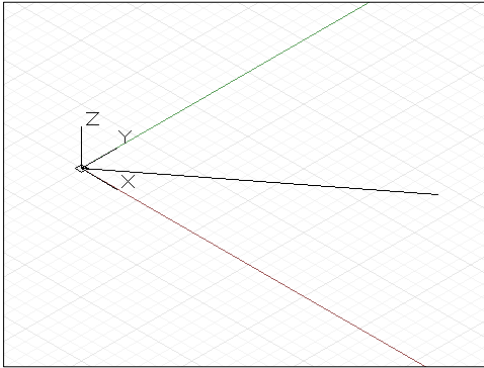


## 12： 循环

循环是指通过一段代码来重复执行指令。循环的次数可以用一个集合来控制,每遍历集合中的一个元素循环就执行一次,或用一个布尔表达式来控制,即循环一直执行知道表达式返回值为**False**。循环可以用来生成集合,寻找解决问题的方法,或者添加重复的值而不需要用**Range**表达式。

**While**语句会首先计算后面布尔表达式的值,然后重复执行代码块中的代码直到布尔值为**False**（原文为**True**，个人感觉不妥）。例如,这个代码不断创建**Line**，直到直线的长度大于10:



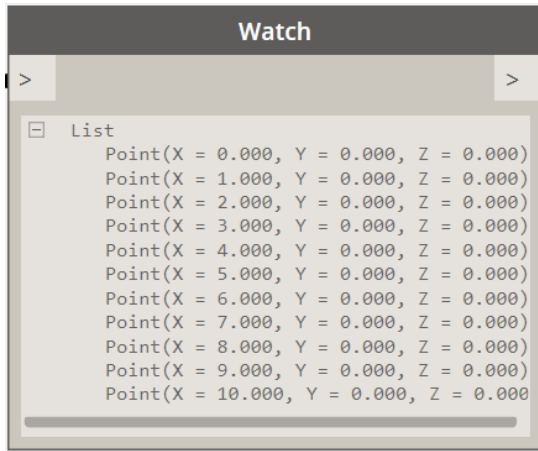
```
geometry = [Imperative]
{
    x = 1;
    start = Point.ByCoordinates(0, 0, 0);
    end = Point.ByCoordinates(x, x, x);
    line = Line.ByStartPointEndPoint(start, end);

    while (line.Length < 10)
    {
        x = x + 1;
        end = Point.ByCoordinates(x, x, x);
        line = Line.ByStartPointEndPoint(start, end);
    }

    return = line;
}
```

在组合的**Dynamo**代码中,如果一个包含多个元素的集合被输入到只有一个值的函数中,函数将单独调用集合中的每一个元素。在一个规范的**Dynamo**代码中,编写者可以选择手动重复提取集合中的一个元素。

**For**语句中提取集合中的一个元素赋值到一个已命名的变量中,集合中的每个元素都执行一次。 **For**语句语法是:  
(“赋值变量” in “集合”)



```
geometry = [Imperative]
{
    collection = 0..10;
    points = {};

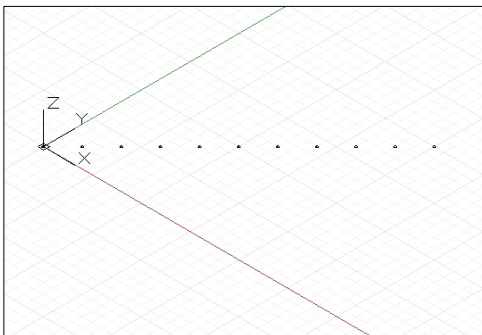
    for (i in collection)
    {
        points[i] = Point.ByCoordinates(i, 0, 0);
    }

    return = points;
}
```

# 13: Replication Guides

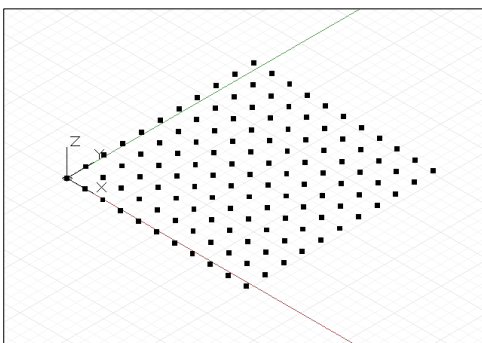
Dynamo语言被创建作为建筑师、设计师和工程师的一个特殊工具集,拥有几种语言功能并且专门针对这些领域。这些领域共同存在的一个问题是:网格状重复排列对象,比如砖墙、瓷砖地板,镶板和柱网。**Range**表达式提供了一个方便的手段生成一维的元素集合,而**replication guides**则提供了一个方便的生成二维和三维集合的方法。

