. The first number, N, is the index of the face which contains the calculated point.

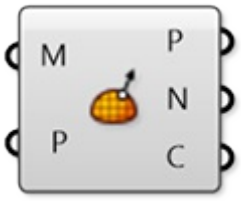
The following four numbers define the *barycentric* coordinates of the point within that face. The coordinates of the

referenced point can be found by multiplying each vertex of the face by these numbers in order and then add the results together. (Of course, this is already done for us, and is given in the Point output). Also note that barycentric coordinates are only unique for triangular faces, meaning that on a quad face the same point could have multiple different parameterizations. Grasshopper avoids this problem by internally triangulating a quad face when calculating a parameter, the result of which is that of the four numbers in a mesh parameter, at least one of them will always be zero.



Barycentric Coordinates

Mesh Eval



The **Mesh Eval** component uses a mesh parameter as an input and returns the referenced point, as well as the normal and color at that point. The color and normal are calculated as interpolations of the vertex colors and vertex normals, using the same barycentric coordinates as the mesh parameter.

1.6.5.2 Combining Mesh Geometry

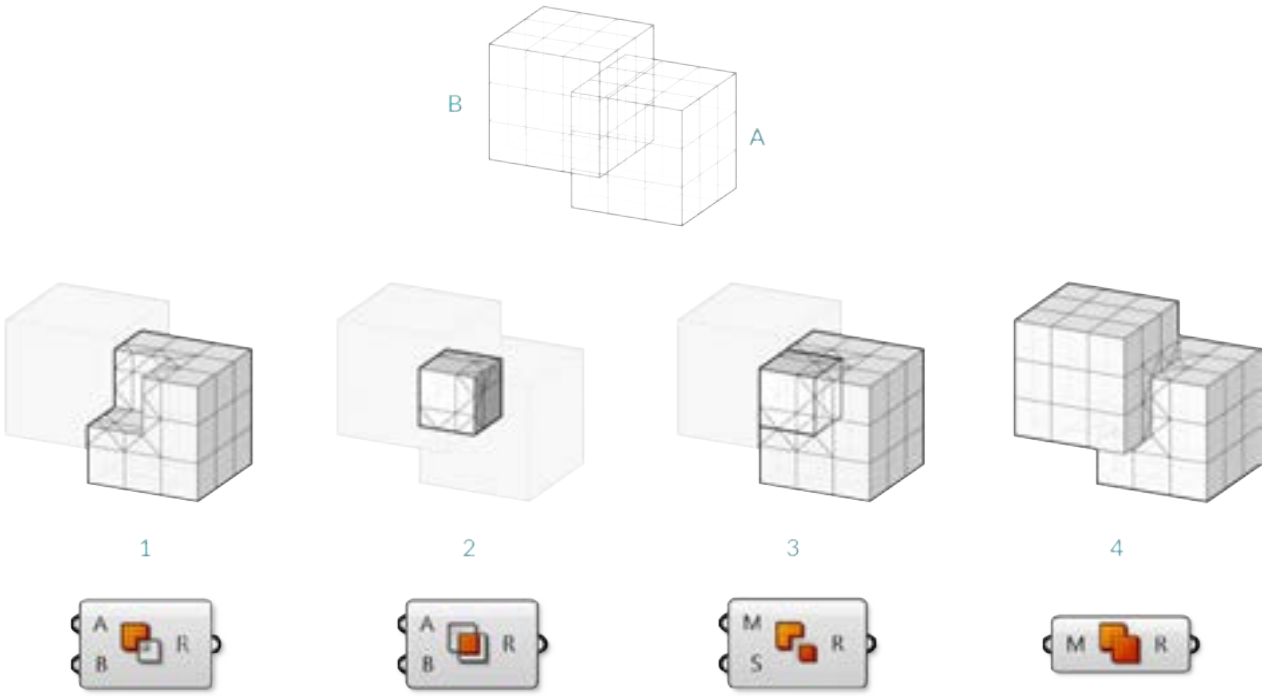
Mesh Join

Unlike joining curves or NURBS surfaces which require adjacency, any meshes can be joined into a single mesh - even meshes that are not touching. Recall that a mesh is simply a list of vertices, and a list of faces. There is no actual requirement for those faces to be connected (Although in most applications, such a mesh would not be very desirable!!).

This component does not weld mesh vertices together, so it is often useful to use this in combination with a **Weld** component.

Mesh Boolean

Meshes in Grasshopper have a set of boolean operations similar to boolean operations for NURBS solids. Boolean operations are order specific, meaning that switching the order of the input meshes Aand B will result in different outputs.

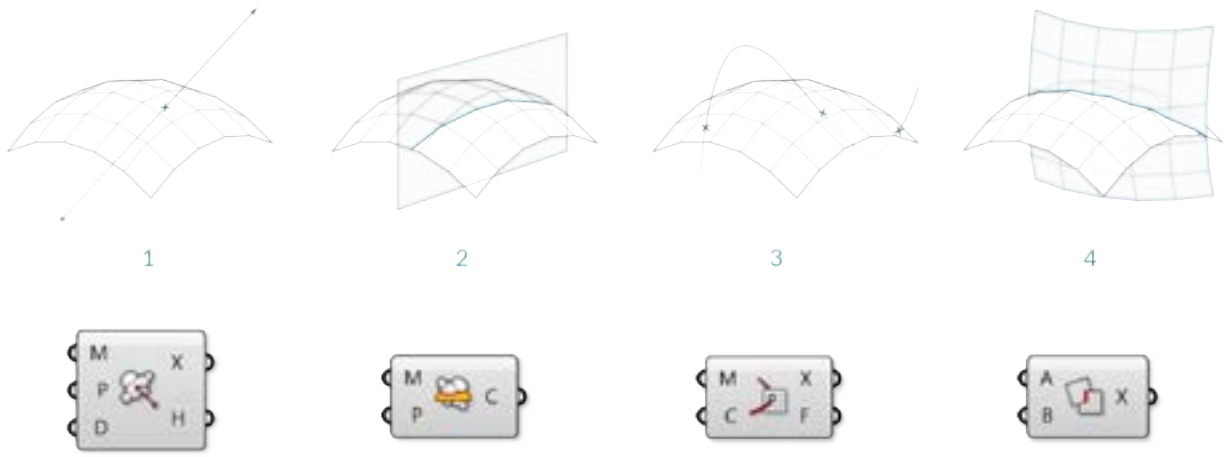


1. Mesh Difference
2. Mesh Intersection
3. Mesh Split
4. Mesh Union

1.6.5.3 Intersections and Occlusions

Intersect

Intersections can be calculated between meshes and other objects: rays, planes, curves, and other meshes

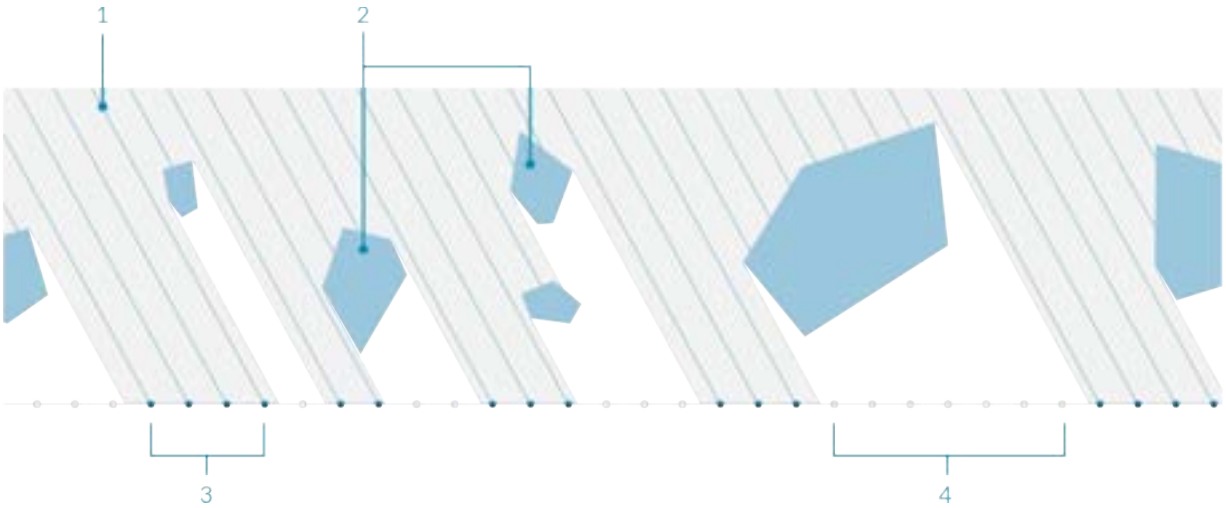


1. Mesh | Ray
2. Mesh | Plane
3. Mesh | Curve
4. Mesh | Mesh

Occlusion

As we have discussed, one of the (many) uses of mesh geometry is for visualizations and creating shaded rendering based on face normals. When rendering, it also necessary to know when an object is in shadow behind another object. The **Occlusion** component in Grasshopper allows us to enter a set of sample points, along with occluding mesh geometry that will 'cast shadows', and a *view ray*, or vector, to indicate the direction that 'light' is coming from.

Such a process can be used to create shadows in rendering, or determine whether objects are hidden from a certain camera view.



1. View Ray to test for occlusion
2. Occluding mesh geometry
3. 'Hit' sample points
4. 'Occluded' sample points

1.6.6 Working with Mesh Geometry

In this section, we will work through an exercise file for producing a complete mesh solid. By the end of this exercise, we will have a dynamic definition to produce custom vases that can be 3D printed.

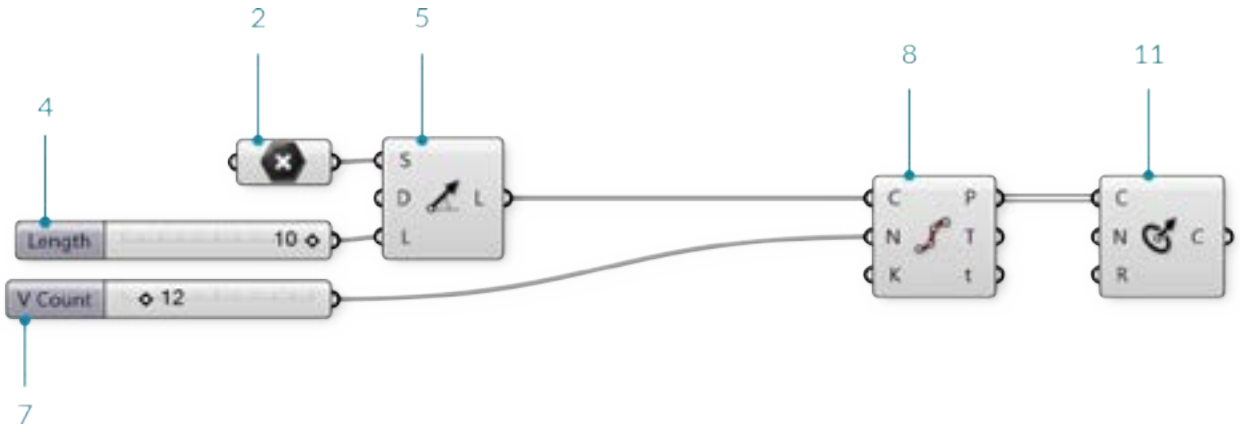
> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

Since this definition is somewhat longer than previous examples in this primer, we will first walk through the basic steps we will take:

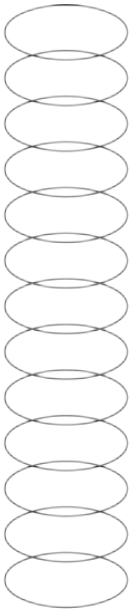
1. Create a series of circles as a base cylinder
2. Use a Graph Mapper component to define the profile of our vase
3. Construct the topology of the mesh faces to produce a single mesh surface
4. Cap the bottom of the mesh
5. Introduce a twist to the vertical orientation for a more dynamic form
6. Add corrugated ridges for a textured vase
7. Offset the mesh surface to give the vase thickness
8. Cap the top gap between the two surfaces to produce a closed solid

01.	Start a new definition, type Ctrl-N (in Grasshopper)	
02.	**Params/Geometry/Point** - Drag and drop a **Point** container onto the canvas	
03.	Reference a point in Rhino by right-clicking the **Point** component and selecting "Set one point". This will serve as the origin point of our vase. > You can create a point manually in Grasshopper by double-clicking the canvas to bring up the search window, then typing the coordinates of the point separated by commas, such as: '0,0,0' (without quotes)	
04.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Name: Length Lower Limit: 1 Upper Limit: 10	
05.	**Curve/Primitive/Line SDL** - Drag and drop a **Line SDL** component onto the canvas	
06.	Connect the **Point** component to the Start (S) input of the **Line SDL** component, and connect the **Number Slider** to the Length (L) input. > The default Direction (D) value of **Line SDL** is the Unit Z vector, which is what we will use for this example	
	Params/Input/Number Slider - Drag and drop a **Number Slider** onto the canvas and set the following values: Name: V Count	

07.	Rounding: Integer Lower Limit: 1 Upper Limit: 100	
08.	**Curve/Division/Divide Curve** - Drag and drop a **Divide Curve** component onto the canvas	
09.	Connect the Line (L) output of the **Line SDL** component to the Curve (C) input of the **Divide Curve** component	
10.	Connect the **V Count** number slider to the Count (N) input of the **Divide Curve** component	
11.	**Curve/Primitive/Circle CNR** - Drag and drop a **Circle CNR** component onto the canvas	
12.	Connect the Points (P) output of the **Divide Curve** component to the Center (C) input of the **Circle CNR** component	



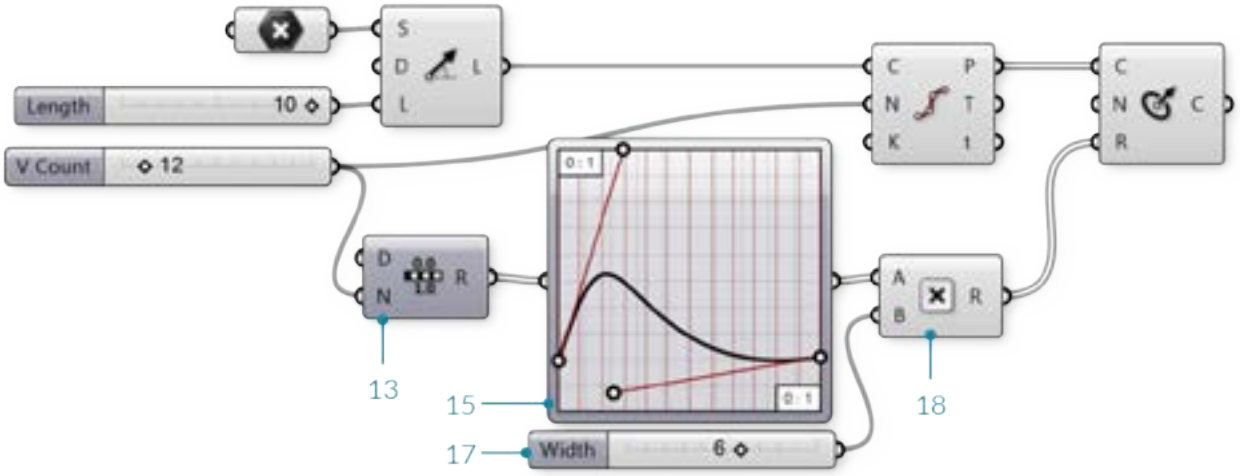
> We have a series of circles stacked vertically. We will use these to make the profile of our vase.



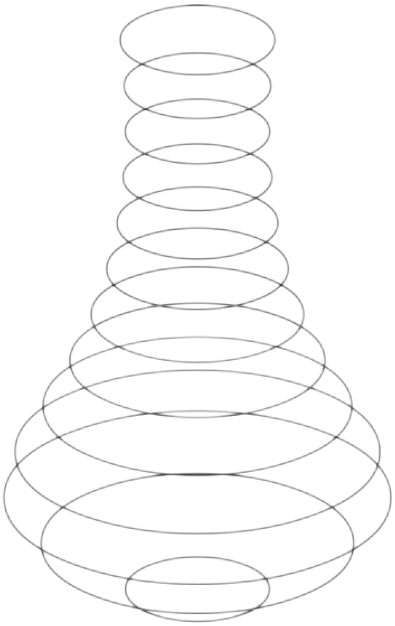
Next, we will use a Graph Mapper to control the radii of the circles.

13.	**Sets/Sequence/Range** - Drag and drop a **Range** component onto the canvas	
14.	Connect the **V Count** number slider to the Steps (N) input of the **Range** component	

15.	**Params/Input/Graph Mapper** - Drag and drop a **Graph Mapper** component onto the canvas	
16.	Right-click the **Graph Mapper**, click 'Graph Types' from the menu and select 'Bezier'	
17.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Name: Width Lower Limit: 0 Upper Limit: 10	
18.	**Maths/Operators/Multiplication** - Drag and drop a **Multiplication** component onto the canvas	
19.	Connect the **Graph Mapper** and the **Width** number slider to the Aand B inputs of the **Multiplication** component	
20.	Connect the Result (R) output of the **Multiplication** component to the Radius (R) input of the **Circle CNR** component	



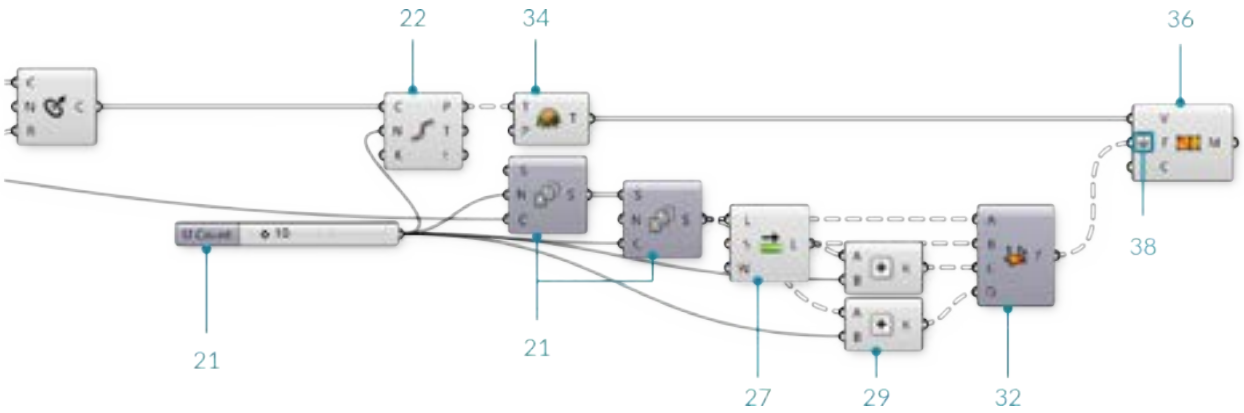
> Use the handles on the **Graph Mapper** to adjust the profile of the circles.
>
> NOTE: It is important to make sure the start point of the bezier curve on the **Graph Mapper** is not at zero. By lifting the start point to a number greater than zero, we produce a flat base for our vase.



We now have a profile for our vase. Next, we will construct a mesh surface. This will require creating mesh vertices and defining mesh faces according to the index of those vertices.

