DesignScript @ Domain Specific Modelling 2016

Robert Aish, Bartlett/UCL and Emmanuel Mendoza, ARM

DesignScript is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation.

In this presentation we are focussing on the application domain, the challenges this presents and how DesignScript address these challenges. This is based on our paper

http://www.dsmforum.org/events/DSM16/Papers/Aish_Mendoza.pdf

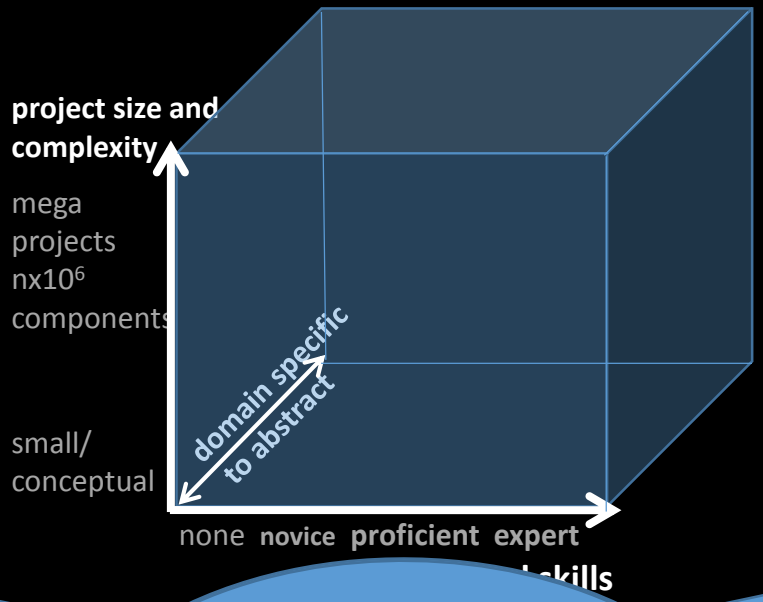
A discussion of the design decisions behind DesignScript and how it is implemented will be presented tomorrow at DSLDI 2016

Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

We are addressing the domain of architecture. We can describe this domain by two different types of buildings and how computer based applications are used in their design

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

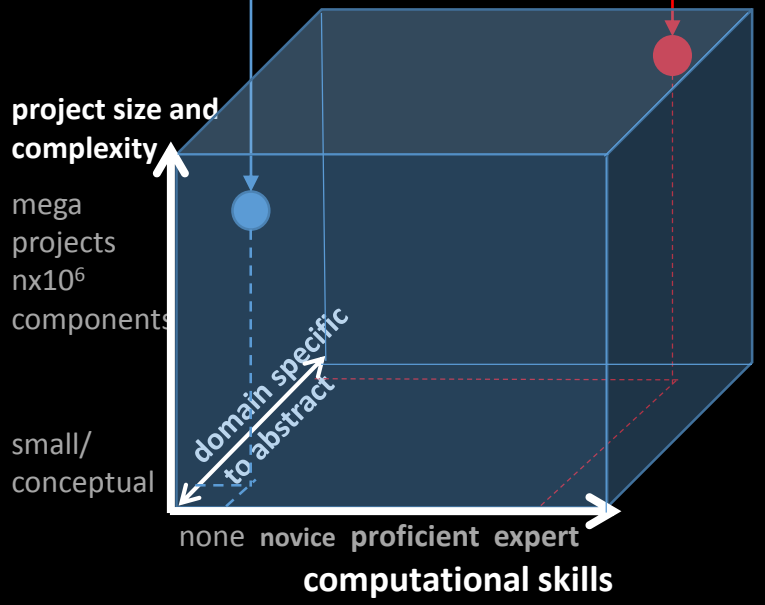


In fact we can describe These differences using three characteristic dimensions:

1. Size and complexity
2. Domain Specific to Abstract
3. The level of computational skill required

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

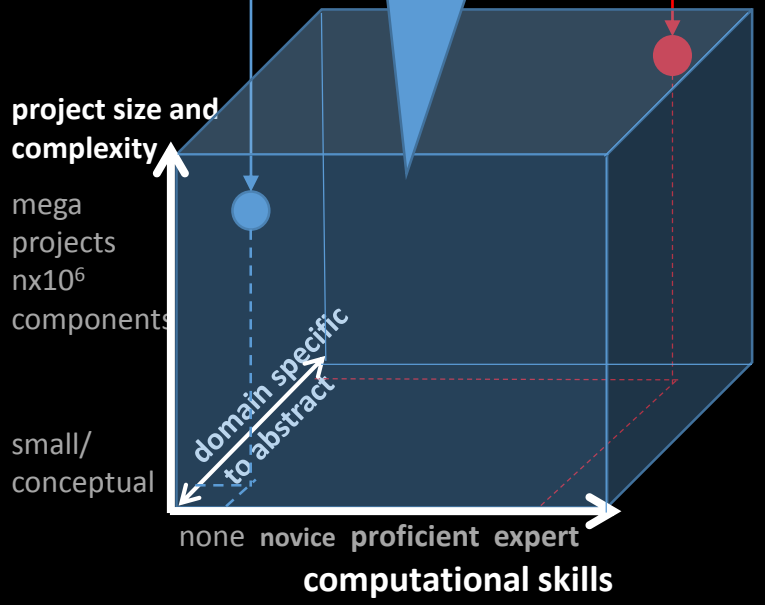
We can position each of these buildings in this space



Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

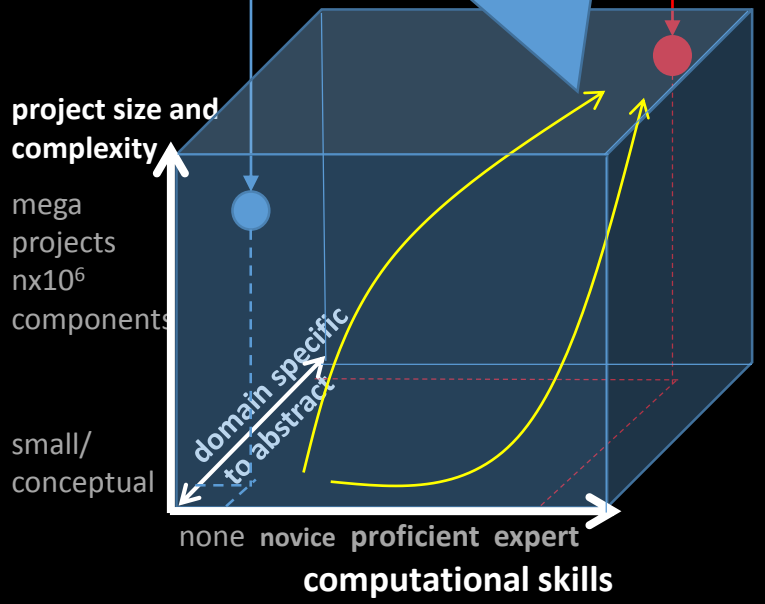
…but what we are really interested in is the gap in the middle…



Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

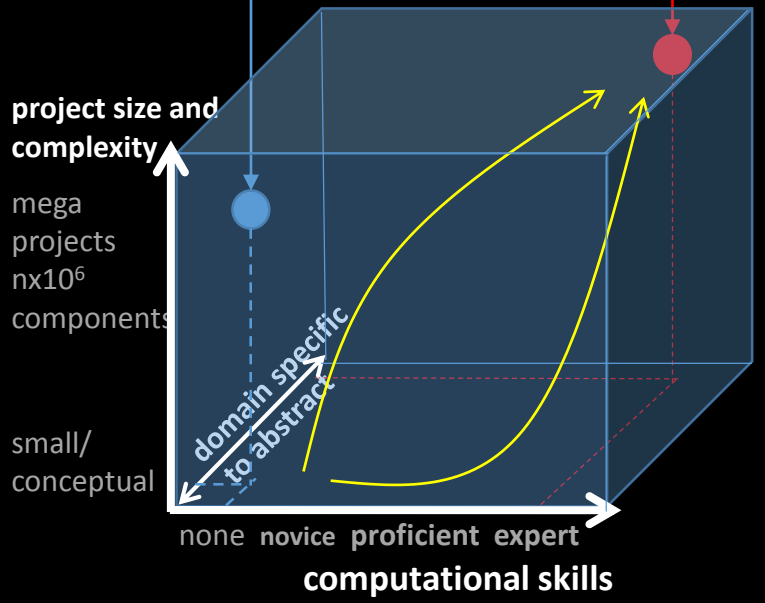
Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

… and the possible
paths a novice user might take to
use more advanced computation to
design more interesting
architecture



Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skills. Modelled by ‘direct manipulation’.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.



Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skill Modelled by 'direct manipulation'

hence the 'challenge'

The challenge is to create programing tools that are accessible to novice end-users, who can then create architecture which progresses beyond the hard-coded restrictions of conventional systems.

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

Moderately complex, ultra domain specific with hard coded components and inter-component relationships. The user requires no computing skil
Modelled by 'direct manipulatio

The challenge is to create programing tools that are accessible to novice end-users, who can then create architecture which progresses beyond the hard-coded restrictions of conventional systems.

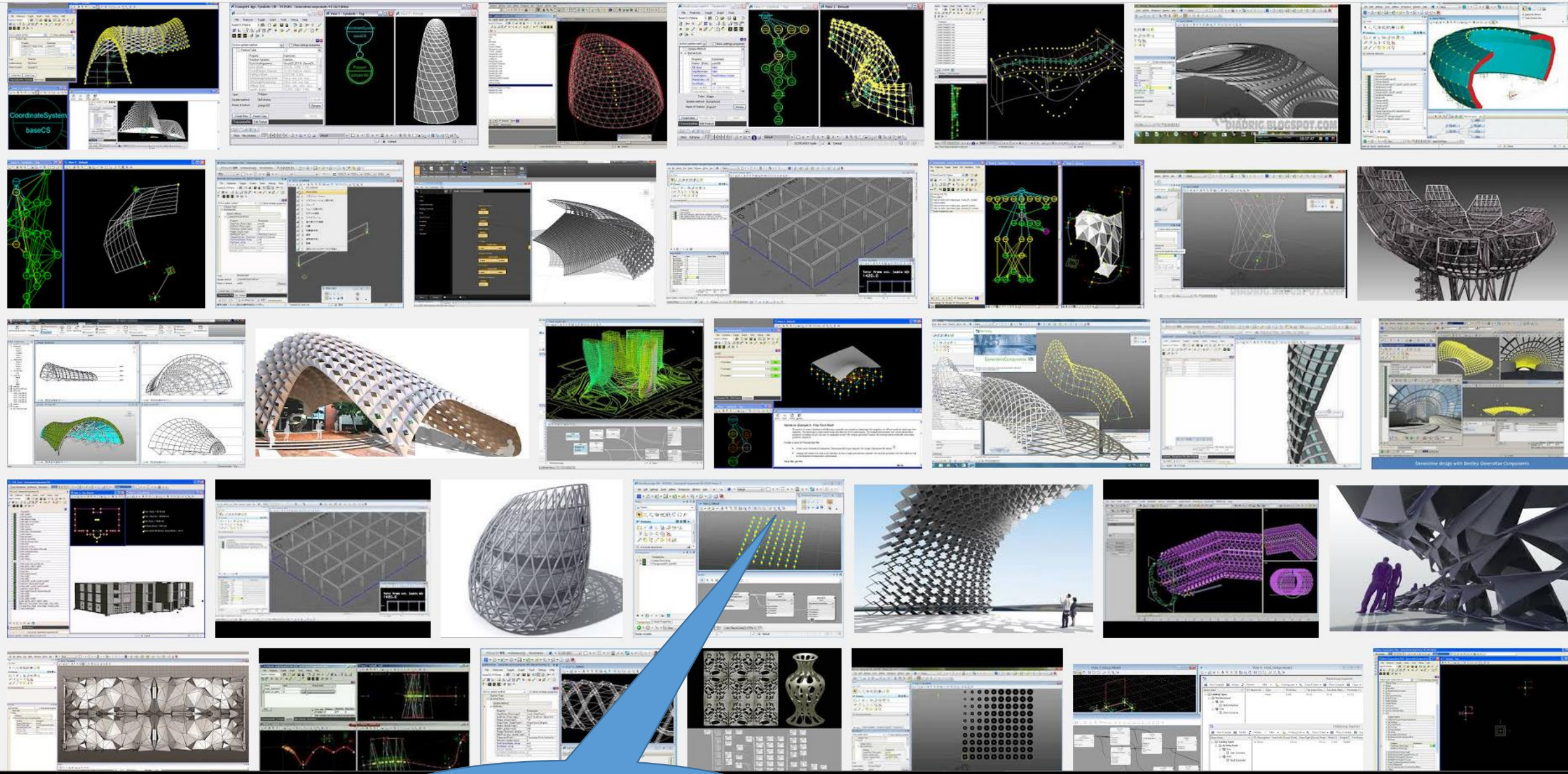
and here is an example

Highly complex, abstract geometry computed using completely general purpose programming tools and geometry libraries. The user has to be an accomplished programmer. The program is the model.

and here is an example

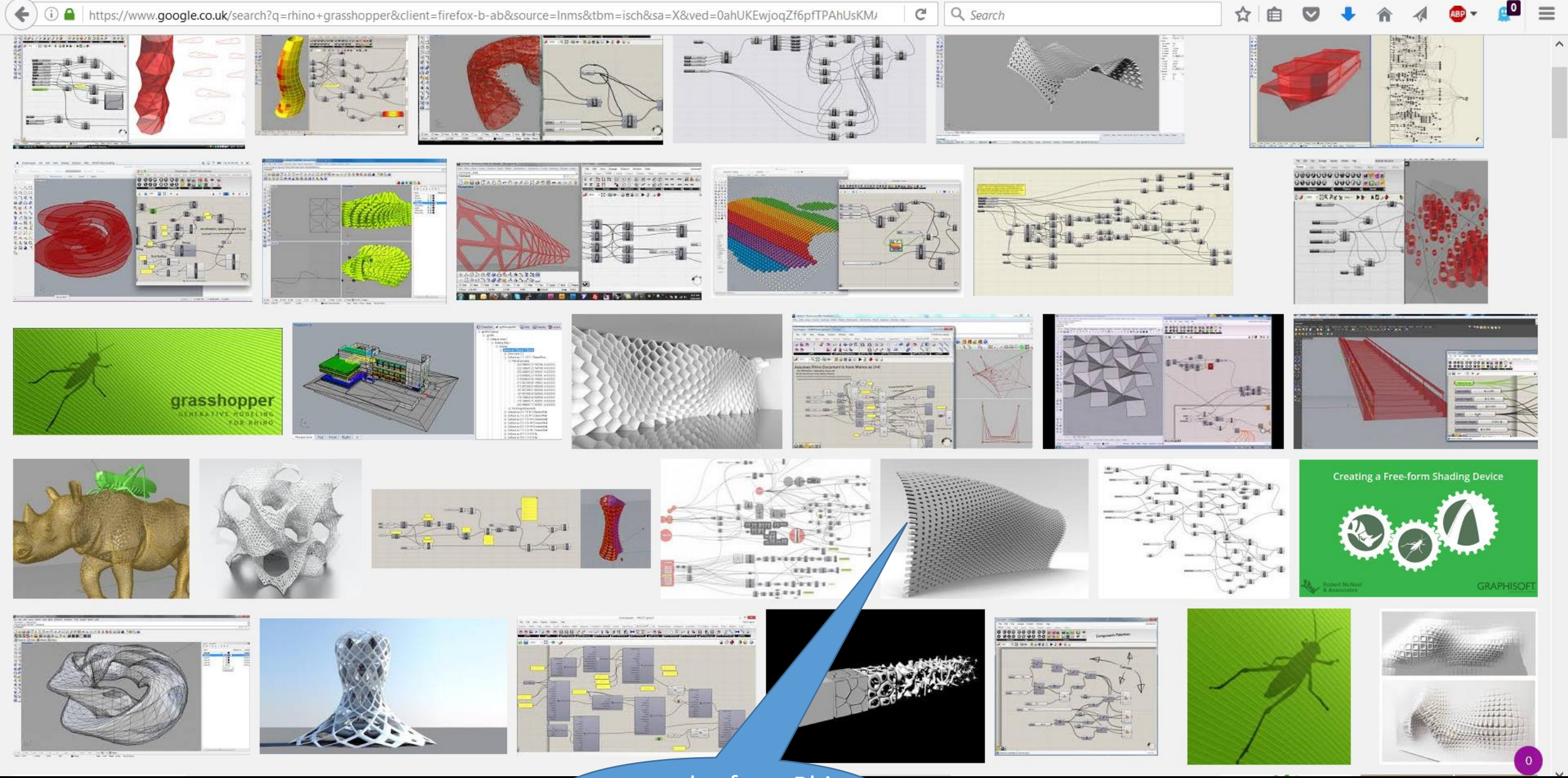
The challenge is to create programing tools that are accessible to novice end-users, who can then create architecture which progresses beyond the hard-coded restrictions of conventional systems.

Marina Bay Bridge, Singapore
http://2.bp.blogspot.com/-TkamurQBnYA/VO0pTt1Y23I/AAAAAAAAAOE/Zety_u2abkU/s1600/helix%2Bbridge.jpg

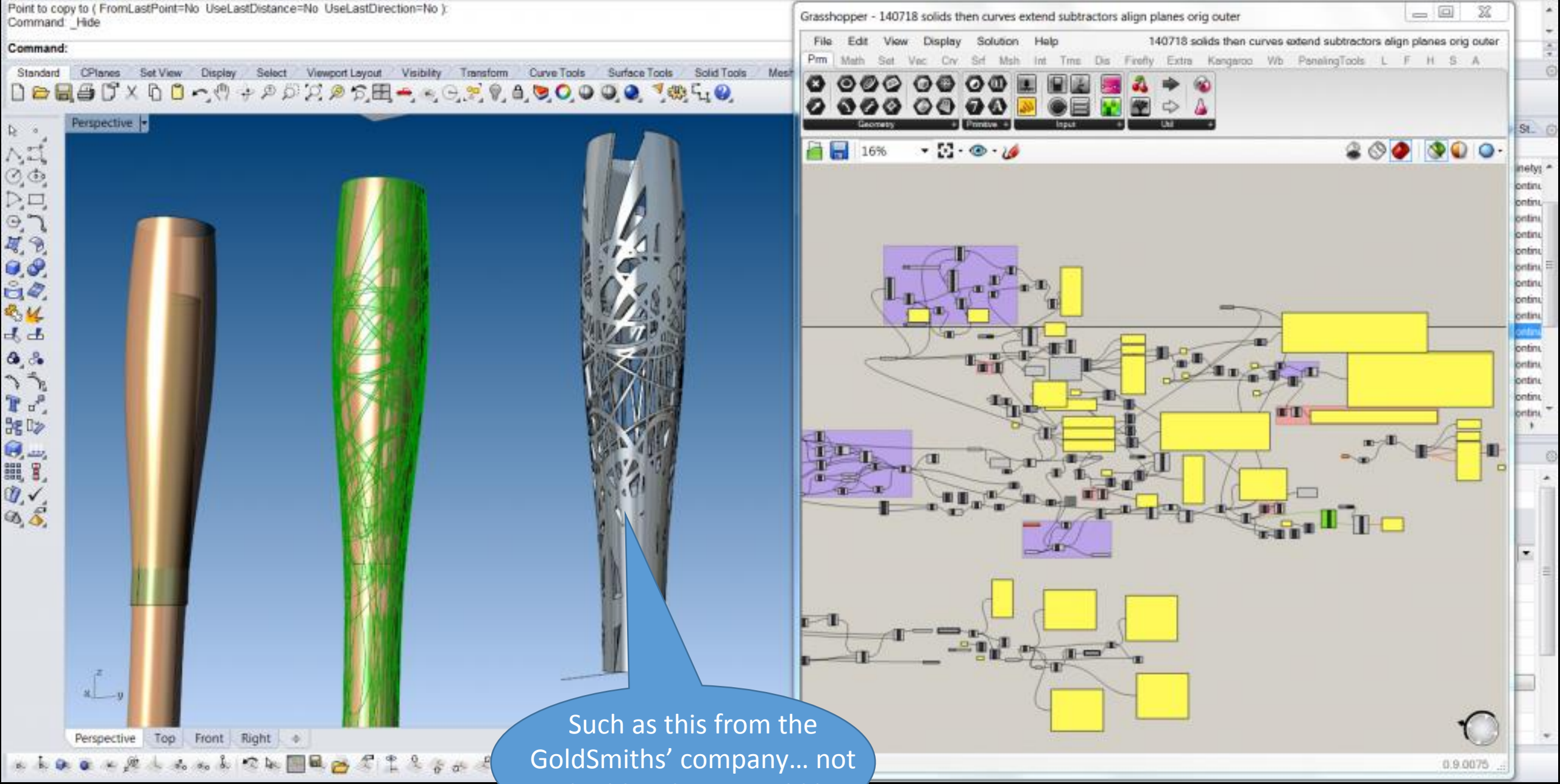


GenerativeComponents:
https://communities.bentley.com/products/products_generativecomponents/w/generative_components_community_wiki

