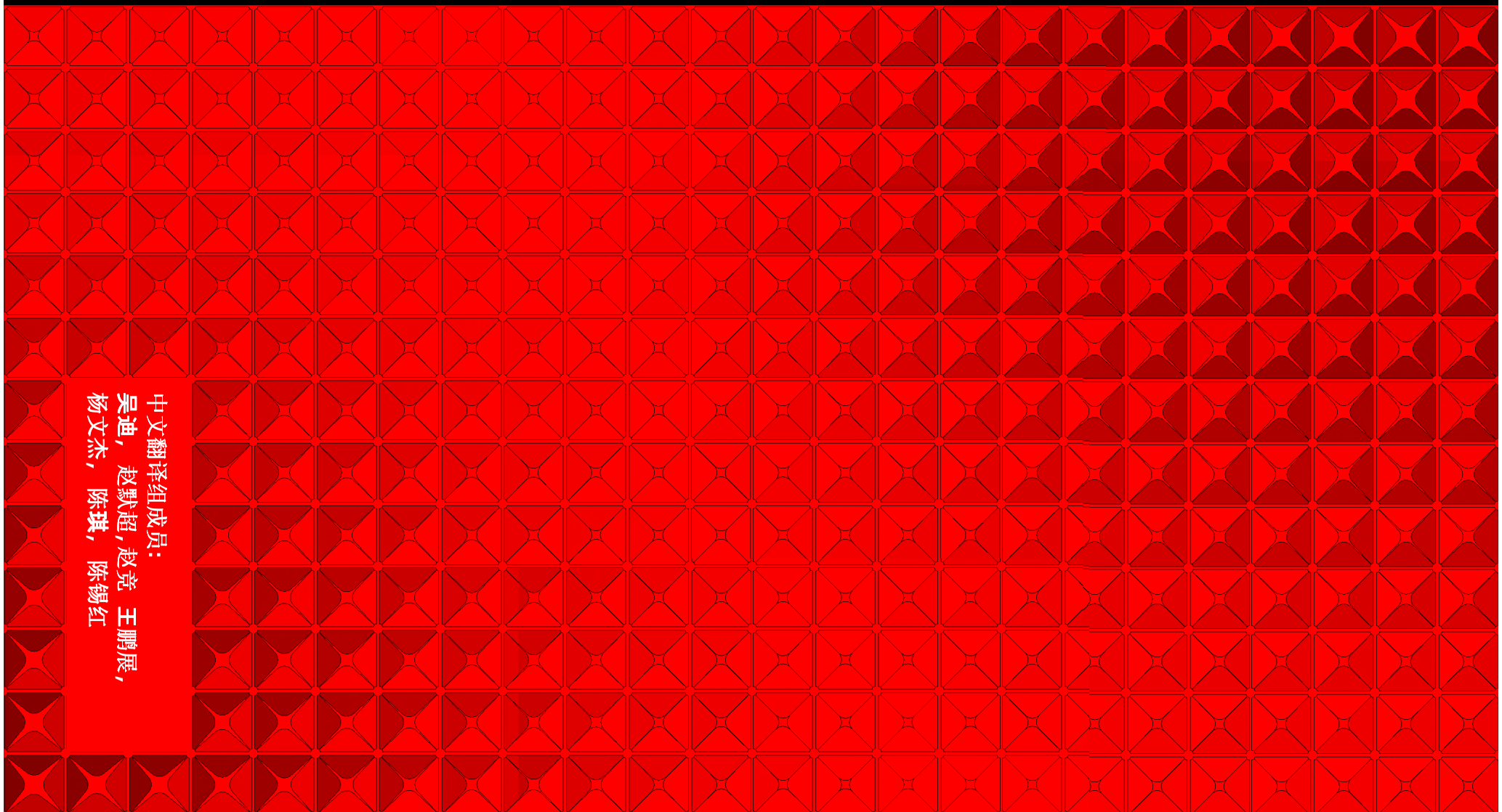
GRASSHOPPER PRIMER

FOR VERSION 0.6.0007 - CHINESE EDITION

BY ANDREW PAYNE & RAJAA ISSA



The Grasshopper Primer, Second Edition – for version 0.6.0007
Copyright 2009 All Rights Reserved

简介

欢迎来到 Grasshopper 的精彩世界。这是第二版的教学手册并在此感谢 Rajaa Issa 为此付出的巨大努力。Rajaa 是 Robert McNeel and Associates 公司的一位软件开发者，而且是许多 Rhino 插件的作者，例如 ArchCut 和更加流行的 PanelingTools 。这个修订版相比初版手册提供了更多更全面的指导，并增加了 70 页的篇幅专门用来介绍如何编写你自己的脚本（Scripting）。

这次 Grasshopper 手册的发布巧碰两件大事：第一件事是新的 0.6.0007 版本 Grasshopper 的发布，新版 Grasshopper 有很大程度的更新，更加丰富了插件的性能。使用者会发现一些关于现行版本中数据存储方面的改变。这个手册希望可以帮助众多新老使用者掌握这些软件系统中的改变；第二件事就是 FLUX 会议，这次会议主题是“Architecture in a Parametric Landscape”，将在加州艺术学院（California College of the Arts）召开。会议将讨论和探索当代建筑和设计和新技术之间的关系，如参数建模，数字生成，脚本。在这之中，会有一个展览和一系列的研讨会专门讨论参数软件系统。我很荣幸可以介绍 Grasshopper 插件，而 Rajaa Issa 和 Gil Akos 将主持高级 Grasshopper 建模和 VB.net Scripting 研讨会。

关于这个手册我们搜集了很多的信息，同时希望它可以一个很好的资源，为那些想学习这个插件的朋友服务。无论如何用户是这个软件最重要的资源，因为当更多人开始探索和理解参数设计的时候，它将会所有人。我鼓励所有阅读这本教程的人加入线上社区并在论坛中提出你们的问题，总会有人会和你分享解决方案的。

想知道更多，请参见
http://www.grashopper.rhino3d.com
十分感谢，并祝君好运！

Andrew Payne
LIFT architects
www.liftarchitects.com
Rajaa Issa
Robert McNeel and Associates
http://www.rhino3d.com/

中文翻译由 ***Shaper3d*** 论坛 Grasshopper 版成员完成，感谢以下成员为翻译工作所做出的努力，如果你对中文版的教程有相关疑问，请至 Grasshopper 中文论坛参与讨论

中文翻译组成员(排名不分先后)：吴迪 赵默超 赵竞 王鹏展 杨文杰 陈琪 陈锡红
中文校订：Jessesn/陈锡红

Shaper3d 论坛
http://bbs.shaper3d.cn
Grasshopper 中文论坛
http://g.shaper3d.cn

目录

介绍
目录

1	开始	**1**
2	软件界面	**2**
3	Grasshopper 对象	**8**
4	静态数据管理	**11**
5	动态数据继承	**13**
6	数据流匹配	**18**
7	数据运算器类型	**21**
7.1	运算器	**21**
7.2	条件语句	**23**
7.2	数列 范围 区间	**25**
7.3	函数与布尔	**27**
7.4	函数与数字型数据	**29**
7.5	三角函数曲线	**32**
8	The Garden of Forking Paths	**36**
8.1	列表和数据管理	**40**
8.2	编织数据	**43**
8.3	转换数据	**46**
8.4	导出数据到 Excel	**48**
9	向量基本原理	**53**
9.1	点/向量 的操作	**55**
9.2	对 Point Attractors 使用 Vector/Scalar Mathematics (修改圆形)	**56**
9.3	对 Point Attractors 使用 Vector/Scalar Mathematics (修改立方体)	**61**
10	曲线类型	**67**
10.1	曲线分析	**72**
11	面的类型	**74**
11.1	面的连接	**76**
11.2	镶板工具	**79**
11.3	表面构架	**84**
11.4	不规则表面构架	**89**
12	脚本编写入门	**92**
13	脚本界面	**93**

13.1 哪里寻找脚本组件 **93**
13.2 输入参数 **93**
13.3 输出参数 **95**
13.4 输出窗口和调试信息 **96**
13.5 脚本组件里面 **97**

14 Visual Basic DotNET **100**
14.1 引言 **100**
14.2 评论 **100**
14.3 变量 **100**
14.4 队列和清单 **101**
14.5 运算器 **103**
14.6 条件性陈述 **104**
14.7 回路 **104**
14.8 巢状回路 **106**
14.9 附属和功能 **107**
14.10 递回 **110**
14.11 Grasshopper 中的推进列表 **113**
14.12 Grasshopper 中的进程树 **114**
14.13 文件输入 /输出 **116**

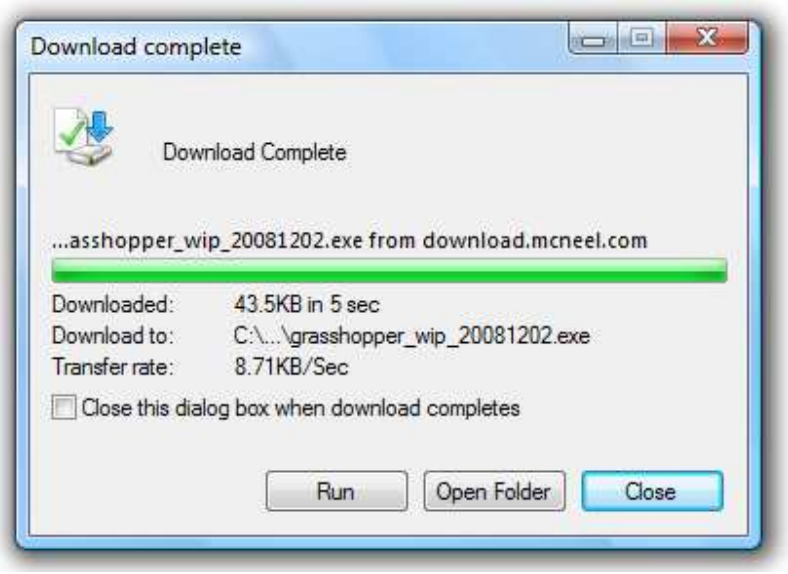
15 Rhino .NET SDK **118**
15.1 概述 **118**
15.2 了解 NURBS **118**
15.3 对象架构 **121**
15.4 分类结构 **123**
15.5 常量与非常量举例 **124**
15.6 点与矢量 **124**
15.7 OnNurbsCurve **126**
15.8 非 OnCurve 衍生的曲线类 **130**
15.9 OnNurbsSurface **132**
15.10 非由 OnSurface 衍生的曲面类 **137**
15.11 OnBrep **138**
15.12 几何变化 **147**
15.13 全局实用函数 **148**

16 帮助 **155**

1 Getting Started（开始）

安装 Grasshopper

下载 Grasshopper 插件请访问 http://Grasshopper.rhino3d.com/. 点击页面左上角 Download（下载）链接进入下一个页面，输入你的电子邮件地址，再对下载链接单击鼠标右键，选择 Save Target As（另存为）。 选择保存位置（注意：本程序不能保存在网络地址后安装，文件必须存在本地磁盘之中），最后将可执行文件保存到该地址。

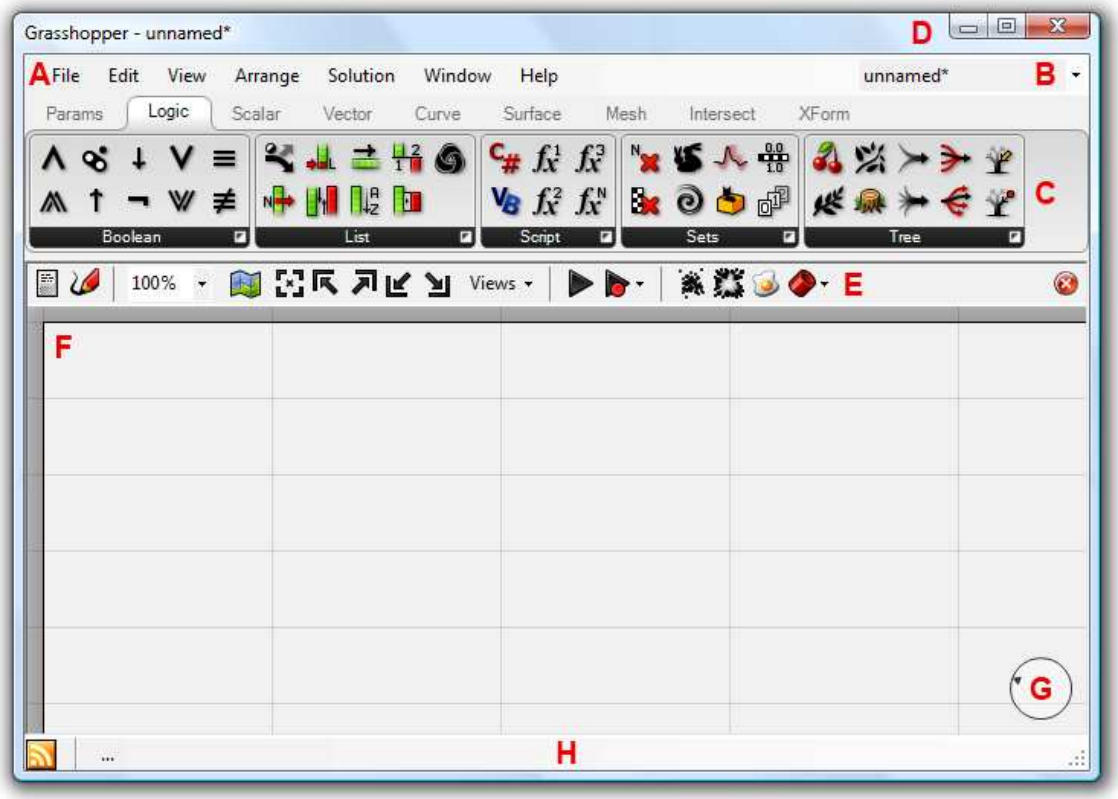


选择下载对话框中的 Run（运行）（注意：你必须装有 Rhino4 及其 sr4b 补丁）

2 The Interface*（软件界面）

主对话框

当你已经加载了插件，在 Rhino 命令栏的提示符后输入“Grasshopper”后会显示 Grasshopper 的主要窗口



这个窗口包含了一些不同的内容，其中大部分对 Rhino 的用户来说是非常熟悉的：

A. The Main Menu Bar 主菜单栏

这个菜单，除了右边 B 区域的文件浏览控制器（file-browser control）以外，和 windows 的经典菜单非常相似。你可以通过这个下拉菜单（文件浏览控制器）在已经载入的不同文件间快速的切换。使用快捷键的时候要注意光标所在视窗的位置，由于它们在已激活的窗口中使用。而这些已激活的窗口有可能是 Rhino 的主窗口或 Grasshopper 插件的窗口或是其他在 Rhino 中运行的窗口。由于目前并没有“Undo（复原）”这一命令，所以你应该对 Ctrl+X（剪切）, Ctrl+S（存档） 与 Del（删除）这些快捷键尤为注意。

B. File Browser Control 文件浏览控制器

如上一部分所言，你可以通过文件浏览控制器在已经载入的文件间快速的切换。

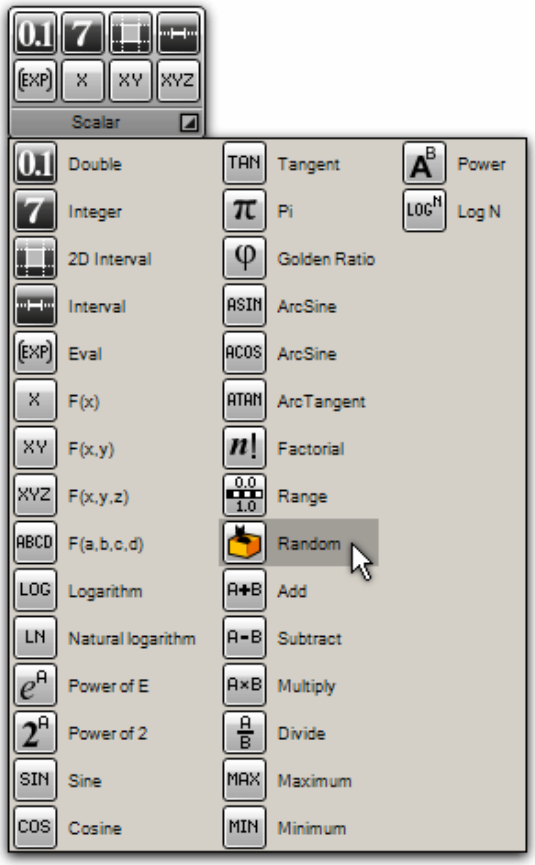
* Source: RhinoWiki

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

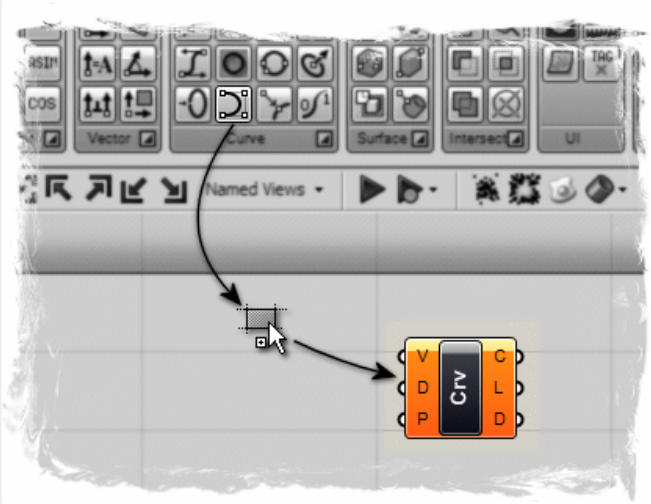
C. Component Panels 运算器面板

这个面板里包括了所有的运算器目录。各个运算器都在相应目录中（例如"Params"目录里是所有原始数据类型，而"Curves"中是所有相关的曲线），而且各个目录都可以工具栏面板里找到。工具栏的高度和宽度都是可以更改的，以适应不同数量的按钮。

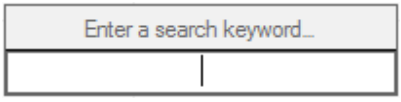
工具栏面板里包含了所有目录中的运算器。由于有一些运算器并不是常用的，所以在工具栏面板中只显示最近用的几个运算器。若要检查所有的运算器，你可以点击面板下方的按钮。



这样便会弹出一个提供了所有运算器按钮的目录面板。你可以在弹出的面板中点击运算器按钮，亦可以直接把按钮拖到工作区（即 Grasshopper 的窗口）上。在目录面板上点击运算器会把该运算器加入到工具栏中以方便接下来的使用。**但点击按钮不会使该运算器列入工作区中**！你**必须**通过拖曳它们至工作区中。



也可以通过输入其名称找到运算器。在工作区的任何位置双击鼠标键会弹出一个搜索对话框，输入你需要的运算器的名称，便会出现一个满足你需求的参数或运算器的列表。

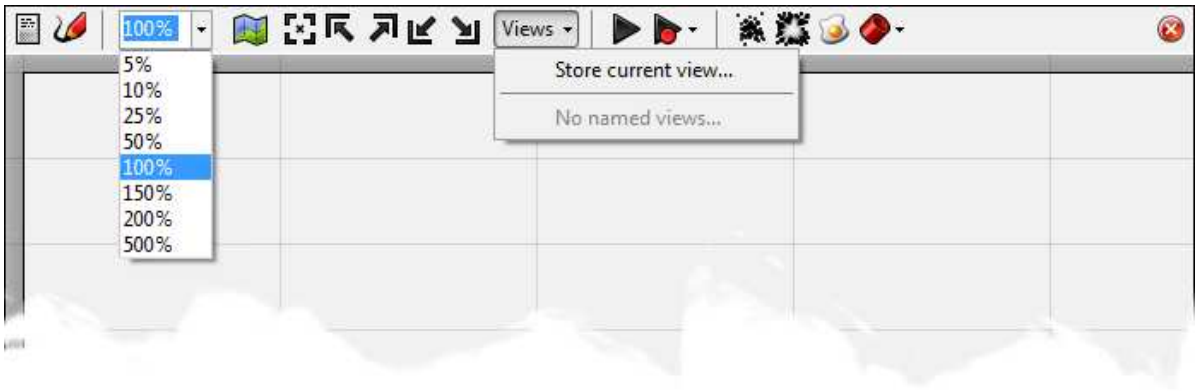


D The Window Title Bar: 窗口标题栏

编辑器的窗口标题栏与常见的windows的窗口的使用方法不同。如果窗口没有最小化或最大化，双击标题栏会收起或展开该窗口。这是一个在 Rhino 和插件间切换的的好办法，因为这样不需要把窗口移到屏幕最下方或者其它窗口的后面就可以直接最小化窗口。注意：如果你关掉了编辑器，Grasshopper 的预览窗口会在视图中消失，但它并不是真的被关闭了。下一次输入 _Grasshopper 的命令时，该窗口及其数据和装载的文件会重新出现。

E The Canvas Toolbar: 工作区工具栏

工作区工具栏提供了常用功能的快捷方式。通过菜单也可使用所有的工具，而且你可以根据自己的喜好选择隐藏工具栏（它可以在 View 菜单中重新激活）。



工作区工具栏包含了下列的工具（从左至右）

1.特性定义编辑器
2.草图工具

草图工具的使用跟Photoshop的铅笔工具和windows的画图工具一样。默认的草图工具设置可以进行改变，如线宽、线型和颜色。但是，它很难画出直线或预设的图形。为了解决这个问题，在工作区上画线后，右键点击该线，选择"Load from Rhino"，然后选择Rhino中预设的图形（可以是2D的图形如矩形、圆、星形等）。选择好了所需图形后，敲击回车，你先前所画的草图线就会用Rhino中设定的图形取代。

3.默认工作区显示比例
4. 导航地图

弹出一个缩略的视窗，以便快速得在工作区上移动。这个功能和 Photoshop 中的缩略视窗很类似

5.放大工作区显示比例（软件将为适应屏幕大小而自动调整画面大小）
6.查看角落（这四个按钮会把屏幕中心移到工作区的四个角落）
7.定义的视角（包含了一个储存和载入已设定视角的菜单）
8.重建命令（强行启动重建一个历史定义）
9.事件重建（默认状态下，Grasshopper 会对 Rhino 和工作区上的变化做出反馈。你可以通过这个菜单关闭反馈）
10 组合群组（把所有选中的对象变成一个群组）

工具尚未完成，预计将在未来发布的版本中完成，请用户在谨慎使用

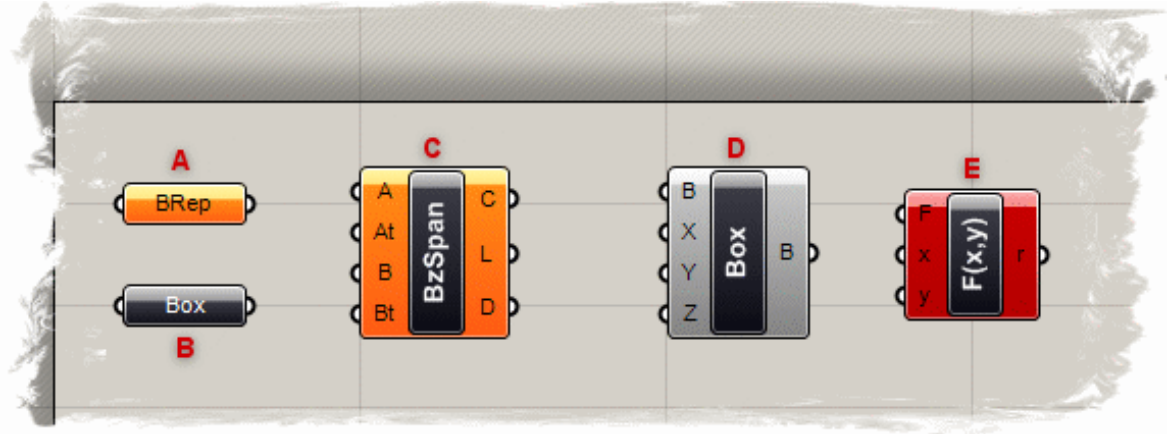
11.解散群组（把选中的群组炸开）

工具尚未完成，预计将在未来发布的版本中完成，请用户在谨慎使用

12.烘焙工具（把选中 Grasshoper 对象的变成 Rhino 中的对象）
13. 预览设置（Grasshopper 对象默认设置为预览显示。你可以在单个对象的基础上取消预览显示，但你亦可以对所有的对象进行预览。如果有很多曲面或碎面的话，关掉着色预览会大幅加快显示速度。
14.隐藏按钮。这个按钮可以隐藏工作区工具栏，亦可通过 View 菜单显示回来

F: The Canvas（工作区）

这是实际你定义及编辑各物件关联的编辑器。工作区里包括所有关联的对象和用户界面工具 G。工作区上的的对象通常根据它们的性质而以不同颜色显示。



A 参量。 设定中出现了问题或警告的参量将会以橘色方盒子的形式呈现。大多数被你拖进工作区里参量由于没有进行数据定义都显示为橙色。

B 参量。 没有错误和警告的参量（即正常参量……）

C 运算器。 运算器是一个较复杂的对象，因为它连接了输入和输出的参量。图中所示的运算器至少有一个关联的错误。你可以根据各个对象间的关系找出错误和警告所在。

D 运算器。 没有错误和警告的运算器。

E 运算器。 至少存在一个错误的运算器。错误可能来自运算器本身或者它所链接的输入/输出参量。在接下来的章节中我们会对运算器的结构有更多的了解。

所有被选中的对象将会以绿色呈现（图中未示）

G: UI Widgets 用户界面工具

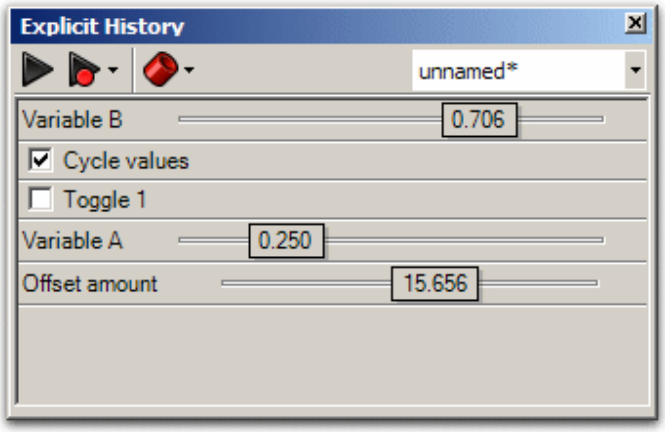
目前，目前仅有的用户界面工具是在工作区右下角的罗盘。提供了一个图像导航工具，显示你目前视角与整个定义之间的关系。这个工具可通过 View 菜单被启用/取消。

H: The Status Bar 状态栏

状态栏是选中的物体和插件中主要操作的反馈。你可以通过右击状态栏的省略号来查看历史操作。
状态栏左下方的橘色方形图标是最近加入到这个界面来的。左击这图标，Grasshopper 用户群的相关网站将被打开。你可以进入 http://Grasshopper.rhino3d.com 访问 Grasshopper 用户群。

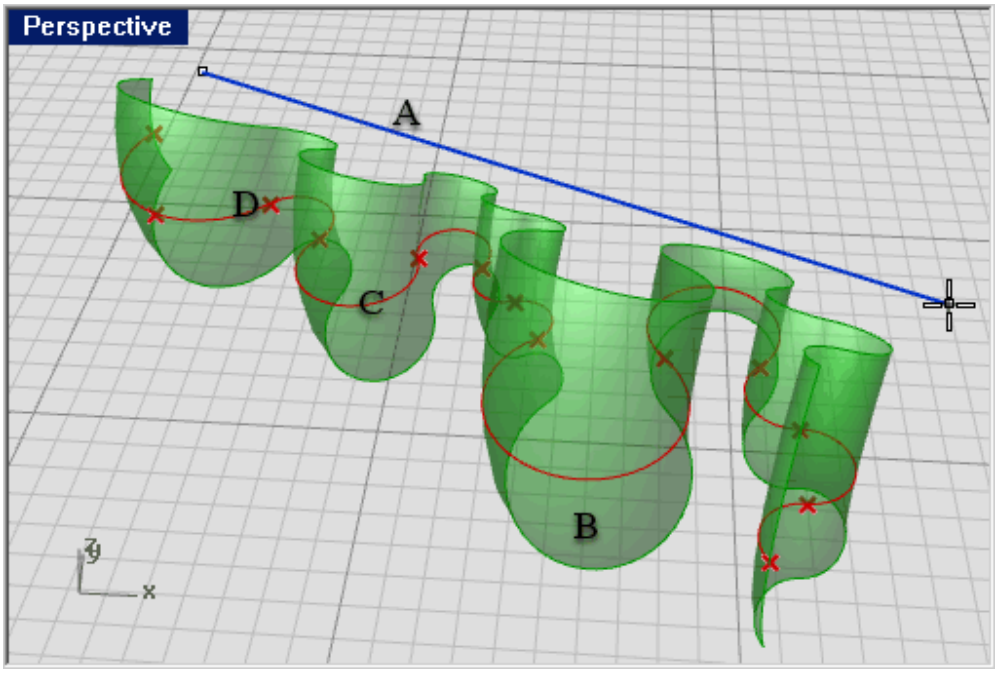
The Remote Control Panel: 远程控制面板

由于历史记录的窗口过大，你可能不想让它一直显示。当然可以最小化或关闭窗口，那样就无法再方便使用了。如果你想用一个小界面用于记录操作的关联值，可以启动远程控制面板。面板记录了所有滑动和逻辑关系（将来可能会有更多可能的记录值）。



远程对话框也提供基本预览、事件和文件转换控制。你可以通过主窗口“查看”菜单或 Grasshopper 面板提示栏开启远程对话框。

Viewport Preview Feedback: （视角预览显示）



a) 蓝色　　显示正在被鼠标选取的图形
b) 绿色　　视图中刚才选取的运算器图形
c) 红色　　视图中刚才未选取的运算器图形
d) 点图形　　用叉而非矩形点来表示这是 Rhino 中的点

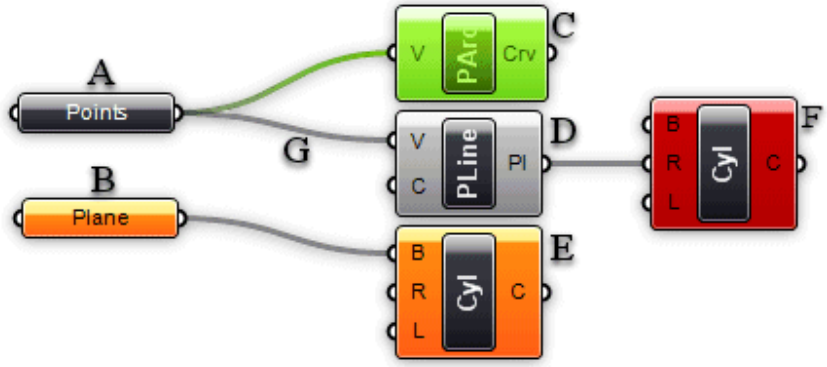
3 Grasshopper Objects*（Grasshopper 对象）

Grasshopper Definition Objects　Grasshopper 对象关联

Grasshopper 由多种不同的项目组成，但一开始你只需熟悉其中两种：

Parameters	参数
Components	运算器

参数包含数据——存储信息，运算器包含动作——处理信息。下面的图示说明你在 Grasshopper 关联中可能遇到的一些项目：



A）包含数据的参数。如果其左边没有连接线说明没有从任何地方获得数据。带有横向字体的细黑模块说明参数没有错误或警告。

B）未包含数据的参数。在关联过程中任何不含数据的项目只会浪费时间和金钱，因而所有数据（一旦被添加）都将显示为橙色用以说明不包含任何数据，并且对输出结果不起作用。一旦参数接受或关联其他数据，就会变为黑色。

C）已选运算器。显示为绿色

D） 正常运算器

E） 含警告的运算器。大多情况运算器都有大量输出和输入参数，因而无法清楚得知哪个项目使其产生警告。也可能是产生若干个警告。你得通过扩展菜单（见下文）追查问题所在。**注意：问题不一定非得全部解决**。也许问题是正常情况下产生的。

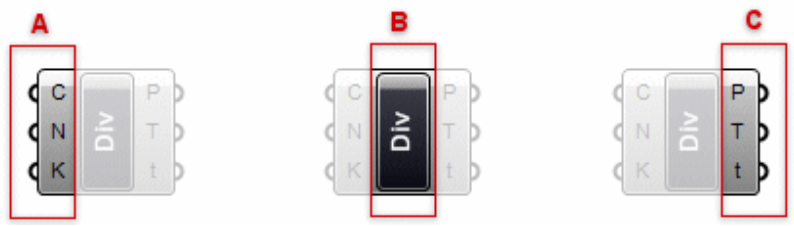
* Source: RhinoWiki
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryPluginObjectsExplained.html

F） 含错误的运算器。和警告类似，很难知道错误从哪里产生。必须用扩展菜单（见以下）。注意如果运算器既有警告又有错误将会以红色显示，错误颜色优先于警告颜色显示。

G）连接。输出、输入参数中含有连接。对于任意运算器来说没有连接的上限，但必须保证不出现循环连接。一旦出现循环的情况，在所有相连的运算器中的第一个将显示错误信息。要了解关于更多连接的信息，参加 Data Inheritance 这一章节。

Component Parts　运算器各部分

运算器通常需要数据来进行活动，并产生一个相应的结果。这就是为什么大多数运算器都有一系列的参数，包括相应的输入和输出参数。输入参数位于左边，输出参数位于右边。

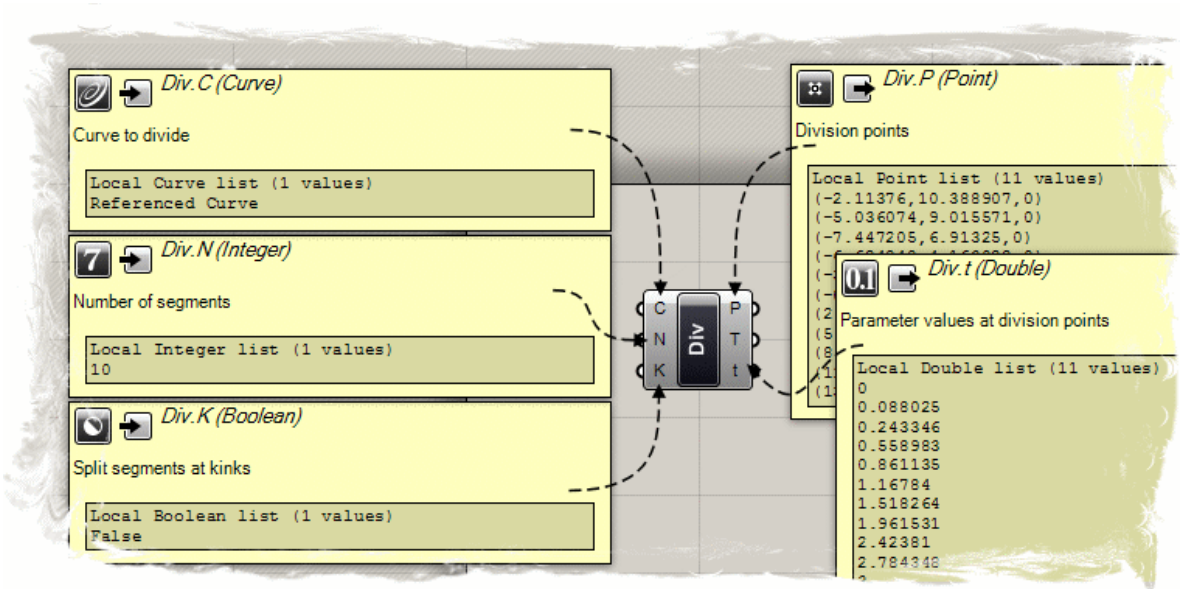


A）分类运算器的三个输入参数。缺省情况下参数名称很短，可以任意改变。

B）分类运算器区域 （通常含有运算器的名称）

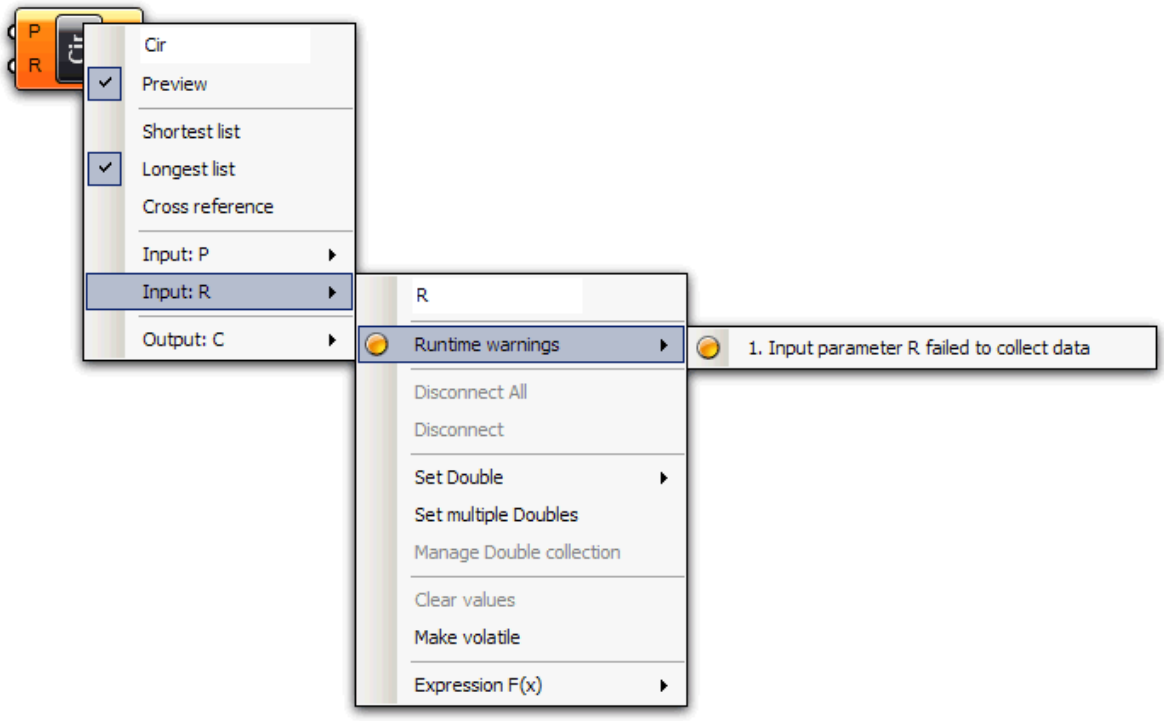
C）分类运算器的三个输出参数

当鼠标停留在运算器项目的不同部分时，你会看到不同的工具条用以说明此位置的特定项目类型。工具条同时现实其类型以及个体参数的数据：



Using Context Popup Menus 使用扩展弹出菜单

工作区上的所有项目都有自己的扩展菜单显示特定运算器的大部分特征。不能完全依赖运算器的信息，因为他们同时显示其包含的子项目的所有菜单。例如一个运算器变为橘红色可能说明是由其附属的一些参数导致的警告。如果你要找出错误，就得使用运算器扩展菜单：



这里你看到的是以 R 为名称的输入参数的主要运算器菜单。菜单通常开始会列出一个包含问题所在对象名单的文字编辑区域。你也可以将名字改变得更容易识别，但缺省状态所有名字都是用屏幕实际用名的缩写表示。菜单的第二项表示项目产生或定义的图形是否要在 Rhino 的视图中显示。关掉一些不含重要信息的运算器能够减少 Rhino 图形显示及过程运算所需的时间。如果参数或运算器不可用，就会以浅白色填充来表示。不是所有的参数/运算器都能在视图中显示，因而预览项目不常被用到。

R 的输入参数扩展菜单含有橘红色警告图标，包含了一列（这里只有 1 项）产生这个参数的警告。

4 Persistent Data Management*（静态数据管理）

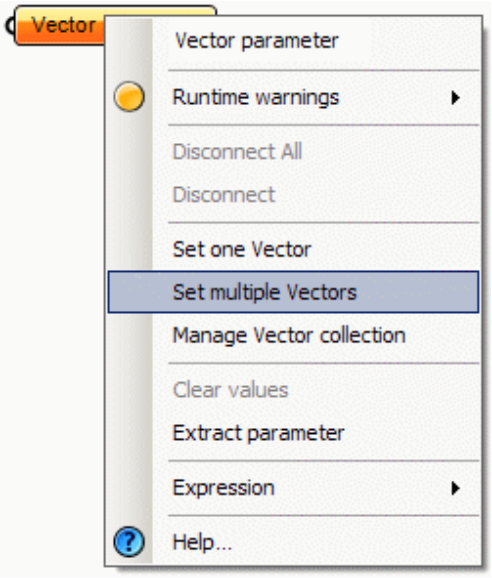
数据类型

参数是用来储存信息的，但参数可以用来储存两种不同类型的数据：动态数据和静态数据。动态数据是从一个或者多个参数上继承而来，一旦一个新的运算开始时即被删除。静态数据是一种特殊的被用户自定义的数据。每当一个参数被连上一个目标之后，静态数据将被忽略而不是被删除。(在这里输出参数是个例外，它既不能储存参数也不能定义参数的来源。输出参数完全由他们的组成单元控制)

静态数据可以从菜单中取出，并且根据不同参数有不同的操作。以 Vector 参数为例，则允许你在菜单里设定一个和多个向量。

但让我们往回几步来看看，来看看默认的 Vector 参数是怎样变换。一但你把它从运算器面板拖拽至工作区上，你将看到如下图所示的变化

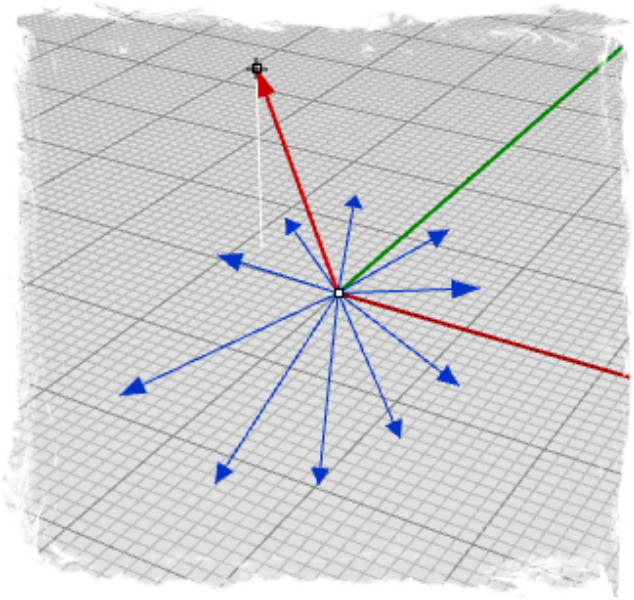
这个参数是橘色的，表示警告。没有关系，它在这里只是告诉你这个参数是空的（他不包含静态数据也没有和动态数据相连接）因此也不影响结果和过程。这种参数的菜单提供两种设定静态数据的方法，Single 和 Multiple



一旦你单击这些选项中的任意一个，Grasshopper 的操作窗口将消失，同时会让你在 Rhino 视窗中拾取一个向量。

* Source: RhinoWiki

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



当你定义完所有需要的向量之后，按回车（Enter）键，它们将成为参数静态数据的一部分。这意味着参数现在已经不是空的了并且从橘黄色变为灰色

在一个点上你可以使用无限多的参数去定义相同的一个向量

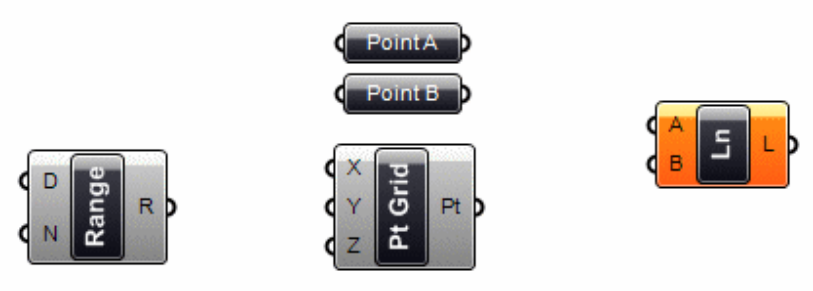
5 Volatile Data Inheritance*（动态数据继承）

数据继承

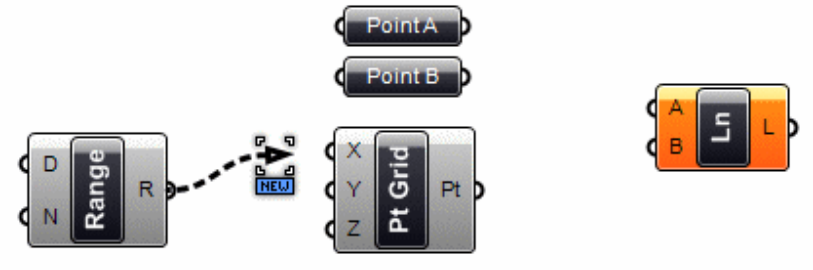
数据被储存在参数之中（不论是动态数据还是静态数据）并且被用于运算器之中。如果数据不被存在永久记录的参数中，那么它必定从其他地方继承。每一个参数（除输出参数之外）都定义着它继承数据的地方并且多数参数都是这样。你可以插入一个双精度参数(仅表示具有相同小数位的数字)到一个整型数据源且将要进行转换处理。插件提供了很多转换方案，但如果没有进行转换，这个参数将在接受端发生错误。例如，如果你将一个 Surface 连接到需要 Point 的地方，这个 Point（点）参数将产生错误信息（通过菜单中的参数询问）并且参数块变红。如果这个参数属于一个运算器，那么这种红色的状态将传递到同运算器的各个参数上，即使它自己本身并不没有错误，整个运算器都会变成红色。

连接管理

因为参数有其本身的数据来源，你可以通过参数询问面板设定这些数据。假设我们有一个小的程序段包含三个运算器和两个参数：



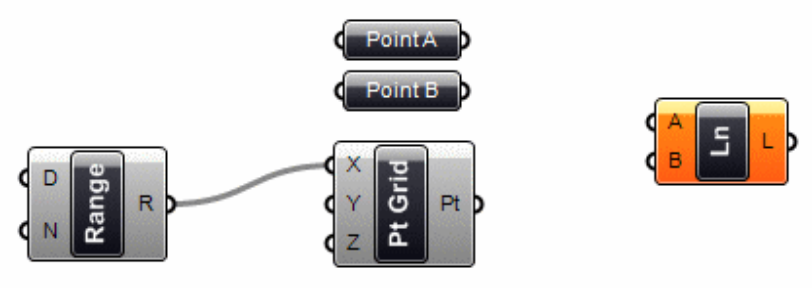
在这个阶段，所有对象都是无关的，我们需要对他们进行连接。不管我们怎么做，只要将他们从左到右连接即可。如果你看是拖动参数边上的小圆圈（我们叫它“把手”），一个连接线将出现并依附于光标：



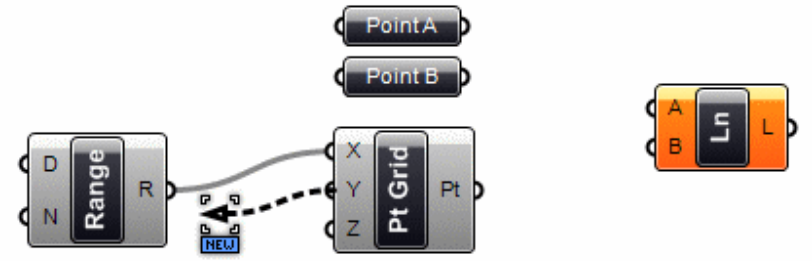
一旦光标（按住鼠标左键不松）靠近一个可用参数，连接线将被吸附并固定住。松开鼠标左键，确定这个连接的建立：

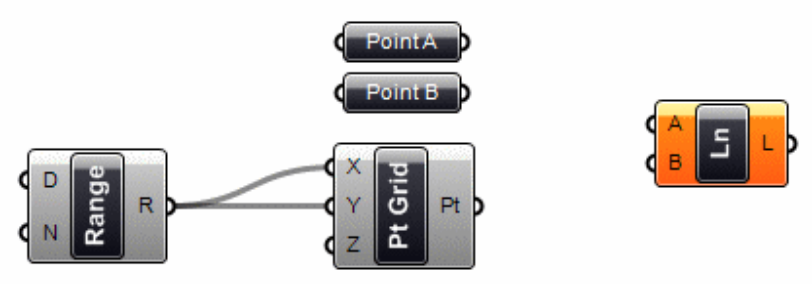
* Source: RhinoWiki

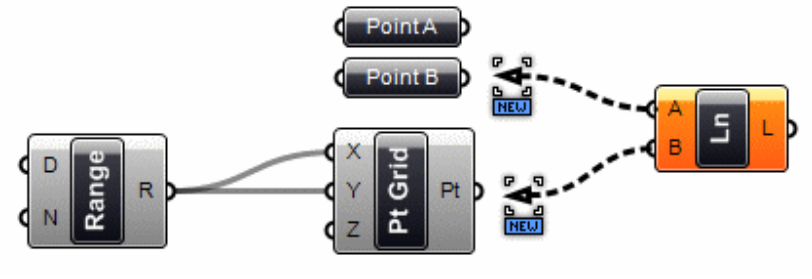
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryVolatileDataInheritance.html

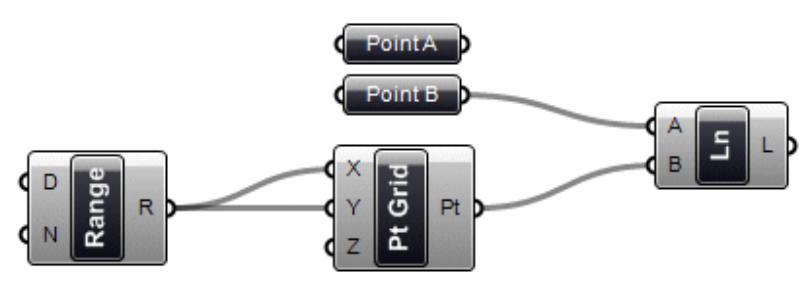


同样，我们也可以对 PtGrid 运算器中的“Y”参数、“Point A”和“Point B”参数做同样的动作：单击+拉出+松开…

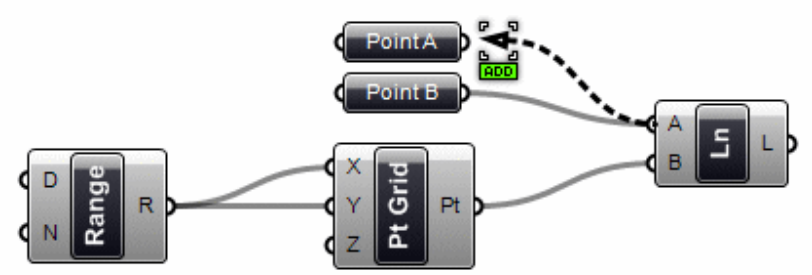


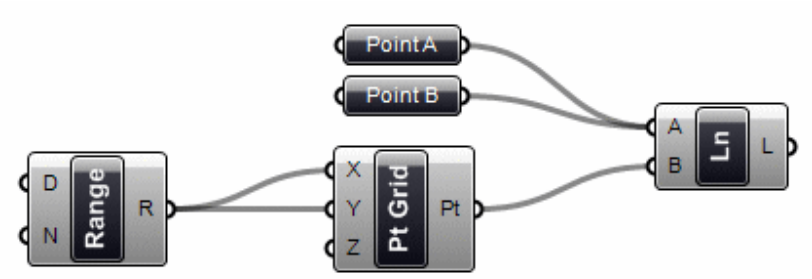






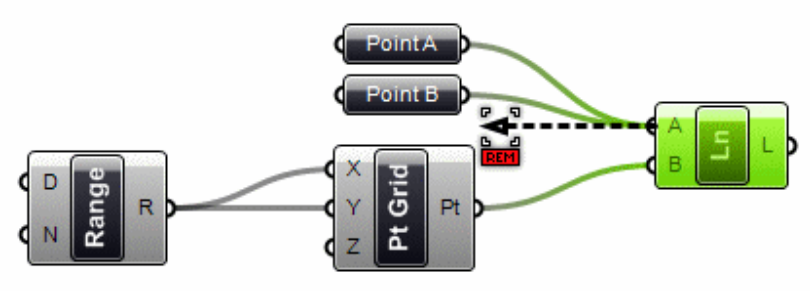
我们可以在两个方向都联系起来。但是要注意在默认状态下一个新的链接会清除掉已有的链接。由于我们假定你大多数情况下只是用单独的链接，你必须用些其他办法来定义多个来源。如果在拖曳链接线时按住Shift，将会增加一条链接线（如下图）：

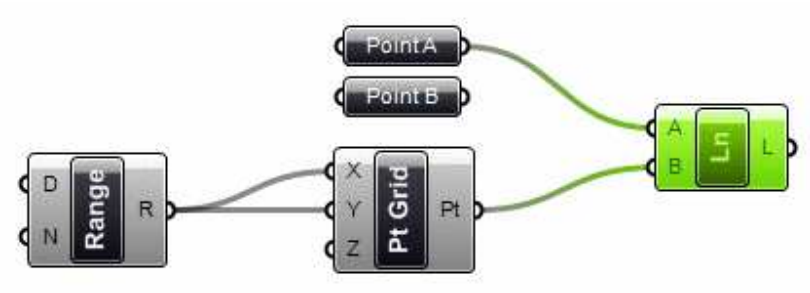




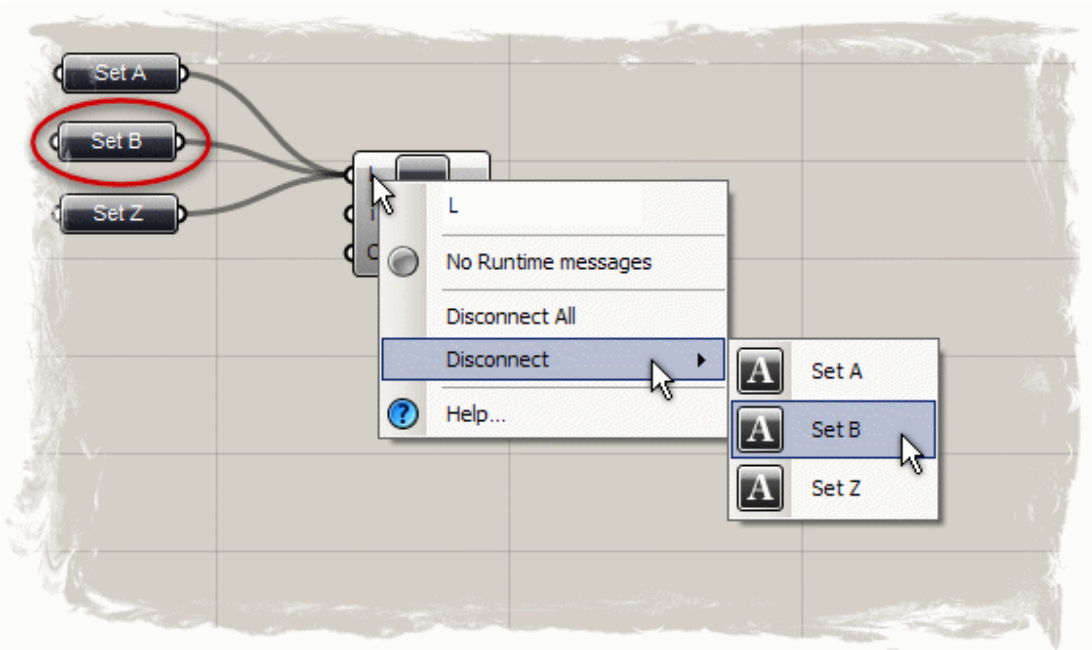
如果“ADD”的光标（如上图）是被激活的状态，当你把它链接到一个源参量时，这个参量将会被列入源列表。如果你需要特别定义一个已经被定义为源的源参量，将不会有响应。你不能从同一个源链接两次。

同样的，如果你按住 CTRL，"REM"光标将会出现（如下图），而且目标源将会从源列表中删除。如果该目标没有被链接或参用，将不会有响应。

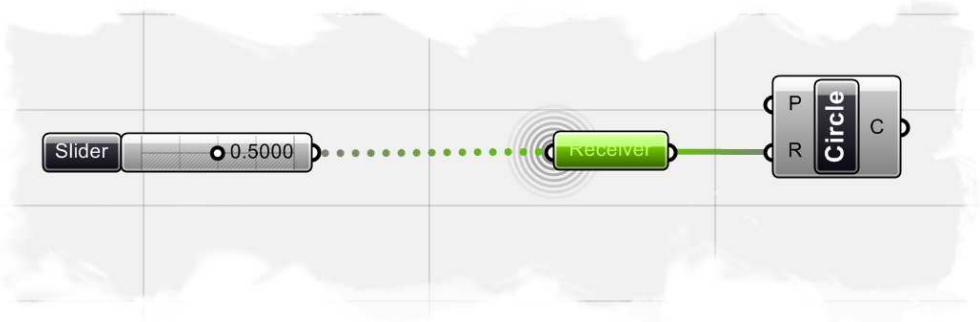




你也可以通过参量菜单取消源的链接（如下图）



Grasshopper 也可以通过不可见的连接线传递数据，这种运算器可以在 Params 中的 Special subcategory 里找到。你可以连接 Receivers 运算器，就像其他的运算器一样。当你在建立连接之后释放鼠标左键，连接线自动消失。这样是因为 Receivers 运算器默认的设置是仅在被选择是显示虚隐的连接线。你可以右击运算器并设置连接线只在 Receiver "selected"的时候显示，或者"always" ，"never"显示。对于输出端可以像其他运算器那样连接。



这里，虚隐的连接线被显示是因为 Receiver 运算器处于被选择状态

Receiver 运算器输入项前的数字 1 表明这是一个输入项的连接点。由于运算器没有被选中，连接线将不被显示出来（但是数据仍然是可以进行传递的）

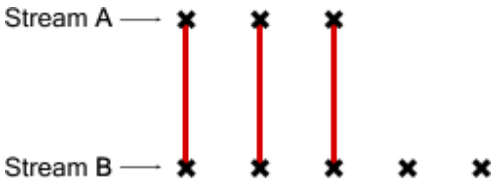
.