**Replication guides**需要两个或三个一维集合,然后生成一维、二维或三维集合。**Replication guides**通过将符号<1>、<2>或<3>放在两个或三个集合名后面。例如,我们可以使用**Range**表达式生成两个一维集合,并使用这两个集合生成一系列点:



```
x_vals = 0..10;  
y_vals = 0..10;  
  
p = Point.ByCoordinates(x_vals, y_vals, 0);
```

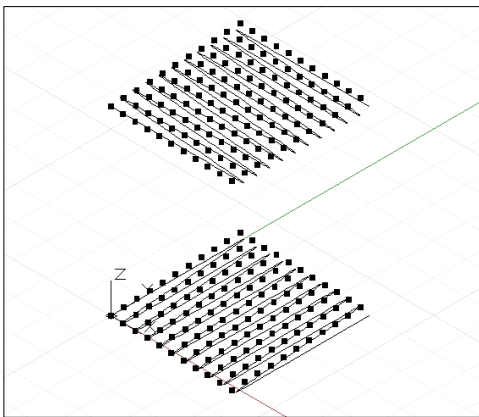
在本例中,x\_vals的第一个元素与y\_vals的第一个元素配对,第二和第二个,依此类推,直到最后。这会生成一些列点: (0,0,0)、(1,1,0)、(2,2,0)、(3,3,0),等.....。如果我们用**replication guide**,那么Dynamo可以通过这两个一维集合生成一个二维集合:



```
x_vals = 0..10;  
y_vals = 0..10;  
  
// apply replication guides to the two collections  
p = Point.ByCoordinates(x_vals<1>, y_vals<2>, 0);
```

通过对x\_vals和y\_vals两个集合应用replication guides, Dynamo会生成两个集合之间所有可能的值, x\_vals中的第一个元素与y\_vals中的所有元素组合,然后x\_vals中的第二个元素与,以此类推,直到所有元素都与y\_vals中的所有元素组合过为止。

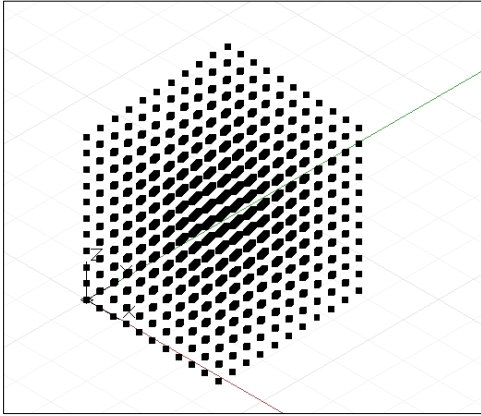
Replication guide中数字(<1>、<2>、或<3>)的顺序决定基础集合的顺序。在以下示例中,用同样的两个一维集合生成一个二维集合,只是交换<1>和<2>的顺序:



```
x_vals = 0..10;  
y_vals = 0..10;  
  
p1 = Point.ByCoordinates(x_vals<1>, y_vals<2>, 0);  
  
// apply the replication guides with a swapped order  
// and set the points 14 units higher  
p2 = Point.ByCoordinates(x_vals<2>, y_vals<1>, 14);  
  
curve1 = NurbsCurve.ByPoints(Flatten(p1));  
curve2 = NurbsCurve.ByPoints(Flatten(p2));
```

curve1和curve2显示了两个序列中点的顺序; 请注意,他们相对于对方都旋转了90度。p1是通过提取x\_vals中的元素和y\_vals中所有元素配对,而p2是通过提取y\_vals中的元素和x\_vals中所有元素配对。

Replication guides也可以用来生成三维数组,通过在第三个集合后加上符号, <3>.



```
x_vals = 0..10;  
y_vals = 0..10;  
z_vals = 0..10;  
  
// generate a 3D matrix of points  
p = Point.ByCoordinates(x_vals<1>,y_vals<2>,z_vals<3>);
```