Velodrome at the 2012 London Olympics
http://www.designboom.com/cms/images/rido/vel01.jpg

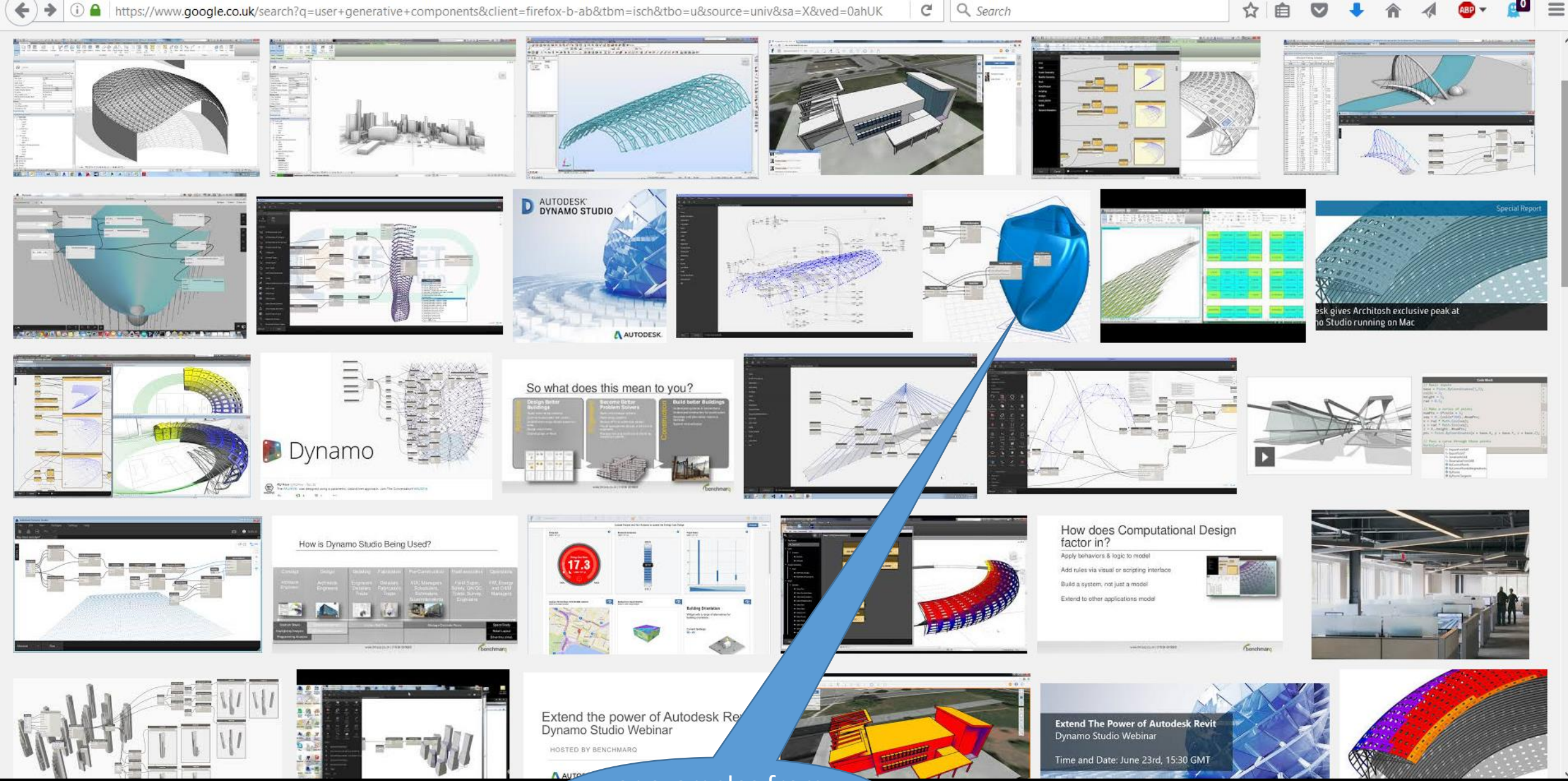


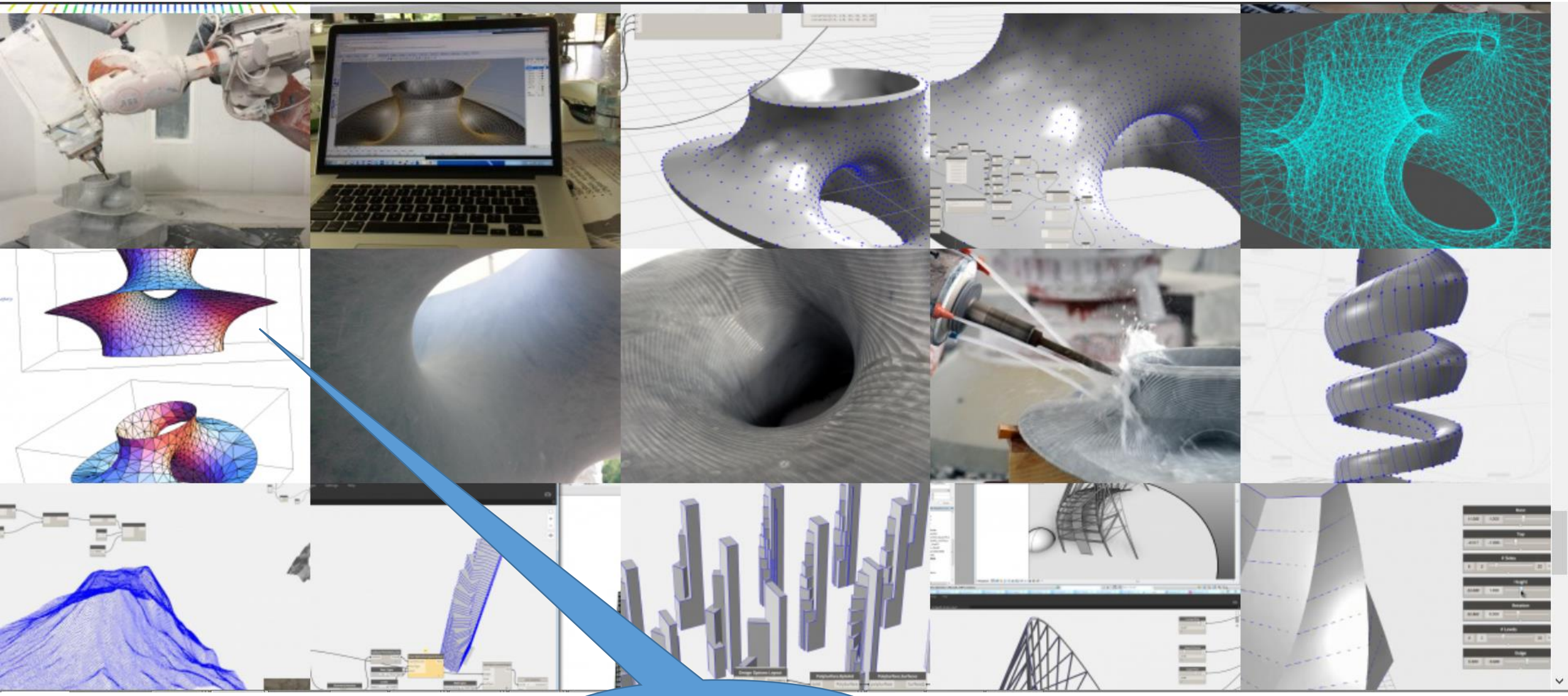
Rhino Grasshopper
http://www.grasshopper3d.com/

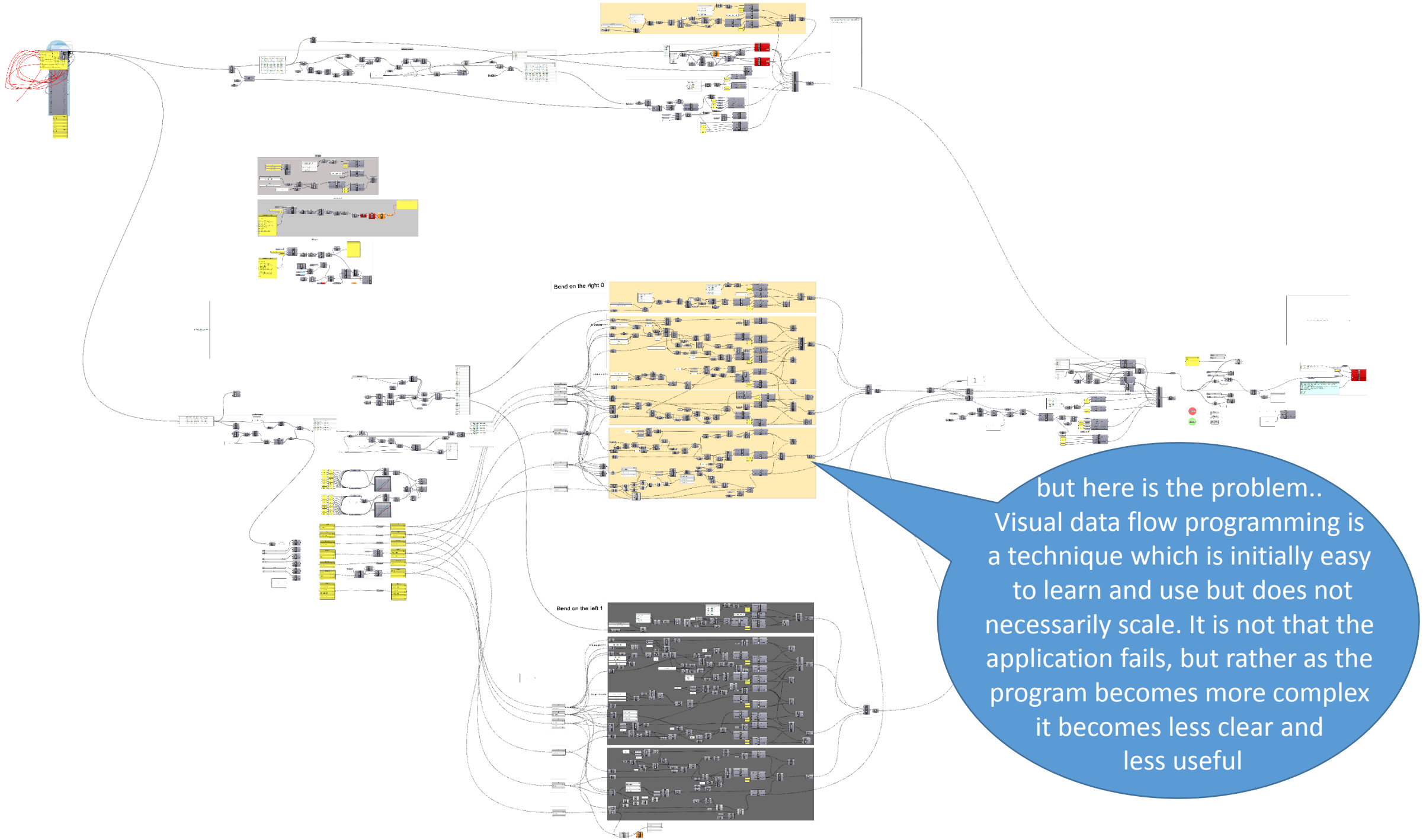


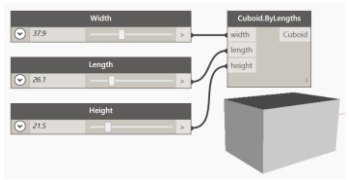
Gold Smiths

http://technical-journal.thegoldsmiths.co.uk/wp-content/uploads/2014/08/Grasshopper-slicing-build-process-1024x549.png
/


https://www.google.co.uk/search?q=user+generative+components&client=firefox-b-ab&tbm=isch&tbo=u&source=univ&sa=X&ved=0ahUK
Search
AUTODESK
DYNAMO STUDIO
AUTODESK.
Special Report
So what does this mean to you?
Dynamo
How is Dynamo Studio Being Used?
How does Computational Design factor in?
Apply behaviors & logic to model
Add rules via visual or scripting interface
Build a system, not just a model
Extend to other applications model
Extend the power of Autodesk Re
Dynamo Studio Webinar
HOSTED BY BENCHMARQ
Extend The Power of Autodesk Revit
Dynamo Studio Webinar
Time and Date: June 23rd, 15:30 GMT
or examples from Dynamo


more examples from
Dynamo


Bend on the right 0
Bend on the left 1
but here is the problem..
Visual data flow programming is a technique which is initially easy to learn and use but does not necessarily scale. It is not that the application fails, but rather as the program becomes more complex it becomes less clear and less useful

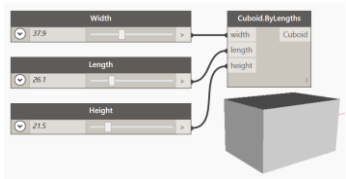


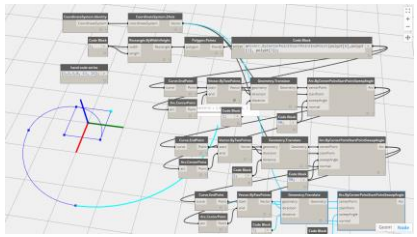
Increasing skills required

1. Visual dataflow programming provides an easily accessible approach for simple computational design problems

Increasing complexity of result

So this is the argument





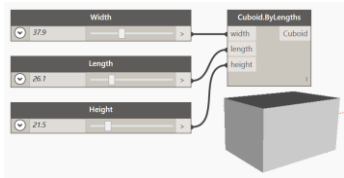
2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow

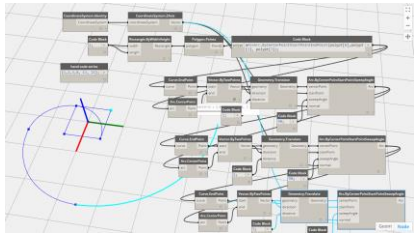
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems

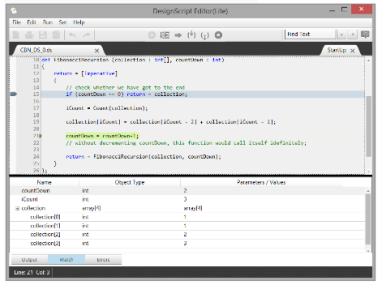
Increasing skills required

Increasing complexity of result

So this is the argument







2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow

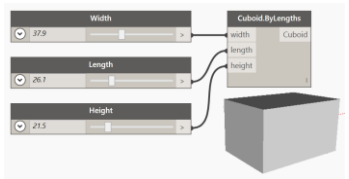
3. The skill level required for programming in conventional imperative high level languages

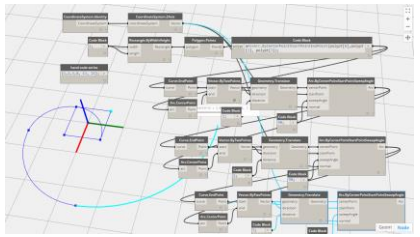
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems

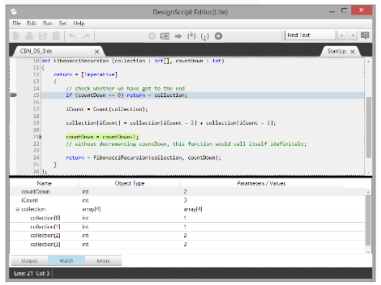
Increasing skills required

Increasing complexity of result

So this is the argument







2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow

3. The skill level required for programming in conventional imperative high level languages

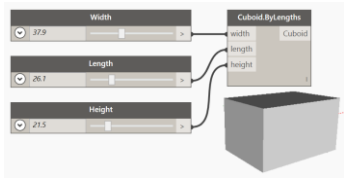
1. Visual dataflow programming provides an easily accessible approach for simple computational design problems

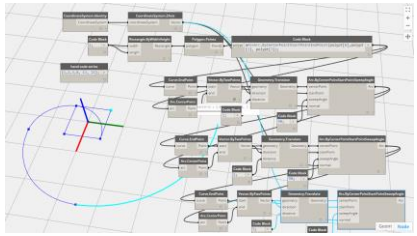
4. The increase in skills required to move from data flow programming to regular high level languages may present 'abstraction barriers' and be beyond the range of novice programmers

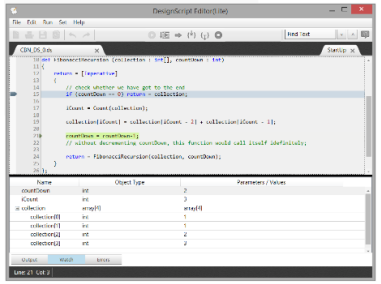
Increasing skills required

Increasing complexity of result

So this is the argument







Increasing skills required

1. Visual dataflow programming provides an easily accessible approach for simple computational design problems

2. As the number of nodes and arcs increases, the visual complexity increases non-linearly, reducing the effectiveness of data flow

3. The skill level required for programming in conventional imperative high level languages

4. The increase in skills required to move from data flow programming to regular high level languages may present 'abstraction barriers' and be beyond the range of novice programmers

Increasing complexity of result

Increasing skills required

1. Visual dataflow programming provides an easily accessible approach for simple computational design problems

[Figure 4]

2. Node to Code: As the number of nodes and arcs increases, a region of the visual data flow graph can be converted to text based code, thus reducing visual complexity and 'seeding' the user's code with the logic previously developed in visual data flow programing

3. Text based Data Flow programming [Figure 5]

4. Hybrid Data Flow-Imperative programming [Figure 6]

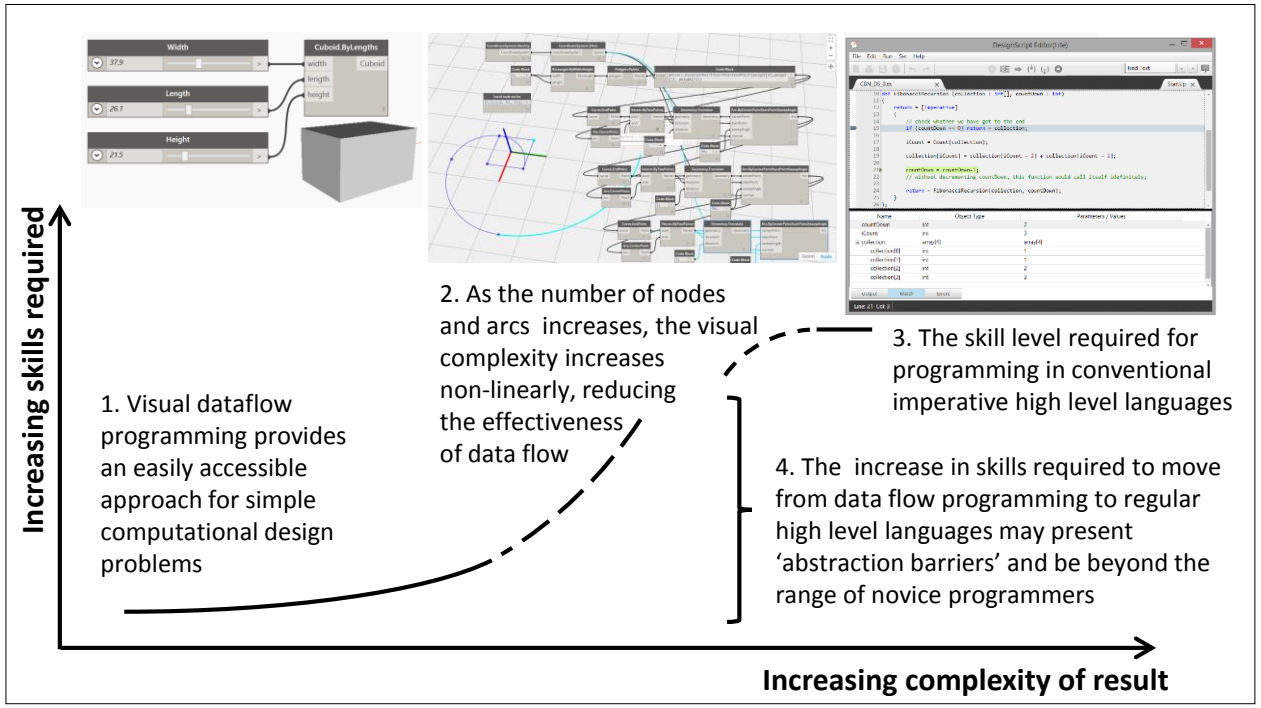
5. Encapsulation, via user functions [Figure 7]