6 Data Stream Matching*（匹配数据流）

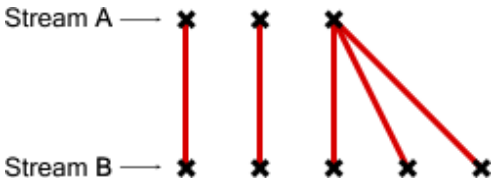
Data matching 数据匹配

数据匹配是一个没有明确解决方案的问题。当一个运算器和不同规模（下图解释的更清楚）的输入数据进行映射的时候，数据匹配问题就会产生。比如一个通过不同点生成线段的运算器。这个运算器必须链接两个提供点坐标的参量（数据流 A 和数据流 B）。参量的数据来源之间没有关系，而运算器不能“看到”它链接的输入参量和输出参量以外的东西：

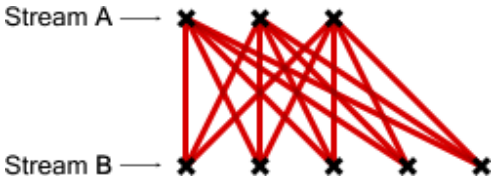
正如你所看到的，在 A 和 B 间连接线段的方式有很多。Grasshopper 目前支持 3 种匹配规则，但是其他规则可能也是可行的。最简单的方式是一对一的链接，直到某一数据流中已没有数据。这被称为“Shortest List”规则：



而“Longest List”规则是一直链接输入参量，直到所有的数据流都没有数据。这是所有运算器的默认链接规则：



最后，“Cross Reference”规则是把所有可能的链接都连接上：



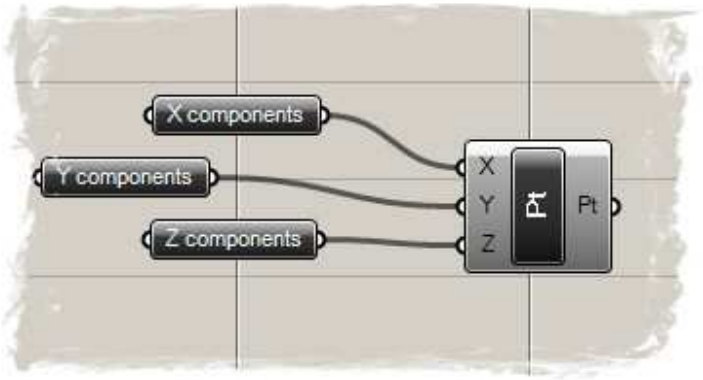
这存在着潜在的风险，因为输出参量的数目可能变得很大。如果涉及到更多的输入参量而这些不稳定的数据开始进行迭代同时链接规则没有改变的话，这个问题变得更加难以解决：

* Source: RhinoWiki

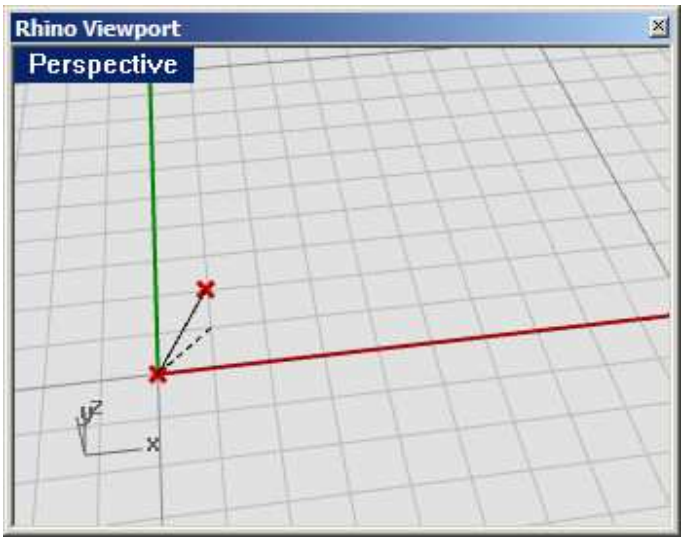
http://en.wiki.mcneel.com/default.aspx/McNeel/ExplicitHistoryDataStreamMatchingAlgorithms.html

比如我们有一个点运算器，它从另一个包含 X,Y,Z 坐标的参量获取如下数据

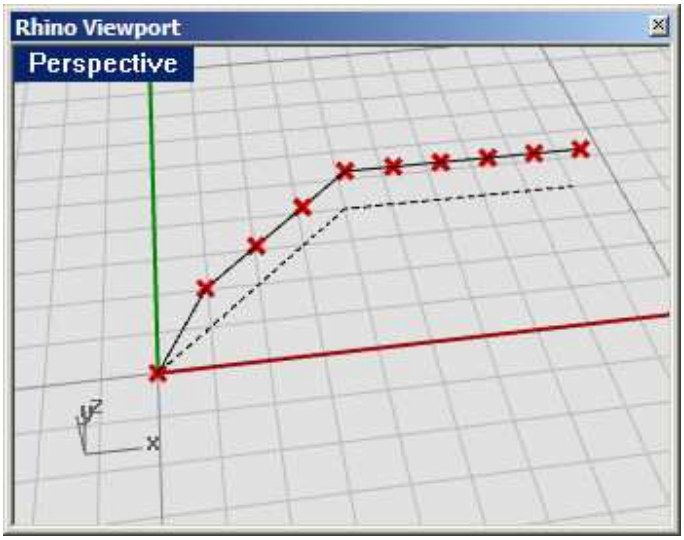
X 坐标: {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0}
Y 坐标: {0.0, 1.0, 2.0, 3.0, 4.0}
Z 坐标: {0.0, 1.0}



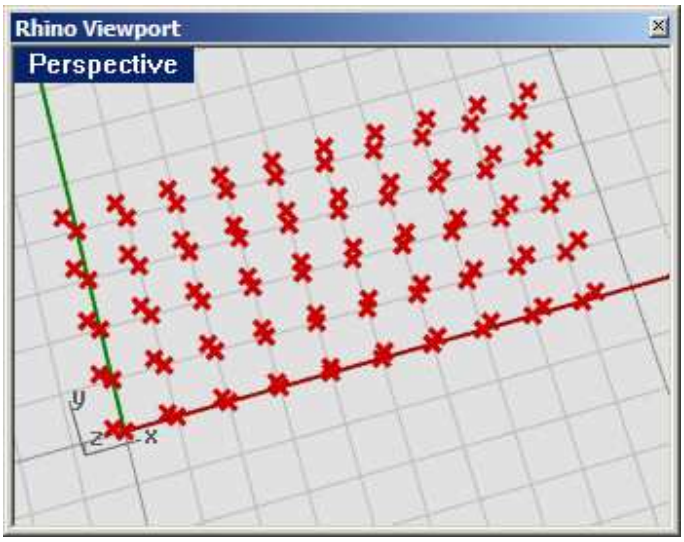
如果通过“最短排列”规则链接这些数据，我们将只能得到两个点，因为 Z 参量只含两个值。这是这些数据的最少的搭配方式：



而“Longest List”规则会生成十个点，重复使用 Y,Z 坐标值直到 X 坐标值全部匹配过。



"Cross Reference"规则会把所有的 XYZ 值全部互相关联，所以最后会有 10X5X2=100 个点



每个运算器都可以被设置为服从某一个规则（通过右击运算器图标弹出的菜单进行设置）

只有一个特别的例外。有些运算器需要从一个或多个输入数据中获得一系列的数据，例如，生成一条复合线需要一系列的点数据。点数据越多，产生的复合线越长，而不会产生多条复合线。这些需要多个值的输入参量被称为列表参量，而且在数据匹配的过程中，它们是不被考虑的。

7 Scalar Component Types（数据运算器类型）

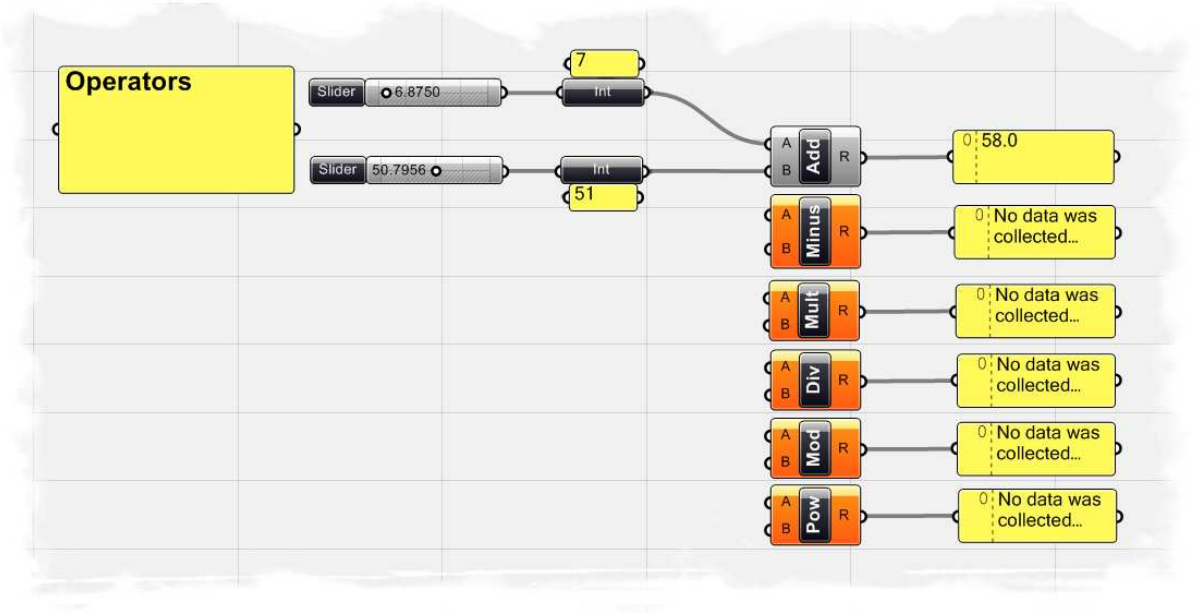
数据运算器类型是在一些数学运算中特别使用的，它包括有：

A) Constants（常数） 返回一个常数例如π，黄金分割比，等等
B) Expressions（表达式） 用来生成一个或多个不同的函数（或数学规则）
C) Intervals（区间） 用来确定两个数据极值间的区域
D) Operators（运算符） 用来定义数学运算符号，如加减乘除
E) Polynomials（多项式） 通过幂运算等运算赋值
F) Trigonometry（三角函数） 返回三角函数的运算结果，如正弦，余弦等
G) Utility (Analysis) [实用工具（分析）] 。用于评价的两个或两个以上的数值

7.1 Operators（运算器）

正如之前所言，运算符运算器是一些根据两个输入数值通过一定规则或函数进行运算出结果的运算器集合。为了更清楚的了解运算运算器，我们通过一个简单的数学定义去探索不同的运算运算器类型。

注意：要打开该定义的完整版本，打开文件随本文附送的源文件夹中的 Scalar_operators.ghx 文件，如下是完整定义的截图



从零开始创建定义

- Params/Special/ Numeric Slider -拖曳一个数字控制条运算器到工作区上
- Right click the slider to set: 右击控制条进行设置
  - Lower limit: 0.0 最小值
  - Upper limit: 100.0 最大值
- 当前值：50（注意：这是个任意的值，并且可在最大最小值间随意变动）
- 选中控制条通过复制(Cntrl+C)粘贴(Cntrl+V)复制一个控制条
- Params/Primitive/Integer-拖曳两个整数运算器到工作区上
- 把控制条 1 链接到第一个整数运算器上
- 把控制条 2 链接到第二个整数运算器上

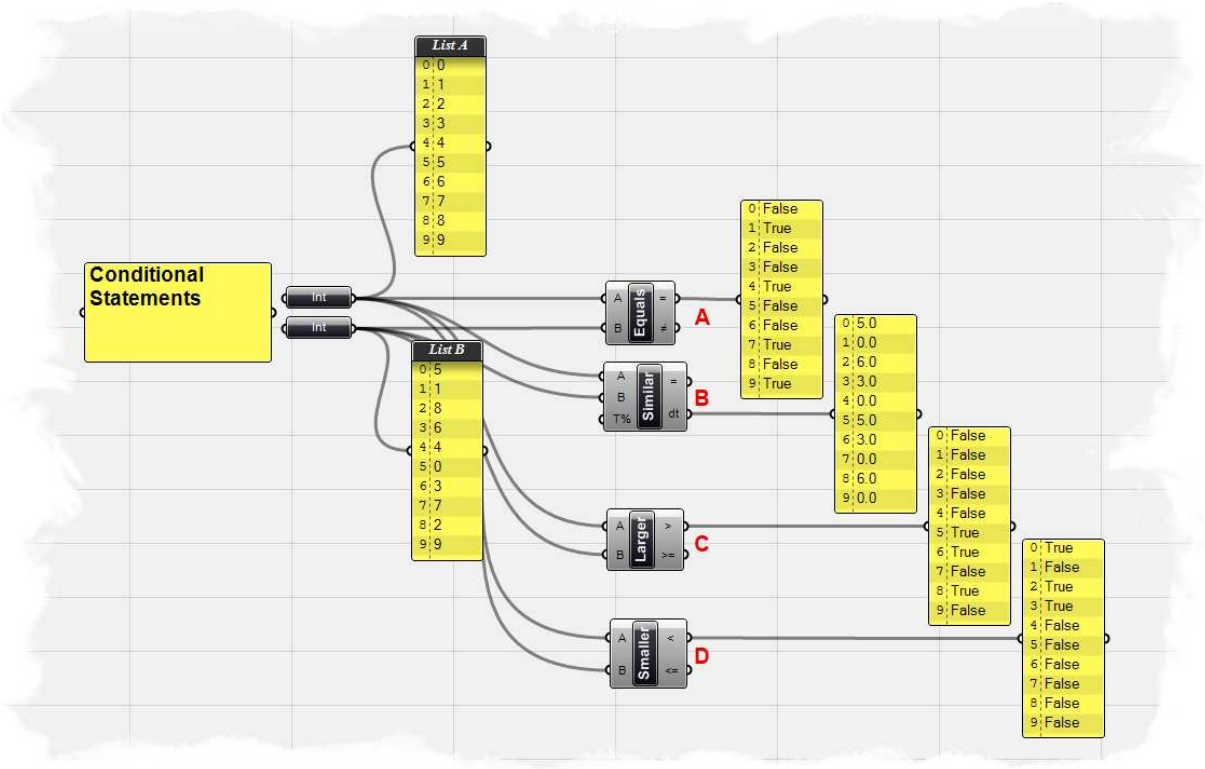
  这个控制条的默认数据类型是浮点型（十进制运算值）。通过链接控制条和整数运算器（Integer component），我们可以把数据类型变成整数型。当我们链接一个“Post-it”面板到每个整数运算器的输出值时，我们会看见随时的变化。左右移动控制条可以发现浮点数字变成了整数。
- Scalar/Operators/Add-拖曳一个加法运算器到工作区上
- 链接第一个整数运算器到加法运算器（Add）的输入 A 端
- 链接第二个整数运算器到加法运算器的输入 B 端
- Params/Special/Panel-拖曳一个“Post-it”面板到工作区上
- 链接加法运算器的 R 输出项到“Post-it”面板的输入项
- 你可以在“Post-it”面板看见两个整数的和
- 拖曳其他的运算运算器到工作区上
  - Subtraction 减
  - Multiplication 乘
  - Division 除
  - Modulus 绝对值
  - Power 幂
- 链接第一个整数运算器到每一个运算运算器的输入 A 端
- 链接第二个整数运算器到每一个运算运算器的输入 B 端
- 拖曳 5 个 Post-it 面板到工作区上并且分别与运算运算器链接起来。

定义已经完成，现在通过滑动控制条，观察 Post-it 面板的变化

7.2 Conditional Statements（条件语句）

你可能已经注意到 Scalar 下 Operators 子菜单中的运算器和上一个版本有些不同，是因为有 4 个运算器（0.6.0007 版本中新增）和其他的数学运算器有些不同的功能。它们用来比较两列数据而不是进行代数上的运算。这四个运算器是 Equality, Similarity, Larger Than，和 Smaller Than，下面进行详细的解释。

注意:想看完整版的定义，请打开自带文件中的 Conditional Statements.ghx.
以下是完整定义的截图



A) Equality　对两列数据进行操作，比较 List A 和 List B 的对应项。如果两个值是相同的，那么将得到一个 Ture 的布尔值，相反如果不相同则得到 False 的布尔值。运算器对列表数据的处理循环方式取决于设定的数据匹配类型（默认的是 Longest List）。运算器有两个输出项，第一个反馈一个布尔值的列表，显示两列数据中相同的数据项；第二个也是反馈一个布尔值的列表，但是现实的是两列数据中不相同的数据项。

B) Similarity　评价两列数据并测试两个数的相似性。它对列表的比较方式和 Equality 运算器相类似，但有一点不同，运算器多出一个百分比的输入项，用来定义 List A 和 List B 中允许出现的偏差。Similarity 运算器也有一个输出项，以表示两组数据相差的绝对值。

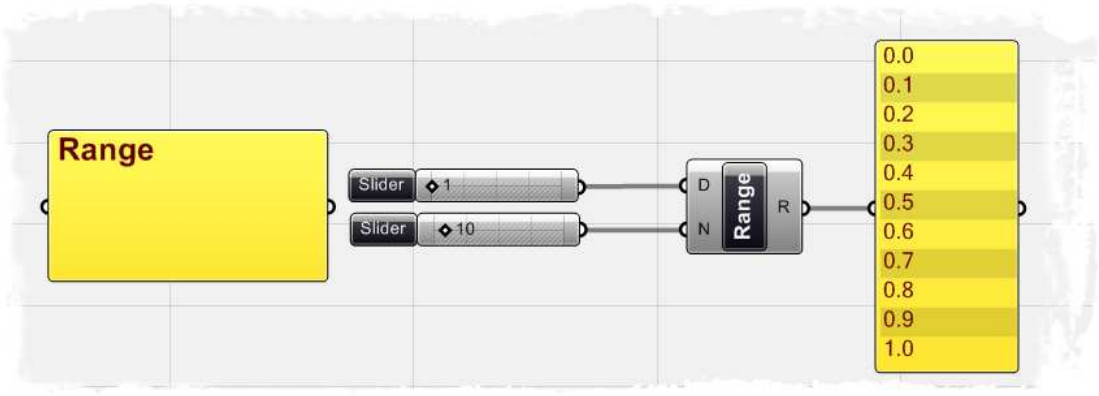
C) Larger Than　运算器需要两组数据并判断 List A 的第一项是否比 List B 的第一项大。两个输出项允许你更改对两组数据的的不同评判标准：大于（>）和大于等于（>=）。

D) Smaller Than　功能和 Larger Than 运算器相反。用来比较 List A 的第一项是否比 List B 的第一项小，并反馈一组布尔值。同样，两个输出项可供选择小于（<）和小于等于（<=）两个不同的评判标准。

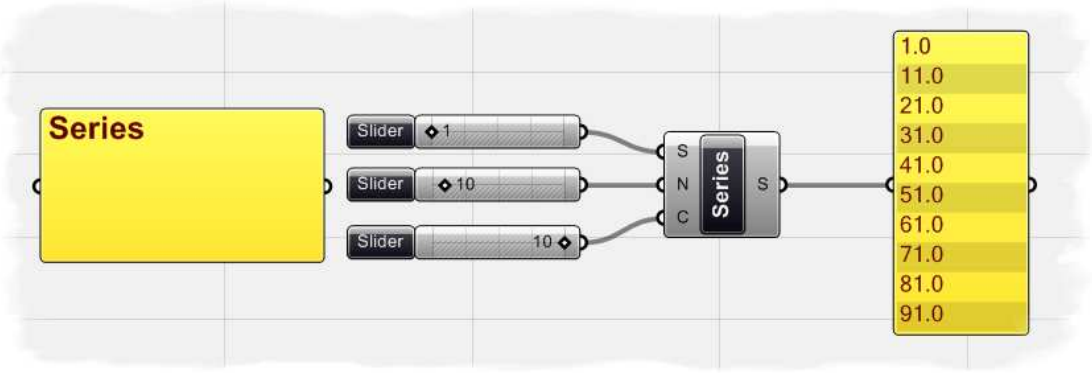
7.2 Range vs. Series vs. Interval(范围，数列，区间)

范围，数列和区间运算器都是两个数值极限之数列的集合，但这些运算器运作方式不同。

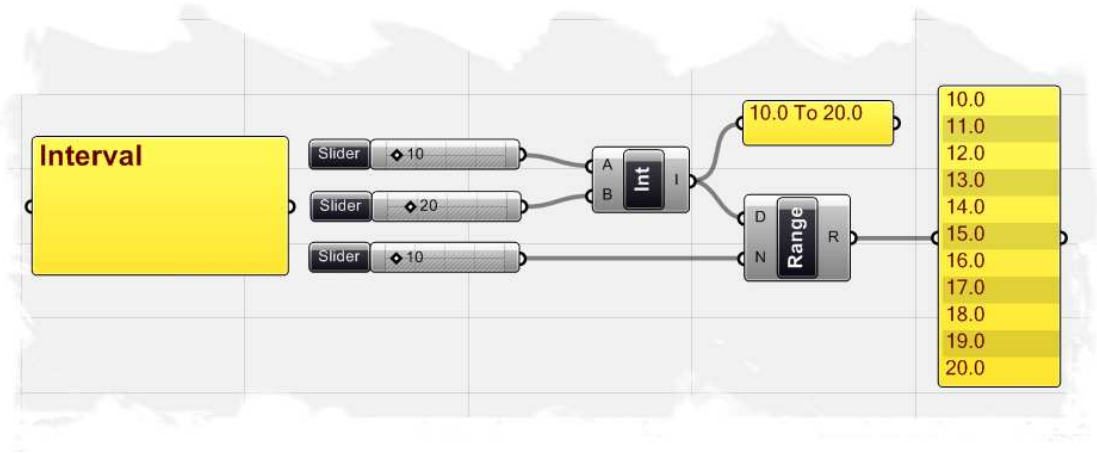
注意：要打开该例子的完整版本，打开文件随本文附送的源文件夹中 Scalar_intervals.ghx 文件.



范围运算器生成最大值和最小值的值域内间隔均匀的一系列数。比如上图，两个控制条链接到范围运算器的输入项。第一个控制条确定了范围的值域。在本例子中，值域被定义为 0 到 1.第二个控制条定义了值域内被分成多少个间隔，本例子中为 10 个。这样输出了 11 个数，依次从 0 增大到 1.（注意，第二个控制条所分的是数的间隔数，故产生 11 个数，而不是 10 个）



数列运算器生成一系列依据初值、增量和数的个数（即数列中的首项，公差和项数）定义的一段不连续的值。这个数列运算器的例子里，3 个控制条链接到数列运算器上。第一个，当链接到 Series-S 输入项，定义了数列的起始点。第二个控制条，设置为 10，定义了数列的增量。因为初值设定为 1 而增量设置为 10，故下一个值为 11.最后，第三个控制条定义了数的个数。由于数值被设定为 10，在 series 中定义的最终输出值显示出十个值，即以 1 开始，每次增加 10



Interval 运算器可以在一个最小值和一个最大值之间产生一系列可能的值。Interval 运算器类似于我们为定义的一个范围的数而使用的 Range 运算器。主要的差别在于
Range 运算器可以产生一个从 0 到一个输入值所形成的一个数据集合。而在 Interval 运算器中，最小值和最大值可以用输入值 A 和 B 开定义。上面的例子，我们使用了两个滑条来定义了一系列 10－20 之间所有可能的值。现在 Interval 的输出值显示为 10.0 到 20.0，这反映出我们新的定义域。如果现在把 Interval 的 I 输出与 Range 的 D 输入相连接，我们能在 Interval 中创造一系列新的值了。正如前面 Range 的例子那样，如果我们把 Range 的步数设为 10，我们将得到一组从 10 到 20 等间距的 11 个数。（我们有很多方式来定义一个 interval,你可以在 Scalar/Interval tab.中找到其他一些方法。在此，我们进介绍了一些简单的方法来定义 Interval,在随后的章节中我们会陆续介绍一些其他的方法）

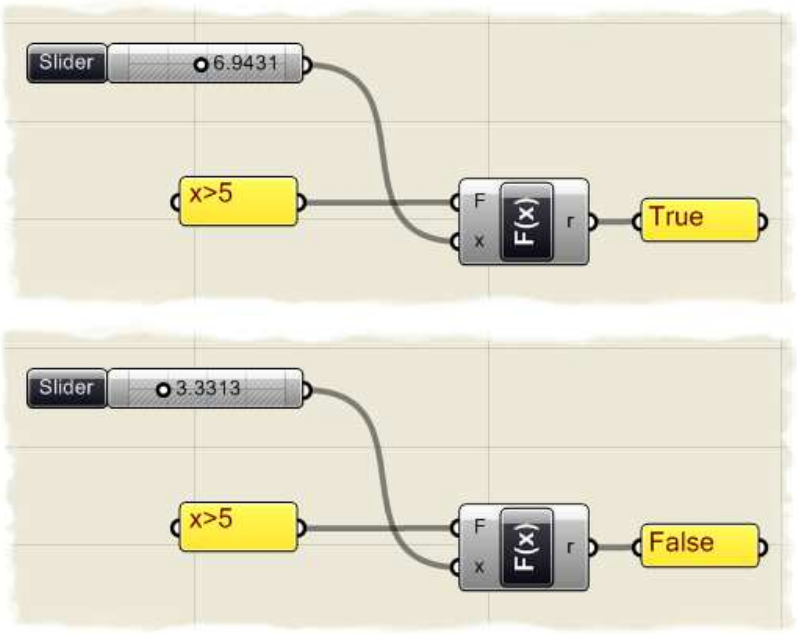
7.3 Functions & Booleans（函数与布尔）

几乎每种编程语言都有执行条件语句的方法。在大多数情况下，编程人员编写一段程序来问这样一个简单的问题：“如果…？”，如果 911 袭击从未发生？如果煤气费用为 10 美元每加仑？这些重要的问题其实代表着一种更高层次的抽象思维。电脑程序也有这样的本领来分析类似“如果…”这样的命题，并根据问题的答案进行相应的操作。下面让我们来看看一个非常简单的由一个条件程序翻译的语句：

如果这个物体是曲线，那么删除它.

这个程序首先检查这个物体是不是一条曲线，在来决定给它相应的布尔值：“True(真)”或“False(假)”，而没有中间状态。当布尔值为“True(真)”时，则物体为曲线；反之当布尔值为 False(假)时，该物体不是一条曲线。语句的第二部分是根据条件语句的结果进行相应的操作：如果物体为曲线，那么删除它；条件语句又叫做 If/Else 语句；如果物体满足一定的标准，就做相应的操作；否则，做其他的操作。

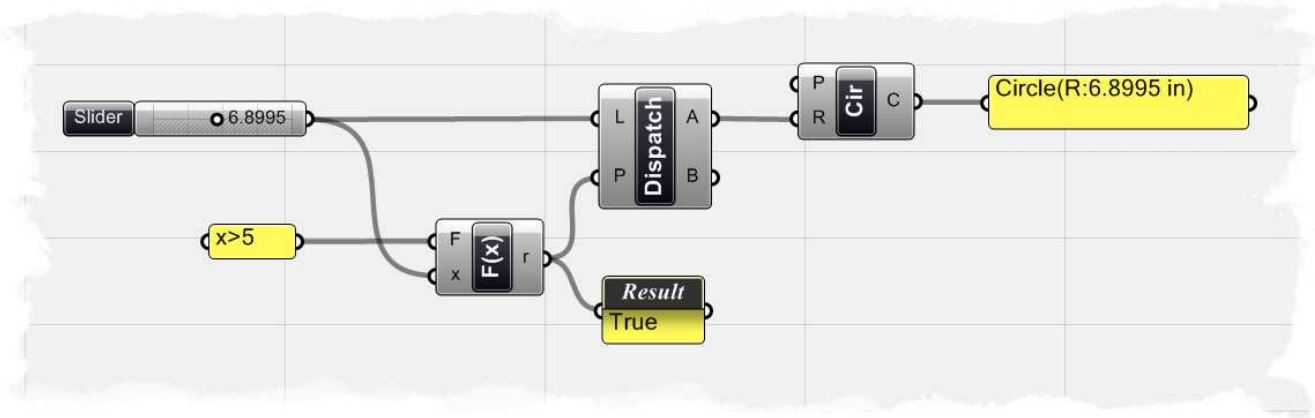
Grasshopper 也有这样的功能，它可以通过函数运算器 的使用来分析条件语句。



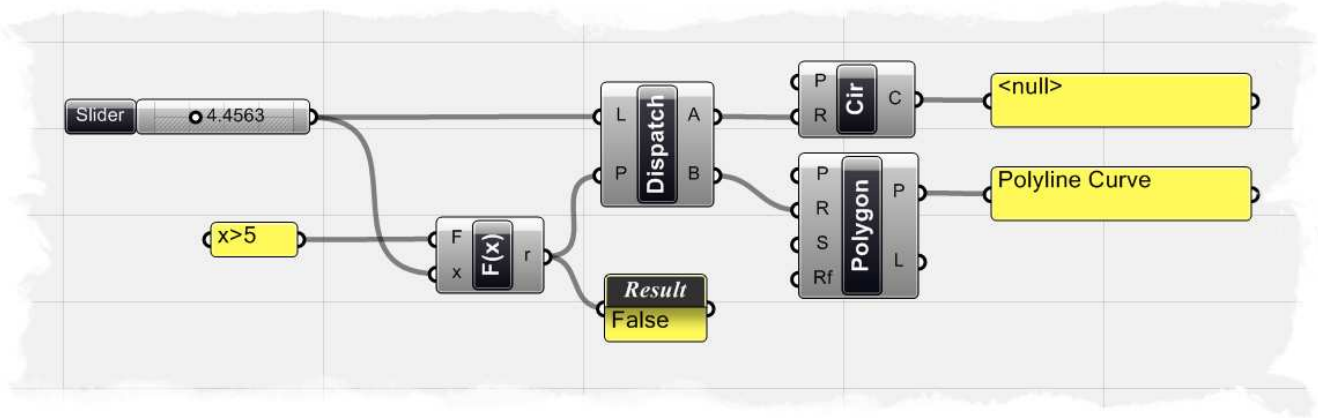
在上面的例子中，我们将一个数字滑块的 X 输入同一个单一变量函数运算器相连（Scalar/Expression/F1）.此外，一个条件语句同函数的 F 输入相连，定义这样一个问题“x 比 5 大么？”如果数字滑块设定的值比 5 大，那么函数的 r 输出会显示布尔值“真”。
相应的如果数字滑块的布尔值设为 5 以下，那么 r 输出将会变为“假”。

一旦我们定义了一个函数的布尔值，我们可以把布尔值“真、假”信息传递给 Dispatch 的 p 输入来进行某个特定的操作。Dispatch 运算器工作时输入一系列信息，如下例所示，我们将数字滑块的信息与 Dispatch 的 L 输入相连，并按照 Dispatch 的 P 输入的布尔值来筛选信息。如果图框中显示为“真”值，则这列信息将会传递给 Dispatch 的 A 输出。如果显示为“假”，Dispatch 将会把信息传递给 B 输出。对于这个例子，我们决定仅当 x 的值大于 5 时创造出一个园。我们把一个 Circle 的运算器（Curve/Primitive/Circle）同 Dispatch 的 A 输出相连，因此当通过 Dispatch

的布尔值为“真”时，一个用数字滑块控制半径的圆将被建造出来。因为没有运算器同 Dispatch 的 B 输出相连；如果布尔值为“假”，什么事都不会发生，圆也不会被创造出来



我们可以通过连接一个对这个定义进行一下小小的深化，连接一个 N-sided Polygon (Curve/Primitive/Polygon)运算器到 Dispatch 运算器的 B 输出项，确保连接的 Polygon 运算器的 R 输入项可以定义多边形的半径。现在如果 number slider 的数字小于 5，那么一个以 number slider 数值为半径的五边形将被创建出来。如果滑竿数值大于 5，那么将创建出一个圆形。通过这种方式，可以在定义中创建多个 If/Else 语句以满足我们的需要。



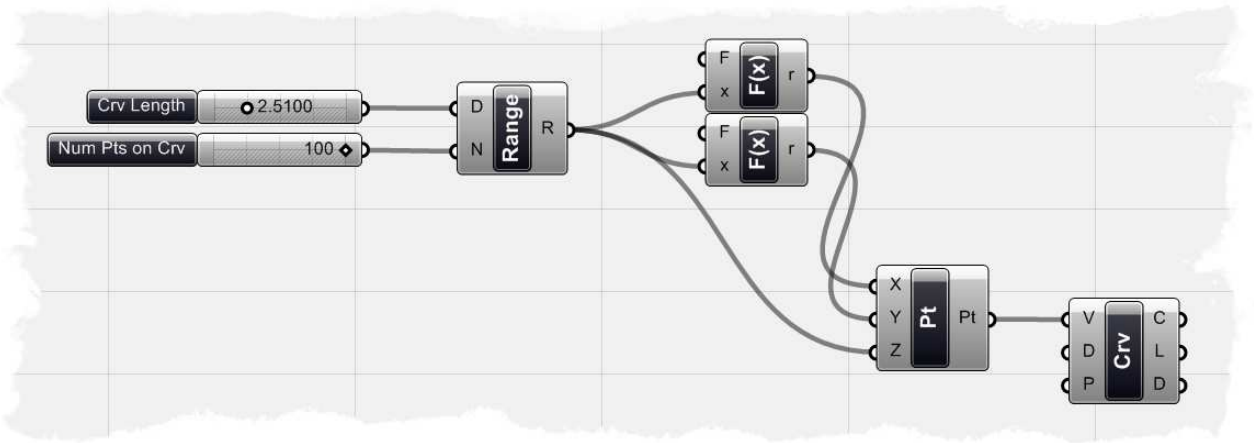
注意：想看完整版的循环布尔测试，请打开文件 If_Else test.ghx

7.4 Functions & Numeric Data(函数与数字型数据)

函数运算器是非常灵活的；也就是说它可以广泛的用于不同的操作。我们已经讨论过怎样使用一个函数运算器来评价一个条件语句并产生一个布尔输出。不管怎样，我们也能使用函数运算器来解决复杂的数学算法并产生数据输出。

在下面的例子中，我们将创造一个类似 Davia Rutten 在他的 Rhinoscript101 手册中提到的一个数学曲线的例子。若想更多的了解 Rhinoscript 或者像得到一份电子版的拷贝的话，请访问 http://en.wiki.mcneel.com/default.aspx/McNeel/Rhinoscript101.html

注意：想要看完成版数字曲线建造的例子，请打开在 Source File 文件中 Function_spiral.ghx 下面是整个函数定义的截屏。

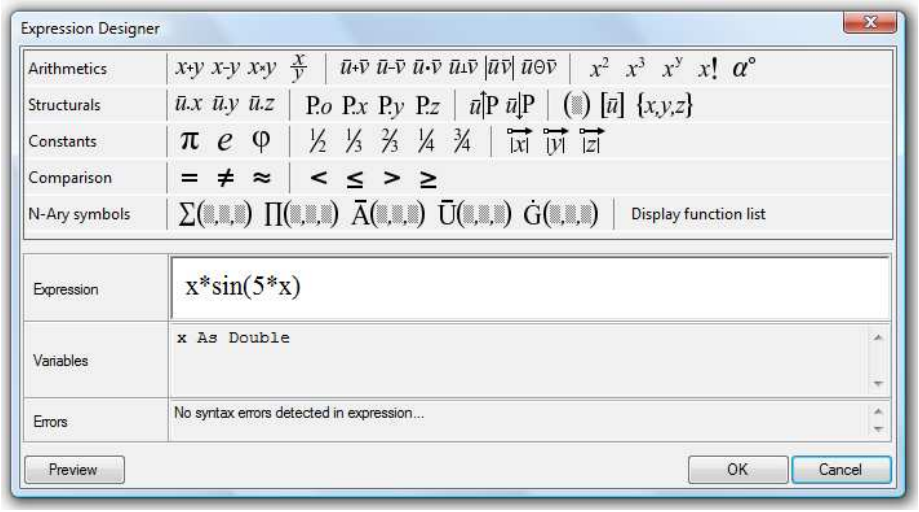


分步解说：

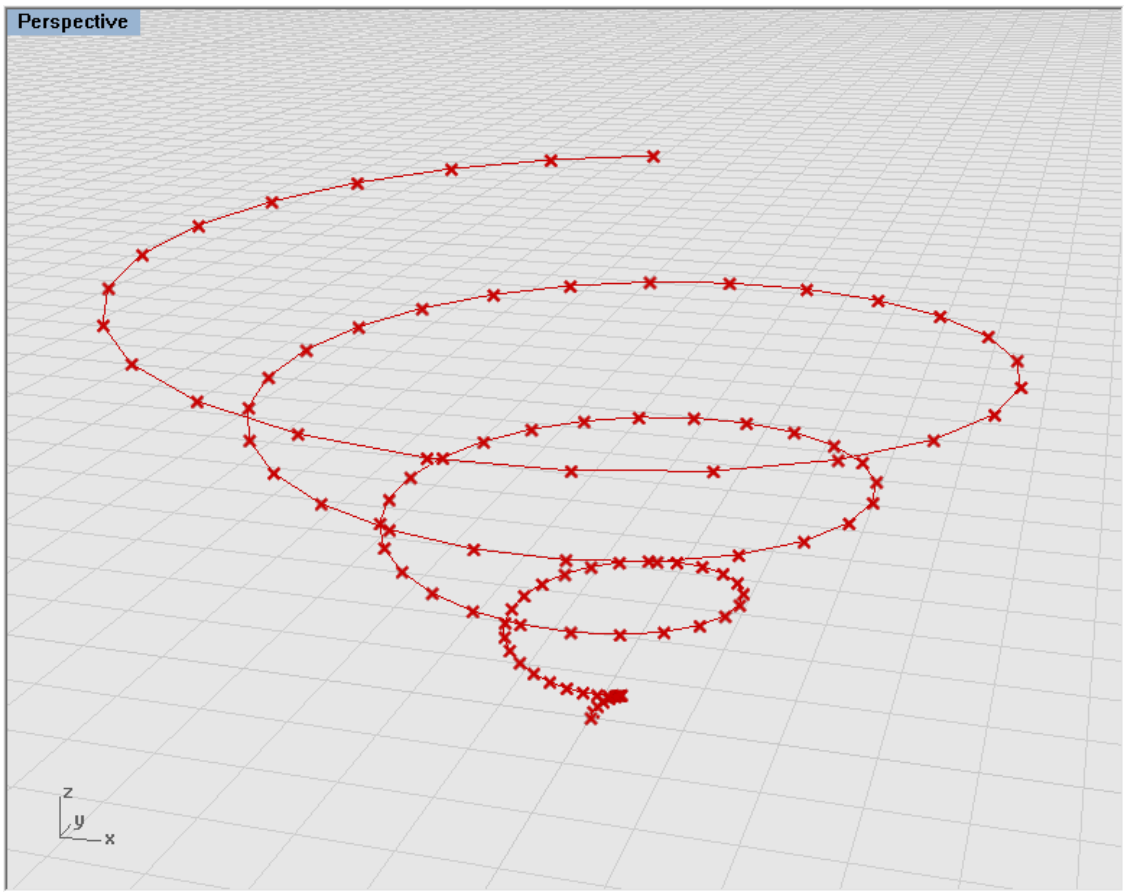
- 在 Logic/Sets/Range 中拖拽一个 Range 运算器到工作区中去
- 在 Params/Special/Slider 中拖拽两个数字滑块到工作区中去
- 右击第一个数字滑竿并如下设定：
  - Name: Crv Length
  - Slider Type: Floating Point (this is set by defualt)
  - Lower Limit: 0.1
  - Upper Limit: 10.0
  - Value: 2.5
- 右击第一个数字滑竿并如下设定：
  - Name: Num Pts on Crv
  - Slider Type: Integers
  - Lower Limit: 1.0
  - Upper Limit: 100.0
  - Value: 100.0
- 连接 Crv Length 滑竿和 Range 运算器的 D 输入项
- 连接 Num Pts on Crv 滑竿和 Range 运算器的 N 输入项

  我们已经创造了一列有 101 个数字组成的数组，它们从 0 到 5 均匀的分布着，我们将把它们输入到函数组建中去
- Logic/Script/F1 中拖拽一个一次函数的 Function 运算器到工作区
- 右击 Function 运算器的 F 输入项并打开 Expression Editor

- 在 Expression Editor 对话框中输入如下等式：
  - **x*sin(5*x)**
    如果你输入了正确的运算公式，你将会在下面Errors 提示中看到“No syntax errors detected in expression”（表达式中无语法错误）
  - 单击 OK 关闭对话框



- 选择 Function 运算器并按 Ctrl+C (复制)和 Ctrl+V (粘贴) 来创建一个运算器的副本
- 右击 Function 运算器副本中的 F 输入项并打开 Expression Editor
- 在 Expression Editor 对话框中输入如下等式：
  - **x*cos(5*x)**
    注意：*两个等式唯一的不同就是用 cos 函数来代替原来的 sin 函数*
  - 单击 OK 关闭对话框
- 连接 Range 运算器的 R 输出项和两个 Function 运算器的 X 输入项
  我们已经将 Range 运算器生成的 101 个数据导入到 Function 运算器中，并按照一定的数学运算法则生成一组新的数字数据。你可以点击两个 Function 运算器的 r 输出项来查看不同等式生成的结果
- Vector/Point/Point XYZ 中拖放一个 Point XYZ 运算器到工作区中
- 连接第一个 Function 运算器的 r 输出项到 Point 运算器的 X 输入项
- 连接第一个 Function 运算器的 r 输出项到 Point 运算器的 Y 输入项
- 连接 Range 运算器的 R 输出项到到 Point 运算器的 Z 输入项
  现在再看 Rhino 的视窗，你将看到螺旋向上的一列点。你可以改变两个滑竿来控制这列点的数量和间距
- Curve/Spline/Curve 中 拖放一个 Curve 运算器到工作区中
- 连接 Point 运算器的 Pt 输出项和 Curve 运算器的 V 输入项
  我们已经创造了一条经过所有点的曲线，我们可以右击 Curve 运算器的 D 输入项来改变 Curve 的阶数（degree）；一个 1 阶的曲线可以在相邻两个点之间生成直线来穿过所有的点。一个 3 阶的曲线将生成一个平滑的 Bezier 曲线，这里的点被当做曲线的控制点，所以曲线并不一定会穿过所有点



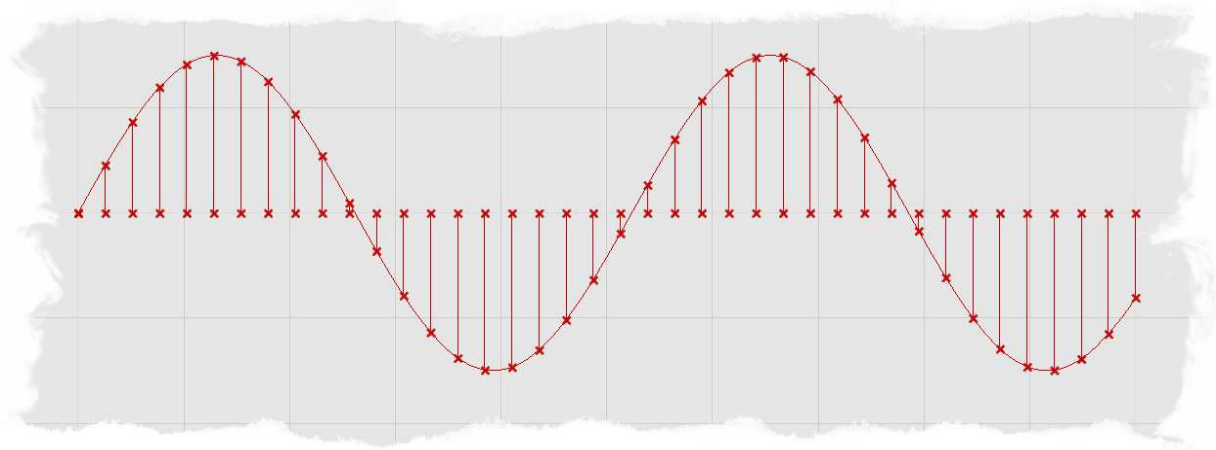
注意：想看这个实例的视频教程，请浏览 Zach Downey 的博客:
http://www.designalyze.com/2008/07/07/generating-a-spiral-in-rhinos-Grasshopper-plugin/

7.5 Trigonometric Curves(三角函数曲线)

上例中我们用 Function 运算器通过复杂的方程式计算来创建螺旋线以及其他的数学曲线。除此之外，Grasshopper 的标量（scalar）组里还有一些其他的三角函数运算器.三角函数,像正弦(sine)、余弦（cosine）、和正切（tangent）对于数学家、科学家以及工程师等都是非常重要的工具，因为他们通过直角三角形两条邻边间的夹角（Theta）来定义这两条边之间的比率。这些函数对于我们后面会谈到的向量分析是非常重要的。除此之外，我们还可以用这些函数来定义周期性的现象，比如正弦波曲线函数跟海浪、声波以及光波的形状很像。在 1822 年， Joseph Fourier，一个法国数学家，发现正弦波曲线函数几乎可以组成或者描述所有的周期性波形。这个过程被称为傅里叶分析（Fourier analysis）.

在下面的实例中，我们将创建一个正弦波形状，这个正弦波上点的数量、波形的长度以及频率都一系列滑动器（slider）来控制。

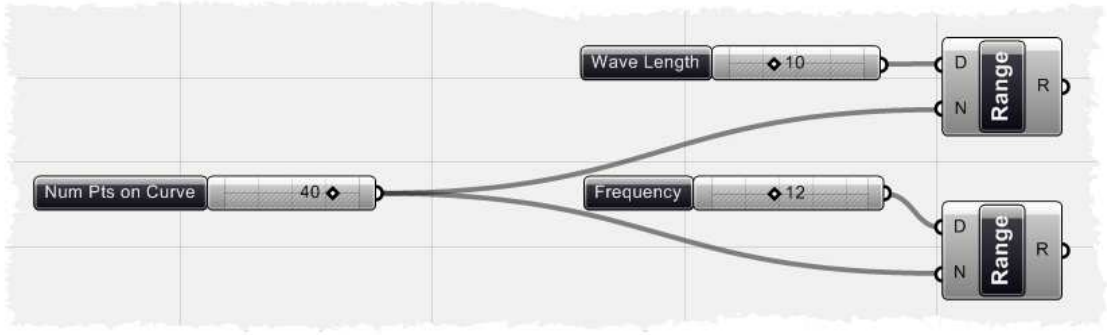
注意：最终的 Grasshopper 定义文件是在这本教材配套的 Source Files 文件夹里的 Trigonometric_curves.Ghx 文件。



从头建立这个定义（definition）:

- Params/Special/Slider - 拖动三个滑竿到工作区。选择第一个滑竿，然后进行如下设置：
  - Name: Num Pts on Curve
  - Slider Type: Integers
  - Lower Limit: 1
  - Upper Limit: 50
  - Value: 40
- 选择第二个滑竿，然后进行如下设置：
  - Name: Wave Length
  - Slider Type: Integers
  - Lower Limit: 0
  - Upper Limit: 30
  - Value: 10
- 选择第三个滑竿，然后进行如下设置：
  - Name: Frequency
  - Slider Type: Integers
  - Lower Limit: 0
  - Upper Limit: 30

    - Value: 12
- Logic/Sets/Range - 拖动两个 Range 运算器到工作区
- 连接 Wave Length 滑竿到第一个 Range 运算器的 D 输入项
- 连接 Frequency 滑竿到第二个 Range 运算器的 D 输入项
- 连接 Num Pts on Curve 滑竿到两个 Range 运算器的 N 输入项

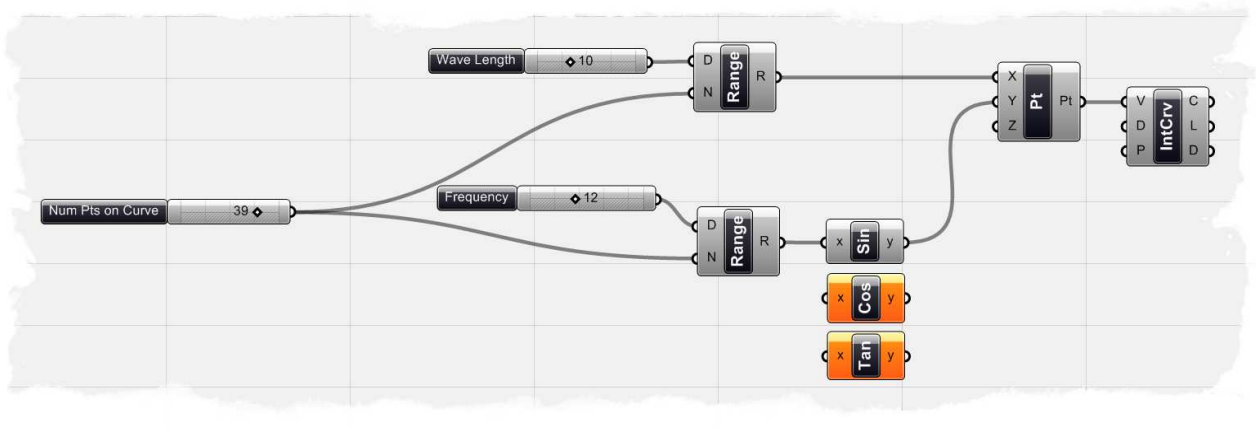


你的定义（definition）应该看起来和上面的差不多，这个定义已经产生了两组数据，第一组数据的分布范围是 0-10，第二组数据的分布范围是 0-12。

- Scalar/Trigonometry/Sine - 拖动一个 Sine 运算器到工作区
- 连接第二个 Range-R 输出项到 Sine-x 输入项
- Vector/Point/Point XYZ - 拖动一个 Point XYZ 运算器到工作区
- 连接第一个 Range-R 输出项到 Point XYZ 运算器的 x 输入项
- 连接 Sine-y 输出项到 Point XYZ-y 输入项

  这时，Rhino 窗口里将产生形成一个正弦波形状的一系列点。因为第一个 Range 输出的数据直接输入 Point 运算器的 x 输入项，中间没有经过三角函数运算，那么这些点的所有值的间隔是永远平均相等的。然而，Point 运算器的 y 输入项的数据经过了 Sine 运算器的计算,那么产生的点的 y 值将会变化,变化的结果是这些点形成了一个波形。你可以改变数值滑动器的值来改变波形的形状。
- Curve/Spline/Curve - 拖动一个 Interpolated Curve 运算器到工作区
- 连接 Point 的 Pt 输出项到 Curve 的 V 输入项

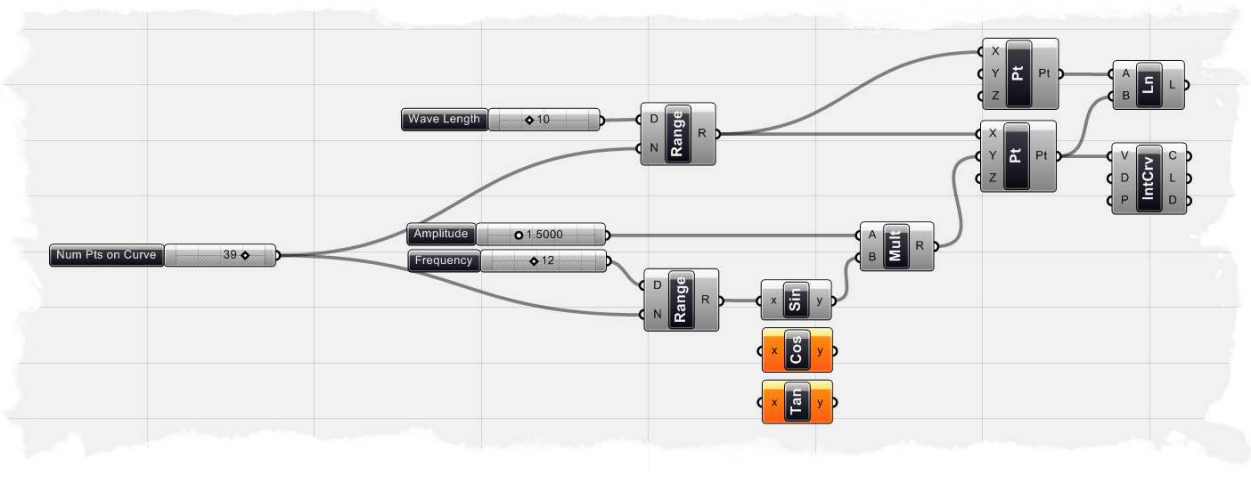
你的定义应该和上面的截图一致。我们已经通过 Num Pts on Curve， Wave Length 和 Frequency 三个滑竿来建立定义，但是我们仍需再设定一个滑竿来控制曲线的振幅。



- Params/Special/Slider 中 拖放一个数字滑竿到工作区中
- 右击这个新的滑竿并如下设定：
  - Name: Amplitude
  - Slider Type: Floating Point
  - Lower Limit: 0.1
  - Upper Limit: 5.0
  - Value: 2.0
- Scalar/Operators/Multiplication 中 拖放一个 Multiplication 运算器到工作区中
- 连接 Amplitude 滑竿到 Multiplication 运算器的 A 输入项
- 连接 Sin 运算器的 y 输出项到 Multiplication 运算器的 B 输入项
- 连接 Multiplication 运算器的 r 输出项到 Point XYZ 运算器的 Y 输入项

  这个连接应该代替原来的连接，Amplitude 滑竿仅仅是用来控制与 Sine 运算器数值相乘的因数。由于这个滑竿是用来控制 Sine 曲线的 Y 值的，所以当你增加 Amplitude 的数值的时候曲线的振幅也将增大
- Vector/Point/Point XYZ 中 拖放另一个 Point XYZ 运算器到工作区中
- 连接第一个 Range 运算器的 R 输出项到新的 Point XYZ 运算器的 X 输入项
- Curve/Primitive/Line 中 拖放一个 Line 运算器到工作区中
- 连接第一个 Point 运算器 Pt 输出项到 Line 运算器的 B 输入项
- 连接第二个 Point 运算器 Pt 输出项到 Line 运算器的 A 输入项

  在定义的最后一部分，我们创建了第二列在 x 轴上等距的点，这相当于 Sine 曲线的 x 坐标。Line 运算器创建线段用由第二列点定义的 x 轴分割由第一列点创建的Sine 曲线。这些新的线段可以让你直观地看到波形曲线的各部分竖直高度。

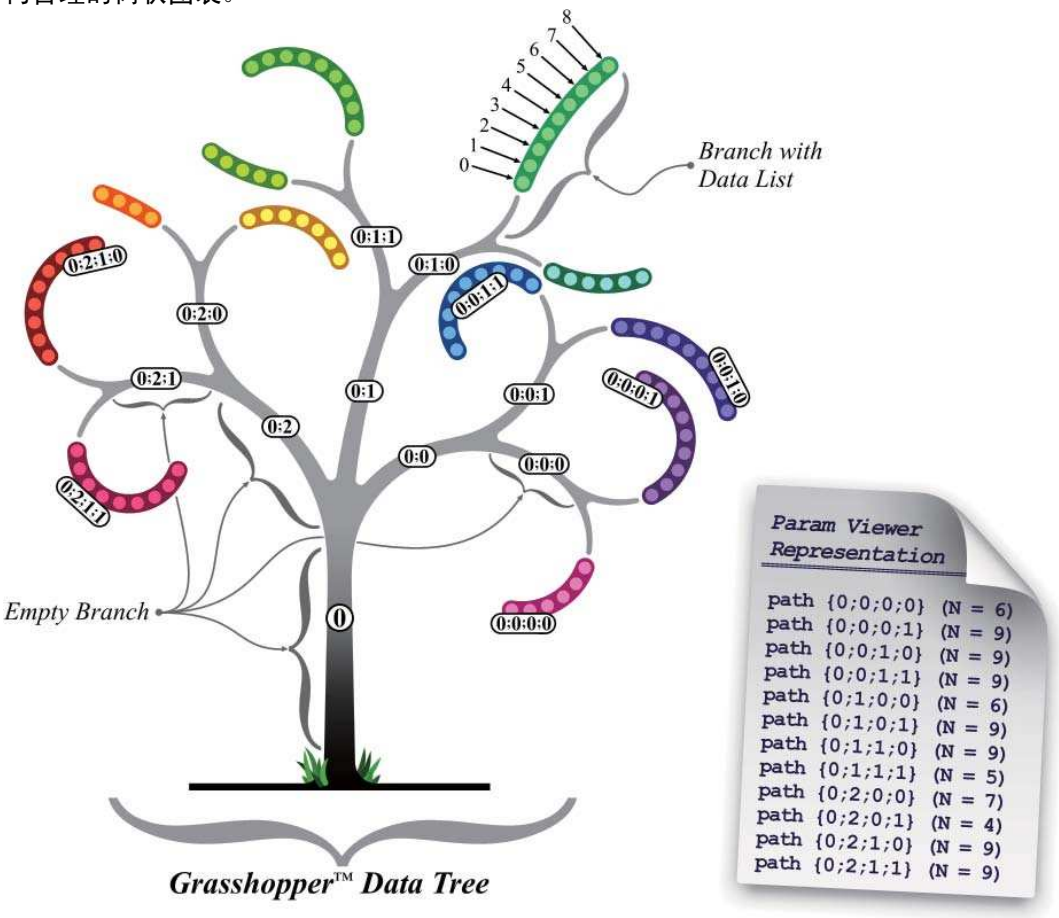


我们已经向你展示了如何建立一个 sine 波动曲线，你也可以通过替换 Sine 运算器做出其他的三角函数曲线，例如 Cosine 或者 Tangent 波动曲线。Cosine 和 Tangent 运算器皆可以在 Scalar/Trigonometry 面板下找到。

注意: 想看这个实例的视频教程，请浏览 David Fano 的博客:
http://designreform.net/2008/06/01/rhino-3d-sine-curve-explicit-history/