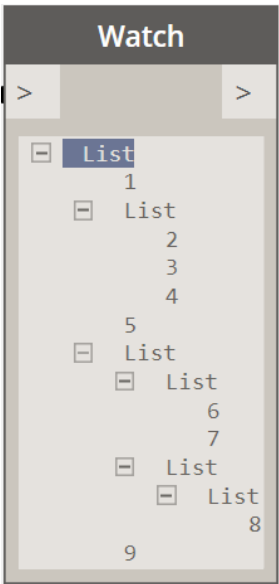
`x_vals`, `y_vals`和`z_vals`中所有可能组合情况构成了一个集合

# 14： 集合层级和锯齿状集合

集合的层级是由集合中最深一层的层级所决定的。一个由一系列单一数据组成的集合的层级是1,而一个由层级是1的集合所组成的集合的层级是2。层级可以大致定义为提取最深一级元素所需要方括号([ ])的数量。层级为1的集合只需要一个方括号来访问最深的数,而层级为3的集合需要三个相互嵌套的方括号。下表列出了1-3层级的集合，及如何返回集合中最深一级的数据。

Rank	Collection	Access 1 <sup>st</sup> Element
1	{1, 2, 3, 4, 5}	collection[0]
2	{ {1, 2}, {3, 4}, {5, 6} }	collection[0][0]
3	{ { {1, 2}, {3, 4} }, { {5, 6}, {7, 8} } }	collection[0][0][0]
...		

通过Range表达式和replication guides生成的高层级的集合通常是一致的,也就是说集合中的每个对象都是在同一深度(可以用相同数量的方括号[ ]来访问)。但是,并不是所有集合包含的对象在同一层级。这些集合称为jagged（不齐的）,也就是随着集合往后遍历，每个元素的层级有的高有的低。下面的代码会生成一个jagged collection（锯齿状集合）：



```
j = {};  
  
j[0] = 1;  
j[1] = {2, 3, 4};  
j[2] = 5;  
j[3] = { {6, 7}, { {8} } };  
j[4] = 9;
```

当试图执行集合不支持的操作时就会失败。

下面的示例展示了如何获取jagged collection（不齐的集合）中的所有元素：



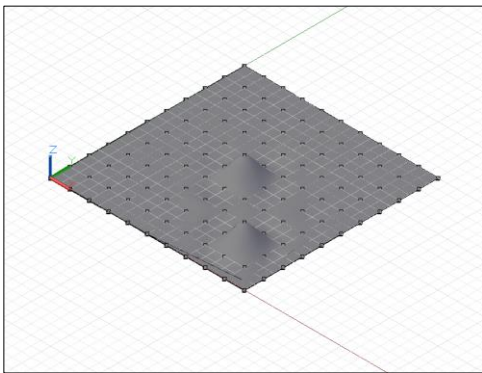
```
// generate a jagged collection
j = {1, {2, 3, 4}, 5, {{6, 7}, {{8}}}, 9};

s = j[0] + " " + j[1][0] + " " + j[1][1] + " " +
    j[1][2] + " " + j[2] + " " +
    j[3][0][0] + " " + j[3][0][1] + " " +
    j[3][1][0][0] + " " + j[4];
```

## 15: 曲面: 插入点曲面, 控制点曲面, 放样, 旋转成型

将NurbsCurve二维化就是NurbsSurface,像自由NurbsCurve一样,NurbsSurfaces也可以采用两种基本方法创建:输入一组基本点然后用Dynamo联系来生成曲面,并显示曲面的控制点。像自由曲线一样,控制点曲面是很有用的,当一个设计师知道精确的表面形状,或需要表面通过一系列约束点。另一方面,控制点曲面可能会更有用,当设计师需要比较不同平滑等级的曲面。