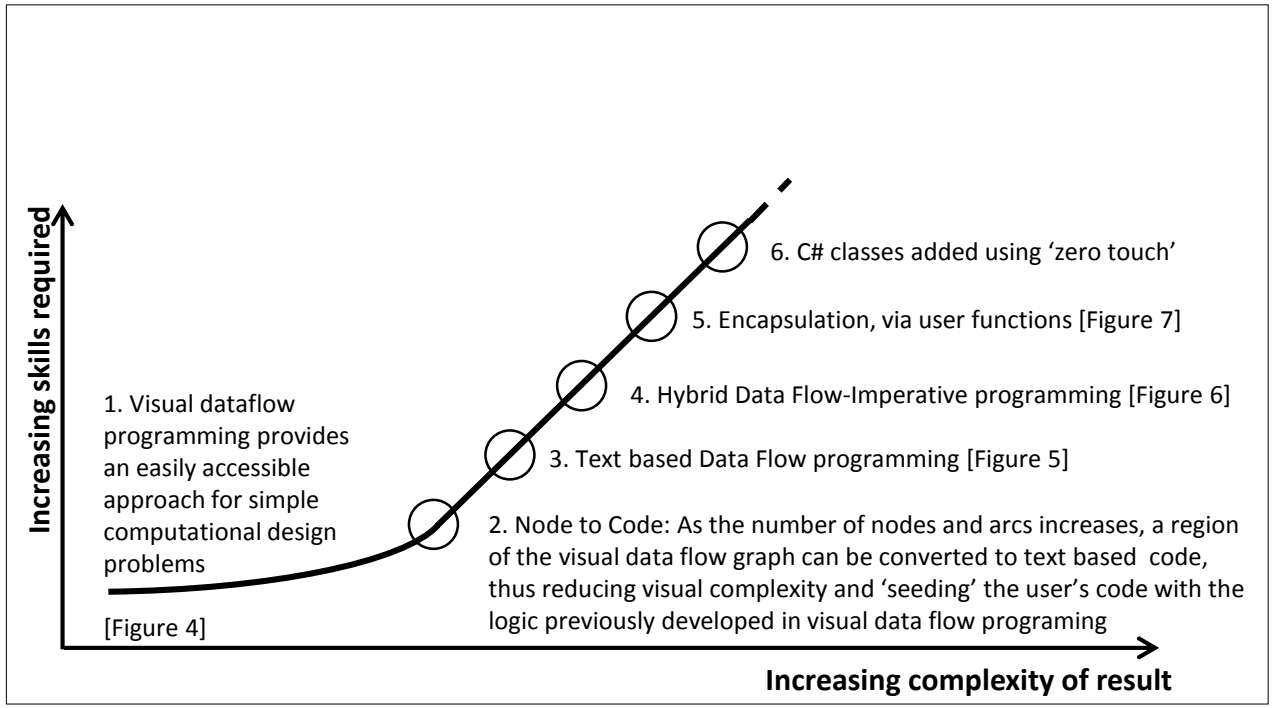
6. C# classes added using 'zero touch'

Increasing complexity of result

DesignScript is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation

… and this is a possible solution

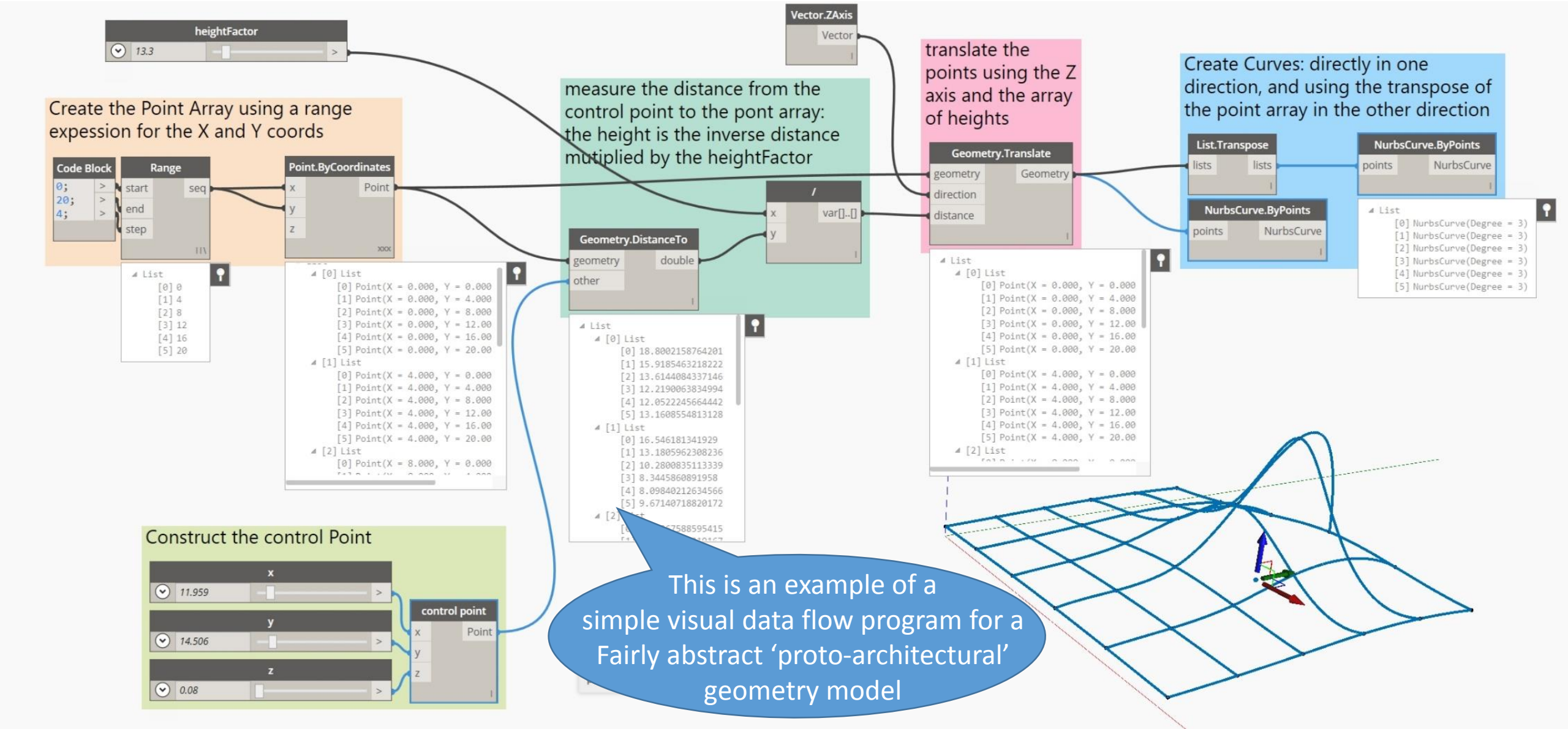


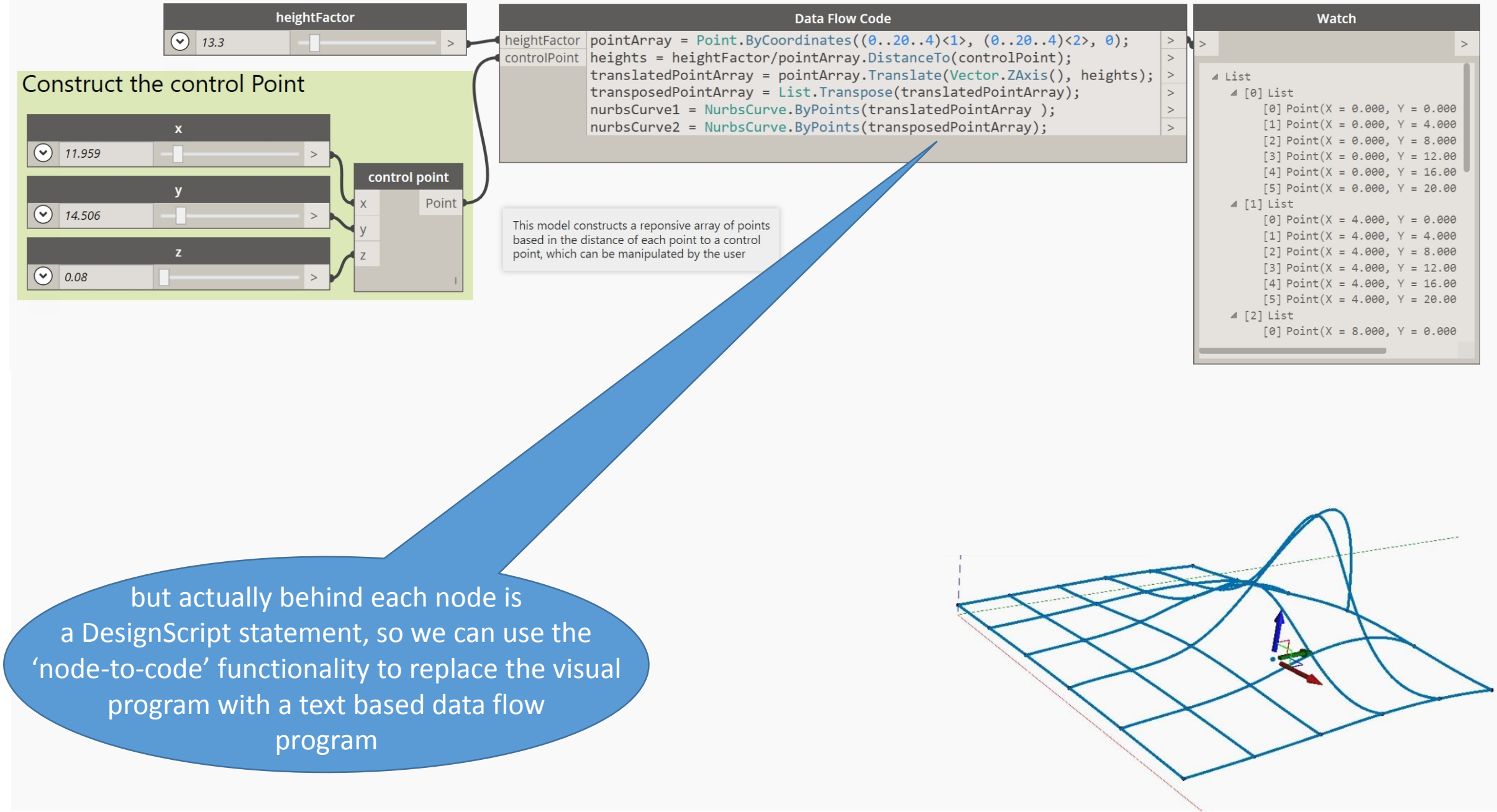


[Figure 4]

DesignScript is a multi-paradigm domain-specific end-user language and modelling environment for architectural and engineering computation. DesignScript implements a series of intermediate programming techniques between visual data flow programing and regular text based programing. This provides an abstraction gradient which allows the gradual introduction of more advanced programing concepts and notation.

… and this is a possible solution


heightFactor
13.3
Vector.ZAxis
Vector
translate the points using the Z axis and the array of heights
Create Curves: directly in one direction, and using the transpose of the point array in the other direction
Create the Point Array using a range expression for the X and Y coords
measure the distance from the control point to the pont array: the height is the inverse distance multiplied by the heightFactor
Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
Geometry.DistanceTo
geometry
other
double
/
x
y
var[]..[]
Geometry.Translate
geometry
direction
distance
Geometry
List.Transpose
lists
NurbsCurve.ByPoints
points
NurbsCurve
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0.000
[1] Point(X = 4.000, Y = 4.000
[2] Point(X = 4.000, Y = 8.000
[3] Point(X = 4.000, Y = 12.00
[4] Point(X = 4.000, Y = 16.00
[5] Point(X = 4.000, Y = 20.00
[2] List
[0] Point(X = 8.000, Y = 0.000
[0] 18.8002158764201
[1] 15.9185463218222
[2] 13.6144084337146
[3] 12.2190063834994
[4] 12.0522245664442
[5] 13.1608554813128
[1] List
[0] 16.546181341929
[1] 13.1805962308236
[2] 10.2800835113339
[3] 8.3445860891958
[4] 8.09840212634566
[5] 9.67140718820172
[0] NurbsCurve(Degree = 3)
[1] NurbsCurve(Degree = 3)
[2] NurbsCurve(Degree = 3)
[3] NurbsCurve(Degree = 3)
[4] NurbsCurve(Degree = 3)
[5] NurbsCurve(Degree = 3)
Construct the control Point
x
11.959
y
14.506
z
0.08
control point
This is an example of a simple visual data flow program for a Fairly abstract ‘proto-architectural’ geometry model


heightFactor
13.3
Data Flow Code
heightFactor
controlPoint
pointArray = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<2>, 0);
heights = heightFactor/pointArray.DistanceTo(controlPoint);
translatedPointArray = pointArray.Translate(Vector.ZAxis(), heights);
transposedPointArray = List.Transpose(translatedPointArray);
nurbsCurve1 = NurbsCurve.ByPoints(translatedPointArray);
nurbsCurve2 = NurbsCurve.ByPoints(transposedPointArray);
Watch
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0.000
[1] Point(X = 4.000, Y = 4.000
[2] Point(X = 4.000, Y = 8.000
[3] Point(X = 4.000, Y = 12.00
[4] Point(X = 4.000, Y = 16.00
[5] Point(X = 4.000, Y = 20.00
[2] List
[0] Point(X = 8.000, Y = 0.000
Construct the control Point
x
11.959
y
14.506
z
0.08
control point
x
y
z
Point
This model constructs a reponsive array of points based in the distance of each point to a control point, which can be manipulated by the user
but actually behind each node is a DesignScript statement, so we can use the ‘node-to-code’ functionality to replace the visual program with a text based data flow program

heightFactor

13.3

Construct the control Point

x

11.959

y

14.506

z

0.08

control point

x

y

z

Point

This model constructs a reponsive array of points based in the distance of each point to a control point, which can be manipulated by the user

Imperative Code

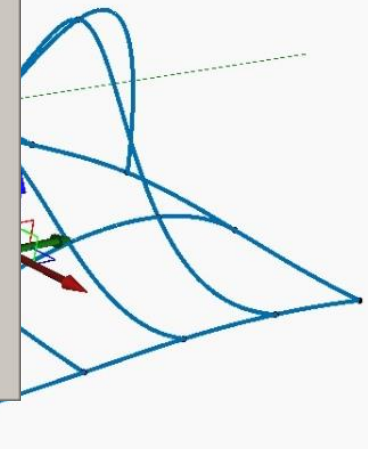
heightFactor
controlPoint

```
heightFactor;
controlPoint;
pointArray  ={};
translatedPointArray ={};
transposedPointArray = {};
nurbsCurve1 ={};
nurbsCurve2 ={};
[Imperative]
{
    xRange = (0..20..4);
    yRange = (0..20..4);

    xCount = Count(xRange);
    yCount = Count(yRange);

    for (i in 0..(xCount-1))
    {
        for (j in 0..(yCount-1))
        {
            pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
        }
    }

    for (i in 0..(xCount-1))
    {
        for (j in 0..(yCount-1))
        {
            height = heightFactor/pointArray[i][j].DistanceTo(controlPoint);
            translatedPointArray[i][j] = pointArray[i][j].Translate(Vector.ZAxis(
        }
        nurbsCurve1[i] = NurbsCurve.ByPoints(translatedPointArray[i]);
    }

    transposedPointArray = List.Transpose(translatedPointArray);
    for (i in 0..(yCount-1))
    {
        nurbsCurve2[i] = NurbsCurve.ByPoints(transposedPointArray[i]);
    }
};
```

Watch

```
List
  [0] List
      [0] Point(X = 0.000, Y = 0.000
      [1] Point(X = 0.000, Y = 4.000
      [2] Point(X = 0.000, Y = 8.000
      [3] Point(X = 0.000, Y = 12.00
      [4] Point(X = 0.000, Y = 16.00
      [5] Point(X = 0.000, Y = 20.00
  [1] List
      [0] Point(X = 4.000, Y = 0.000
      [1] Point(X = 4.000, Y = 4.000
      [2] Point(X = 4.000, Y = 8.000
      [3] Point(X = 4.000, Y = 12.00
      [4] Point(X = 4.000, Y = 16.00
      [5] Point(X = 4.000, Y = 20.00
  [2] List
      [0] Point(X = 8.000, Y = 0.000
```

and here is the same program written using Imperative code. Ultimately this is more flexible and expressive but requires more skill from the user



starter function

```
def FibonacciStarter (count)
{

 // seed with starter values;
    returnCollection = {0,1};

    return = FibonacciRecursion(returnCollection, count-1);
};
```

Integer Slider

16

Code Block

```
howMany | result = FibonacciStarter(howMany);
```

Debugging is also considered.
With a pure data flow program,
the user can trace trough the graph
of connected nodes and inspect the values
created. But with Imperative code, which can be
iterative and recursive there are internal states
which do not manifest themselves external to the
nodes. This is where a conventional IDE is
important to allow the user to inspect
the internal behaviour of the
code.

DesignScript Editor(Lite)

File Edit Run Set Help

Find Text

CBN_DS_0.ds | StartUp

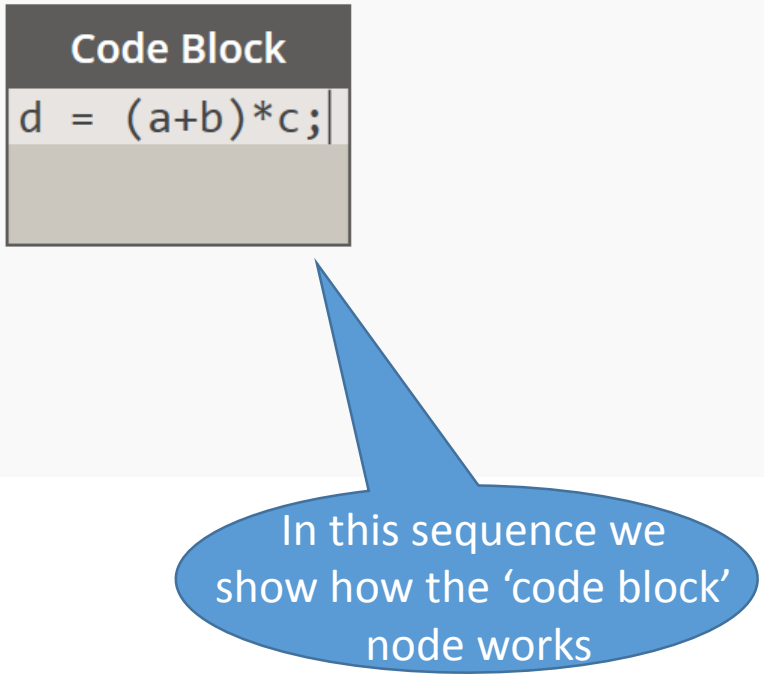
```
10 def FibonacciRecursion (collection : int[], countDown : int)
11 {
12     return = [Imperative]
13     {
14         // check whether we have got to the end
15         if (countDown == 0) return = collection;
16
17         iCount = Count(collection);
18
19         collection[iCount] = collection[iCount - 2] + collection[iCount - 1];
20
21         countDown = countDown-1;
22         // without decrementing countDown, this function would call itself idefinitely;

           return = FibonacciRecursion(collection, countDown);
25     }
26 };
```

Name	Object Type	Parameters / Values
countDown	int	2
iCount	int	3
collection	array[4]	array[4]
collection[0]	int	1
collection[1]	int	1
collection[2]	int	2
collection[3]	int	3

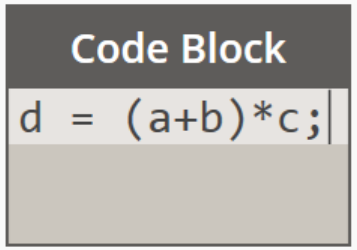
Output | Watch | Errors

Line: 21 Col: 3

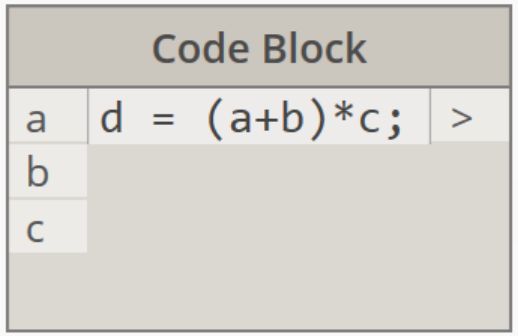
1. Double click to create an empty code block node and type the program statement