8 The Garden of Forking Paths

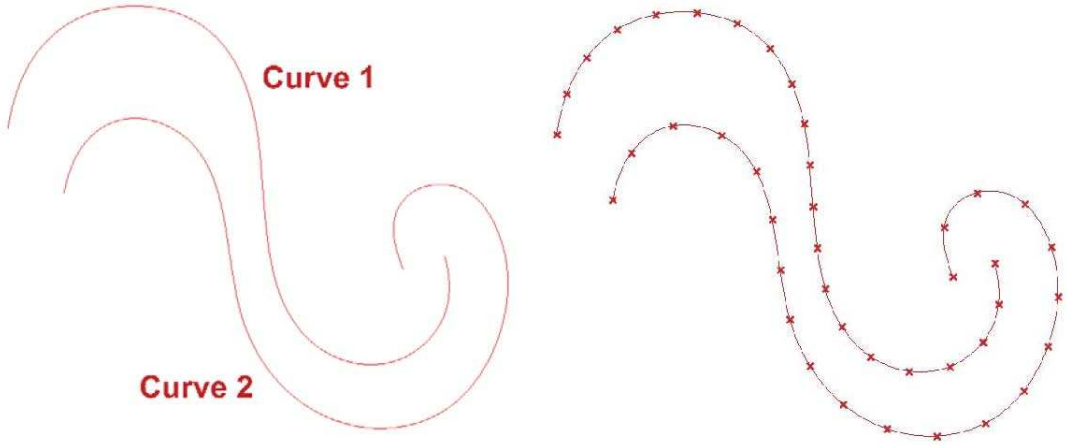
在 Grasshopper 升级到 0.6.00xx 以前，在一个参量（parameter）中的数据是一个单列，因此不需要给这个数据列编号。但是，现在 Grasshopper 中数据存储的方式经过了一个比较大的修改。现在，在一个单独的参量中允许存在多条数据列。由于多条数据列的存在，那么就需要一种识别每一条数据列的方法。下面这张图片，由 David Rutten 创建，代表一种适度复杂但是结构合理的树状图表。



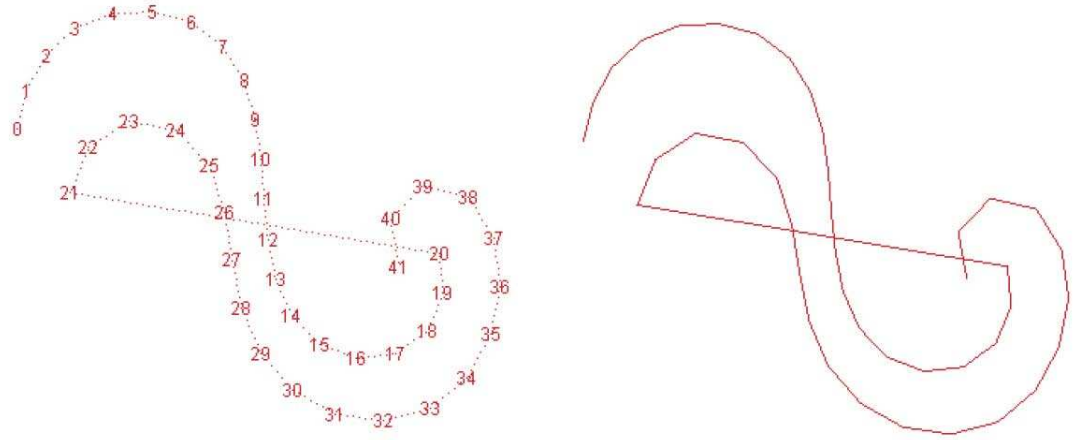
在上面的这张图片里，有一个单独的主要树枝（你可以叫它树干，但是因为也许会有多个主干，这个名称可能会有点不恰当）编号是 path{0}。这个路径（path）不包括任何数据， 但是包括了三个子分支每一个子分支继承了它的父支的编号{0}并且拥有他们自己的子编号（0，1 和 2 等等）。也许叫这个“编号”是不对的，因为“编号”往往会暗示这仅仅是一单个数字。很可能我们叫这个为“路径（path）”会更好一点，因为这个有点类似于硬盘上的文件夹结构。每一个子分支又有两个子子分支，这些子子分支也不包括任何数据。当我们这样一层一层的到达第四层嵌套的时候，我们终于遇到一些数据（这些数据列由带颜色的线代表，数据由明亮的圆圈代表）。 每一个子子子分支（或者说是第四级的分支）是一个重点的分支，这意味着这些分支不再继续细分。

因此，每一单个数据项在整个树状的数据结构中只属于并且仅仅属于一个分支，每一单个数据项拥有一个唯一的编号指定了它在这个分支中的位置。每一个分支有一个路径编号指定这个分支在树形结构中的位置。

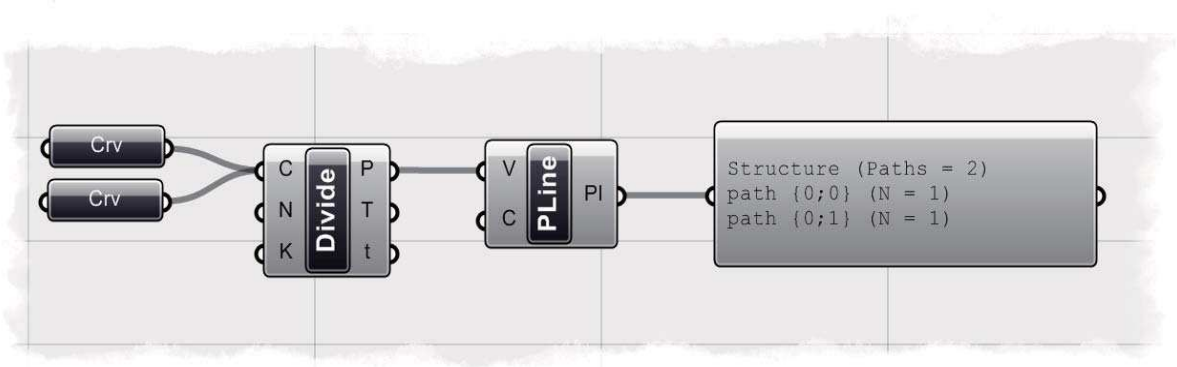
为了进一步解释这种数据的树形结构，让我们来看一个非常简单的例子。在 **Rhino** 中创建两条曲线，在 Grasshopper 中我们将他们分别赋予两个 curve 参量。然后使用一个 Divide Curve (Curve/Division/Divide Curve) 运算器将这两条曲线等分为 20 段，最终每条曲线上得到 21 个等分点。然后我们将这些点输入一个 Polyline 运算器(Curve/Spline/Polyline),这样我们会得到一条新的多线。



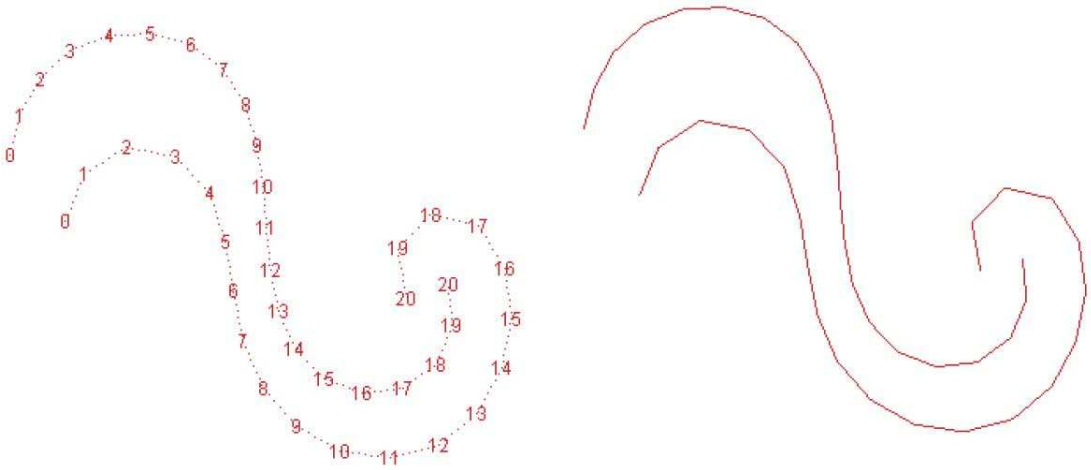
如果我们是使用 0.5 或者更早版本的 Grasshopper，那么这个 Polyline 运算器仅仅会输出一条使用了这 42 个点的复合线，这是因为在这些版本里，这些点数据仅仅是一个单列的形式存储的，结果 Polyline 运算器的程序就是通过这些点画一条复合线。如果我们使用 0.5 版本的 Grasshopper，这些点的编号和输出结果如图所示：



但是，现在我们知道 Grasshopper 拥有了包含路径的能力，我们现在可以使用它来控制 Polyline 运算器的表现。如果我们完全按照以上的步骤，但是仅仅使用 0.600xx 或者更高版本的 Grasshopper，我们的 Polyline 运算器将创建两条多线，因为它将这些数据识别为两条路径，每一条路径里的曲线被分为了 20 段。我们可以使用 Parameter Viewer (Params/Special/Param Viewer)来查看它的数据结构。下面是这个例子的一个截图，表明数据结构拥有两个路径 path{0;0}和 path{0;1}。



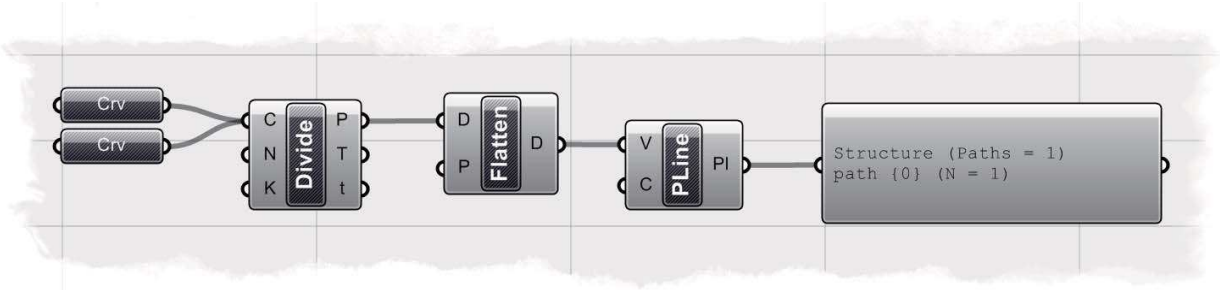
这两条多线的点的编号和产生的结果应该像下图：



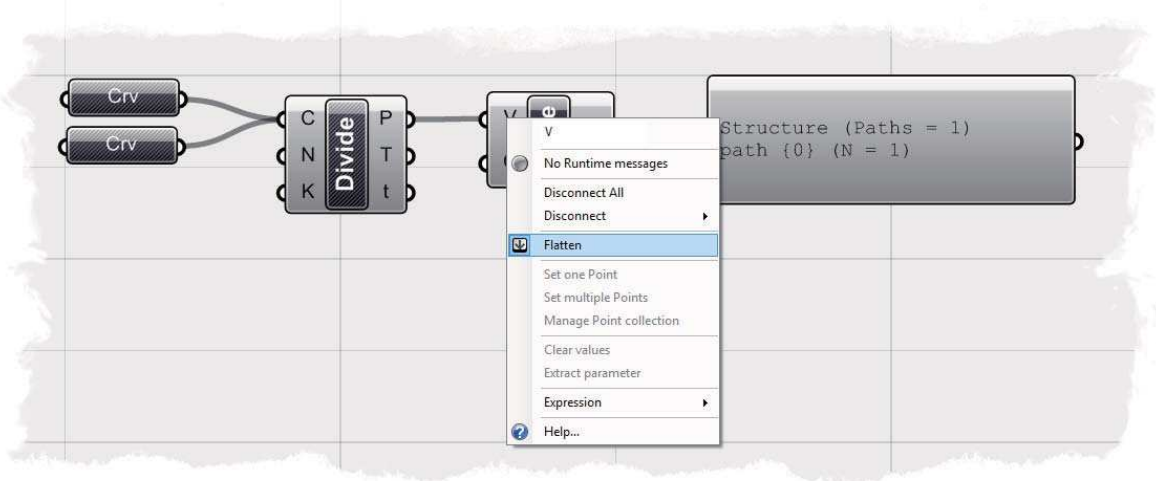
现在我们知道我们的 Grasshopper 定义里有两个路径，但是如果我们想要一个路径，想要一条多线的结果，我们应该怎么做？Grasshopper 现在又很多个新的与数据结构相关的运算器，你可以在 Logic/tree 子分类里找到他们。这些运算器可以帮助你控制你的数据结构。在这里，我们使用 Flatten Tree 运算器 (Logic/Tree/Flatten Tree)来将所有的路径合并为一单个数据列，就像数据在 0.5 版本里显示的一样。

下面这张图片里，你可以看到，当使用里 Flatten 运算器之后，我们的数据结构仅仅有一个路径，因此仅仅产生了一条复合线。这和使用以前的版本里产生的结果是一样的。

你也可以简化这个 Grasshopper 定义，方法是简单的选中 Polyline 运算器 V 参量右键菜单里的 Flatten 选项。 这样做了之后，你会发现数据结构从两条路径变为一条路径（通过 Parameter



Viewer 查看）。



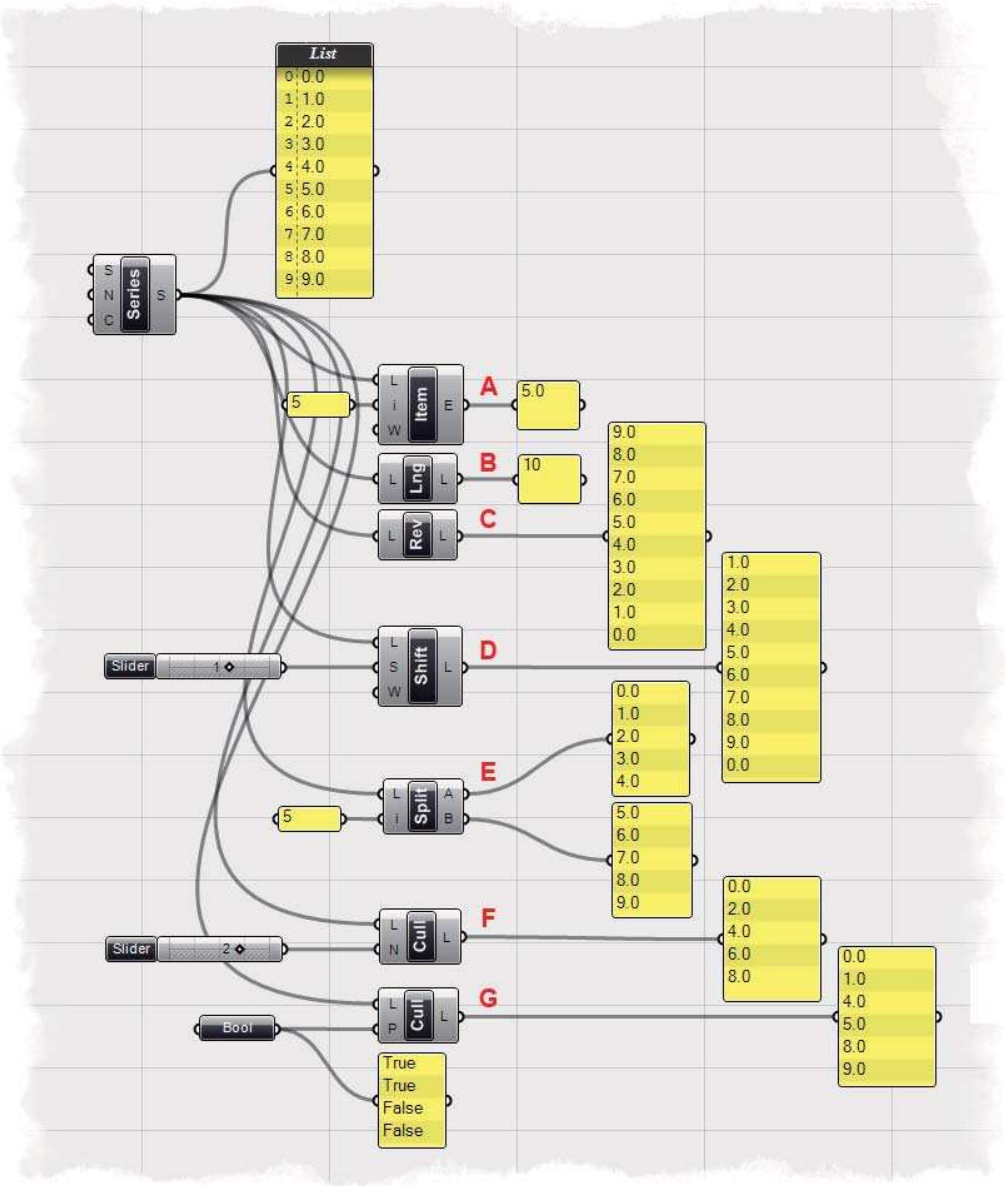
注意: 如果你想看这个例子完成后的定义，你需要打开 **Rhino** 文件 Curve Paths_base file.3dm. 和文件 Curve Paths.ghx，这两个文件在本文档附带的源文件里。

8.1 数据列 & 数据的管理

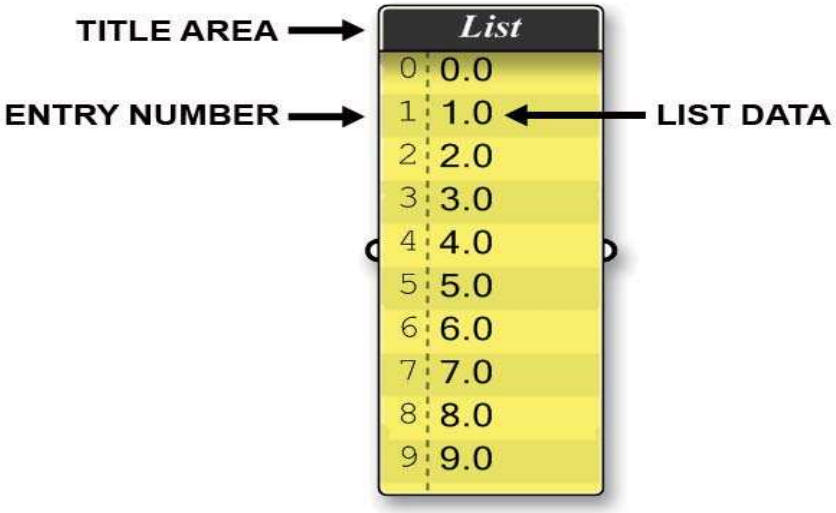
把 Grasshopper 按照一种数据流动的关系来考虑是有帮助的，因为这种图形化的界面的设计让信息(数据)从运算器里流入和流出。但是，是数据（比如点、曲线、曲面、字符串、布尔值和数字等）决定了这些流入与流出运算器的信息。这样，理解如何操控这些数据列是理解 Grasshopper 的关键所在。

下面是一个如何控制一串数字数据的例子，在这个例子中我们谁使用到大量的 List 运算器，你可以再 Logic/Lists 子分类里找到他们。

注意:如果你想看最后的定义，打开本文档附带的源文件夹里的 List Management.ghx 文件。



最为开始，我们创建一个 Series 运算器，起始值为 0，间隔值为 1，总数为 10.那么，连接到 Series 运算器的 Post-it 面板就会显示下面的数字： 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 和 9.0.
注意: Post-it 面板的默认设置是在数据前面显示数字的编号，你可以通过在 Post-it 面板上右击然后



改变输入数据的预览来关闭或者显示这些编号。在这个例子中，我们将保持显示这些编号，这些编号可以让我们精确地看到每一个数据分配的编号。

A) 这些数据输入到 **List Item** 运算器(Logic/List/List Item)。这个运算器用来一列数据中特定的一个（当然也可以是几个）。我们是通过特定的数据的在所在数据列里的编号找到他们的。在任何数据列里，第一个数据的编号是 0，第二个是 1，第三个是 2，往后以此类推。如果你找编号为-5 的数据，你会得到一个错误，因为没有这样编号所对应的存储数据的位置。我们已经将 Series-S 输出端和 List Item-L 输入端相连。然后，我们输入一个整数到 List Item-i 输入端，这个整数定义了我们想检索的数据的编号。因为我们设置这个值为 5，List Item 运算器的输出端将会显示一个对应的数据。在这个例子里，这个数据是 5.0。

B) **List Length** 运算器(Logic/List/List Length)本质上是计算输入数据的数量并且输出最后的数字，或者说是数据列的长度。在这个例子中，我们将 Series-S 输出端与 List Length-L 输入端相连，结果显示这一列数据里有十个数据。

C) 通过使用 **Reverse List** 运算器(Logic/List/Reverse List)我们可以反转一列数据排列的顺序。

D) **Shift** 运算器(Logic/List/Shift List)将会根据你输入的数据偏移值向上或者先下移动。我们已经连接 Series-S 输出端到 Shift-L 输入端，同时将一个滑竿连到 Shift-S 输入端。我们已经将滑竿的类型设置为整数，这样 shift 运算器使用的偏移值将只会得到整数值。如果我们将滑竿值设置为-1，数据列里所有的值将会向下移动一个值，同理，如果我们将这个值设置为 1，所有的值将会向上移动 1。我们还可以设置是否包裹（Wrap），这是一个布尔值，我们可以在 Shift-W 上右击，然后选择“设置布尔值”来选择“True(真)”或者“False(假)”。在这个例子中，这

个 shift 的偏移值为+1，下面我们可以决定怎么对待第一个第一个值。如果我们设置是否包裹的值为真，那么第一个值将会被移动到这一列数据的最后。如果我们设置是否包裹的值为假，这个值将会向上偏移出这一列数据，就是将这个值从这一数据列里移除了。

E) Split List 运算器。他将一列数据分为两个短一些的数据。通过输入你希望将这一列数据分开的数据的编号来实现这个目的。在这个例子中，我们让 Split List 运算器将数据列在第五个数据后分开，因此输出结果将会这样：数据列 A - 0.0, 1.0, 2.0, 3.0, 4.0　数据列 B - 5.0, 6.0, 7.0, 8.0, 9.0。

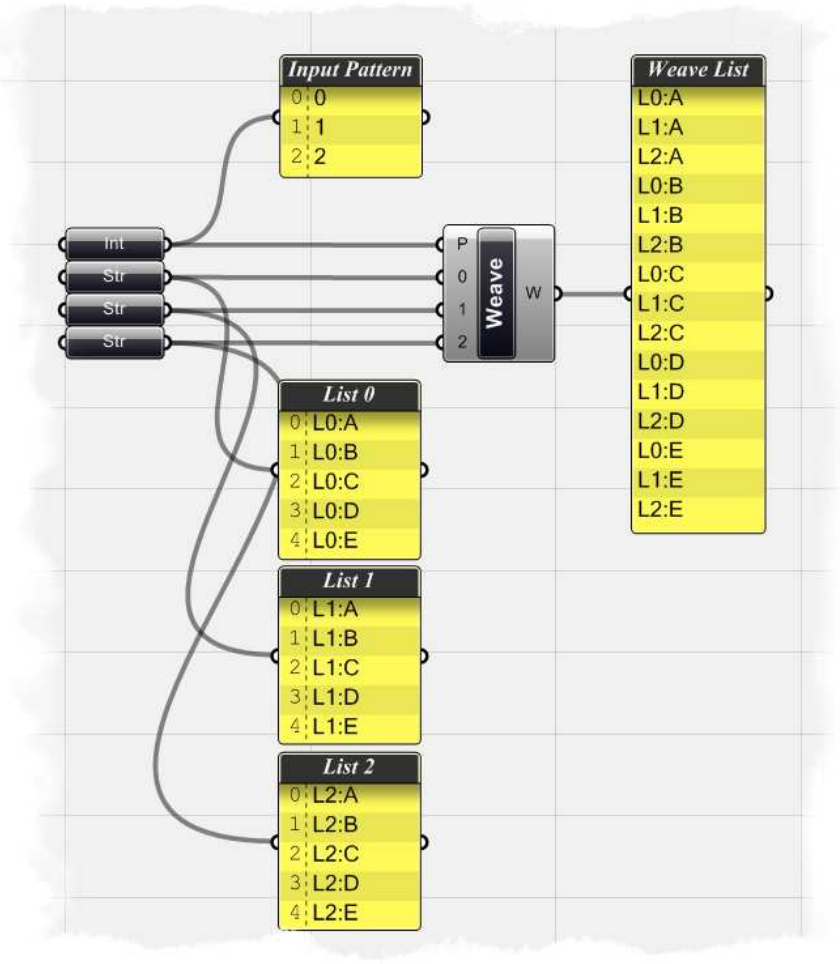
F) Cull Nth 运算器(Logic/Sets/Cull Nth)会移除输入的数据列的每第 N 个数据，很显然，N 应该是一个整数。在这个例子中，我们连接一个滑竿到 Cull Nth-N 输入端。我们已经设置滑竿的值为 2，因此 Cull Nth 运算器将会间隔一个移除数据列里的数据。Cull Nth-L 输出端输出的是从旧的数据列里移除数据后的新的数据列：0.0, 2.0, 4.0, 6.0, 和 8.0。如果我们将滑竿的值改为 3， Cull Nth 运算器将从数据列里移除每第三个数据，结果会是： 0.0, 1.0, 3.0, 4.0, 6.0, 7.0, 和 9.0。

G) Cull Pattern 运算器(Logic/Sets/Cull Pattern)跟 Cull Nth 运算器有点像，但是这个运算器是使用一串布尔值来形成一个所谓的图案（Pattern），而不是使用数值。如果布尔值是“Ture(真)”，那么对应的数据将会被保留，如果布尔值是“False(假)”，那么对应的数据将会被移除。在这个例子中，我们设置的布尔图案为：真，真，假，假。因为仅仅有四个布尔值而我们的数据列有十个值，那么这个布尔图案将会不断的被重复使用，直到数据列里最后一个数据。根据这个图案，输出的结果应该是： 0.0, 1.0, 4.0, 5.0, 8.0,和 9.0.

8.2 编织数据（ Weaving Data）

在上一部分，我们解释了我们如何使用一系列不同的运算器来理解 Grasshopper 是如何处理里面的数据列的。此外，我们也可以使用 Weave 运算器来控制数据列的顺序，Weave 运算器的位置是 Logics/Lists/Weave。

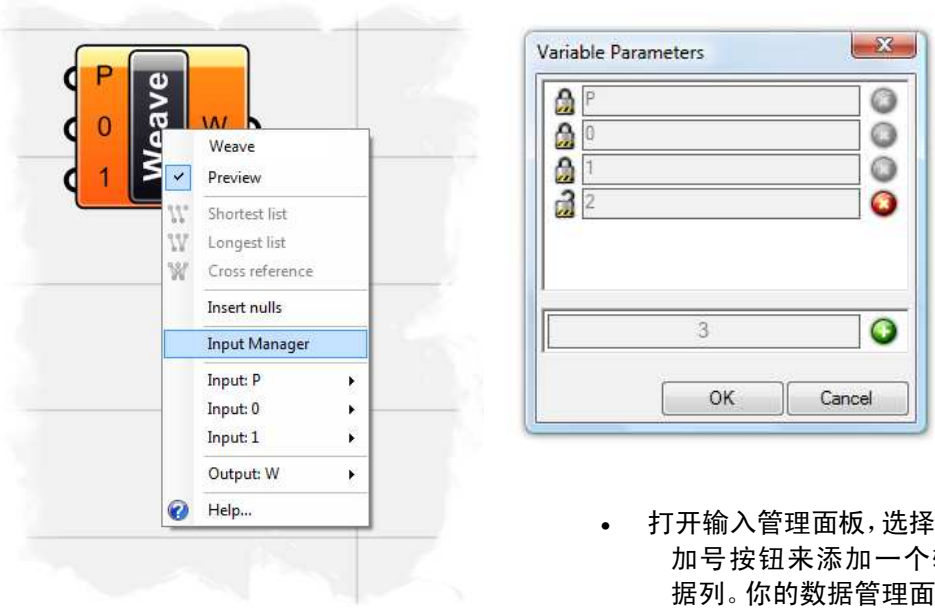
注意:如果想查看下面例子的最终定义，请打开源文件夹里的 Weave Pattern.ghrx 文件。



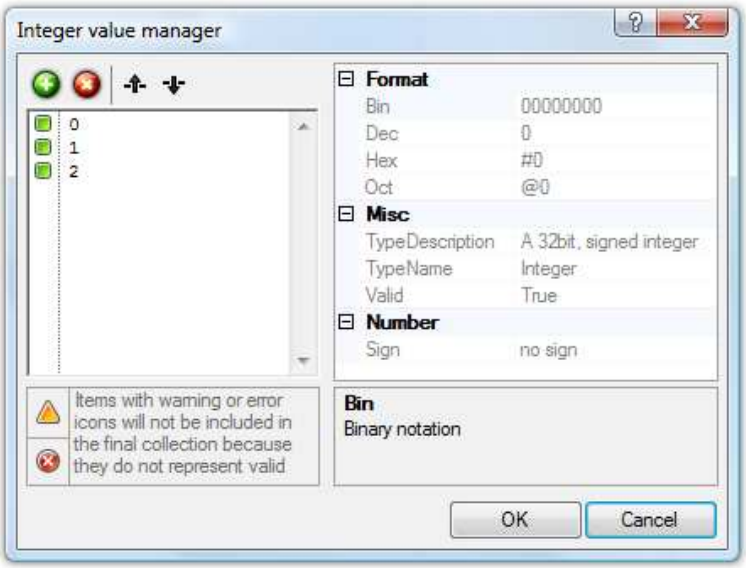
从头创建这个 Grasshopper 定义:

- Logic/List/Weave –拖动一个 Weave 运算器到工作区。

  默认的情况下你会看到三个输入端：第一个，P 输入端，代表着编织数据所采用的数值图案；接下来的两个，标签分别是 0 和 1，允许你输入两列数据参与 Weave 运算器的编织运算。但是如果你想要更多的数据列参与编织运算该怎么办？如果你在 Weave 运算器的中部右击，你会打开 “Input Manager ” 面板（输入管理），那样你就可以根据需要添加尽可能多的数入列。当输入管理面板打开的时候，点击绿色的加号按钮来添加数据列，通过点击红色的叉号按钮来删除一个数据列。



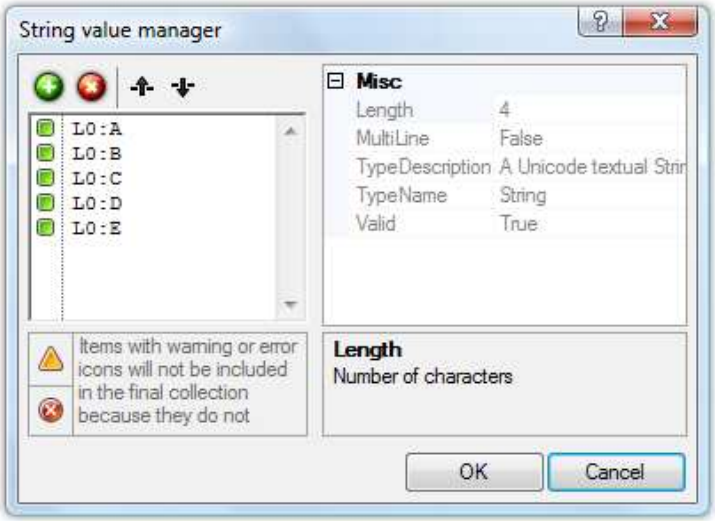
- 打开输入管理面板，选择绿色的加号按钮来添加一个输入数据列。你的数据管理面板应该看起来和上面图片上的差不多。
- Params/Primitive/Integer – 拖曳一个整数参量到工作区。
- 在整数参量上右击，然后选择“Manage Integer Collection”（"管理整数集"）
- 通过点击顶部的绿色加号按钮来添加一个整数到整数集中。添加三个数值到整数集中，并且改变每个数值的值使得你的数据列显示像下面的样子： 0, 1, 2。你的管理面板应该看起来和下面的图片差不多。



这个整数集将决定我们的编织图案。通过改变这些数值的顺序，我们可以快速的改变最后总的数据的顺序。

- Params/Primitive/String – 拖动一个 String 运算器到工作区。
- t 在 String 运算器上右击并选择“Manage String Collection”（ "管理字符串集"）。

和上面处理整数集的方式类似，我们将添加一个用于编织操作的字符串的数据列。这是我们的第一个数据列，但是我们可以通过复制和黏贴的方法来得到另外两条数据列，这样我们就有了三条数据列用于编织的操作。添加 5 个字符串到字符串集：L0:A, L0:B, L0:C, L0:D, L0:E.这个 L0 的前缀仅仅告诉我们这些字符串是在编号为 0 的数据列里的，这有助于我们跟踪定位这些字符串在新的数据列里被编织到了哪个新的位置。你的字符串管理器面板应该看起来和下面的图片差不多：



- 选择这个字符串参量，然后通过 Ctrl+C (复制) and Ctrl+V (粘贴)操作来得到另外两个字符串参量。
- 在第二个字符串参量上右击并且打开字符串集管理面板，改变字符串集如下：L1:A, L1:B, L1:C, L1:D, L1:E。
- 在第三个字符串参量上右击并且打开字符串集管理面板，改变字符串集如下：L2:A, L2:B, L2:C, L2:D, L2:E。
- 将整数参量和 Weave-P 输入端相连。
- 将第一个字符串参量和 Weave-0 输入端相连。
- 将第二个字符串参量和 Weave-1 输入端相连。
- 将第三个字符串参量和 Weave-2 输入端相连。
- Params/Special/Post-it Panel – 拖动一个 Post-it Panel 到工作区。
- 将 Weave-W 输出端和 Post-it Panel 相连，这样我们就可以看到这些数据。

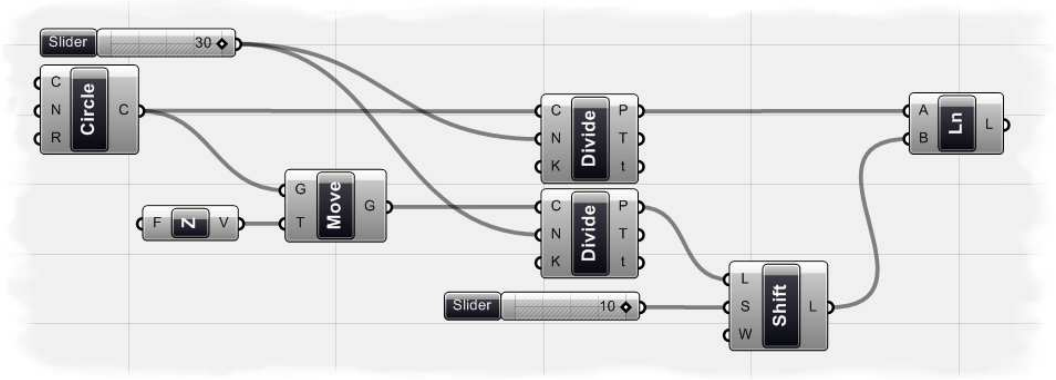
  这个 Post-it Panel 现在应该显示一列已经被编织过的数据，新的数据列里的数据的第一个应该是编号为 0 的数据列里的第一个，同样的，第二个数据应该是编号为 1 的数据列里的第一个，第三个数据应该是编号为 2 的数据列里的第一个。然后第二个循环开始，第四个数据是数据列 0 里的第二个 数据，往后的以此类推。试着改变整数集的顺序来体会编织后的数据的顺序的相应的改变。

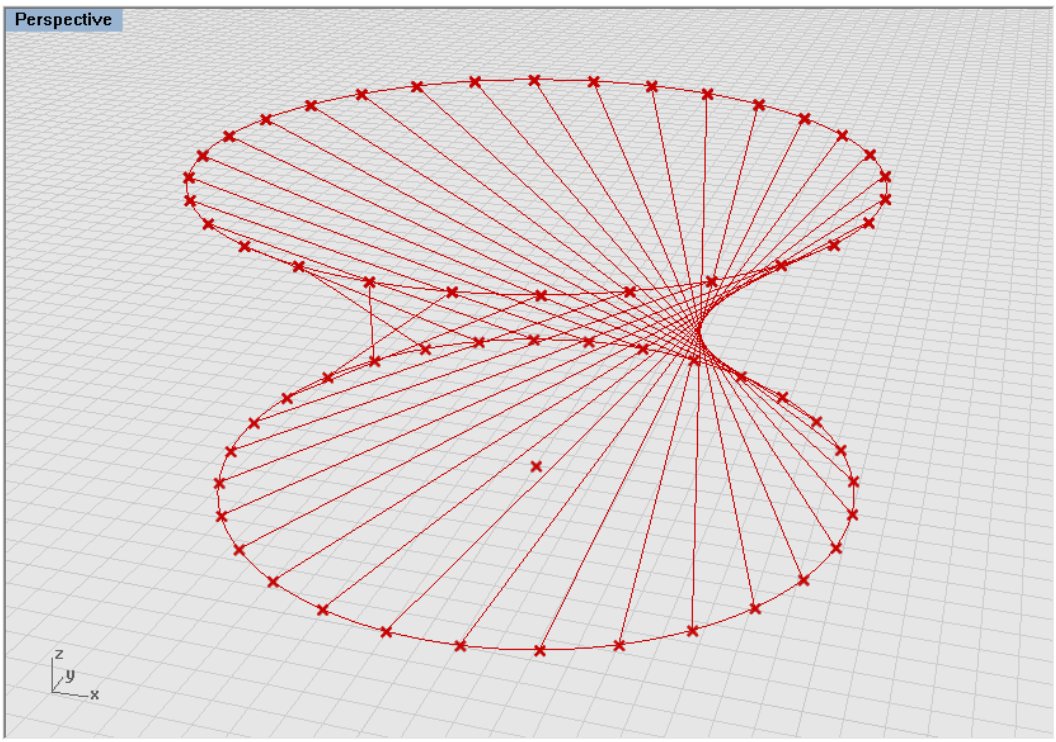
8.3 偏移数据（Shifting Data）

在 8.1 中我们已经讨论了如何使用 Shift 运算器来根据一个偏移值移动一个列中的所有的数据向上或者向下一定的距离。下面这个例子由 David Rutten（注：Grasshopper 的作者 ）创建，向我们演示了对于两个圆环上的两列点数据列，如何使用 shift 运算器。你可以在下面地址找到更多的信息：

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

注意:想看到这个文件的最终定义，打开源文件夹里的 Shift Circle.ghx 文件。下面是最终定义文件的样子：





从头创建这个定义:

- Curve/Primitive/Circle CNR – 拖曳一个 Circle CNR (Center, Normal, and Radius) 运算器到工作区
- 在 Circle-C 输入端上右击然后参照一个点
- 在 Rhino 的对话框里，键入"0,0,0" 并且点击确定。
- 在 Circle-R 输入端右击，然后将数值设置为 10.0。

- Vector/Constants/Unit Z – 拖曳一个 Unit Z 向量到工作区。
- 在 F 输入端上右击，然后将数值设置为 10.0。
- X Form/Euclidean/Move –拖动一个 Move 运算器到工作区。
- 将 Unit Z-V 输出端和 Move-T 输入端相连。
- 将 Circle-C 输出端和 Move-G 输入端相连。

  上面我们刚刚创建了一个以点 0，0，0 为圆心，以 10 单位为半径的圆。然后我们使用 Move 运算器将这个圆沿着 Z 轴方向相对 10 个单位距离的位置进行了复制。
- Curve/Division/Divide Curve – 拖曳两个 Divide Curve 运算器到工作区。
- 将 Circle-C 输出端和第一个 Divide-C 输入端相连。
- Move-G 输出端和第二个 Divide-C 输入端相连。
- Params/Special/Slider – 拖曳一个数值滑竿到工作区。
- 双击 s l i d e r 并且进行如下设置：
  - Slider Type: Integers（类型：整数）
  - Lower Limit: 1.0（下限：１．０）
  - Upper Limit: 30.0（上限：３０．０）
  - Value: 30.0（当前值：３０）
- 将数值滑竿分别和两个 Divide Curve-N 输入端相连。

  现在你应该能看到每一个圆圈上均匀的分布 30 个点。
- Logic/List/Shift List – 拖曳一个 Shift List 运算器到工作区。
- 将第二个 Divide Curve-P 输出端和 Shift List–L 输入端相连了。
- Params/Special/Slider–拖动一个数值滑竿到工作区。
- 选择这个新的滑竿并且进行如下设置：
  - Slider Type: Integers
  - Lower Limit: -10.0
  - Upper Limit: 10.0
  - Value: 10.0
- 将数值滑竿和 Shift List–S 输入端相连。
- 在 Shift List-W 输入端上右击并且将布尔值设置为 True。

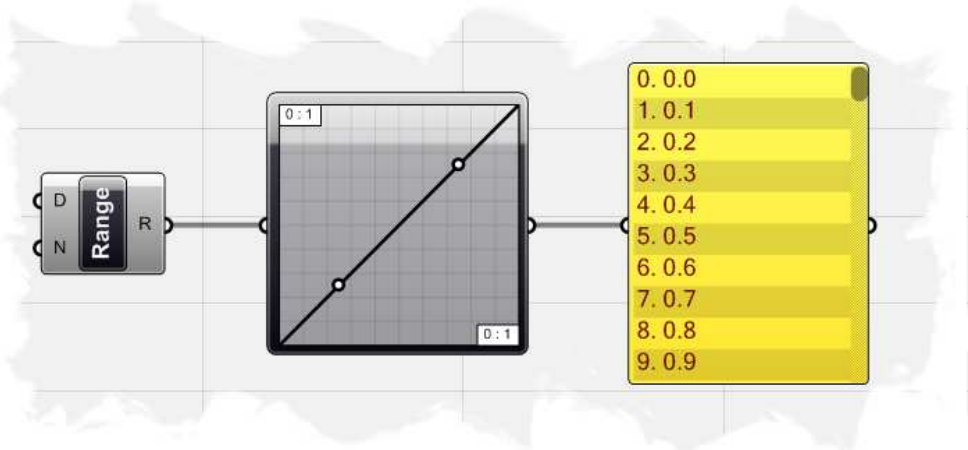
  这样，我们将这些点的编号向上移动了 10 的距离。通过设置 Warp 值为 True，这些所有的数据项形成了一个闭合的循环。
- Curve/Primitive/Line –拖动一个 Line 运算器到工作区。
- 将第一个 Divid Curve–P 输出端和 Line–A 输入端相连。
- 将 Shift-L 输出端和 Line-B 输入端相连。

  现在，我们已经创建了一系列的直线段将没有经过 Shift 运算器处理的点集合经过 Shift 运算器处理过的点集相连。我们可以通过改变滑竿的值来控制 Shift 运算器的偏移值来查看这些直线段会做出怎样的相应的改变。

8.4 将数据导出到 Excel

在有很多种情况下你需要从 Grasshopper 导出数据以便将这些信息导入到其它的软件里做进一步的分析。

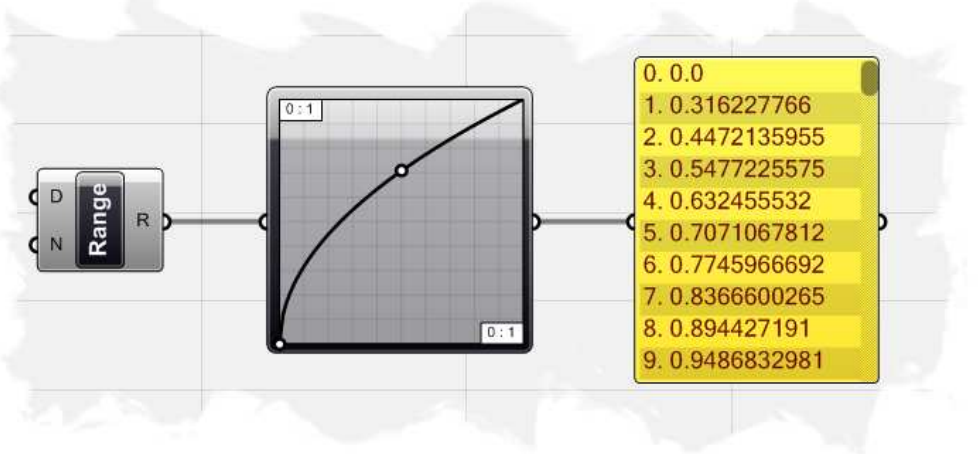
注意: 如果想看到下面例子的最终的 dfinition 请打开源文件夹里的 Stream Contents_Excel.ghx 文件。



作为开始，我们拖曳一个 **Range** 运算器 (Logic/Sets/Range) 到工作区，然后设置数值区间为 0.0 到 10.0。通过在 Range-N 输入端右击，我们设置这个“台阶”值为 100，这样我们就可以 0.0 和 10.0 这个区间里均匀地得到 101 个数值。

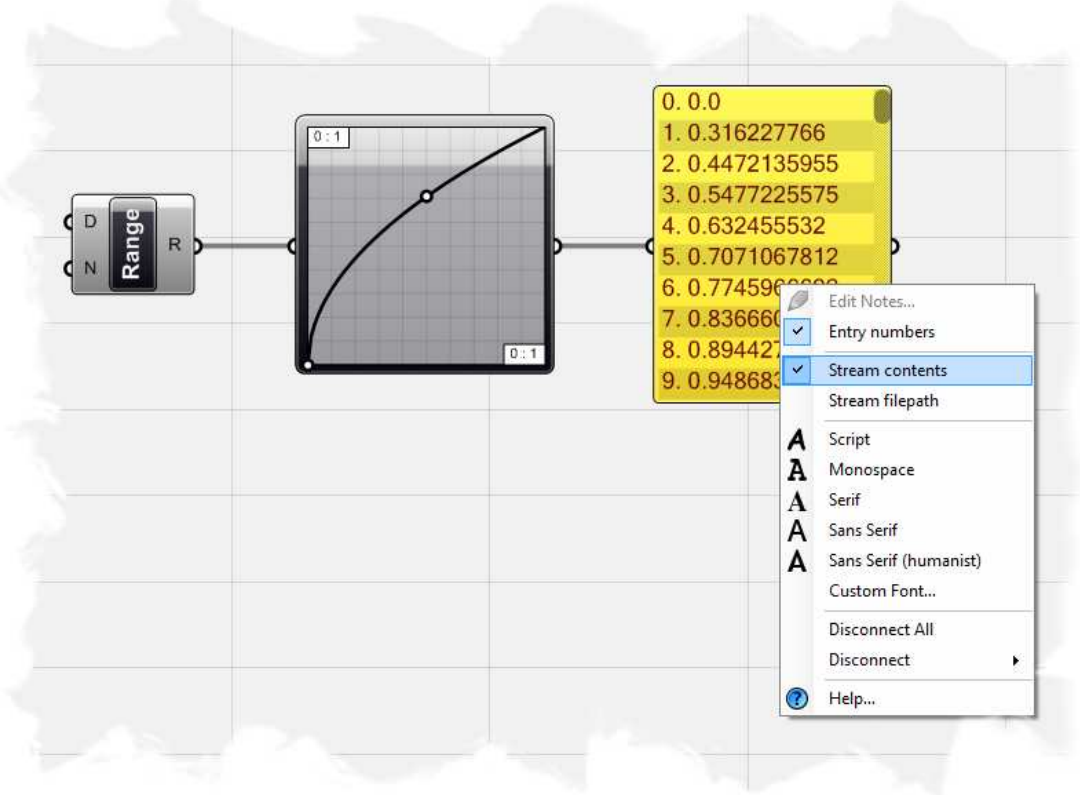
然后我们拖曳一个 **Graph Mapper** 运算器 (Params/Special/Graph Mapper) 到工作区，在 Graph Mappper 上右击，并将类型（type）设置为 **Linear**。然后将 Range-R 输出端和 Graph Mapper 输入端相连。最后一步，拖曳一个 **Post-it Panel** 运算器到工作区，将它与 Graph Mapper 相连。

因为 Graph Mapper 的类型设置为 linear ，因此输出的数据列（在 Post-it Panel 里显示）按照线性风格显示。但是，如果我们在 Graph Mapper 运算器上右击，然后设置类型为 **Square Root** ，我们将会看到一个新的数据列，这个新的数据列代表着一个对数函数。

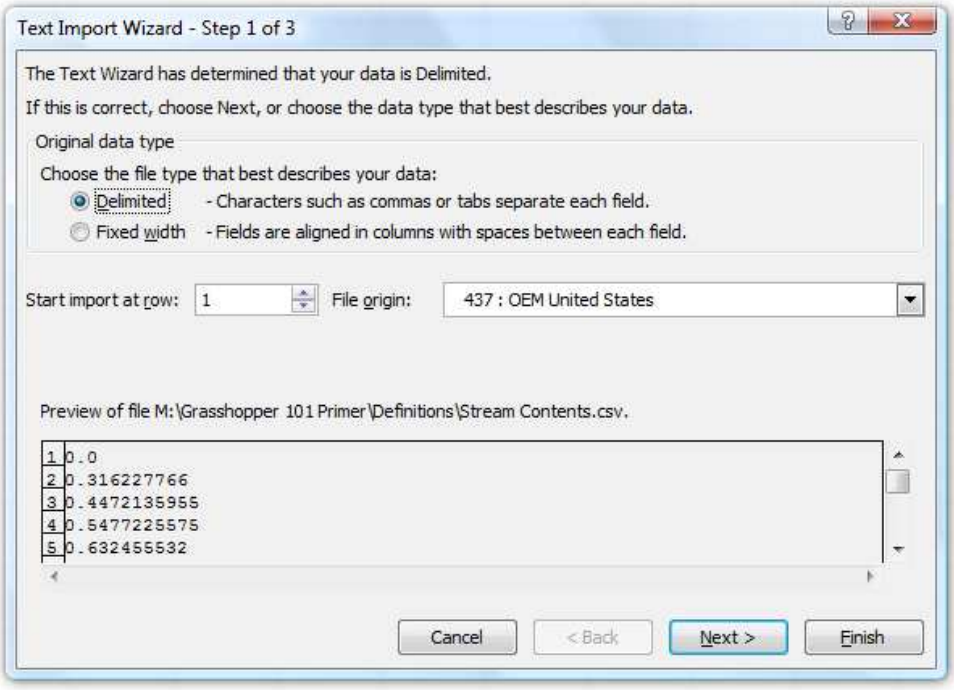


为了导出 Post-it Panel 里的数据，只需要简单的在它的面板上右击，然后选择 **Stream Contents**。然后根据提示，将文件命名并将它**保存**在你的硬盘上某个指定的位置。在这个例子里，我们将这个文件保存在下面的位置： C:/Tutorials/Exporting

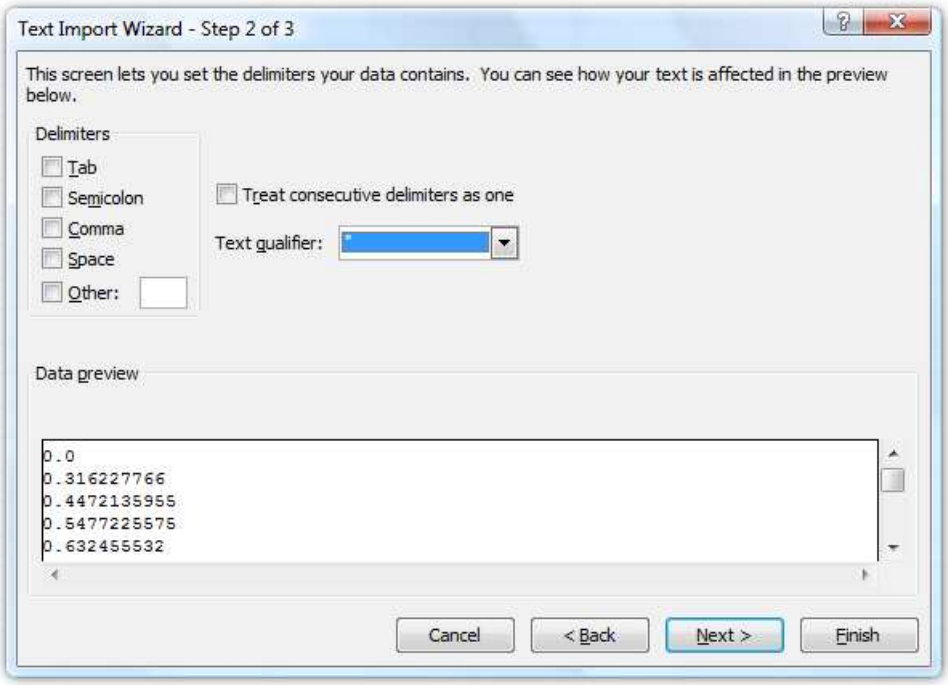
Data/Stream_Contents.csv。有很多种类型的文件格式你可以选择使用来保存你的数据，包括 Text Files (.txt), Comma Separated Values (.csv), 与 Data Files (.dat) 等等。尽管这些格式中的很多种都可以导出到 Excel 里，我个人还是比较倾向于使用 CSV 格式，因为这种文件格式被设计用来以一种图表的结构来存储数据。CSV 里的每一列的数据对应着 CSV 文件表格样式里的某一个排（row），在一个排里，每一个字段被逗点分开，每一个字段属于一个表格栏，在我们的例子中，每一列仅仅包含一个数据，因此我们将不会用到多个表格栏，但是，这种格式为我们提供了一种直接得到复杂的多页面表格的可能性。



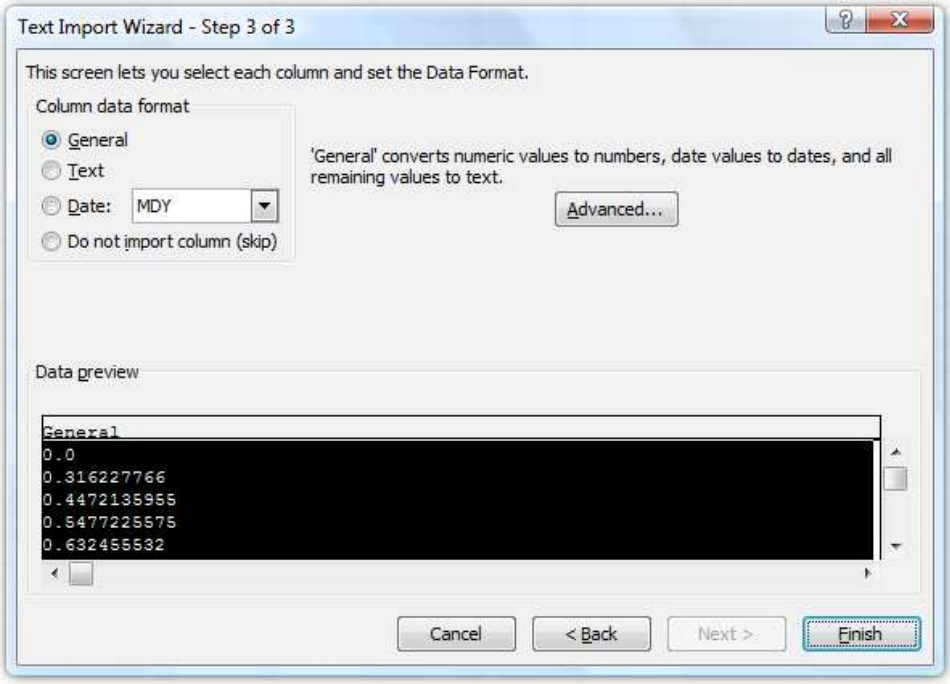
下面我们将数据导入到 Microsoft Excel 2007。首先运行 Excel，并且选择“数据”标签，在下拉菜单里选择“**导入外部数据…**”**(Get External Data from Text)**然后选择“**导入数据**”**(Text Import Wizard)**，然后在文件类型里找到 CSV 格式，然后找到上面你保存的那个 Stream_Contents.csv 文件。下面跟随着导出向导的提示，将数据导入到 Excel 里。在导入向导的第一步里确保选中了“**分隔符号**”，然后选择“下一步”到第二步。



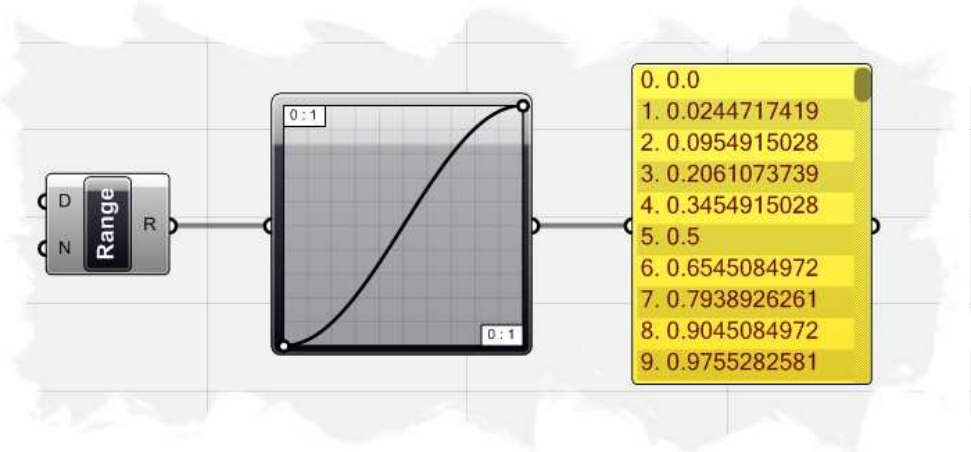
导入向导的第二步允许你设定分隔符号的类型。分隔符号是一种 CSV 文件存储的符号（比如句号，逗号，或者空格），这是用来指示在哪里数据应该被分到另外一个制表栏里。因为我们的每一列里只有一个数据，因此不需要任何特殊的分隔符号，我们只需要单击“下一步”进入到第三步。



导入向导的第三步允许你告诉 Excel 你希望数据在 Excel 里显示的格式。“常规”选项将数值数据转换为数字，“日期”选项将数据转换为日期（年/月/日），“不导入此列”选项 将保持原有数据的格式。在我们的这个例子里，选择“常规”，然后点击“完成”。