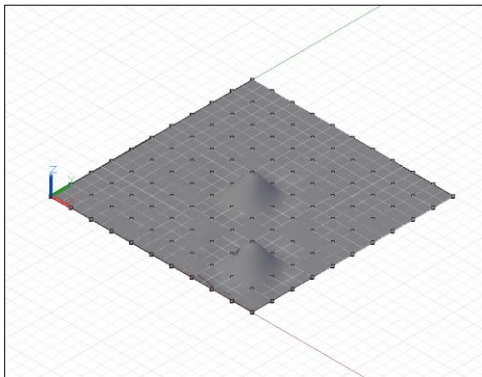
创建一个插入点曲面,只需生成一个接近曲面的形状二维的点集合。这个点集必须是规整的,也就是层级不能是不齐的(not jagged)。用"NurbsSurface.ByPoints"就可以用这些点创建曲面。



```
// python_points_1 is a set of Points generated with  
// a Python script found in Appendix 1
```

```
surf = NurbsSurface.ByPoints(python_points_1);
```

自由NurbsSurfaces也可以通过曲面的控制点来创建。像NurbsCurves,控制点可以被看作是由直线段组成的一个四边形网格,这取决于表面的光滑度,决定了最终的表面形式。创建一个控制点NurbsSurface,"NurbsSurface.ByPoints"命令包括了两个额外的参数。分别表示曲面UV两个方向上的控制线的阶数(degree)。

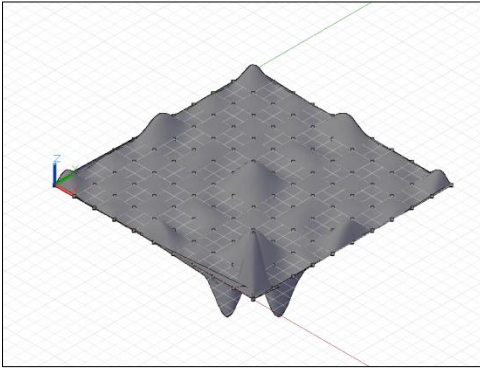


```
// python_points_1 is a set of Points generated with  
// a Python script found in Appendix 1
```

```
// create a surface of degree 2 with smooth segments  
surf = NurbsSurface.ByPoints(python_points_1, 2, 2);
```

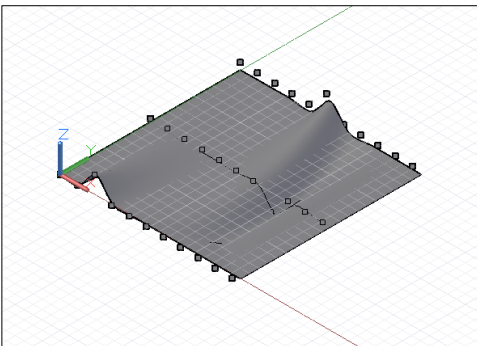


我们可以通过提高NurbsSurface的阶数来改变最终的曲面：



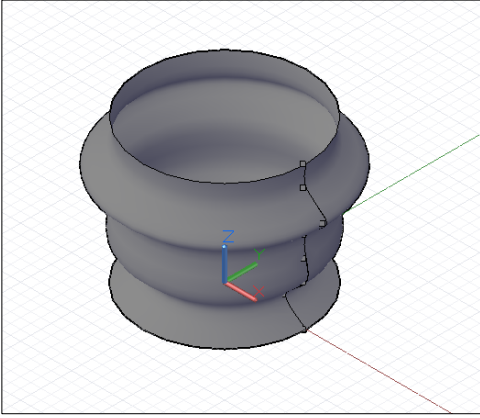
```
// python_points_1 is a set of Points generated with  
// a Python script found in Appendix 1  
  
// create a surface of degree 6  
surf = NurbsSurface.ByPoints(python_points_1, 6, 6);
```

正如曲面可以由一组输入点创建一样,也可以通过一组基准曲线来创建。这就是所谓的放样。通过"Surface.LoftFromCrossSections"来创建放样曲面,只需要一组曲线接入唯一的参数即可。



```
// python_points_2, 3, and 4 are generated with  
// Python scripts found in Appendix 1  
  
c1 = NurbsCurve.ByPoints(python_points_2);  
c2 = NurbsCurve.ByPoints(python_points_3);  
c3 = NurbsCurve.ByPoints(python_points_4);  
  
loft = Surface.LoftFromCrossSections({c1, c2, c3});
```