1. Double click to create an empty code block node and type the program statement

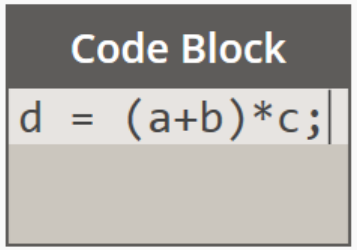


2. Clicking outside the code block node and input and output ‘ports’ are automatically generated

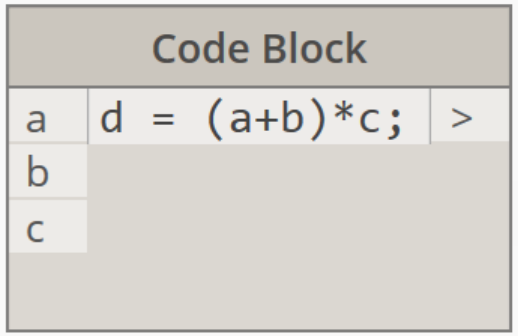


In this sequence we show how the ‘code block’ node works

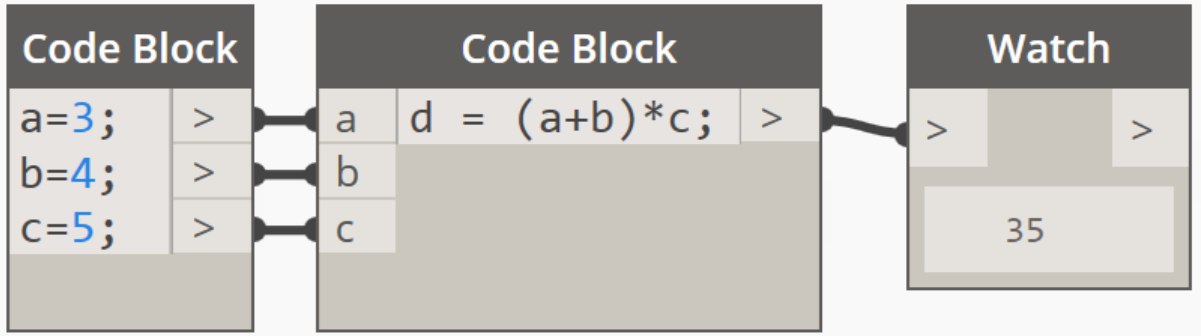
1. Double click to create an empty code block node and type the program statement



2. Clicking outside the code block node and input and output ‘ports’ are automatically generated

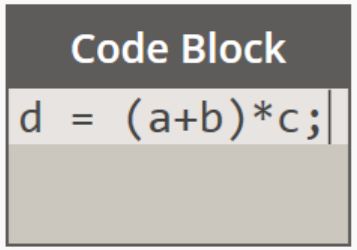


3. Connect up the input and output ports to provide sufficient data to run the program

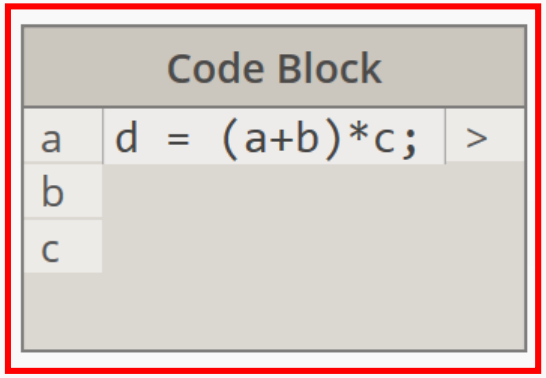


This sequence show how the ‘code block’ node works

1. Double click to create an empty code block node and type the program statement

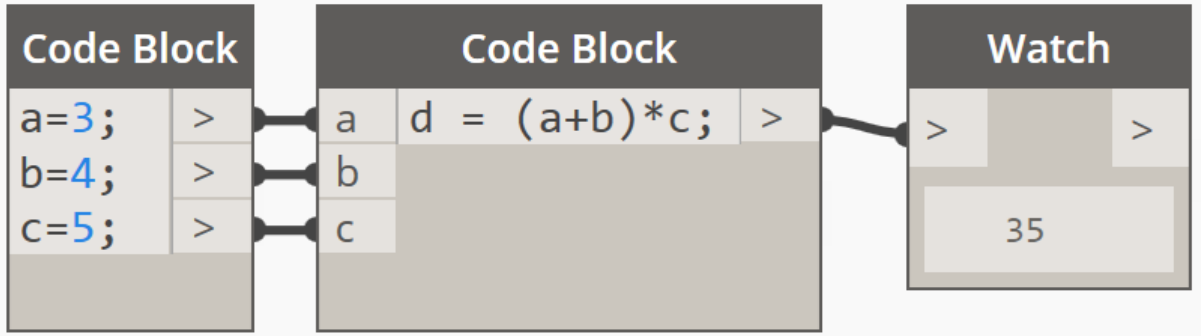


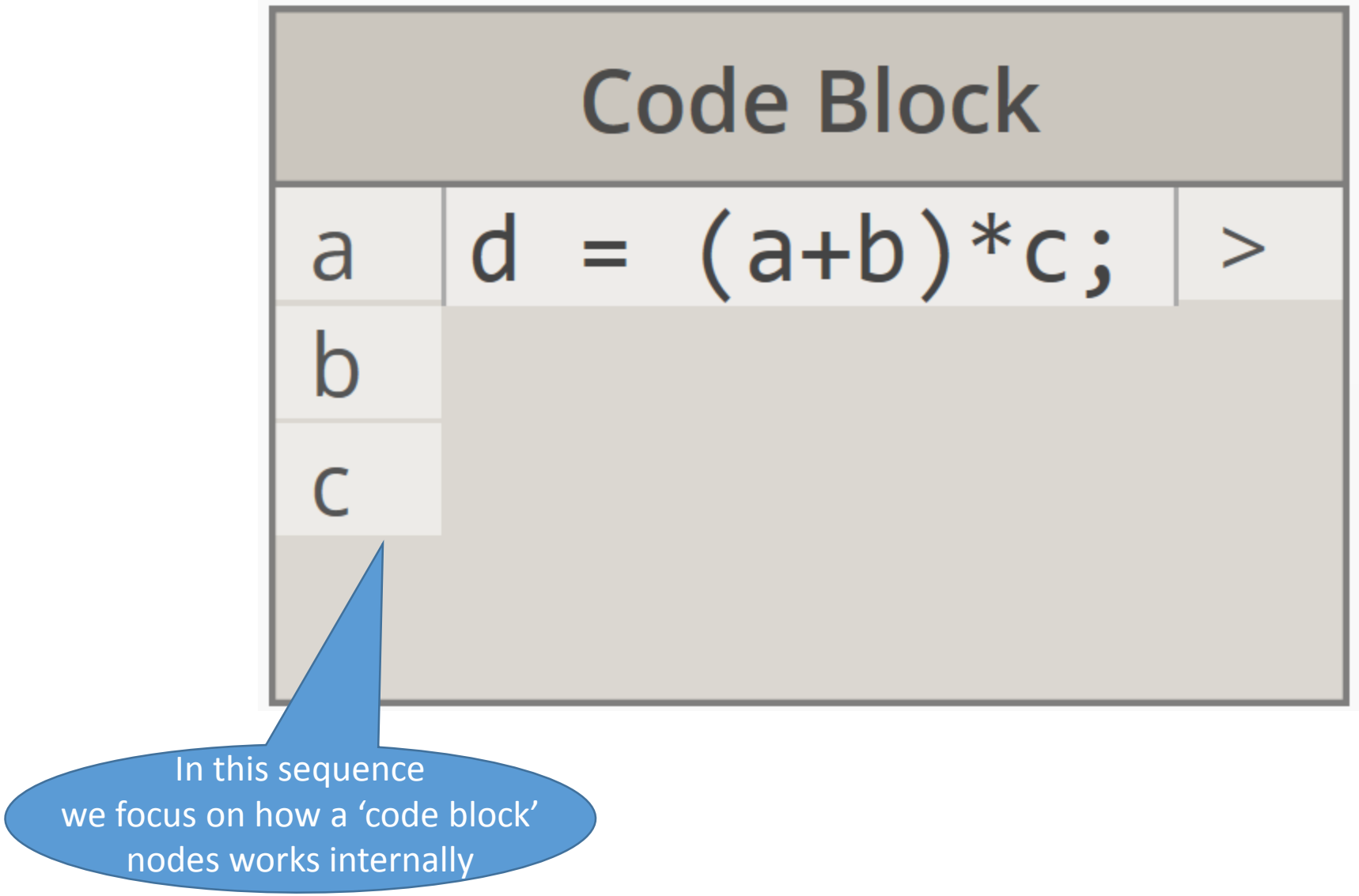
2. Clicking outside the code block node and input and output ‘ports’ are automatically generated



but let’s focus on how a ‘code block’ nodes works internally

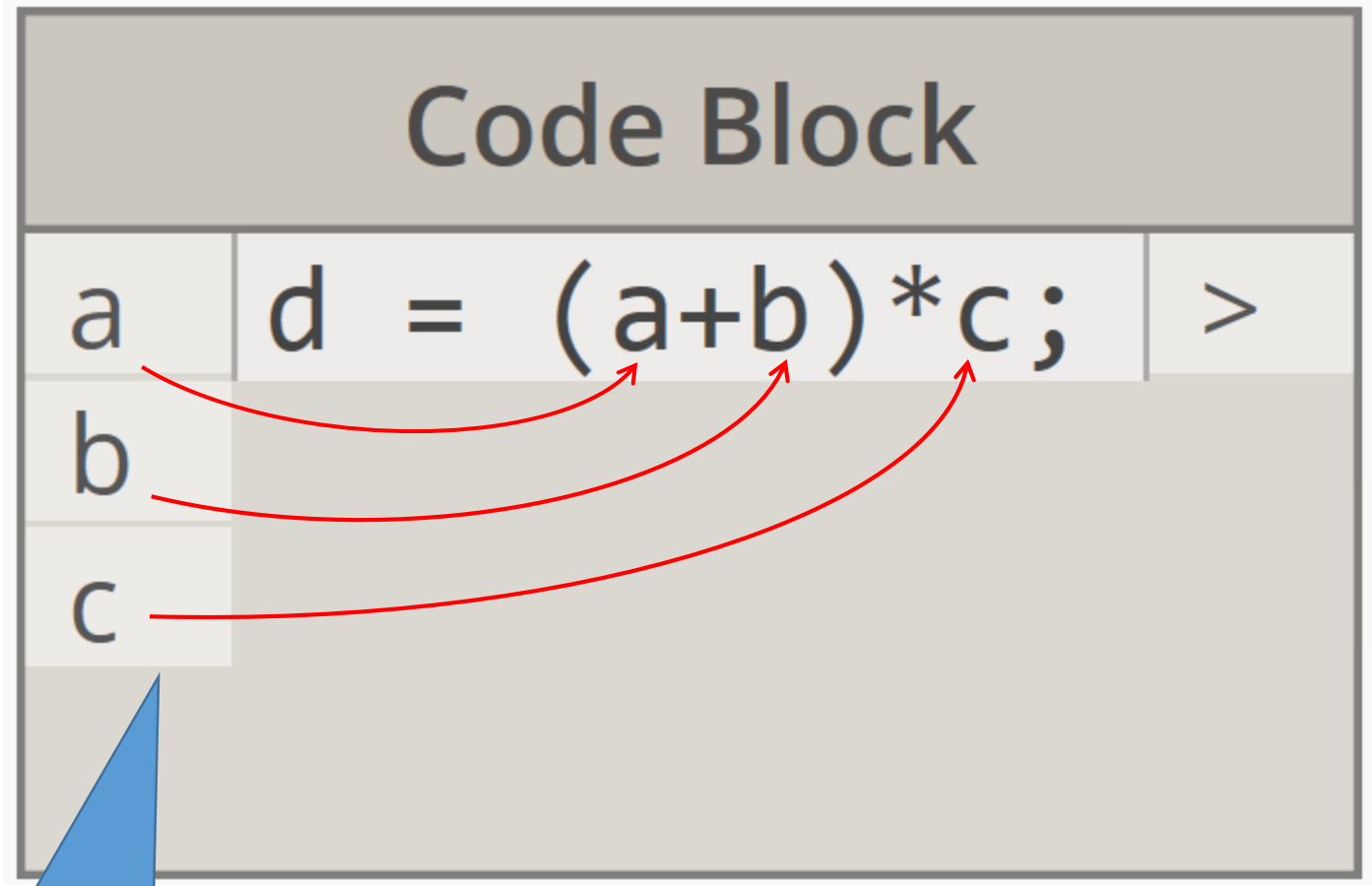
3. Connect up the input and output ports to provide sufficient data to run the program





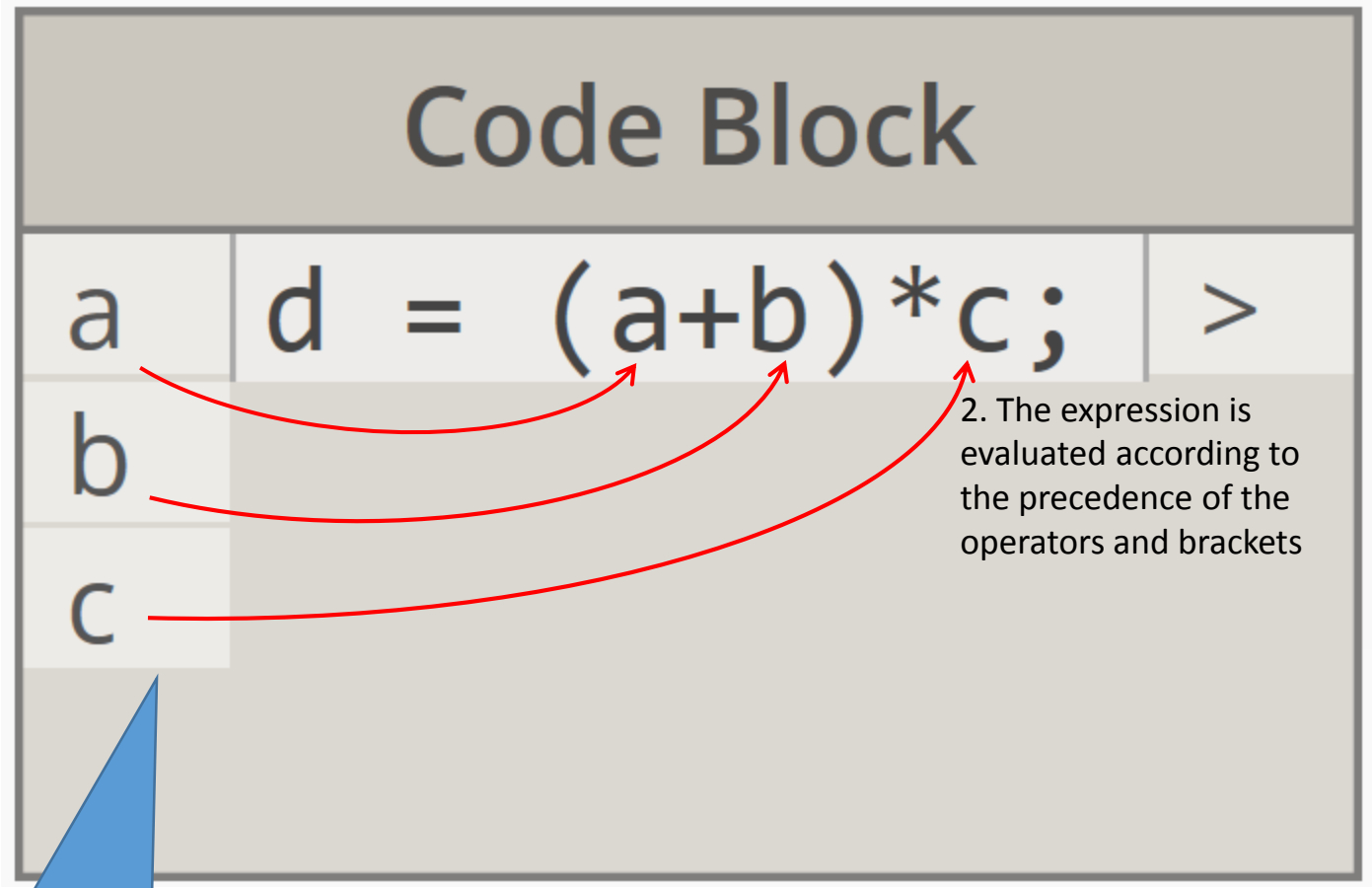
In this sequence
we focus on how a ‘code block’
nodes works internally

1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports



In this sequence
we focus on how a ‘code block’
nodes works internally

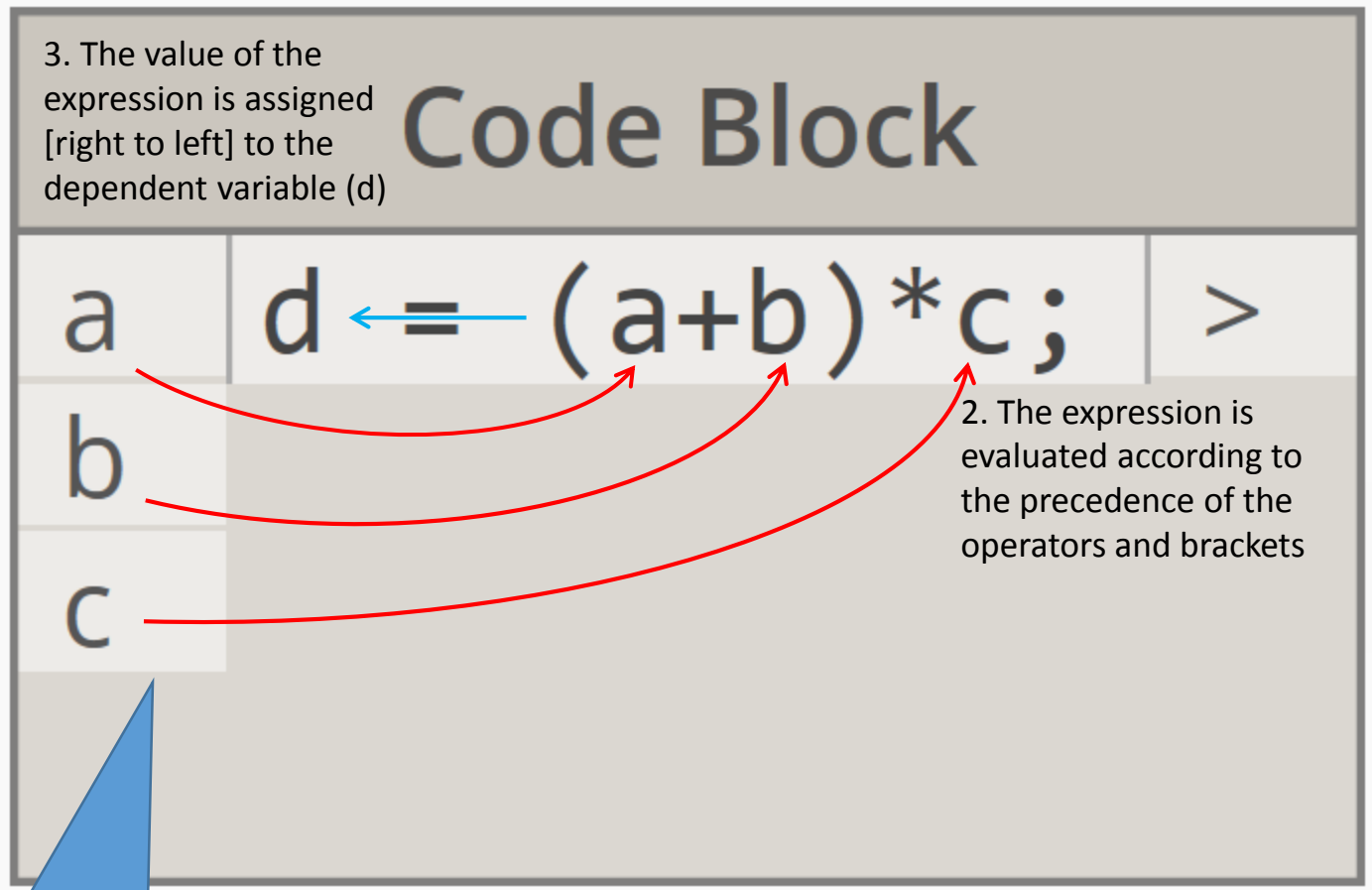
1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports



2. The expression is evaluated according to the precedence of the operators and brackets

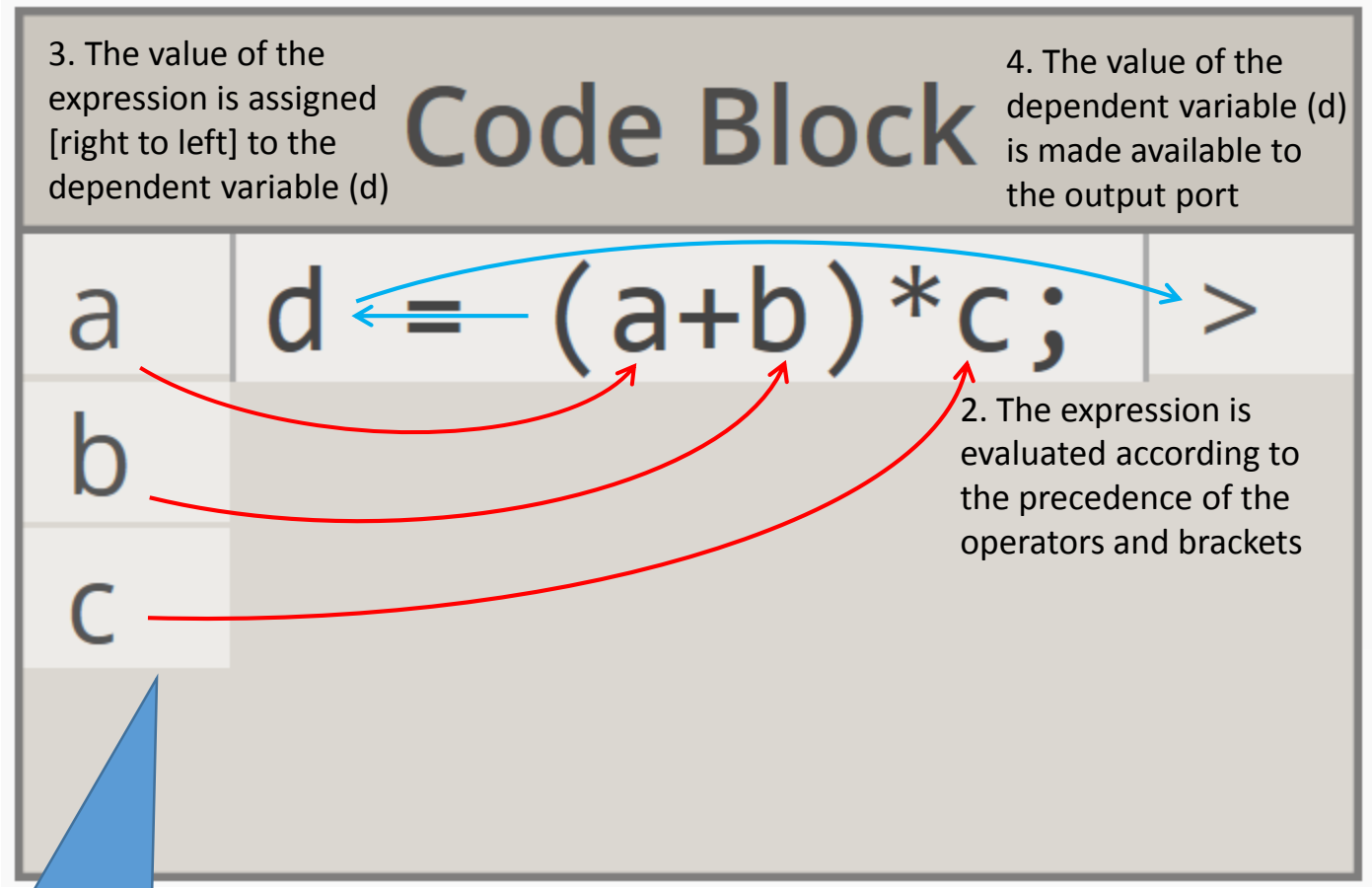
In this sequence
we focus on how a ‘code block’
nodes works internally

1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports



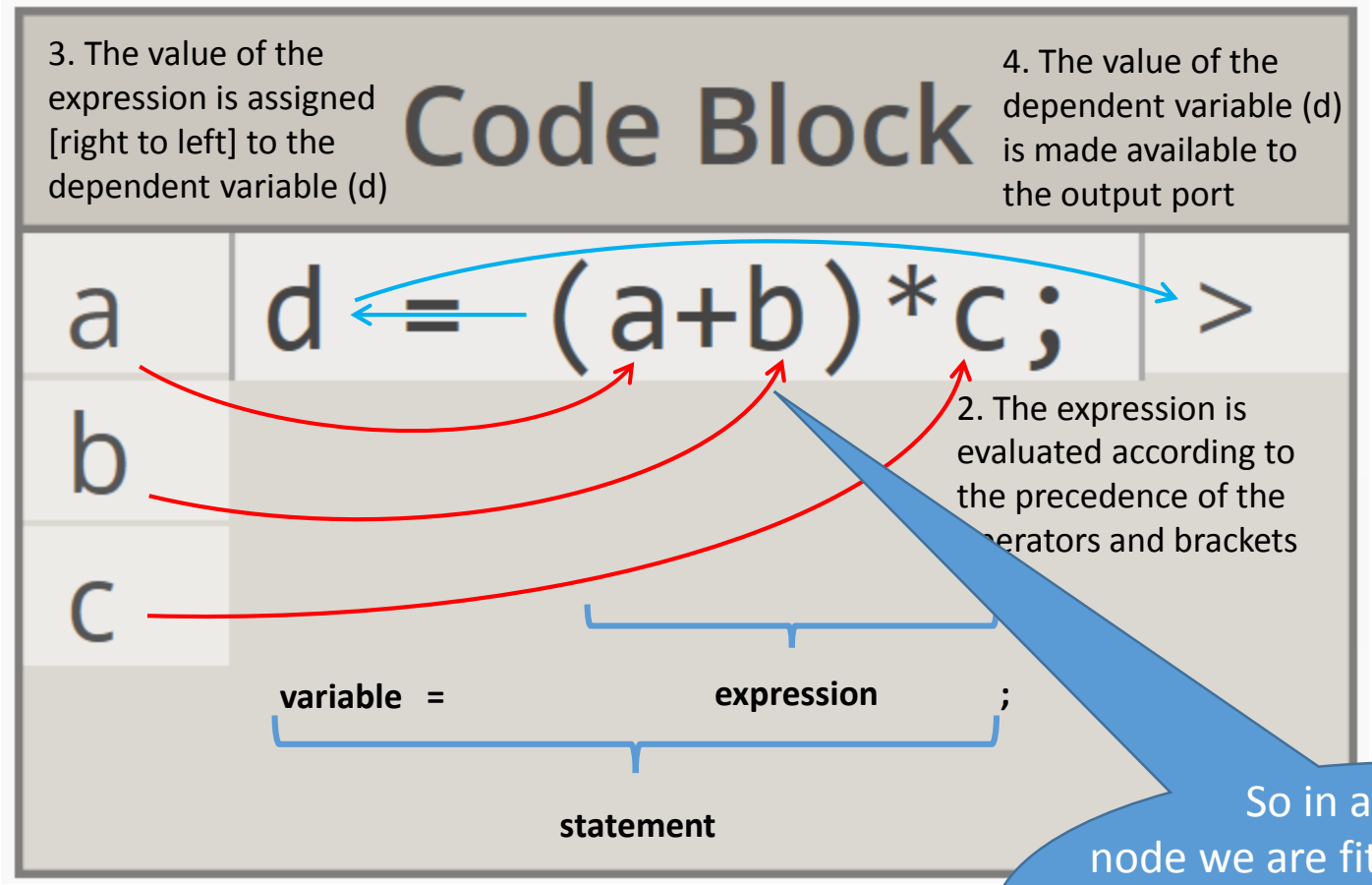
In this sequence
we focus on how a ‘code block’
nodes works internally

1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports

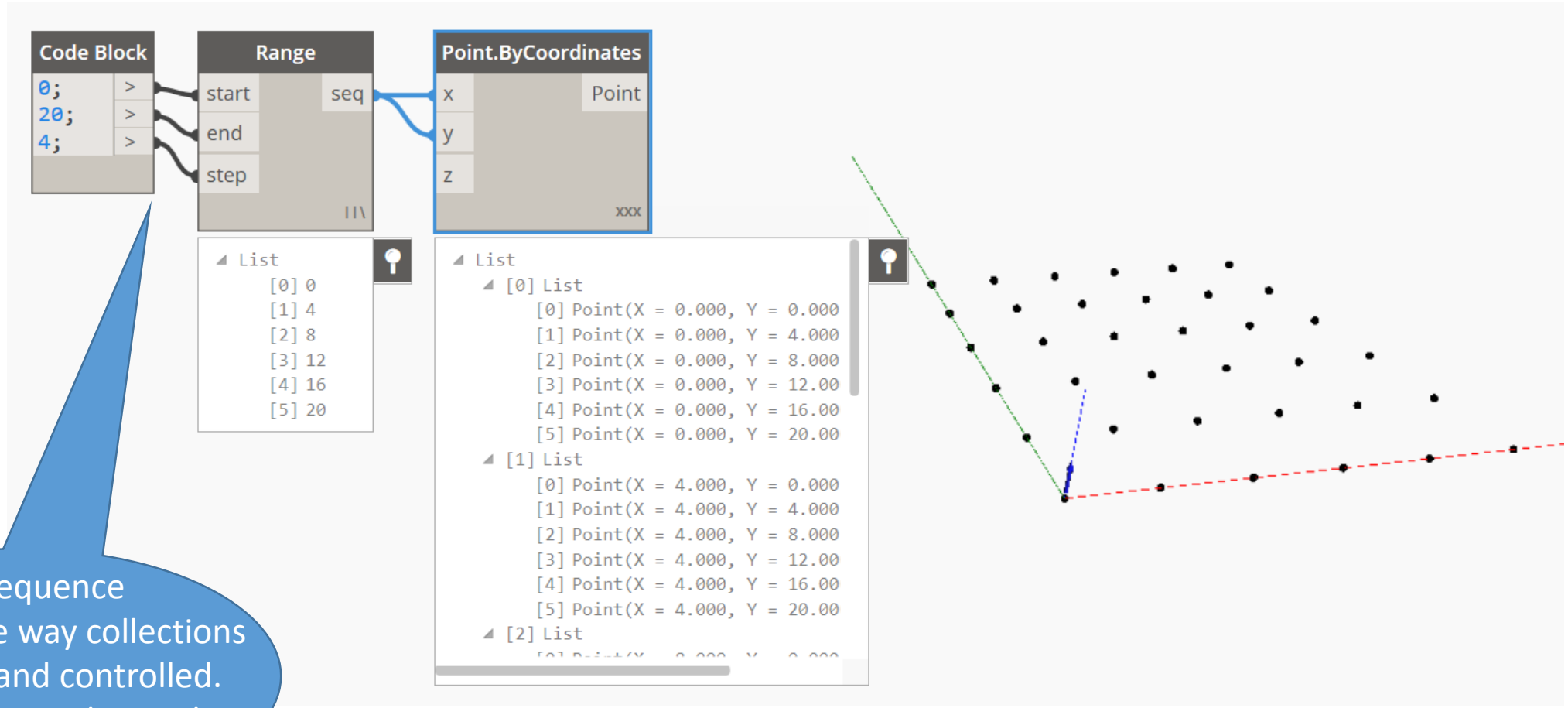


In this sequence
we focus on how a ‘code block’
nodes works internally

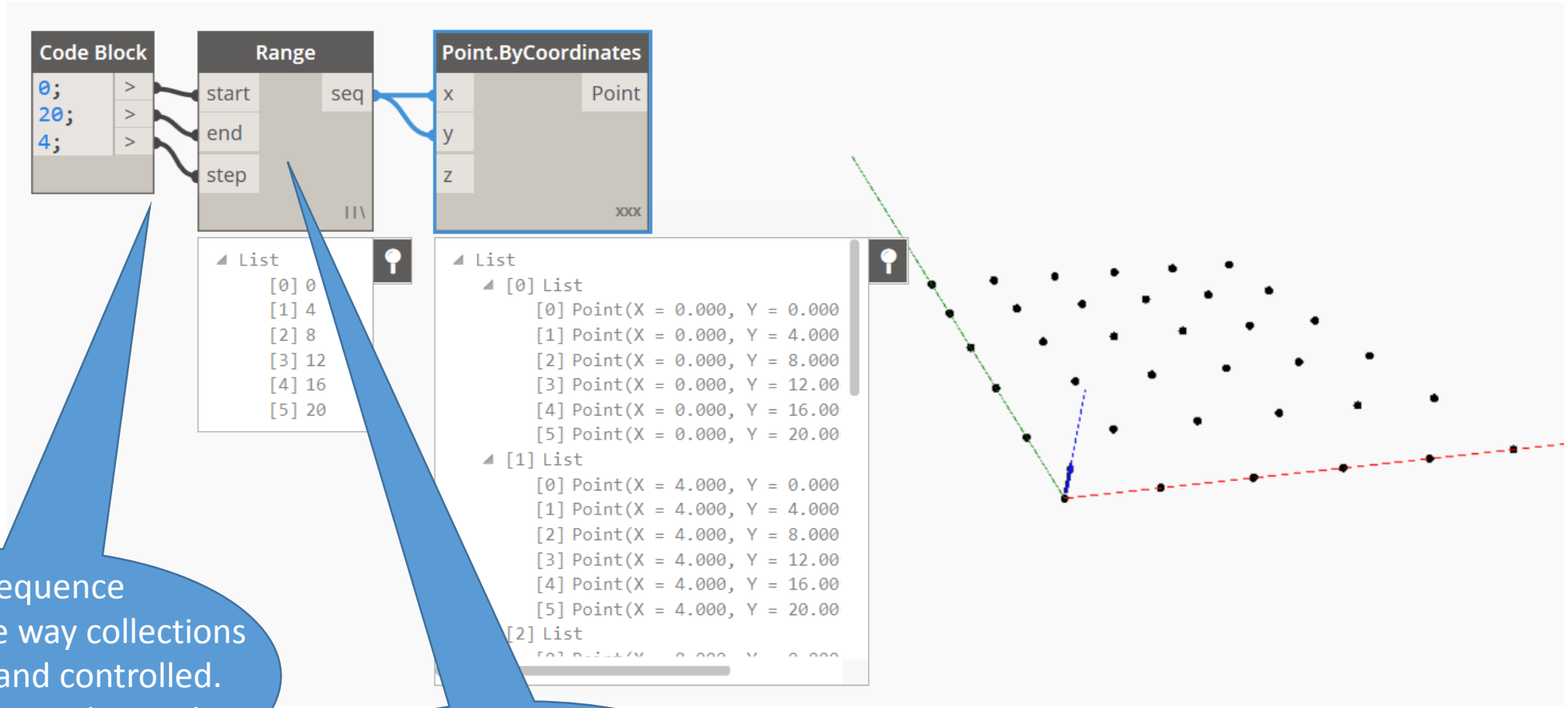
1. The independent variables (a, b, c) in the expression are assigned their respective values from the ‘input’ ports



So in a code block node we are fitting a conventional program statement that operates right to left into the data flow convention that operates left to right

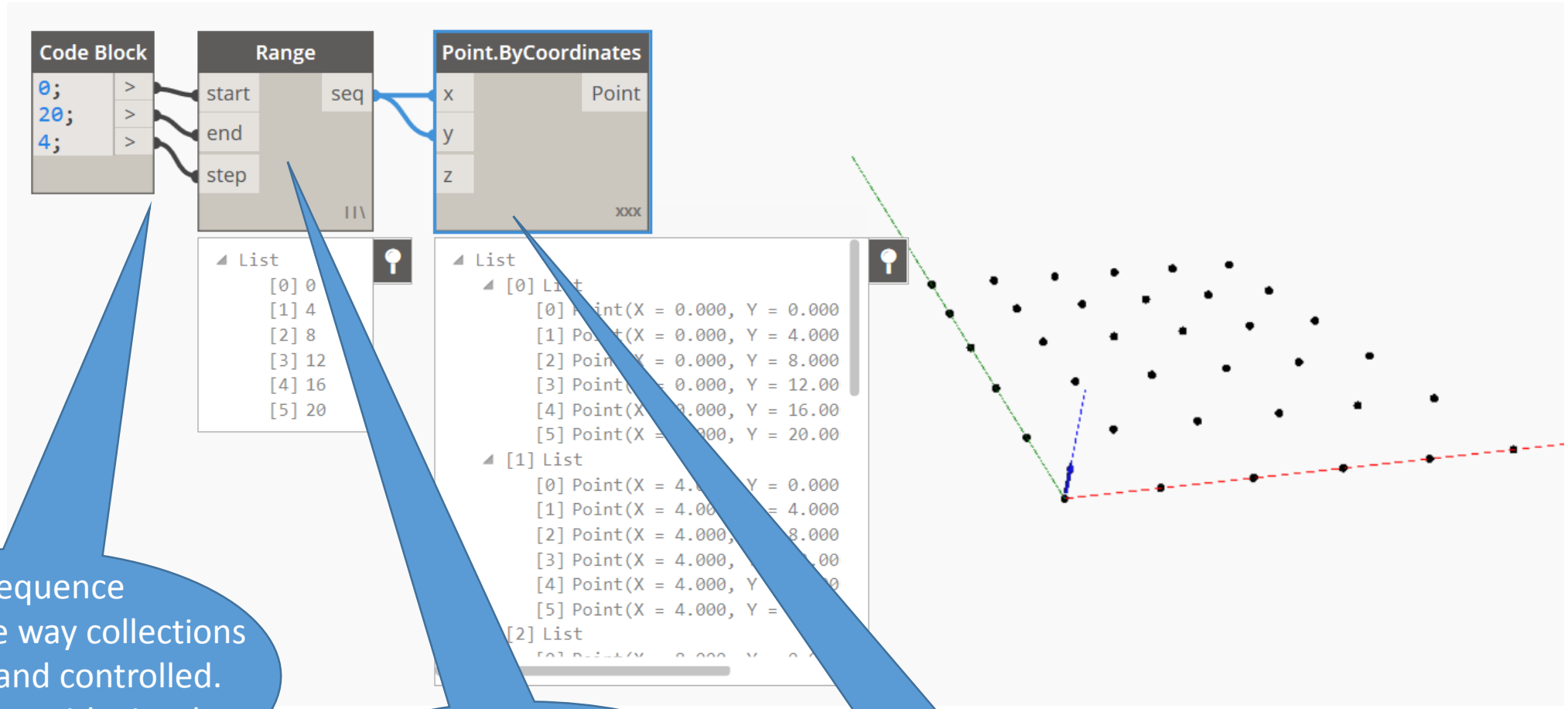


This sequence focuses on the way collections are created and controlled. First we start with visual programming



This sequence focuses on the way collections are created and controlled. First we start with visual programming