21.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Name: U Count Rounding: Even Lower Limit: 2 Upper Limit: 100	
22.	**Curve/Division/Divide Curve** - Drag and drop a **Divide Curve** component onto the canvas	
23.	Connect the Circle (C) output of the **Circle CNR** component to the the Curve (C) input of the **Divide Curve** component, and connect the **U Count** number slider to the Count (N) input > The Points (P) output of this component are the vertices we will use for our mesh	
24.	**Sets/Sequence/Series** - Drag and drop two **Series** components onto the canvas	
25.	Connect the **U Count** number slider to the Step (N) input of the first **Series** component, and connect the **V Count** number slider to the Count (C) input of the same **Series** component	
26.	Connect the Series (S) output of the first **Series** component to the Start (S) input of the second **Series** component, and connect the **U Count** number slider to the Count (C) input	
27.	**Sets/List/Shift List** - Drag and drop a **Shift List** component onto the canvas	
28.	Connect the output of the second **Series** component to the List (L) input of the **Shift List** component	
29.	**Maths/Operators/Addition** - Drag and drop two **Addition** components onto the canvas	
30.	Connect the output of the second **Series** component and the **U Count** number slider to the Aand B inputs of the first **Addition** component	
31.	Connect the output of the **Shift List** component and the **U Count** number slider to the Aand B inputs of the second **Addition** component	
32.	**Mesh/Primitive/Mesh Quad** - Drag and drop a **Mesh Quad** component onto the canvas	
33.	Connect the following to the inputs of the **Mesh Quad** component: A- Second **Series** component B - **Shift List** C - First **Addition** component D - Second **Addition** component > We have just create the initial topology for our mesh. These faces will be combined with the vertices. The order of these connections is crucial, so go ahead and double check all the connections at this point!	
34.	**Sets/Tree/Flatten** - Drag and drop a **Flatten Tree** component onto tha canvas	
35.	Connect the Points (P) output of the **Divide Curve** component to the Tree (T) input of the **Flatten Tree** Component	
36.	**Mesh/Primitive/Construct Mesh** - Drag and drop a **Construct Mesh** component onto the canvas	

37.	Connect the Tree (T) output of the **Flatten Tree** component to the Vertices (V) input of the **Construct Mesh** component	
38.	Connect the Face (F) output of the **Mesh Quad** component to the Faces (F) input of the **Construct Mesh** component. Right-click the F (Faces) input and select 'Flatten'	

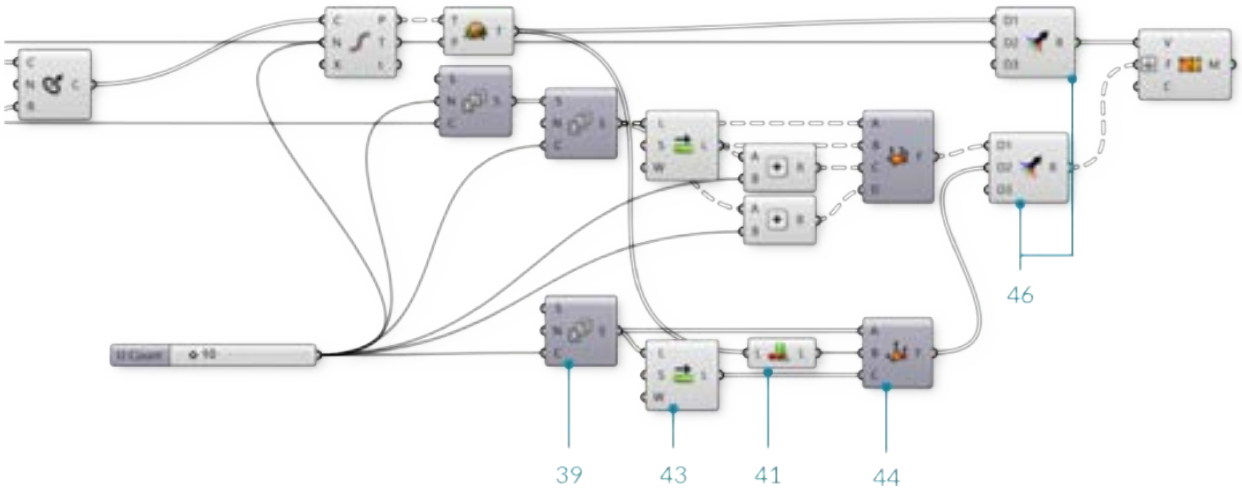


> We now have a mesh surface for our vase.

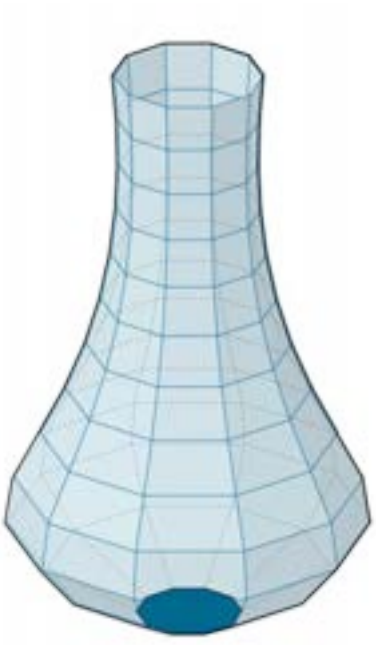
Next we will close the bottom of the vase. To do this, we will add the original origin point to our list of vertices, and then construct triangle mesh faces from the bottom edge to that point.

39.	**Sets/Sequence/Series** - Drag and drop a **Series** component onto the canvas	
40.	Connect the **U Count** number slider to the Count (C) input of the **Series** component	
41.	**Sets/List/List Length** - Drag and drop a **List Length** component onto the canvas	
42.	Connect the Tree (T) output of the **Flatten Tree** component to the List (L) input of the **List Length** component > This will be the index of the origin point after we add it to the existing list of vertices.	
43.	**Sets/List/Shift List** - Drag and drop a **Shift List** component onto the canvas	
	Mesh/Primitive/Mesh Triangle - Drag and drop a **Mesh Triangle** component onto the	

44.	canvas	
45.	Connect the following to the inputs of the **Mesh Triangle** component: A- Newest **Series** component B - **List Length** C - **Shift List**	
46.	**Sets/Tree/Merge** - Drag and drop two **Merge** components onto the canvas	
47.	Connect the Tree (T) output of the **Flatten Tree** component to the D1 input, and connect the initial **Point** component to the D2 input of the first **Merge** component	
48.	Connect the Faces (F) of the **Mesh Quad** component to the D1 input, and connect the **Mesh Triangle** output to the D2 input of the second **Merge** component	
49.	Connect the first **Merge** component to the Vertices (V) input of the **Construct Mesh** component, and connect the second **Merge** component to the Faces (F) input of the **Construct Mesh** component.	

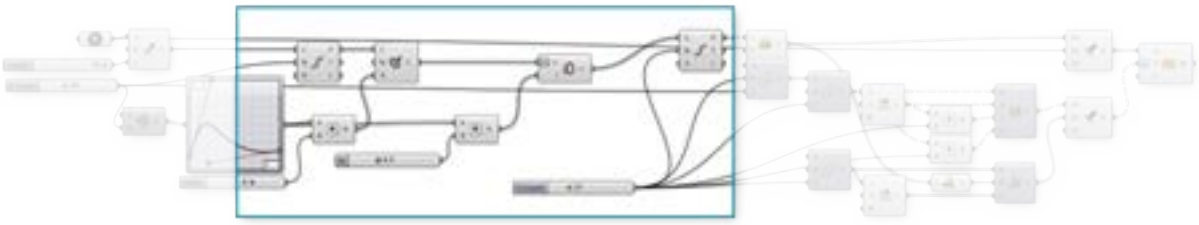


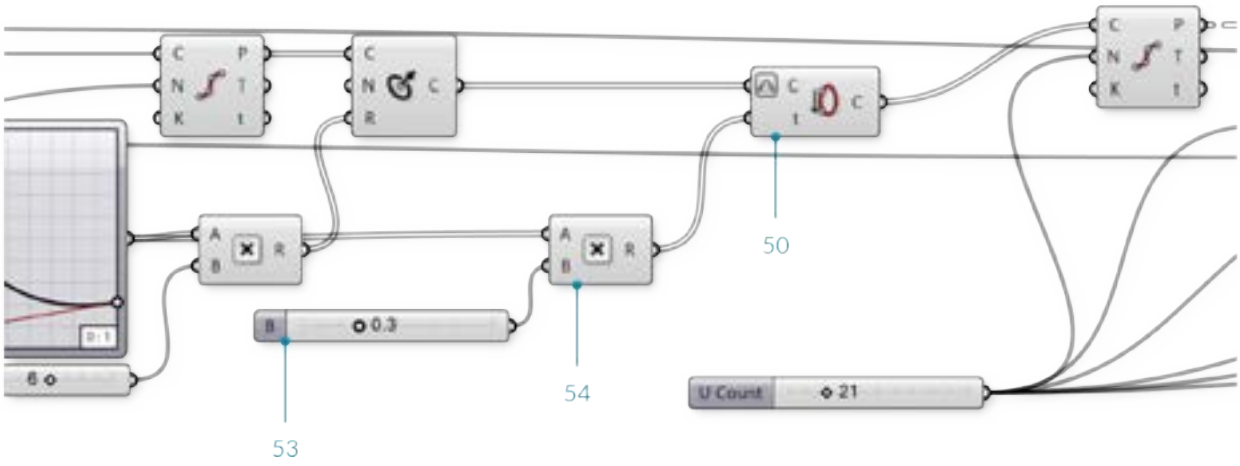
> We have capped the bottom of the vase with triangle mesh faces.



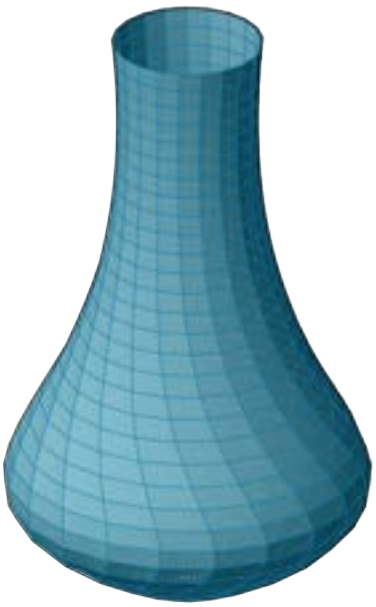
We will now add some detailing to the vase. We will start by adding a curve to the vertical direction by adjusting the *seam* of the original circles

50.	**Curve/Util/Seam** - Drag and drop a **Seam** component onto the canvas	
51.	Connect the Circle (C) output of the **Circle CNR** component to the Curve (C) input of the **Seam** component	
52.	Right click the Curve (C) input of the **Seam** component and select 'Reparameterize'	
53.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas. We will use the default settings for this slider	
54.	**Maths/Operator/Multiplication** - Drag and drop a **Multiplication** component onto the canvas.	
55.	Connect the output from the **Graph Mapper** to the Ainput, and the newest **Number Slider** to the B input of the **Multiplication** component	
56.	Connect the Result (R) of the **Multiplication** component to the Parameter (t) input of the **Seam** component	





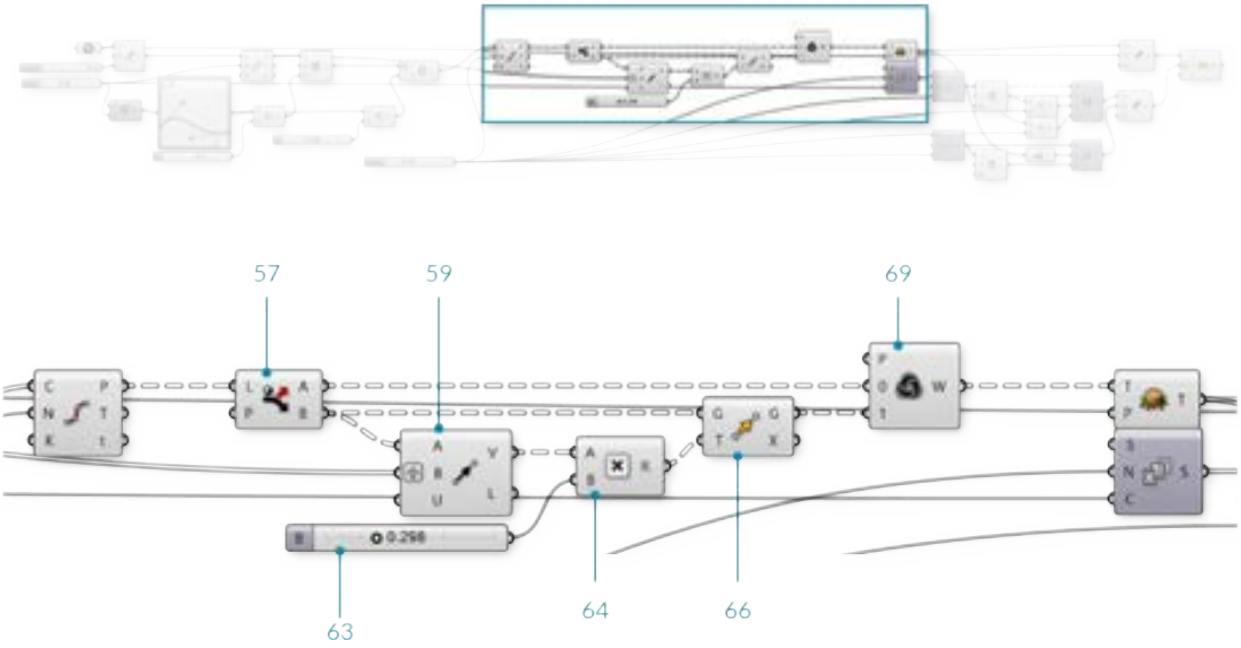
> The curvature is achieved by changing the *seam* position of the initial circles, and uses the same Graph Mapper as the vase profile.



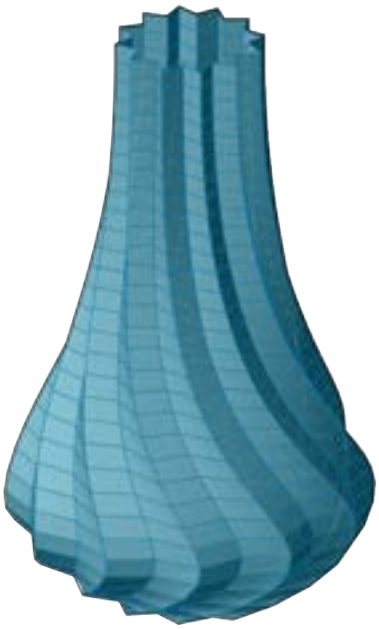
Next we will add some vertical ridges to the vase.

57.	**Sets/List/Dispatch** - Drag and drop a **Dispatch** component onto the canvas	
58.	Connect the Point (P) output of the second **Divide Curve** component to the List (L) input of the **Dispatch** component > We are using the default Pattern (P) input of the **Dispatch** component to separate the points into two lists with alternating points	
59.	**Vector/Vector/Vector 2Pt** - Drag and drop a **Vector 2Pt** component onto the canvas	
60.	Connect the B output of the **Dispatch** component to the Ainput of the **Vector 2Pt** component	
61.	Connect the Points (P) of the *first* **Divide Curve** component to the B input of the **Vector 2Pt** component	
62.	Right-click the B input of the **Vector 2Pt** component and select 'Graft', and right-click the Unitize (U) input, go to 'Set Boolean' and select 'True' > This creates a unit vector for each point that points towards the center of the circle	
63.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas. We will use the default settings	
64.	**Maths/Operator/Multiplication** - Drag and drop a **Multiplication** component onto the canvas	
65.	Connect the Vector (V) output of the **Vector 2Pt** component to the Ainput, and connect the **Number Slider** to the B input of the **Multiplication** component	
66.	**Transform/Euclidean/Move** - Drag and drop a **Move** component onto the canvas	
67.	Connect the B output of the **Dispatch** component to the Geometry (G) input of the **Move** component	
68.	Connect the Result (R) output of the **Multiplication** component to the Motion (T) input of the **Move** component	

69.	**Sets/List/Weave** - Drag and drop a **Weave** component onto the canvas	
70.	Connect the Aoutput of the **Dispatch** component to the 0 input of the **Weave** component	
71.	Connect the Geometry (G) output of the **Move** component to the 1 input of the **Weave** component	
72.	Connect the Weave (W) output of the **Weave** component to the Tree (T) input of the **Flatten Tree** component	

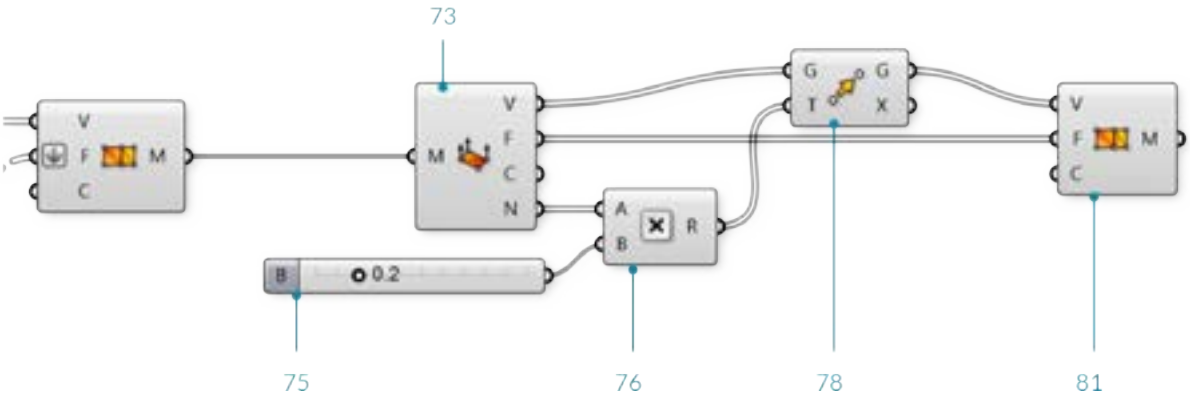


> Remember to go back and adjust your sliders and graph mapper to see how the model changes, and to make sure everything still works. This is known as 'flexing' the model, and should be done frequently to check for mistakes in the definition.

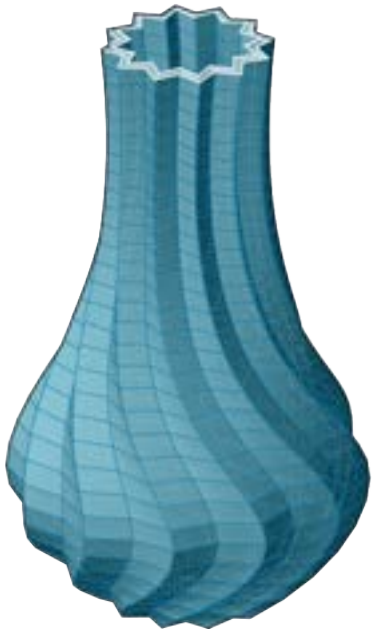


We now have a single surface for our vase. If we wanted to print this vase using a 3D printer, we need it to be a closed solid. We will create a solid by offsetting the current mesh, then combining the original mesh and the offset mesh.