旋转成型曲面是一个额外的类型的表面, 由一个基础曲线围绕一个中心轴旋转而成。如果插入点曲面是插曲点曲线的二维化,那么旋转曲面就是圆和弧线的二维化。旋转曲面的形状由几个参数确定: 一个基本曲线(代表曲面的“边缘”);旋转轴基点(曲面的基点);旋转轴的方向(中央“核心”方向);扫描开始角度;扫描的角度。这些都是"Surface.Revolve constructor"的输入参数。



```
pts = {};  
pts[0] = Point.ByCoordinates(4, 0, 0);  
pts[1] = Point.ByCoordinates(3, 0, 1);  
pts[2] = Point.ByCoordinates(4, 0, 2);  
pts[3] = Point.ByCoordinates(4, 0, 3);  
pts[4] = Point.ByCoordinates(4, 0, 4);  
pts[5] = Point.ByCoordinates(5, 0, 5);  
pts[6] = Point.ByCoordinates(4, 0, 6);  
pts[7] = Point.ByCoordinates(4, 0, 7);  
  
crv = NurbsCurve.ByPoints(pts);  
  
axis_origin = Point.ByCoordinates(0, 0, 0);  
axis = Vector.ByCoordinates(0, 0, 1);  
  
surf = Surface.ByRevolve(crv, axis_origin, axis, 0,  
    360);
```

## 16：几何体参数化

---

在计算设计,曲线和曲面经常用作后续几何体的基本构成要素。为了让这些早期几何体可以成为后续几何体的参考物体,脚本必须能够提取这个对象整个表面上的点和方向。曲线和表面都可以提取这些特性,这被称为参数化。

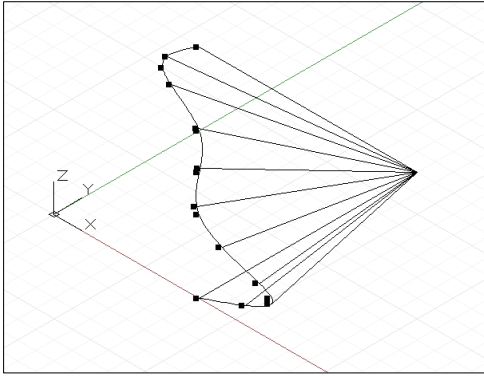
所有在曲线的点可以被认为是拥有一个唯一的参数区间从0到1。如果我们要创建一个基于几个控制点或插入点的NurbsCurve,第一点参数是0,最后一点参数是1。不可能提前知道一个确切的参数所对应的中间点是什么,这可能听起来像一个严重的缺陷,虽然通过一系列实用函数来缓和。曲面和曲线一样有类似的参数化,虽然有两个参数,而不是一个,称为u和v。如果我们通过以下几个点创建一个曲面:

```
pts = { {p1, p2, p3},  
        {p4, p5, p6},  
        {p7, p8, p9} };
```

p1 的参数  $u = 0$   $v = 0$ , 而 p9 的参数  $u = 1$   $v = 1$ 。

这种参数化并不是特别有用如果用确切的点生成曲线,其主要用途是确定位置如果中间点由NurbsCurve和NurbsSurface生成。

Curves有"PointAtParameter"命令, 这个命令需要一个在0到1之间的浮点数, 然后会返回这个数所对应曲线上的点。比如, 下面的代码返回了参数值在 0, .1, .2, .3, .4, .5, .6, .7, .8, .9, 和 1 处的点:



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(6, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(3, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);

crv = NurbsCurve.ByPoints(pts);

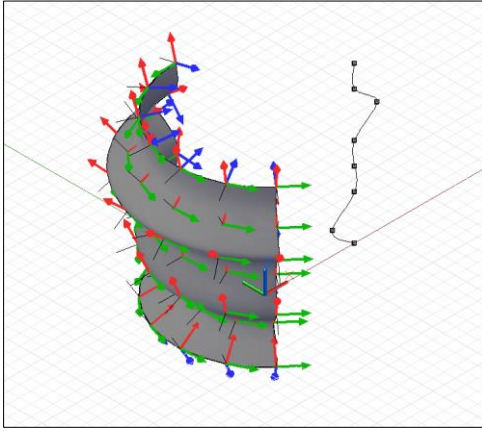
pts_at_param = crv.PointAtParameter(0..1..#11);