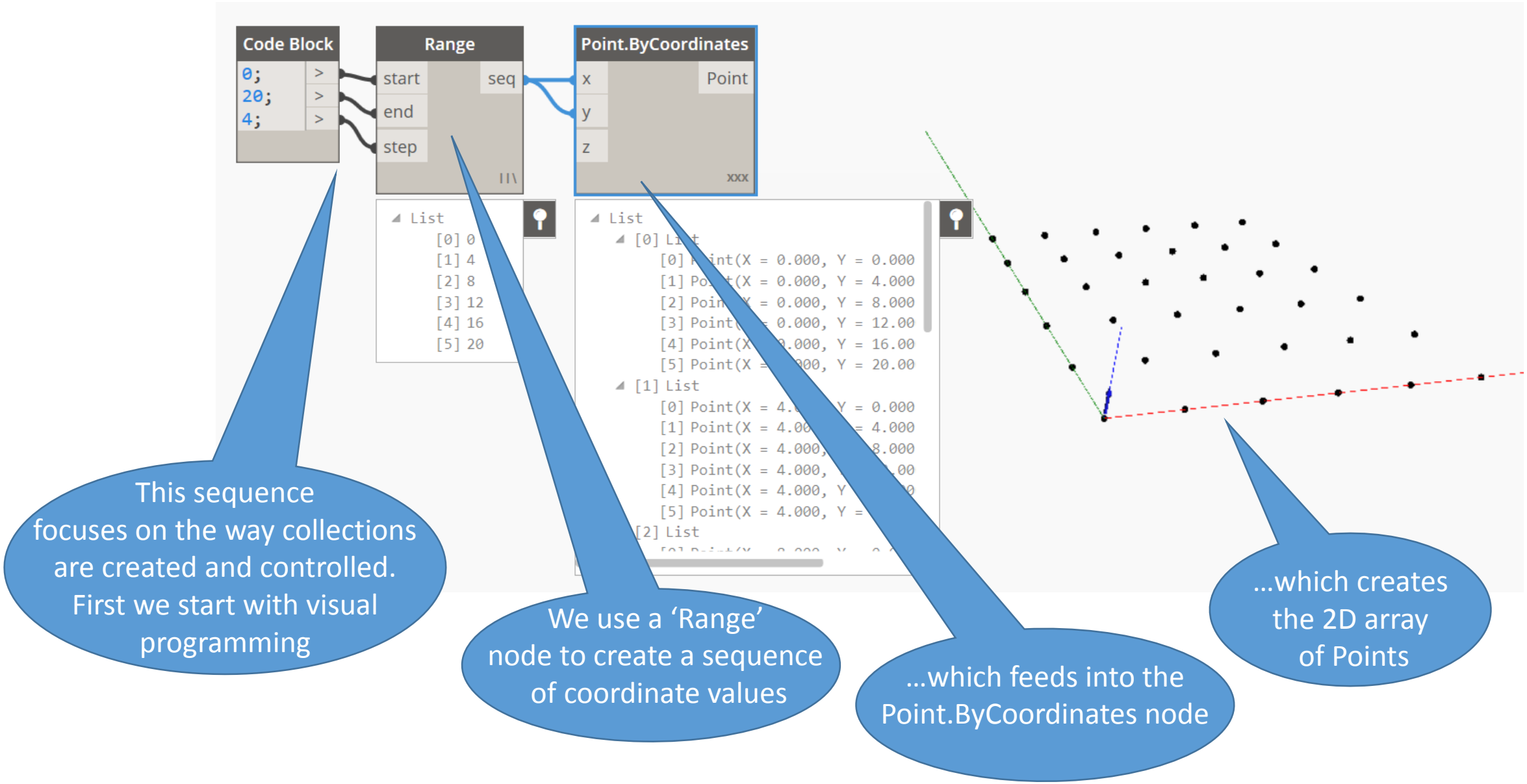
We use a ‘Range’ node to create a sequence of coordinate values

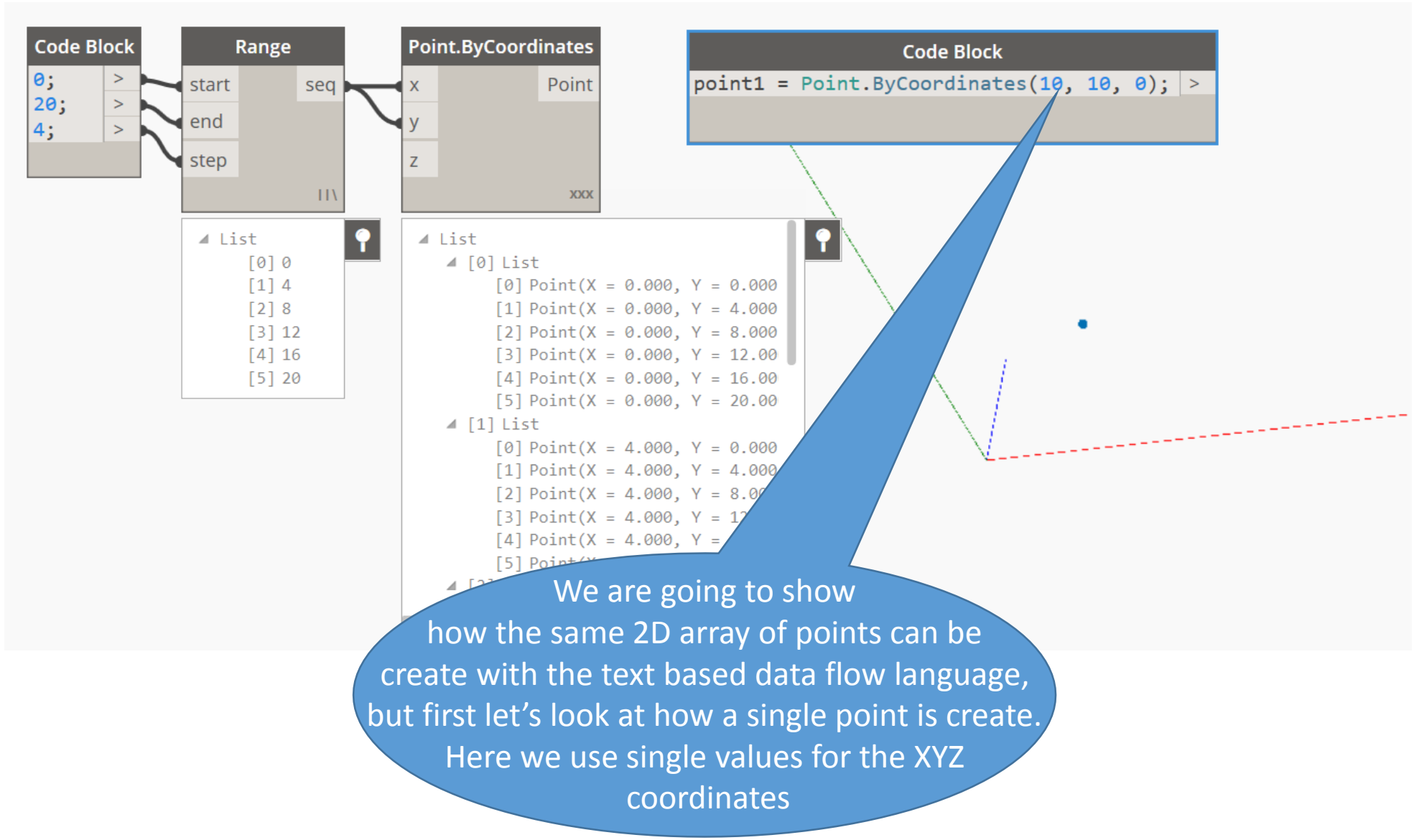


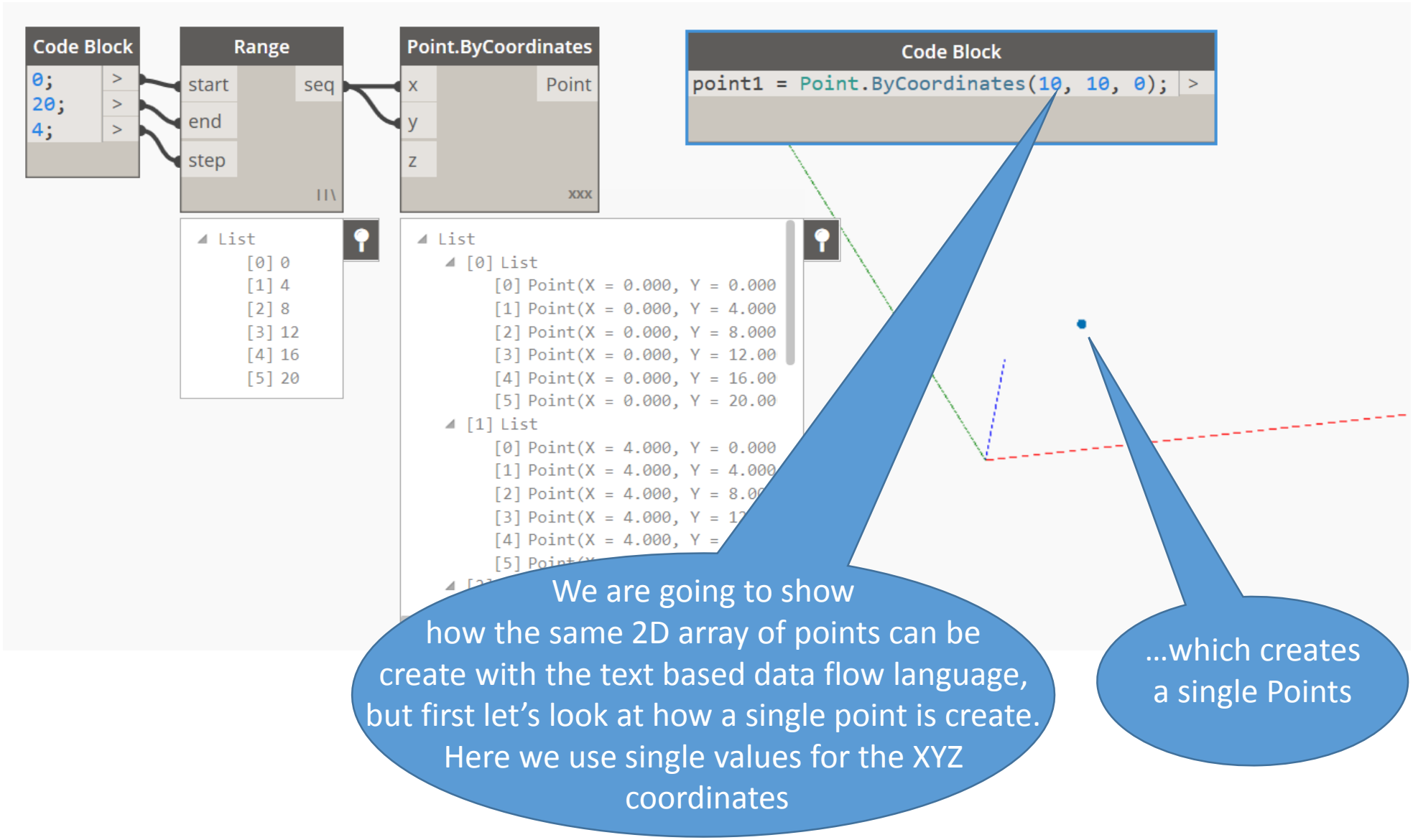
This sequence focuses on the way collections are created and controlled. First we start with visual programming

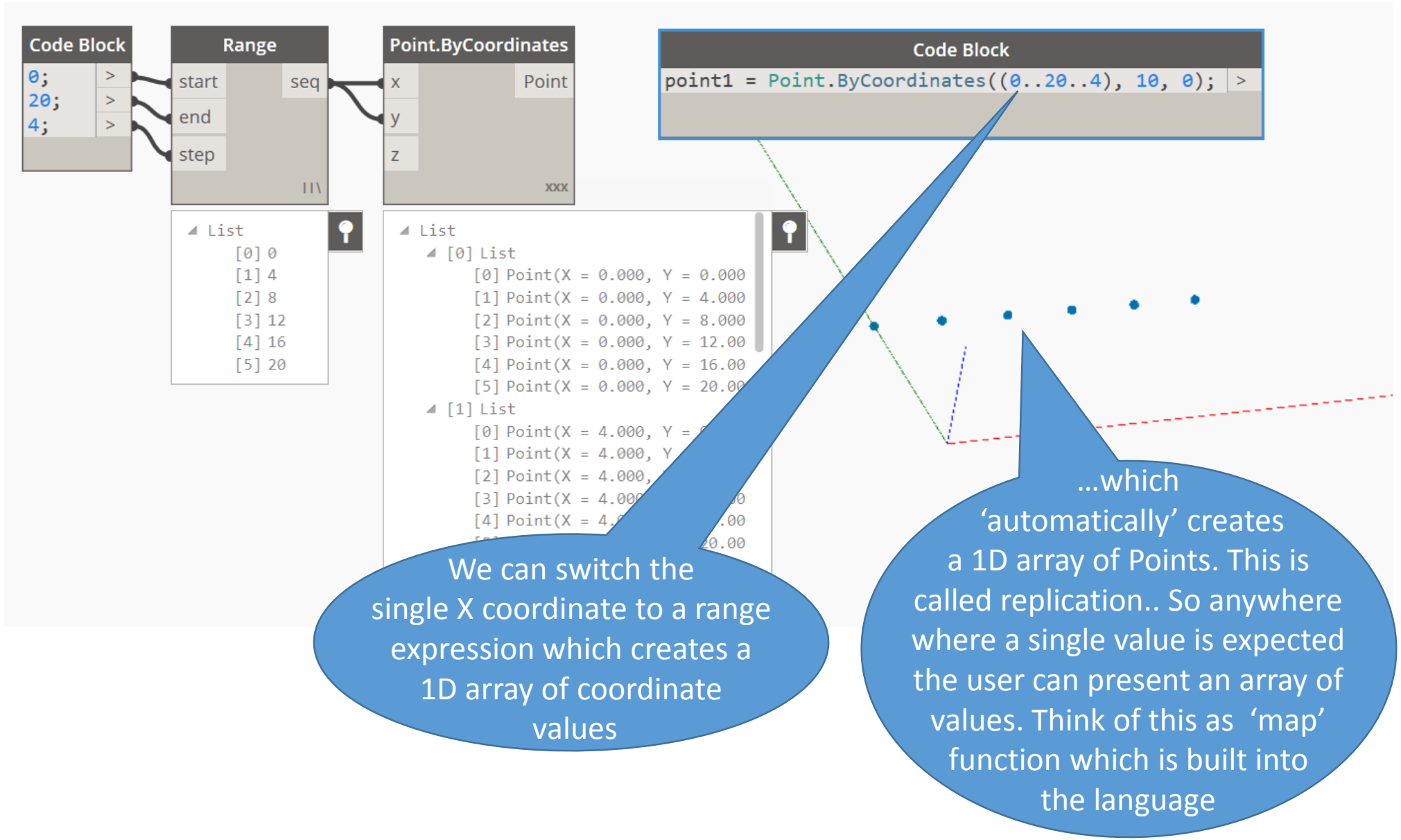
We use a ‘Range’ node to create a sequence of coordinate values

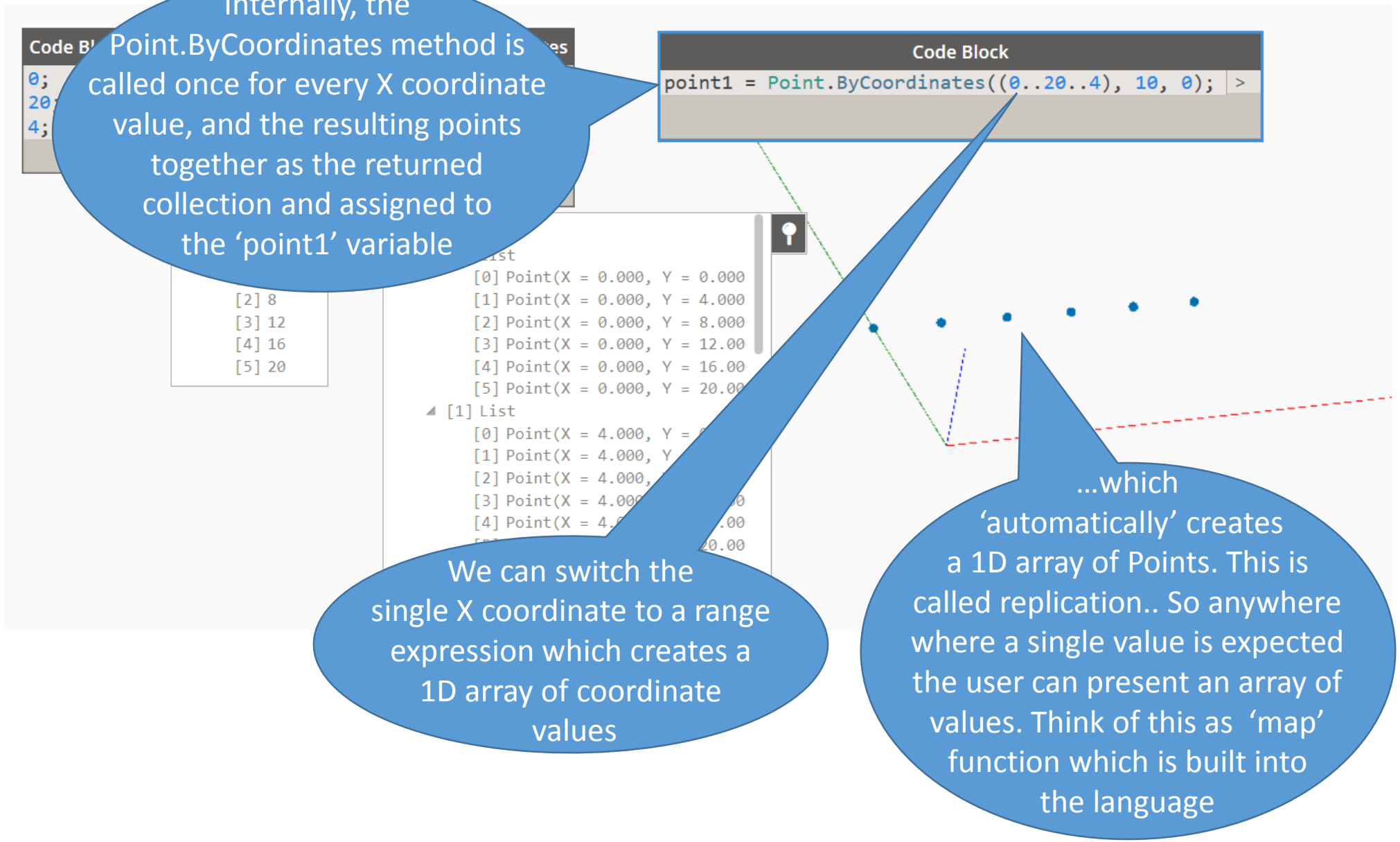
…which feeds into the Point.ByCoordinates node

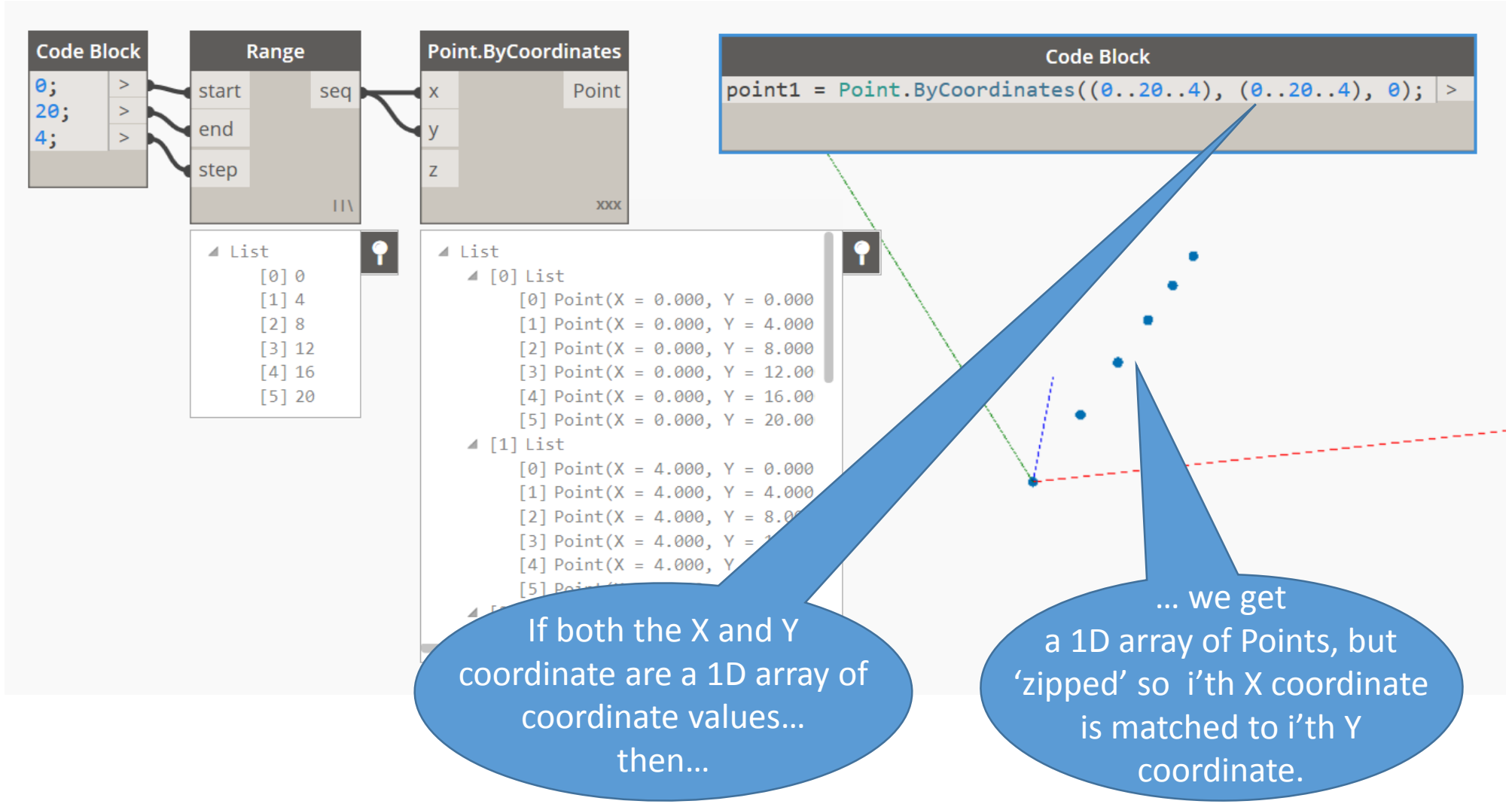

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0.000
[1] Point(X = 4.000, Y = 4.000
[2] Point(X = 4.000, Y = 8.000
[3] Point(X = 4.000, Y = 12.00
[4] Point(X = 4.000, Y = 16.00
[5] Point(X = 4.000, Y = 20.00
[2] List
This sequence focuses on the way collections are created and controlled. First we start with visual programming
We use a ‘Range’ node to create a sequence of coordinate values
…which feeds into the Point.ByCoordinates node
…which creates the 2D array of Points