73.	**Mesh/Analysis/Deconstruct Mesh** - Drag and drop a **Deconstruct Mesh** component onto the canvas	
74.	Connect the Mesh (M) output of the **Construct Mesh** component to the Mesh (M) input of the **Deconstruct Mesh** component	
75.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas. We will use the default settings	
76.	**Maths/Operator/Multiplication** - Drag and drop a **Multiplication** component onto the canvas	
77.	Connect the Normals (N) output of the **Deconstruct Mesh** component to the Ainput, and connect the **Number Slider** to the B input of the **Multiplication** component	
78.	**Transform/Euclidean/Move** - Drag and drop a **Move** component onto the canvas	
79.	Connect the Vertices (V) output of the **Deconstruct Mesh** component to Geometry (G) input of the **Move** component	
80.	Connect the Result (R) output of the **Multiplication** component to the Motion (T) input of the **Move** component	
81.	**Mesh/Primitive/Construct Mesh** Drag and drop a **Construct Mesh** component onto the canvas	
82.	Connect the Geometry (G) output of the **Move** component to the Vertices (V) input of the **Construct Mesh** component	
83.	Connect the Faces (F) output of the **Deconstruct Mesh** component to the Face (F) input of the **Construct Mesh** component	



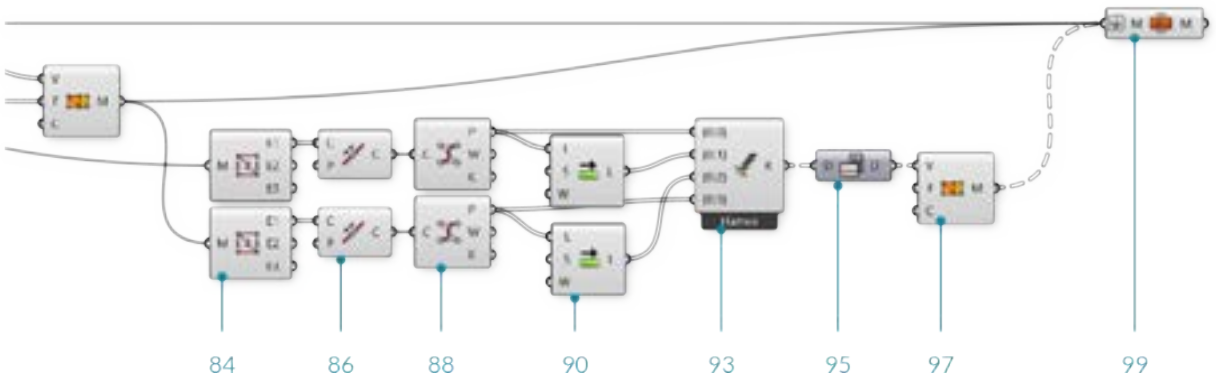
> By offseting the mesh according to the vertex normals, we now have an 'inside' and an 'outside' mesh, but we still have a gap at the top between the two mesh geometries

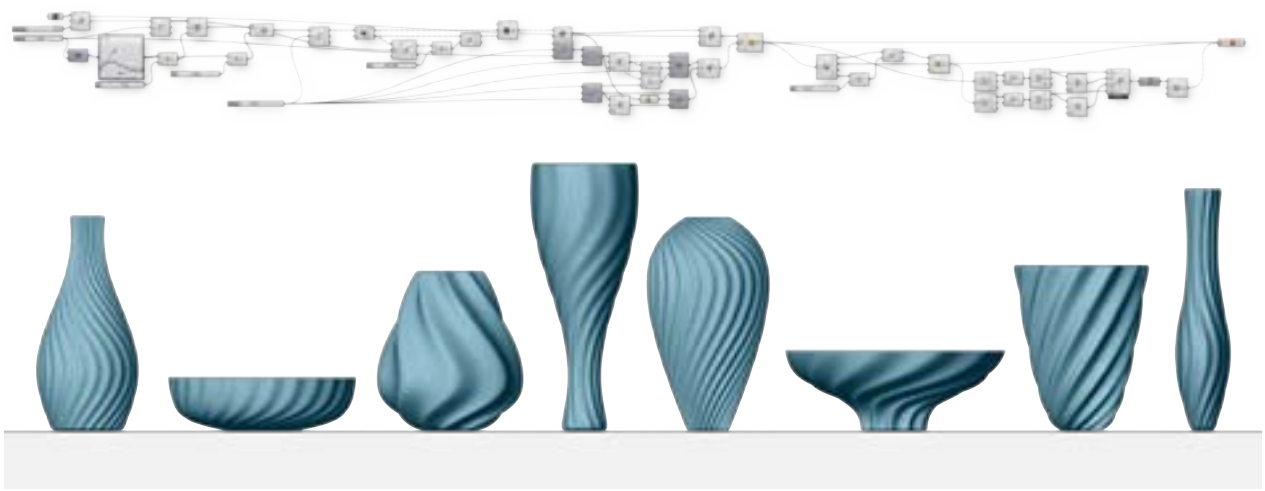


The final step will be to create a closed mesh by creating a new mesh geometry to close the gap and then joining the meshes together.

84.	**Mesh/Analysis/Mesh Edges** - Drag and drop a **Mesh Edges** component onto the canvas	
85.	Connect the Mesh (M) output of the first **Construct Mesh** component to the Mesh (M) input of the **Mesh Edges** component	
86.	**Curve/Util/Join Curves** - Drag and drop a **Join Curves** component onto the canvas	
87.	Connect the Naked Edges (E1) output of the **Mesh Edges** component to the Curves (C) input of the **Join Curves** component	
88.	**Curve/Analysis/Control Points** - Drag and drop a **Control Points** component onto the canvas	
89.	Connect the Curves (C) output of the **Join Curves** component to the Curve (C) input of the **Control Points** component > By joining the curves and then extrating the control points, we ensure that the order of the points is consistent along the rim of the vase, which is important for making the resulting mesh orientable and manifold	
90.	**Sets/List/Shift List** - Drag and drop a **Shift List** component onto the canvas	
91.	Connect the Points (P) output of the **Control Points** component to the List (L) input of the **Shift List** component	
92.	Repeat steps 84 through 91 for the second **Construct Mesh** component	
93.	**Sets/Tree/Entwine** - Drag and drop an **Entwine** component onto the canvas	
94.	Zoom in to the **Entwine** component to show the option to add an extra input. We will need four inputs. Connect them in the following way: {0;0} - Points (P) from first **Control Points** component {0;1} - output from first **Shift List** {0;2} - output from second **Shift List** {0;3} - Points (P) from second **Control Points** component	

95.	**Sets/Tree/Flip Matrix** - Drag and drop a **Flip Matrix** component onto the canvas	
96.	Connect the Result (R) from the **Entwine** component to the Data (D) input of the **Flip Matrix** component	
97.	**Mesh/Primitive/Construct Mesh** - Drag and drop a **Construct Mesh** component onto the canvas	
98.	Connect the Data (D) outut of the **Flip Matrix** component to the Vertices (V) input of the **Construct Mesh** component	
99.	**Mesh/Util/Mesh Join** - Drag and drop a **Mesh Join** component onto the canvas	
100.	Connect all three **Construct Mesh** components to the **Mesh Join** component by holding down the Shift key while connecting the wires (or use a **Merge** component). Right-click the Mesh (M) input of the **Mesh Join** component and select 'Flatten'	



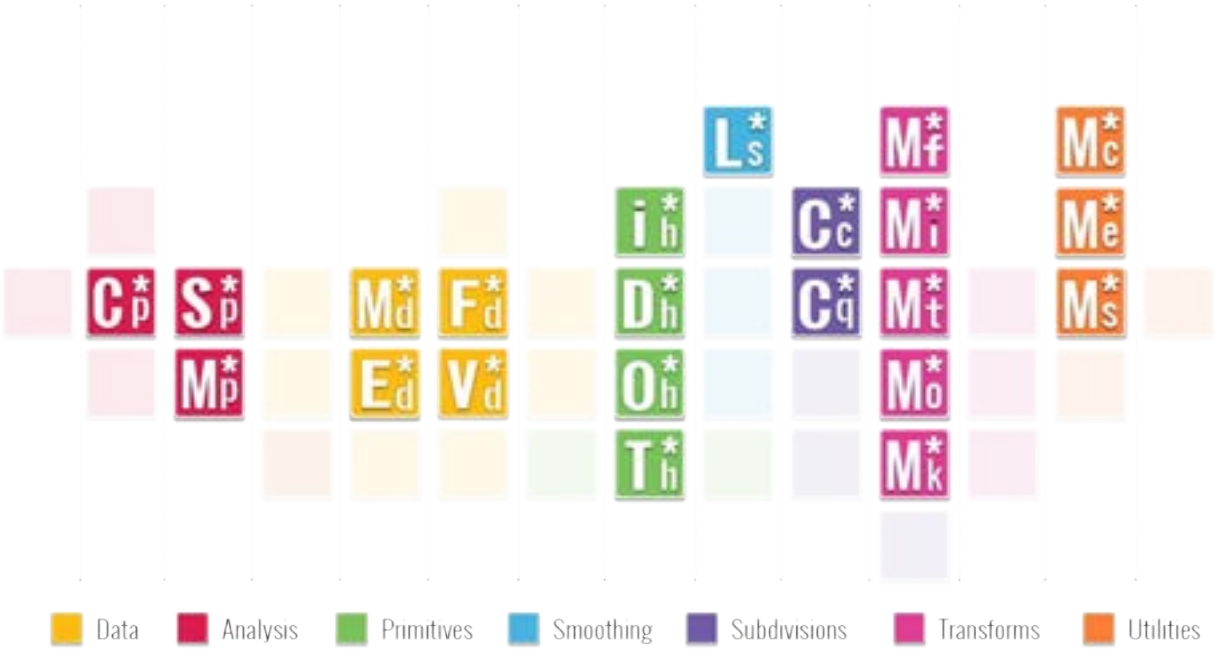


2. Extensions

Foundations are meant to be built upon. This volume features an array of key plugins for Grasshopper that will extend the application's functionality *and* your ability to take your designs further.

2.1. Element*

Element* is a mesh geometry plugin for Grasshopper, enabling mesh creation, analysis, transformation, subdivision, and smoothing. Element* provides access to mesh topology data using the Plankton half-edge data structure for polygon meshes.

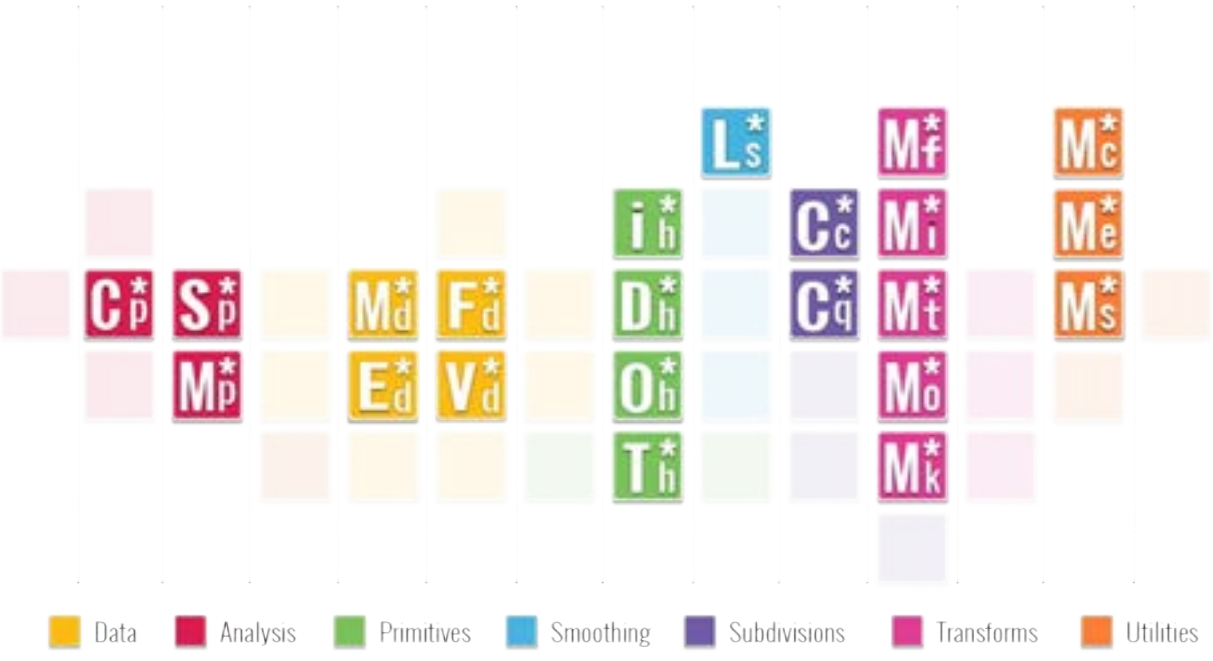


2.1.1. Element*

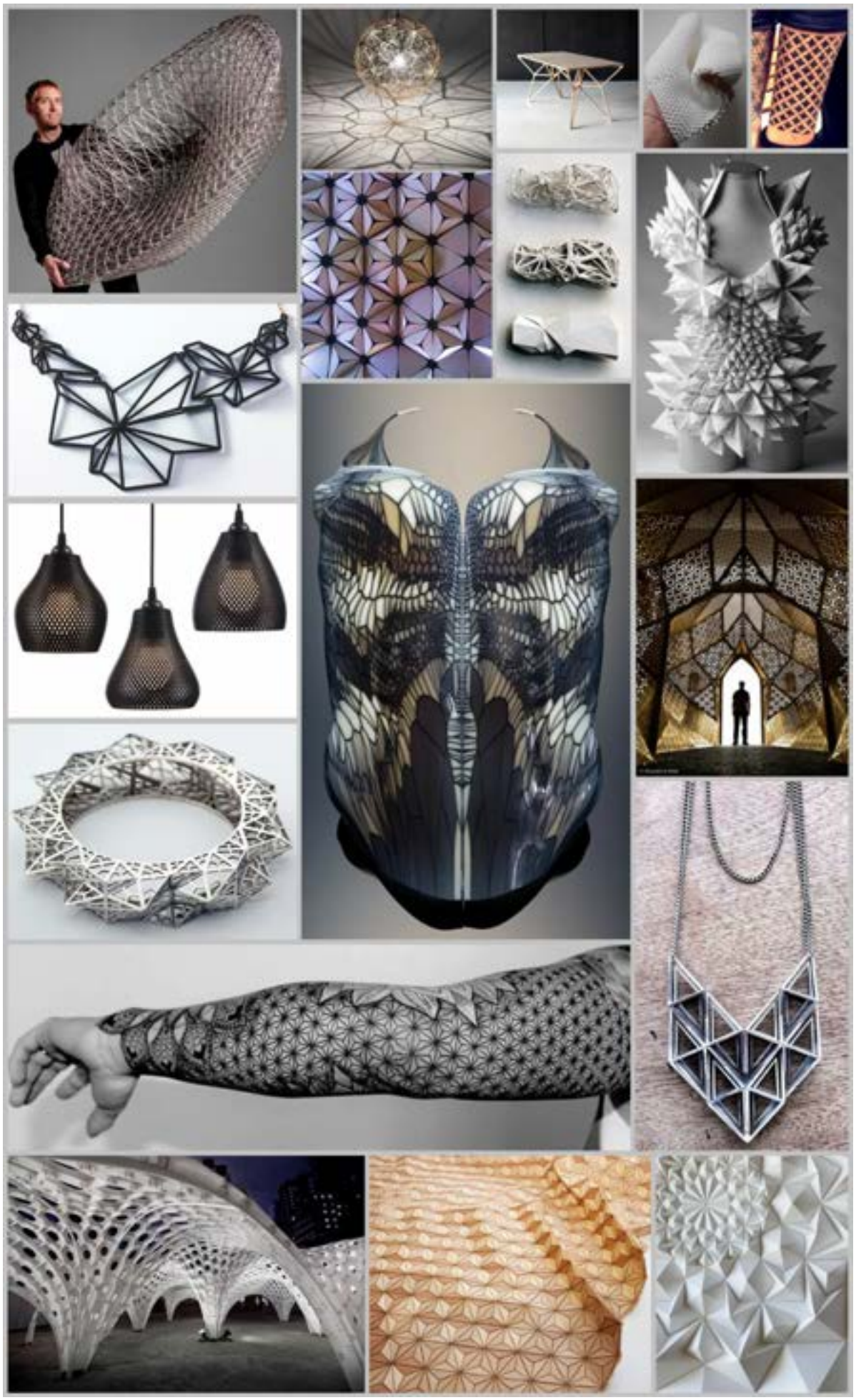
Integrating the use of Meshes in your workflow offers a wide variety of opportunities to create shapes that range from faceted to smooth. Element* allows you to go further, by giving you more intuitive access to analyzing mesh topology and smoothing routines. This chapter is a User's Guide for Element* Version 1.1 and will get you up to speed.

> Download the Element* plug-in to get started

Element* components are categorized based on their operations. Much like the periodic table, which provides a framework for analyzing chemical behavior, Element provides a framework for analyzing and exloring geometry based on mesh data and operations. We imagine new components will be created by analysing the relationships between components in each category.



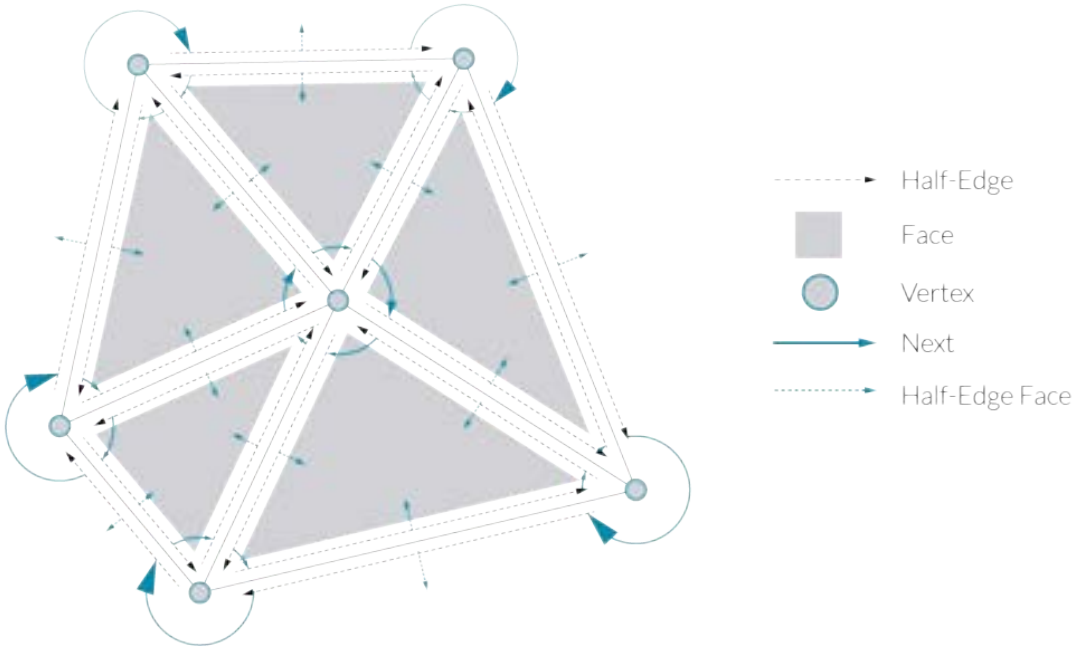
Below are some inspirational images of the types of products and applications that could be generated using Element*.