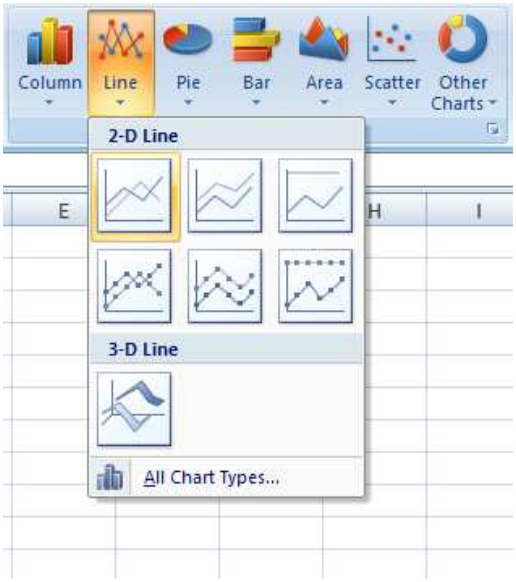


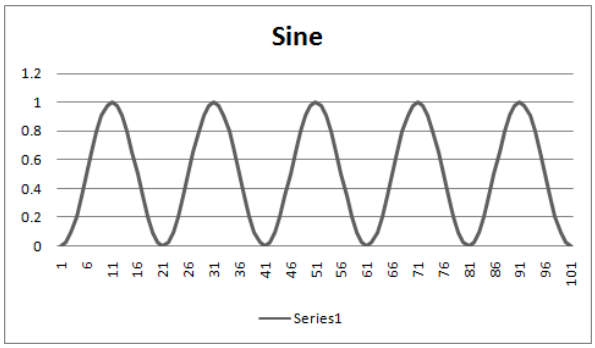
然后你会被提示选择一个单元格导入这些数据，我们使用默认的选择 A1。然后你会看到这 101 个数据在 A 制表栏里，跟 Grasshopper 里 Post-It Panal 里的数据时对应的。在 Grasshopper 中数据是不停的流动的，因此 GRASSHOPPER 里的任何改变将会自动更新这个 CSV 文件。回到 GRASSHOPPER 里改变 Graph Mapper 的类型为正弦（**Sine**）。注意 Post-it Panel 里的数据已经更新了。



切换到 Microsoft Excel 然后在“数据”标签下拉菜单下选择**“更新数据”(Refresh All)**，然后根据提示，选择那个你先前保存的 CSV 文件 ，将它重新载入到 Excel 里面。制表栏 A 里的数据将会更新。

现在，选择从 A1 到 A101 的所有的单元格（选择 A1，按住 shift，然后选择 A101），然后选择**“插入”(Insert)**标签，然后选择“**图表…”(Line Chart)**，然后选择第一个二维直线图表图表。

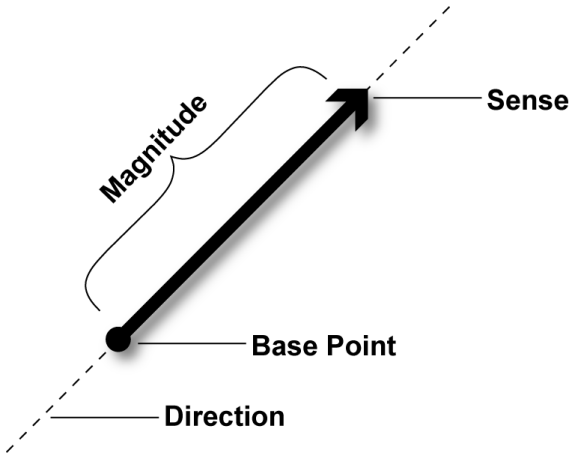




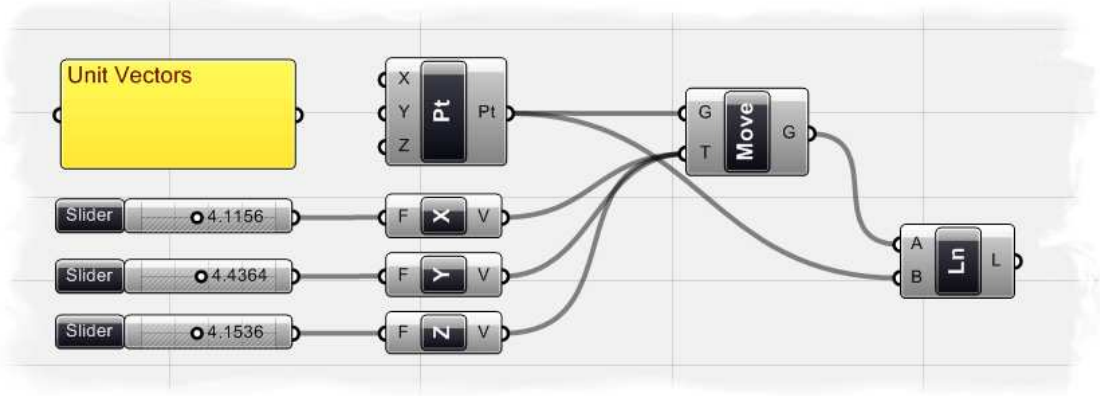
你将会看到一个直线图表，这个图表反映了和 Graph Mapper 里一样的形状。你可以 Grasshopper 里进行无限多次的改变，然后在 Excel 里选择**更新数据(Refresh All)**，你就会看到 Excel 里相应的改变。

9 Vector Basics(向量基本原理)

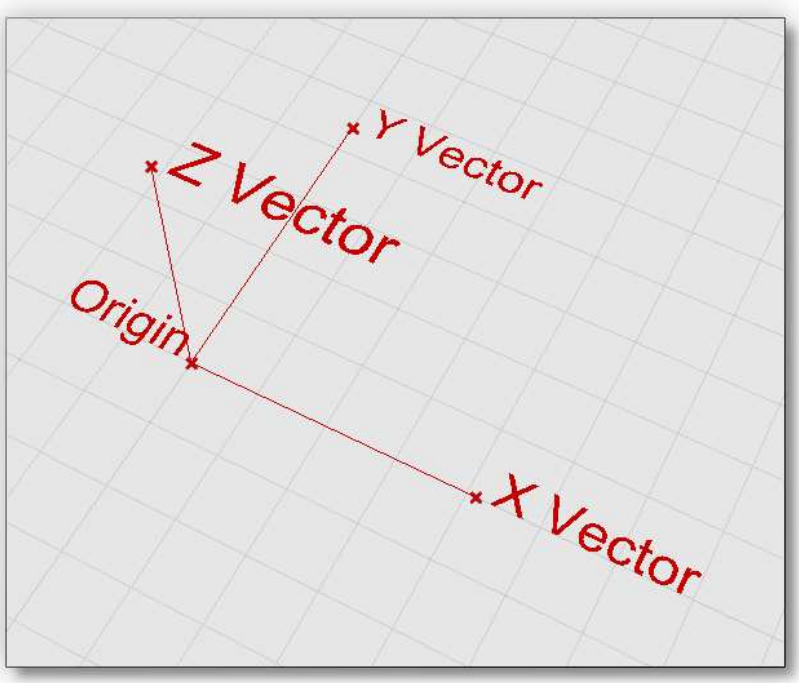
从物理学中，我们知道一个向量是一个具有长度、方向和作用点的几何对象。一个向量通常表示为一个有确定方向的线段 （通常用箭头表示），连接了起点 A 和终点 B。向量的长度即是线段的长度，其方向即是 B 点相对于 A 点的方向：怎样将点 A 移动到点 B



Rhino 中，向量与点是不能辨别的。它们都是用笛卡儿坐标系中代表 X、Y、Z 坐标值的三个双精度浮点数 （一种能存储带小数的数值的变量）来表示的。不同之处在于点的坐标是绝对的，而向量则是相对的。当我们处理表示点的三个数值时，它们表示空间中的一个特定坐标。而当我们处理表示向量的三个数值时，它们表示一个特定的方向。向量之所以被称为相对的，是因为它们仅仅表明箭头的起点和终点之间的不同。*注意：向量不是实际存在的几何实体，他们只是信息*。这意味着，Rhino 中没有向量的视觉化表示，不过我们可以用向量信息去指导特定的几何操作，例如平移、旋转和定位。



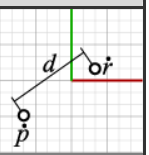
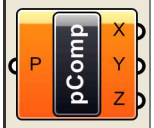
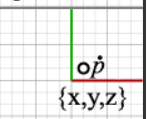
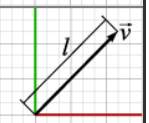
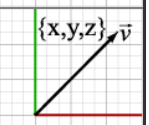
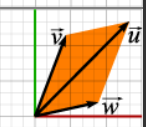
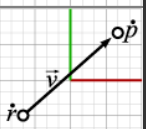
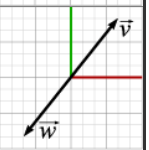
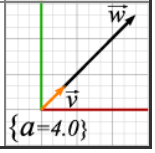
在上例中，我们首先用 Point XYZ　运算器(Vector/Point/Point XYZ)在原点 （0，0，0）处创建一个点。然后我们将 Point　运算器的 Pt 输出项与 Move 运算器的 G 输入项相连，准备沿某向量方向平移该点的复制点。为此，我们拖拽 Unit X、Unit Y　、Unit Z 运算器到工作区上(Vector/Constants)。这些运算器指定了直角坐标系 XYZ 中的一个向量方向。我们可以通过连接数字滑块与每个 UnitVector （单位向量）运算器的输入项来指定向量的长度。通过在连接 Unit Vector 的输出项与 Move 运算器的 T 输入项的同时按住 shift 键，我们可以连接多个运算器。现在你可以看一下 Rhino 的视图，你将看到一个点在原点处，以及三个分别沿 X、Y、Z 轴移动后的新点。任意改变数字滑块的值将看到每个向量的长度也发生变化。欲得到向量的视觉化表示，方法类似于画一个箭头，我们可以创建从原点到每个平移后新点的线段。为此，拖拽一个 Line 运算器到工作区上(Curve/Primitive/Line)。 连接 Move 运算器的 G 输出项与 Line 运算器的 A 输入项，以及 Point 运算器的 Pt 输出项与 Line 运算器的 B 输入项。以下是 Unit Vector （单位向量）定义的截图。



注释：向查看上例完成的定义，请在 Source Files 中打开文件 UnitVectors.ghx 文件 。

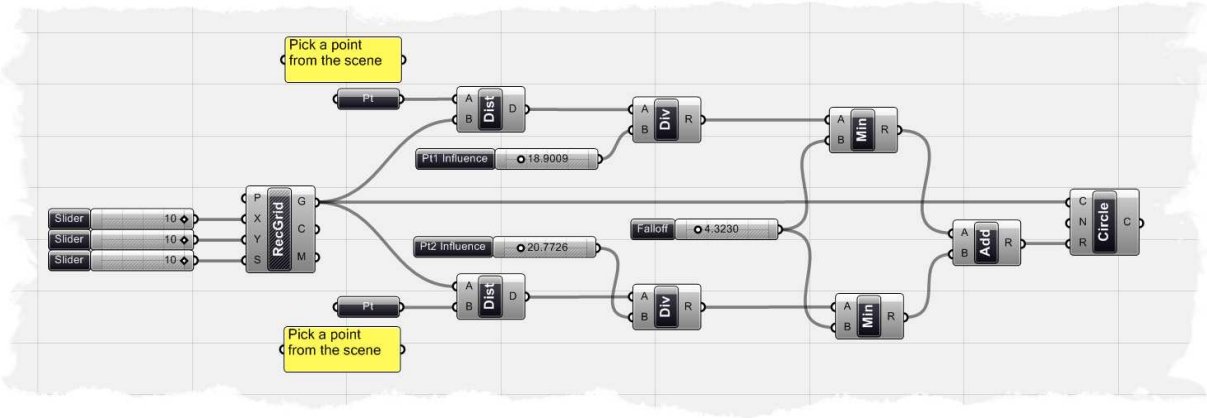
9.1 Point/Vector Manipulation （点/ 向量操作）

Grasshopper 拥有一整套 Point/Vector 运算器来执行“向量数学”中的基本操作。下表列举了一些最常用的运算器及其功能。

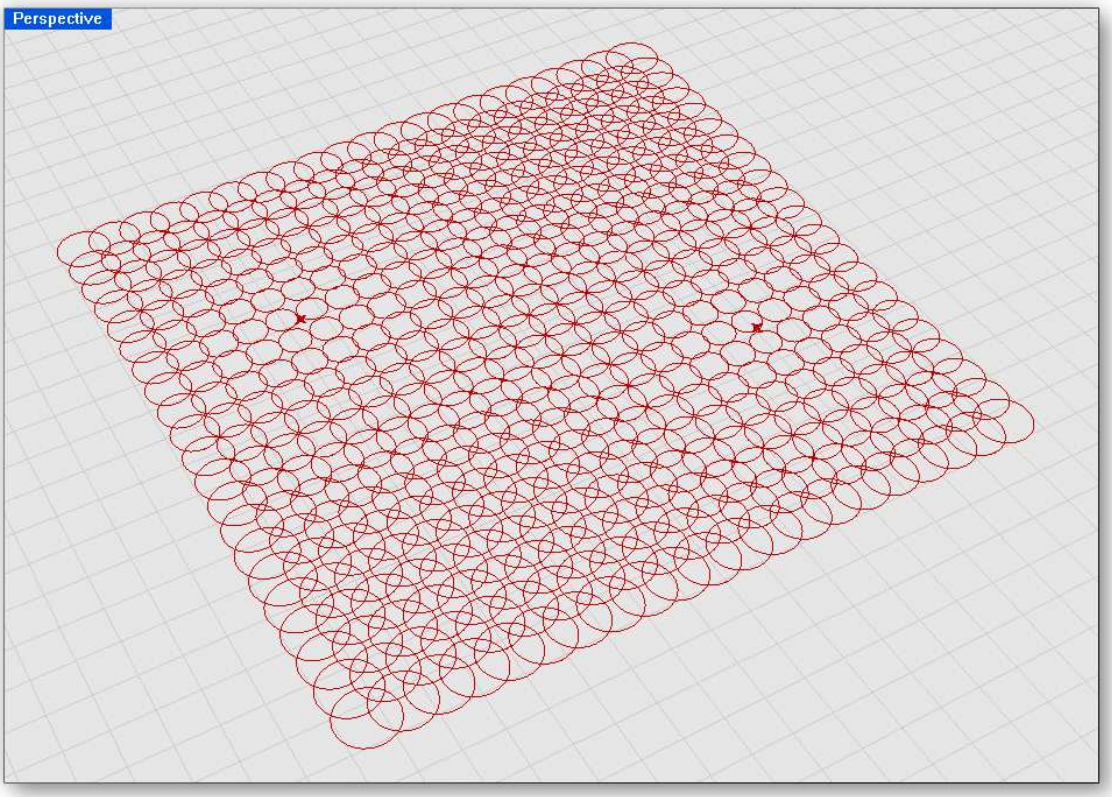
运算器	位置	描述	示例
	Vector/Point/**Distance**	计算两点间距离 （A 和 B 输入项）	
	Vector/Point/**Decompose**	将一个点分解为 X、Y、Z 运算器	
	Vector/Vector/**Angle**	计算两向量间夹角，输出其弧度值	
	Vector/Vector/**Length**	计算向量的长度	
	Vector/Vector/**Decompose**	将一个向量分解为它的 X、Y、Z 运算器	
	Vector/Vector/**Summation**	将向量 1 运算器 （A 输入项）与向量 2 运算器 （B 输入项）相加	
	Vector/Vector/**Vector2pt**	在两个定点间创建一个向量	
	Vector/Vector/**Reverse**	取所有坐标的相反数使某向量反向，长度不变	
	Vector/Vector/**Unit Vector**	将向量各坐标除以向量的长度，得到长度为 1 的单位向量，有时这种操作也被称	
	Vector/Vector/**Multiply**	将向量的各坐标乘以一个特定的因数	

9.2 Using Vector/Scalar Mathematics with Point Attractors (Scaling Circles)
对 Point Attractors 使用 Vector/Scalar Mathematics (缩放圆形)

我们已经知道了矢量和标量数学的基础知识，我们来看个例子。这个例子中有一个由圆圈构成的网格，我们要根据这些圆圈的圆心距离一个点的距离来缩放这些圆圈



注意注意：：要看这个例子的最终定义，在 Source Files 中打开文件 Attractor_2pt_circles.ghx
以上的图就是要生成下面的缩放的圆圈图所需的定义。



开始创建截图中的这个定义：

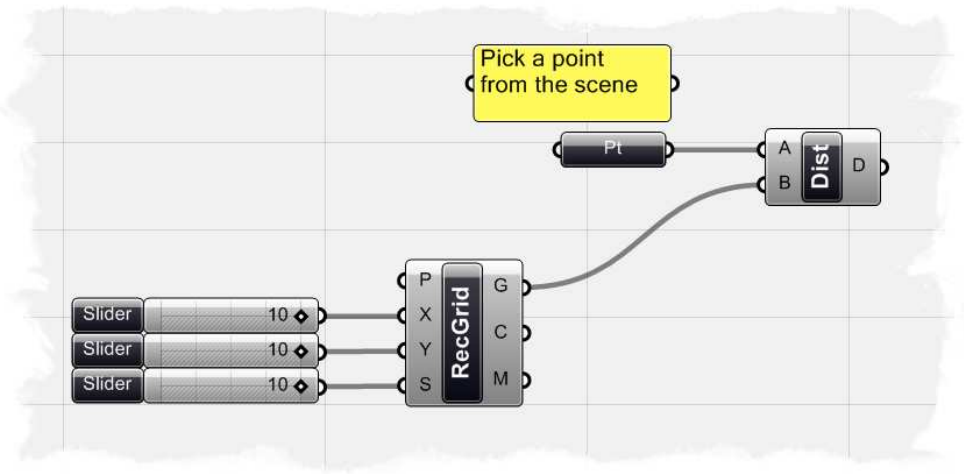
- Params/Special/Numeric Slider—一开始就要把这三个滑块拖到工作区上
- 右键单击这三个滑块来设置：
  - Slider Type （滑块类型）：Integers （整数）
  - Lower Limit （下限）：0.0
  - Upper Limit （上限）：10.0
  - Value （数值）：10.0
- Vector/Point/Grid Rectangular—拖拽一个 Rectangular Point Grid 运算器到工作区
- 连接第一个滑块到 Pt Grid 运算器的 X 输入项
- 连接第二个滑块到 Pt Grid 运算器的 Y 输入项
- 连接第三个滑块到 Pt Grid 运算器的 S 输入项

  这个 Rectangular Point Grid 运算器生成了一个点组成的网格，其中 P 输入项是这个 网格的原点（在我们的例子中我们用 0，0，0）这个网格运算器创建了很多有 X 和 Y 坐标的点，并由数字滑块定义。要注意我们设置了限值为 10.0.如果你数一下行和列，你会发现都是 20 列。这是因为它是从中心点开始向两边平移生成的。所以你得到了两倍数目的 X 和 Y 的点。这个 S 输入项决定了点之间的空间
- Params/Geometry/Point—拖放一个 Point 运算器到工作区上

  这是一个固有的 implicit 运算器，因为它的输入值是固定值。（有关固定值类型请看第四章）。这个运算器和以前我们用过的其他的 xyz 运算器不同，它不会创造一个点，除非你在场景中真的设置一个点。当然，在 Rhino 的场景中必须已经有一个点了。
- 在 Rhino 的场景中，在对话框中输入“Point”并在场景的其他地方任意设置一个点。在这里，我们要把点放在顶视图中，这样所有的点才都在 xy 平面上。

  我们在场景中创造了一个 attractor point 物体后，我们可以关联它们到我们刚刚在 Grasshopper 中的点。
- 右击 Point 运算器并选择“Set One Point”
- 当提示后，选择你在 Rhino 场景中创建的 attractor point 物体

  现在我们已经创建了一个点的网格并且在场景中引用了一个 attractor point。我们可以用矢量数学来计算每个点到 attractor point 的距离。.
- Vector/Point/Distance—拖放一个 Distance 运算器到工作区
- 连接 attractor point 运算器输出项到 Distance 运算器的 A 输入项
- 连接 rectangular Grid Point 运算器的 G 输出项到 Distance 运算器的 B 输入项



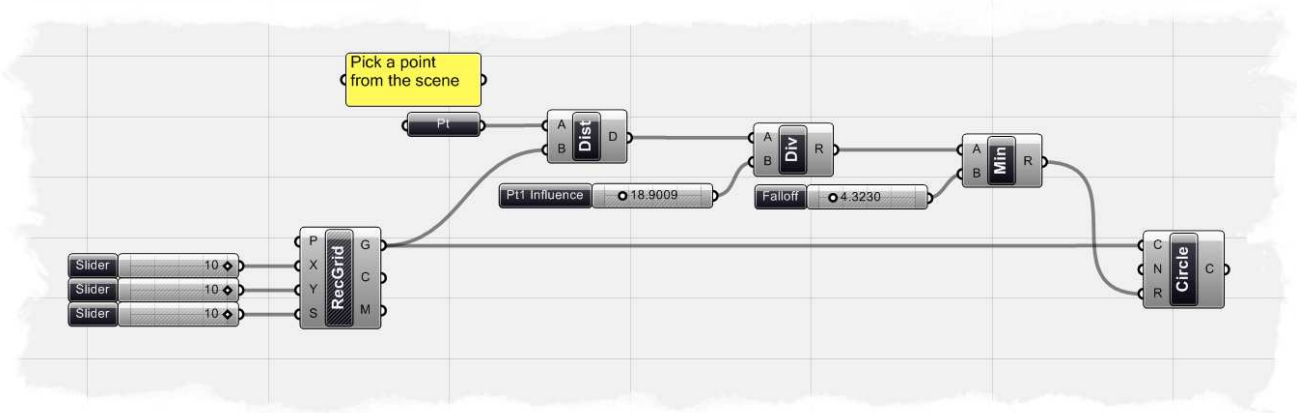
第一步的定义请看上面截屏。如果我们把鼠标停留在每个 Distance 运算器的 D 输出项上，我们可以看到一组关联的数显示这些点到 attractor point 的距离。我们将用这些值来定义每个圆圈的半径，但首先我们得缩小这些数字来得到更适合的半径。

- Scalar/Operators/Division—拖放一个 Division 运算器到工作区
- 连接 Distance 运算器的 D 输出项到 Division 运算器的 A 输入项
- Params/Special/Numeric Slider —拖放一个 numeric slider 运算器到工作区
- 右键单击滑块并按照以下设置：
  - Name: Pt1 Influence
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 25.0
- 连接 Pt1 Influence 滑块到 Division 运算器的 B 输入项

  因为距离值相当大，我们需要一个缩放参量来把数字减小到可控制的范围内。我们用到数字滑块作为分子来得到一个新的输出值，这样就可以用来赋值给圆圈的半径了。我们可以直接输入这些值到 Circle 运算器中，但为了让我们的定义更准确，我们可以设置一个最小值来不让圆圈半径小于这个值。
- Scalar/Utility/Minimum—拖放一个 Minimum 运算器到工作区
- 连接 Division 运算器的 R 输出项到 Minimum 运算器的 A 输入项
- Params/Special/Numeric Slider —拖放一个 Numeric Slider 运算器到工作区
- 右键单击滑块并按照以下设置：
  - Name: Falloff
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 30.0
  - Value: 5.0
- 连接 Falloff 滑块到 Minimum 的 B 输入项
- Curve/Primitive/Circle CNR——拖放一个 Circle CNR (Center, Normal, and Radius)运算器到工作区

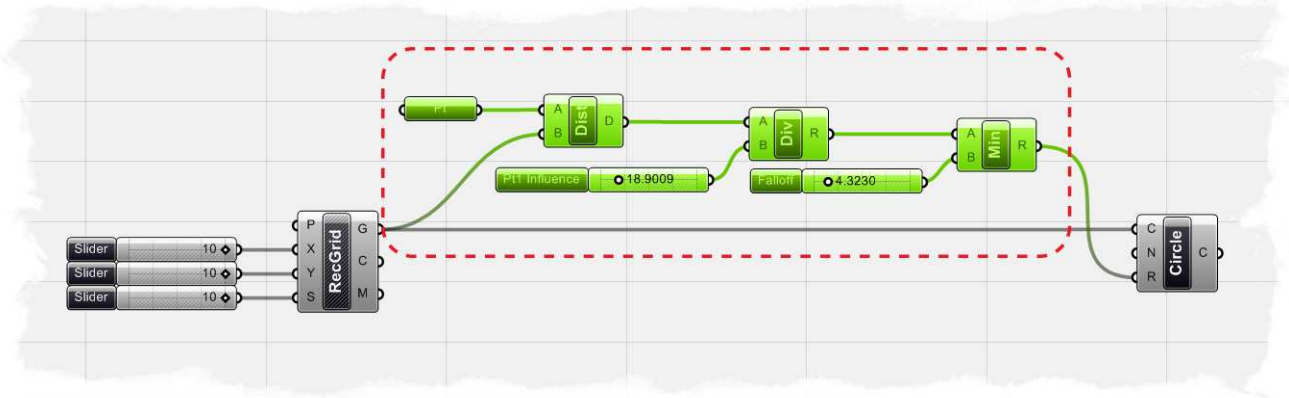
  我们希望每个圆圈的圆心都落到我们事先定义好的网格中。
- 连接 Rectangular Point Grid 运算器的 G 输出项到 Circle 的 C 输入项
- 连接 Minimum 运算器的 R 输出项到 Circle 运算器的 R 输入项

- 右键单击 Rectangular Point Grid 运算器并取消对 Preview 的勾选



你们的设置应该如上图中所示。现在我们已经创建了一系列经过缩放的圈圈并且这些圆圈是以他们的圆心到 attractor point 的距离为依据缩放的。但是，如果我们想再加入一个 attractor point 该怎么做呢？因为我们已经有了以上的设置，所以我们仅仅需要复制并且粘贴一些运算器就可以实现他了。

- 选择以下运算器：Attractor Point parameter, Distance component, Pt 1 Influence slider, Division component, Falloff slider, Minimum 并且按下 Ctrl-c (复制) 和 Ctrl-v (粘贴)来复制这些运算器
- 将复制的部分向工作区下方移动一点，以保证这些运算器不会互相重叠在一起

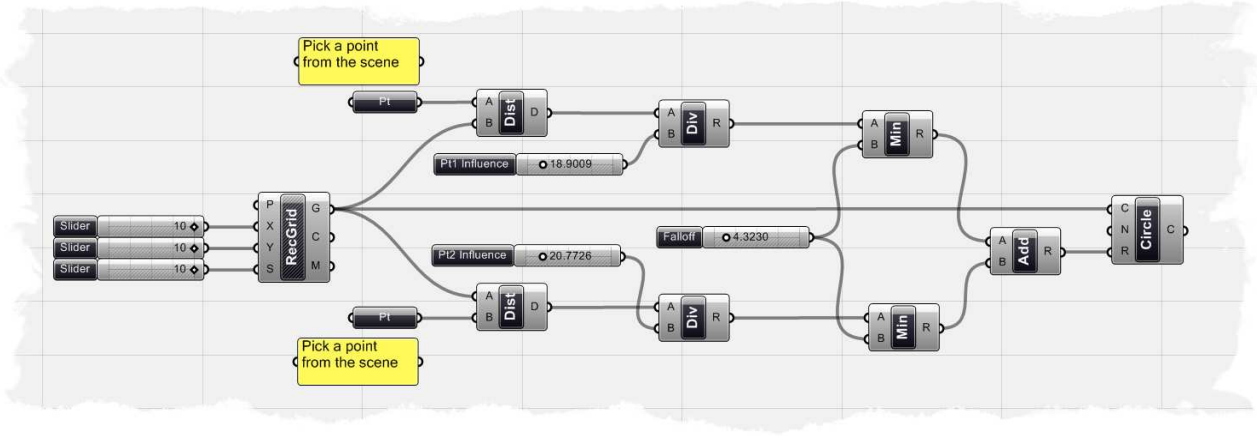


我们需要再指定一个 point attractor…但是当然，在做这个之前我们首先要在 Rhino 视图中的创建这个点

- Rhino 视图中，在对话框中键入"Point"命令并将这个点放置在视图中任意一个位置。就像我们曾经做过的一样，在顶视图中放置这一个点以保证这个点是在 Rhino 中的 XY 平面上。
- 右键单击 duplicated Point 运算器并选择"Set One Point"
- 当提示后，选择我们刚才在 Rhino 视图中创建的 attractor point

  现在我们有 2 行运算器，他们计算了与 rectangular Point Grid 的距离并可以让我们控制圆的半径。但是，我们需要将这 2 组由 Minimum 运算器的输出的数据组组成一组数据。为了达到这个目的，我们需要使用 Addition 运算器。
- Scalar/Operators/Addition——拖放一个 Addition 运算器到工作区
- 连接第一个 Minimum 运算器的 R 输出项到 Addition 运算器的 A 输入项

- 连接第二个 Minimum 运算器的 R 输出项到 Addition 运算器的 B 输入项
- 现在，连接 Addition 运算器的 R 输出项到 Circle 运算器的 R 输入项(*注意: 这会取代原来的连接*)

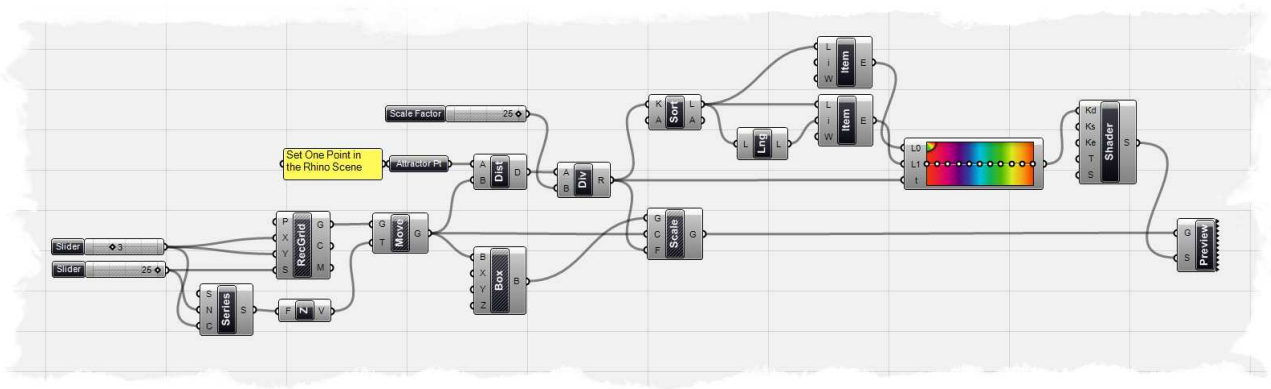


如果完全正确的话，你们的设置应该和上面的截图是一样的。你可能会注意到我已经删除了一个 Falloff sliders 并且我使用了一个滑块来控制 2 个 Minimum 运算器。试着任意调整 Pt Influence 和 Falloff 滑块的值来观察圆圈是如何根据距离缩放的。

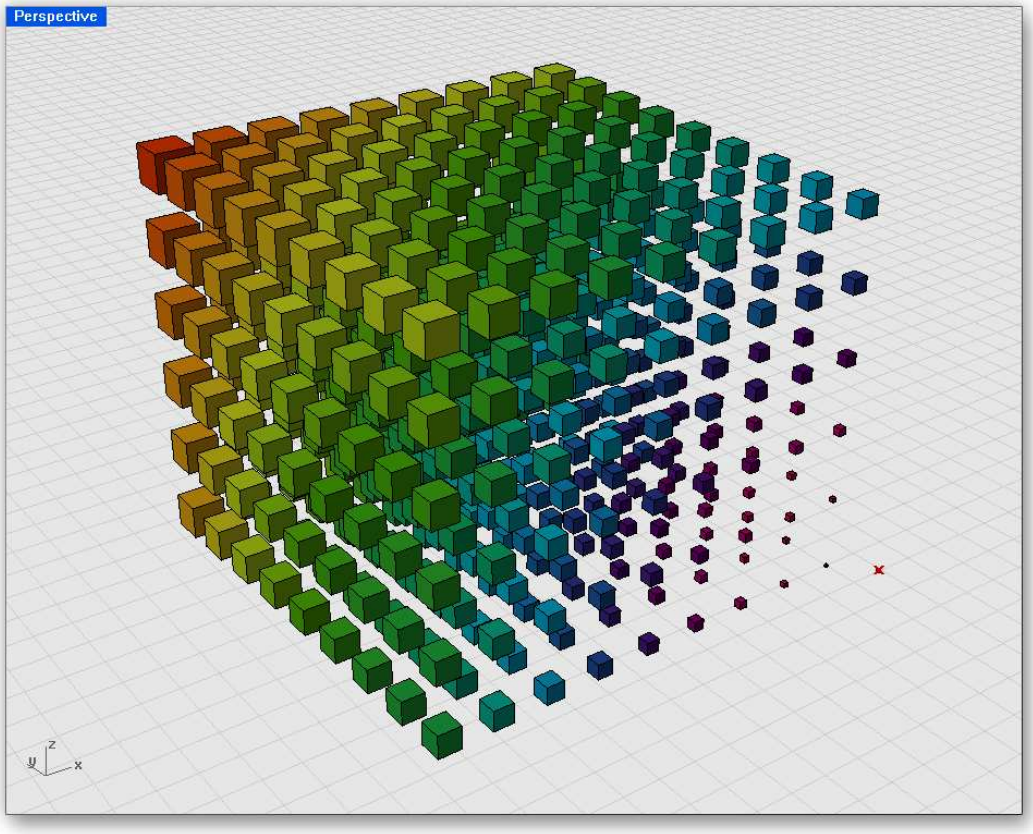
注意: 要看这个例子的视频教程，请访问 David Fano's 的:
http://designreform.net/2008/07/08/grasshopper-patterning-with-2-attractor-points/

9.3 Using Vector/Scalar Mathematics with Point Attractors (Scaling Boxes)
对 Point Attractors 使用 Vector/Scalar Mathematics (修改立方体)

我们已经示范了使用 Vector/Scalar Mathematics 来定义圆圈的半径，这些半径都是由圆心到其他点物体的距离决定的。但是我们也可以用同样的内核的运算器来缩放物体并通过了 Grasshopper 的 shader 运算器来定义物体的颜色。接下来的例子生成效果如图所示，我们来一步步看是怎么定义的。



注意：要看这个例子最后完成的定义，在 Source Files 夹打开文件 Color Boxes.ghx



第一步：首先创建一个三维点阵

- Params/Special/Numeric Slider ——拖放 2 个 sliders 运算器到工作区
- 右键单击第一个 slider 并设置：
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 10.0
  - Value: 3.0
- 右键单击第二个 slider 并设置：
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 25.0
  - Value: 25.0
- Vector/Point/Grid Rectangular ——拖放 1 个 rectangular Point Grid 运算器到工作区
- 连接第一个 slider 分别到 Point Grid 运算器的 X 输入项和 Y 输入项
- 连接第二个 slider 到 Point Grid 运算器的 S 输入项

  在你的场景中能看到一个点阵网格，其中第一个滑块控制 x 和 y 轴上的点的数目（ 记住，由于它是从中心点生成的，所以行和列的数目总是会加倍），这些点之间的空间由第二个滑块控制。我们需要复制这个点阵到 z 轴来形成一个三维体量
- Logic/Sets/Series——拖放 1 个 Series 运算器到工作区
- 连接第二个 slider 到 Series 运算器的 N 输入项
- 连接第一个 slider 到 Series 运算器的 C 输入项

  运算器会计算我们复制到 z 方向上点的网格数目。你可能注意到其中有个小错误，由于我们是从中心生成的，所以尽管我们设置这个 xy 轴的数目到 3，实际上在各个方向我们都有 7 个点，由于我们最后会有三维的立方点阵，我们需要复制 7 个到 z 轴中，这样才连续。为了统一，我们要写一个简单的说明来加倍 series count
- 右击 Series 运算器的 C 输入项并滚动选取 Expression Tab
- Expression editor 中输入一下等式：(C*2)+1

  我们现在告诉运算器用 Series 运算器的 C 输入项去乘以二并加上 1，因为我们的原始值是 3，所以 Series count 会是 7.
- Vector/Constants/Unit Z ——拖放一个 Unit Z vector 运算器到工作区
- 连接 Series 运算器的 S 输出项到 Unit Z 运算器的 F 输入项

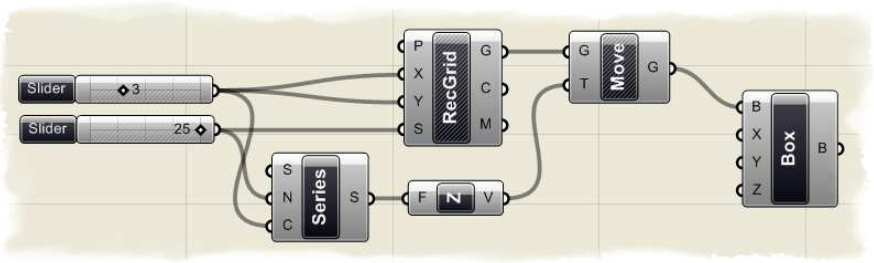
  如果鼠标停留在 Unit Z vector 运算器上，你会看到我们定义了 7 个值，z 值在每个上面都增加了 25，即我们在第二个 slider 指定的数值。我们将用这个向量值来定义我们复制好的点阵的空间距离

- X Form/Euclidean/Move ——拖放一个运算器到工作区

- 连接 Point Grid 运算器的 G 输出项到 Move 运算器的 G 输入项
- 连接 Unit Z 运算器的 V 输出项到 Move 运算器的 T 输入项

  如果你看 Rhino 中的场景，你会注意到我们的点不像是三维的立方体。总之，它看起来像一个倾斜的矩形点阵。是这是因为默认数据匹配规则被设置到 longest list 中了。右击运算器，可以改变规则为 cross reference。现在看起来应该就是三维的了。（看第六章数据流的匹配）
- Surface/Primitive/Center Box –拖放一个 Center Box 运算器到工作区
- 连接 Move 运算器的 G 输出项到 the Center Box 运算器的 B 输入项
- 右击 Point Grid 和 Move component 运算器并关掉 Preview

  下面是设置截屏.



第二步：解决缩放器和向量

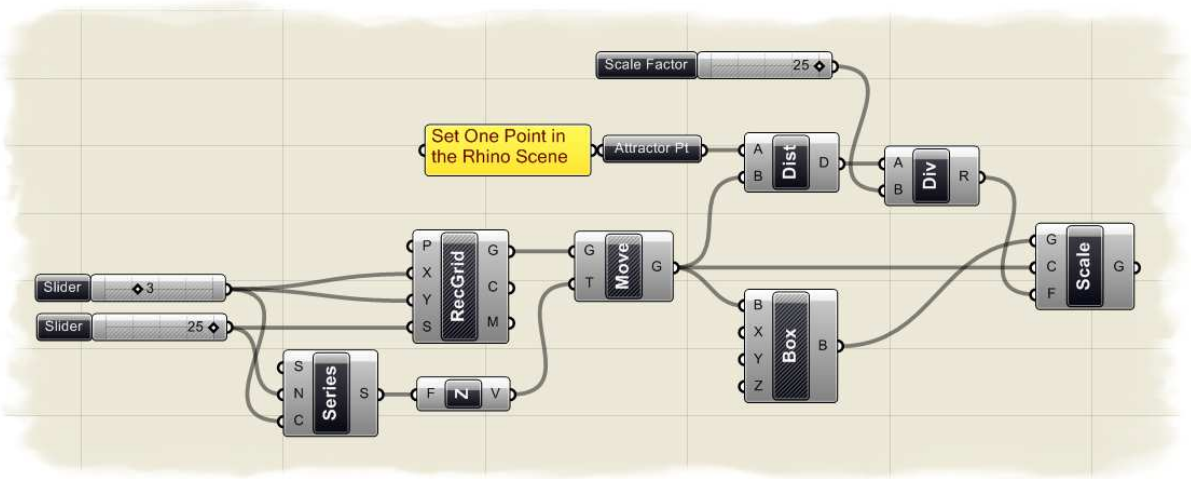
- 至 Params/Geometry/Point –拖放一个 Point 运算器到工作区
- 右击 Point 运算器并重命名为“Attractor Pt”
- 正如在缩放圆圈的例子中，我们需要事先赋值于 Rhino 场景中的点 。
- 在 Rhino 的场景中，在对话栏中输入 Point 并在场景中随意设置一个点。
- 回到 Grasshopper，右击 Attractor Pt 运算器选择“Set One Point”
- 当提示后，选择你刚刚在 Rhino 中创建的点

  你现在可以看到有一个小红 X 在点上面表明这个 Attractor Pt 运算器已经赋值了。如果你在场景中移动它，Grasshopper 会自动更新它的位置
- Vector/Point/Distance –拖放一个 Distance 运算器到工作区
- 连接 Attractor Pt 的输出项到 Distance 运算器的 A 输入项
- 连接 Move 运算器的 G 输出项到 Distance 运算器的 B 输入项

  如果我们停留鼠标到 Distance 运算器的 D 输出项，我们可以看到一个数值表单表明各个点到 Attractor Pt 的距离。为了把这些值作为缩放参数，我们需要把它们的值减小到适合的水平。.
- 至 Scalar/Operators/Division –拖放一个 Division 运算器到工作区
- 连接 Distance 运算器的 D 输出项到 Division 运算器的 A 输入项
- Params/Special/Numeric Slider –拖放一个 Numeric Slider 到工作区
- 右击滑块并设置：
  - Name: Scale Factor
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 25.0
  - Value: 25.0
- 连接缩放参数滑块到 Division 运算器的 B 输入项

- X Form/Affine/Scale –拖放一个 Scale 运算器到工作区
- 连接 Center Box 运算器的 B 输出项到 Scale 运算器的 G 输入项
- 连接 Division 运算器的 R 输出项到 Scale 的 F 输入项
- 右击 Center Box 运算器并关闭 Preview

下面是目前为止的截屏。现在 Rhino 里的所有 box 都根据他们到 attractor 的距离进行了缩放。我们可以进一步给它们加上颜色来表明他们的缩放程度.

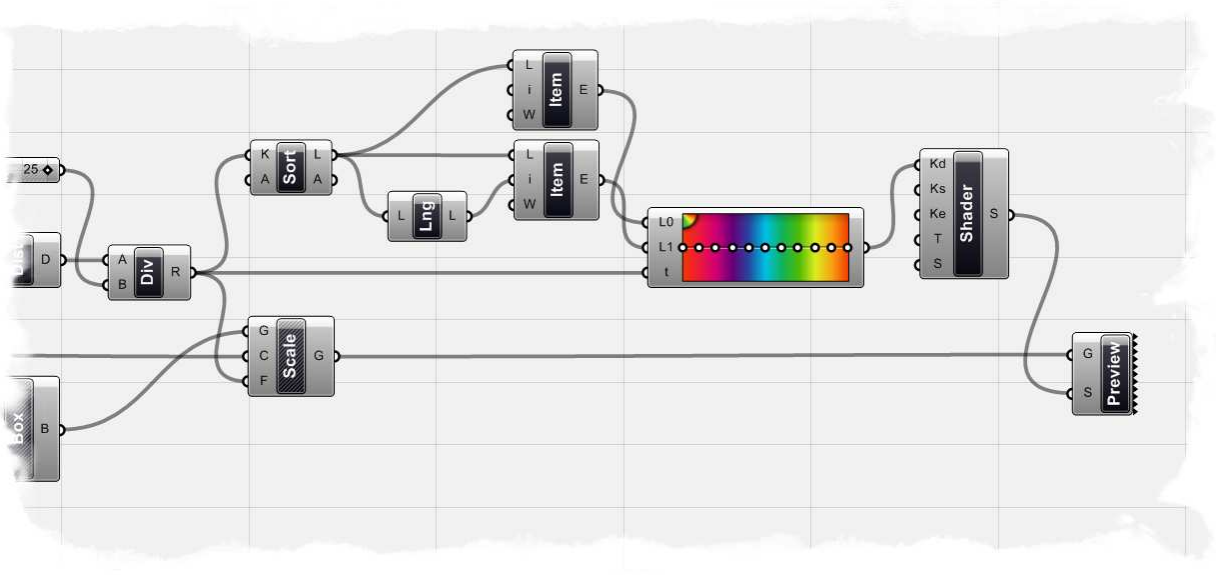


第三步：给每个 box 的颜色赋值

- Logic/List/Sort List –拖放一个 Sort List 运算器到工作区
  为了根据 box 到 attractor 点的距离来给 box 颜色赋值，我们需要两个数值，最近点和最远点， 这样我们就要查找我们的距离表，来得到表单的头上和末尾的值。
- Logic/List/List Item –拖放一个 List Item 运算器到工作区
- 连接 Sort List 运算器的 L 输出项到 List Item 运算器的 L 输入项
- 右击 List Item 运算器的 I 输入项，设置整数 0.0
  这就找到了整个表单的第一个，也就是最小的距离
- Logic/List/List Length –拖放一个 List Length 运算器到工作区
- 连接 Sort List 运算器的 L 输出项到 List Length 运算器的 L 输入项
  这个长度运算器可以告诉我们表单里有多少入口，我们可以把这些信息输入别的表单运算器来找到表里的最大值
- Logic/List/List Item-拖放另一个 List Item 运算器到工作区
- 连接 Sort List 运算器的 L 输出项到 the List Item 运算器的 L 输入项
- 接 List Length 运算器的 L 输出项到第二个 List Item 运算器的 I 输入项
  如果你把鼠标停留在第二个 List Item 运算器的 E 输入项上，你会发现运算器中并没有找到最大值。这是因为 Grasshopper 的数值总是把第一个数值作为初始值 0 来储存。所以如果我们的表长度表明有 100 个值在表中，而第一个数字是以 0 开头的话，我们最后一个值实际上是 99.我们必须对第二个表单 List Item 运算器的 I 输入项上加一个说 明，减去一个 1，这样才能找到实际的最后值。
- 击第二个 List Item 运算器的 I 输入项并选“Expression Editor”

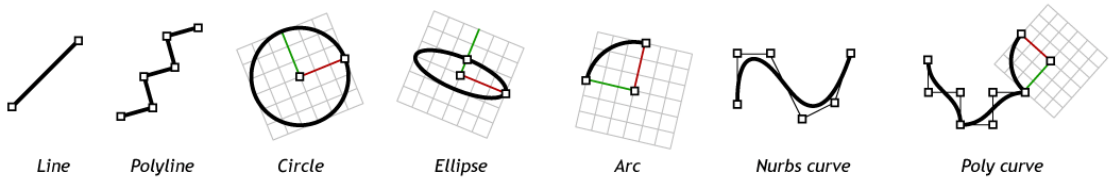
- 在对话框中输入下面等式：i-1
  现在如果停留鼠标在第二个 List Item 运算器的 E 输入项上，你可以看到一个数值与距离值的关联，就是那个距离 attractor 点最远的距离.
- Params/Special/Gradient –拖放一个 Gradient 运算器到工作区
- 连接第一个 List Item 运算器的 E 输出项(与最近距离关联的距离值)到 the Gradient 运算器的 L0 输入项
- 连接第二个 List Item 运算器的 E 输出项(与最远距离关联的距离值)到 the Gradient 运算器的 L1 输入项
- 连接 Division 运算器的 R 输出项到 the Gradient 运算器的 T 输入项
  TL0 输入项定义了哪个数值可以代表左边的斜率，在此例中，左边的斜率表明了最近的 box 到 attractor 点的距离，L1 输入项定义了哪个数值代表右边的斜率，即代表最远的距离。Gradient 运算器中的 T 输入项的值代表了你在斜率范围内想制表的数值表单。我们输入了全部的缩放参量，这样每个 box 的缩放都会和颜色变化的斜率相关联
- Vector/Color/Create Shader –拖放一个 Create Shader 运算器到工作区
- 连接 Gradient（梯度）运算器的输出项到 Shader 运算器的 Kd 输入项
  Shader （着色）运算器有很多输入值来帮助定义你希望你的预览成什么样。下面简介了输入值怎样影响渲染结果。
  Kd: 定义了 Shader（着色）颜色的扩散。这会定义每个物体的初始颜色，扩散颜色由三个整数值 0-255 决定，表示了红绿蓝的值。
  Ks: 定义了特殊的高光并需要输入 RGB 的三位数值。
  Ke: 定义了 Shader（着色）的 emmissivity，或者说 Shader（着色）的明度。
  T: 定义了 Shader （着色）的透明度。
  S: 定义了 Shader （着色）的光泽度，0 说明没有，100 最大
  我们连接了 Gradient （梯度）滑块到 Shader （着色）运算器的固有色输入项，因此 Gradient （梯度）图案会呈现我们的 Box 的主要颜色颜色。你可以通过选择梯度类型中的其中一个小白点改变梯度的颜色并定义颜色的输入和输出的值。你也可以上下拖动 Gradient（梯度）的长度来控制你希望改变颜色值的位置。另外，这里有很多预设值在 Gradient （梯度）运算器中。右击 Gradient（梯度）类型就可以选择四种预设类型中的一个。此例选择的是 spectrum 光谱。
- Params/Special/Custom Preview –拖放一个 Custom Preview 运算器到工作区
- 连接 Scale 运算器的 G 输出项到 Custom Preview 运算器的 G 输入项
- 连接 Shader 运算器的 S 输出项到 Custom Preview 运算器的 S 输入项
- 右击并关掉 scale 运算器的 Preview

下面就是我们经过这三步后的截屏。如果你在 rhino 中移动你的 attractor 点，这个缩放的 box 以及颜色会自动更新的。



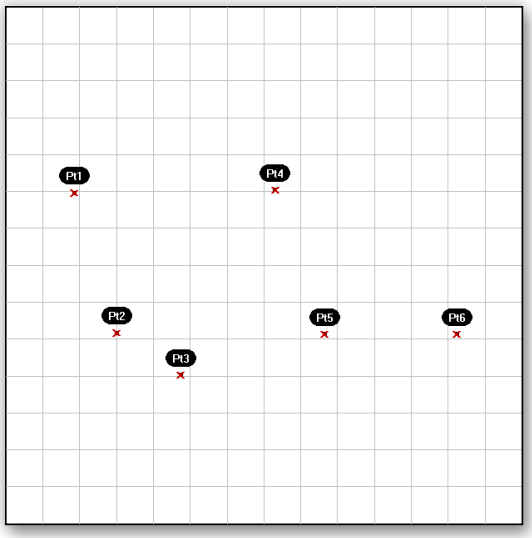
10 Curve Types（曲线类型）

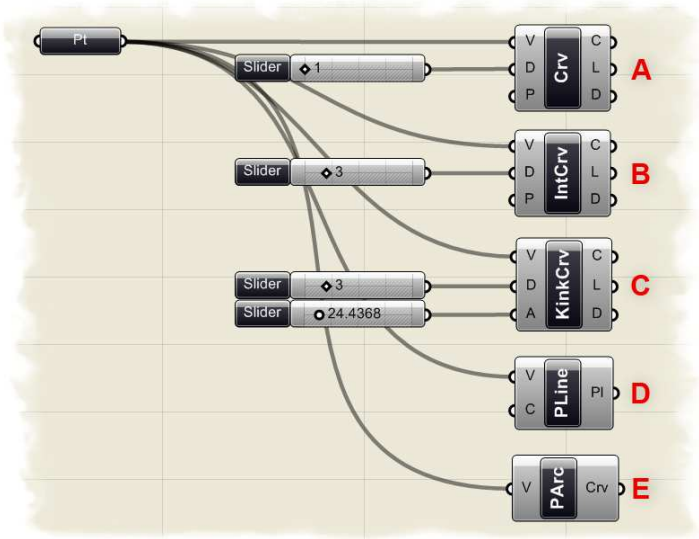
由于曲线是几何物体，它们有大量的属性和特性可用来描述或者研究。例如，每个曲线都有起点和终点坐标，当这两个坐标点间距是 0，曲线就闭合了。同样，每个曲线都有很多控制点，如果都在一个平面内，这曲线也是平面的。一些属性对整个曲线都适用，有些就只有一些特定点是适用的。例如，平面性是一个普适属性，而正交矢量是一个特定的属性。同样， 一些属性只适用一些曲线类型。我们目前讨论了 Grasshopper 的 ***Primitive Curve 运算器***，如 直线，圆，椭圆，以及圆弧



Grasshopper 也有一系列的工具来表达 Rhino 的更高级曲线，如 NURBS 曲线和复合曲线。下面， 由这个例子来看看 Grasshopper 的 spline 运算器。但首先我们需要创建一些用来定义曲线所的点。

打开和这个手册一起的 Source Files 文件，打开 Curve Types.3dm。在场景中，有 6 个点在 x-y 平面上。我从左到右地标记了他们，如右图，顺序是根据在 Grasshopper 中选择的顺序。

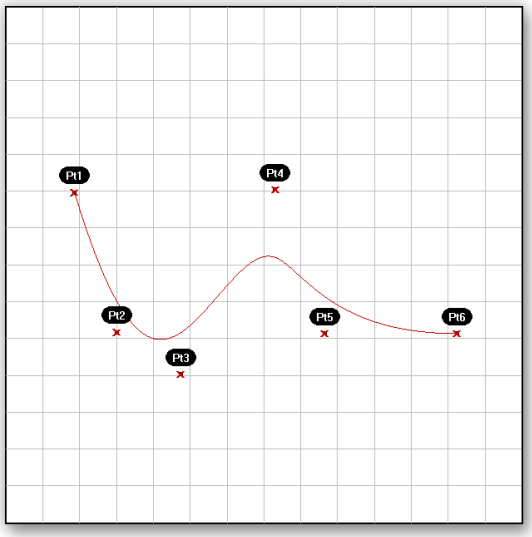




在 Grasshopper 中，在 Source Files 中打开 Curve Types.ghx 你可以看到一个点运算器链接到几个曲线运算器，这几个运算器用不同的方法定义了曲线。我们分别来看每个运算器，首先我们必须对 RhinoSet　场景中的点赋值到 Grasshopper 的 Point 参量中。要这样做就右击 Point 参量选择 Set MultiplePoints.当弹出选择每个点时，确定按照顺序选择，从左到右。这时，一条曲线出现了，选完 回车回到 Grasshopper，全部的 6 个点都有个小红 x 在顶部，表明这个点已经被赋值到 Grasshopper 的 Point 参量中。.

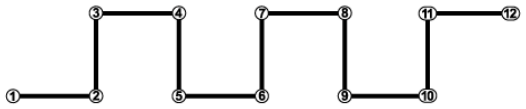
A) NURBS Curves (Curve/Spline/Curve)

非统一均匀 B 样条曲线， 或 NURBS 曲线都是 Rhino 中可利用的精确定义的一种曲线。除了帮助定义曲线位置的控制点外，NURBS 曲线也有特定的属性比如阶数 Degree，节点 Knot，权重 Weight。有一整本书（至少有非常长的篇章）都在写 NURBS 曲线背后的数学原理，要获得更多信息，请访问 http://en.wikipedia.org/wiki/NURBS.

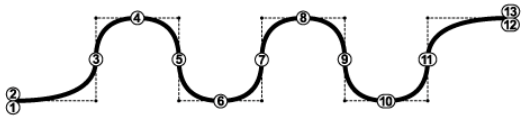


URBS Curve 运算器的 V 输入项定义了曲线的控制点，选择了 Rhino 场景中的点后就可以准确地描述这些控制点了。NURBS Curve 运算器的 D 输入项设定了曲线的阶数。阶数一般都是 1-11 的正整数。一般来说阶数决定了控制点影响曲线的范围，范围越高，值越大。下页的表出自 David Rutten 的手册，Rhinoscript 101，说明了改变阶数会如何影响 nurbs 曲线的结果。

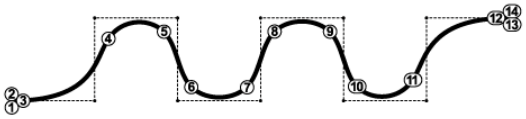
NURBS curve knot vectors as a result of varying degree



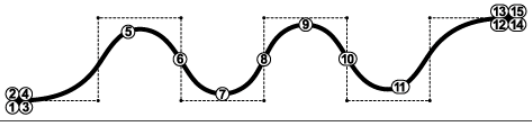
A D^1 nurbs curve behaves the same as a polyline. It follows from the knotcount formula that a D^1 curve has a knot for every control point. Thus, there is a one-to-one relationship.



A D^2 nurbs curve is in fact a rare sighting. It always looks like it is over-stressed, but the knots are at least in straightforward locations. The spline intersects with the control polygon halfway each segment. D^2 nurbs curves are typically only used to approximate arcs and circles.

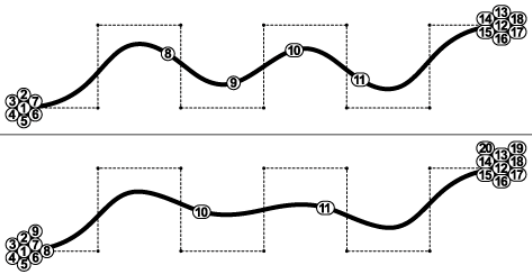


D^3 is the most common type of nurbs curve and -indeed- the default in Rhino. You are probably very familiar with the visual progression of the spline, eventhough the knots appear to be in odd locations.



D^4 is technically possible in Rhino, but the math for nurbs curves doesn't work as well with even degrees. Odd numbers are usually preferred.

D^5 is also quite a common degree. Like the D^3 curves it has a natural, but smoother appearance. Because of the higher degree, control points have a larger range of influence.



D^7 and D^9 are pretty much hypothetical degrees. Rhino goes all the way up to D^{11}, but these high-degree-splines bear so little resemblance to the shape of the control polygon that they are unlikely to be of use in typical modeling applications.