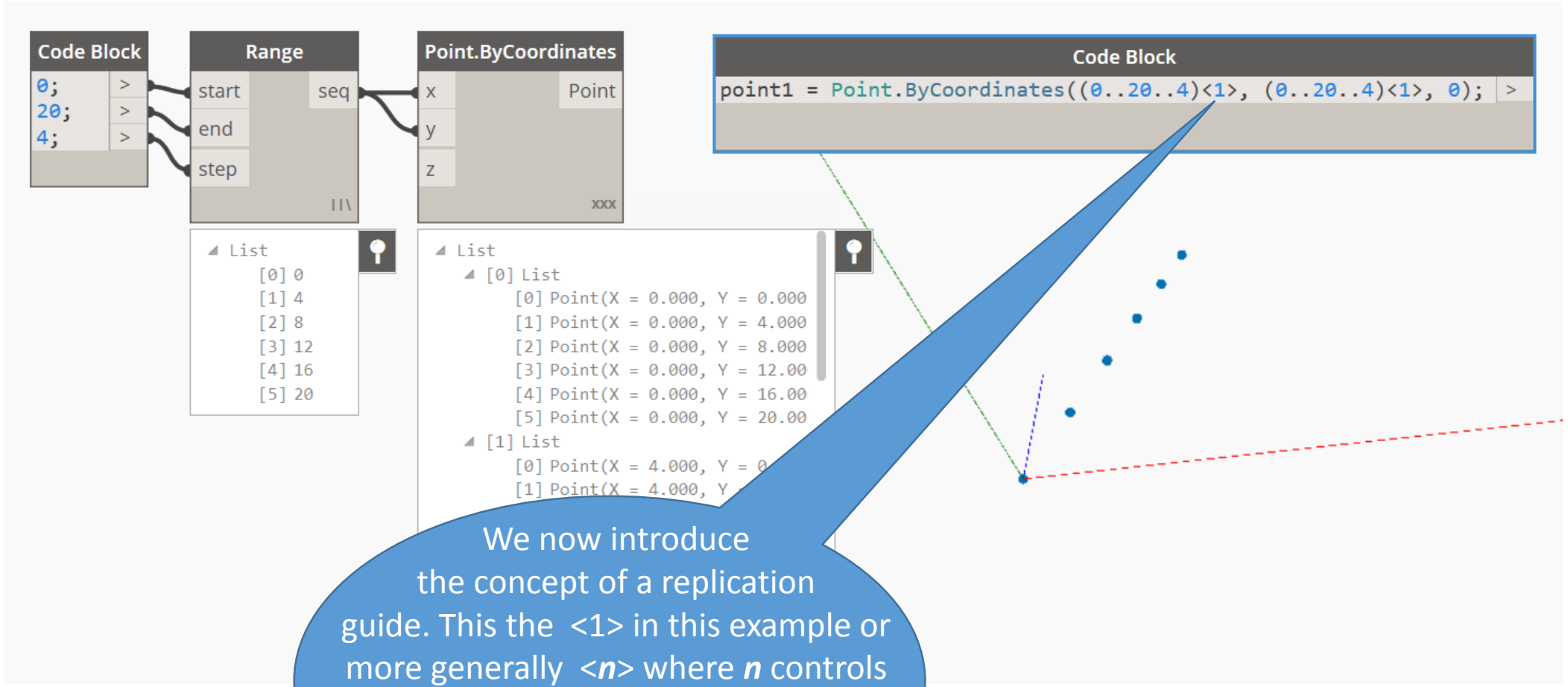

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0.000
[1] Point(X = 4.000, Y = 4.000
Code Block
point1 = Point.ByCoordinates(10, 10, 0);
We are going to show
how the same 2D array of points can be
create with the text based data flow language,
but first let's look at how a single point is create.
Here we use single values for the XYZ
coordinates

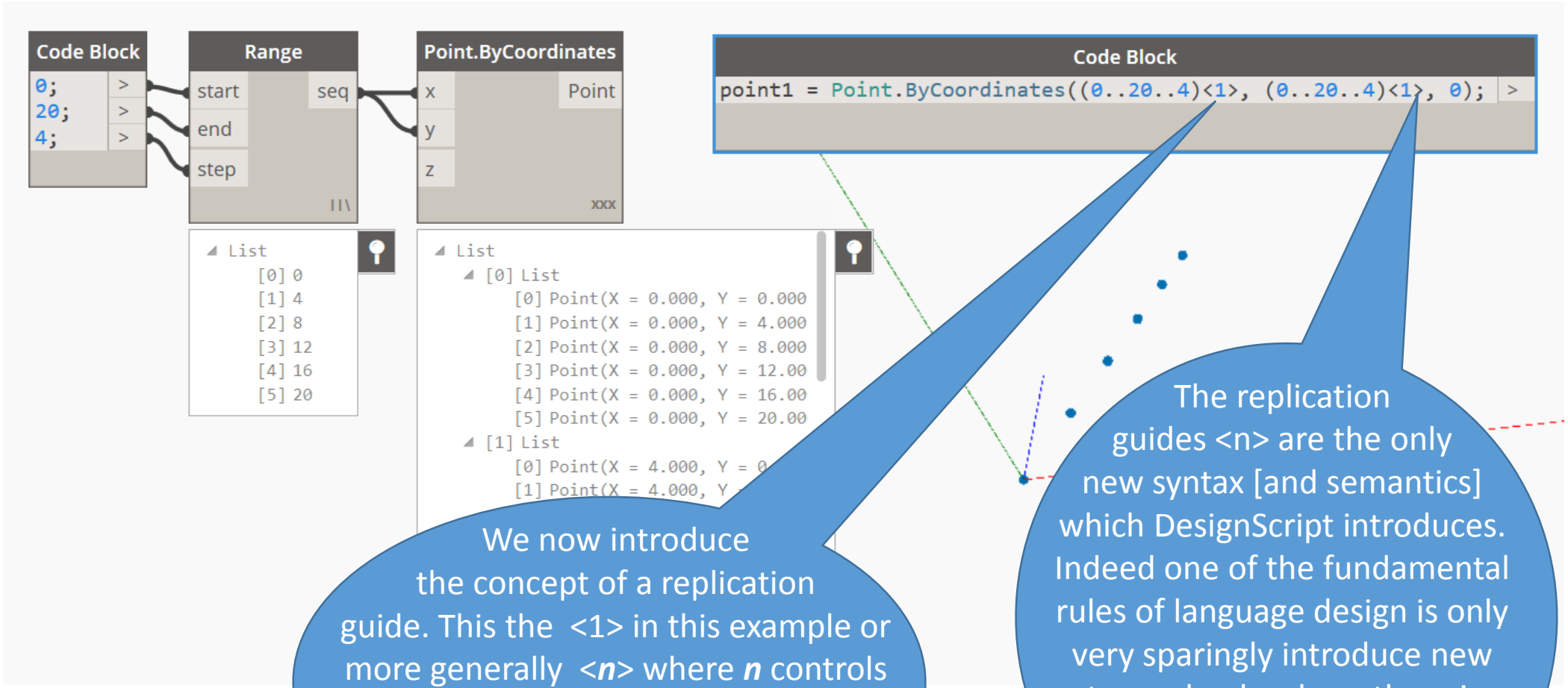

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0.000
[1] Point(X = 4.000, Y = 4.000
Code Block
point1 = Point.ByCoordinates(10, 10, 0);
We are going to show
how the same 2D array of points can be
create with the text based data flow language,
but first let's look at how a single point is create.
Here we use single values for the XYZ
coordinates
...which creates
a single Points

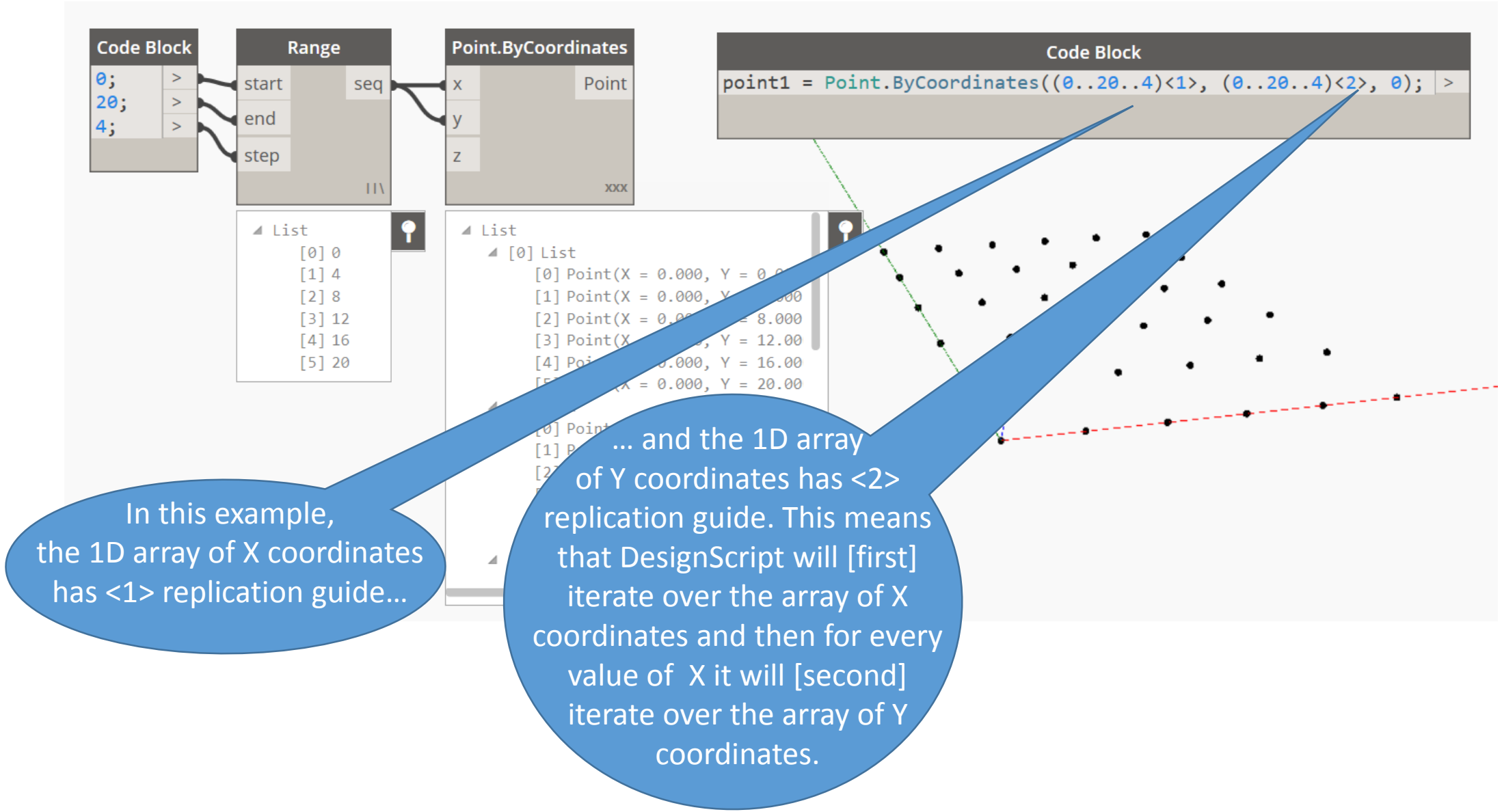

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y =
[1] Point(X = 4.000, Y
[2] Point(X = 4.000,
[3] Point(X = 4.000
[4] Point(X = 4.
Code Block
point1 = Point.ByCoordinates((0..20..4), 10, 0);
We can switch the single X coordinate to a range expression which creates a 1D array of coordinate values
…which ‘automatically’ creates a 1D array of Points. This is called replication.. So anywhere where a single value is expected the user can present an array of values. Think of this as ‘map’ function which is built into the language

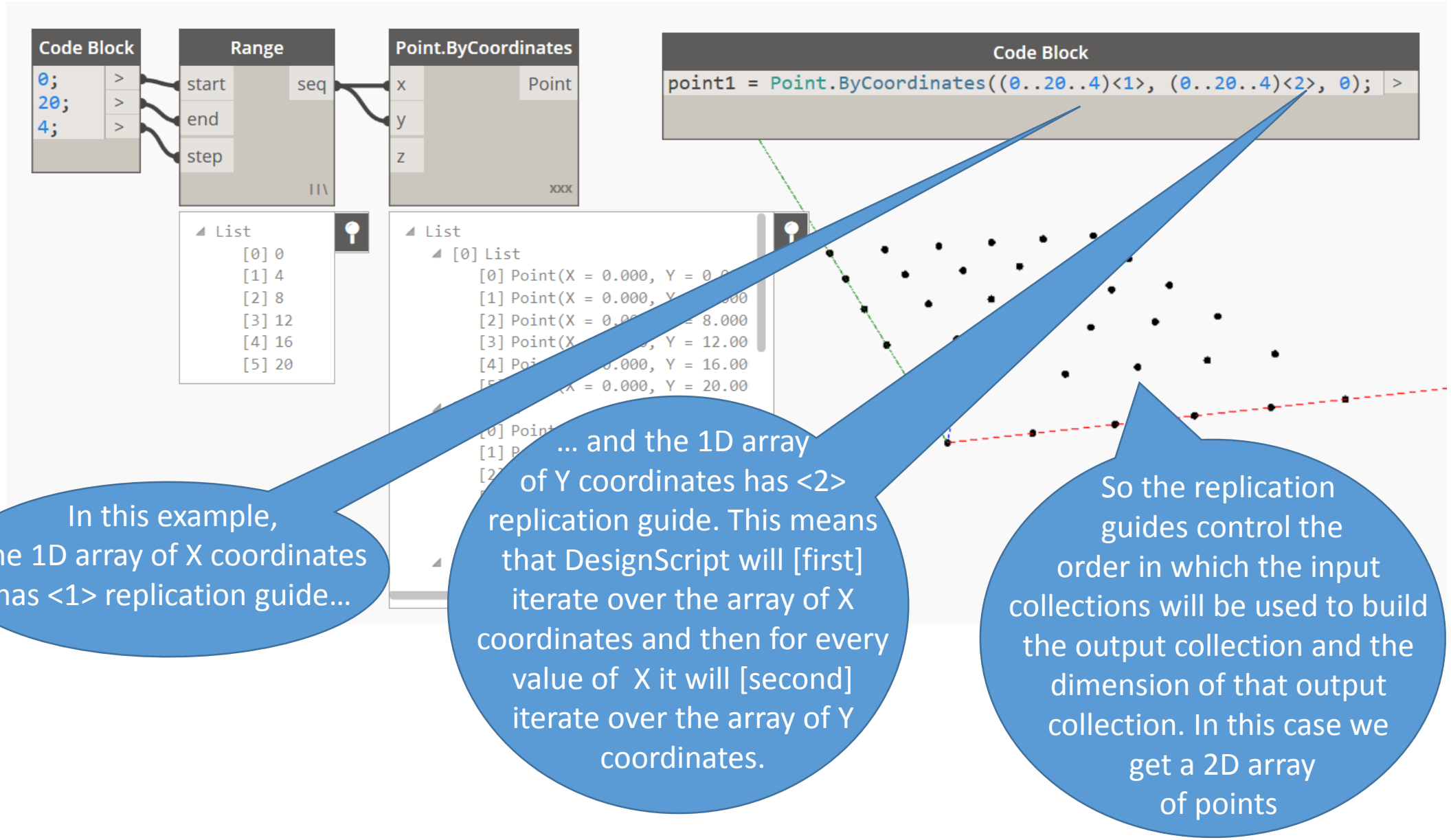

Internally, the Point.ByCoordinates method is called once for every X coordinate value, and the resulting points together as the returned collection and assigned to the ‘point1’ variable
Code Block
point1 = Point.ByCoordinates((0..20..4), 10, 0);
[2] 8
[3] 12
[4] 16
[5] 20
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
We can switch the single X coordinate to a range expression which creates a 1D array of coordinate values
…which ‘automatically’ creates a 1D array of Points. This is called replication.. So anywhere where a single value is expected the user can present an array of values. Think of this as ‘map’ function which is built into the language

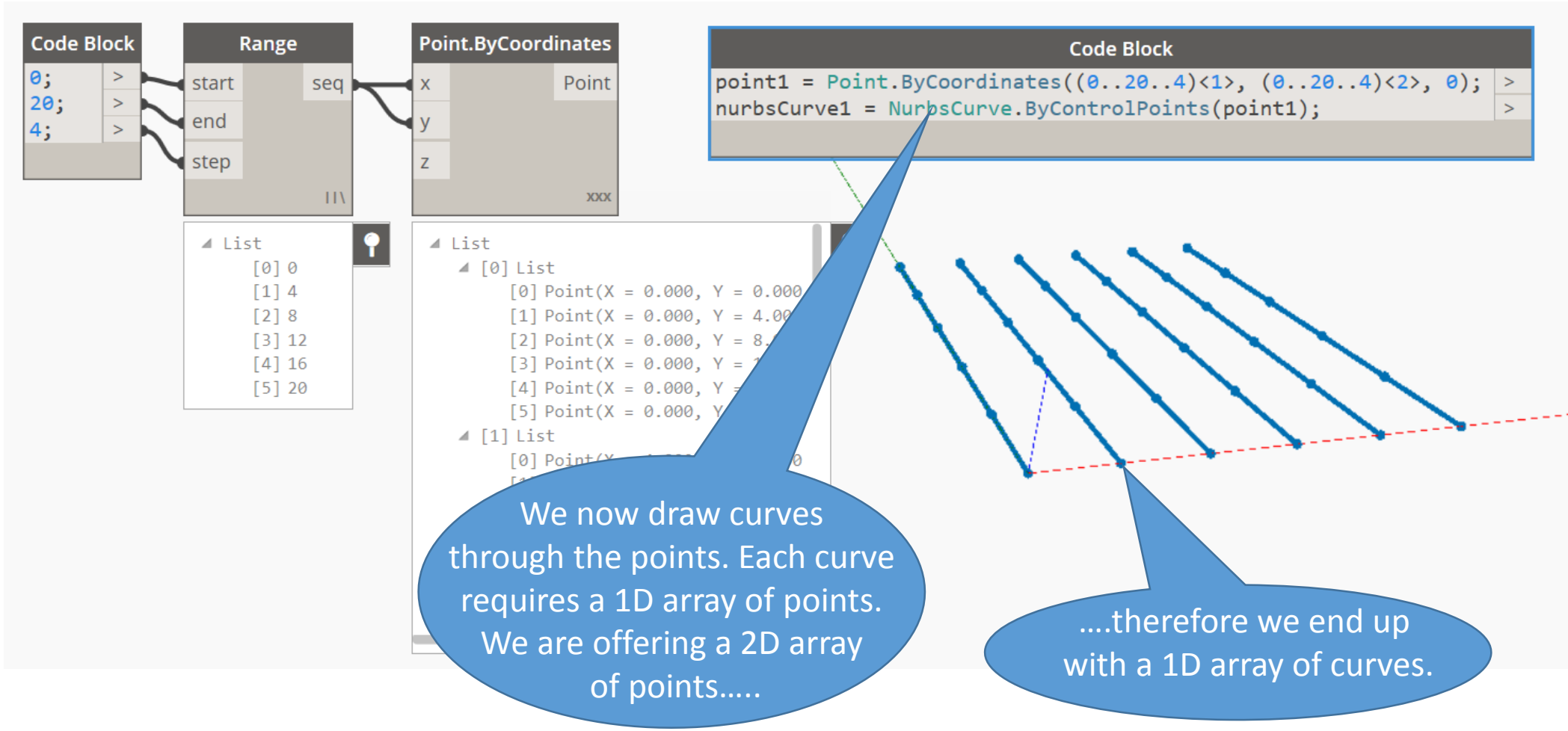

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0.000
[1] Point(X = 4.000, Y = 4.000
Code Block
point1 = Point.ByCoordinates((0..20..4), (0..20..4), 0);
If both the X and Y coordinate are a 1D array of coordinate values… then…
… we get a 1D array of Points, but ‘zipped’ so i’th X coordinate is matched to i’th Y coordinate.