2.1.2. Half Edge Data

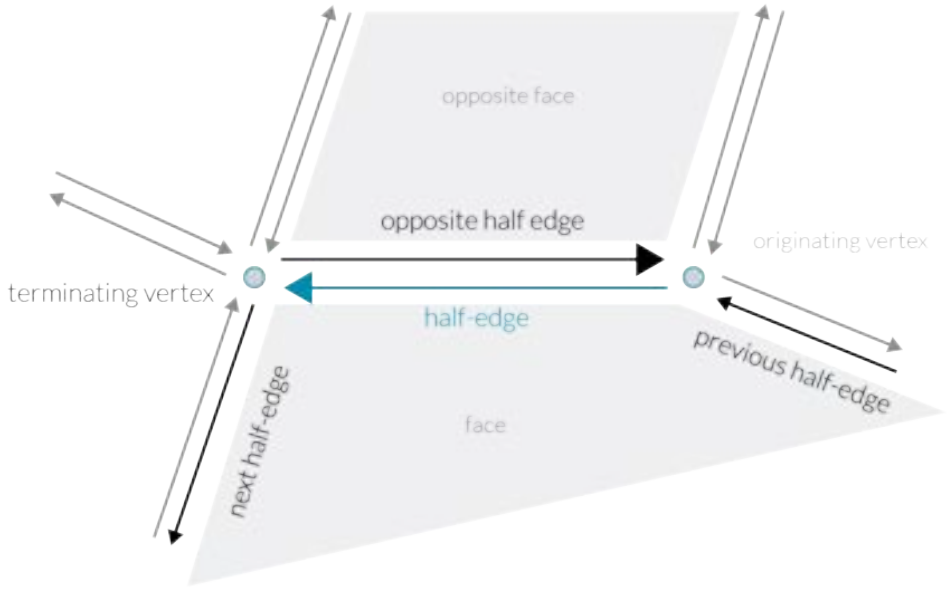
In the Grasshopper primer, we looked at how Grasshopper defines a mesh using a Face-Vertex data structure. This is a relatively simple data structure and is widely used in applications that use meshes, but can be computationally inefficient for more advanced algorithms. The Element* add-on restructures the mesh using Half-Edge data, an edge-centered data structure, which allows for efficient queries of adjacent vertices, faces, and edges, which can vastly improve on algorithm speed and performance. This structure is capable of maintaining incidence information of vertices, edges and faces. This method facilitates the creation of new patterns and geometries all based on the topological relationship of the base geometry.

The half-edge data structure is a representation for a mesh in which each edge is split up into two half-edges with opposite directions. This allows explicit and implicit access to data from one mesh element to adjacent elements.



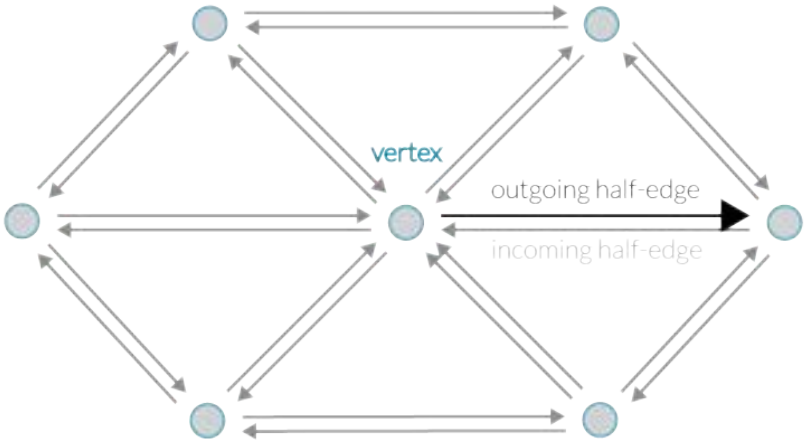
2.1.2.1 Half-Edge Connectivity

The half-edge highlighted in blue explicitly stores indices to its termination point, adjacent half-edges, and the face it belongs to. The other information (gray) can be accessed implicitly.



2.1.2.2 Vertex Connectivity

The vertex highlighted in blue explicitly stores an index to one of its outgoing half-edges. The other information (gray) can be accessed implicitly.



2.1.3. Element* Components

2.1.3.1 Analyse

1

2

3

1. Mesh Closest Point
2. Mesh Evaluate
3. Mesh Sample Plus

Element* Mesh Closest Point

Unlike Grasshopper's **Mesh Closest Point** component, this component also calculates the normal and color at the outputed point, eliminating the need for a **Mesh Eval** component and simplfying the canvas workspace.

Element* Mesh Evaluate

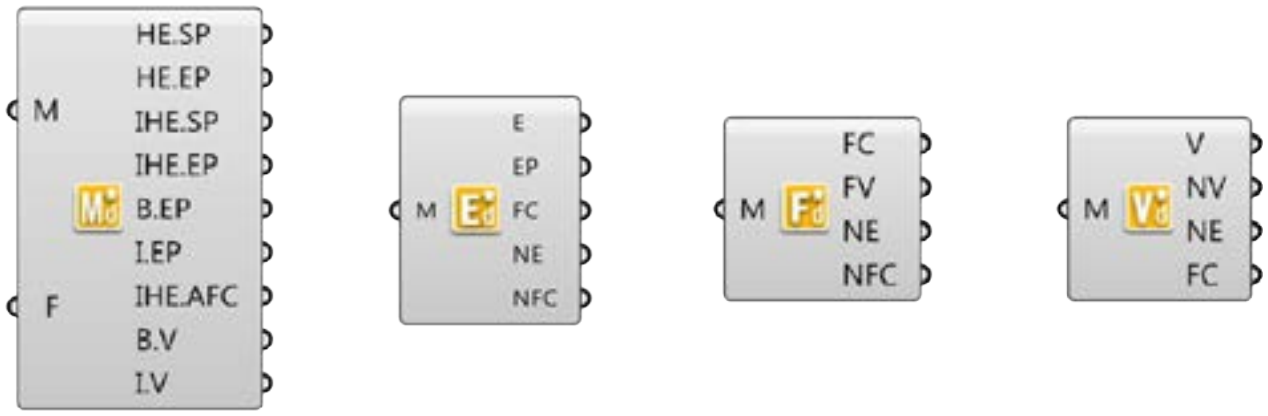
The built in Grasshopper **Mesh Eval** component requires a mesh parameter as an input, which can be extracted from a **Mesh Closest Point** component, but which can be difficult to construct manually. Element's closest point component allows direct input of the index of a mesh face and the barycentric coordinates.

Note - barycentric coordinates are defined such that they always add to 1. If the input values of U,V, and W do not add to 1, this component will maintain the ratio of the three values while normalizing them. For example, if you had the input values of 2, 2, and 4 the mesh parameter would be calculated as {0.25;0.25;0.5}

Element* Mesh Sample Plus

This component is used to quickly extract color information from a mesh. It returns the Alpha, Red, Green, Blue, Hue, Saturation, and Luminosity values of the inputed points. If the given points are not on the mesh, this component will calculate the closest point. This component uses Parallel Computing for speed.

2.1.3.2 Data



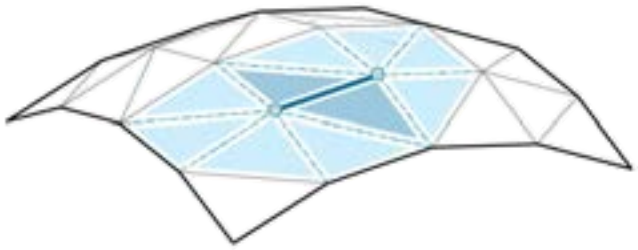
1. Data Visualizer
2. Edge Neighbors
3. Face Neighbors
4. Vertex Neighbors

Element* Data Visualizer

This component is used to help visualize the half-edge data of the faces of an input mesh.

Element* Edge Neighbors

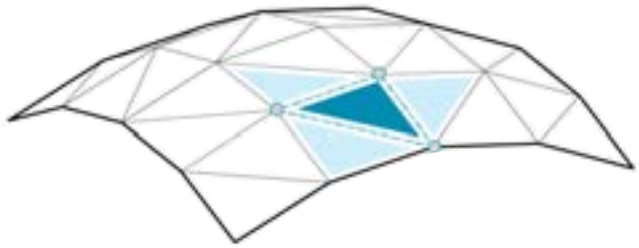
This component provides access to the adjacency data structured according to the edges of the input mesh. The output data is provided as a tree with one branch for each edge in the mesh. It returns the mesh edges, the edge end points, center ponts of the faces adjacent to each edge (dual), the neighboring edges as line objects (arranged in clockwise order) , and neighbouring face centers (center points of faces adjacent to the edge start and end points)



Edge Neighbors - Edges, End vertices, Adjacent face centers, Neighboring edges, and Neighbouring face centers

Element* Face Neighbors

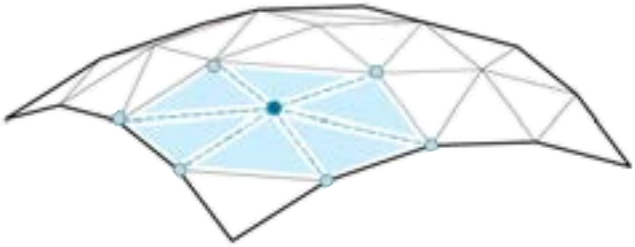
This component is similar to the others in this section, but the data is organized in a tree according to the faces of the mesh, with one branch per face. The outputs are the face centers, vertices of each face (arranged in counter clockwise order), neighbouring edges (arranged in counter clockwise order), and the centers of neighboring faces (arranged in counter clockwise order).



> **Face Neighbors** - Face centers, face vertices, neighbouring edges, neighbouring face centers

Element* Vertex Neighbors

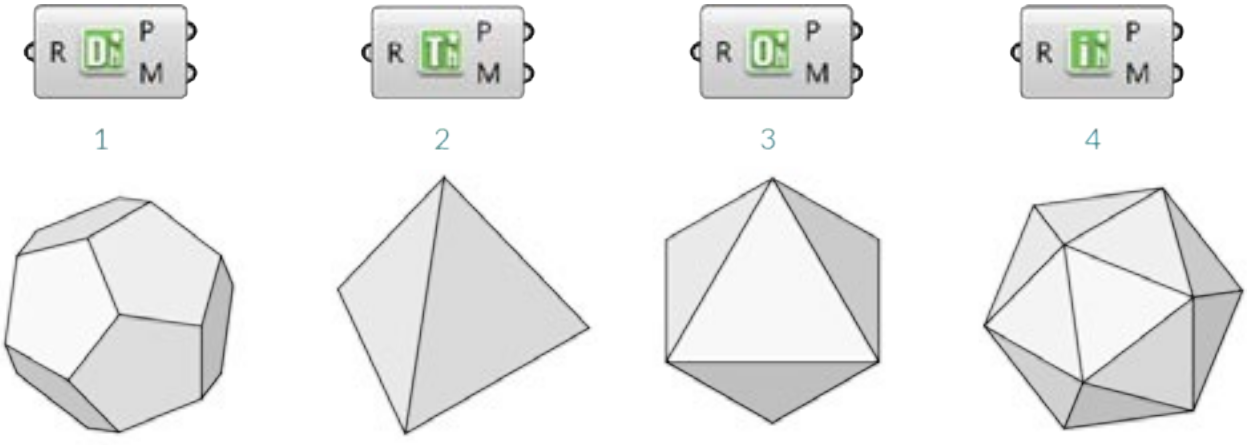
This component outputs the mesh vertices, neighboring vertices (arranged in clockwise order), neighbouring edges (arranged in clockwise order), and neighbouring face centers (arranged in clockwise order) structured in a tree according to the vertices of the mesh.



> **Vertex Neighbors** - Vertices, neighbouring vertices, neighbouring edges, neighbouring face centers

2.1.3.3 Primitives

Element* provides four additional mesh primitives: the Dodecahedron, Tetrahedron, Octahedron, and Icosahedron. These components take a single number as input for the radius, and produce meshes centered at the origin, and composed of one face per side. With the addition of the Cube, which is already availible through Grasshopper's built-in primitives, these make up the five Platonic solids.

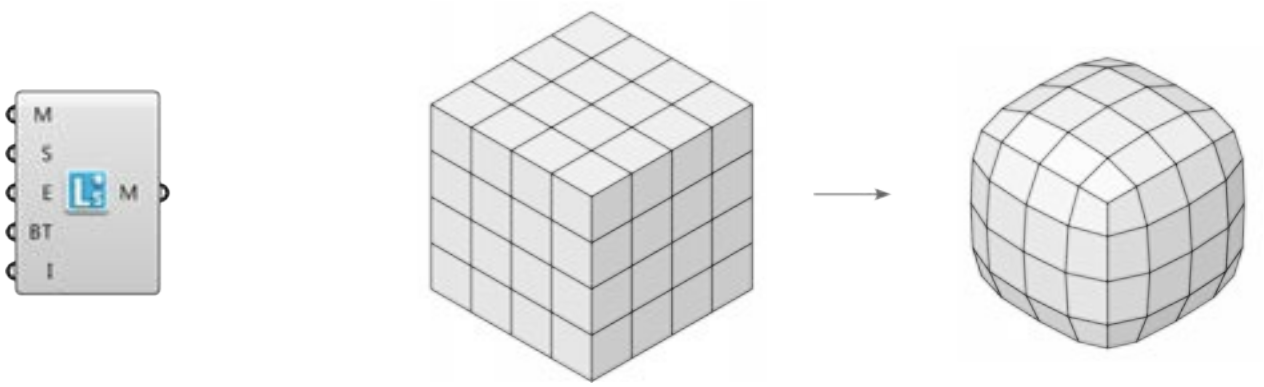


1. Dodecahedron

2. Tetrahedron
3. Octahedron
4. Icosahedron

2.1.3.4 Smooth

Element* Smooth provides an optimized smoothing algorithm that is more efficient than Grasshopper's **Smooth Mesh** for large datasets. It makes use of the Lapacian Smoothing algorithm for Half-Edge structured meshes. It does not change the topology or vertex count of welded meshes, but will combine identical vertices if there are any duplicates caused by an unwelded mesh. We can specify the smoothing strength, boundary condition, boundary tolerance as well as the number of iterations.



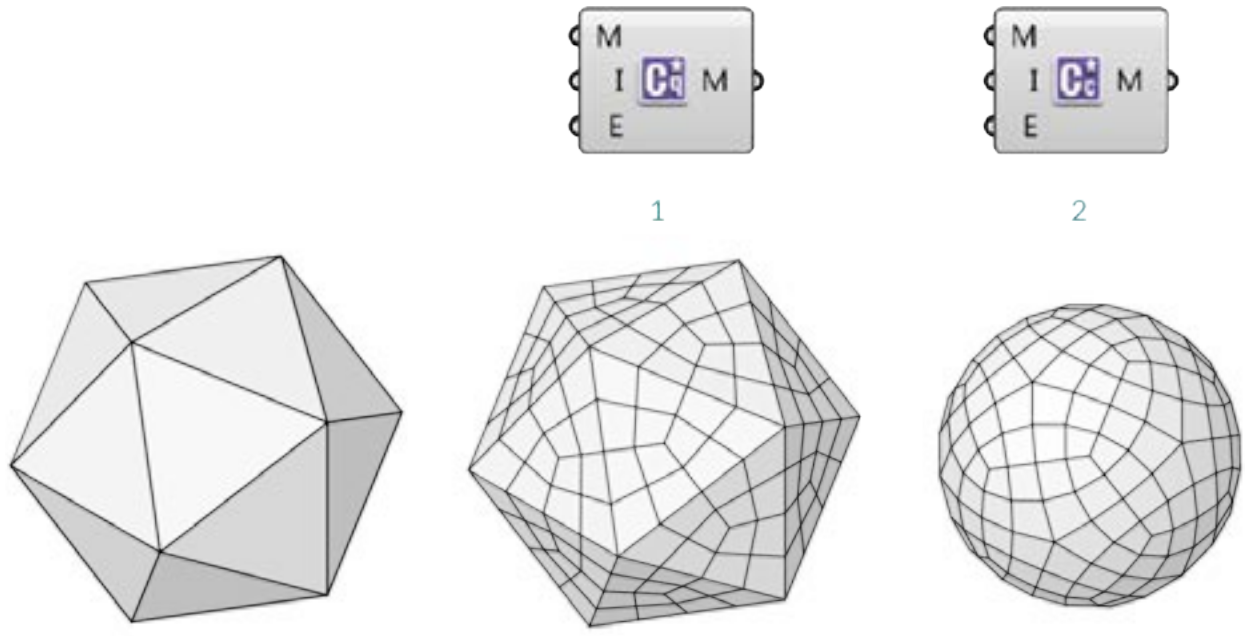
2.1.3.5 Subdivide

Element* Catmull Clark Subdivision

This is a recursive subdivision defined by the Catmull Clark algorithm. We can specify the number of iterations as well as how to handle naked edge conditions.

Element* Constant Quad

This subdivison component will create an all quad mesh by adding a face for each edge of the mesh.



1. Constant Quad subdivision
2. Catmull Clark subdivision

2.1.3.6 Transform

1. Mesh Windown
2. Mesh Frame
3. Mesh Thicken
4. Mesh Offset
5. Mesh Poke Face

These components provide a number of different transformations described below. Each component has the additional capability of accepting per-vertex distance data to allow for variations of the transformation magnitudes across the mesh.

Element* Mesh Window

Reconstructs a new mesh on the inside of a face based on an offset value. This component accepts either a mesh or a list of closed polylines as input.

Element* Mesh Frame

Outputs a frame around mesh faces. Each resultant face will have a new hole in the center. This component accepts either a mesh or a list of closed polylines as input.

Element* Mesh Thicken

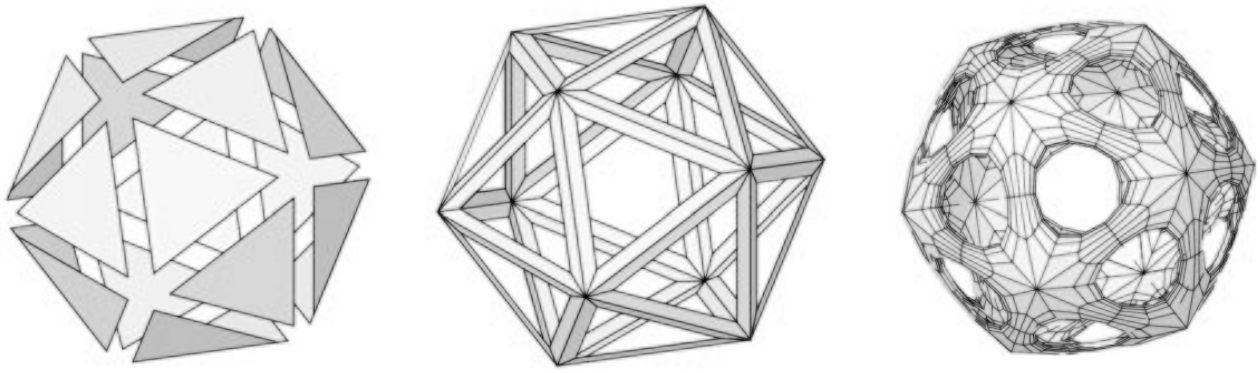
This component will thicken an input mesh along the vertex normals, and according to provided distance values.

Element* Mesh Offset

This component creats an offset of the input mesh based on the vertex normals.

Element* Mesh Poke Face

First the mesh face goes through the frame operation then the face inner is split the selected faces and allows the user to specify the push or pull amount from the center of the original polygon. For example, a four-sided polygon (quad) is split into 4 three-sided polygons with one shared vertex in the middle. The height input allows you to transform that vertex.



1 2 3

1. Mesh Window
2. Mesh Frame
3. Icosohedron transformed with mesh frame, then thickend and subdivided

2.1.3.7 Utility

1 2 3

1. Mesh Combine & Clean
2. Mesh Edges
3. Mesh Status

The **Element* Mesh Combine and Clean**

This component combines multiple meshes, has options for either welding a mesh based on input angle or combining identical vertices. We can also flip the orientation of the mesh. This component also detects potential topology issues and returns Remarks and Warnings with detailed explanantions. In the event that combining identical vertices creates bad topology the component will return the input list of meshes instead of a combined merged mesh. The user can also choose to combine the mesh without merging any of its vertices.

The **Element* Mesh Edges**

This component returns the mesh naked edges, mesh edges, face polylines and if the mesh is unwelded it will return the unwelded mesh edges.