注释：NURBS 曲线 Knot(节点) 向量是改变曲线 Degree(阶数) 的一个原因

¨ 1 阶的 NURBS 曲线类似于 polyline（复合线）。它遵循计算节点数目的法则，每个控制点都有一个节点。这样使得它们是一一对应的关系。

¨ 2 阶的 NURBS 曲线实际上不太多，看起来总是压力过度，但至少节点是排列在一条直线上的。样条线和控制点多边形相交于每段的平分点位置。2 阶 NURBS 曲线几乎只用在弧和圆形中。

¨ 3 阶的 NURBS 是最常见的一种 NURBS 曲线，事实上也是 Rhino 默认的曲线类型。你可能会对样条线的结构十分熟悉，哪怕节点出现在很多奇怪的位置上。

¨ Rhino 中 4 阶的 NURBS 曲线技术上来说是没有问题的，当和相同阶数的曲线匹配还有些问题。通常阶数推荐使用奇数

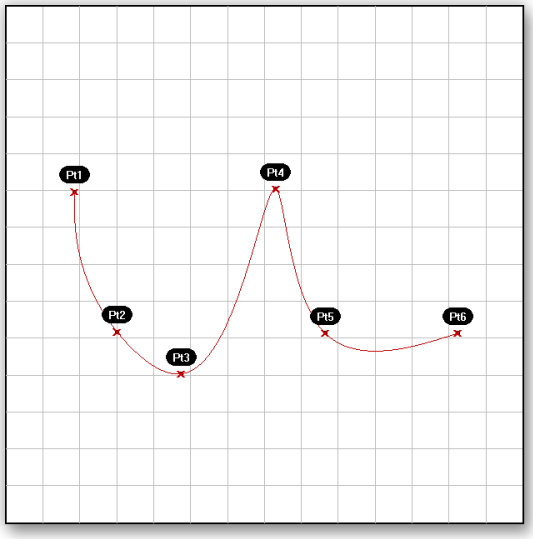
¨ 5 阶曲线也是比较常见的阶数。和 3 阶曲线的属性一样 ，但是更加的顺滑。因为阶数越高，控制点的影响范围会越大。

¨ 7 阶与 9 阶都是比较少用的阶数。Rhino 最高支持到 11 阶，但阶数越高的样条线形状会接近控制多边形形状，因此他们在常见的模型中很少应用到。

在我们的范例中，我们有给 curve D 的输入链接一个数字滑块来定义曲线阶数。从左至右 的拖动滑块，我们可以及时的看到每个控制点的变化。NURBS Curve-P Inputt 使用一个 布尔值来定义曲线的周期性. 一个 False 的输入值将会创建一个开放的 NURBS 曲线,True 的输入

值将会得到一个封闭的 NURBS 曲线。NURBS 曲线的三个输出部分看起来很清晰明了，C 用来定义作为输出结果的线，L 用来提供输出一个线的长度值， D 用来输出曲线的区间值 （或是曲线阶数 0 至 X ）

B) Interpolated Curves 内插点曲线 (Curve/Spline/Interpolate)

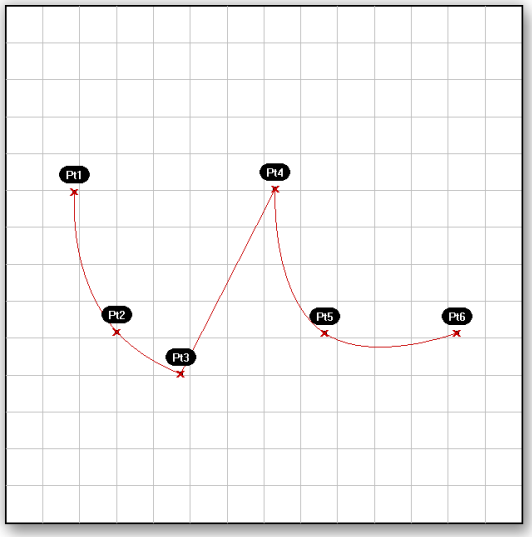


Interpolated Curves 与 Curve 稍微有些不同，内插点曲线通过控制点.你发现，通过指定的坐标来创建 NURBS Curve 是比较困难的事情. 即使我们移动单个控制点，来创建通过控制点的曲线将也是非常的困难。 输入 ， Interpolated Curves. V-Input 比较类似控 Nurbs Curve 的 V 输入项,接着，他会询问指定点来创建曲线。当然，Interpolated Curves 方式所创建的曲线会自动的通过这些点，不考虑曲线的阶数. 在创建 N 控制点曲线，我们仅在阶数设置为 1 的时候才能有这样的效果。当然，像 URBS Curve 运算器中，D- input 定义所建曲线的阶数。当然，这种方式仅用在阶数输入值为奇数的时候，所以它不可能创建一个两阶的插值曲线。此外，P-input 决定所创建曲线的周期性。你在运算器的输出项会看到一些输出模式，C, L 和 D outputs 一般指定曲线的长度和曲线域。

C) Kinky Curves 尖点曲线 (Curve/Spline/Kinky Curve)

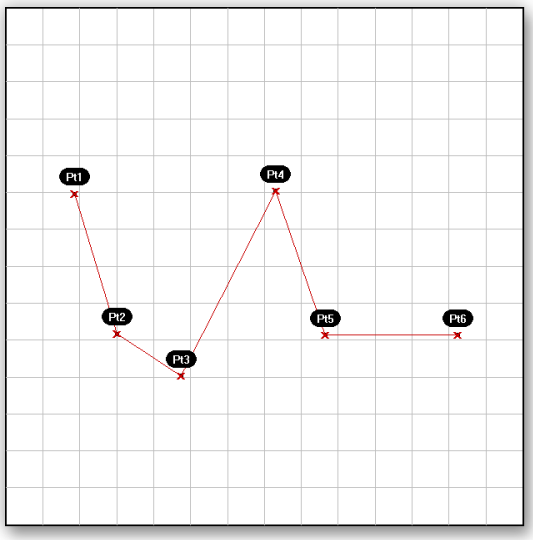
Kinky Curve 其实就是变异的 Interpolated Curves，它的很多属性和 Interpolated Curves 是一样的，唯一的一点不同就是在 KinkyCurve 之中允许控制线转角处的角度阀值，将一个 numeric slider 和 Kinky Curve 运算器的 A 输入项相连，来观察阀值的实时变化。在这里需要注意的是 A 输入项需要 一个 radians （弧度）的输入，在这个例子 里，在 A 输入项中有一个 expression 来将滑竿输入的数值从角度转变为弧度

D) Polyline Curves 复合曲线 (Curve/Spline/Polyline)



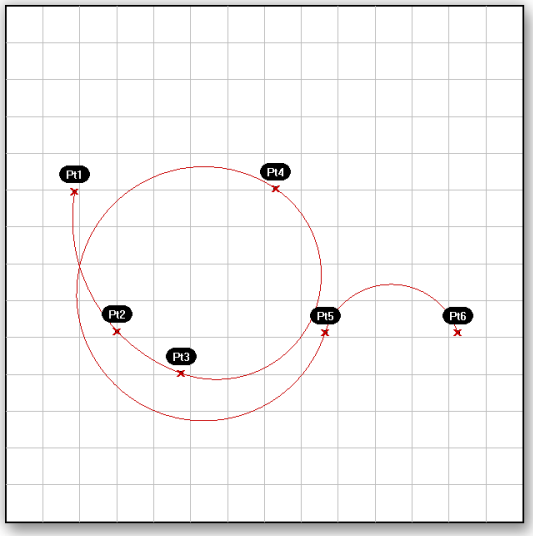
在 Rhino 中 Polyline curve 可能是最平滑的曲线了。这是因为一个 Polyline curve 可以由线部分，多义线部分，角度等于 1 的 NURBS 曲线组成。让我们从 Polyline curve 的背后说起。本质上来讲，Polyline curve 类似于一列点。不同的是我们将多义线中的点看做一个系列，在这些点之间形成连续的线段。就像先前提到的那样——一个角度等于 1 的 NURBS 曲线。由于多曲线一个连接两个或者多个点线段的集合，这就使得它总是经过它的控制点；使得它在某些方面类似于 Interpolated Curve （内插点曲

线）。像上述的曲线类型，Polyline curve 运算器的 V 输入项需 要指定一系列点以确定线的范围。运算器中的 C 输入项定义多义线是否闭合，如果第一个点的位置和最后一个点位置不一致，生成的线将成为一个闭合的圆环。多义线运算器的输出项和前面的例子稍有不同， 在这里，结果只有曲线本身。你可以使用 Grasshopper 里提供的其他曲线分析类型的运算器来获得曲线的其他属性。



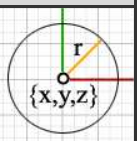
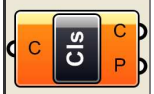
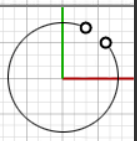
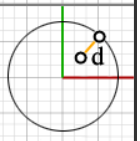
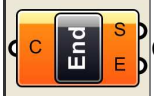
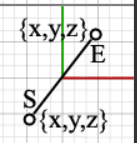
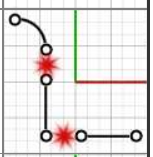
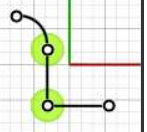
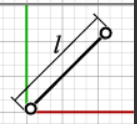
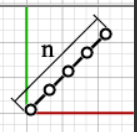
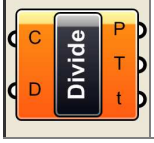
E) Poly Arc 复合弧线 (Curve/Spline/Poly Arc)

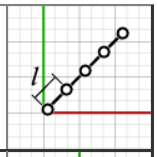
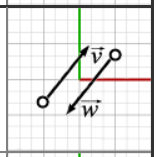
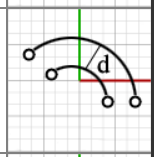
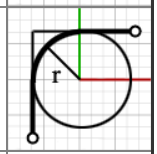
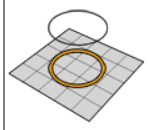
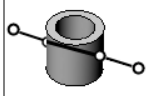
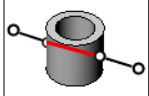
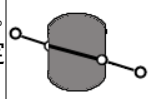
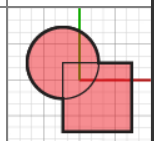
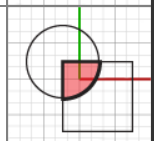
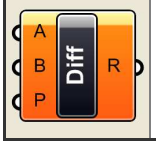
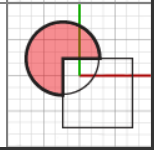
Poly-arc 和上面介绍的 Polyline curve 几乎相同，只是在这里面， 运算器定义用弧线连接所有点而不存在直线，每一个多义弧线都是独一无二的，运算器计算每一个控制点所需要的切点以生成一条无比平滑的曲线，并在每段弧线间的过渡都是连续的。这个运算器中没有其他的输入项，只有最初的一系列点，并且输出项也只有曲线。



10.1 Curve Analytics 曲线分析曲线分析

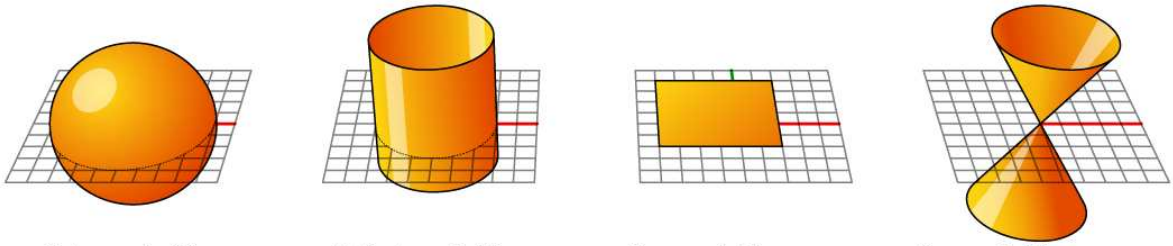
要使用 Grasshopper 里现有的所有可用分析工具编写一个教程实在是很困难，所以在这里我列了一张表格，包含最常用的一些运算器。

Component	Location	Description	Example
	Curve/Analysis/Center	找到圆或者弧的中心点和半径	
	Curve/Analysis/Closed	测试曲线是否是闭合或者具有周期性的	
	Curve/Analysis/Closest Point	找到曲线上距离空白区样本点最近的点	
	Curve/Analysis/End Points	提取曲线上的端点	
	Curve/Analysis/Explode	使曲线的几个组成部分断开	
	Curve/Utility/Join Curves	连接线部分以组成曲线	
	Curve/Analysis/Length	测量曲线或者线段的长度	
	Curve/Division/Divide Curve	将曲线断开成多段等长的部分	
	Curve/Division/Divide Distance	将曲线按照指定的间距断开成多段	

	Curve/Division/Divide Length	将曲线按照指定的间距断开成多段	
	Curve/Utility/**Flip**	使用指定引导曲线转换曲线方向	
	Curve/Utility**/Offset**	依据指定距离偏移曲线	
	Curve/Utility/**Fillet**	用指定的半径将锐角曲线进行倒角	
	Curve/Utility/**Project**	将曲线投影到 Brep 上 (Brep 是一个曲面的集合， 类似于 Rhino 中的 polysurface)	
	Intersect/Region/Split with Brep(s)	用一个或者更多的 Brep 分离曲线	
	Intersect/Region/**Trim with Brep(s)**	用一个或者更多的 Breps 剪切曲线。在 Ci(内部曲线)和 Co(外部曲线)指定方向，确定需要进行剪切的部分	
	Intersect/Region/**Trim with Region(s)**	用一个或者更多的 Region （范围）剪切曲线。Ci(内部曲线)和 Co (外部曲线)指定方向，确定需要进行剪切的部分	
	Intersect/Boolean**/Region Union**	找出两个封闭平面曲线的轮廓线	
	Intersect/Boolean/**Region Intersection**	找出两个封闭平面曲线相交交集的部分	
	Intersect/Boolean/**Region Difference**	找出两个平面曲线差集部分	

11 Surface Types*（曲面类型）

除了一些基本的面的类型如球形，圆锥形，平面形和圆筒形，Rhino 还支持三种自由曲面类型，其中最常用的就是 NURBS 曲面。和曲线一样，所有可能的曲面形状都可以通过 NURBS 曲面的方式进行描绘，这也是 Rhino 的默认面类型。在这里，这种重要的面类型将是我们学习的重点。



Sphere primitive {Plane; Radius}

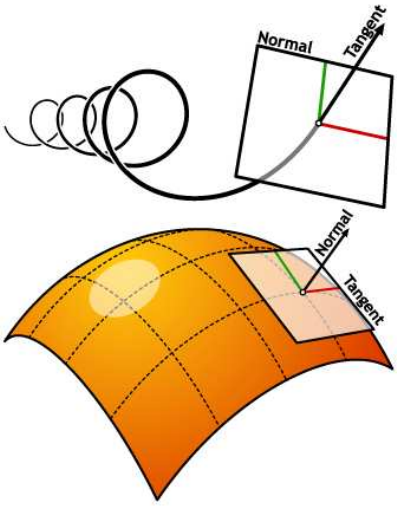
Cylinder primitive {Plane; Radius; Height}

Plane primitive {Plane; Wid

Cone primitive

s

NURBS 曲面和 NURBS 曲线是很相似的。它们的对于形状，法线，切线，曲率和其他一些属性的计算标准是一样的，但是也存在一些明显的不同。例如，曲线有切线方向和法线平面而曲面则有法线方向和切线平面。这就意味着曲线缺乏方位而曲面则缺少方向。这对于所有曲线和曲面 s 类型都是适用的，你不得不去习惯于他们。通常在编写含有曲面或者曲线的代码时，你要去假 s 设这些方位和方向，尽管有的时候是一些错误的假设。



事实上由于 NURBS 表面含有矩形的 U V 曲线网格，所以在 NURBS 表面之中有两个隐含的几
何方向。就算这些方向经常是不定的，我们还是会去适用它们，因为它们使得我们的工作
变得更简单。

Grasshopper 处理 NURBS 表面的方式和 Rhino 是一样的，因为它们产生的面是基于同一种核心算法的。不过，因为 Grasshopper 在 Rhino 视窗上对曲面进行即时的直观显示 （这就是为什么在对结果进行烘焙之前你不能选择 Grasshopper 产生的几何体），这些网格参数给的很低，以保证 Grasshopper 很高的运算速度。你可能会注意到曲面中存在的一 些小的平面，但这仅仅出现在 Grasshopper 的结果显示中，任何烘焙出来的几何体仍然使用高标准的网格参数设定。

* Source: Rhinoscript 101 by David Rutten
http://en.wiki.mcneel.com/default.aspx/McNeel/RhinoScript101

Grasshopper 选择使用两种方式去处理面。首先，就像我们已经讨论过的一样，通过 NURBS 面的方式。一般来说，所有的表面分析运算器都可以用在 NURBS 面上，例如在一个特定的面上求面积或者找到它的曲率。虽然这是一个相当复杂的数学运算，但还是比较容易解决的，因为计算机没有考虑到第三个方向的维度，例如深度或厚度。但如何让 Grasshopper 去描述有三个维度的面？McNeel 公司的开发者将这个难题扔给了我们，并创造了第二种方法：建立一个可以单独控制的物体，就像在 Rhino 界面中那样。使用 Brep 或边界表示。

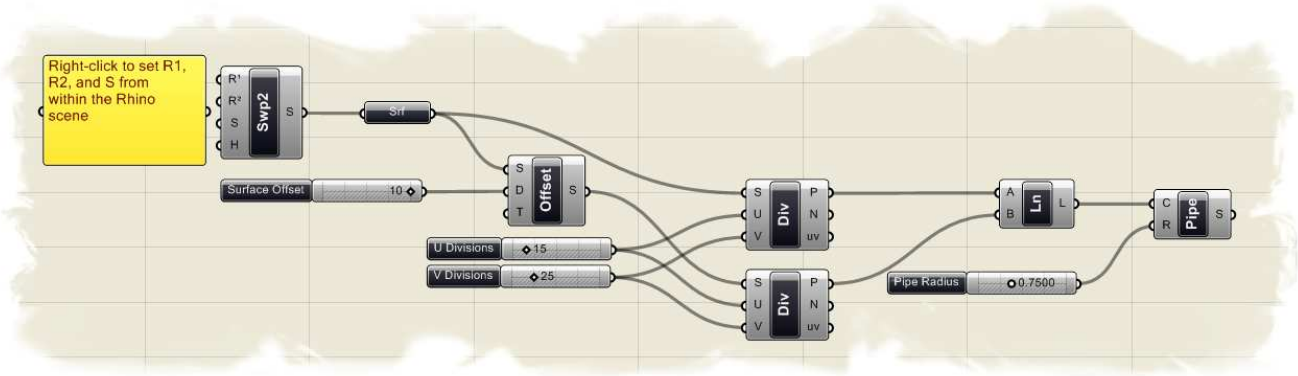
Brep 可以被当做是一个立体的或者类似于 Rhino 中的多边形物体。它还是由众多 NURBS 曲面组成， 但是他们在一起建立一个有厚度的对象，即使理论上 NURBS 曲面是没有厚度的。由于 Brep 基本上是由面构成的，所以一些基本的 NURBS 表面分析运算器还是可以使用的，但还有一些就不可以了。这是因为 Grasshopper 内置的翻译逻辑会尝试将物体转变为所需要的输入形式。如果一个运算器需要一个 Brep 但是你给它一个面，那么这个面将被转换成为一个 Brep。同样的原理也适用于 Numbers 和 Integers, Colors 和 Vectors, Arcs 和 Circles。一下是一些翻译定义，例如：

- Curve → Number (对 Curve 长度赋值)
- Curve → Interval (对 Curve 的域赋值)
- Surface → Interval2D (对 Surface 的 uv 域赋值)
- String → Number (对 string 进行赋值,即使是一个完整的语句)
- Interval2D → Number (对 interval 的 area 赋值)

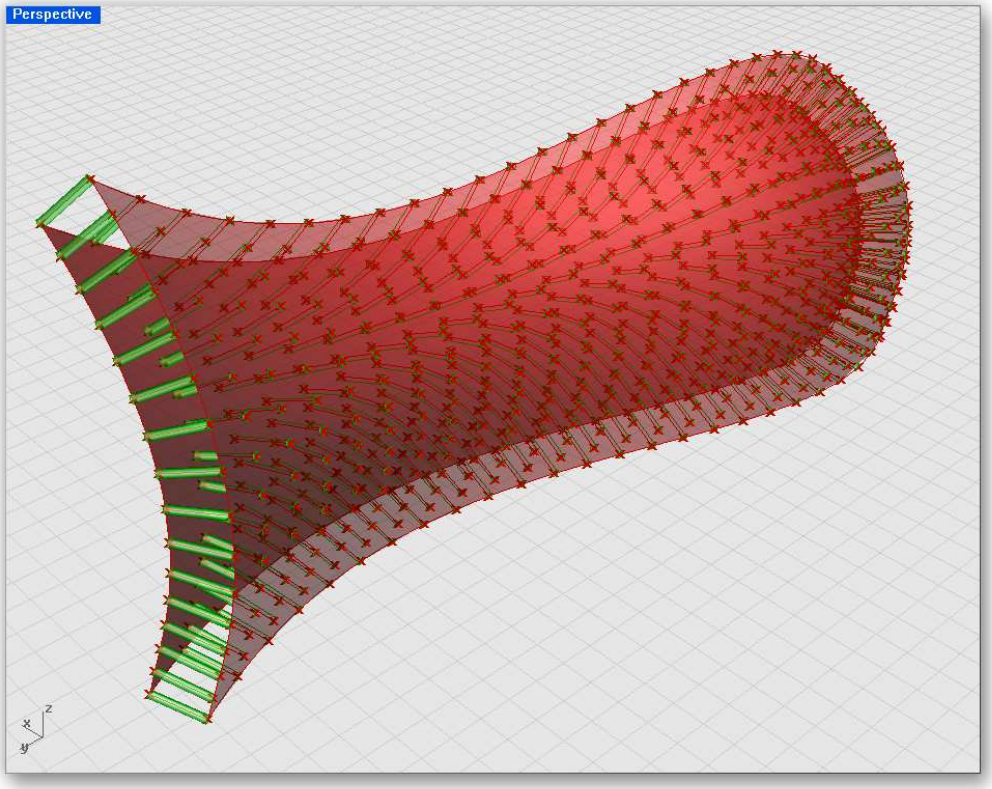
还存在更多的自动转换的翻译逻辑，且将会有更多的数据类型被添加进来，这个表格只会变得更长。现在我们面类型已经有了足够的了解，来看看下面几个不同的例子吧~ ！

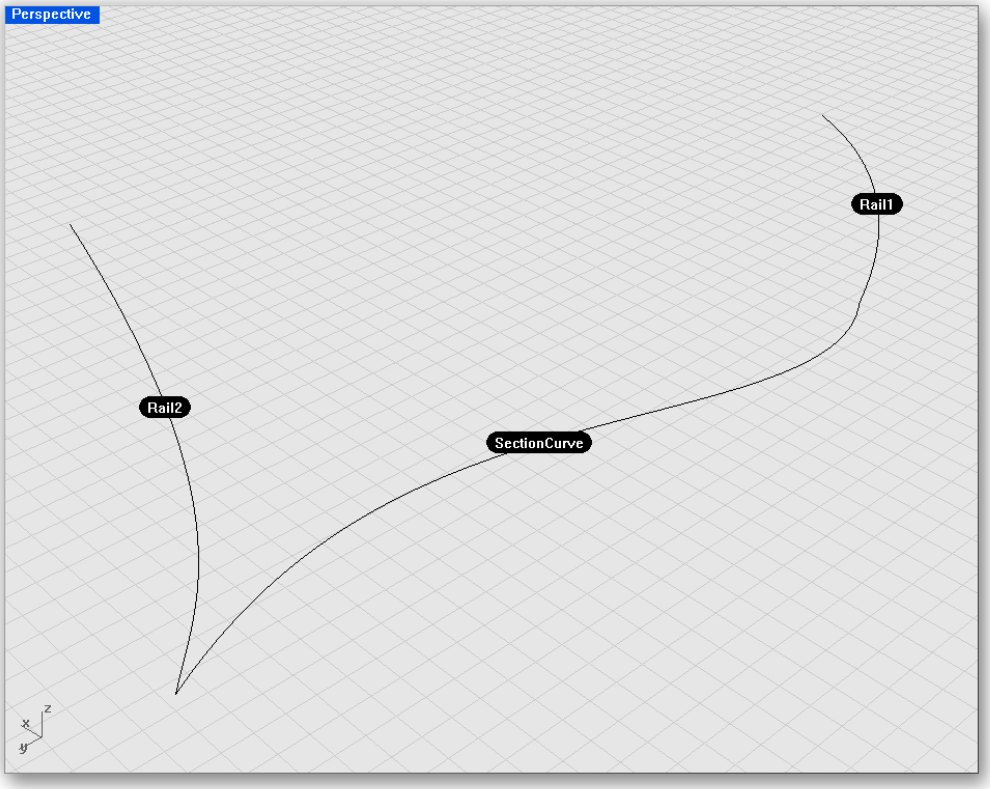
11.1 Surface Connect（面的连接）

这个例子是由 David Fano 在 Design Reform 中发布的，这是一个极好的例子来展示一系列对面进行操纵的技术。在这个例子中我们将用到 Sweep2Rail, Surface Offset, 和 SurfaceDivision 运算器来建立如下的模型。开始干活~，在 Rhino 中打开 Source FileSurfaceConnect.3dm，在这个文件中你可以看到为这个例子提供所需框架的三条曲线（2 条轨道线和 1 个截面线）。



注意:想看最后的成图和连接图，在 Grasshopper 中打开 SurfaceConnect.ghx （文件位于 Source Files 中）





从头开始建立模型:

- Surface/Freeform/Sweep2Rail -拖放一个 Sweep 2 Rails 运算器到工作区
- 右击 Sweep2Rail 运算器的 R1 输入项，并选择 "Set one Curve"
- 选择如上图中的第一条曲线
- 右击 Sweep2Rail 运算器的 R2 输入项，并选择"Set one Curve"
- 选择如上图中的第二条曲线
- 右击 Sweep2Rail 运算器的 S 输入项，并选择"Set one Curve"
- 选择如上图中的 section curve

  如果按照以上步骤选中所有曲线之后，你会看到在三条曲线之间生成一个曲面.
- Surface/Freeform/Offset -拖放一个 Surface Offset 运算器到工作区
- 连接 Sweep2Rail 运算器的 S 输出项到 Offset 运算器的 S 输入项上
- Params/Special/Slider -拖放一个 Numeric Slider 运算器到工作区
- 右击 slider 并按照如下设定：
  - Name: Surface Offset
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 10.0
  - Value: 10.0
- 将 slider 连接到 Surface Offset 运算器的 D 输入项

  你现在将看到一个新的曲面，相对于原来曲面偏移 10 个单位 （或者其他在滑竿上设置的数值）
- Surface/Utility/Divide Surface -拖放两个 Divide Surface 运算器到工作区
- 连接 Sweep2Rail 运算器的 S 输出项到第一个 Divide Surface 运算器的 S 输入项

你将看到一系列的点出现在第一个 Sweep2Rail 形成的曲面上。这是因为 Divide Surface 运算器缺省 UV 值默认为 10。 这样 Divide Surface 运算器在曲面的每个方向上建立 10 个点,最后在曲面上建立一张由点组成的网格。 如果你将将每个点都连接在分割线上，你将得到 “ ISO 曲线”，曲面内在的网格。

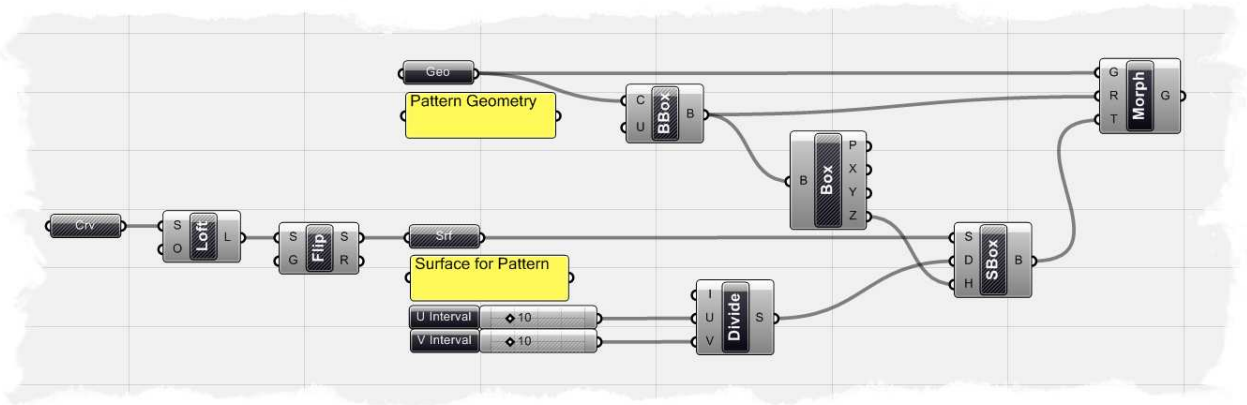
- 连接 Surface Offset 运算器的 S 输出项到另一个 Divide Surface 运算器的 S 输入项
  同样，一系列点出现在偏移后的曲面上。
- Params/Special/Slider -拖放两个 numeric sliders 到工作区
- 右击第一个 slider 并按照如下设定：
  - Name: U Divisions
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 15.0
- 右击第一个 slider 并按照如下设定：
  - Name: V Divisions
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 100.0
  - Value: 25.0
- 连接 U Divisions 滑竿到两个 Divide Surface 运算器的 U 输入项
- 连接 V Divisions 滑竿到两个 Divide Surface 运算器的 V 输入项
  这两个滑竿现在控制着两个曲面相应方向上的分割点数。由于两个曲面有着相同的分割点数，因此每个点都有相同的索引编号，我们可以以简单的线条将里面的曲面和外面的曲面相连接。
- Curve/Primitive/Line -拖放一个 Line 运算器到工作区
- 连接第一个 Divide Surface 运算器的 P 输出项到 Line 运算器的 A 输入项相
- 连接第二个 Divide Surface 运算器的 P 输出项到 Line 运算器的 B 输入项相
  就是这么简单。你现在应该可以看到一系列的线将两个曲面上的每两对应的点连接了起来。我们现在可以对定义进行深化，对线的厚度赋值。
- Surface/Freeform/Pipe -拖放一个 Pipe 运算器到工作区
- 连接 Line 运算器的 L 输出项到 Pipe 运算器的 C 输入项
- Params/Special/Slider -拖放一个 Numeric Slider 到工作区上
- 右击 slider 并按照如下设定：
  - Name: Pipe Radius
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 2.0
  - Value: 0.75
- 连接 Pipe Radius 滑竿到 Pipe 运算器的 R 输入项

11.2 Paneling Tools 镶板工具

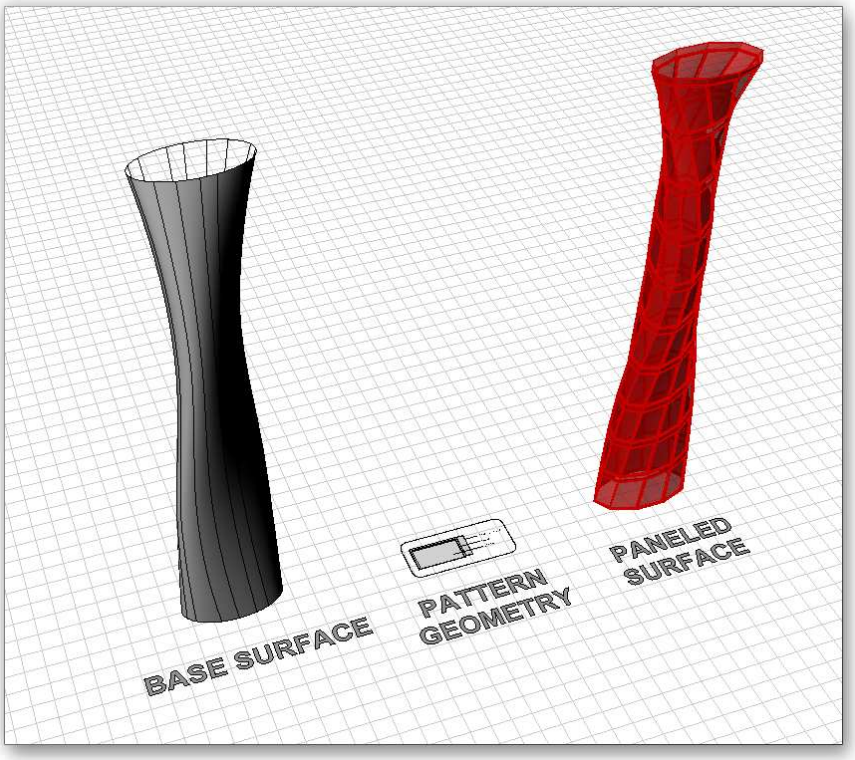
McNeel 最近发布了一个很好的 Rhino 插件程序名字叫 paneling tools。这个插件能够让你把定制的几何体块沿着一个表面阵列。Grasshopper 也有一些运算器 （component）能被用于重新创造类似于 Paneling Tools 的生成方法,在接下来的教程中,我们将介绍如何通过使用 interval 运算器来细分一个表面,我们也会涉及到一些在 GH 中的新的变形运算器。

想了解插件 Paneling Tools 的更多信息，请参阅：
http://en.wiki.mcneel.com/default.aspx/McNeel/PanelingTools.html.

下面是一个把一个几何图形 （例如一个窗户分割框系统）沿着一个塔楼的表面流动的 GH 文件（定义）的截图。



注意：想了解这个实例最终的定义 （一个 Grasshoppper 文件就是一个定义 ），请在 GH 里打开文件 Paneling Tool.ghx,这个文件位于与本文档配套的 Source Files 文件夹。



下面我们从头建立GH的定义，首先我们需要一系列曲线用于放样建立塔楼曲面。在Rhino中，打开位于源文件夹里的 Panel Tool_base.3dm 文件,你会找到四个椭圆,我们将用这四个椭圆用放样的方法定义塔楼曲面,除此之外里面还有一些现成的几何图案供我们使用。

首先我们建立一个放样曲面：

- Params/Geometry/Curve –拖动一个 Curve 运算器到工作区。
- 右击这个 Curve 运算器，然后选择“Set Multiple Curves”
- 然后，根据提示，在 Rhino 视图里从下到上依次选择四个椭圆。
- 点击回车键确认选择。

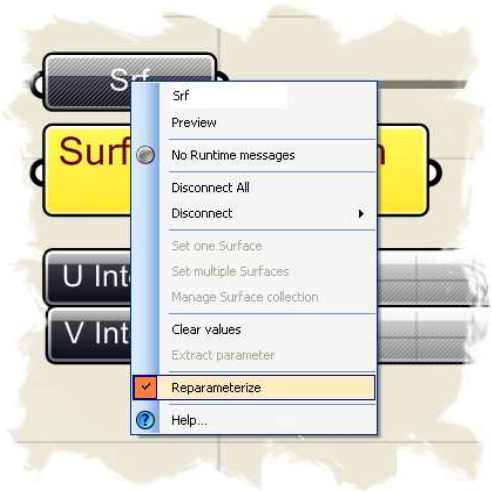
  就像我们前面的几个例子一样，我们已经用 Grasshopper 定义出了一些 Rhino 几何体。
- Surface/Freeform/Loft –拖动一个 Loft 运算器到工作区。
- 连接 Curve 运算器到 Loft 运算器的 L 输入项

  如果你右击 Loft 运算器的 O 输入项，你会找到和 Rhino 很像的常规常规放样选项。在这个实例中，默认的设置就可以，但是以后你可能要调整这些选项以满足你的特殊要求。

* ***非强制性步骤:*** *在GH 的图形界面里几乎不可能看出一个放样曲面的法向是否正确。在这个实例中，我（原作者）注意到我的默认的放样曲面恰恰是面向里面的,这导致所有的嵌板都是朝里的。这时，我们可以用 Flip 运算器来反转放样曲面的法线，如果你在自己的 定义中正常的话，你可以删除这个 Flip 运算器，然后重新连接其他运算器。*

- Surface/Utility/Flip –拖动一个 Flip 运算器到工作区
- 连接 Loft 运算器的 L 输出项和 Flip 运算器的 S 输入项
- Params/Geometry/Surface –拖动一个 Surface 运算器到工作区
- 连接 Flip 运算器的 S 输出项和 Surface 运算器
- 在 Surface 运算器上右击然后选中 Reparameterize 左边的复选框 （如图）

  通常，我们的曲面有一个从零（代表曲面的最底部）到代表曲面顶点的数字的间隔，我们并不关心这个代表曲面的上限的数字具体是多少，因为我们可以重新参数化（Reparameterize）这个曲面。这个选项意味，我们将重 新设置这个间隔 （上限）。这个步骤很关键，否则我们的表面将不会准确的细分。

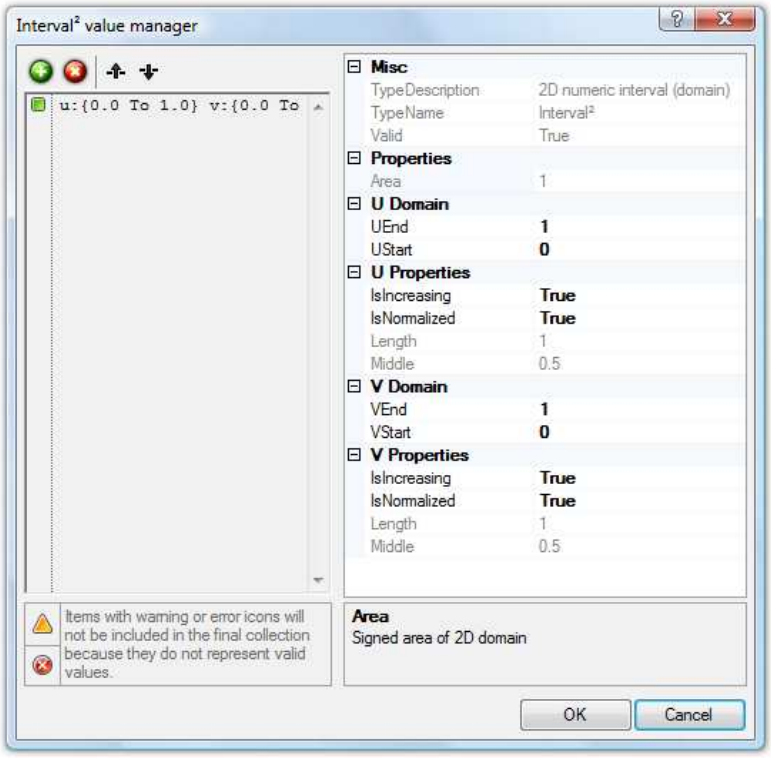


- Scalar/Interval/Divide Interval2 –拖动一个 Divide Two Dimensional Interval 运算器到工作区。

  我们需要在细分表面之前设定间隔范围。
- 右击 Divide Interval 的 I 输入项并选择"Manage Interval Collection"
- 点击绿色加号按键添加一个区间到集合中

默认的区间设置为u:{0.0 To 0.0} v:{0.0 To 0.0}，但是我们需要的U、V 方向的区间范围是 0 to 1

- 将 U-End 的值修改为 1.0
- 将 U-End 的值修改为 1.0

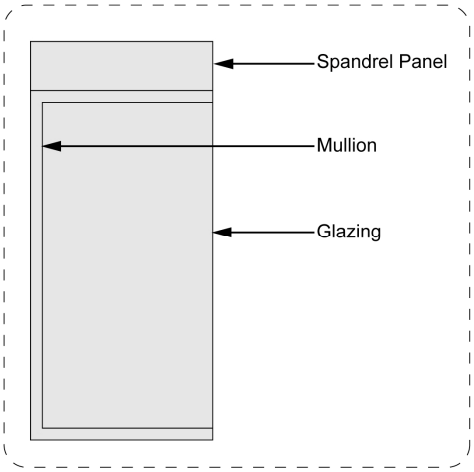


你们的Interval Collection Manager 设置应该如上图所示。点击OK 确定。如果你把鼠标悬停在 Divide Interval 的I 输入项，你会发现U 和V 方向的基础间隔已经和 重新参数化的曲面相适应。

- Params/Special/Slider -拖动两个 slider 到工作区
- 右击第一个 slider，然后进行如下设置：
  - Name: U Interval
  - Slider Type: Integers
  - Lower Limit: 5.0
  - Upper Limit: 30.0
  - Value: 10.0

- 右击第一个 slider，然后进行如下设置：
  - Name: V Interval
  - Slider Type: Integers
  - Lower Limit: 5.0
  - Upper Limit: 30.0
  - Value: 10.0
- 连接名称为 U Interval 的 slider 到 Divide Interval 运算器的 U 输入项
- 连接名称为 V Interval 的 slider 到 Divide Interval 运算器的 V 输入项

- Xform/Morph/Surface Box -拖动一个 Surface Box 运算器到工作区
- 连接 Surface 运算器输出项到 Surface Box 运算器的 S 输入项
- 连接 Divide Interval 运算器的 S 输出项到 Surface Box 运算器的 D 输入项
- 分别在 Curve, Loft, and Surface 运算器上右击，关闭他们的“Preview”



*现在，通过两个 slider 的控制，我们已经把这个曲面细分为 100 个区域。过程是这样的：首先，我们创建了一个和此曲面相适应的间隔范围（二者相等）；然后，将这的间隔分别在 U 方向和 V 方向细分十次；就这样（10*10），最终在曲面上形成了 100 个区域。你可以改变 U 和 V 的值（通过 slider 控制）来控制曲面上形成的区域的数量。下面，让我们 回头创建一个用于放在每一个细分区域的几 何图样。在右边这个场景中，你会看到一个 由托梁嵌板、竖挺、玻璃嵌板组成的窗户系统（类似）。下面我们将用这个系统沿着这个曲面的表面创建一个建筑表皮系统。*

- Params/Geometry/Geometry -拖动一个 Geometry 运算器到工作区
- 在 Geometry 运算器上右击，然后选择"Set Multiple Geometries"
- 根据提示，在 Rhino 窗口中选择 spandrel panel、mullion 和 glazing panel
- 点击回车键确认选择
- Surface/Primitive/Bounding Box -拖动一个 Bounding Box 运算器到工作区
- 连接 Geometry 运算器输出项到 Bounding Box 运算器的 C 输入项

我们选择使用 Bounding Box 运算器有两个原因。首先，Bounding Box 运算器可以决定用于创建曲面的集合图案的厚度。例子中我们使用规则的矩形盒子作为要使用的图案，那么这个厚度很容易确定。但是假如你使用一个比较有机的形状，那么这个厚度问题就比难以解决。通过这个运算器的使用，就可以直接将这个厚度信息传递给 Surface Box 运算器。其次， 我们还可以使用 Bounding Box 作为 BoxMorph 运算器的一个参考 Brep 。下面我们来介绍 BoxMorph 运算器的使用。

- 右击 Bounding Box 运算器的 U 输入项并将 boolean 设置为 True

这一步很重要，这样设置以后 bounding box 运算器会将所有 3 Brep bjects 转化为另一个 box。

- Surface/Analysis/Box Components -拖动一个 Box Components 运算器到工作区
- 连接 Bounding Box-B 输出项项到 Box Components 运算器的 B 输入项
- 连接 Box Components 运算器的 Z 输出项到 Surface Box 运算器的 H 输入项

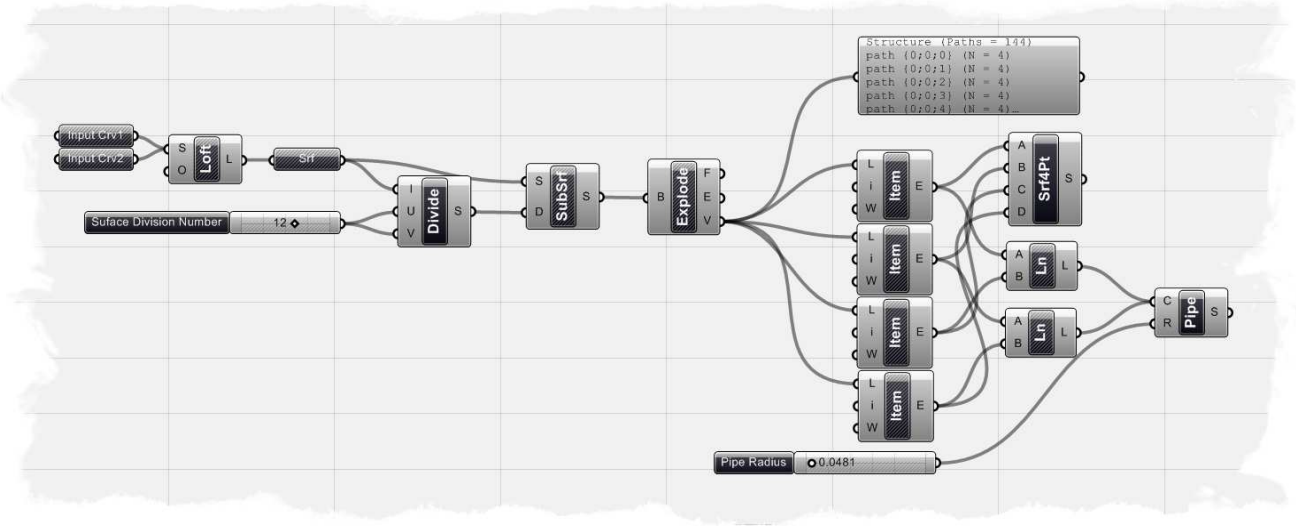
我们就快完成了。

- Xform/Morph/Box Morph - 拖动一个 Box Morph 运算器到工作区
- 连接 Pattern Geometry 输出项到 Box Morph 运算器的 G 输入项
- 连接 Bounding Box 运算器的 B 输出项到 Box Morph 运算器的 R 输入项
- 连接 Surface Box 运算器的 B 输出项到 Box Morph 运算器的 T 输入项
- 右击 Morph Box 运算器并将 data matching 设置为 Cross Reference
- 右击 Surface Box 运算器，然后关闭“Preview”

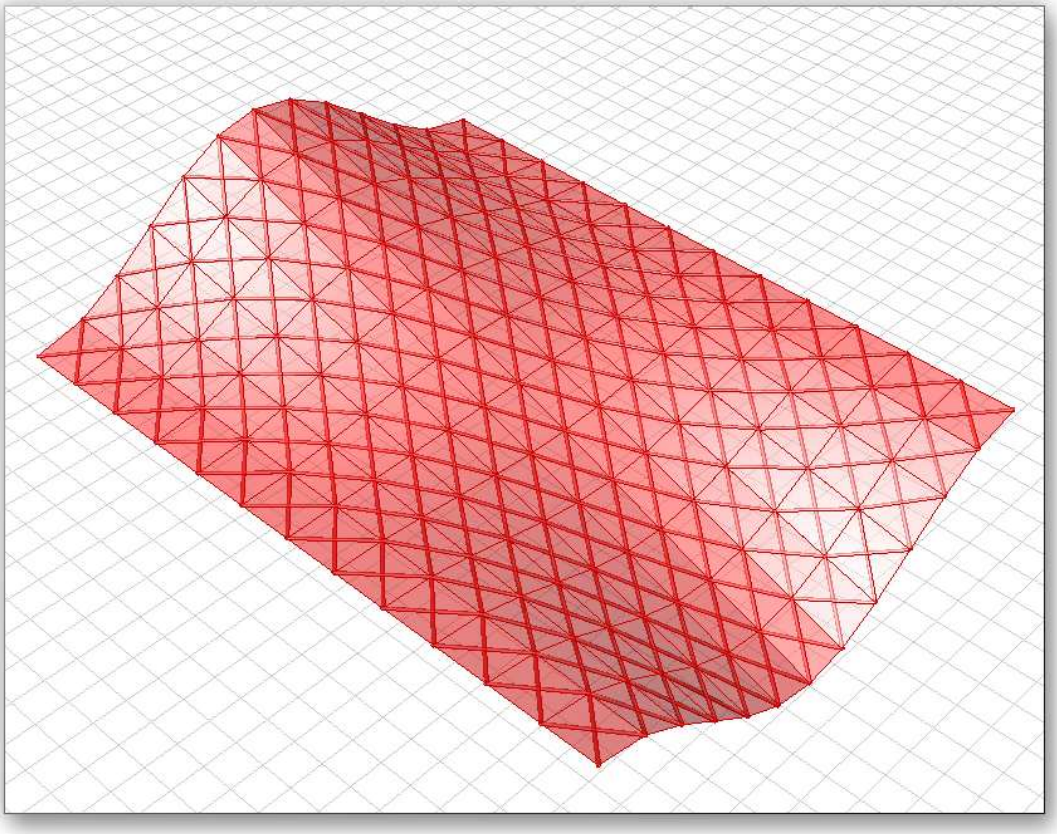
解释一下最后一部分：我们把这个几何图形连接到 Morph Box 运算器，这个几何图形将会被复制到每一个细分区域；我们使用代表 Windows 系统的 Bounding Box 作为参考几何图形；最后， 我们输入 100 个盒子形状的细分区域作为目标盒子的位置来复制几何图形。按照以上步骤，你现在你可以改变：曲面、几何图案以及 U 方向和 V 方向的细分数量，来达到：使用任意图案在任意表面上创建任意细分的建筑表皮系统的目的。

11.3 Surface Diagrid 表面构架

在上个实例中我们已经展示了如何使用类似 Paneling Tools 的 GH 的 defiinition 来建立建筑表皮，但是这个实例将真正的展示如何通过控制数据流来通过任意曲面创建一个菱形网格结构。第一步，先在 Rhino 里打开 Surface Diagrid.3dm 文件。在这个场景里，你会看到两个对称的余弦曲线，属于我们将用到的放样曲面所用到的两个对称的余弦曲线边缘。



Note: 想了解这个实例最终的定义,请在 GH 里打开文件 Surface Diagrid.ghx,这个文件位于与本文档配套的 Source Files 文件夹。



让我们从头建立这个实例的定义文件：

- Params/Geometry/Curve –拖动两个 Curve 运算器到工作区
- 右击第一个 Curve 运算器，然后重命名为“Input Crv1”
- 右击 Input Crv1 运算器，然后选择“Set One Curve”
- 根据提示，选择 Rhino 场景里的一条曲线
- 右击第二个 Curve 运算器，然后重命名为“Input Crv2”
- 右击 Input Crv2 运算器，然后选择“Set One Curve”
- 根据提示，选择另一条曲线
- Surface/Freeform/Loft –拖动一个 Loft 运算器到工作区
- 连接 Input Crv1 到 Loft 运算器的 S 输入项
- 按住 Shift 键不放，连接 Input Crv2 component 到 Loft 运算器的 S 输入项

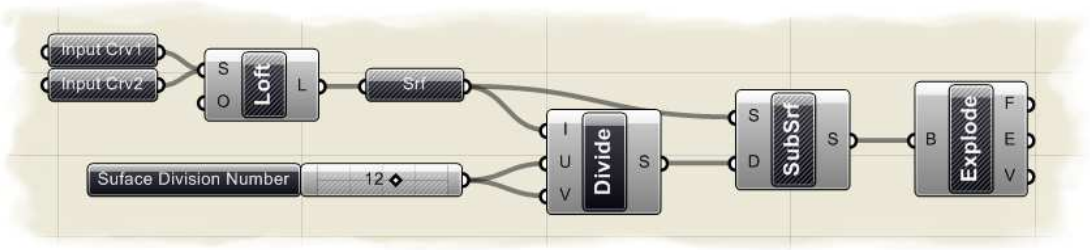
  你会看到在 Rhino 场景中，两条曲线之间生成了一个放样曲面。
- Params/Geometry/Surface –拖动一个 Surface 运算器到工作区
- 连接 Loft 运算器的 L 输出项到 Surface 运算器输入项
- Scalar/Interval/ Divide Interval2 –拖动一个 Divide Two Dimensional Interval 运算器到工作区

  和上一个实例中一样，我们将细分这个曲面为更小的曲面区域。为此，需要建立一个 U 和 V 方向的间隔，这个间隔将控制我们的细分表面区域
- 连接 Surface 运算器的输出项到 Divide Interval 运算器的 I 输入项
- Params/Special/Numeric Slider –拖动一个 slider 到工作区
- 右击 slider 上，然后进行如下设置：

  - Name: Surface Division Number
  - Slider Type: Integers
  - Lower Limit: 0.0
  - Upper Limit: 20.0
  - Value: 12.0
- 连接这个 Slider 的输出项到 Divide Interval 运算器的 U 和 V 输入项
- Surface/Utility/Isotrim –拖动一个 Isotrim 运算器到工作区
- 连接 Surface 运算器的输出项到 Isotrim 运算器的 S 输入项
- 连接 Divide Interval 运算器的 S 输出项到 Isotrim 运算器的 D 输入项
- 分别在 Loft 和 Surface 运算器上右击，关闭他们的预览显示

  现在，有会得到一系列的细分表面，这些表面是和你设置在 slider 的数值相适应的。因为我们是将一个 slider 连接到 U 和 V 输入项，所以你会看到，当你向左或者向右移动 slider 数值时，U 和 V 两个方向的细分结果一起变化。如果你想单独控制 U 和 V 方向的细分值，你可以增加一个 slider。
- Surface/Analysis/Brep Components –拖动一个 Brep 运算器到工作区

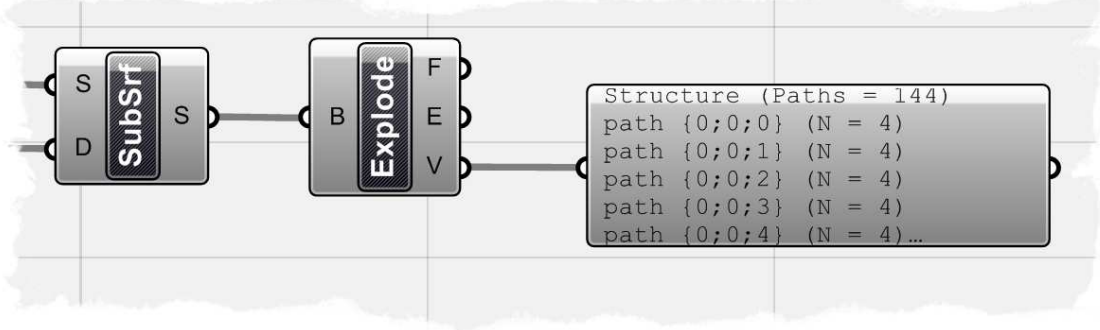
  这个 Brep 运算器能够把一个或者一系列 Brep 的相关元素，例如面、边缘、顶点等，提取出来使用。在这个实例中，我们想知道一个角点的位置，以便利用这些角点在每个细分区域对这些角点进行对角线连接
- 连接 Isotrim 运算器的 S 输出项到 Brep 运算器的 B 输入项



在我们的定义中，到现在为止，看起来是上面图片所显示的样子。我们已经把这个曲面细分为更小的曲面区域，并且得到了每个区域曲面面的角点。接下来，我们要为这些角点编号，以便创立一个菱形网格结构系统。

- Param/Special/Parameter Viewer -拖放一个 Parameter Viewer 运算器到工作区
- 连接 Brep 运算器的 V 输出项到 Parameter Viewer 的输出项

  这个 Parameter Viewer 运算器将会帮助我们检查我们的 tree structure。在这个例子中的 structure 共有 144 path，每个分支的末端都有 4 个数据录入（在我们这个例子中，这些数据是每个细分表面的 4 个角点）。我们可以用 List Item 运算器进入到每个 path 或 sub-path 中并检索出具体的数据录入。为了创建这个构架，我们首先给知道每个细分表面的第一个、第二个、第三个、第四个，角点在笛卡尔坐标系中的位置。



- Logic/List/List Item -拖放 4 个 List Item 运算器到工作区
- 连接 Brep 运算器的 V 输出项到全部 4 个 List Item 运算器的 L 输入项
- 右击第一个 List Item 的 i 输入项并设置 Integer 为 0
- 右击第一个 List Item 的 i 输入项并设置 Integer 为 1
- 右击第一个 List Item 的 i 输入项并设置 Integer 为 2
- 右击第一个 List Item 的 i 输入项并设置 Integer 为 3

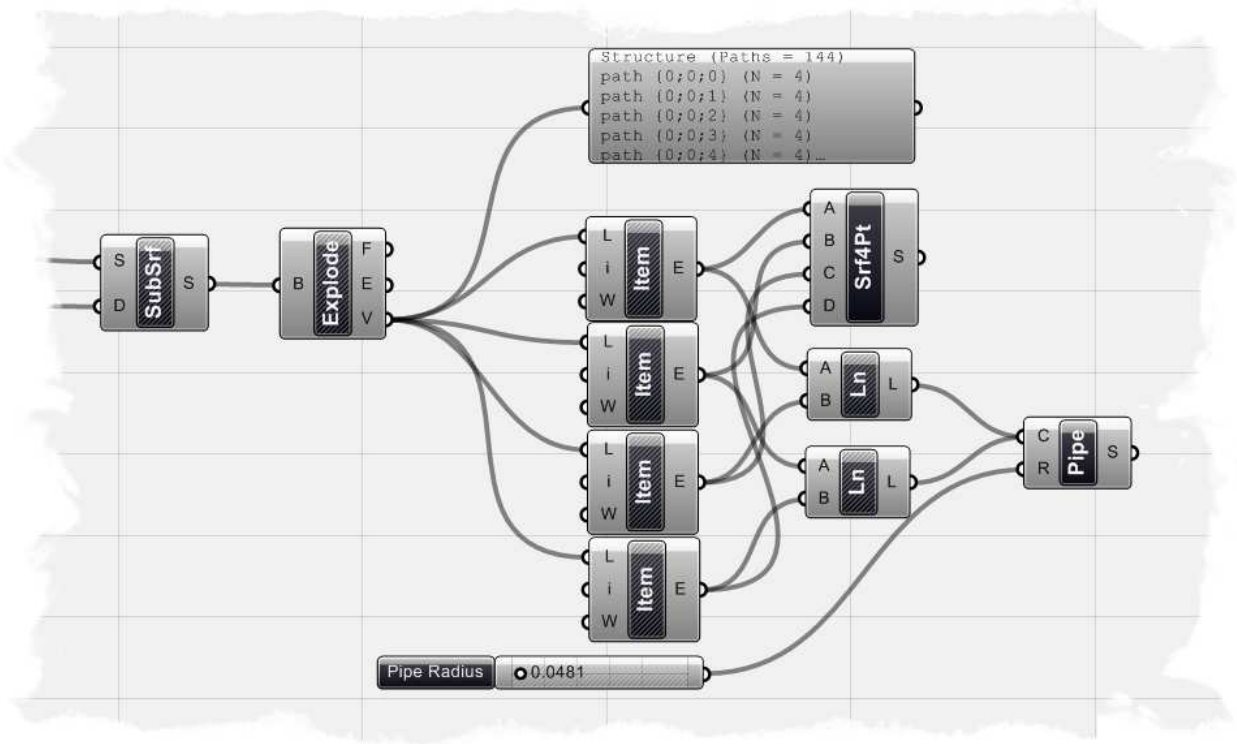
  大家可能会注意到 List Item 运算器的指数每次增加一个单位。因为我们的 tree structure 已经建立并且 Grasshopper 明白我们一共有 144 不同的 path，每个 path 都包含 4 项。。。我们要做的就是从每个 path 中检索出第一项，第二项，直到第四项。所以 4 个 List Item 运算器定义了全部细分表面的每个角点，并且我们已经将他们区分开来，现在可以准确的连接出每个对角线。
- Curve/Primitive/Line -拖放两个 Line 运算器到工作区
- 连接第一个 List Item 运算器的 E 输出项到第一个 Line 运算器的 A 输入项
- 连接第三个 List Item 运算器的 E 输出项到第一个 Line 运算器的 B 输入项
- 连接第二个 List Item 运算器的 E 输出项到第二个 Line 运算器的 A 输入项
- 连接第四个 List Item 运算器的 E 输出项到第二个 Line 运算器的 B 输入项