// draw Lines to help visualize the points
lines = Line.ByStartPointEndPoint(pts_at_param,
    Point.ByCoordinates(4, 6, 0));
```

相似的, **Surfaces**也有"PointAtParameter"命令, 他需要两个参数, 所希望生成的点所在曲面区间上的 $u$ 和 $v$ 值。

虽然提取曲线或曲面上的个别点很有用, 但是常常需要知道指定位置的特殊参数, 比如曲线或曲面的方向。

"CoordinateSystemAtParameterAlongCurve"和"CoordinateSystemAtParameter"不仅可以得到相应位置的点, 而且可以得到相应位置的曲线和曲面的坐标系统。比如, 下面这段代码沿着旋转曲面生成了一系列坐标系, 然后将相应的坐标原点沿着各自所在位置的曲面法线方向偏移, 最后连接成线:



```
pts = {};
pts[0] = Point.ByCoordinates(4, 0, 0);
pts[1] = Point.ByCoordinates(3, 0, 1);
pts[2] = Point.ByCoordinates(4, 0, 2);
pts[3] = Point.ByCoordinates(4, 0, 3);
pts[4] = Point.ByCoordinates(4, 0, 4);
pts[5] = Point.ByCoordinates(5, 0, 5);
pts[6] = Point.ByCoordinates(4, 0, 6);
pts[7] = Point.ByCoordinates(4, 0, 7);

crv = NurbsCurve.ByPoints(pts);

axis_origin = Point.ByCoordinates(0, 0, 0);
axis = Vector.ByCoordinates(0, 0, 1);

surf = Surface.ByRevolve(crv, axis_origin, axis, 90,
    140);

cs_array = surf.CoordinateSystemAtParameter(
    (0..1..#7)<1>, (0..1..#7)<2>);

def make_line(cs : CoordinateSystem) {
    lines_start = cs.Origin;
    lines_end = cs.Origin.Translate(cs.ZAxis, -0.75);

    return = Line.ByStartPointEndPoint(lines_start,
        lines_end);
}

lines = make_line(Flatten(cs_array));
```

如上所述, 一个曲线或曲面的参数区间并不是和长度相对应, 也就是0.5处并不是中点, 0.25并不是曲线或曲面长度的四分之一。为了解决这个问题, **Curves** 有一个命令允许你通过一个特殊长度来分割曲线。

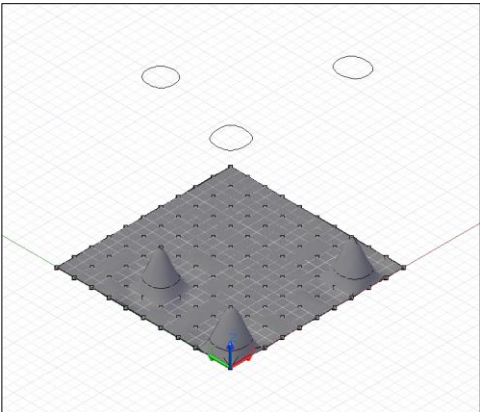
# 17： 相交和剪切

迄今为止的许多例子都集中在如何用低维物体生成高维物体。相交可以实现用高维物体创建低维物体，而剪切和选择剪切命令可以修改已经创建了的物体。

在Dynamo中相交支持所有几何对象，这意味着理论上可以用任何几何体与其他几何体相交。自然有些物体间相交是毫无意义的,例如两个点相交,所产生的对象永远是输入点本身。其他对象相交所得到的几何体可以概括到下表中的。以下图表列出了各种相交操作的结果：

Intersect:				
With:	Surface	Curve	Plane	Solid
Surface	Curve	Point	Point, Curve	Surface
Curve	Point	Point	Point	Curve
Plane	Curve	Point	Curve	Curve
Solid	Surface	Curve	Curve	Solid

下面的例子展示了一个plane 和一个NurbsSurface相交得到的结果. 生成了一系列 NurbsCurve，当然可以像用其他曲线一样使用他们。



```
// python_points_5 is a set of Points generated with
// a Python script found in Appendix 1

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

WCS = CoordinateSystem.Identity();

p1 = Plane.ByOriginNormal(WCS.Origin.Translate(0, 0,
0.5), WCS.ZAxis);