The **Element* Mesh Status**

This component returns mesh information based on the topology. There are two modes which we can view the information, the first is Mesh Info which returns Geometry data such as Mesh Validity, Vertex Count, Face Count, and

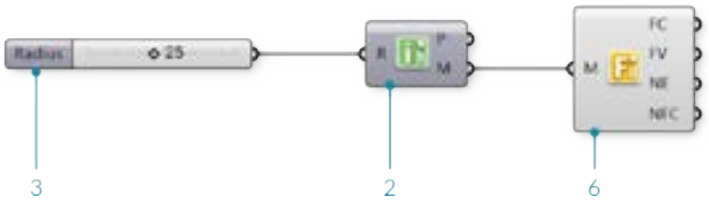
Normal Count. The other returns the Mesh Status, which is the condition of the mesh, whether it has non manifold edges, degenerate face count, naked edge count, and disjoined mesh count. This component does not operate on a mesh it simply returns the information to the user. There is also an option to combine identical vertices, therefore the user can see the effects this would have on the mesh.

2.1.4. Exercise

In this section, we will work through a simple exercise using the Element* primitives as a base. We will incorporate the half-edge data structure as well using both features of the transform components (uniform and per vertex)

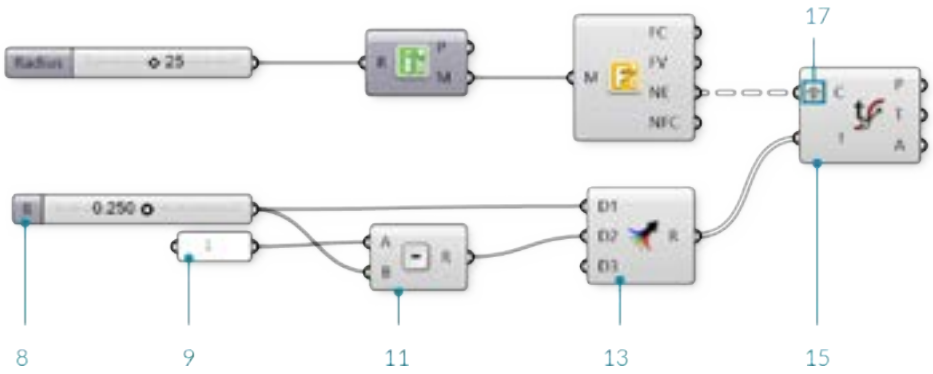
> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

01.	Start a new definition, type Ctrl-N (in Grasshopper)	
02.	**Element*/Primitive/Icosohedron** - Drag and drop the **Icosohedron** component onto the canvas	
03.	**Params/Input/Number Slider** - Drag and drop the **Number Slider** component onto the canvas	
04.	Connect the **Number Slider** to the Radius (R) input of the **Icosohedron** component	
05.	Double-click the **Number Slider** and set appropriate values. For this example, we used: Name: Radius Rounding: Integer Lower Limit: 5 Upper Limit: 50 Value: 25	
06.	**Element*/Data/Face Neighbors** - Drag and drop the **Face Neighbors** component onto the canvas	
07.	Connect the Mesh (M) output of the **Icosohedron** component to the Mesh (M) input of the **Face Neighbors** component.	



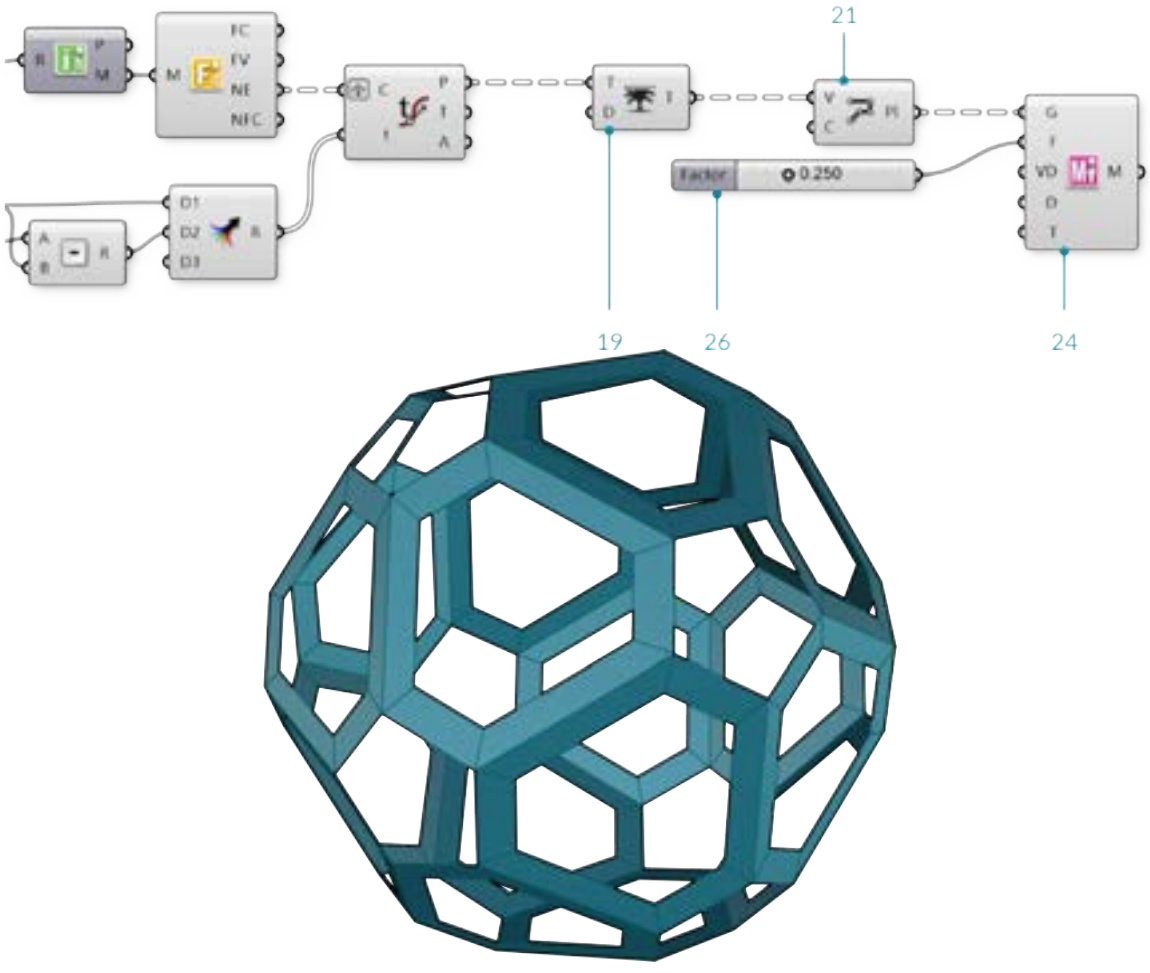
> Looking at the data of the Neighboring Face Edges (NE) output, we see that we have a tree with 20 branches, where each branch contains three lines. The 20 branches each represent a face of the icosohedron which has 20 sides, while the three lines are the edges of each triangular face.

08.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Rounding: Float Lower Limit:0 Upper Limit: 0.5	
09.	**Params/Input/Panel** - Drag and drop a **Panel** component onto the canvas	
10.	Double-click the **Panel** component and enter "1" into the text-field	
11.	**Math/Operators/Subtraction** - Drag and drop a **Subtraction** component onto the canvas	
12.	Connect the **Panel** with a value of "1" into the Ainput and connect the number slider to the B input of the **Subtraction** component	
13.	**Sets/Tree/Merge** - Drag and drop a **Merge** component onto the canvas	
14.	Connect the **Number Slider** to the D1 input of **Merge**, and connect the output R of the **Subtraction** component to the D2 input of **Merge**	
15.	**Curve/Analysis/Evaluate Curve** - Drag and drop an **Evaluate Curve** component onto the canvas	
16.	Connect the Face Edges (NE) output of the **Face Neighbors** component to the Curve (C) input of the the **Evaluate Curve** component	
17.	Right click the Curve (C) input of the **Evaluate Curve** component and select Graft. This will create a new branch for each edge.	
18.	Connect the Result (R) output of the **Merge** component to the Parameter (t) input of the **Evaluate Curve** component. Because we grafted the Curve input, each edge is evaluated at both parameters from **Merge**	

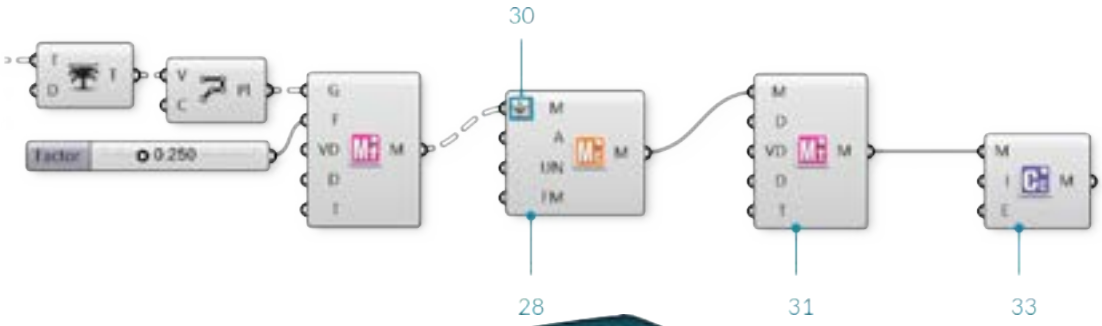


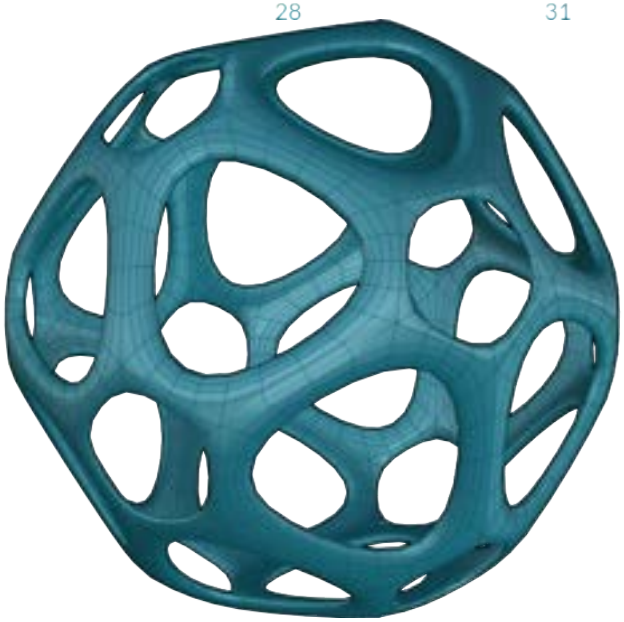
19.	**Sets/Tree/Trim Tree** - Drag and drop a **Trim Tree** component onto the canvas	
	Connect the Points (P) output of **Evaluate Curve** to the Tree (T) input of the **Trim Tree**	

20.	component. > The default value of Depth (D) input for **Trim Tree** is 1. This reduce the depth of our data tree one level by merging the outer most branch. The result is 20 branches, each with six points.	
21.	**Curve/Spline/Polyline** - Drag and drop a **Polyline** component onto the canvas	
22.	Connect the Tree (T) output of the **Trim Tree** component to the Vertices (V) input of the **Polyline** component	
23.	Right click the Closed (C) input of the **Polyline** component, click "Set Boolean" and set the value to True > This has created a closed polyline of six sides for each original face of the mesh.	
24.	**Element*/Transform/Mesh Frame** - Drag and drop a **Mesh Frame** component onto the canvas.	
25.	Connect the Polyline (Pl) output of the **Polyline** component to the Geometry (G) input of the **Mesh Frame** component > Note that the **Mesh Frame** component can take either meshes or a list of closed polyline curves as input	
26.	**Params/Input/Number Slider** - Drage and drop a **Number Silder** component onto the canvas. We will keep the default range of 0 to 1 for this slider	
27.	Connect the **Number Slider** to the Factor (F) input of the **Mesh Frame** component	



28.	**Element*/Utility/Mesh Combine and Clean** - Drag and drop a **Mesh Combine and Clean** component on the canvas	
29.	Connect the Mesh (M) output of **Mesh Frames** to the Mesh (M) input of the **Mesh Combine and Clean** component	
30.	Right click the Mesh (M) input of **Mesh Combine and Clean** and select Flatten > By flattening the tree of meshes, **Combine and Clean** will merge all 20 face meshes into a single mesh	
31.	**Element*/Transform/Mesh Thicken** - Drag and drop a **Mesh Thicken** component onto the canvas	
32.	Connect the Mesh (M) output of **Combine and Clean** to the Mesh (M) input of **Mesh Thicken**	
33.	**Element*/Subdivide/Catmull Clark Subdivision** - Drag and drop a **Catmull Clark Subdivision** component onto the canvas	
34.	Connect the Mesh (M) output of **Mesh Thicken** to the Mesh (M) input of the **Catmull Clark Subdivision** component	

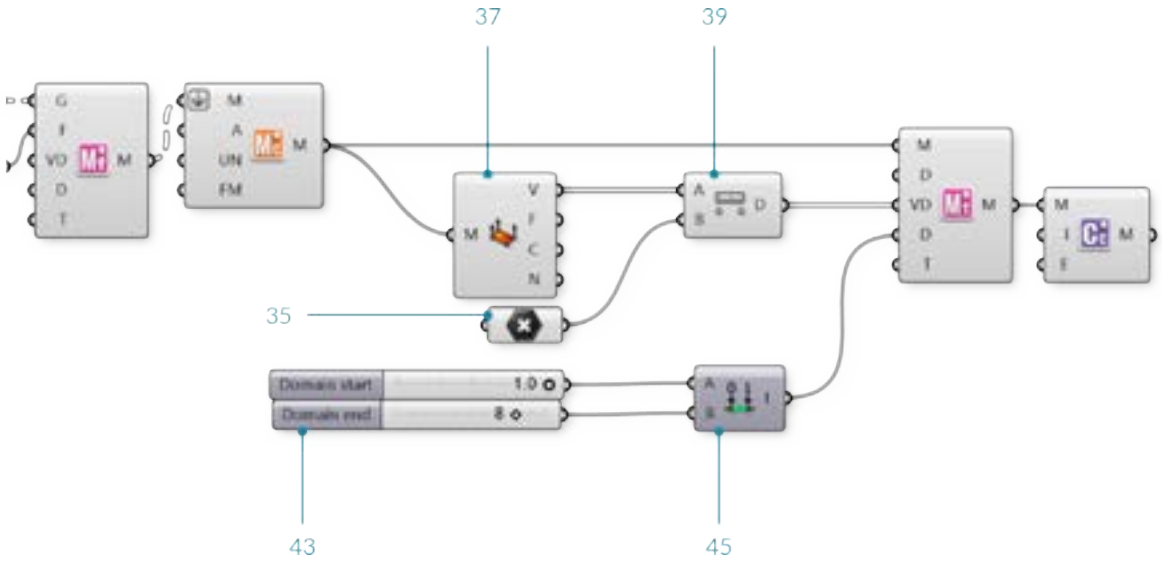


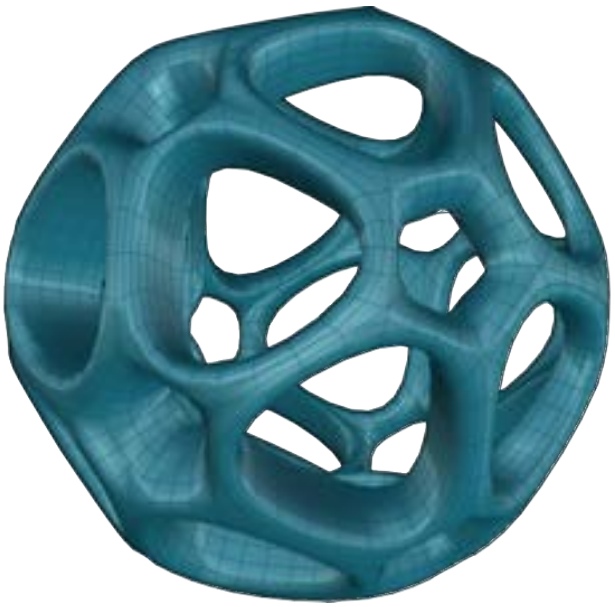


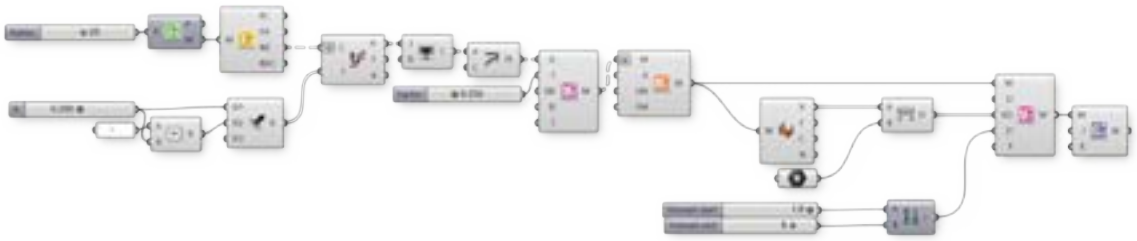
> We have truncated the triangular faces of the initial mesh, effectively also creating rings around each original vertex. We have also created a frame for each face, then thickened the mesh and refined it with subdivision. Next we will take advantage of the Per Vertex capabilities of the transform components by using an attractor point.

35.	**Params/Geometry/Point** - Drag and drop a **Point** parameter onto the canvas	
	Right click the **Point** parameter and select "Set on point" to select a point in the Rhino viewport	

36.	Tip - you can also create a point directly in Grasshopper by double-clicking the canvas to bring up the Search window, then typing a point coordinate such as "10,10,0" (without the quotes)	
37.	**Mesh/Analysis/Deconstruct Mesh** - Drag and drop a **Deconstruct Mesh** component onto the canvas	
38.	Connect the Mesh (M) output of the **Combine and Clean** component to the Mesh (M) input of the **Deconstruct Mesh** component. We will use this to extract the vertices of our combined mesh, and then apply an attractor point to these vertices	
39.	**Vector/Point/Distance** - Drag and drop a **Distance** component onto the canvas	
40.	Connect the Vertices (V) output of the **Deconstruct Mesh** component to the Ainput of the the **Distance** component	
41.	Connect the **Point** parameter to the B input of the **Distance** component	
42.	Connect the Distance (D) output of the **Distance** component to the PerVectex Data (VD) input of the **Thicken** component	
43.	**Params/Input/Number Slider** - Drag and drop two **Number Slider** components onto the canvas. We will use these to set the lower and upper limits for the **Mesh Thicken** component	
44.	Double-click the **Number Sliders** and set the values. In this example, we left the first slider at default values, and set the Upper Limit of the second slider to 5.0	
45.	**Maths/Domain/Construct Domain** - Drag and drop a **Construct Domain** component onto the canvas	
46.	Connect the two number sliders to the Aand B inputs of the **Construct Domain** component	
47.	Connect the Domain (I) output of the **Construct Domain** component to the Min and Max Values (D) input of the **Mesh Thicken** component.	
48.	Right click the Type (T) input of the **Thicken** component, select "Set Integer" and enter a value of 1 You can also enable the PerVertex Data by using a **Boolean Toggle** component set to True.	