  这时候，你应该看到表面上生成一系列的线，这些线表示了构架的结构。
- Surface/Freeform/Pipe -拖动一个 Pipe 运算器到工作区
- 连接第一个 Line-L 输出项到 Pipe-C 输入项
- 按住 Shift 键不放，连接的二个 Line-L 输出项到 Pipe-C 输入项

  下面添加一个 slider 来控制 pipe 的半径，从而得到一个可以控制的结构厚度。
- Params/Special/Number Slider –拖动一个 slider 到工作区
- 右击 slider，然后进行如下设置：
  - Name: Pipe Radius
  - Slider Type: Floating Point
  - Lower Limit: 0.0
  - Upper Limit: 1.0
  - Value: 0.05
- 连接名字为 Pipe Radius 的 slider 到 Pipe-R 输入项

  最后，我们再利用每四个角点创建一系列平面，用以取代原来的都是曲面的子面域。
- Surface/Freeform/4Point Surface –拖动一个 4 Point Surface 运算器到工作区
- 连接第一个 List Item-E 输出项到 Point Surface-A 输入项
- 连接第二个 List Item-E 输出项到 Point Surface-B 输入项
- 连接第三个 List Item-E 输出项到 Point Surface-C 输入项
- 连接第四个 List Item-E 输出项到 Point Surface-D 输入项

  请确认除 4 Point Surface 运算器和 Pipe 运算器以外所有运算器的 Preview 选项都已关闭。

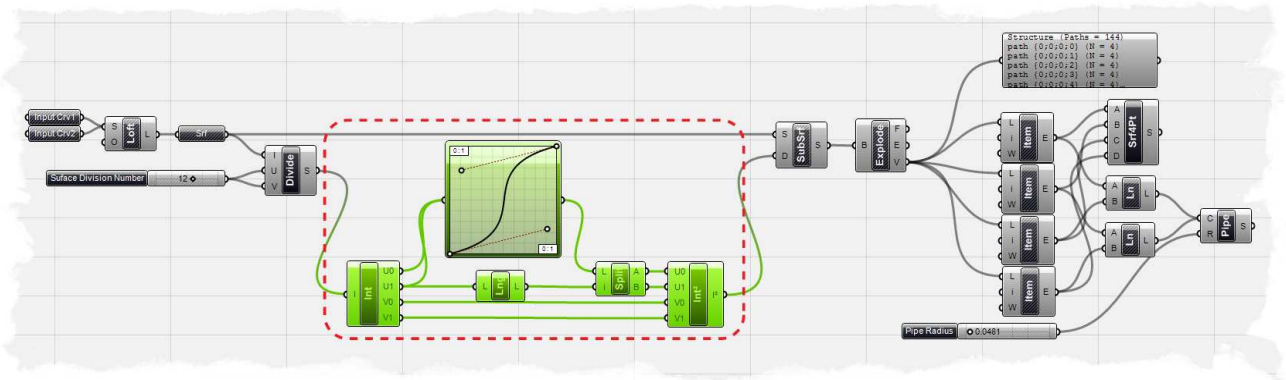


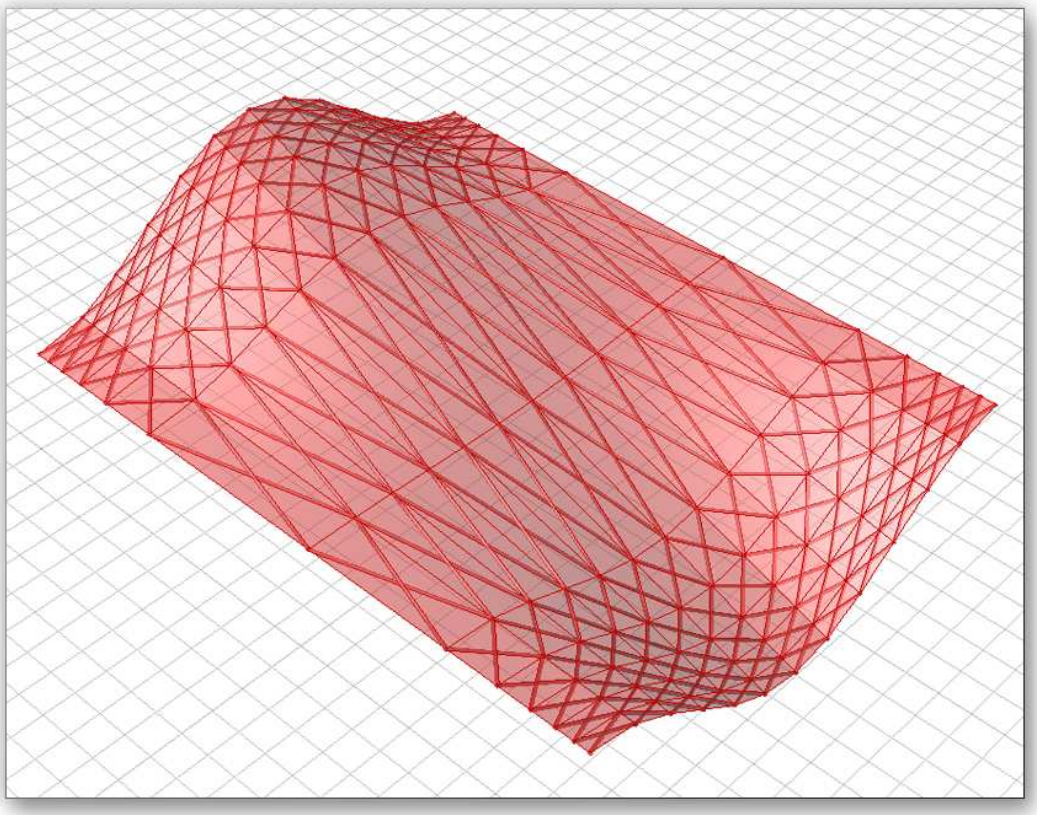
最终的定义文件应该看起来和上面的图片差不多。这个定义适用于任意的曲面，你可以试试用更复杂的曲面来取代这个实例中的放样曲面。

11.4 Uneven Surface Diagrid 不规则表面构架

在上一个例子中，我们演示了如何细分一个表面产生间隔均匀的构架结构。构架之所以是均匀的，因为我们使用的区间是均匀的…但是我们可以使用一些型的运算器和 Graph Mapper 来控制区间和构架的间隔。这个教程将建立在前一个教程的基础上，所以我现在假设你们已经建立好均匀间隔构架的定义。下面是不规则间隔构架最终的定义截图，并且我们需要添加的运算器都已用绿色表示了出来。

注意：想看这个例子最终的定义，请在 Grasshopper 打开 Source Files 中的 Uneven Surface Diagrid.ghx



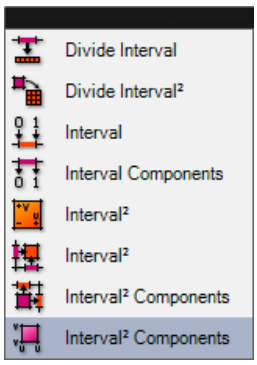


虽然在上一个定义中，我们连接了 Divide Interval 运算器到 Isotrim 运算器。但我们的定义将以断开他们作为开始。

- 右击 Isotrim-D 输入项并选择"Disconnect All"

  因为我要插入一些运算器，所以我们最好将所以 Isotrim 以后的运算器都向工作区的右边移动一段距离。

- Scalar/Interval/Interval Components -拖放一个 Interval Components 运算器（将区间划分了 4 个 number）到工作区



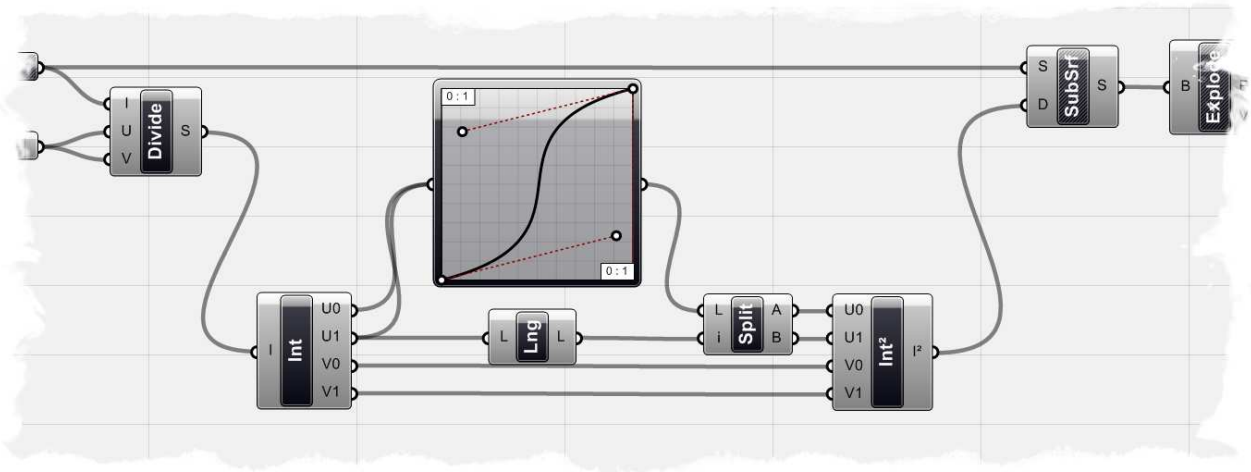
- 连接 Divide Interval-S 输出项到 Interval Components-I 输入项
- Params/Special/Graph Mapper -拖放一个 Graph Mapper 运算器到工作区
- 连接 U0 输出项到 Graph Mapper 输入项
- 按住 shift k 键，连接 U1 输出项到 Graph Mapper 输入项
- 右击 Graph Mapper 并选择一个 graph 类型

  我认为 bezier 图表在这种情况下是最好的选择。你可以通过改变 Graph Mapper 运算器中的 bezier 控制扳手来改变曲线图。你可能会问自己这么做的目的。其实，基本上我们所做的是将均匀的间隔区间分解为每一个可以存取的 U、V 值的列表。我们已经将 U 数据列表反馈到 Graph Mapper 中，并通过一个特定的表格图形来调整它。最终我们将这些调整过的数据重新输入到一个新的区间并反馈到 Isotrim 运算器中。

- Logic/List/List Length -拖放一个 List Length 运算器到工作区
- 连接 U1 输出项到 Length-L 输入项
- Logic/List/Split List – 拖放一个 Split List 运算器到工作区
- 连接 Graph Mappe 的输出项到 Split List 运算器的 L 输入项
- 连接 List Length-L 输出项到 Split List-i 输入项
- Scalar/Interval/Interval 2d – 拖放一个 two dimensional Interval 运算器到工作区
- 连接 Split List-A 输出项到 two dimensional Interva 运算器的 U0 输入项
- 连接 Split List-B 输出项到 two dimensional Interval 运算器的 U1 输入项
- 分别连接 decomposed interval 的 V0 和 V1 输出项到 two dimensional Interval 运算器的 V0 和 V1 输入项

  因为我们将 2 组数据列表连接到 Graph Mapper 的输入项，所以我需要把他们分为两个部分，以保证我们可以我们可以正确的将数据列表反馈回 two dimensional interval 中。我们已把所有的 U 值连接到 Graph Mapper，所以我们 U 方向的细分将由 bezier 曲线控制。因为我们将 decomposed interval 的 V0 和 V1 直接链接到 two dimensional interval 的 V0 和 V1，所以 V 方向上不会发生任何变化。但是，你同样可以按照相同的步骤连接另一个 Graph Mapper 运算器到 v0 和 v1 列表，用同样的方法控制 v 方向的细分。现在，我们所需要做的就是将最新建立的区间连接到 Isotrim 运算器中。

- 连接 two dimensional Interval-I2 输出项到 Isotrim-D 输入项
  即使最后一部分的定义是一样的，图表类型将控制间隔的细分和构架。你可以通过调整 bezier 曲线来提高支撑表面的结构的数量支撑表面。你定义的中间部分应该如下图所示。



12 An Introduction to Scripting（脚本编写入门）

可以通过使用编写脚本的组件或者通过使用 VB DotNET、C# 等程序语言去写代码来扩展 Grasshopper 的功能。以后可能还会有更多语言得以应用进来。用户代码被内置于一个动态生成的阶级模板，随后被编译成一个以 DotNET 框架传送且使用 CLR 编译器的集合。这个集合仅存在于电脑内存中，直到退出 Rhino 后才会被卸载。

Grasshopper 里面的脚本组件可以访问到 Rhino DotNET SDK 的阶级，这些就是插件开发人员用来开发 Rhino 的插件的功能。事实上，Grasshopper 就完全作为一个 DotNET 插件来编写的 Rhino 插件，使用非常类似 SDK 方式进入这些脚本组件的。

但是什么时候我们开始使用这些脚本组件呢？事实上，你可能从来不需要去使用他，但是有些场合你也用得上他。当你想得到一个 Grasshopper 组件所没有的功能时你就会用上他。或者当你编写一个使用递归功能的可生产系统，例如分形。

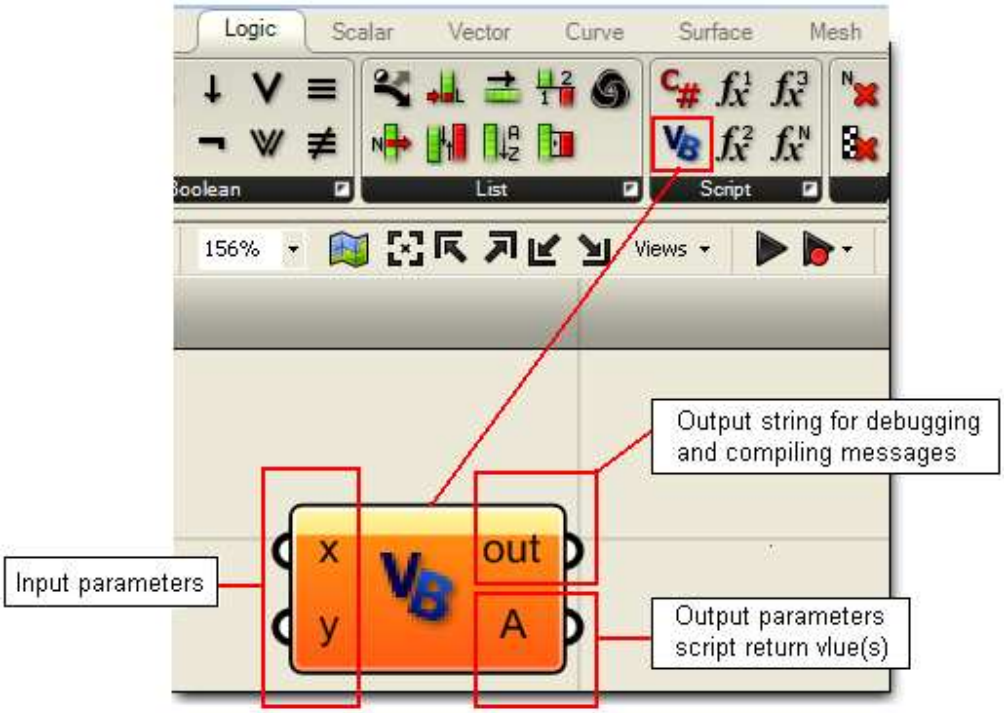
这本入门小册子介绍了一个关于怎样在 Grasshopper 中通过 VB DotNET 编程语言使用脚本组件的总体概况。它包括三个部分。第一部分是关于脚本组件的界面。第二部分包括一个 VB DotNET 语言的简单回顾。第三部分讨论了 Rhino DotNET SDK，几何进阶和实用功能。最后有一个列表，告诉你能从哪里得到进一步的帮助的信息。

13 The Scripting Interface（脚本界面）

13.1 哪里寻找脚本组件

VB DotNet 脚本组件可以在 Logic 板块被找到。现在有两个脚本组件。一个用来写 Visual Basic 代码，另一个用于写 C#代码。毋庸置疑以后将会支持更多的脚本语言。

拖拽组件按钮并且在工作界面释放，就可以将一个脚本组件添加到工作界面了。



默认的脚本组件有两个输入数据以及两个输出数据。用户可以改变他们的名字、类型与数值。

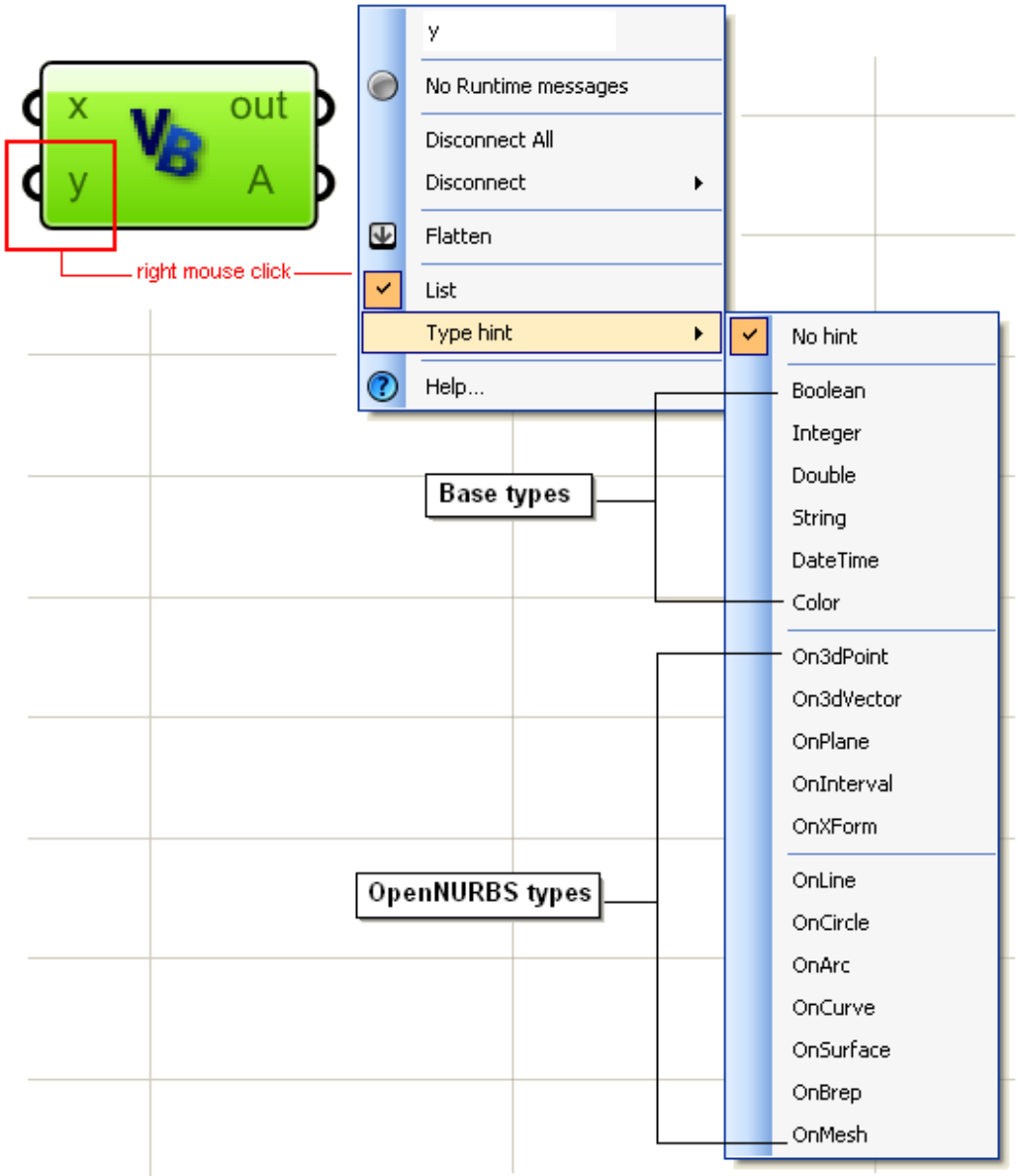
- **X**: 一般类型的第一个输入数据（客体）
- **Y**: 一般类型的第二个输入数据（客体）
- **Out**: 汇集信息后的一系列输出数据
- **A**: 客体类型反馈回来的输出类型

13.2 输入参数

默认情况下，有两个输入参数：x 和 y。我们可以编辑参数的名字，删除或者增加参数，还可以制定参数类型。如果你鼠标右键点击任何一个输入参数，你将看到以下的一个菜单：

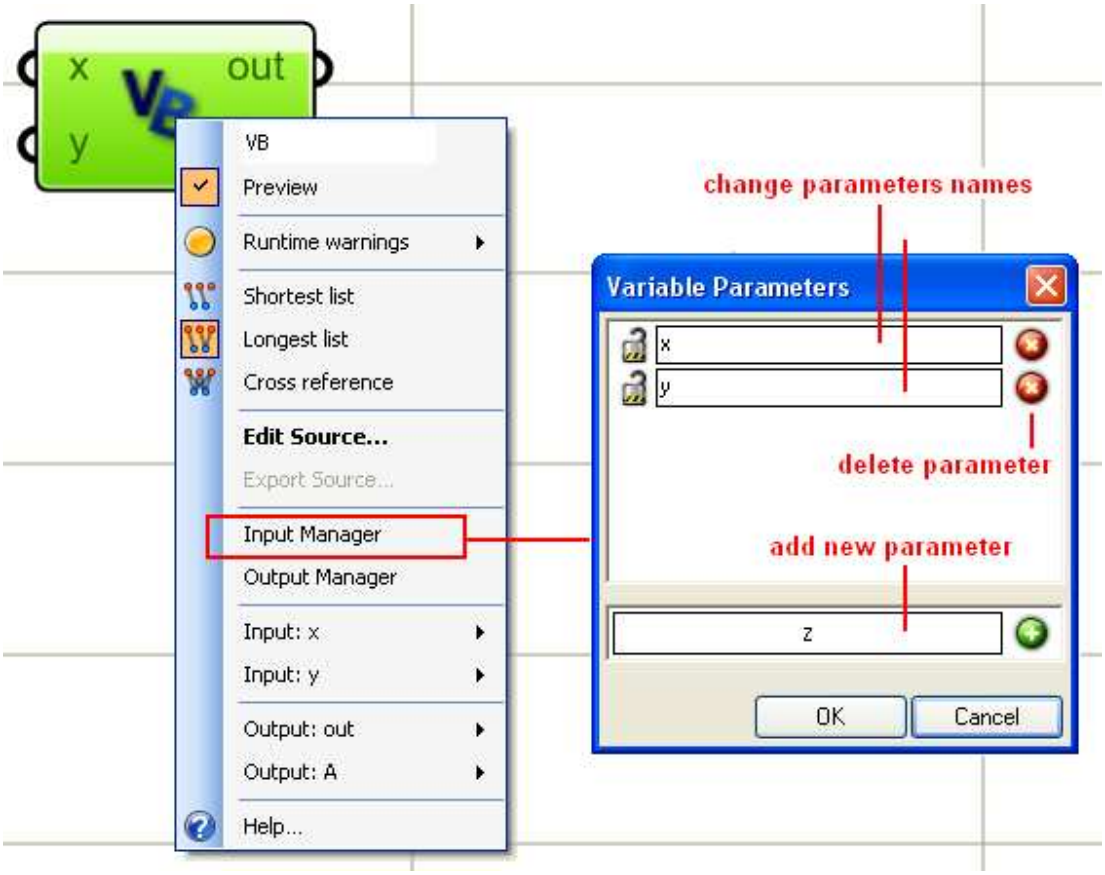
- **Parameter Name(参数名字)**: 你可以点击它并且输入新的名字。
- **Run Time Message(运行时间信息)**: 错误和警告信息。

- **Disconnect** and **Disconnect All(断开连接和断开所有连接)**：与其他 Grasshopper 组件一样的作用。
- **Flatten (变平)**：使数据变平。为了避免数据清单出现巢状，它能将这样的数据转变成各个元素的简单排列。
- **List(数列)**：指示出输入指数能否是一系列数据。
- **Type Hint(类型建议)**：输入参数被默认设定为一般的类型“客体”。最好指定一个类型让这些代码可读性更强，更有效。 以“On”开头的类型是开放 NURBS 曲线类型。



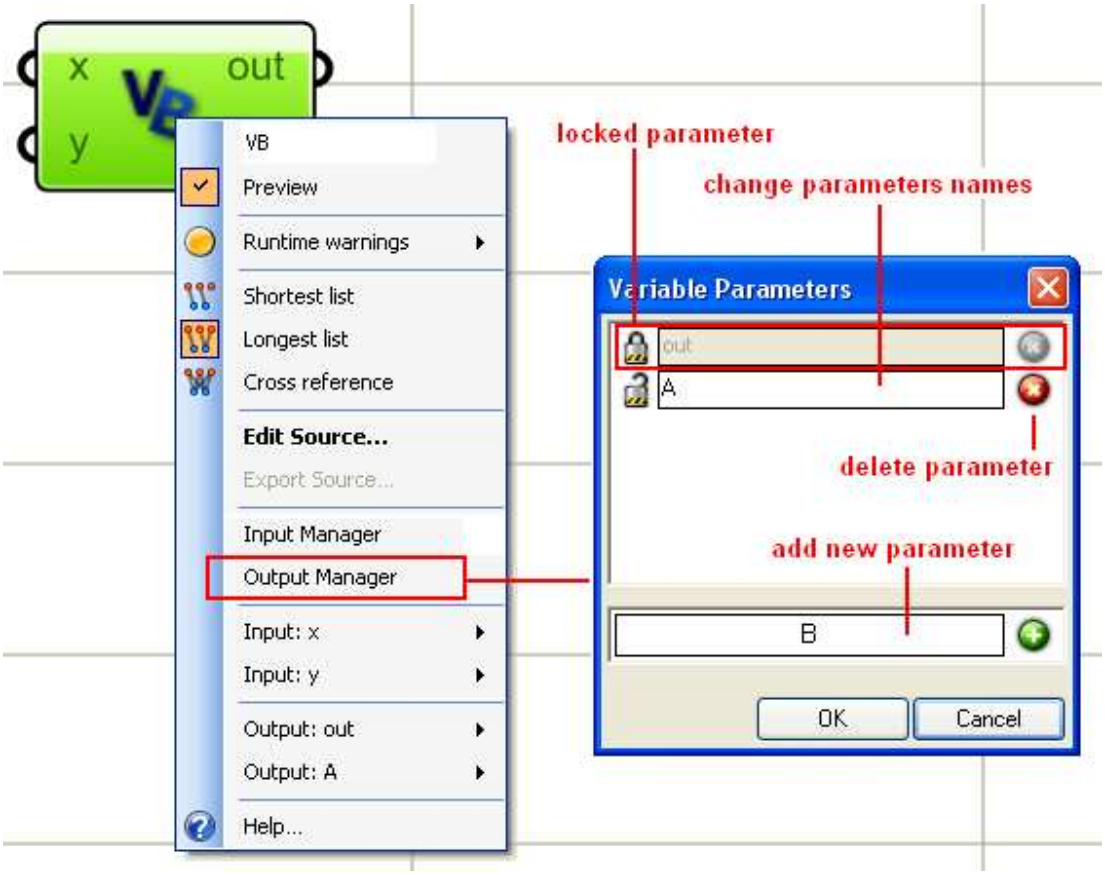
输入的参数可以通过主组件菜单控制。如果鼠标右键点击组件的中央，你将看到一个包括输入数据与输出数据详细信息的菜单。你可以使用这个菜单打开输出数据管理器并且改变参数名字，新建新参数或者如图中所示的把参数删除。

值得注意的是你的脚本工作签名（输入参数以及类型）只能通过这个菜单更改。一旦你开始编辑这个源头，只有功能本身会变，参数是不会变的。



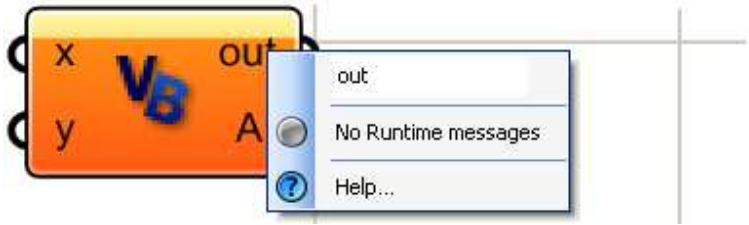
13.3 输出参数

你可以通过使用主组件菜单来自定义你需要的输出参数或返回数据。不像输入参数，输入参数没有类型规定。他们是被定义为一般系统“客体”类型，并且其间的运算能够指定类型、阵列或者什么都不指定。下面的图片显示了怎样使用输出数据管理器来设置输出数据。值得注意的是输出的参数是不能被删除的。它包括调试串以及用户调试串。

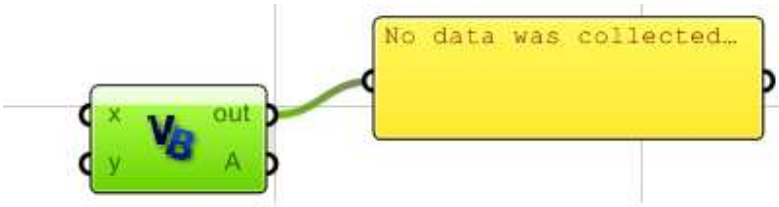


13.4 输出窗口和调试信息

被默认称为“out”的输出窗口提供调试信息。它列出了所有收集到的错误和警告。用户可以在代码中对其输入价值带帮助调试。当代码没有正确运行时，仔细地阅读收集到的信息是非常有用的。



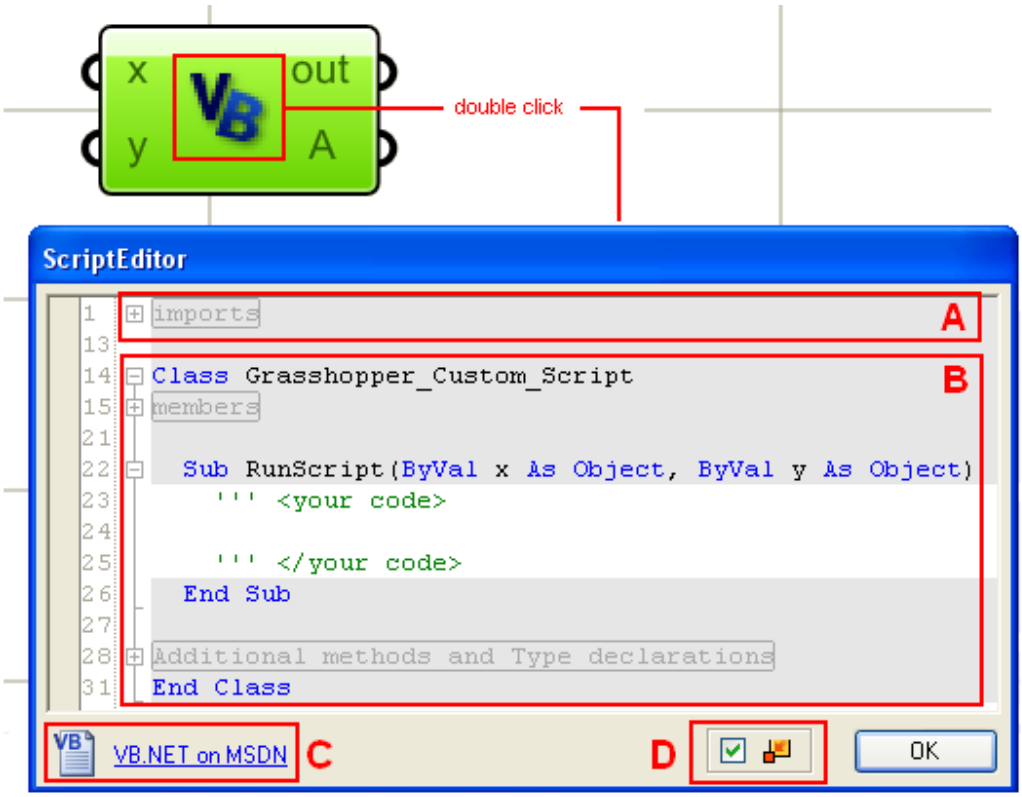
将输出的数据串与一个文字组件连接起来去看收集到的信息和调试信息是一个不错的办法。



13.5 内部脚本组件

要打开你的脚本组件，只需点击脚本组件的中央或者从组件菜单中选择“Edit Source…”。脚本组件包涵了两部分。他们是：

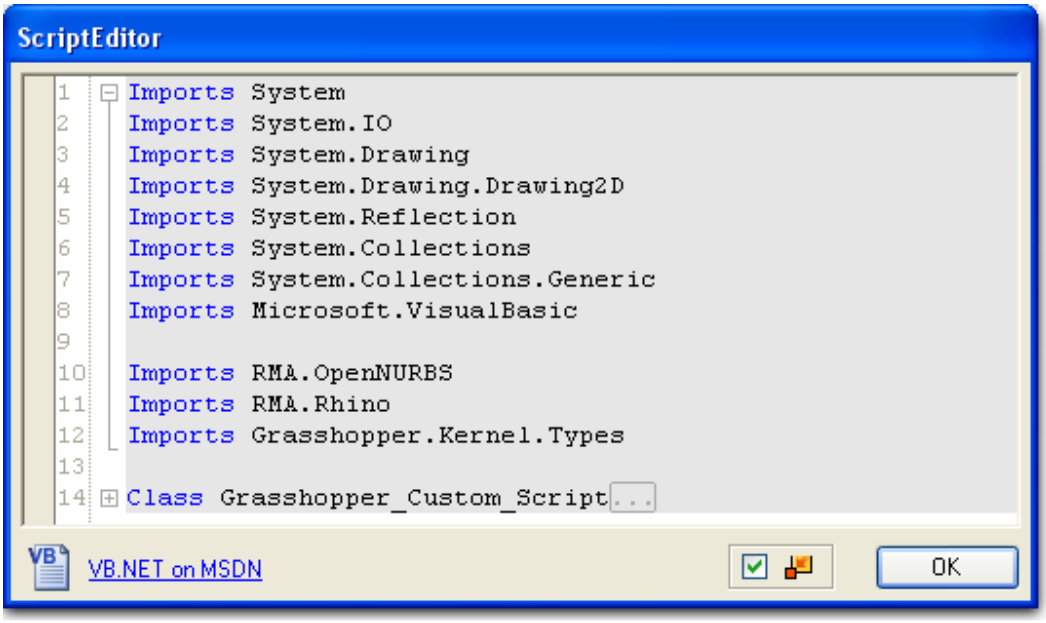
A: 输入
B: Grasshopper_Custom_Script 级
C: 在 VB.NET 上连接到微软开发者的网络帮助
D: 核对盒子来激活脚本组件的核心特征的输出数据




```
imports
Class Grasshopper_Custom_Script
members
  Sub RunScript(ByVal x As Object, ByVal y As Object)
    ''' <your code>
    ''' </your code>
  End Sub
Additional methods and Type declarations
End Class
```

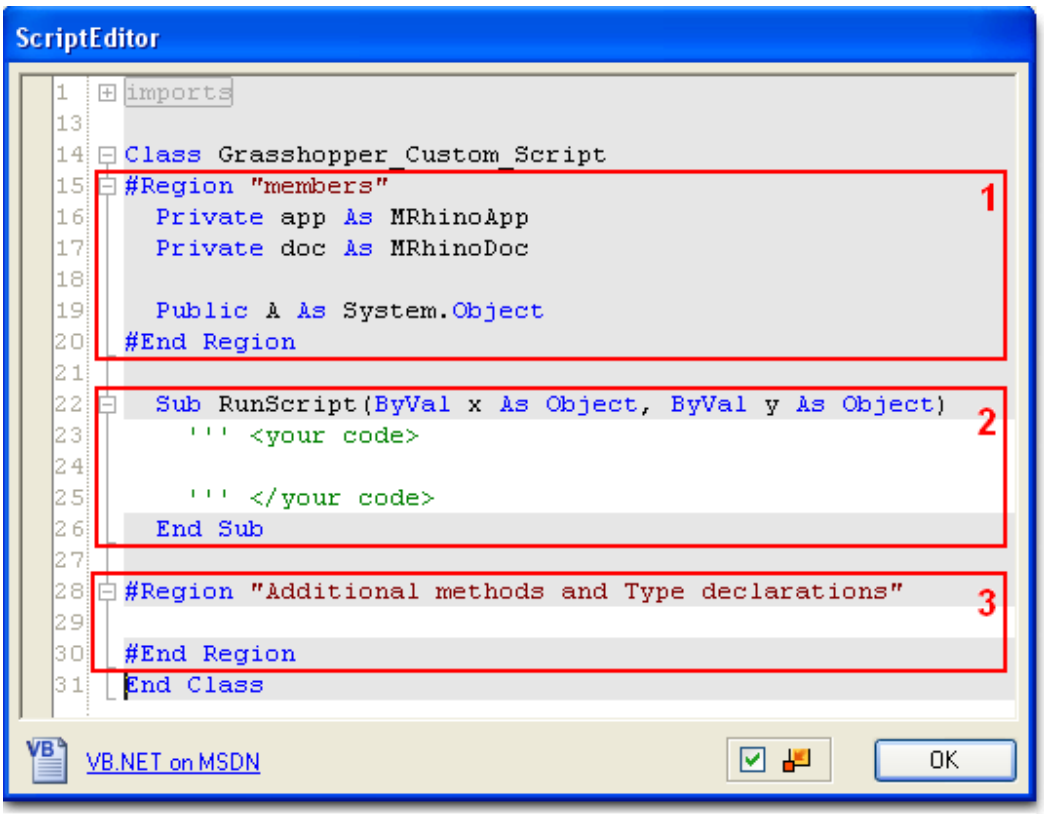
A: 输入

输入是你可能用到的代码中的外部 dll 文件。他们中的大多数是 DotNET 系统中的输入，但是也有两个 Rhino 的 dll 文件：RMA.openNURBS 和 RMA.Rhino。这些包括了 rhino 所有的几何和实用功能。还有的是 Grasshopper 特有的类型。



B: Grasshopper_Custom_Script 级

Grasshopper_Custom_Script 级包括三部分：

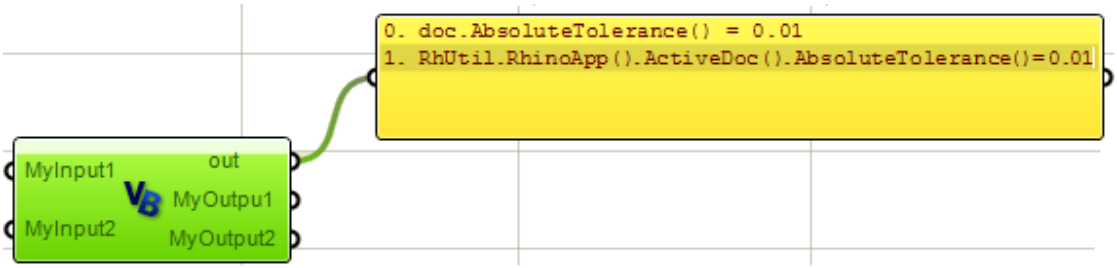


1. **Members(成员)**: 这包括了两个参考系。一个是参考现有的 rhino 应用程序（app 文件），另一个是参考活动的文档（doc 文件）。Rhino 的应用文件和文档也可以直接通过使用 RhnioUtil 来获得。成员族还包括脚本功能的返回数值或者输出数据。返回值被定义为一个一般系统类型，并且用户不能改变这个类型。

2. **RunScript(运行脚本)**: 这是用户编写他们代码的主要功能。

3. **Additional methods and Type Declaration (额外的途径和类型申明**:) 你可以把额外的功能和类型放在这里。

下面的例子显示了获得文档绝对容差的两条途径。第一个是通过使用连带脚本组件(doc 文件)的参考文档，第二个是通过 RhUtil （Rhino 实用功能）。注意到当你在输出窗口输入容差数值时，这两个功能会带来同样的结果。并且也注意到在这个例子中有两个输出数据（我的输出数据 1 和我的输出数据 2）、他们被列在脚本级的成员族中。




```
Class Grasshopper_Custom_Script
#Region "members"
  Private app As MRhinoApp
  Private doc As MRhinoDoc

  Public MyOutpu1, MyOutput2 As System.Object
#End Region

  Sub RunScript(ByVal MyInput1 As Object, ByVal MyInput2 As Object)
    ''' <your code>
    Dim tol As Double

    tol = doc.AbsoluteTolerance()
    Print(" doc.AbsoluteTolerance() = " & tol)

    tol = RhUtil.RhinoApp().ActiveDoc().AbsoluteTolerance()
    Print(" RhUtil.RhinoApp().ActiveDoc().AbsoluteTolerance()=" & tol)

    ''' </your code>
  End Sub
```

14 Visual Basic DotNET

14.1 引言

在网上和书籍中有很多关于 VB.NET 的文献。以下是你将在代码中用得上的基本知识回顾。

14.2 注释(comment)

尽多的注释你的代码是一个非常好的尝试。你将会惊奇的发现你会很快忘记你所做的东西！在 VB.NET,你可以使用一个撇号来表示，接下来几行是一个注释，运算器将会忽略它们。在 Grasshopper 中，注释是灰色的。例如：

```
'This is a comment ,i … I can write anything i like!
'Realy… anything
```

14.3 变量(Variables)

你可以认为变量是一个数据容器。不同变量有不同的尺寸，这决定于它要适应的数据类型。例如一个 int32 的变量在内存中占了 32 比特，并且以容器的名字作为变量的名字。一旦定义一个变量，其他的代码可以通过使用代码的名字找回容器里的内容。

让我们定义一个容器或一个变量叫 x 或 Int32 类型，并赋予它一个初始值为“10”.然后，让我们指派新的整数数值“20”给 x。以下是它在 VB DotNET 里面的

```
Dim x as Int32 = 10
    '如果你现在输出 x 的值，你将得到 10
x = 20
    '从现在开始，x 将成为 20
```

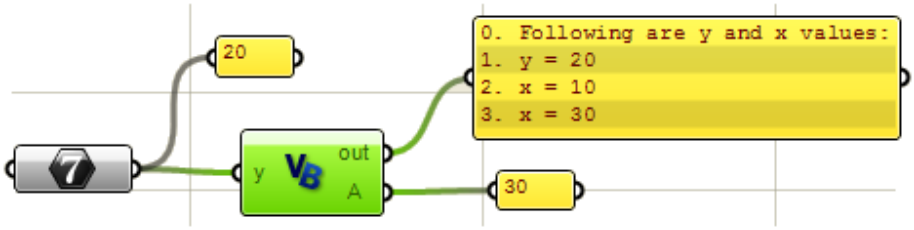
这里还有其他些常用的类型的例子：

```
Dim x as Double = 20.4          '定义关键词意味着我们将要定义一个变量
Dim b as Boolean = True         'Double, Boolean and String  是所有最基本的或者说
Dim name as String = "Joe"'系统定义的类型的样本
```

以下的 Grasshopper 例子使用了三个变量：

x: 是一个整数变量，在代码中被定义
y: 是一个整数变量，作为输入指数实现其功能
A: 是一个输出变量。

这个例子通过代码在输出窗口中输出了变量数值。如之前提到过的，这是一个很好的方式去看你的代码中发生了什么问题，并且通过调试能够最大限度减少对外部编辑器的依赖。




```
Sub RunScript(ByVal y As Integer)
  'Print variables values
  Print("Following are y and x values:")

  'Print input value (y)
  Print("y = " & y)

  'Declare x as an integer variable
  Dim x As int32 = 10

  'Print x initial value
  Print("x = " & x)

  'Set x value to be whatever was there plus input y
  x = x + y
  Print("x = " & x)

  'Assign x to output
  A = x
End Sub
```

指定有意义的变量名字，以便你能快速认出他们。这会让代码更具可读性并且比较容易调试。在这一章中的例子中，我们将尝试坚持良好的编程练习。

14.4 队列和清单（Array and lists ）

在 VB.NET 中有很多种方法定义队列。队列有分为单一数列和多维数列。你可以定义数列的大小或者使用动态数列。如果你知道数列中各元素的数目，你就可以以这个方式宣布这个被定义了大小的数列。

```
'一维数列
Dim myArray(1) As Integer
myArray (0) = 10
myArray (1) = 20
```

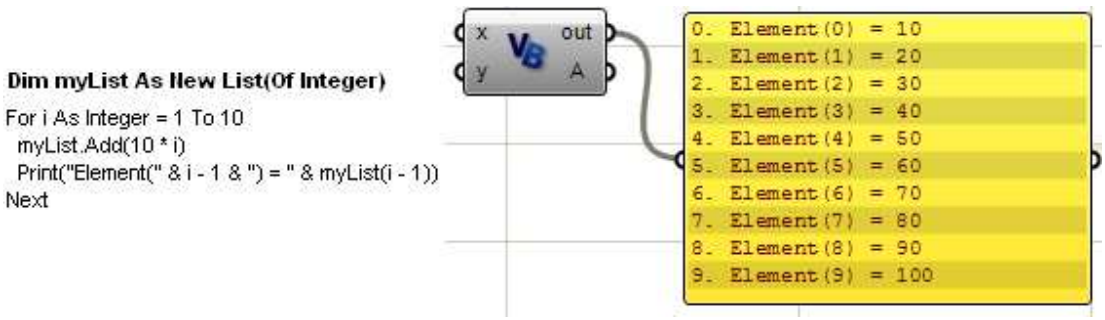
```
'二维数列
Dim my2DArray (1,2) As Integer
my2DArray (0, 0) = 10
my2DArray (0, 1) = 20
my2DArray (0, 2) = 30
my2DArray (1, 0) = 50
my2DArray (1, 1) = 60
my2DArray (1, 2) = 70
```

```
'宣布并指派数值
Dim myArray() As Integer    = {10,20}
```

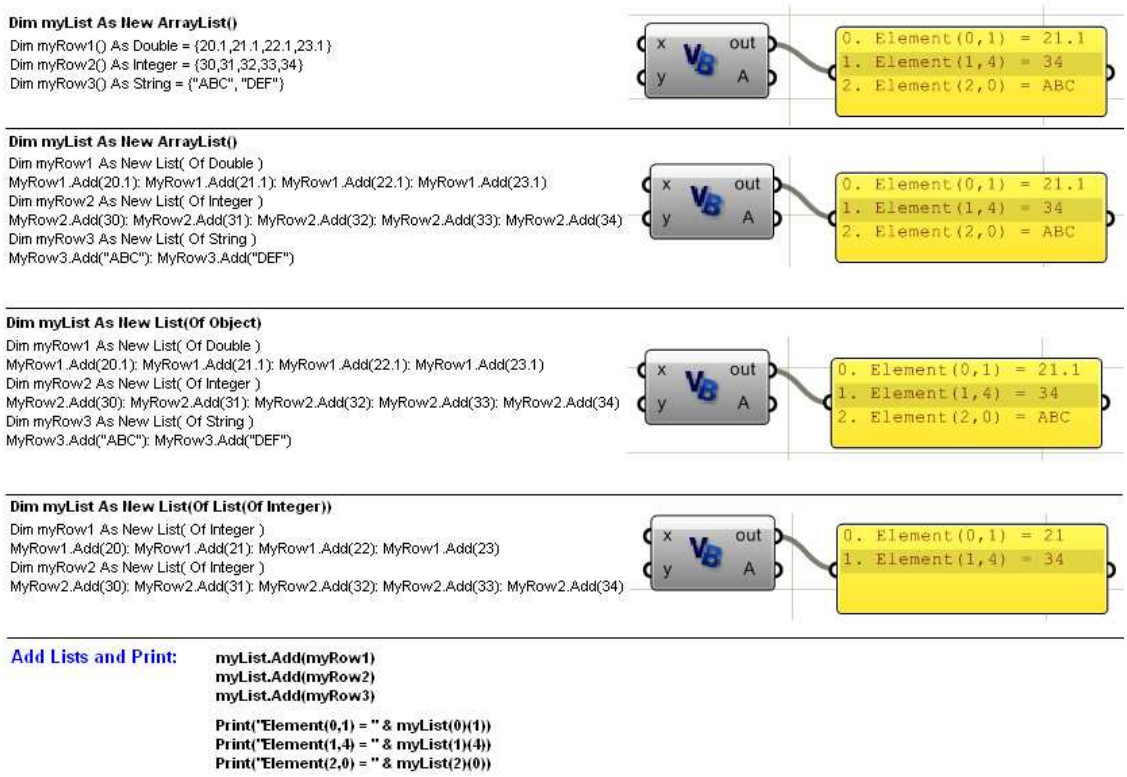
```
'宣布并指派数值
Dim my2DArray(,)As Integer = {{10,20,30},{40,50,60}}
```

记住在 VB.NET 中的数列是零为基准的，所以当你宣布一个数列大小为（9）时，这个数列有十个元素。多维数列也是这样的。

对于一维的动态数列，你可以宣布一个新的“列表”，就像接下来例子中所说的那样，并且开始添加元素到里面去。



你可以使用巢状列表或者行列列表来宣布相同的或混合类型的动态多维行列。请看下面的例子：



14.5 运算器

在 VB.NET 中有很多内置的运算器。它们运算一个或更多的操作数。这是一个常用运算器的表格，以资参考。

Type	Operator	Description
算术运算器	^	一个数是另一个数的乘方。
	*	两个数的乘积。
	/	两个数相除并返回一个浮动小数点数值。
	\	两个数相除并返回一个整数数值。
	Mod	两个数相除并返回余数。
	+	两个数相加或返回一个数字表达式的正值。
	-	返回两个数字表达式的差值或者一个数字表达式的负值。
指派运算器	=	给变量指派一个数值。
	^=	变量的值是一个数字表达式的乘方并且将这个结果指派回到变量中去。
	*=	将变量的数值与数字表达式的数值相乘，并将这个结果指派回到变量中去。
	/=	将变量的数值与数字表达式的数值相除，并将这个带浮动小数点的结果指派回到变量中去。
	\=	将变量的数值与数字表达式的数值相除，并将这个整数的结果指派回到变量中去。
	+=	将一个数字表达式的数值与一个数字变量的数值相加，并将这个结果指派回到变量中去。还可以被用来将一个线性表达式与线性变量相关联，并将这个结果指派回到变量中去。
	-=	从一个变量的数值中减去一个表达式的数值，并且将这个结果指派回到变量中去。
	&=	将一个线性表达式与一个线性变量或特性相关联，并且将这个结果指派回到变量或特性中去。
比较运算器	<	小于
	<=	小于或等于
	>	大于
	>=	大于或等于
	=	等于
	<>	不等于
关联运算器	&	生成两个表达式的线性关联。
	+	生成两个线性表达式。
逻辑运算器	And	两个布尔表达式的逻辑连词。
	Not	一个布尔表达式的逻辑否定。
	Or	两个布尔表达式的逻辑分离。
	Xor	两个布尔表达式的逻辑排除。

14.6 条件性陈述

你可以将条件性陈述认为是一群带闸门的代码，当闸门打开的条件被满足时，这些代码就开始运行。最常用的条件性陈述是“if”陈述，它以如下格式出现**“IF<条件> Then <代码> End IF”**。

```
    '单行，如果陈述不需要 End If: 条件=(x<y)，代码=(x=x+y)
    If x < y Then x = x + y

    '多行，需要 End If 来结束这一群代码
If x < y Then
    x = x + y
End If
```

还可以用**“Else If … Then”**和**“Else”**来选择另一个代码群来执行。例如：

```
If x < y Then
    x = x + y    '执行这一行，然后走到“End If”这一步之后
Else If x > y Then
    x = x – y    '执行这一行，然后走到“End If”这一步之后
Else
    x = 2*x      '执行这一行，然后走到“End If”这一步之后
End If
```

还有“选择情况”的陈述。这被用来根据一个表达式的不同值（例子中的“index”），执行不同的代码群。例如：

```
Select Case index
    Case 0 '如果 index=0 执行下一行，否则直接跳转到下一个情况
        x = x * x
    Case 1
        x = x ^ 2
    Case 2
        x = x ^ (0.5)
End Select
```

14.7 回路

只要回路的条件被满足，回路允许一遍又一遍地重复执行回路内的代码群。回路有不同的种类。这里我们将解释最常用的两种回路。

“For … Next” 回路

这是最常用的回路形式。这回路的结构是这样的：

```
For < 内容=开始值> To <结束值> [Step <过程值>]

    'for 循环体从这里开始
    [陈述/代码在回路内被执行]
```

```
    [ Exit For ] '可选的：在任何一点可以退出回路

    [ 其他陈述]


    [ Continue For ]    '可选的：不执行下面的回路陈述

    [ 其他陈述]

    'for 循环体从这里结束（就在 "Next"之前）
    'Next 意味着：回到 for 循环体的开始，检查看内容是否已经超过结束值
    '如果内容已经超过结束值，那么退出循环并且执行 "Next " 之后的陈述
    '否则，通过过程值增加内容
Next
[ For 循环之后的陈述]
```

以下的例子使用了一个循环来重复一系列的地址：

```
'一系列的地址
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

'回路内容
Dim i As Integer
Dim place As String
Dim count As Integer = places_list.Count()

'回路从 0 开始，到 count-1 （count=3，但是最后 places_list 的内容=2）
For i=0 To count-1
    place = places_list(i)
    Print( place )
Next
```

如果在一个阵列的对象中循环，你可以用 For…Next 的循环来重申这一阵列元素，而不必使用一个标签。下面的例子还可以写成这样：

```
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

For Each place As String In places_list
    Print( place )
Next
```

"While … End While" 循环
这也是一个常用的循环。这个循环的结构看起来是：

```
While < 某些条件为真时>

    'While 回路从这里开始
      [ 回路内被执行的陈述]
```

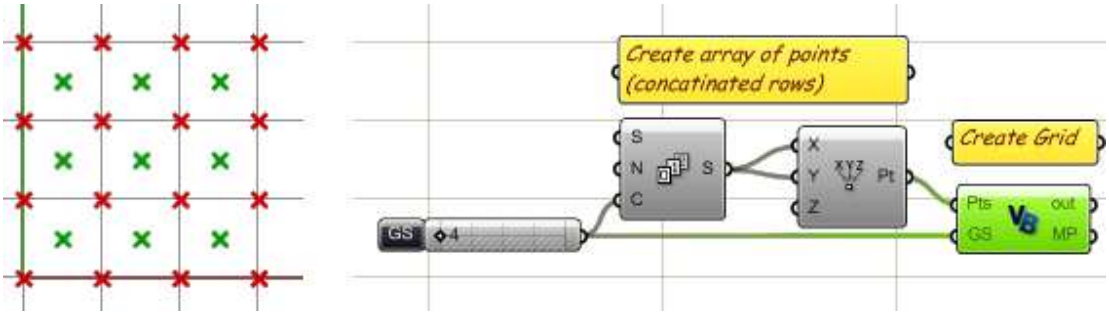
```
    [ Exit While ]    '可选择的退出回路

     [ 其他陈述]

     [ Continue While ] '可选择的退出执行接下来的循环。

     [ 其他陈述]

    'While 回路在此结束
    '回到回路的起点，确认条件是否仍然为真，然后执行主体操作
    '如果条件不为真，则退出循环并且执行"End While"之后的陈述

End While
[ While 回路之后的循环]
```

这是用 while 回路重写的前面的例子：

```
Dim places_list As New List( of String )
places_list.Add( "Paris" )
places_list.Add( "NY" )
places_list.Add( "Beijing" )

Dim i As Integer
Dim place As String
Dim count As Integer = places_list.Count()

i = 0
While i < count          '判断(i<count) 是真是假
    place = places_list(i)
    Print( place )
    i = i + 1
End While
```

14.8 巢状回路

巢状回路是一个回路里包含另一个回路的回路。例如，如果我们有一网格的点，为了能在表中提取每一个点，我们需要使用一个巢状回路。下面的例子告诉我们怎样将一个一维点数列转化成一个二维网格。然后它将用来推动网格去寻找巢室中点。



这个脚本包含两部分：

- 首先将一个一维阵列转化成我们称为“网格”的二维阵列。

然后推动网格去寻找巢室中点。
这两部分我们都使用巢状回路。这是 Grasshopper 里的脚本定义：

```
Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)

    'Create a grid of points
    Dim Grid As New ArrayList()

    Dim i As Integer
    Dim j As Integer

    'Nested loop to covert 1D array to 2D grid
L1  For i = 0 To Pts.Count() - 1 Step GS
      'Declare a row of points
      Dim Row As New List( Of On3dPoint )
L2    For j = i To i + GS - 1
        'Get a reference od the point
        Dim pt As On3dPoint
        pt = Pts(j)

        'Add point to the row
        Row.Add(pt)
      Next
      'Add row to the grid
      Grid.Add(Row)
    Next

    'Process the grid to find mid points of cells
    Dim mid_points As New List( Of On3dPoint )
L1  For i = 1 To Grid.Count() - 1
      'Get first and second rows
      Dim Row0 As List( Of On3dPoint )
      Row0 = Grid(i - 1)
      Dim Row1 As List( Of On3dPoint )
      Row1 = Grid(i)

L2    For j = 1 To Row0.Count() - 1
        Dim mid_pt As New On3dPoint
        mid_pt = (Row0(j-1) + Row0(j) + Row1(j-1) + Row1(j)) / 4
        mid_points.Add(mid_pt)
      Next
    Next

    'Assign mid point to output
    MP = mid_points
End Sub
```

14.9 附属和功能

运行脚本是所有脚本组件使用的主要功能。当你打开 Grasshopper 里的一个默认脚本组件时你将看到这个：

```
Sub RunScript(ByVal x As Object, ByVal y As Object)
  ‘’’<your code…>
End Sub
```