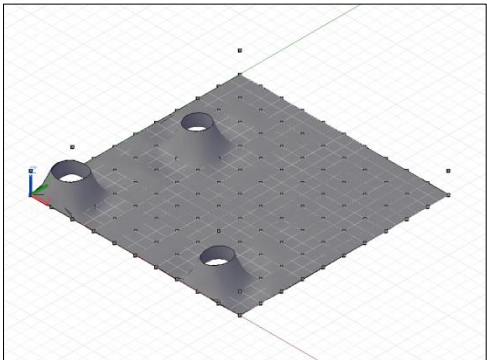
// intersect surface, generating three closed curves
crvs = surf.Intersect(p1);

crvs_moved = crvs.Translate(0, 0, 10);
```

剪切命令和相交命令类似，因此它几乎支持各种几何体。不同于相交，剪切有更多的限制：

剪切体:					
被剪切体:	Point	Curve	Plane	Surface	Solid
Curve	Yes	No	No	No	No
Polygon	NA	No	Yes	No	No
Surface	NA	Yes	Yes	Yes	Yes
Solid	NA	NA	Yes	Yes	Yes

剪切命令需要注意的是：必须指定一个“选择”点， 通过这个点觉得哪边需要保留哪边删除。Dynamo将删除距离这个点最近的部分。



```
// python_points_5 is a set of Points generated with
// a Python script found in Appendix 1

surf = NurbsSurface.ByPoints(python_points_5, 3, 3);

tool_pts = Point.ByCoordinates((-10..20..10)<1>,
                               (-10..20..10)<2>, 1);

tool = NurbsSurface.ByPoints(tool_pts);

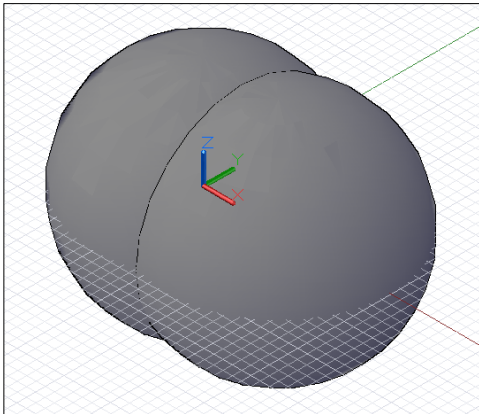
pick_point = Point.ByCoordinates(8, 1, 3);

// trim with the tool surface, and keep the surface
// closest to pick_point
result = surf.Trim(tool, pick_point);
```

# 18：几何体布尔运算

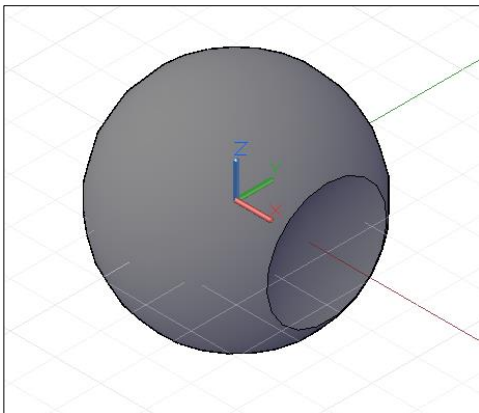
**Intersect, Trim, and SelectTrim** 主要用语低纬物体，比如点、曲线和曲面。另一方面，实体元素有一些额外的命令，可以用来改变他们的形体。类似剪切（Trim），从一个物体中减去一部分 和通过把多个物体组合在一起形成一个大的物体。

**Union** 命令需要两个实体元素，然后生成一个在这两个实体相交部分之外的完整实体。重叠部分也包含在了最终形体中。下面的例子把两个球体组合成了一个新的几何体：



```
s1 = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin, 6);  
  
s2 = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin.Translate(4, 0,  
    0), 6);  
  
combined = s1.Union(s2);
```

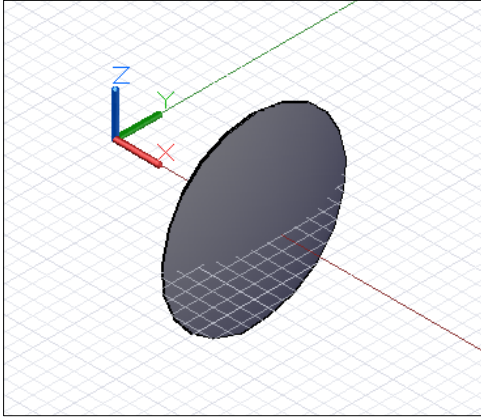
另一个命令(**Difference**)), 类似剪切，从被剪切体中减去剪切体与被剪切体相交的部分。下面的例子，我们从球体中减去了一小块：