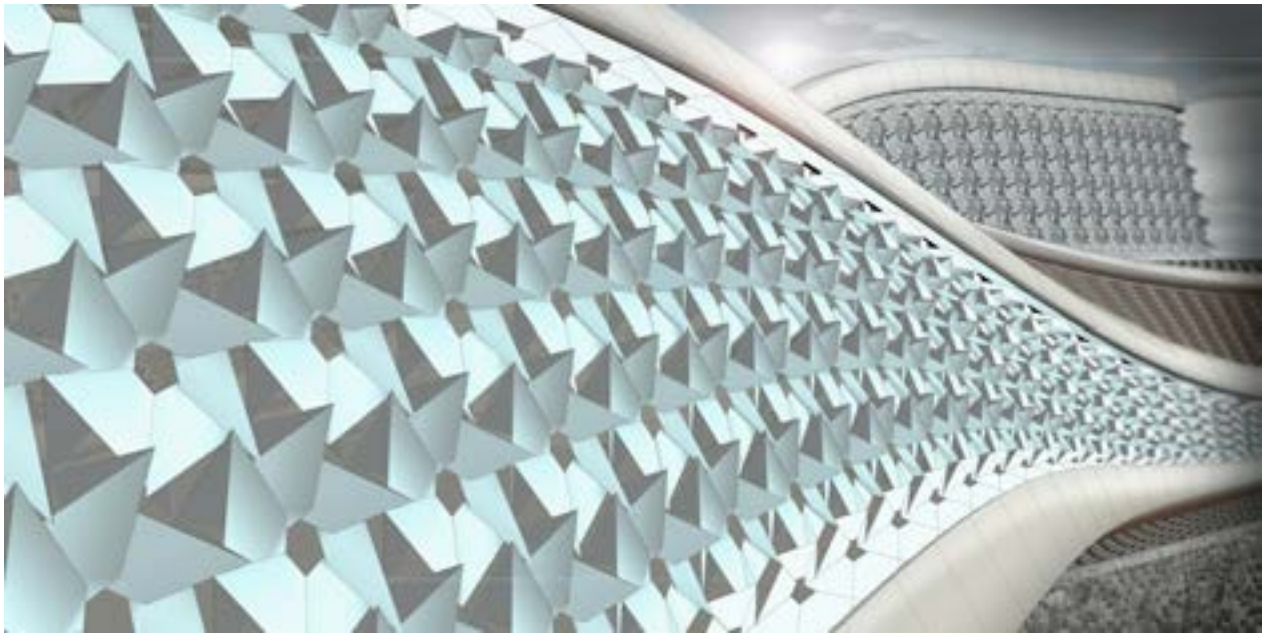






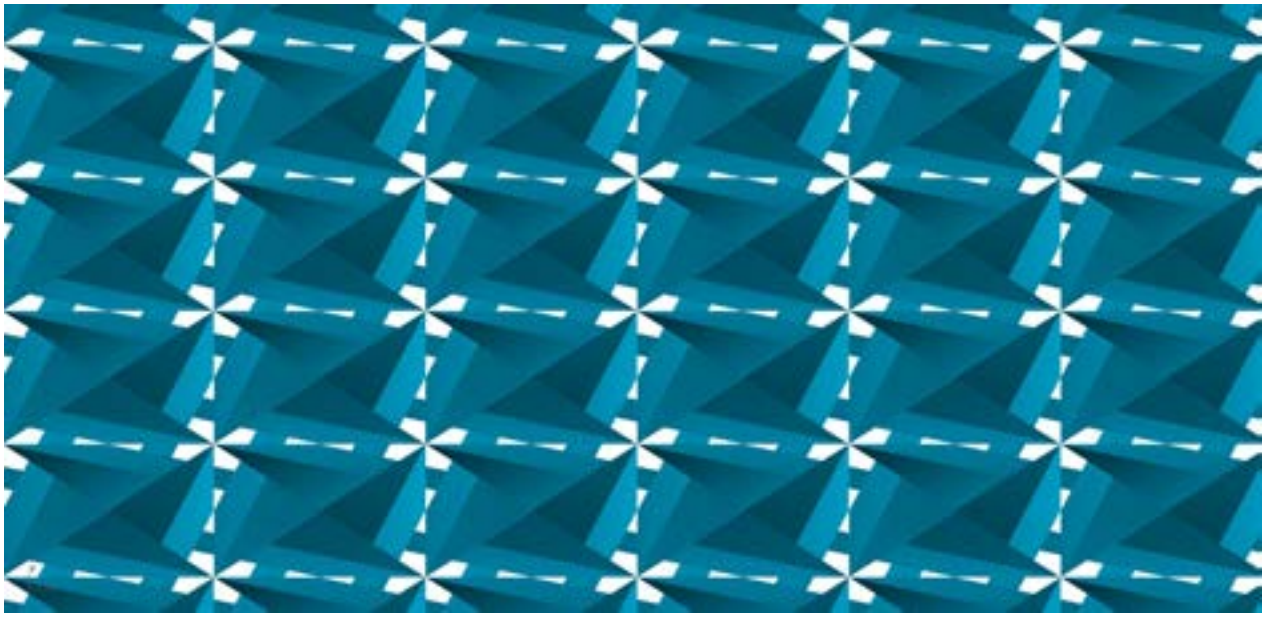
2.1.5. Element* Architectural Case Study

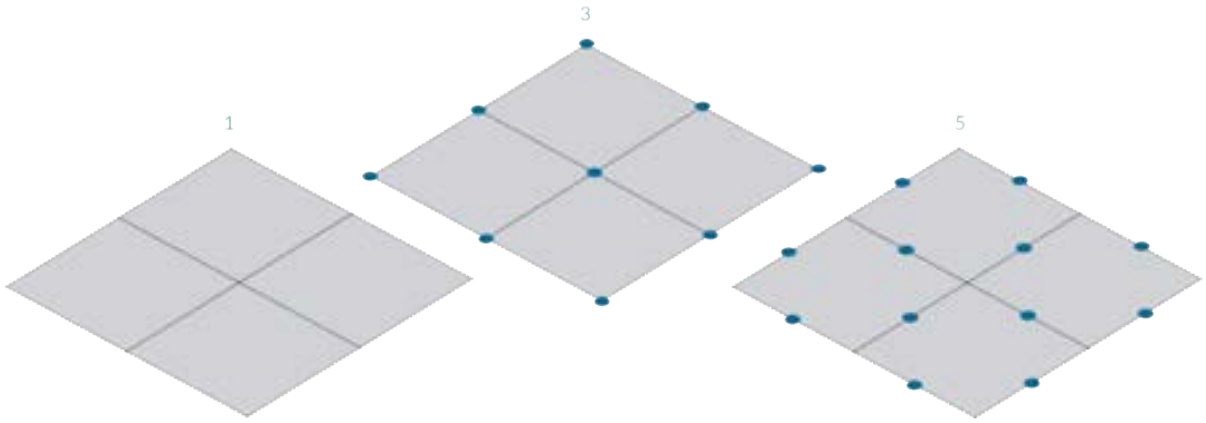
In this section, we will work through a simple exercise file that is meant as an introduction to working with Element tools. We will explore some patterning and facade treatments in the field of Architecture which will incorporate Half Edge data structures along with basic Element components without the use of per vertex features.



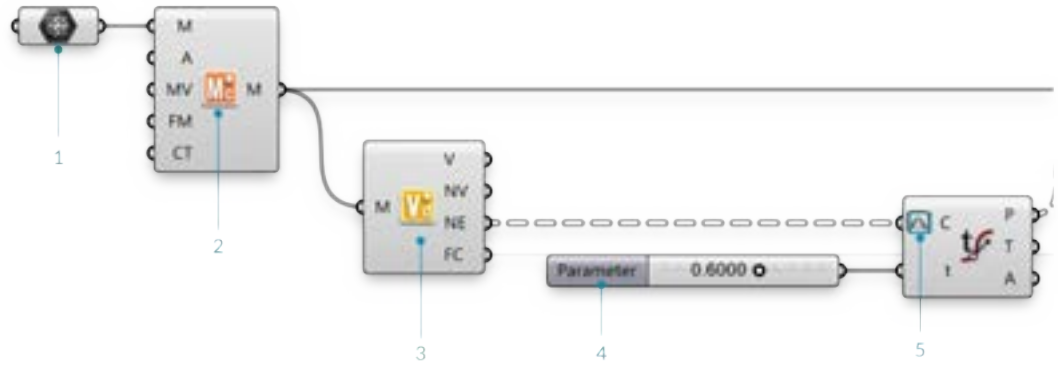
2.1.5.1 Example 1

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

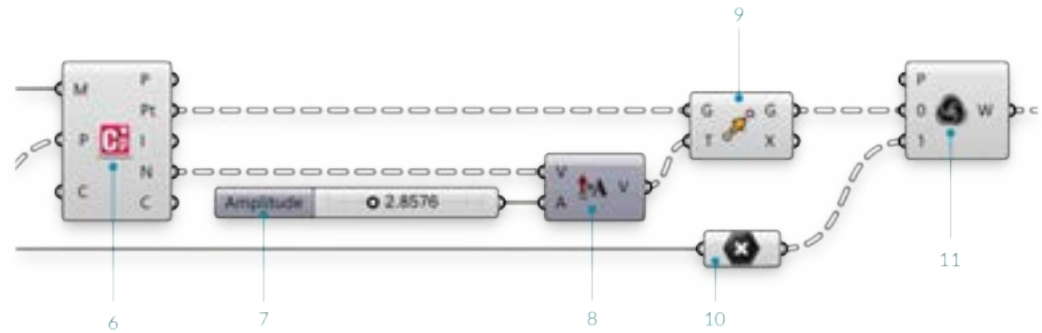


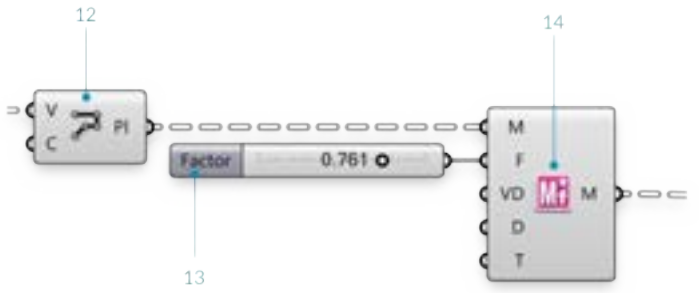


00.	Create a meshplane in Rhino with **XFaces = 2 & YFaces = 2** and Start a new definition, type Ctrl-N (in Grasshopper)	
01.	**Params/Geometry/Mesh** - Drag and drop a **Mesh** container onto the canvas	
01b.	Reference a mesh in Rhino by right-clicking the **Mesh** component and selecting "Set one Mesh". > We are going to use a simple mesh plane to walk through the definition, feel free to swap out the mesh with your own mesh	
02.	**Element*/Utility/Mesh Combine and Clean** - Drag and drop a **Element* Mesh Combine and Clean** component on the canvas	
03.	**Element*/Data/Vertex Neighbors** - Drag and drop the **Element* Vertex Neighbors** component onto the canvas	
04.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Lower Limit: 0.0000 Upper Limit: 1.0000	
05.	**Curve/Analysis/Evaluate Curve** - Drag and drop a **Evaluate Curve** container onto the canvas	
05b.	Connect the Neighbouring Edges (NE) output of the **Element* Vertex Neighbors** component to the Curve (C) input of the **Evaluate Curve** component	
05c.	Connect the **Number Slider** to the Float (t) input of the **Evaluate Curve** component and set the value to 0.5000	
05d.	Right click the Curve (C) input of the **Evaluate Curve** component and enable **Reparameterize**	

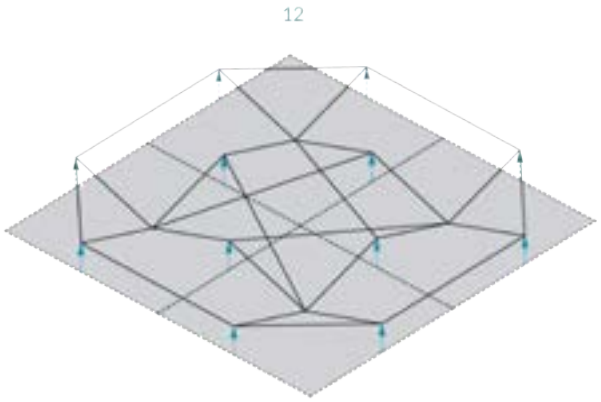


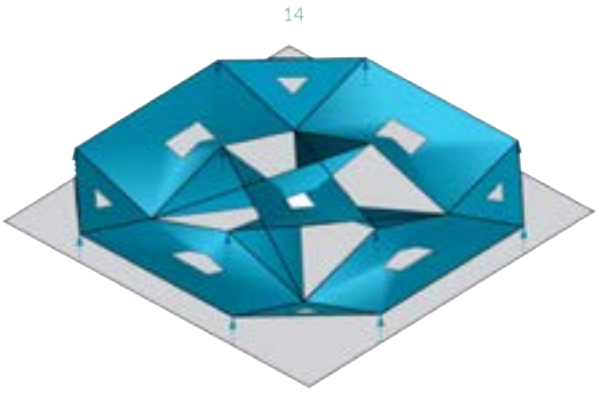
06.	**Element*/Analyse/Mesh Closest Point** - Drag and drop a **Element* Mesh Closest Point** container onto the canvas	
06a.	Connect the Mesh output (M) of the **Element*/Utility/Mesh Combine and Clean** component to the Mesh (M) input of the **Element* Mesh Closest Point** component	
06b.	Connect the Points output (P) of the **Curve/Analysis/Evaluate Curve** component to the Point (P) input of the **Element* Mesh Closest Point** component	
07.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Rounding: Float Lower Limit:0 Upper Limit: 10.000	
08.	**Vector/Vector/Amplitude** - Drag and drop a **Amplitude** component onto the canvas	
09.	**Transform/Euclidean/Move** - Drag and drop a **Move** component onto the canvas	
10.	**Params/Geometry/Point** - Drag and drop a **Point** container onto the canvas	
10b.	Connect the Face Centers output (FC) of the **Element* Vertex Neighbors** component to the **Point** component	
11.	**Sets/List/Weave** - Drag and drop a **Weave** component onto the canvas	





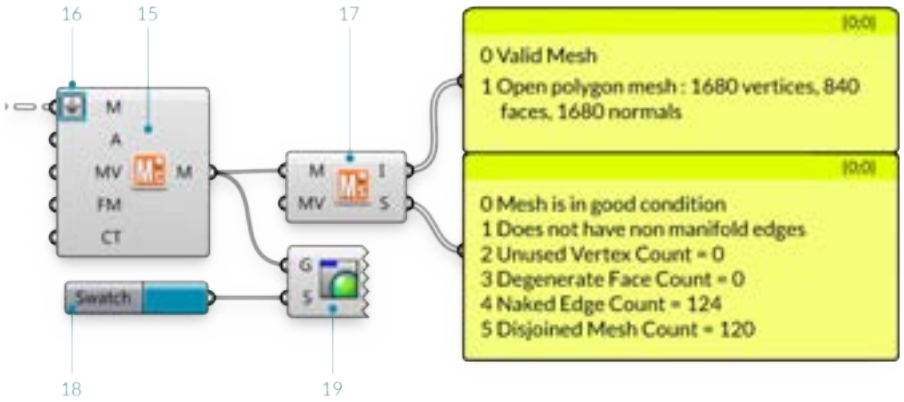
12.	**Curve/Primitive/Polyline** - Drag and drop a **Polyline** component onto the canvas	
12b.	Connect the Weave output (W) of the **Weave** component to the Vertices (V) input of the **Polyline** component	
12c.	Right click the Closed (C) input of the **Polyline** component, click "Set Boolean" and set the value to True > This has created a closed polyline.	
13.	**Params/Input/Number Slider** - Drage and drop a **Number Silder** component onto the canvas. We will keep the default range of 0 to 1 for this slider	
14.	**Element*/Transform/Mesh Frame** - Drag and drop a **Element* Mesh Frame** component onto the canvas.	
14b.	Connect the Polyline (Pl) output of the **Polyline** component to the Geometry (G) input of the **Mesh Frame** component > Note that the **Mesh Frame** component can take either meshes or a list of closed polyline curves as input	
14c.	Connect the **Number Slider** to the Factor (F) input of the **Mesh Frame** component	

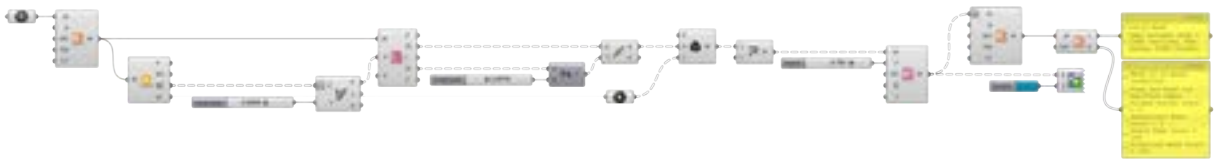




15.	**Element*/Utility/Mesh Combine and Clean** - Drag and drop a **Element* Mesh Combine and Clean** component on the canvas	
	Right click the Combine Type (CT) input of the **Element* Mesh Combine and Clean** component, click "Set Integer" and set the value to 1.	

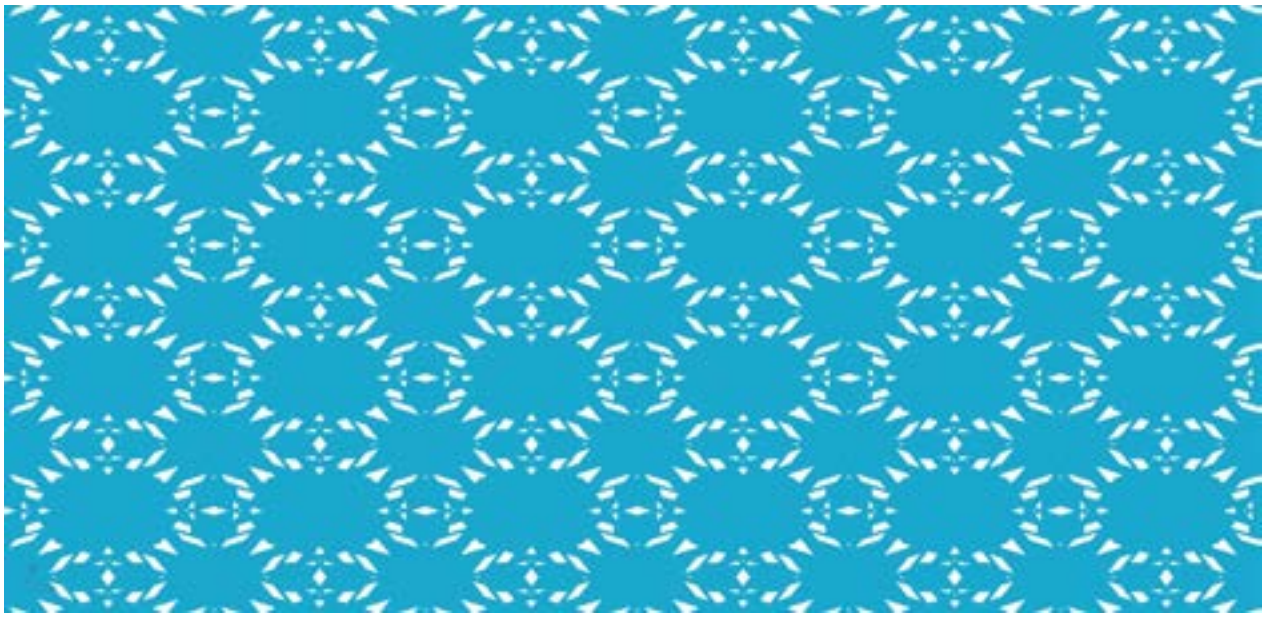
15b.	> The Combine Type input has two options (0, which combines and cleans the meshes) and (1, which joins the meshes in the list without merging vertices). In this example we want to join the meshes	
16.	Right click the Mesh (M) input of the **Element* Mesh Combine and Clean** component, click "Flatten". > This will flatten the list so we can join the mesh list together.	
17.	**Element*/Utility/Mesh Status** - Drag and drop a **Element* Mesh Status** component on the canvas	
17b	Connect the Info (I) and Status (S) outputs of **Element* Mesh Status** to a **Params/Input/Panel** component > The mesh **Info** output contains mesh validity information, closed or open type and mesh component counts (vertices, faces, normals). The mesh **Status** informs the user if the mesh is in "Good" condition as well as data regarding non manifold edges, unused vertex count, degenerate face count, naked edge count and disjoined mesh count.	
18.	**Params/Input/Colour Swatch** - Drag and drop a **Colour Swatch** component on the canvas	
19.	**Display/Preview/Custom Preview** - Drag and drop a **Custom Preview** component on the canvas	