Sub… End Sub: 嵌入代码功能块的关键词
“RunScript”: 附属的名称
“(…)”: 附属名称后的插入语，嵌入输入参数
“ByVal x As Object,…”: 所谓的输入参数

每个输入参数需要定义以下内容：

- **ByRef（通过参照）**或 **ByVal（通过数值）**：确定一个参数是通过数值还是参照传递。
- 参数的名字。
- 在关键词“As”之前的参数类型。

值得注意的是，Grasshopper RunScript sub 里的输入参数都是通过数值传递的***（关键词 ByVal ）***。这意味着他们是原始输入数值的复制品，在脚本里对这些参数所做的任何改变不会改变原始的输入。但是，如果你在你的脚本组件里定义更多的附属/功能，你可以通过参照传递参数***（ByRef）***。通过参照传递参数意味着对功能块里面参数的任何改变都会改变传递的原始数值，只要这个功能存在。

你可以在 RunScript 里面编写所有的代码，但是，你可以根据需要定义外部的附属和功能块。那为什么要用外部的功能呢？

- 简化主要的功能代码。
- 让代码更具可读性。
- 隔离并且重新利用公共功能。
- 定义特殊的功能比如递归。

一个 **Sub（附属）**和一个 **Function(功能块)**之间到底有什么不同呢？如果你不需要一个返回值时你可以定义一个附属。功能块可以让你返回一个结果。基本上你会指派一个数值给功能的名字。例如：

```
Function AddFunction( ByVal x As Double, ByVal y As Double )
    AddFunction = x + y
End Function
```

这意味着，你不需要有一个功能来返回一个数值。附属可以通过输入通过参照传递的参数。在下面的例子中，”rc”就是用来返回结果的：

```
Sub AddSub( ByVal x As Double, ByVal y As Double, ByRef rc As Double )
    rc = x + y
End Sub
```

这是使用功能块和附属的访问功能：

```
Dim x As Double = 5.34
Dim y As Double = 3.20
Dim rc As Double = 0.0
'能用一下任一方式得到结果
rc = AddFunction( x, y )    '将功能结果指派给 "rc"
AddSub( x, y, rc )          'rc 通过参照传播，将有另外一个结果
```

在巢状回路部分，我们以例子说明了从一系列的点建立一个网格然后计算中点。这两个功能截然不同，可以分开在一个外部附属以及可能重新应用在下面的代码中。这是一个用外部功能块重写前面的网格例子。

```
Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)

  'Create a grid of points
  Dim Grid As New ArrayList()

  'Call grid function
1 Call CreateGrid(Pts, Grid, GS)

  'Call mid points function
  Dim mid_points As New List( Of On3dPoint )
2 Call FindMidPoints(Grid, mid_points)

  'Assign mid point to output
  MP = mid_points
End Sub

#Region "Additional methods and Type declarations"

'Function to convert 1d array to 2d array
1 Sub CreateGrid( ByVal Pts As List(...

'Function to find grid mid points
2 Sub FindMidPoints(ByVal Grid As ArrayList, mid_points As List(...

#End Region
End Class
```

这两个附属展开时是这样子的：

```
#Region "Additional methods and Type declarations"

  'Function to convert 1d array to 2d array
  Sub CreateGrid( ByVal Pts As List(Of On3dPoint), ByRef Grid As ArrayList, ByVal GS As Integer )

    Dim i As Integer
    Dim j As Integer

    For i = 0 To Pts.Count() - 1 Step GS
      'Declare a row of points
      Dim Row As New List( Of On3dPoint )
      For j = i To i + GS - 1
        'Get a reference od the point
        Dim pt As On3dPoint
        pt = Pts(j)

        'Add point to the row
        Row.Add(pt)
      Next
      'Add row to the grid
      Grid.Add(Row)
    Next

  End Sub

  'Function to find grid mid points
  Sub FindMidPoints(ByVal Grid As ArrayList, mid_points As List(Of On3dPoint ))

    Dim i As Integer
    Dim j As Integer

    For i = 1 To Grid.Count() - 1
      'Get first and second rows
      Dim Row0 As List( Of On3dPoint )
      Row0 = Grid(i - 1)
      Dim Row1 As List( Of On3dPoint )
      Row1 = Grid(i)

      For j = 1 To Row0.Count() - 1
        Dim mid_pt As New On3dPoint
        mid_pt = (Row0(j - 1) + Row0(j) + Row1(j - 1) + Row1(j)) / 4
        mid_points.Add(mid_pt)
      Next
    Next

  End Sub
#End Region
```

14.10 Recursion 递归

递归功能是特殊种类的功能，它调叫自身直到中止条件被满足。递归经常被用于数据搜索，细分以及生成系统。我们将讨论一个现实递归如何运作的例子。更过递归的例子，可以在 Grasshopper 维基百科和画廊网页找到。

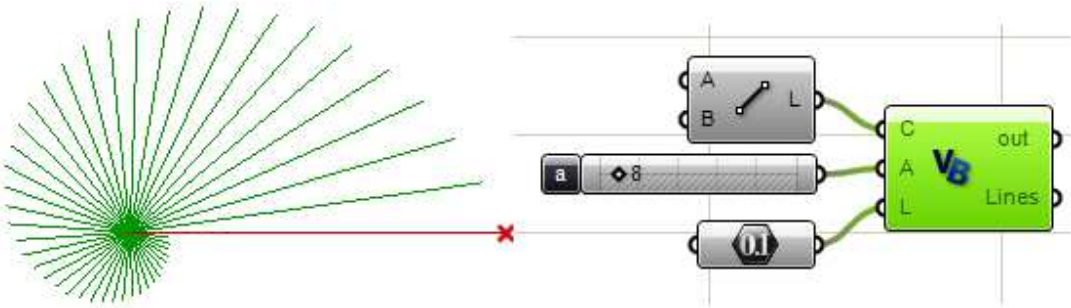
下面的例子选取输入线段的一小部分，并且旋转一个特定角度。他不断重复这个动作知道直线长度小于一个最小值。

输入参数是

- 标示线(C).
- 以弧度表示的角度(A). Slider 以度数显示度数，但请把它转化为弧度。
- 最小长度 (L) – 作为一个中止条件。
-

输出参数是：：

- 线的阵列。



为了对比，我们将分别采用反复和递归的方法解决一样的例子

递归方法解决。注意到在“DivideAndRotate”附属中有：

- 退出附属的中止条件。
- 相同功能的调用（递归功能调用自身）。
- “AllLines” (线的阵列) 通过参照传播，以保持添加新线到列表中。

```
Sub RunScript(ByVal C As OnLine, ByVal A As Double, ByVal L As Double)

    'Declare all lines
    Dim AllLines As New List( Of OnLine )

    'Call recursive function
    Call DivideAndRotate(Line, AllLines, A, L)

    'Assign return value
    Lines = AllLines
End Sub

#Region "Additional methods and Type declarations"

  Sub DivideAndRotate(ByVal Line As OnLine,
                      ByRef AllLines As List(Of OnLine),
                       ByVal angle As Double,
                       ByVal MinLength As Double)

    'Check the stopping condition
    If Line.Length() < MinLength Then Exit Sub

     'Take a portion of the line
     Dim new_line As New OnLine(Line)
     Dim end_pt As New On3dPoint
     end_pt = new_line.PointAt(0.95)

     new_line.To = end_pt

     'Rotate
     new_line.Rotate(angle, OnUtil.On_zaxis, Line.from)

     AllLines.Add(new_line)

     'Call self
     Call DivideAndRotate(new_line, AllLines, angle, MinLength)

  End Sub

#End Region
```

这是使用一个“while”回路，用反复方法实现一样的功能：

```
Sub RunScript(ByVal C As OnLine, ByVal A As Double, ByVal L As Double)

    'Declare all lines
    Dim AllLines As New List( Of OnLine )

    'Find current length
    Dim current_L As Double = C.Length()

    Dim new_line As OnLine
    new_line = C

    'Loop until length is less than min length
    While current_L > L
      'Generate the new line
      new_line = DivideAndRotate(new_line, A)

      'Add to list
      AllLines.Add(new_line)

      'Stopping condition
      current_L = new_line.Length()
    End While

    'Assign return value
    Lines = AllLines
  End Sub

#Region "Additional methods and Type declarations"

 Function DivideAndRotate(ByVal L As OnLine, ByVal A As Double) As OnLine

    'Take a portion of the line
    Dim new_line As New OnLine(L)
    Dim end_pt As New On3dPoint
    end_pt = new_line.PointAt(0.95)

    new_line.To = end_pt

    'Rotate
    new_line.Rotate(A, OnUtil.On_zaxis, L.from)

    'Function return
    DivideAndRotate = new_line

 End Function

#End Region
```

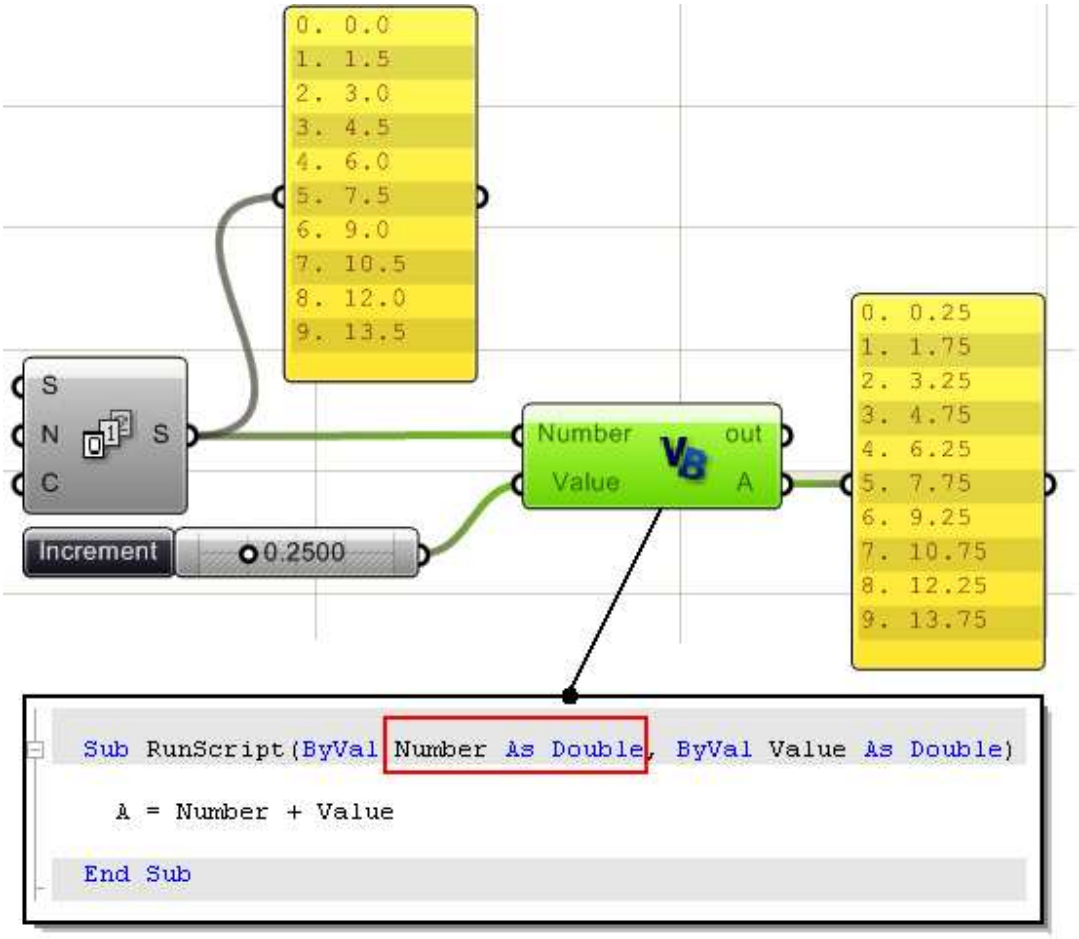
14.11 Grasshopper 中的推进列表

Grasshopper 的脚本组件可以以两种方式推进输入参数的列表：

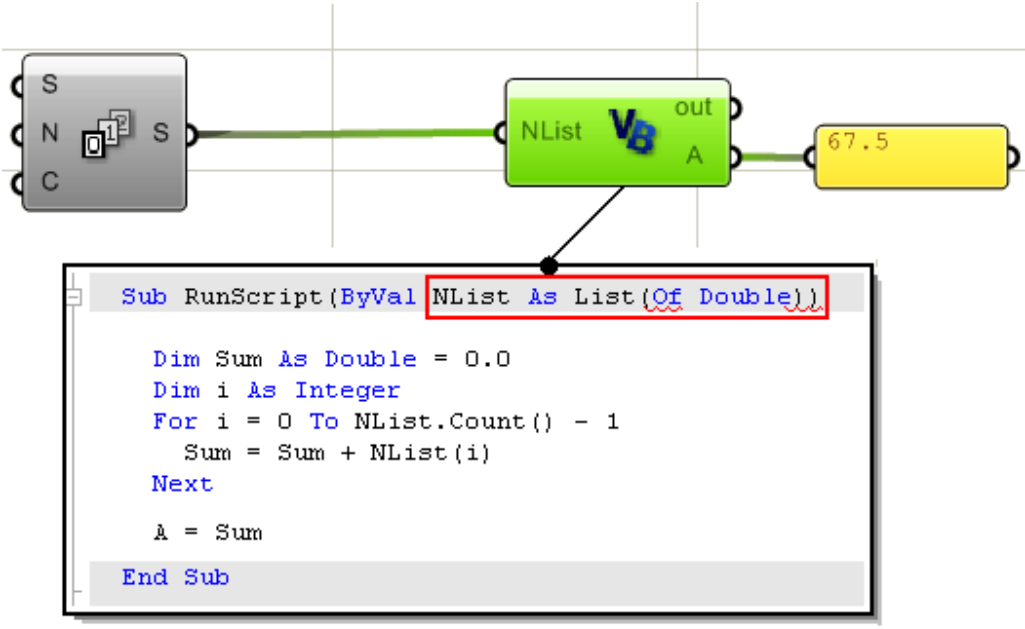
1. 一次推进一个输入数值（组件被调用的次数与输入队列中数值的数目相同）
2. 一次推进所有的输入数值（组件被调用一次）。

如果你需要单独推进一个列表中的每个元素，用第一种方法比较方便。例如，如果你有一列数字，你想逐个增加“10”，你可以使用第一种方法。但是如果你需要给所有元素全部加上的总体功能，你需要使用第二张方法，并且把整个列表当真一个输入数值。

下面的例子告诉我们怎样用第一种方法，即一次推进一个输入数值的方法，推进一系列数据。在这个案例中，RunScript 功能块被调用了 10 次（每个数字一次）。值得注意的关键点是，输入参数是作为一个“Double”被传递，有别于我们在下个例子中将看到的“List (of Double)”。



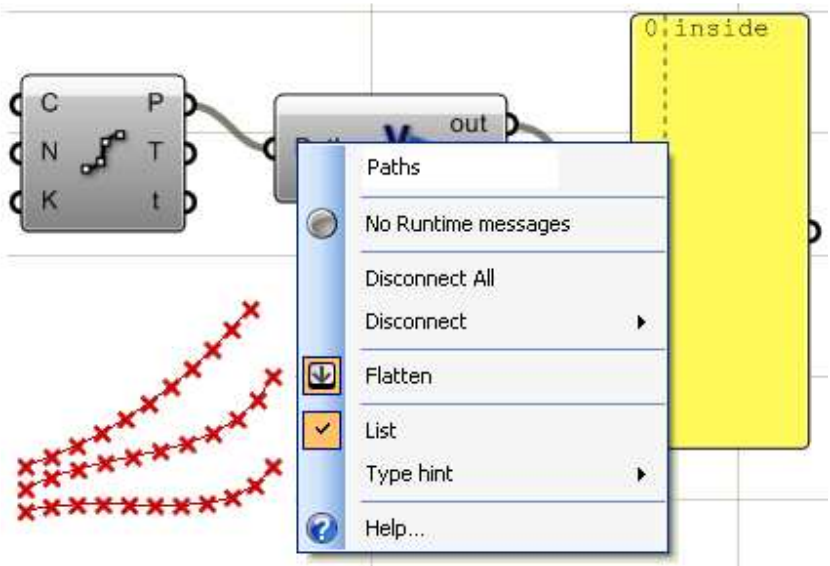
接下来，我们输入数字列表。你可以通过在输入参数处单击鼠标右键，选中“列表”。RunScript 功能块只被调用一次。



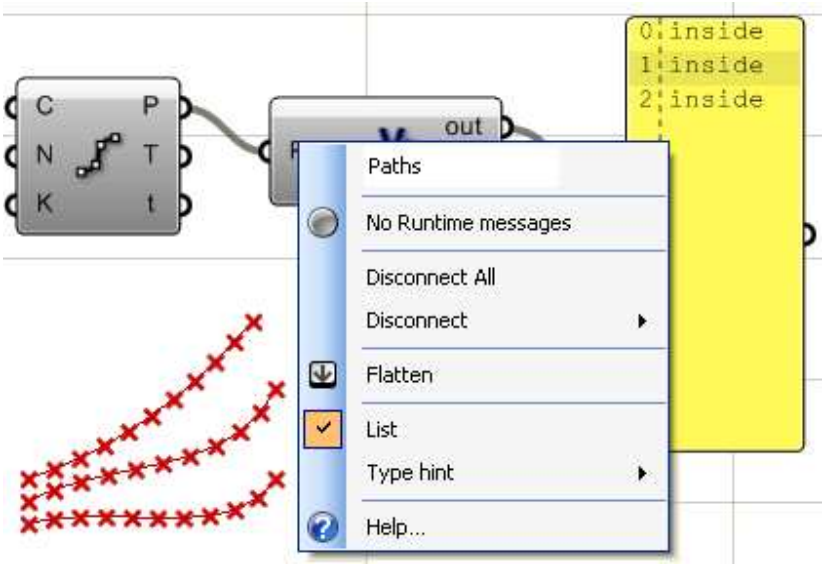
14.12 Grasshopper 中的进程树

树（或说是多维数据）可以一次一个元素的推进，或者一次一个分支（路径）的推进，又或者一次推进所有路径。例如，如果我们将三段曲线每段都分为十个部分，我们将得到一个有三个分支或说途径，每个又有 11a 个点的结构。如果我们以此为输入数据我们将得到如下：

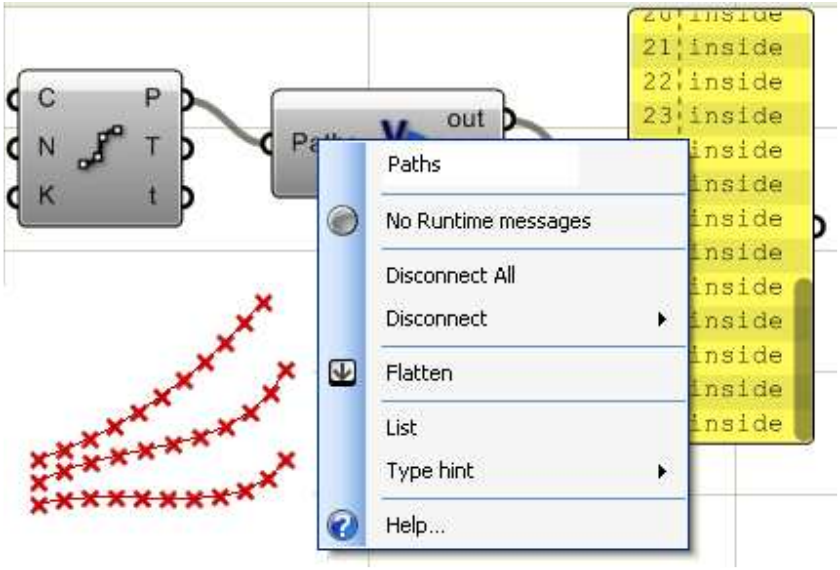
A: 如果“**Flatten(**变平**)**”和“**List(**列表**)**”同时被勾上，这个组件将被调用一次，并且平列表的所有点都被传播：



B: 如果只有“**List(**列表**)**”被勾上，这个组件每次将被调用三次，一条曲线上的分界点 **List(**列表**)**被传递：



C: 当什么都没有被勾上，这个功能块将在每个分界点被调用（这个例子中会被调用 33 次）。如果只有“**Flatten(**变平**)**”被勾上，也是如此：



这是 VB 组件中的代码。基本上只输入单词“inside”来现实这个组件被调用：

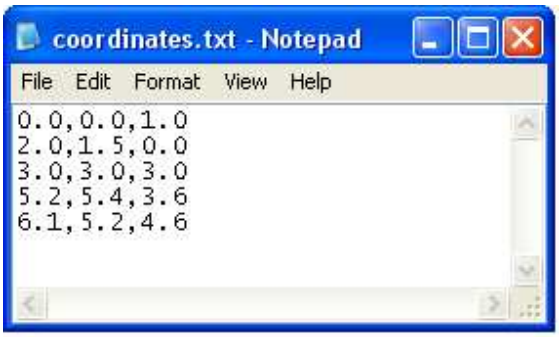
```
Sub RunScript(ByVal Paths As Object)

  Print("inside")

End Sub
```

14.13 文件 File I/O

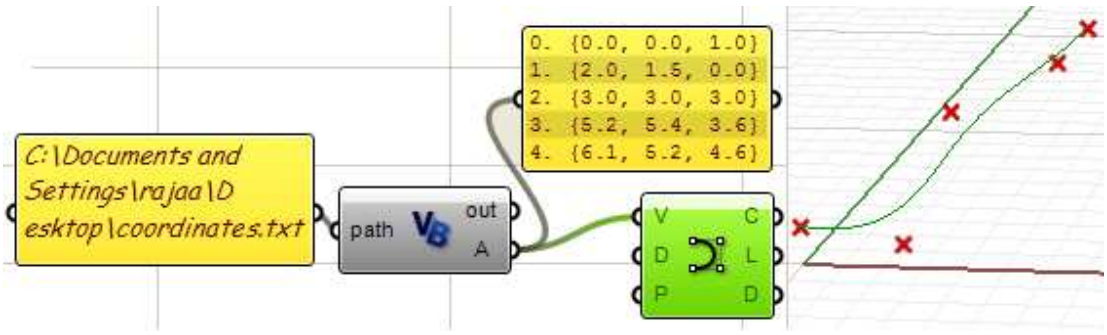
有很多途径读取并且写进 VB.NET 的文件，并且很多指导和文件可以在因特网和打印资料中找到。大体上来讲，读取一个文件包括以下几点：

打开文件。一般需要一个途径导向。
读取数据串（整个数据串或逐行）。
用一些分隔符号象征数据串
投影每一个象征（这个案例中加倍）。
储存结果。

我们将介绍一个简单的例子，从文档文件中分析点。使用下面的文档文件格式，我们将读取每一行作为一个单独的点并且使用第一个值为点的 x 坐标，第二个作为 y 坐标，第三个作为 z 坐标。我们将使用这样点作为曲线控制点。



VB 组件被当作一串数据被接受，它是被读取以及输出成三维点的文件的路径。



这是脚本组件里的代码。它几乎没有诱骗性错误代码，以保证文件存在并且有内容。

```
Sub RunScript(ByVal path As String)

  'Check if file exists
  If (Not IO.File.Exists(path)) Then
    Print("Exit without reading")
    Return
  End If

  'Read the file
  Dim lines As String() = IO.File.ReadAllLines(path)

  'Check that file is not empty
  If (lines Is Nothing) Then
    Print("File has no content")
    Return
  End If

  'Declare list of points
  Dim pts As New List(Of On3dPoint)

  'Loop through lines
  For Each line As String In lines
    'Tokenize line into array of strings separated by ","
    Dim parts As String() = line.Split(",".ToCharArray())

    'Make sure that each line has exactly 3 values
    If UBound(parts) <> 2 Then Continue For

    'Convert each coordinate from string to double
    Dim x As Double = Convert.ToDouble(parts(0))
    Dim y As Double = Convert.ToDouble(parts(1))
    Dim z As Double = Convert.ToDouble(parts(2))

    pts.Add(New On3dPoint(x, y, z))
  Next

  A = pts

End Sub
```

15 Rhino .NET SDK

15.1 概述

Rhino .NET SDK 提供访问 OpenNURBS 几何库与公用函数的功能. 当你下载.NET SDK 时会包含一个帮助文件，可以获得非常多的相关资料. 下载地址如下:
http://en.wiki.mcneel.com/default.aspx/McNeel/Rhino4DotNetPlugIns.html

在这一节,我们将重点讨论关于 Rhino 几何库与公用函数的 SDK 部分. 我们会举例演示如何利用 Grasshopper 的 VB 脚本来创建与修改几何图形.

15.2 了解 NURBS

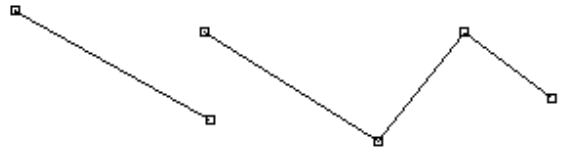
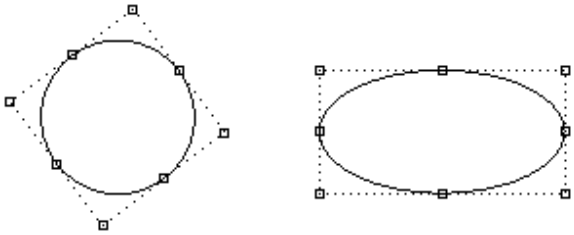
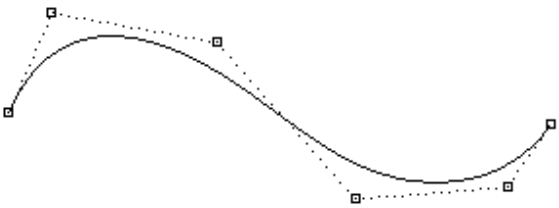
Rhino 是一个使用 Non-Uniform Rational Basis Spline（非均匀有理 B 样条，简称 NURBS)来定义曲线与曲面几何的 NURBS 建模工具. NURBS 是一个精确的曲线与曲面的数学方程式且编辑非常的直观.

这个网址(http://en.wikipedia.org/wiki/NURBS)有许多的参考书籍与资料帮助你深入的了解 NURBS 知识. 了解 NURBS 的基础知识对于帮助你使用 SDK 课程与函数是非常有用的，而且是必不可少的学习内容.

四个项目来定义一条 NURBS 曲线. Degree（阶数）, control points（控制点）, knots（节点）与 evaluation rules（估算法则）:

Degree 阶数

阶数一般会为 1、2、3 或 5 的正整数. Rhino 允许在阶数 1-11 的范围下作业. 下面列举几个曲线以及曲线阶数的范例:

	**直线** 与 **复合直线** 是阶数为 1 的 NURBS 线条 Order(曲线次数)= 2 (次数 = 阶数 + 1)
	**圆** 与**椭圆** 都是阶数为 2 的 NURBS 曲线. 他们的也都是有理且非均匀的曲线. Order(曲线次数) = 3.
	自由 Free form **曲线** 一般常用阶数为 3 的 NURBS 曲线. Order (曲线次数)= 4 5 阶的曲线也比较常用，其他阶数的曲线不常用

Control points 控制点

NURBS 曲线控制点是一组至少为(阶数+1)个的点.通过移动控制点来改变曲线的形状是最常用曲线编辑的方法.
控制点有个关联的参数叫做 Weight **权重**. 一般情况下，权重都是正数.当一条曲线所有控制点的权重都相同的时候(通常为 1)，这条曲线我们称之为无理曲线.我们有一些案例来说明如何在 Grasshopper 中交互式的改变控制点的权重.

Knots or knot vector 节点与节点矢量

每条 NURBS 曲线都有一个列表数据与之相关的叫做节点矢量. 节点有点难理解和定义，好在 SDK 函数已经为你做了这些工作. 当然，有些知识对于学习节点矢量还是非常有用的:

Knot multiplicity 节点重数
节点值重复的次数称之为基点重数. 有些节点值的重复数不能大于曲线的阶数. 了解这些知识对理解节点是非常有帮助的.

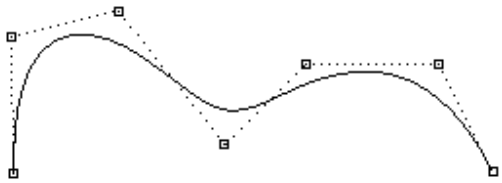
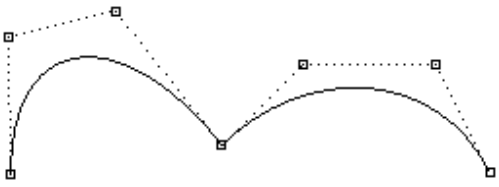
Full multiplicity knot 全复节点 即节点重复次数等于曲线阶数. 全复节点固定 在曲线两端，这也是为何曲线末端控制点与曲线端点一致的原因. 如果曲线的中间节点矢量是全复节点，那么曲线会通过该控制点且该点必须为锐角点.

Simple knot 单纯节点: 即一个节点仅呈现一个数值.

Uniform knot vector 均匀节点矢量 需具备的两个条件:

1. 节点个数= 控制点个数 + 阶数 – 1.
2. 以一个全复节点开始,接下来是一个单纯节点，最后一全复节点结束，且节点值等差.这是一种典型周期曲线. 后面我们会看到其不同于周期曲线的工作方式.

这里有两条控制点相同当节点矢量不同的曲线:

	阶数 = 3 控制点数量 s = 7 节点矢量= (0,0,0,1,2,3,5,5,5)
	阶数 = 3 控制点数量= 7 节点矢量= (0,0,0,1,1,1,4,4,4) ***提示**: 全复节点在曲线的中间则会成为锐角点且曲线会经过对应的控制点.*

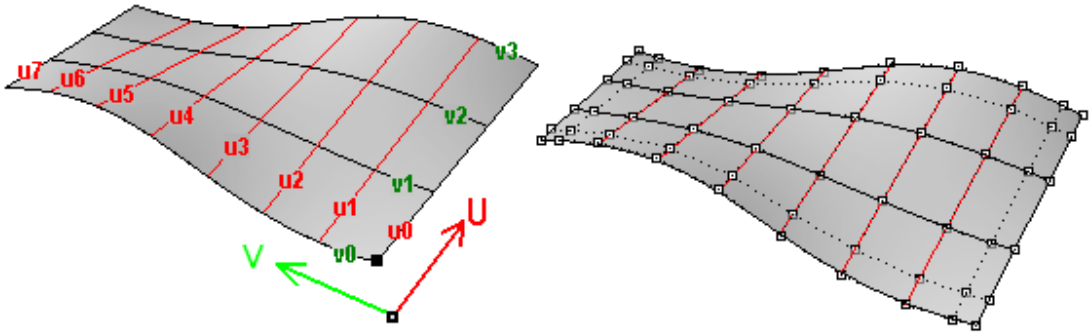
Evaluation rule 估算法则

估算法则一个使用赋予点数据的数学方方程式. 这个方程式包括阶数、控制点与节点.

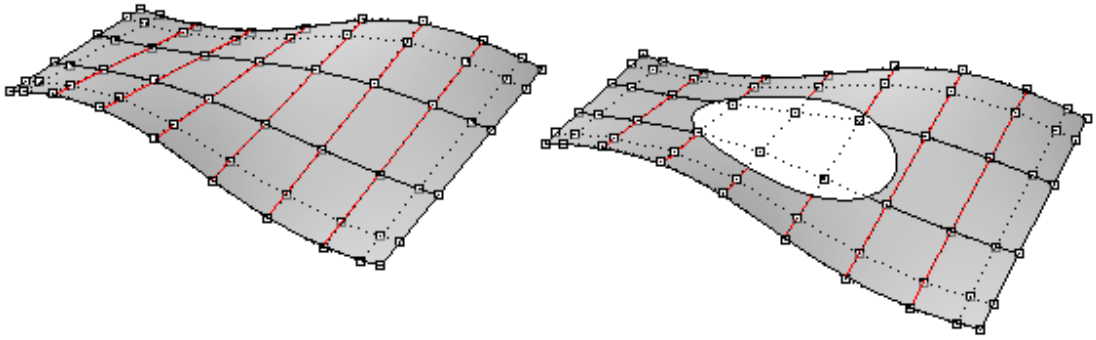
利用这个方程式,SDK 函数生成一条曲线的参数，且在曲线上生成与这条曲线相对应的控制点. 参数是一个位于曲线区间内部的数字. 区间通常是增加的且有固定的两个值:最小值(m_t(0)) 一般为曲线的起点和最大值(m_t(0)) 一般为曲线的终点.

NURBS 曲面

你可以把 NURBS 曲面理解为带有两个方向的 NURBS 曲线网格. NURBS 曲面的形状由曲面两个方向（U 向、V 向）的控制点与阶数来定义.欲获得更多相关信息请 参考 Rhino 帮助文件-词汇 部分



NURBS 曲面可以被剪切或是恢复剪切.剪切曲面可以理解为一个基本的 NURBS 曲面与一条封闭的曲线来剪切指定曲面的形状. 每个曲面都有一个封闭的曲线来定义其外边界（外围）且未交叉的内封闭曲线定期其内边界（内围）.把带有外边界的曲面和一个基本 NURBS 曲面一样且都没有孔的曲面叫做未剪切曲面.

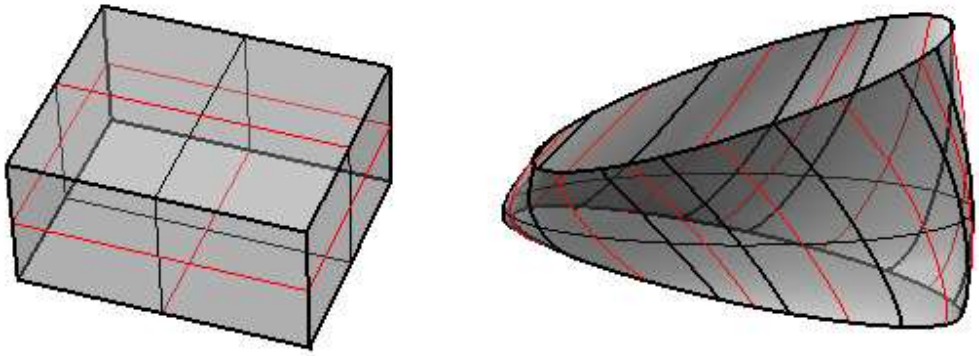


上图左边的是未剪切曲面. 右边为剪切曲面，和左边形状相同仅中间有一个椭圆形的孔. 注意当剪切曲面时是不会影响 NURBS 曲面的结构.

Polysurfaces 复合曲面

一个复合曲面由一个以上的曲面(通常为剪切曲面)组合在一起. 每个曲面都有各自的参数且 UV 方向都不会匹配. 复合曲面与剪切曲面都使用 boundary representation (缩写为 BRep)来表示. 用来基本的描叙曲面、边缘和剪切几何的法向数据以及各自之间的关系. 例如描叙每个面, 它周围的边缘与剪切状态,曲面对应的法向, 以及与邻边面的关系等. 后面我们将描叙 BReps 的变量部分以及有些时候他们是如何互相连接在一起的.

OnBrep 在 OpenNURBS 中是最复杂的数据结构且很难理解, 但好在 Rhino SDK 中有许多的工具和规则函数来帮助我们创建和操作 BReps.



15.3 OpenNURBS Objects Hierarchy 对象架构

SDK 帮助文件列出了所有分类架构. 下面是一个关于创建与操作几何对象子层构架的解析图. 当你在编写脚本的时候你会用到的. 想了解更多的相关细节请参考 SDK 帮助文件.

OnObject (所有 Rhino 分类都源自 OnObject)

- **OnGeometry** (*分类来源或包涵 OnObject*)
    - OnPoint
        - OnBrepVertex
        - OnAnnotationTxtDot
    - OnPointGrid
    - OnPointCloud
    - OnCurve (*abstaract class)*
        - OnLineCurve
        - OnPolylineCurve
        - OnArcCurve
        - OnNurbsCurve
        - OnCurveOnSurface
        - OnCurveProxy
            - OnBrepTrim
            - OnBrepEdge
    - OnSurface (*abstract classt*)

        - OnPlaneSurface
        - OnRevSurface
        - OnSumSurface
        - OnNurbsSurface
        - OnProxySurface
            - OnBrepFace
            - OnOffsetSurface
    - OnBrep
    - OnMesh
    - OnAnnotation

- **Points and Vectors** *(not derived from OnGeometry)*
    - On2dPoint (good for parameter space points)
    - On3dPoint
    - On4dPoint (good for representing control points with x,y,z and w for weight)
    - On3dVector

- **Curves** *(not derived from OnGeometry)*
    - OnLine
    - OnPolyline (is actually derived from OnPointArray)
    - OnCircle
    - OnArc
    - OnEllipse
    - OnBezierCurve

- **Surfaces** *(not derived from OnGeometry)*
    - OnPlane
    - OnSphere
    - OnCylinder
    - OnCone
    - OnBox
    - OnBezierSurface

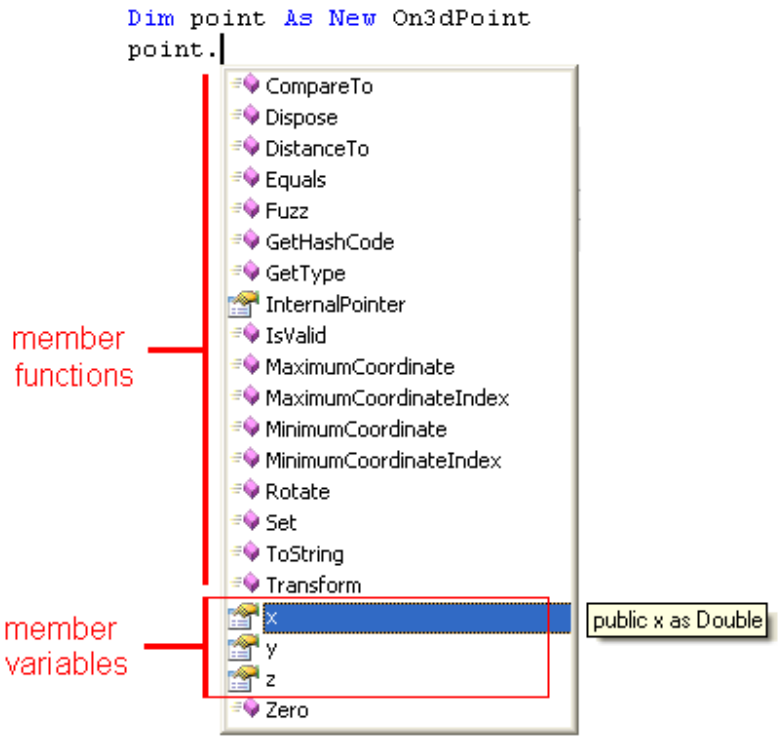
- **Miscellaneous**
    - OnBoundingBox (For objects bounding box calculation)
    - OnInterval (Used for curve and surface domains)
    - OnXform (for transforming geometry objects: move, rotate, scale, etc.)
    - OnMassProperties (to calculate volume, area, centroid, etc)

15.4 Class structure 分类结构

一个典型的分类结构(用户定义数据结构)由四个部分组成:

- **Constructor 构造函数**: 常用于创建举例类.
- **Public member variables 公用变量部分**: 是一种数据存储类. OpenNURBS 变量部分通常开始使用"**m_**"来迅速隔离.
- **Public member functions 公用函数**: 包括所用来创建、更新和操作变量部分或是执行功能类的所有函数.
- **Private members 隐秘部分**: 这是工具类函数与内部使用变量函数.

如果你想彻底的了解一个分类,通过 auto-complete 特征你可以看到分类中的所有函数部分与可变量部分. 注意翻转任何一个函数或变量时,函数的签名都会显示出来. 一旦你开始输入参数至函数, auto-complete 将会显示你所键入的参数的位置. 这是一个用来测定每级分类变量函数的最好办法. 这里有一个 On3dPoint 分类的范例:



根据分类情况可以一步或多步从当前分类复制数据至一个新分类. 例如, 创建一个新的 On3dPoint 分类然后复制当前点的数据至新的分类. 方法如下:

```
Use the constructor when you instantiate an instance of the point class
Dim new_pt as New On3dPoint( input_pt )

Use the "= operator" if the class provides one
Dim new_pt as New On3dPoint
new_pt = input_pt

You can use the "New" function if available
Dim new_pt as New On3dPoint
```

```
new_pt.New( input_pt )

There is also a  "Set" function sometimes
Dim new_pt as New On3dPoint
new_pt.Set( input_pt )

Copy member variables one by one.  A bit exhaustive method
Dim new_pt as New On3dPoint
new_pt.x =  input_pt.x
new_pt.y =  input_pt.y
new_pt.z =  input_pt.z

OpenNURBS geometry  classes provide "Duplicate" function that is very efficient to use
Dim new_crv as New OnNurbsCurve
new_crv = input_crv.DuplicateCurve()
```

15.5 Constant vs Non-constant Instances 常量与非常量举例

Rhino .NET SDK 提供两个分类的设置. 第一个是常量且通常前面都会放置一个"I"来命名; 例如 IOn3dPoint. 相对应的非常量分类将会需要使用不带"I"的相同名字;例如 On3dPoint. 你可以复制一个常量分类或看到一个可变量部分与一些函数，但你不能修改其可变量.

Rhino .NET SDK 是基于 Rhino C++ SDK 开发的. C++ 编程语言提供及时传递变量分类的能力切允许使用 SDK 函数.另一方面, DotNET 就没有这个概念且因此每个都有两个不同版本的分类.

15.6 Points and Vectors 点与矢量

有许多用来存储和操作点与矢量的分类. 例如精确的复制点.有三种不同类型的点:

Class name	Member variables	Notes
On2dPoint	**x** as Double **y** as Double	主要使用空间点参数. 分类名字中带有 "**d**" 代表双精度移动点编号. 其他名字带有"**f**"的分类使用单精度.
On3dPoint	**x** as Double **y** as Double	通常在 3 维坐标空间使用描述的点
On4dPoint	**x** as Double **y** as Double **z** as Double **w** as Double	用于控制点，控制点除了具有三维信息之外还带有权重信息.

点与矢量的操作包括:

Vector Addition:

```
Dim add_v As New On3dVector = v0 + v1
```

Vector Subtraction:

```
Dim subtract_vector As New On3dVector = v0 – v1
```

Vector between two points:

```
Dim dir_vector As New On3dVector = p1 – p0
```

Vector dot product (if result is positive number then vectors are in the same direction):

```
Dim dot_product As Double = v0 * v1
```

Vector cross product (result is a vector normal to the 2 input vectors)

```
Dim normal_v As New On3dVector = OnUtil.ON_CrossProduct( v0, v1 )
```

Scale a vector:

```
Dim scaled_v As New On3dVector = factor * v0
```

Move a point by a vector:

```
Dim moved_point As New On3dPoint = org_point + dir_vector
```

Distance between 2 points:

```
Dim distance As Double = pt0.DistanceTo( pt1)
```

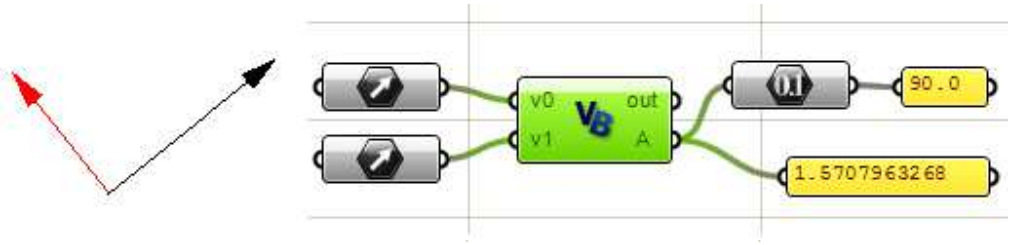
Get unit vector (set vector length to 1):

```
v0.Unitize()
```

Get vector length:

```
Dim length As Double = v0.Length()
```

下面的范例显示如何计算两个矢量方向之间的夹角.



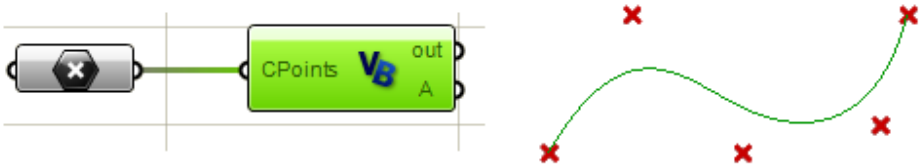

```
Sub RunScript(ByVal v0 As On3dVector, ByVal v1 As On3dVector)

    ' Unitize the input vectors
    v0.Unitize()
    v1.Unitize()
    Dim dot As Double = OnUtil.ON_DotProduct(v0, v1)

    ' Force the dot product of the two input vectors to
    ' fall within the domain for inverse cosine, which
    ' is -1 <= x <= 1. This will prevent runtime
    ' "domain error" math exceptions.
    If (dot < -1.0) Then dot = -1.0
    If (dot > 1.0) Then dot = 1.0

    A = System.Math.Acos(dot)

End Sub
```

15.7 OnNurbsCurve

去创建一个 nurbs 曲线，你需要提供以下数据

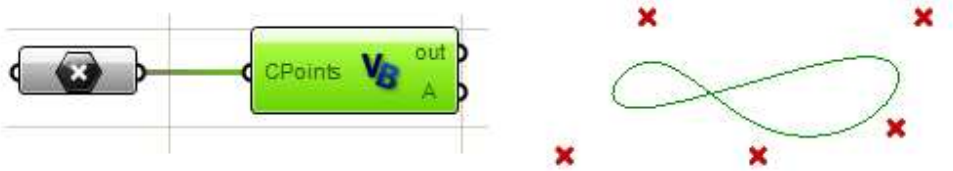
- **Dimension** 尺寸，一般为 3。
- **Order** 秩序：曲线阶数+1。
- **Control Point** 控制点（点序列）
- **Knot Vector** 节点向量（数字序列）
- **Curve Type** 曲线类型 (固定性的或周期性的).

节点向量是可以通过一些函数来帮助创建的，所以基本上我们所需要决定的只有阶数以及一列控制点就可以了。如下所示即是创建固定曲线的范例。




```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim nc As New OnNurbsCurve

  'Create open (Clamped) Nurbs Curve
  nc.CreateClampedUniformNurbs(dimension, order, CPoints.ToArray())

  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

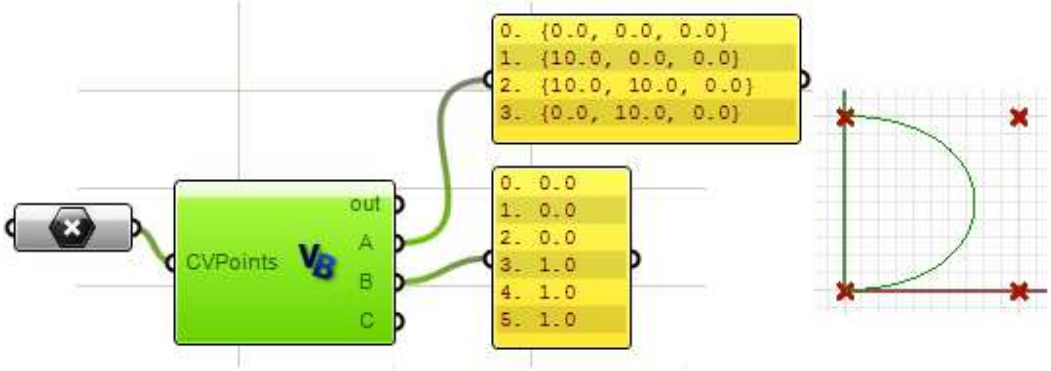
对于平滑闭合曲线来说，应该创建周期性曲线。如下所示即为创建周期性曲线的例子，与上例使用相同的输入控制点和曲线阶数。



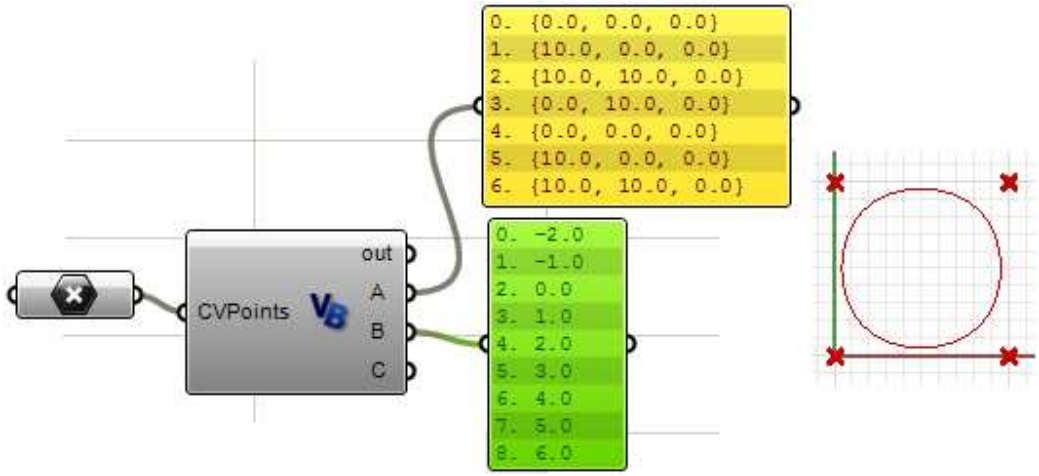
```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim nc As New OnNurbsCurve

  'Create closed (Periodic) Nurbs Curve
  nc.CreatePeriodicUniformNurbs(dimension, order, CPoints.ToArray())

  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

clamped vs periodic NURBS curve

固定曲线(clamped curve)通常是开放的，端点与控制点重合。周期曲线(Periodic curve)是光滑闭合曲线。要理解他们二者的区别最好的方法是通过对控制点的比较。
如下所示，运算器创建了固定 NURBS 曲线并显示了输出值：

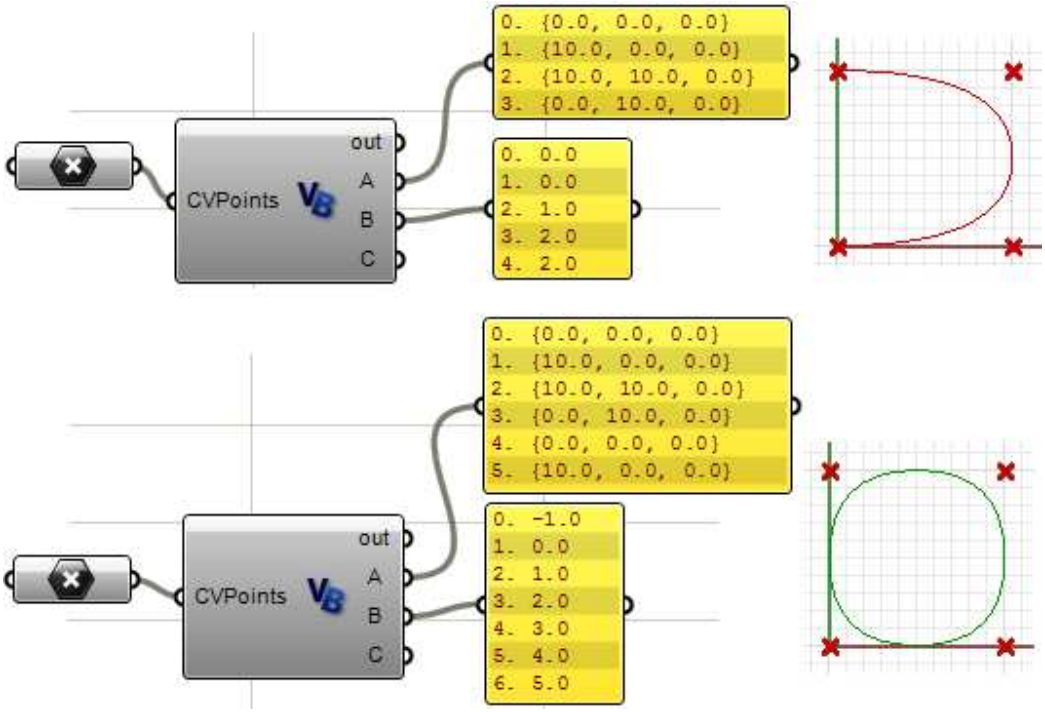


以下为使用相同输入值（控制点与曲线阶数）的周期曲线：



注意周期曲线把四个输入值变成了 7 个控制点（4+阶数），而固定曲线只使用 4 个控制点。周期曲线的节点向量使用简单节点而固定曲线的起点和终点都有充分的多样性。

以下是阶数为 2 的曲线展示的相似例子。你可能已经猜到，周期曲线的控制点和节点数量会随曲线阶数的改变而改变。

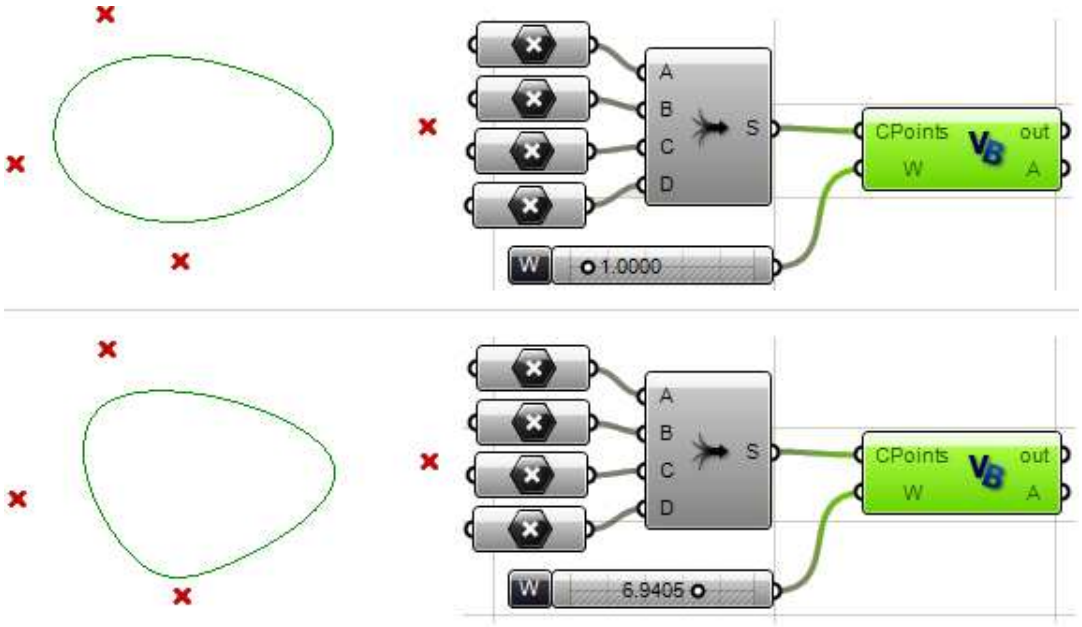


以下是上例中使用控制点向量（CV）点和节点来操纵的代码。

```
'Output control points
Dim count As Double = nc.CVCount()
Dim i As Integer
Dim cvs As New List( Of On3dPoint )
For i = 0 To count - 1
  Dim cv As New On3dPoint(0, 0, 0)
  nc.GetCV(i, cv)
  cvs.Add(cv)
Next

'Output knots
Dim knots As New List( Of Double )
count = nc.KnotCount()
For i = 0 To count - 1
  knots.Add(nc.Knot(i))
Next
```

weights 权重

控制点的权重统一在 nurbs 曲线中设为 1，但是这个数值可以在合理的 nurbs 曲线范围内改变。以下的例子就展示了如何在 Gh 中交互修改控制点的权重。



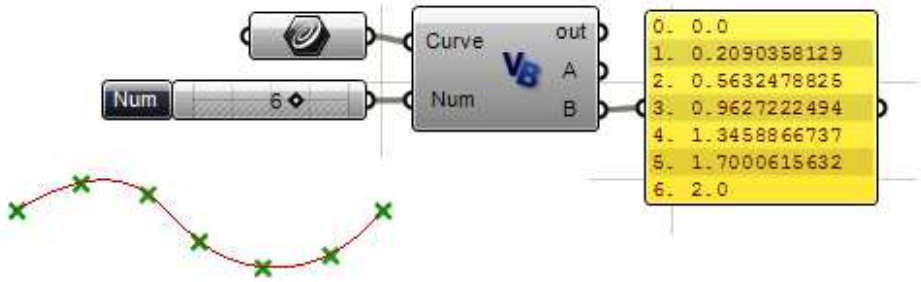

```
Sub RunScript(ByVal CPoints As List(Of On3dPoint), ByVal W As Double)

  Dim i As Integer

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim cv_count As Integer = CPoints.Count
  Dim nc As New OnNurbsCurve
  nc.CreatePeriodicUniformNurbs(dimension, order, CPoints.ToArray())
  nc.MakeRational()

  'Assign weights
  Dim cv As New On3dPoint
  For i = 0 To cv_count - 1
    nc.GetCV(i, cv)
    cv = cv * W
    nc.SetCV(i, cv)
    nc.SetWeight(i, W)
  Next


  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

分隔 NURBS 曲线

分割一条曲线为许多片断包括以下步骤：

- 找出曲线域，即参数空间间隔。
- 列出等分曲线的参数。
- 在三维曲线中找到点。

接下来的例子就展示了如何做到以上步骤。注意这里包含了一个 RhUtil name space 下的全局函数，它可以直接使用一些片断或弧长来分割曲线，我们接下来会图解说明。




```
Sub RunScript(ByVal Curve As OnCurve, ByVal Num As Integer)

  Dim min As Double = Curve.Domain().Min()
  Dim max As Double = Curve.Domain().Max()

  'Find the step value
  Dim step_value As Double = (max - min) / (Num - 1)

  Dim Points As New List( Of on3dPoint )

  Dim t_list(Num) As Double
  For i As Integer = 0 To Num
    t_list(i) = i / Num
  Next

  If (Curve.GetNormalizedArcLengthPoints(t_list, t_list)) Then
    For i As Integer = 0 To Num
      Dim pt As On3dPoint = Curve.PointAt(t_list(i))
      Points.Add(pt)
    Next
  End If