```
s = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin, 6);  
  
tool = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin.Translate(10, 0,  
    0), 6);  
  
result = s.Difference(tool);
```

相交命令**Intersect**可以得到两个实体之间相同的部分。在下面的例子中，**Difference**命令换成了**Intersect**，就得到了上一个例子中减去的那块实体：

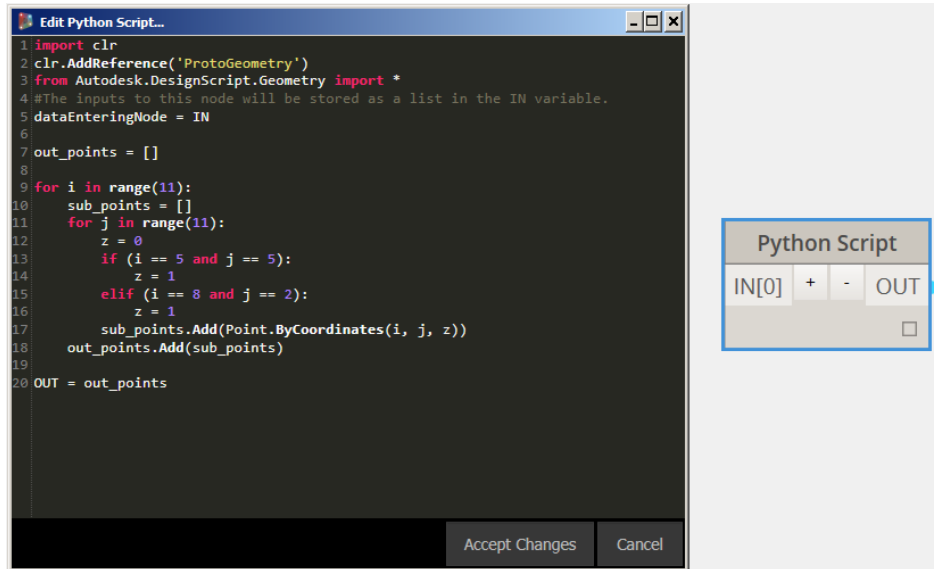




```
s = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin, 6);  
  
tool = Sphere.ByCenterPointRadius(  
    CoordinateSystem.Identity().Origin.Translate(10, 0,  
    0), 6);  
  
result = s.Intersect(tool);
```

# A-1: Python Point Generators

下面几个例子用Python代码生成了几种点阵。把这些代码像下面这样粘贴到一个Python Script node中：



## python\_points\_1

```
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 5 and j == 5):
            z = 1
        elif (i == 8 and j == 2):
            z = 1
        sub_points.Add(Point.ByCoordinates(i, j, z))
    out_points.Add(sub_points)

OUT = out_points
```

## python\_points\_2

```
out_points = []

for i in range(11):
    z = 0
    if (i == 2):
        z = 1
    out_points.Add(Point.ByCoordinates(i, 0, z))

OUT = out_points
```

## python\_points\_3

```
out_points = []

for i in range(11):
    z = 0
    if (i == 7):
        z = -1
    out_points.Add(Point.ByCoordinates(i, 5, z))

OUT = out_points
```

## python\_points\_4

```
out_points = []

for i in range(11):
    z = 0
    if (i == 5):
        z = 1
    out_points.Add(Point.ByCoordinates(i, 10, z))

OUT = out_points
```

## python\_points\_5

```
out_points = []

for i in range(11):
    sub_points = []
    for j in range(11):
        z = 0
        if (i == 1 and j == 1):
            z = 2
        elif (i == 8 and j == 1):
            z = 2
        elif (i == 2 and j == 6):
            z = 2
        sub_points.Add(Point.ByCoordinates(i, j, z))
    out_points.Add(sub_points)

OUT = out_points
```