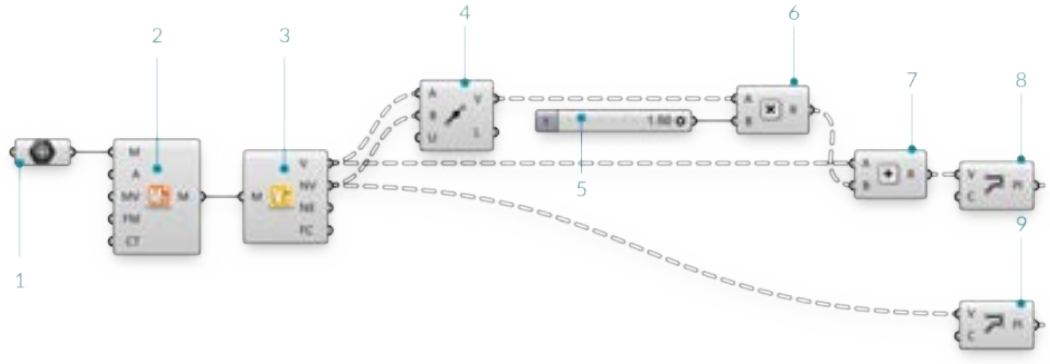


2.1.5.2 Example 2

> Example files that accompany this section: http://grasshopperprimer.com/appendix/A-2/1_gh-files.html

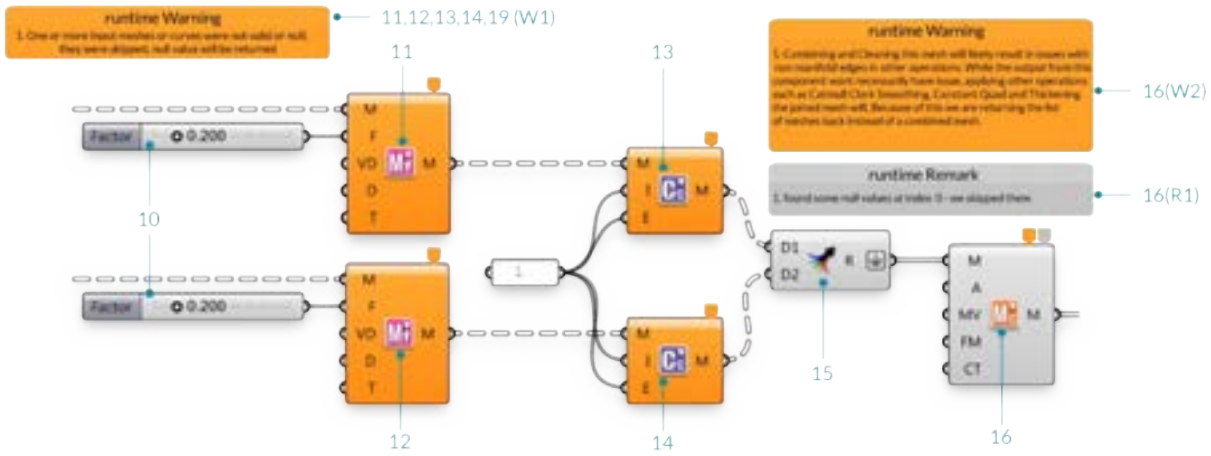


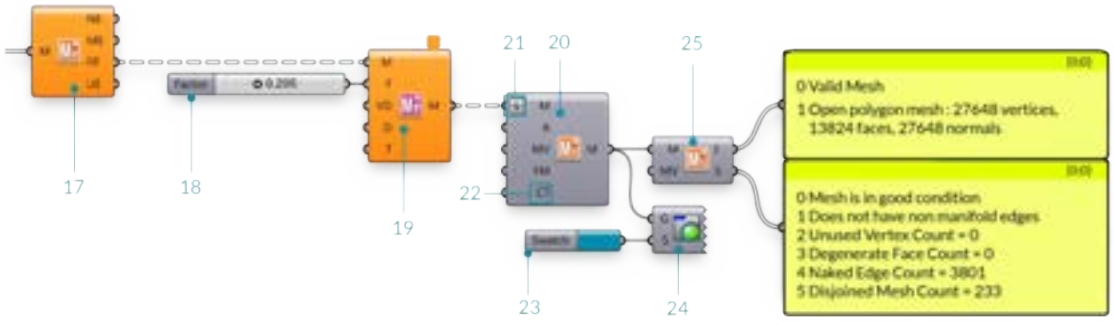
00.	Create a meshplane in Rhino with **XFaces = 2 & YFaces = 2** and Start a new definition, type Ctrl-N (in Grasshopper)	
01.	**Params/Geometry/Mesh** - Drag and drop a **Mesh** container onto the canvas	
01b.	Reference a mesh in Rhino by right-clicking the **Mesh** component and selecting "Set one Mesh". > We are going to use a simple mesh plane to walk through the definition, feel free to swap out the mesh with your own mesh	
02.	**Element*/Utility/Mesh Combine and Clean** - Drag and drop a **Element* Mesh Combine and Clean** component on the canvas	
03.	**Element*/Data/Vertex Neighbors** - Drag and drop the **Element* Vertex Neighbors** component onto the canvas	
04.	**Vector/Vector/Vector2Pt** - Drag and drop a **Vector2Pt** component onto the canvas	
05.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Rounding: Float Lower Limit:0 Upper Limit: 2.000	
06.	**Maths/Operator/Multiplication** - Drag and drop a **Multiplication** component onto the canvas	
07.	**Maths/Operators/Addition** - Drag and drop two **Addition** components onto the canvas	
08.	**Curve/Primitive/Polyline** - Drag and drop a **Polyline** component onto the canvas	
09.	**Curve/Primitive/Polyline** - Drag and drop a **Polyline** component onto the canvas	



10.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Rounding: Float Lower Limit:0 Upper Limit: 1.000	
11,12.	**Element*/Transform/Mesh Frame** - Drag and drop a **Element* Mesh Frame** component onto the canvas.	
11b,12b.	Connect the Polyline (Pl) output of the **Polyline** component to the Geometry (G) input of the **Mesh Frame** component > Note that the **Mesh Frame** component can take either meshes or a list of closed polyline curves as input	
11c,12c.	Connect the **Number Slider (10)** to the Factor (F) input of the **Mesh Frame** component	
13,14.	**Element*/Subdivide/Catmull Clark Subdivision** - Drag and drop a **Catmull Clark Subdivision** component onto the canvas > We will set the Iterations input (I) value to 1 as well as the **Edge Condition** input (E) to a value of 1. The edge condition input options are 0 = Fixed, 1 == Smooth, 2 == Corners Fixed.	
15.	**Sets/Tree/Merge** - Drag and drop two **Merge** components onto the canvas	
15b.	Right click the Result (R) output of the **Merge** component and click "Flatten".	
16.	**Element*/Utility/Mesh Combine and Clean** - Drag and drop a **Element* Mesh Combine and Clean** component on the canvas	

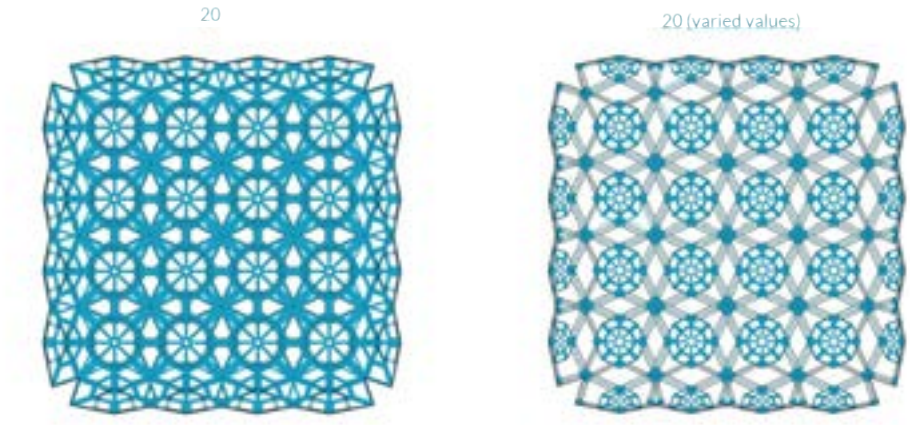
> Components have detailed remarks and warnings to inform the user of the current or potential issues that might come about from interaction with other components. In some instances you might use the Element *Combine and Clean component to join and combine identical vertices on a mesh which could lead to non manifold edges if that mesh is thickend later on. The Element* Combine and Clean component will inform you of this issue that will return the list back to you. You have the option of setting the Combine Type to a value of 1 which will combine the meshes in the list but not combine identical vertices.

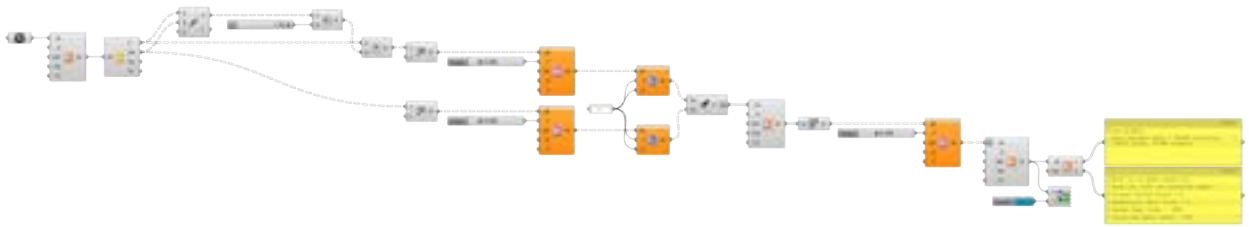




17.	**Element*/Utility/Mesh Edges** - Drag and drop a **Element* Mesh Edges** component on the canvas
17b	Connect the Mesh (M) output of the **Element* Mesh Combine and Clean** component (16) to the Mesh input (M) of the **Element* Mesh Edges** component
18.	**Params/Input/Number Slider** - Drag and drop a **Number Slider** component onto the canvas and set the following values: Rounding: Float Lower Limit:0 Upper Limit: 1.000
19.	**Element*/Transform/Mesh Frame** - Drag and drop a **Element* Mesh Frame** component onto the canvas.
19b	Connect the Face Polylines (FP) output of the **Element* Mesh Edges** component to the Mesh input (M) of the **Element* Mesh Frame** component
19c	Connect the **Number Slider** to the Float (f) input of the **Element* Mesh Frame** component
20.	**Element*/Utility/Mesh Combine and Clean** - Drag and drop a **Element* Mesh Combine and Clean** component on the canvas
21.	Right click the Mesh (M) input of the **Element* Mesh Combine and Clean** component and click "Flatten".
	Right click the Combine Type (CT) input of the **Element* Mesh Combine and Clean**

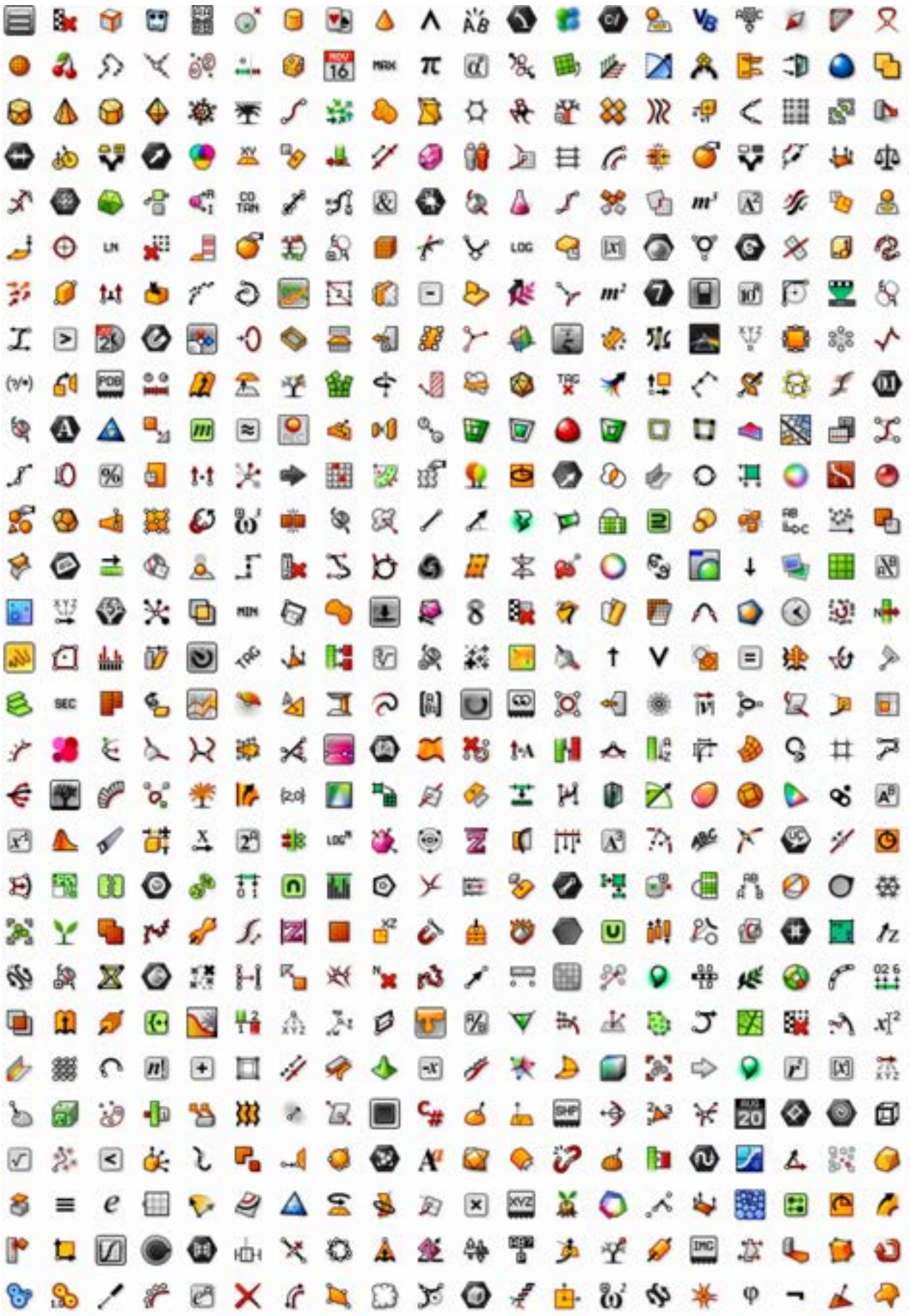
22.	component, click "Set Integer" and set the value to 1. > The Combine Type input has two options (0, which combines and cleans the meshes) and (1, which joins the meshes in the list without merging vertices). In this example we want to join the meshes	
23.	**Params/Input/Colour Swatch** - Drag and drop a **Colour Swatch** component on the canvas	
24.	**Display/Preview/Custom Preview** - Drag and drop a **Custom Preview** component on the canvas	
25.	**Element*/Utility/Mesh Status** - Drag and drop a **Element* Mesh Status** component on the canvas	
25b	Connect the Info (I) and Status (S) outputs of **Element* Mesh Status** to a **Params/Input/Panel** component > The mesh **Info** output contains mesh validity information, closed or open type and mesh component counts (vertices, faces, normals). The mesh **Status** informs the user if the mesh is in "Good" condition as well as data regarding non manifold edges, unused vertex count, degenerate face count, naked edge count and disjoined mesh count.	





Appendix

The following section contains useful references including an index of all the components used in this primer, as well as additional resources to learn more about Grasshopper.



2.1. Index

This index provides additional information on all the components used in this primer, as well as other components you might find useful. This is just an introduction to over 500 components in the Grasshopper plugin.

Parameters

Geometry

P.G.Crv	Curve Parameter Represents a collection of Curve geometry. Curve geometry is the common denominator of all curve types in Grasshopper.	
P.G.Circle	Circle Parameter Represents a collection of Circle primitives.	
P.G.Geo	Geometry Parameter Represents a collection of 3D Geometry.	
P.G.Pipeline	Geometry Pipeline Defines a geometry pipeline from Rhino to Grasshopper.	
P.G.Pt	Point Parameter Point parameters are capable of storing persistent data. You can set the persistent records through the parameter menu.	
P.G.Srf	Surface Parameter Represents a collection of Surface geometry. Surface geometry is the common denominator of all surface types in Grasshopper.	

Primitive

P.P.Bool	Boolean Parameter Represents a collection of Boolean (True/False) values.	
P.P.D	Domain Parameter Represents a collection of one-dimensional Domains. Domains are typically used to represent curve fragments and continuous numeric ranges. Adomain consists of two numbers that indicate the limits of the domain, everything in between these numbers is part of the domain.	
P.P.D2	Domain^2 Parameter Contains a collection of two-dimensional domains. 2D Domains are typically used to represent surface fragments. Atwo-dimensional domain consists of two one-dimensional domains.	
P.P.ID	Guid Parameter Represents a collection of Globally Unique Identifiers. Guid parameters are capable of storing persistent data. You can set the persistent records through the parameter menu.	
P.P.Int	Integer Parameter Represents a collection of Integer numeric values. Integer parameters are capable of storing persistent data. You can set	

	the persistent records through the parameter menu.	
P.P.Num	Number Parameter Represents a collection of floating point values. Number parameters are capable of storing persistent data. You can set the persistent records through the parameter menu.	
P.P.Path	File Path Contains a collection of file paths.	

Input

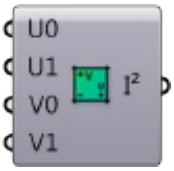
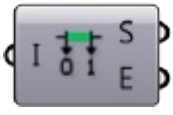
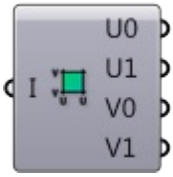
P.I.Toggle	Boolean Toggle Boolean (true/false) toggle.	
P.I.Button	Button Button object with two values. When pressed, the button object returns a true value and then resets to false.	
P.I.Swatch	Color Swatch Aswatch is a special interface object that allows for quick setting of individual color values. You can change the color of a swatch through the context menu.	
P.I.Grad	Gradient Control Gradient controls allow you to define a color gradient within a numeric domain. By default the unit domain (0.0 ~ 1.0) is used, but this can be adjusted via the L0 and L1 input parameters. You can add color grips to the gradient object by dragging from the color wheel at the upper left and set color grips by right clicking them.	
P.I.Graph	Graph Mapper Graph mapper objects allow you to remap a set of numbers. By default the {x} and {y} domains of a graph function are unit domains (0.0 ~ 1.0), but these can be adjusted via the Graph Editor. Graph mappers can contain a single mapping function, which can be picked through the context menu. Graphs typically have grips (little circles), which can be used to modify the variables that define the graph equation. By default, a graph mapper objects contains no graph and performs a 1:1 mapping of values.	
P.I.Slider	Number Slider Aslider is a special interface object that allows for quick setting of individual numeric values. You can change the values and properties through the menu, or by double-clicking a slider object. Sliders can be made longer or shorter by dragging the rightmost edge left or right. Note that sliders only have an output (ie. no input).	
P.I.Panel	Panel Apanel for custom notes and text values. It is typically an inactive object that allows you to add remarks or explanations to a Document. Panels can also receive their information from elsewhere. If you plug an output parameter into a Panel, you can see the contents of that parameter in real-time. All data in Grasshopper can be viewed in this way. Panels can also stream their content to a text file.	
P.I.List	Value List	

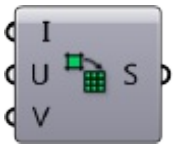
Utilities

P.U.Cin	Cluster Input Represents a cluster input parameter.	
P.U.COut	Cluster Output Represents a cluster input parameter.	
P.U.Dam	Data Dam Delay data on its way through the document.	
P.U.Jump	Jump Jump between different locations.	
P.U.Viewer	Param Viewer A viewer for data structures.	
P.U.Scribble	Scribble A quick note.	Doubleclick Me!