  A = Points
  B = t_list

End Sub
```

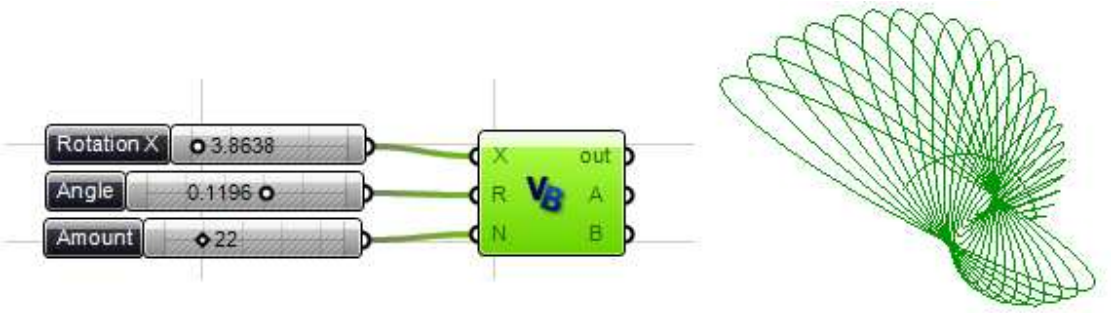
15.8 非 OnCurve 衍生的曲线类

虽然所有的曲线都可以用 NURBS 曲线来表示，有时用其他曲线几何体来表示曲线也是有用的。原因之一是他们的数学表示方法理解起来比 NURBS 简单且更有代表性。
相对而言，得到非 OnCurve 衍生出来 NURBS 曲线容易一些。基本上你需要转变成对应的一类。以下表格显示了这种对应。

Curves Types	OnCurve Derived Types
OnLine	OnLineCurve
OnPolyline	OnPolylineCurve
OnCircle	OnArcCurve or OnNurbsCurve (use GetNurbsForm() member function)
OnArc	OnArcCurve or OnNurbsCurve (use GetNurbsForm() member function)
OnEllipse	OnNurbsCurve (use GetNurbsForm() member function)

OnBezierCurve	OnNurbsCurve (use GetNurbsForm() member function)

以下是使用 OnEllipse 和 OnPolyline 类的实例

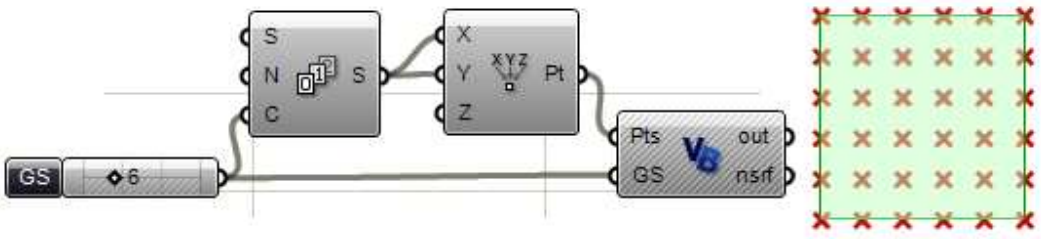



```
Sub RunScript(ByVal X As Object, ByVal R As Object, ByVal N As Object)
  'Declare a new list of OpenNURBS circles
  Dim c_list As New List(Of OnEllipse)

  'Declare list of lines
  Dim p_list As New On3dPointArray

  For i As Int32 = 1 To N
    'Declare a new circle
    Dim c As New OnEllipse(OnUtil.On_xy_plane, i / 2, i)
    'Rotate the circle
    C.Rotate(R * i, New On3dVector(0, 1, 0), New On3dPoint(X, 0, 0))
    'Add the circle to the list
    c_list.Add(c)
    'Add center point
    p_list.Append(C.Center())
  Next

  Dim polyline As New OnPolyline(p_list)
  'Assign the list to the output value A
  A = c_list
  'Assign polyline to output value B
  B = polyline
End Sub
```

15.9 OnNurbsSurface

与我们讨论过的 OnNurbsCurve 类相似，创建 OnNurbsSurface 你需要知道：

- 尺寸，一般为 3。
- 在 u 轴和 v 轴的秩序：曲线阶数+1。
- 控制点
- 在 u 轴和 v 轴的节点向量
- 曲面类型 (固定性的或周期性的).

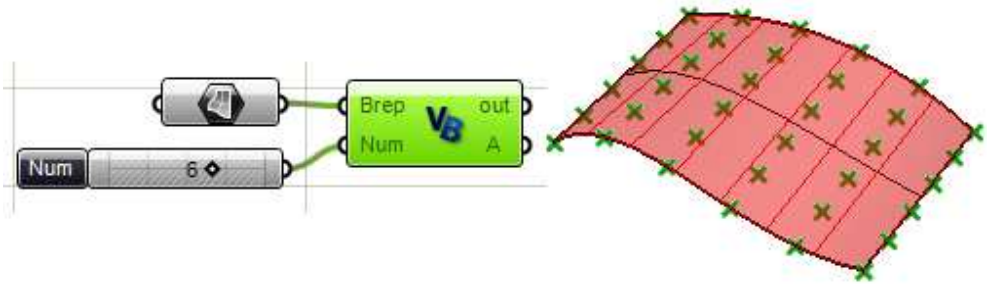



```
Sub RunScript(ByVal Pts As List(Of On3dPoint), ByVal GS As Integer)
  'Create a grid of points
  Dim Grid As New ArrayList()
  'Call grid function
  Call CreateGrid(Pts, Grid, GS)
  'Call create nurbs surface function
  Dim ns As OnNurbsSurface
  ns = CreateNS(Grid, GS)

  'Assign mid point to output
  nsrf = ns

End Sub
Sub CreateGrid( ByVal Pts As List(Of On3dPoint),
                ByRef Grid As ArrayList, ByVal GS As Integer )
  Dim i As Integer
  Dim j As Integer
  For i = 0 To Pts.Count() - 1 Step GS
    'Declare a row of points
    Dim Row As New List( Of On3dPoint )
    For j = i To i + GS - 1
      'Get a reference od the point
      Dim pt As On3dPoint
      pt = Pts(j)
      'Add point to the row
      Row.Add(pt)
    Next
    'Add row to the grid
    Grid.Add(Row)
  Next
End Sub
```

```
Function CreateNS(ByVal cvpoints As ArrayList,
                  ByVal GS As Integer) As OnNurbsSurface
  Const Degree As Integer = 3

  'Make the surface
  Dim orderU As Integer = Degree + 1
  Dim orderV As Integer = Degree + 1

  Dim ns As New OnNurbsSurface
  ns.Create(3, False, orderU, orderV, GS, GS)

  'Add cv points
  Dim i As Integer
  Dim j As Integer
  Dim pt As On3dPoint
  For i = 0 To GS - 1
    For j = 0 To GS - 1
      pt = cvpoints(i)(j)
      ns.SetCV(i, j, pt)
    Next
  Next

  'Set knots for open surface
  ns.MakeClampedUniformKnotVector(0)
  ns.MakeClampedUniformKnotVector(1)

  CreateNS = ns
End Function
```

另一个典型的例子是分割面域。接下来的例子就把面域在两个方向上等分成了许多点（点的数量必须大于 1 否则无意义）以下为步骤：

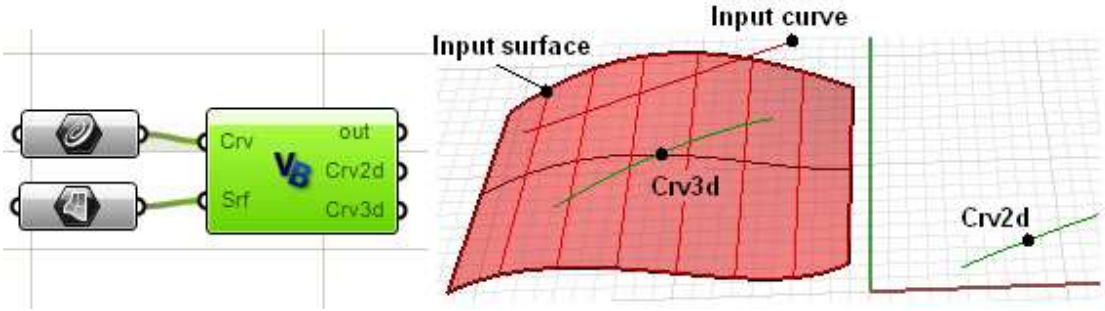
- 规格化面域（设面域间隔为 0-1）
- 利用点数计算步数
- 使用 u 轴和 v 轴的参数用嵌套递归来计算面上的点。



```
Sub RunScript(ByVal Brep As OnBrep, ByVal Num As Integer)
  'Find step - Num must be > 1
  Dim StepValue As Double = 1 / (Num - 1)

  Dim nSrf As New OnNurbsSurface
  nSrf = Brep.Face(0).NurbsSurface

  'Normalize domain in u and v directions
  nSrf.SetDomain(0, 0, 1)
  nSrf.SetDomain(1, 0, 1)

  Dim Points As New List( Of on3dPoint )
  Dim i As Double = 0
  Dim j As Double = 0
  For i = 0 To 1 Step StepValue
    For j = 0 To 1 Step StepValue
      Dim Pt As New On3dPoint
      Pt = nSrf.PointAt(i, j)
      Points.Add(Pt)
    Next
  Next

  A = Points
End Sub
```

OnSurface 类拥有很多函数，这些对操纵和控制面的工作非常有用。接下来的例子就展示了如何拖拽一条曲线成曲面。

在 Gh 脚本运算器中有两个输出。第一个是相对于面域的参数空间曲线（三维曲线在 xy 平面的水平展示）。第二个是曲线在三维空间中。我们通过“推进”二维参数空间曲线到曲面上来得到三维曲线。



```
Sub RunScript(ByVal Crv As OnCurve, ByVal Srf As OnBrep)

  'Get pulled curve in 2D parameter space
  Dim pull_crv As OnCurve
  pull_crv = Srf.m_S(0).Pullback(Crv, doc.AbsoluteTolerance())

  'Get the pulled curve in 3D space
  Dim push_crv As OnCurve
  push_crv = Srf.m_S(0).Pushup(pull_crv, doc.AbsoluteTolerance())

  'Output both curves
  Crv2d = pull_crv
  Crv3d = push_crv

End Sub
```

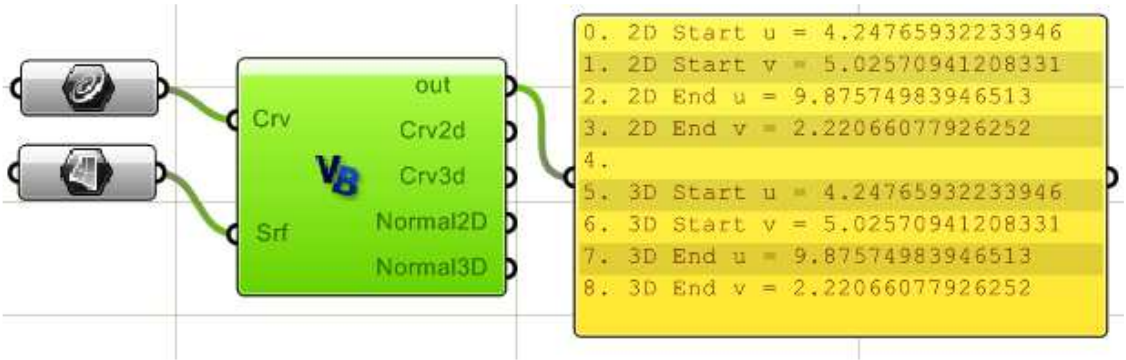
运用上面的例子，我们会计算出被拖拽的曲线起点与终点的标准向量。以下两种方法皆可达成：

- 使用拖拽的二维曲线的起点和终点，这将成为参数空间里曲面的起点和终点。
- 或者使用推进三维曲线终点，寻找离曲面最近的点，然后使用结果参数去寻找曲面标准。

```
Sub GetEndNormals2D(ByVal crv2d As OnCurve,
                    ByVal srf As OnSurface,
                    ByRef EndVectors2D As List(Of On3dVector ))
  Dim start_normal As On3dVector
  Dim end_normal As On3dVector

  'find start and end points in parameter space
  Dim start2d As New On2dPoint
  start2d = crv2d.PointAtStart()
  Dim end2d As New On2dPoint
  end2d = crv2d.PointAtEnd()

  'Output parameters
  'Surface parameters are the x and y of the 2d curve end points
  Print("2D Start u = " & start2d.x)
  Print("2D Start v = " & start2d.y)
  Print("2D End u = " & end2d.x)
  Print("2D End v = " & end2d.y)
  Print("")

  'Call surface normal function
  start_normal = srf.NormalAt(start2d.x, start2d.y)
  end_normal = srf.NormalAt(end2d.x, end2d.y)

  EndVectors2D.Add(start_normal)
  EndVectors2D.Add(end_normal)
End Sub
```

```
Sub GetEndNormals3D(ByVal crv3d As OnCurve,
                    ByVal srf As OnSurface,
                    ByRef EndVectors3D As List(Of On3dVector ))
  'Declare start and end normal
  Dim start_normal As On3dVector
  Dim end_normal As On3dVector

  'Find start and end points in parameter space
  Dim start3d As New On3dPoint
  start3d = crv3d.PointAtStart()
  Dim end3d As New On3dPoint
  end3d = crv3d.PointAtEnd()

  'Declare parameters
  Dim u As Double
  Dim v As Double

  'Get surface closest point
  srf.GetClosestPoint(start3d, u, v)
  start_normal = srf.NormalAt(u, v)

  'Output start parameters
  Print("3D Start u = " & u)
  Print("3D Start v = " & v)

  srf.GetClosestPoint(end3d, u, v)
  end_normal = srf.NormalAt(u, v)

  'Output end parameters
  Print("3D End u = " & u)
  Print("3D End v = " & v)

  EndVectors3D.Add(start_normal)
  EndVectors3D.Add(end_normal)
End Sub
```

下图所示的运算器表示了使用以上两函数输出的终点参数值。注意两种方法都如我们所料生成了相同的参数。

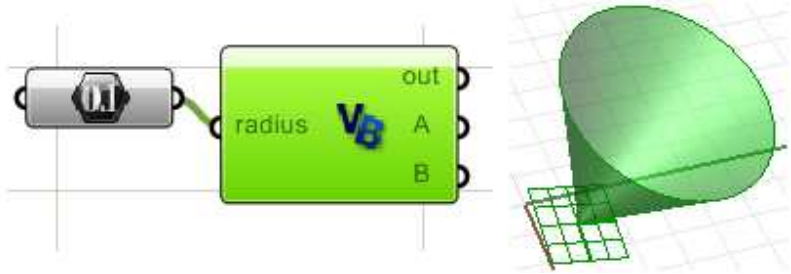


15．10 非由 OnSurface 衍生的曲面类

OpenNURBS 提供非由 OnSurface 衍生的曲面类。它们是有效的数学曲面定义并且可以转换成 OnSurface 衍生类型去用于采取 OnSurface 的函数中，

Basic Surface Types **基本曲面类型**	**OnSurface derived Types** **OnSurface 衍生类型**
OnPlane	OnPlaneSurface or OnNurbsSurface (use OnPlane.GetNurbsForm() function)
OnShpere	OnRevSurface or OnNurbsSurface (use OnShpere.GetNurbsForm() function)
OnCylinder	OnRevSurface or OnNurbsSurface (use OnCylinder.GetNurbsForm() function)
OnCone	OnRevSurface or OnNurbsSurface (use OnCone.GetNurbsForm() function)
OnBezierSurface	OnNurbsSurface (use GetNurbsForm() member function)

Here is an example that uses OnPlane and OnCone classes:
以下是运用 OnPlane 和 OnCone 类的例子：



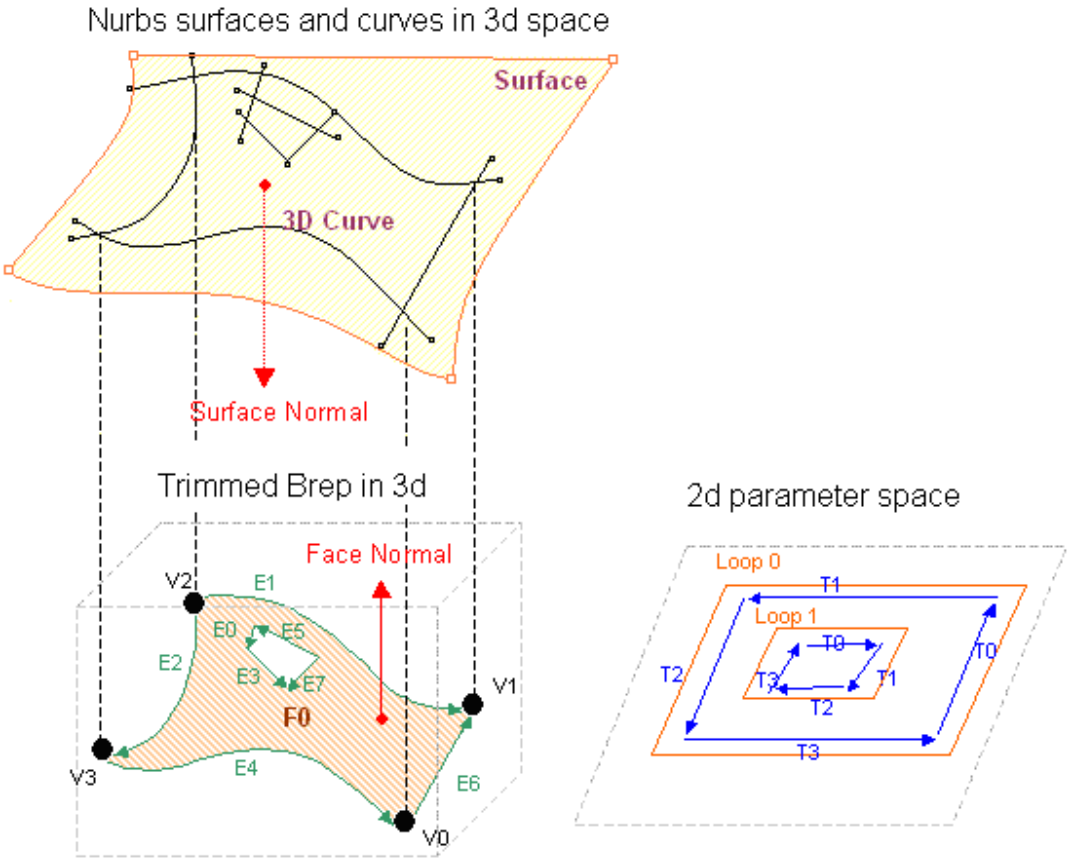

```
Sub RunScript(ByVal radius As Double)
  'Create a plane from origin and normal
  Dim plane As New OnPlane
  Dim origin As New On3dPoint(1, 1, 0)
  Dim normal As New On3dVector(1, 1, 3)
  plane.CreateFromNormal(origin, normal)

  'Define height value
  Dim height As Double = 5
  'Create cone
  Dim cone As New OnCone(plane, height, radius)

  'Assign output parameter
  A = cone
  B = plane
End Sub
```

15．11 OnBrep

边界表示(B-Rep)是用物体边界面来明确表示它们。你可以把 OnBrep 当作是三个不同的部分：

- 几何：三维 nurbs 曲线和曲面的几何。也可以是参数空间或修剪曲线的二维曲线。
- 三维拓补结构：面，边界和顶点。每个面参考一个 nurbs 曲面。每个面也知道所有属于该面的循环。边界参考三维曲线。每个边界有一个使用该边界和两个顶点的修剪列表。顶点参考空间中的三维点。每个顶点也同样有一个边界列表，这些边界就在端点间。
- 二维拓补结构：二维参数空间表示面和边界。在参数空间中，二维修剪曲线呈顺时针或逆时针取决于他们是表面外循环还是内循环的一部分。每一个有效面有且仅有一个外循环但可以有足够多的内循环（洞）。每个修剪参考一个边界，两个顶点，一个循环和二维曲线。

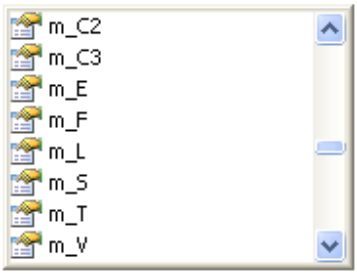
以下图表就表示了这三个部分以及他们是如何互相联系的。顶端的部分表示了下层的三维 nurbs 曲面和曲线的几何体用一个洞来定义了 brep 面，中间的是三维拓补结构，包括 brep 面，外边界，内边界（bounding a hole）和顶点。然后是有着修剪和循环的参数空间



OnBrep 成员变量

OnBrep 类成员变量包括所有三维与二维的几何与拓补信息。一旦你创造出一例 brep 类，你就可以查阅所有元函数和变量。以下图片显示出了元函数的自我完成。表中列出了数据类型及描述。

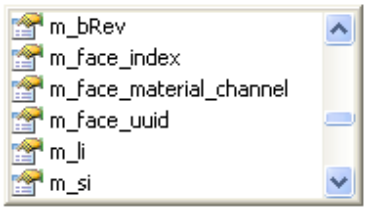
```
Dim brep As New OnBrep
brep.
```



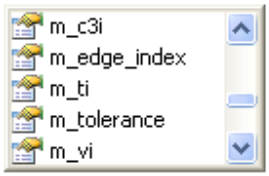
Topology members: describe relationships among different brep parts **类型成员：描述不同 brep 部分间的关系**	
OnBrepVertexArray **m_V**	Array of brep vertices (OnBrepVertex)
OnBrepEdgeArray **m_E**	Array of brep edges (OnBrepEdge)
OnBrepTrimArray **m_T**	Array of brep trims (OnBrepTrim)
OnBrepFaceArray **m_F**	Array of brep faces (OnBrepFace)
OnBrepLoopArray **m_L**	Array of loops (OnBrepLoop)
Geometry members: geometry data of 3d curves and surfaces and 2d trims **几何成员：三维曲线和曲面及二维修剪的几何数据**	
OnCurveArray **m_C2**	Array of trim curves (2D curves)
CnCurveArray **m_C3**	Array of edge curve (3D curves)
ONSurfaceArray **m_S**	Array of surfaces

注意每个 OnBrep 元函数基本上是其他类的一个阵列。举例来说，m_F 是 OnBrepFace 的阵列参考。OnBrepFace 是由 OnSurfaceProxy 衍生出的一类，并有自己的元函数和变量。以下是 OnBrepFace, OnBrepEdge, OnBrepVertex, OnBrepTrim and OnBrepLoop classes 的成员变量。

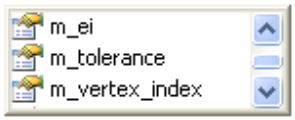
```
Dim brep_face As New OnBrepFace
brep_face.
```



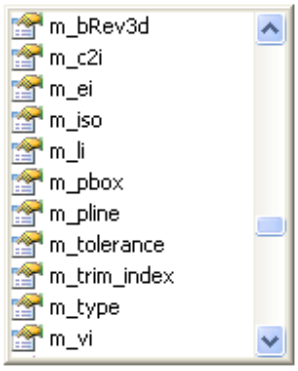

```
Dim brep_edge As New OnBrepEdge
brep_edge.
```



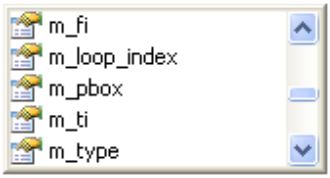
```
Dim brep_vertex As New OnBrepVertex
brep_vertex.
```



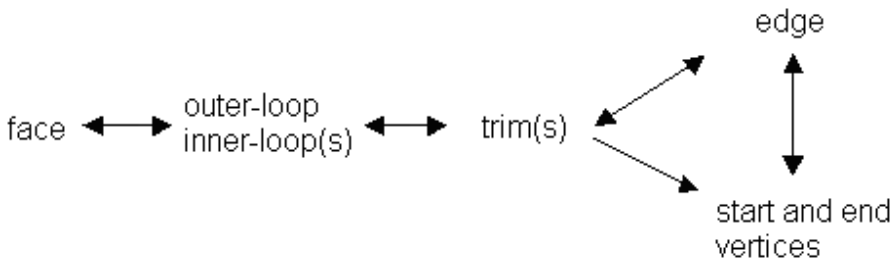

```
Dim brep_trim As New OnBrepTrim
brep_trim.
```



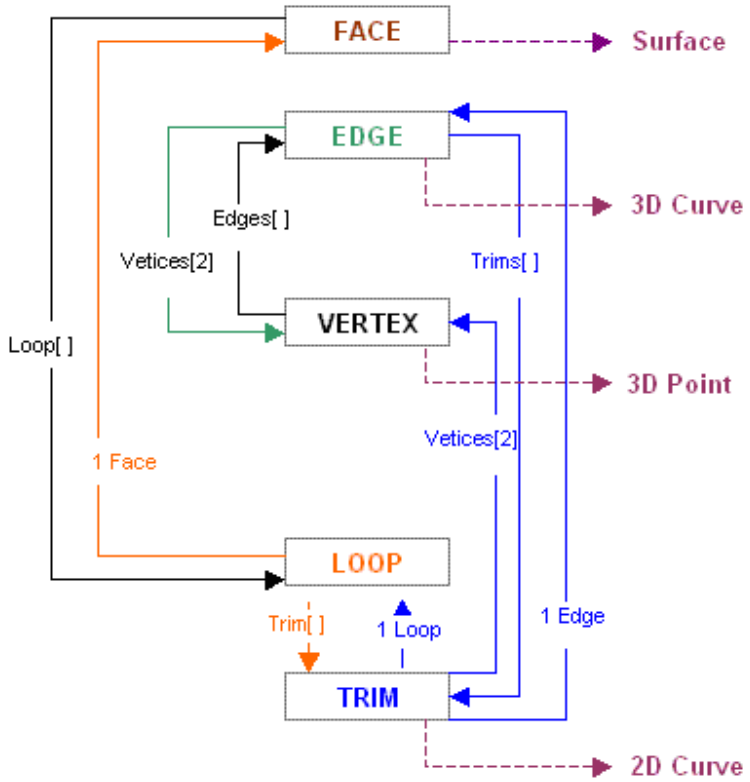

```
Dim brep_loop As New OnBrepLoop
brep_loop.
```



以下图表表示了 OnBrep 成员变量及他们如何互相参照。你可以使用其中信息得到任何特定部分的 brep。举例来说，每个面知道它的循环，每个循环有它的修剪列表，通过每个修剪你可以得到它的边缘和两个顶点，等等。



以下是另一详图，表示了 brep 各部分是如何衔接的以及如何从一部分到另一部分



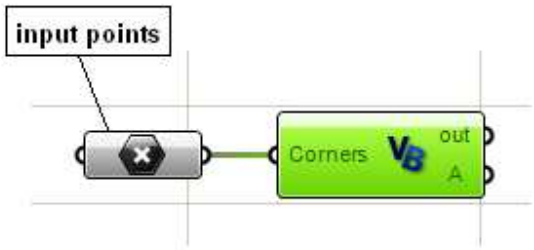
我们会用接下来的几个例子展现如何去创建一个 OnBrep，操纵不同的部分，抽取 brep 信息。我们同样会展示如何使用一些类函数以及全局函数。

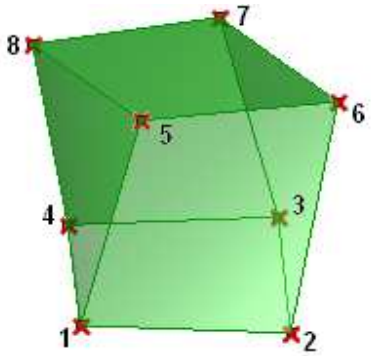
创建 OnBrep

有多种方法创建一个新的实例 OnBrep 类：

- 复制已存的 brep
- 从一个已存的 brep 中复制或抽取。
- 使用创建函数使 OnSurface 作为一个输入参数。
- 这里有五个使用不同曲面类型的超载函数：
  - 来自 SumSurface
  - 来自 RevSurface
  - 来自 PlanarSurface
  - 来自 OnSurface
- 使用全局效用函数
  - 来自 OnUtil such as ON_BrepBox, ON_BrepCone, etc.
  - 来自 RhUtil such as RhinoCreatEdgeurface or RhinoSweep1 among others.

这是一个从角点创建 brep 立方体的例子





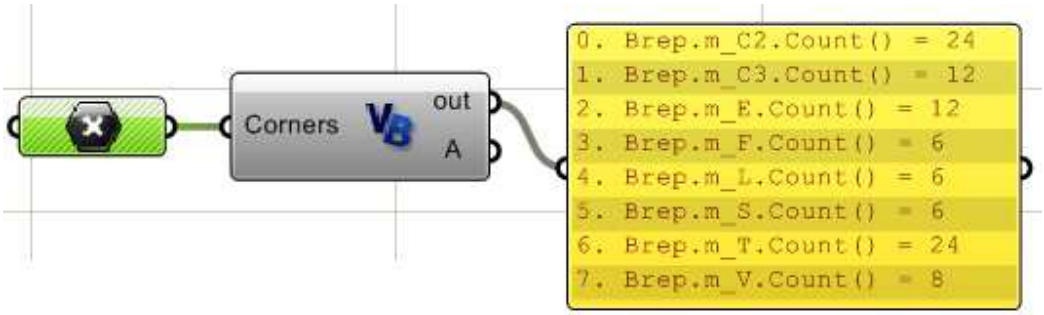

```
Sub RunScript(ByVal Corners As List(Of On3dPoint))

  ' Build the brep from corners
  Dim Brep As OnBrep = OnUtil.ON_BrepBox(Corners.ToArray())

  A = Brep
End Sub
```

操纵 OnBrep 数据

以下例子表示如何抽取 brep 立方体顶点

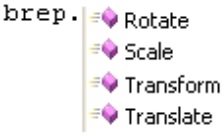
```
Sub RunScript(ByVal Corners As List(Of On3dPoint))

  ' Build the brep from corners
  Dim Brep As OnBrep = OnUtil.ON_BrepBox(Corners.ToArray())

  Dim myCorners As New List( Of On3dPoint )
  Dim v As OnBrepVertex
  Dim i As Integer

  For i = 0 To Brep.m_V.Count() - 1
    'get reference to OnBrepVertex
    v = Brep.m_V(i)

    'Get vertex point (loction)
    Dim pt As New On3dPoint
    pt = v.point

    'Add point to array
    myCorners.Add(pt)
  Next

  A = Brep
  B = myCorners
End Sub
```

以下例子表示在一个 brep 立方体中如何得到几何与拓补部分的数字（面，边界，修剪，顶点等）。



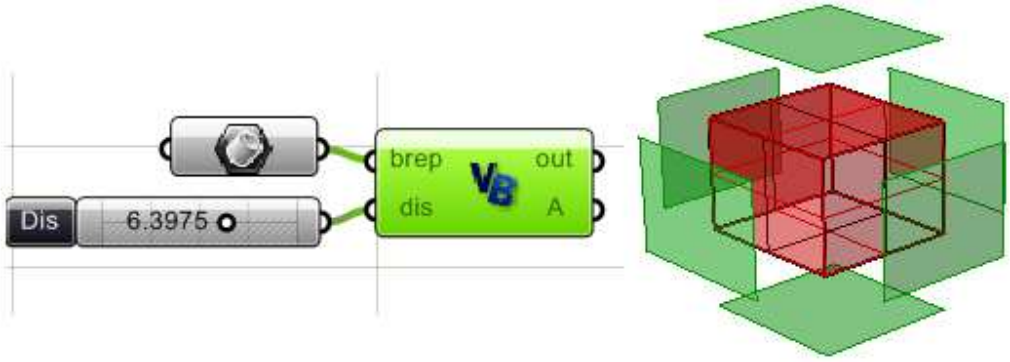
转换 OnBreps

所有从 OnGeometry 衍生而来的类型都继承有四个变换函数。前三个可能是最常用的，分别是旋转，缩放和变形。但是也存在一种通用的“ Trabsform ”函数，采取了以 OnXform 类定义的 4 × 4 变换矩阵。我们会在下一章节中讨论 OnXform。



编辑 OnBrep

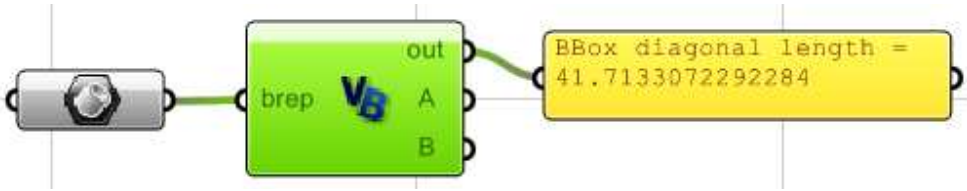
大多数 OnBrep 类元函数是牛人们用来创建和编辑 breps 的工具。尽管如此，还有一些全局函数用来进行布尔运算，相交或分离 breps，我们将会在另一章中举例说明。
McNeel’s wiki DotNET 上有一个很出色的详例，从零开始创建了一个 brep，应该会给你一个从零开始建立一个有效的 OnBrep 的好例子。
这里有一个抽取 OnBrep 面并且通过移动使它们远离 brep 中心的例子。此例使用从中心建立立方体选框（bounding box center）。



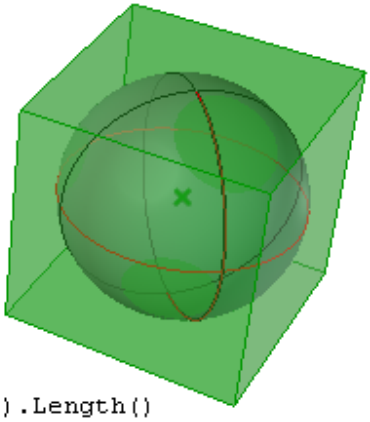
```
Sub RunScript(ByVal brep As OnBrep, ByVal dis As Double)

  Dim faces As New List(Of OnBrep)

  'Loop through brep faces to extract them
  For fi As Integer = 0 To brep.m_F.Count() - 1
    'Decalre new brep
    Dim face As New OnBrep
    face = brep.DuplicateFace(fi, False)

    'Add to faces array
    faces.Add(face)
  Next

  'Find brep bounding box center
  Dim center As New On3dPoint
  center = brep.BoundingBox().Center()

  'Loop through faces and move away from center by dis
  Dim dir As New On3dVector
  For i As Integer = 0 To faces.Count() - 1
    Dim face As OnBrep
    face = faces(i)

    'Find ceneter of each extracted face
    Dim face_center As On3dPoint
    face_center = face.BoundingBox().Center()

    'Find translation vector
    dir = face_center - center
    dir.Unitize()
    dir *= dis

    'Move face away from center
    face.Translate(dir)
  Next

  'Assign output
  A = faces


End Sub
```

其他 OnBrep 元函数

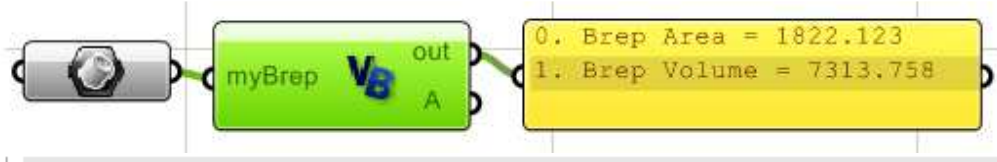
OnBrep 类有许多其它函数，他们有些是从父类中继承的，有些是特殊的 OnBrep 类函数。所有几何类，包括 OnBrep,都有叫做“BoundingBox()”的元函数。OpenNURBSl 类中之一就是 OnBoundingBox，能给出有用的几何信息。如下例子展示了寻找一个 brep bounding box 及它的中心和对角线长度的过程。



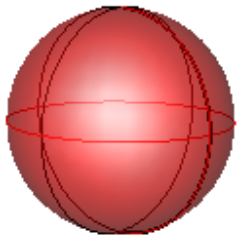

```
Sub RunScript(ByVal brep As OnBrep)

  'Find brep bounding box
  Dim bbox As New OnBoundingBox
  bbox = brep.BoundingBox()

  'Find bounding box ceneter
  Dim center As New On3dPoint
  center = bbox.Center()

  'Print bounding box diagonal length
  Dim length As Double = bbox.Diagonal().Length()
  Print("BBox diagonal length = " & length)

  A = bbox
  B = center

End Sub
```



另一值得研究的领域是体块性质。OnMassProperties 类和一些它的函数如下图所示作一举例。

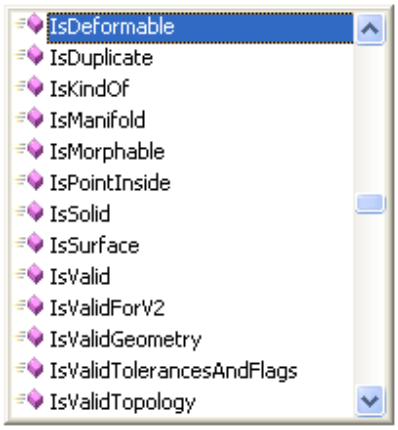



```
Sub RunScript(ByVal myBrep As Object)

  'Find and print brep area
  Dim a_mass As New OnMassProperties
  myBrep.AreaMassProperties(a_mass)
  Dim area As Double = a_mass.Area()
  Print("Brep Area = " & area)

  'Find and print brep volume
  Dim v_mass As New OnMassProperties
  myBrep.VolumeMassProperties(v_mass)
  Dim vol As Double = v_mass.Volume()
  Print("Brep Volume = " & vol)

End Sub
```



还有一些以 “Is”开头的函数经常重新调整一个布尔值（true 或者 false）。他们能查询你正在使用的 brep 情况。举例来说，如果你想知道一个 brep 是否是闭合的复合曲面，就使用 OnBrep.IsSolid()函数。此函数对查询 brep 是否有效或是否含有有效几何体同样适用。以下是一些 OnBrep 类查询函数的列表：

```
Dim brep As New OnBrep
brep.Is…
```



以下例子查询一个已知点是否在 brep 之内：

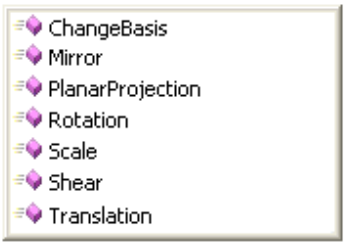
以下是查询点是否在内的编码：

```
Sub RunScript(ByVal brep As OnBrep, ByVal point As On3dPoint)
  'Test if input point is inside brep
  Dim tol As Double = doc.AbsoluteTolerance()
  Dim stricktly_inside As Boolean = True
  Dim is_inside As Boolean

  'Call brep function to test the point
  is_inside = brep.IsPointInside(point, tol, stricktly_inside)

  Print(is_inside)

End Sub
```

15.12 几何变化

OnXform 是用来存储和操纵变换矩阵的阶数。这包括定义（但是不局限于此）一个矩阵来移动，旋转，缩放或修剪对象物体。
OnXform 的 m_xform 是一个 4x4 的双倍精度数矩阵。此 class 还拥有支持矩阵运算的函数，例如反向和变位运算。这里有一些与创建不同变化相关的元函数

```
Dim xform As New OnXform
xform.
```



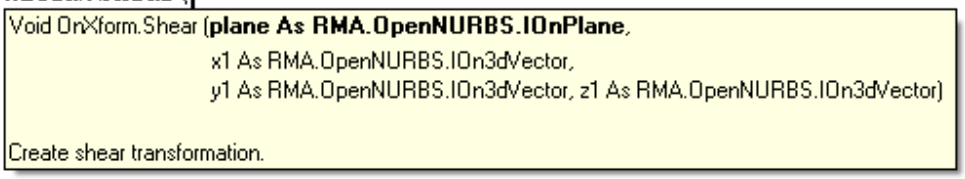
有一个适用于所有函数的自动完成的特征，一旦选择一个函数，自动完成将显示所有的函数重载。如图中所示，无论是三个数字还是一个向量都能被转换。

```
Dim xform As New OnXform
xform.Translation(
```

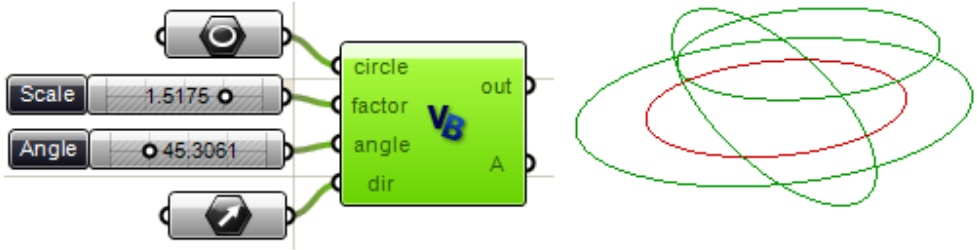
如下有另一些 OnXform 函数：

```
Dim xform As New OnXform
xform.PlanarProjection(
```


```
Dim xform As New OnXform
xform.Shear(
```



接下来的例子将使一个输入的圆输出为三个圆。第一个圆是原始圆缩放后的圆，第二个是旋转后的圆，第三个是转化后的圆。



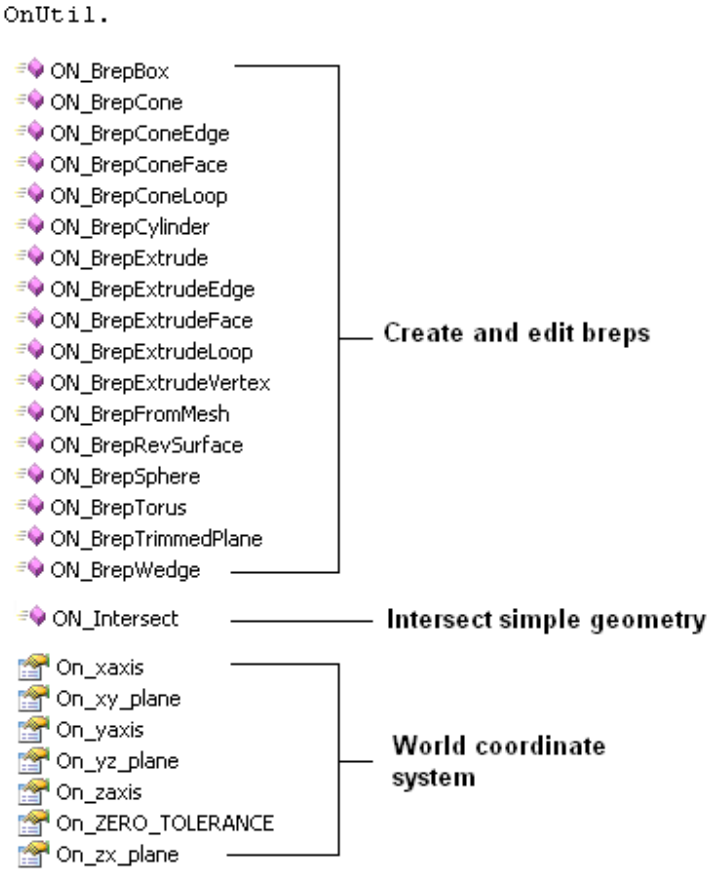

```
Sub RunScript(ByVal circle As OnCircle,
              ByVal factor As Double,
              ByVal angle As Double, ByVal dir As On3dVector)

  Dim circles As New List( Of OnCircle)

  'Scaled circle
  Dim scale As New OnXform
  scale.Scale(OnUtil.On_origin, factor)
  Dim s_circle As New OnCircle(circle)
  s_circle.Transform(scale)
  circles.Add(s_circle)

  'Rotated circle
  Dim rotate As New OnXform
  rotate.Rotation(angle, OnUtil.On_yaxis, OnUtil.On_origin)
  Dim r_circle As New OnCircle(circle)
  r_circle.Transform(rotate)
  circles.Add(r_circle)

  'Moved circle
  Dim move As New OnXform
  move.Translation(dir)
  Dim m_circle As New OnCircle(circle)
  m_circle.Transform(move)
  circles.Add(m_circle)

  'Assign output
  A = circles

End Sub
```

15.13 Global utility functions（全局实用函数）

除了在每一个 CLASS 范围内的元函数，Rhino.NET SDK 还提供了在名为 OnUtil 和 RhUtil 空间下的全局函数。我们将通过使用下列例子来介绍使用这类函数。

OnUtil（OnUtil）

下列是与几何相关的在 OnUtil 之下的可利用的函数的摘要：



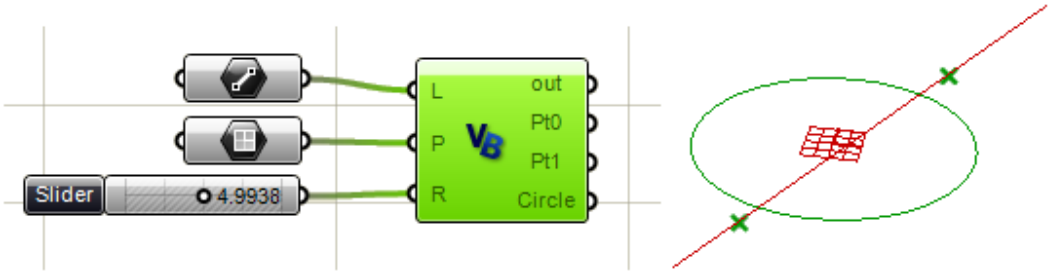
OnUtil intersections（OnUtil 交叉）

ON_Intersect 效用函数有 11 个函数重载。这里有一个列表，内有分割的几何体和返回值（比如：“IOnLine”的首字母“I”意思是一个恒定的例子已经通过）

Intersected geometry	output
IOnLine with IOnArc	Line parameters (t0 & t1) and Arc points (p0 & p1) 线参数(t0 & t1)和弧线点(p0 & p1)
IOnLine with IOnCircle	Line parameters (t0 & t1) and circle points (p0 & p1) 线参数(t0 & t1)和圆点(p0 & p1)
IOnSphere with IOnShere	OnCircle
IOnBoundingBoc with IOnLine	Line parameters (OnInterval)
IOnLine with IOnCylinder	2 points (On3dPoint)
IOnLine with IOnSphere	2 points (On3dPoint)
IOnPlane with IOnSphere	OnCircle
IOnPlane with IOnPlane with IOnPlane	On3dPoint
IOnPlane with IOnPlane	OnLine
IOnLine with IOnPlane	Parameter t (Double)

IOnLine with IOnLine	Parameters a & b (on first and second line as Double)

以下一个例子展示了一条线一个面和一个球体交接的结果：




```
Sub RunScript(ByVal L As OnLine, ByVal P As OnPlane, ByVal R As Double)

  Dim point0 As New On3dPoint
  Dim point1 As New On3dPoint
  Dim circle0 As New OnCircle

  'Declare the sphere
  Dim sphere As New OnSphere(OnUtil.On_origin, R)

  'Intersect line with sphere
  OnUtil.ON_Intersect(L, sphere, point0, point1)

  'Intersect plane with sphere
  OnUtil.ON_Intersect(P, sphere, circle0)

  'Assign output
  Pt0 = point0
  Pt1 = point1
  Circle = circle0
End Sub
```

RhUtil(Rhino 效用)

Rhino 效用有许多与几何相关的函数。列表下的新的展开基于用户的要求。这里是与几何相关的函数的截图：

RhUtil.

Points

- RhinoArePointsCoplanar
- RhinoPointInPlanarClosedCurve
- RhinoProjectPointsToBreps
- RhinoIsPointInBrep
- RhinoIsPointOnFace

Curve

- RhinoConvertCurveToPolyline
- RhinoCurveBrepIntersect
- RhinoCurveFaceIntersect
- RhinoDivideCurve
- RhinoDoCurveDirectionsMatch
- RhinoExtendCrvOnSrf
- RhinoExtendCurve
- RhinoExtendLineThroughBox
- RhinoExtrudeCurveStraight
- RhinoExtrudeCurveToPoint
- RhinoFairCurve
- RhinoFitCurve
- RhinoFitLineToPoints
- RhinoInterpCurve
- RhinoInterpolatePointsOnSurface
- RhinoMakeCubicBeziers
- RhinoMakeCurveClosed
- RhinoMakeCurveEndsMeet
- RhinoMergeCurves
- RhinoOffsetCurve
- RhinoOffsetCurveOnSrf
- RhinoPlanarClosedCurveContainmentTest
- RhinoPlanarCurveCollisionTest
- RhinoProjectCurvesToBreps
- RhinoPullCurveToMesh
- RhinoRebuildCurve
- RhinoRemoveShortSegments
- RhinoRepairCurve
- RhinoShortPath
- RhinoSimplifyCurve
- RhinoSimplifyCurveEnd

Surface

- RhinoChangeSeam
- RhinoCreateSurfaceFromCorners
- RhinoExtendSurface
- RhinoFitSurface
- RhinoIntersectSurfaces
- RhinoMakeG1Surface
- RhinoRailRevolve
- RhinoRebuildSurface
- RhinoRepairSurface
- RhinoRetrimSurface
- RhinoRevolve
- RhinoSrfControlPtGrid
- RhinoSrfPtGrid

Mesh

- RhinoMeshBooleanDifference
- RhinoMeshBooleanIntersection
- RhinoMeshBooleanSplit
- RhinoMeshBooleanUnion
- RhinoMeshBox
- RhinoMeshCone
- RhinoMeshCylinder
- RhinoMeshObjects
- RhinoMeshOffset
- RhinoMeshPlane
- RhinoMeshSphere
- RhinoRepairMesh
- RhinoSplitDisjointMesh
- RhinoUnifyMeshNormals

Brep

- RhinoBooleanDifference
- RhinoBooleanIntersection
- RhinoBooleanUnion
- RhinoBrepCapPlanarHoles
- RhinoBrepClosestPoint
- RhinoBrepGet2dProjection
- RhinoBrepGet2dSection
- RhinoBrepSplit
- RhinoCreate1FaceBrepFromPoints
- RhinoCreateEdgeSrf
- RhinoIntersectBreps
- RhinoJoinBrepNakedEdges
- RhinoJoinBreps
- RhinoMakePlanarBreps
- RhinoMergeAdjoiningEdges
- RhinoMergeBrepCoplanarFaces
- RhinoMergeBreps
- RhinoRepairBrep
- RhinoSplitBrepFace
- RhinoStraightenBrep
- RhPlanarRegionBoolean
- RhPlanarRegionDifference
- RhPlanarRegionIntersection
- RhPlanarRegionUnion
- RhinoSdkLoft
- RhinoSdkLoftSurface
- RhinoSweep1
- RhinoSweep2

Utility

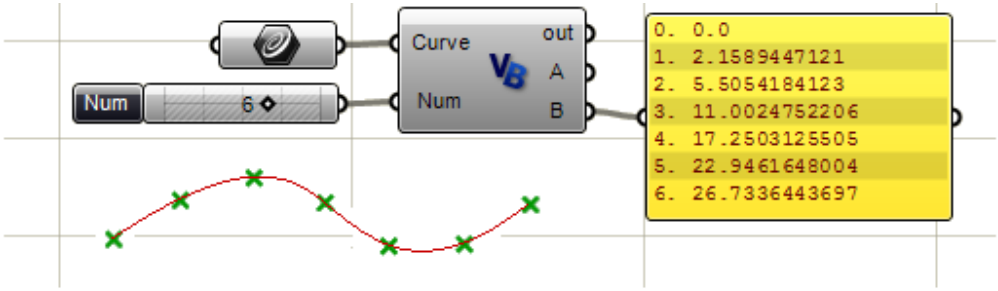
- RhinoActiveCPlane
- RhinoApp
- RhinoFitPlaneToPoints
- RhinoPlaneThroughBox
- RhinoProjectToPlane
- RhinoTriangulate3dPolygon

RhUtil Divide curve（RhUtil 划分曲线）

通过使用 RhUtil.RhinoDivideCurve 效用函数，可以采取曲线上的线段数量或者长度划分曲线。下面是函数参数的分析：

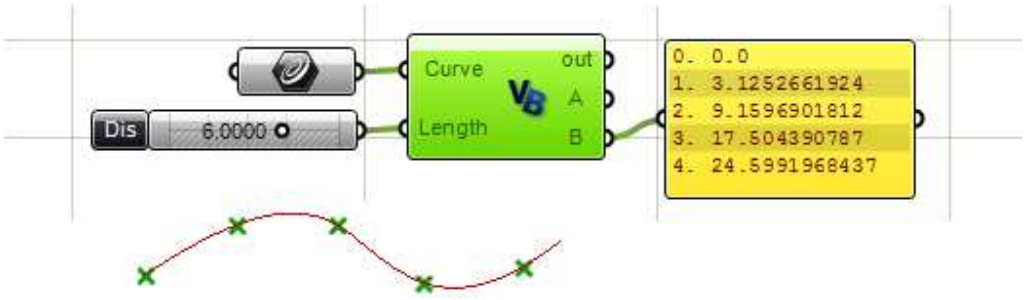
RhinoDivideCurve: 函数名
Curve: 带分割的曲线常量
Num: 片段的数量
Len: 用来除以的曲线长度
False: 反曲线的标志（可设定为 TRUE 或者 FALSE）
True: 包含终点（可设定为 TRUE 或者 FALSE）
crv_p: 分割点列表
crv_t: 曲线上分割点参数列表

通过片段数量来分割曲线的例子：




```
Sub RunScript(ByVal Curve As OnCurve, ByVal Num As Integer)

  Dim crv_p As New On3dPointArray
  Dim crv_t As New Arraydouble

  'A utility function to divide by number or curve length
  RhUtil.RhinoDivideCurve(Curve, Num, 0, False, True, crv_p, crv_t)

  A = crv_p
  B = crv_t

End Sub
```

通过弧线长度来分割曲线的例子：



```
Sub RunScript(ByVal Curve As OnCurve, ByVal Len As Double)

  Dim crv_p As New On3dPointArray
  Dim crv_t As New Arraydouble

  'A utility function to divide by number or curve length
  RhUtil.RhinoDivideCurve(Curve, 0, Len, False, True, crv_p, crv_t)

  A = crv_p
  B = crv_t

End Sub
```

RhUtil 曲线通过固定点（插入点曲线）

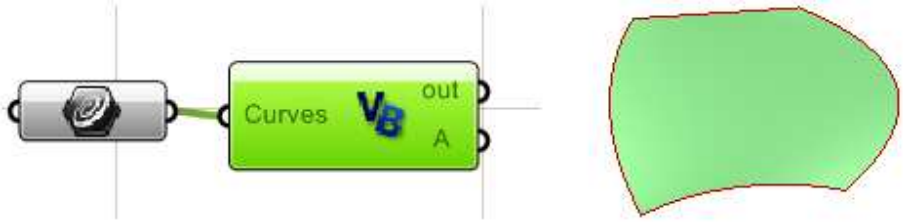
RhinoInterpCurve: 函数名
3: 曲线级
pt_array: 曲线将通过的点
Nothing: 起始切线
Nothng: 终止切线
0: 统一的节点

接下来的例子将展示通过输入一列 On3dPoints 点得到通过这些点的 nurbs 曲线

```
Sub RunScript(ByVal Points As List(Of On3dPoint))

  Dim pt_array As New ArrayOn3dPoint
  Dim i As Integer

  For i = 0 To Points.Count() - 1
    pt_array.Append(Points(i))
  Next

  'Create an interpolated nurbs curve
  Dim crv As New OnNurbsCurve
  crv = RhUtil.RhinoInterpCurve(3, pt_array, Nothing, Nothing, 0)

  If( crv.IsValid() ) Then
    'Set return vlue to list
    A = crv
  End If

End Sub
```

RhUtil Create edge surface （RhUtil 创建边沿曲面）

下面例子中的输入值是四条曲线构成的列表，输出值是一个边沿曲面。




```
Sub RunScript(ByVal Curves As List(Of OnCurve))

  Dim nc_list(3) As OnNurbsCurve
  For i As Integer = 0 To 3
    nc_list(i) = New OnNurbsCurve()
  Next

  ' Get nurb form of each curve
  For i As Integer = 0 To 3
    Curves(i).GetNurbForm(nc_list(i))
  Next

  ' Create the edgesurface
  Dim Brep As OnBrep = RhUtil.RhinoCreateEdgeSrf(nc_list)

  A = Brep
End Sub
```

16 Help（帮助）

哪里可以找到更多的关于 Rhino DotNET SDK 的信息

McNeel Wiki 里面有许多信息和实例。开发者一直积极的添加新的材料和实例。这个是极好的资源，你可以通过一下链接访问：
http://en.wiki.mcneel.com/default.aspx/McNeel/Rhino4DotNetPlugIns.html

Forums and discussion groups 论坛和讨论小组

Rhino 社区非常具有活跃性与拥护性。Grasshopper 讨论论坛是一个起步的好地方
http://grasshopper.rhino3d.com/

你也可以向 Rhino Developer Newsgroup 提问。你可以通过 McNeel 的开发者页面获得开发者新闻组的信息。
http://www.rhino3d.com/developer.htm

你也可以加入 Rhino3d 与 Grasshopper 中文论坛一起讨论 Rhino 与 Grasshopper 的相关问题
Shaper3d 论坛 http://bbs.shaper3d.cn
Grasshopper 中文论坛 http://g.shaper3d.cn

Debugging with Visual Studio 通过 Visual Studio 调试

对于更多的无法通过 GRASSHOPPER 脚本运算器来调试的复杂编码，你可以使用 Microsoft 提供的免费的 Visual Studio Express 或者开发小组的全部版本。
关于如何获得以及怎样使用 express visual studio 的细节，请点以下链接：
http://en.wiki.mcneel.com/default.aspx/McNeel/DotNetExpressEditions
如果你有渠道得到 Visual Studio 的全部版本。下面的页面将帮助你安装：
http://en.wiki.mcneel.com/default.aspx/McNeel/Rhino4DotNetPlugIns.html

Grasshopper Scripting Samples（ Grasshopper 脚本实例）

Grasshopper 收藏和讨论小组有许多脚本运算器的例子，你能找到有用

关于中文版教程

中文翻译由 **Shaper3d** 论坛 Grasshopper 版成员完成，**Shaper3d** 论坛是一个专业的 Rhino3d 中文资讯论坛，包括 Rhino3d 平台下的所有插件（包括所有第三方开发的插件），您有任何的 Rhino3d 在使用、学习与技术上的问题都竭诚欢迎前来论坛交流，争创国内最专业的 Rhino3d 资讯论坛

Shaper3d 论坛：http://bbs.shaper3d.cn
Grasshopper 分版直接访问地址：http://g.shaper3d.cn

中文翻译组人员名单（排名不分先后）：
吴迪 赵默超 赵竞 王鹏展 杨文杰 陈琪 陈锡红
校对：陈锡红/Jessesn