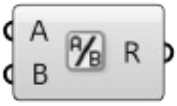
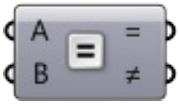
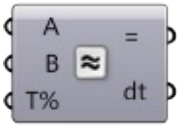
Maths

Domain

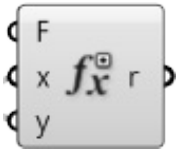
M.D.Bnd	Bounds Create a numeric domain which encompasses a list of numbers.	
M.D.Consec	Consecutive Domains Create consecutive domains from a list of numbers.	
M.D.Dom	Construct Domain Create a numeric domain from two numeric extremes.	
M.D.Dom2Num	Construct Domain² Create a two-dimensional domain from four numbers.	
M.D.DeDomain	Deconstruct Domain Deconstruct a numeric domain into its component parts.	
M.D.DeDom2Num	Deconstruct Domain² Deconstruct a two-dimensional domain into four numbers.	
M.D.Divide	Divide Domain² Divides a two-dimensional domain into equal segments.	

M.D.Divide	Divide Domain² Divides a two-dimensional domain into equal segments.	
M.D.Inc	Includes Test a numeric value to see if it is included in the domain.	
M.D.ReMap	Remap Numbers Remap numbers into a new numeric domain.	

Operators

M.O.Add	Addition Mathematical addition.	
M.O.Div	Division Mathematical division.	
M.O.Equals	Equality Test for (in)equality of two numbers.	
M.O.And	Gate And Perform boolean conjunction (AND gate). Both inputs need to be True for the result to be True.	
M.O.Not	Gate Not Perform boolean negation (NOT gate).	
M.O.Or	Gate Or Perform boolean disjunction (OR gate). Only a single input has to be True for the result to be True.	
M.O.Larger	Larger Than Larger than (or equal to).	
M.O.Multiply	Multiplication Mathematical multiplication.	
M.O.Smaller	Smaller Than Larger than (or equal to).	
M.O.Similar	Similarity Test for similarity of two numbers.	
M.O.Sub	Subtraction Mathematical subtraction.	

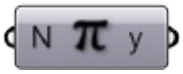
Script

M.S.Eval	Evaluate Evaluate an expression with a flexible number of variables.	
M.S.Expression	Expression Evaluate an expression.	

Trig

M.T.Cos	Cosine Compute the cosine of a value.	
M.T.Deg	Degrees Convert an angle specified in radians to degrees.	
M.T.Rad	Radians Convert an angle specified in degrees to radians.	
M.T.Sin	Sine Compute the sine of a value.	

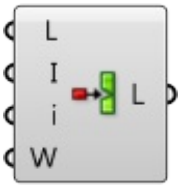
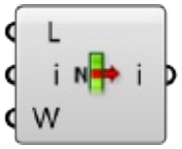
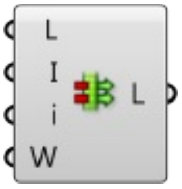
Utilities

M.U.Avr	Average Solve the arithmetic average for a set of items.	
M.U.Phi	Golden Ratio Returns a multiple of the golden ratio (Phi).	
M.U.Pi	Pi Returns a multiple of Pi.	

Sets

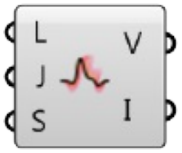
List

S.L.Combine	Combine Data Combine non-null items out of several inputs.	
S.L.CrossRef	Cross Reference Cross Reference data from multiple lists.	
S.L.Dispatch	Dispatch Dispatch the items in a list into two target lists. List dispatching is very similar to the [Cull Pattern] component, with the exception that both lists are provided as outputs.	
S.L.Ins	Insert Items Insert a collection of items into a list.	

		
S.L.Item	List Item Retrieve a specific item from a list.	
S.L.Lng	List Length Measure the length of a list. Elements in a list are identified by their index. The first element is stored at index zero, the second element is stored at index one and so on and so forth. The highest possible index in a list equals the length of the list minus one.	
S.L.Long	Longest List Grow a collection of lists to the longest length amongst them.	
S.L.Split	Split List Split a list into separate parts.	
S.L.Replace	Replace Items Replace certain items in a list.	
S.L.Rev	Reverse List Reverse the order of a list. The new index of each element will be N-i where N is the highest index in the list and i is the old index of the element.	
S.L.Shift	Shift List Offset all items in a list. Items in the list are offset (moved) towards the end of the list if the shift offset is positive. If Wrap equals True, then items that fall off the ends are re-appended.	
S.L.Short	Shortest List Shrink a collection of lists to the shortest length amongst them.	
S.L.Sift	Sift Pattern Sift elements in a list using a repeating index pattern.	
S.L.Sort	Sort List Sort a list of numeric keys. In order for something to be sorted, it must first be comparable. Most types of data are not comparable, Numbers and Strings being basically the sole exceptions. If you want to sort other types of data, such as curves, you'll need to create a list of keys first.	
S.L.Weave	Weave	

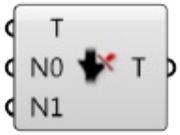
	Weave a set of input data using a custom pattern. The pattern is specified as a list of index values (integers) that define the order in which input data is collected.	

Sets

S.S.Culli	Cull Index Cull (remove) indexed elements from a list.	
S.S.Cull	Cull Pattern Cull (remove) elements in a list using a repeating bit mask. The bit mask is defined as a list of Boolean values. The bit mask is repeated until all elements in the data list have been evaluated.	
S.S.Dup	Duplicate Data Duplicate data a predefined number of times. Data can be duplicated in two ways, either copies of the list are appended at the end until the number of copies has been reached, or each item is duplicated a number of times before moving on to the next item.	
S.S.Jitter	Jitter Randomly shuffles a list of values. The input list is reordered based on random noise. Jittering is a good way to get a random set with a good distribution. The jitter parameter sets radius of the random noise. If jitter equals 0.5, then each item is allowed to reposition itself randomly to within half the span of the entire set.	
S.S.Random	Random Generate a list of pseudo random numbers. The number sequence is unique but stable for each seed value. If you do not like a random distribution, try different seed values.	
S.S.Range	Range Create a range of numbers. The numbers are spaced equally inside a numeric domain. Use this component if you need to create numbers between extremes. If you need control over the interval between successive numbers, you should be using the [Series] component.	
S.S.Repeat	Repeat Data Repeat a pattern until it reaches a certain length.	
S.S.Series	Series Create a series of numbers. The numbers are spaced according to the {Step} value. If you need to distribute numbers inside a fixed numeric range, consider using the [Range] component instead.	

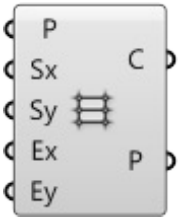
Tree

S.T.Explode	Explode Tree Extract all the branches from a tree.	

S.T.Flatten	Flatten Tree Flatten a data tree by removing all branching information.	
S.T.Flip	Flip Matrix Flip a matrix–like data tree by swapping rows and columns.	
S.T.Graft	Graft Tree Typically, data items are stored in branches at specific index values (0 for the first item, 1 for the second item, and so on and so forth) and branches are stored in trees at specific branch paths, for example: {0;1}, which indicates the second sub-branch of the first main branch. Grafting creates a new branch for every single data item.	
S.T.Merge	Merge Merge a bunch of data streams.	
S.T.Path	Path Mapper Perform lexical operations on data trees. Lexical operations are logical mappings between data paths and indices which are defined by textual (lexical) masks and patterns.	
S.T.Prune	Prune Tree Removes all branches from a Tree that carry a special number of Data items. You can supply both a lower and an upper limit for branch pruning.	
S.T.Simplify	Simplify Tree Simplify a tree by removing the overlap shared amongst all branches.	
S.T.TStat	Tree Statistics Get some statistics regarding a data tree.	
S.T.Unflatten	Unflatten Tree Unflatten a data tree by moving items back into branches.	

Vector

Grid

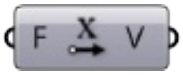
V.G.HexGrid	Hexagonal 2D grid with hexagonal cells.	
V.G.RecGrid	Rectangular 2D grid with rectangular cells.	

V.G.SqGrid	Square 2D grid with square cells	

Point

V.P.Pt	Construct Point Construct a point from {xyz} coordinates.	
V.P.pDecon	Deconstruct Deconstruct a point into its component parts.	
V.P.Dist	Distance Compute Euclidean distance between two point coordinates.	

Vector

V.V.X	Unit X Unit vector parallel to the world {x} axis.	
V.V.Y	Unit Y Unit vector parallel to the world {y} axis.	
V.V.Vec2Pt	Vector 2Pt Create a vector between two points.	

Curve

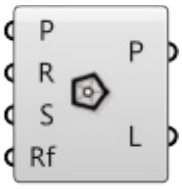
Analysis

C.A.CP	Control Points Extract the nurbs control points and knots of a curve.	

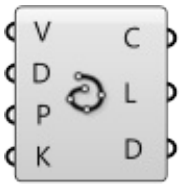
Division

C.D.Divide	Divide Curve Divide a curve into equal length segments.	

Primitive

C.P.Cir	Circle Create a circle defined by base plane and radius.	
C.P.Cir3Pt	Circle 3Pt Create a circle defined by three points.	
C.P.CirCNR	Circle CNR Create a circle defined by center, normal and radius.	
C.P.Line	Line SDL Create a line segment defined by start point, tangent and length.	
C.P.Polygon	Polygon Create a polygon with optional round edges.	

Spline

C.S.IntCrv	Interpolate Create an interpolated curve through a set of points.	
C.S.KinkCrv	Kinky Curve Construct an interpolated curve through a set of points with a kink angle threshold.	
C.S.Nurbs	Nurbs Curve Construct a nurbs curve from control points.	
C.S.PLine	PolyLine Create a polyline connecting a number of points.	

Util

C.U.Explode	Explode Explode a curve into smaller segments.	

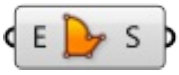
<C.U.Join	Join Curves Join as many curves as possible.	
C.U.Offset	Offset Offset a curve with a specified distance.	

Surface

Analysis

S.A.DeBrep	Deconstruct Brep Deconstruct a brep into its constituent parts.	

Freeform

S.F.Boundary	Boundary Surfaces Create planar surfaces from a collection of boundary edge curves.	
S.F.Extr	Extrude Extrude curves and surfaces along a vector.	
S.F.ExtrPt	Extrude Point Extrude curves and surfaces to a point.	
S.F.Loft	Loft Create a lofted surface through a set of section curves.	
S.F.RevSrf	Revolution Create a surface of revolution.	
S.F.Swp2	Sweep2 Create a sweep surface with two rail curves.	

Primitive

S.P.BBox	Bounding Box Solve oriented geometry bounding boxes.	

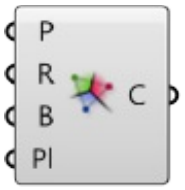
		

Util

S.U.SDivide	Divide Surface Generate a grid of {uv} points on a surface.	
S.U.SubSrf	Isotrim Extract an isoparametric subset of a surface.	

Mesh

Triangulation

M.T.Voronoi	Voronoi Planar voronoi diagram for a collection of points.	

Transform

Affine

T.A.RecMap	Rectangle Mapping Transform geometry from one rectangle into another.	

Array

T.A.ArrLinear	Linear Array Create a linear array of geometry.	

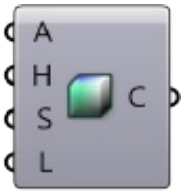
Morph

T.M.Morph	Box Morph Morph an object into a twisted box.	

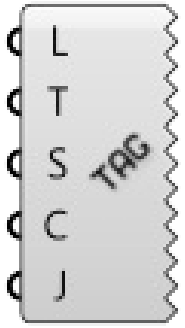
T.M.SBox	Surface Box Create a twisted box on a surface patch.	

Display

Color

D.C.HSL	Colour HSL Create a colour from floating point {HSL} channels.	

Dimensions

D.D.Tag	Text tags A text tag component allows you to draw little Strings in the viewport as feedback items. Text and location are specified as input parameters. When text tags are baked they turn into Text Dots.	
D.D.Tag3D	Text Tag 3D Represents a list of 3D text tags in a Rhino viewport	

Preview

D.P.Preview	Custom Preview Allows for customized geometry previews.	

Vector

D.V.Points	Point List Displays details about lists of points.	

2.2. Grasshopper Example Files

These example files accompany the Grasshopper Primer, and are organized according to section.

1.2.	
	1.2.5_the grasshopper definition.gh

1.3.	
	1.3.2.1_attractor definition.gh
	1.3.3_operators and conditionals.gh
	1.3.3.4_trigonometry components.gh
	1.3.3.5_expressions.gh
	1.3.4_domains and color.gh
	1.3.5_booleans and logical operators.gh

1.4.	
	1.4.1.2_grasshopper spline components.gh
	1.4.3_data matching.gh
	1.4.4_list creation.gh
	1.4.5_list visualization.gh
	1.4.6_list management.gh
	1.4.7_working with lists.gh

1.5.	
	1.5.1.3_morphing definition.gh
	1.5.2.1_Data Tree Visualization.gh
	1.5.3_working with data trees.gh
	1.5.3.6_weaving definition.gh
	1.5.4_rail intersect definition.gh

2.3. Resources

There are many resources available to learn more about Grasshopper and parametric design concepts. There are also over a hundred plugins and add-ons that extend Grasshopper's functionality. Below are some of our favorites.

Plug-in Communities

food4Rhino (WIP) is the new Plug-in Community Service by McNeel. As a user, find the newest Rhino Plug-ins, Grasshopper Add-ons, Textures and Backgrounds, add your comments, discuss about new tools, get in contact with the developers of these applications, share your scripts. http://www.food4rhino.com/

Grasshopper add-ons page http://www.grasshopper3d.com/page/addons-forgrasshopper

Add-ons We Love

DIVA-for-Rhino allows users to carry out a series of environmental performance evaluations of individual buildings and urban landscapes. http://diva4rhino.com/

Element is a mesh geometry plugin for Grasshopper, enabling mesh creation, analysis, transformation, subdivision, and smoothing. http://www.food4rhino.com/project/element

Firefly offers a set of comprehensive software tools dedicated to bridging the gap between Grasshopper and the Arduino micro-controller. http://fireflyexperiments.com

GhPython is the Python interpreter component for Grasshopper that allows you to execute dynamic scripts of any type. Unlike other scripting components, GhPython allows the use of rhinoscriptsyntax to start scripting without needing to be a programmer. http://www.food4rhino.com/project/ghpython

HAL is a Grasshopper plugin for industrial robots programming supporting ABB, KUKAand Universal Robots machines. http://hal.thibaultschwartz.com/

Extends Grasshopper's ability to create and reference geometry including lights, blocks, and text objects. Also enables access to information about the active Rhino document, pertaining to materials, layers, linetypes, and other settings. http://www.food4rhino.com/project/human

Karamba is an interactive, parametric finite element program. It lets you analyze the response of 3-dimensional beam and shell structures under arbitrary loads. http://www.karamba3d.com/

Kangaroo is a Live Physics engine for interactive simulation, optimization and form-finding directly within Grasshopper. http://www.food4rhino.com/project/kangaroo

Fold panels using curved folding and control panel distribution on surfaces with a range of attractor systems. http://www.food4rhino.com/project/robofoldkingkong

LunchBox is a plug-in for Grasshopper for exploring mathematical shapes, paneling, structures, and workflow. http://www.food4rhino.com/project/lunchbox

Meshedit is a set of components which extend Grasshopper's ability to work with meshes. http://www.food4rhino.com/project/meshedittools

Parametric tools to create and manipulate rectangular grids, attractors and support creative morphing of parametric patterns. http://www.food4rhino.com/project/pt-gh

Platypus allows Grasshopper authors to stream geometry to the web in real time. It works like a chatroom for parametric geometry, and allows for on-the-fly 3D model mashups in the web browser.
http://www.food4rhino.com/project/platypus

TT Toolbox features a range of different tools that we from the Core Studio at Thornton Tomasetti use on a regular basis, and we thought some of you might appreciate these. http://www.food4rhino.com/project/tttoolbox

Weaverbird is a topological modeler that contains many of the known subdivision and transformation operators, readily usable by designers. This plug-in reconstructs the shape, subdivides any mesh, even made by polylines, and helps preparing for fabrication. http://www.giuliopiacentino.com/weaverbird/

Additional Primers

The Firefly Primer This book is intended to teach the basics of electronics (using an Arduino) as well as various digital/physical prototyping techniques to people new to the field. It is not a comprehensive book on electronics (as there are already a number of great resources already dedicated to this topic). Instead, this book focuses on expediting the prototyping process. Written by Andrew Payne. http://fireflyexperiments.com/resources/

Essential Mathematics Essential Mathematics uses Grasshopper to introduce design professionals to foundation mathematical concepts that are necessary for effective development of computational methods for 3D modeling and computer graphics. Written by Rajaa Issa.
http://www.rhino3d.com/download/rhino/5.0/EssentialMathematicsThirdEdition/

Generative Algorithms Aseries of books which is aimed to develop different concepts in the field of Generative Algorithms and Parametric Design. Written by Zubin Khabazi. http://www.morphogenesism.com/media.html

Rhino Python Primer This primer is intended to teach programming to absolute beginners, people who have tinkered with programming a bit or expert programmers looking for a quick introduction to the methods in Rhino. Written by Skylar Tibbits. http://www.rhino3d.com/download/IronPython/5.0/RhinoPython101

General References

Wolfram MathWorld is an online mathematics resource., assembled by Eric W. Weisstein with assistance from thousands of contributors. Since its contents first appeared online in 1995, MathWorld has emerged as a nexus of mathematical information in both the mathematics and educational communities. Its entries are extensively referenced in journals and books spanning all educational levels. http://mathworld.wolfram.com/

Further Reading

Burry, Jane, and Mark Burry. *The New Mathematics of Architecture.* London: Thames & Hudson, 2010.

Burry, Mark. *Scripting Cultures: Architectural Design and Programming.* Chichester, UK: Wiley, 2011.

Hensel, Michael, Achim Menges, and Michael Weinstock. *Emergent Technologies and Design: Towards a Biological Paradigm for Architecture.* Oxon: Routledge, 2010.

Jabi, Wassim. *Parametric Design for Architecture.* Laurence King, 2013.

Menges, Achim, and Sean Ahlquist. *Computational Design Thinking.* Chichester, UK: John Wiley & Sons, 2011.

Menges, Achim. *Material Computation: Higher Integration in Morphogenetic Design.* Hoboken, NJ: Wiley, 2012.

Peters, Brady, and Xavier De Kestelier. *Computation Works: The Building of Algorithmic Thought.* Wiley, 2013.

Peters, Brady. *Inside Smartgeometry: Expanding the Architectural Possibilities of Computational Design.* Chichester: Wiley, 2013.

Pottmann, Helmut, and Daril Bentley. *Architectural Geometry.* Exton, PA: Bentley Institute, 2007.

Sakamoto, Tomoko, and Albert Ferré. *From Control to Design: Parametric/algorithmic Architecture.* Barcelona: Actar-D, 2008.

Woodbury, Robert. *Elements of Parametric Design.* London: Routledge, 2010.