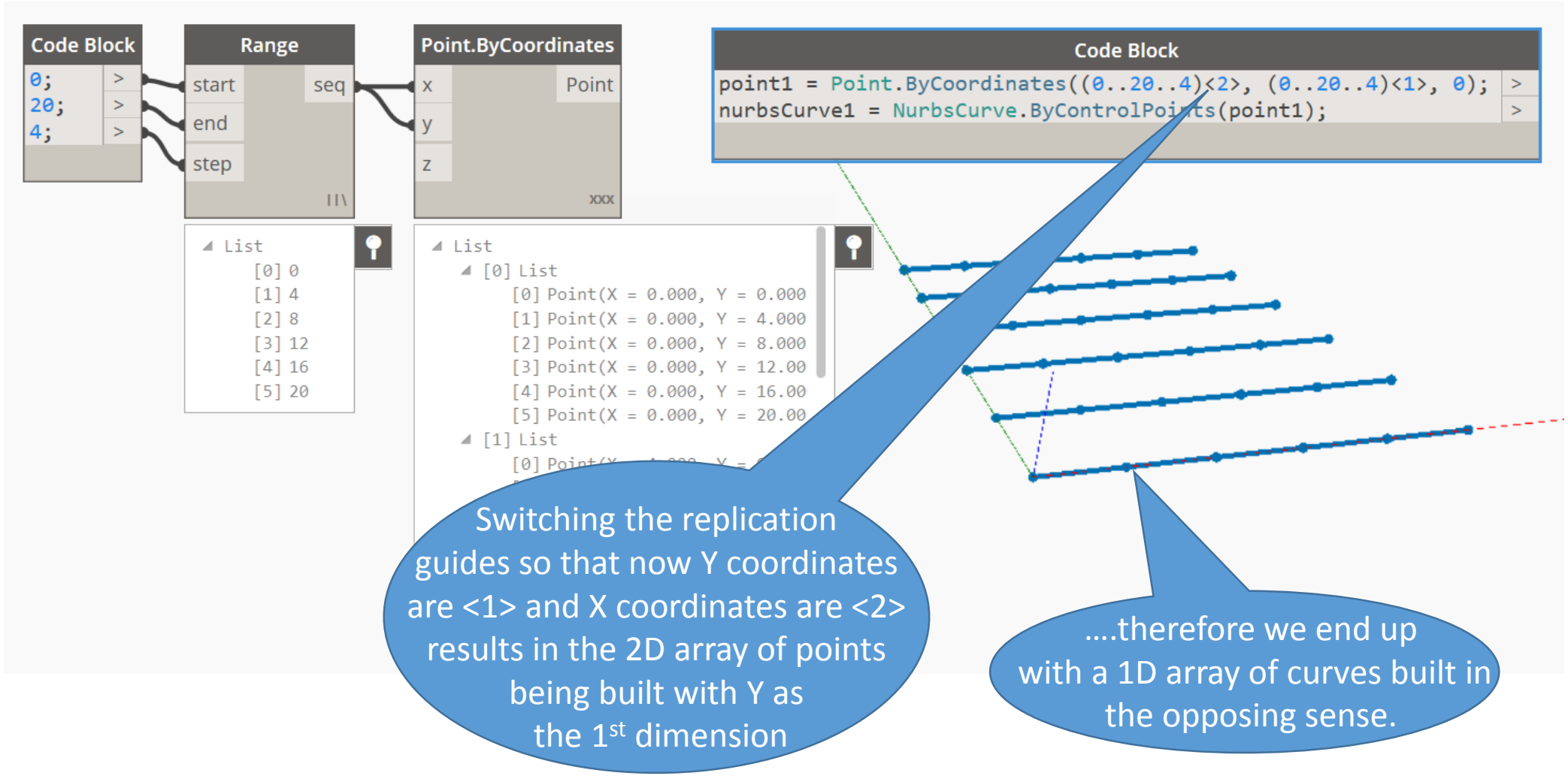

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0
[1] Point(X = 4.000, Y
Code Block
point1 = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<1>, 0);
We now introduce the concept of a replication guide. This the <1> in this example or more generally <n> where n controls how the different sets of inputs are combined to create the 'cartesian product' set of the resulting output


Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
[0] Point(X = 4.000, Y = 0
[1] Point(X = 4.000, Y
Code Block
point1 = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<1>, 0);
We now introduce the concept of a replication guide. This the <1> in this example or more generally <n> where n controls how the different sets of inputs are combined to create the 'cartesian product' set of the resulting output
The replication guides <n> are the only new syntax [and semantics] which DesignScript introduces. Indeed one of the fundamental rules of language design is only very sparingly introduce new syntax and only where there is an overwhelming reason. Where possible re-use well established syntax.


Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
Code Block
point1 = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<2>, 0);
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000, Z = 0.000)
[1] Point(X = 0.000, Y = 4.000, Z = 0.000)
[2] Point(X = 0.000, Y = 8.000, Z = 0.000)
[3] Point(X = 0.000, Y = 12.000, Z = 0.000)
[4] Point(X = 0.000, Y = 16.000, Z = 0.000)
[5] Point(X = 0.000, Y = 20.000, Z = 0.000)
In this example, the 1D array of X coordinates has <1> replication guide…
… and the 1D array of Y coordinates has <2> replication guide. This means that DesignScript will [first] iterate over the array of X coordinates and then for every value of X it will [second] iterate over the array of Y coordinates.


Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
Code Block
point1 = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<2>, 0);
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000, Z = 0.000)
[1] Point(X = 0.000, Y = 4.000, Z = 0.000)
[2] Point(X = 0.000, Y = 8.000, Z = 0.000)
[3] Point(X = 0.000, Y = 12.000, Z = 0.000)
[4] Point(X = 0.000, Y = 16.000, Z = 0.000)
[5] Point(X = 0.000, Y = 20.000, Z = 0.000)
In this example, the 1D array of X coordinates has <1> replication guide…
… and the 1D array of Y coordinates has <2> replication guide. This means that DesignScript will [first] iterate over the array of X coordinates and then for every value of X it will [second] iterate over the array of Y coordinates.
So the replication guides control the order in which the input collections will be used to build the output collection and the dimension of that output collection. In this case we get a 2D array of points


Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.00
[2] Point(X = 0.000, Y = 8.
[3] Point(X = 0.000, Y =
[4] Point(X = 0.000, Y =
[5] Point(X = 0.000, Y
[1] List
[0] Point(
Code Block
point1 = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<2>, 0);
nurbsCurve1 = NurbsCurve.ByControlPoints(point1);
We now draw curves through the points. Each curve requires a 1D array of points. We are offering a 2D array of points.....
....therefore we end up with a 1D array of curves.

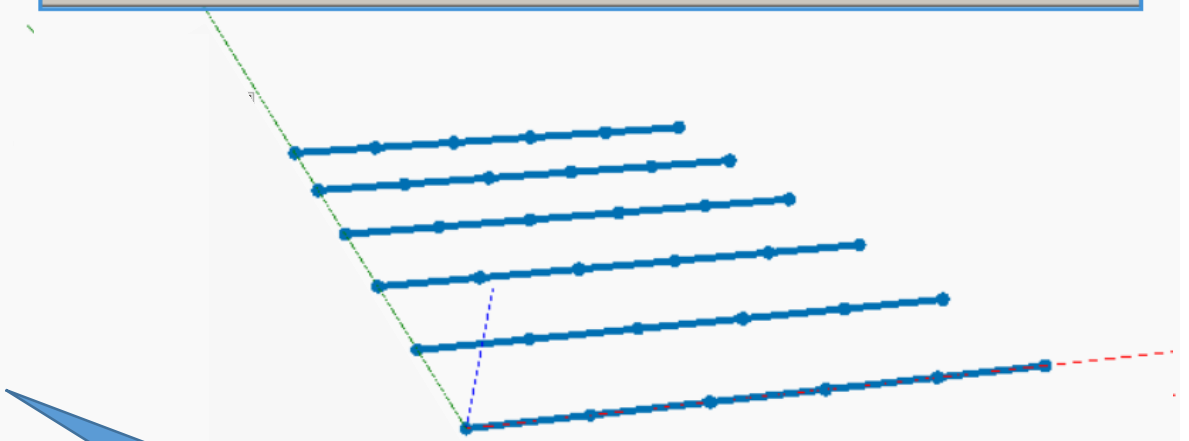

Code Block
0;
20;
4;
Range
start
end
step
seq
Point.ByCoordinates
x
y
z
Point
xxx
List
[0] 0
[1] 4
[2] 8
[3] 12
[4] 16
[5] 20
List
[0] List
[0] Point(X = 0.000, Y = 0.000
[1] Point(X = 0.000, Y = 4.000
[2] Point(X = 0.000, Y = 8.000
[3] Point(X = 0.000, Y = 12.00
[4] Point(X = 0.000, Y = 16.00
[5] Point(X = 0.000, Y = 20.00
[1] List
Code Block
point1 = Point.ByCoordinates((0..20..4)<2>, (0..20..4)<1>, 0);
nurbsCurve1 = NurbsCurve.ByControlPoints(point1);
Switching the replication guides so that now Y coordinates are <1> and X coordinates are <2> results in the 2D array of points being built with Y as the 1st dimension
....therefore we end up with a 1D array of curves built in the opposing sense.

Code Block

```
pointArray  = {};
nurbsCurve1 = {};
[Imperative]
{
    xRange = (0..20..4);
    yRange = (0..20..4);

    xCount = Count(xRange);
    yCount = Count(yRange);

    for (i in 0..(xCount-1))
    {
        for (j in 0..(yCount-1))
        {
            pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
        }
    }

    for (i in 0..(xCount-1))
    {
        nurbsCurve1[i] = NurbsCurve.ByPoints(pointArray[i]);
    }
};
```

Code Block

```
point1 = Point.ByCoordinates((0..20..4)<2>, (0..20..4)<1>, 0);
nurbsCurve1 = NurbsCurve.ByControlPoints(point1);
```



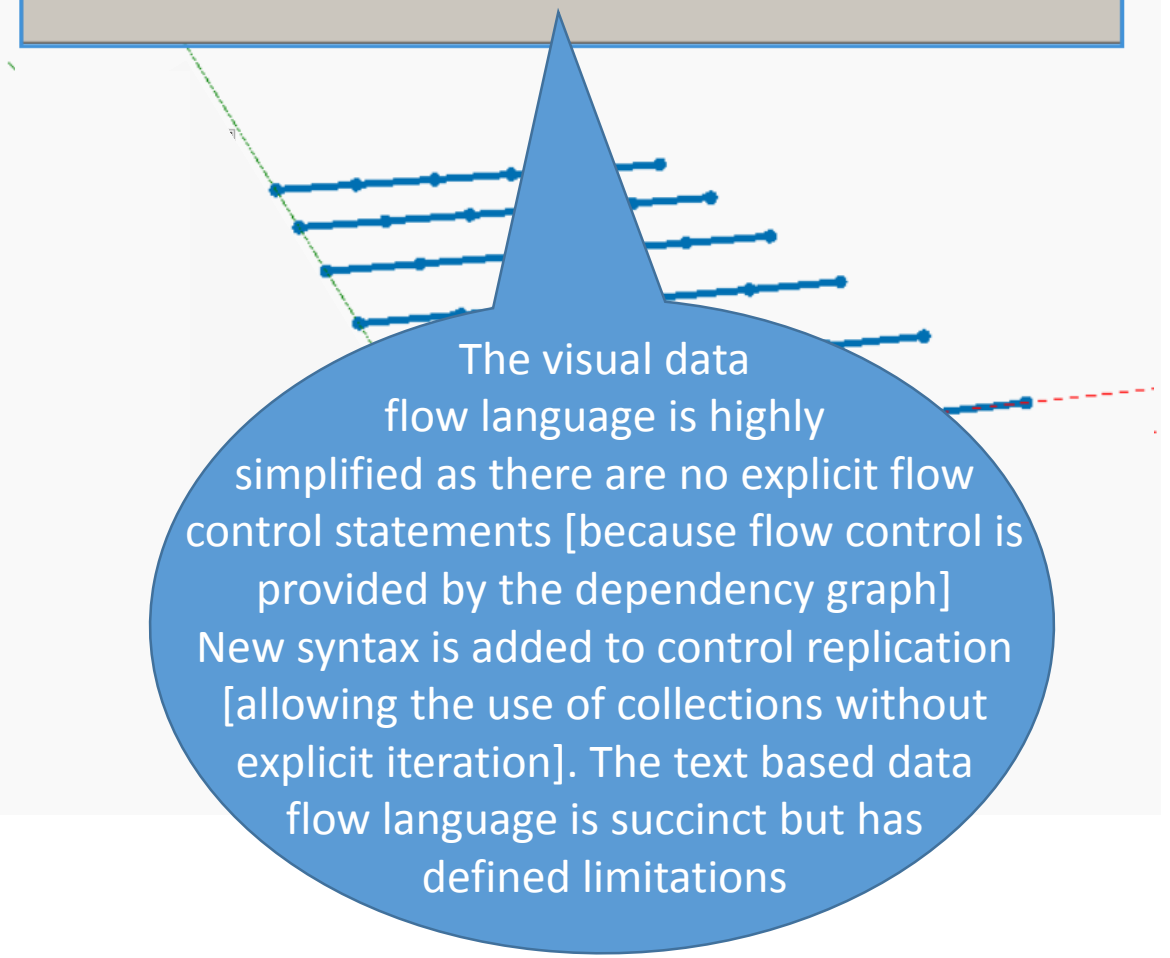
It is of course possible to build exactly the same example using Imperative programming. So what conclusion can we draw?

Code Block

```
pointArray  = {};
nurbsCurve1 = {};
[Imperative]
{
    xRange = (0..20..4);
    yRange = (0..20..4);

    xCount = Count(xRange);
    yCount = Count(yRange);

    for (i in 0..(xCount-1))
    {
        for (j in 0..(yCount-1))
        {
            pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
        }
    }

    for (i in 0..(xCount-1))
    {
        nurbsCurve1[i] = NurbsCurve.ByPoints(pointArray[i]);
    }
};
```

Code Block

```
point1 = Point.ByCoordinates((0..20..4)<2>, (0..20..4)<1>, 0);
nurbsCurve1 = NurbsCurve.ByControlPoints(point1);
```



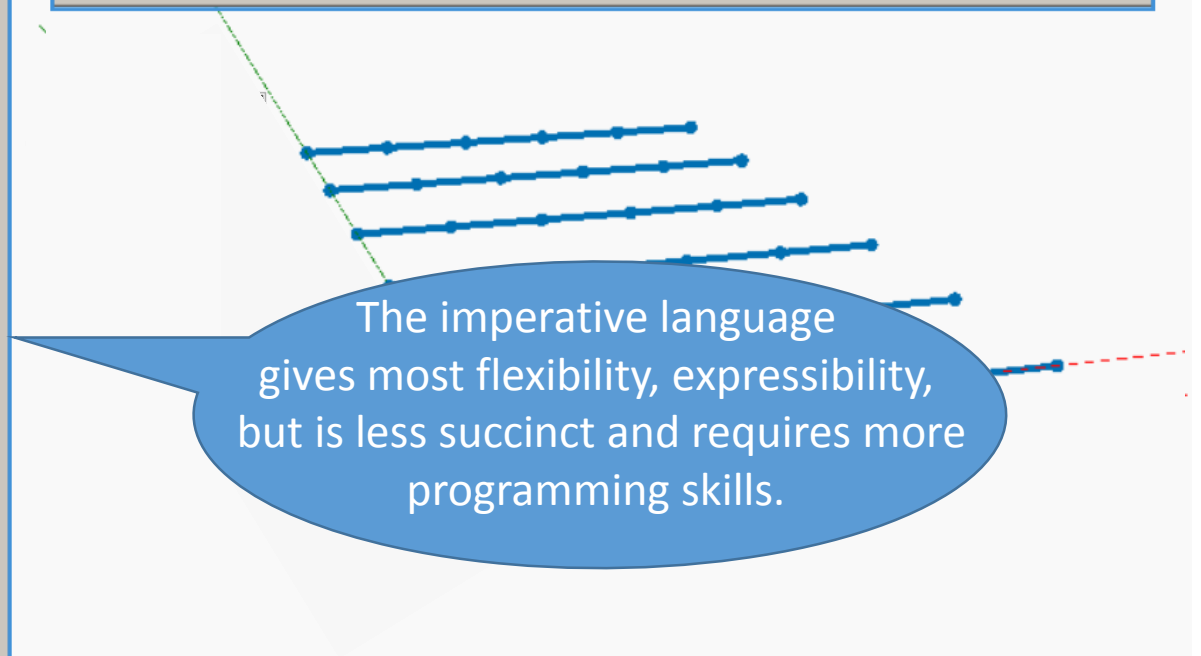
The visual data flow language is highly simplified as there are no explicit flow control statements [because flow control is provided by the dependency graph] New syntax is added to control replication [allowing the use of collections without explicit iteration]. The text based data flow language is succinct but has defined limitations

Code Block

```
pointArray  = {};
nurbsCurve1 = {};
[Imperative]
{
    xRange = (0..20..4);
    yRange = (0..20..4);

    xCount = Count(xRange);
    yCount = Count(yRange);

    for (i in 0..(xCount-1))
    {
        for (j in 0..(yCount-1))
        {
            pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
        }
    }

    for (i in 0..(xCount-1))
    {
        nurbsCurve1[i] = NurbsCurve.ByPoints(pointArray[i]);
    }
};
```

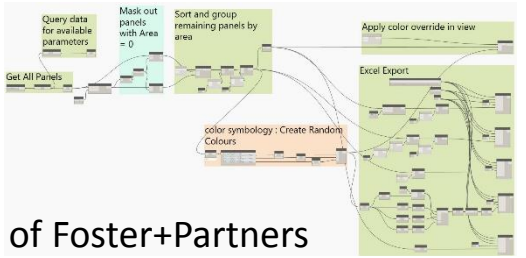
Code Block

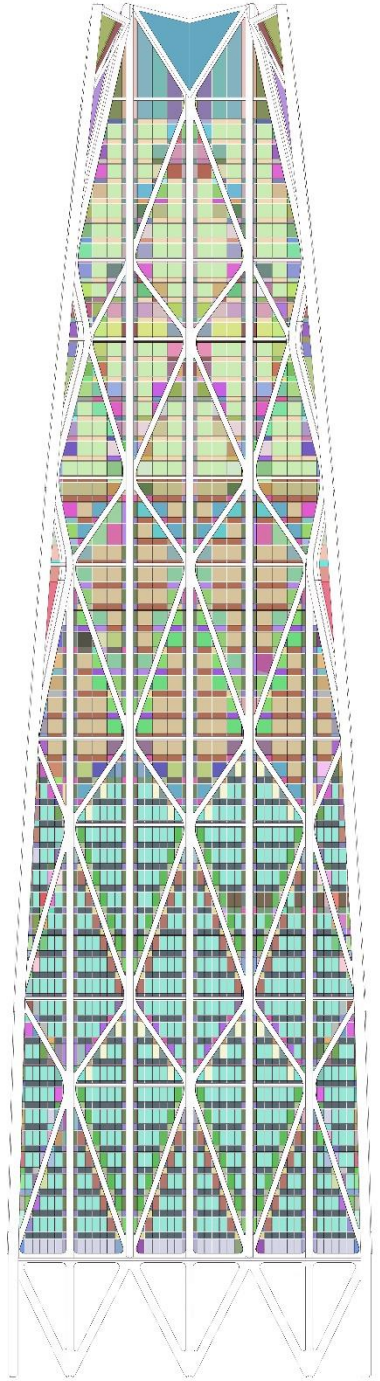
```
point1 = Point.ByCoordinates((0..20..4)<2>, (0..20..4)<1>, 0);
nurbsCurve1 = NurbsCurve.ByControlPoints(point1);
```



http://ww4.hdnux.com/photos/43/33/10/9286179/3/920x920.jpg

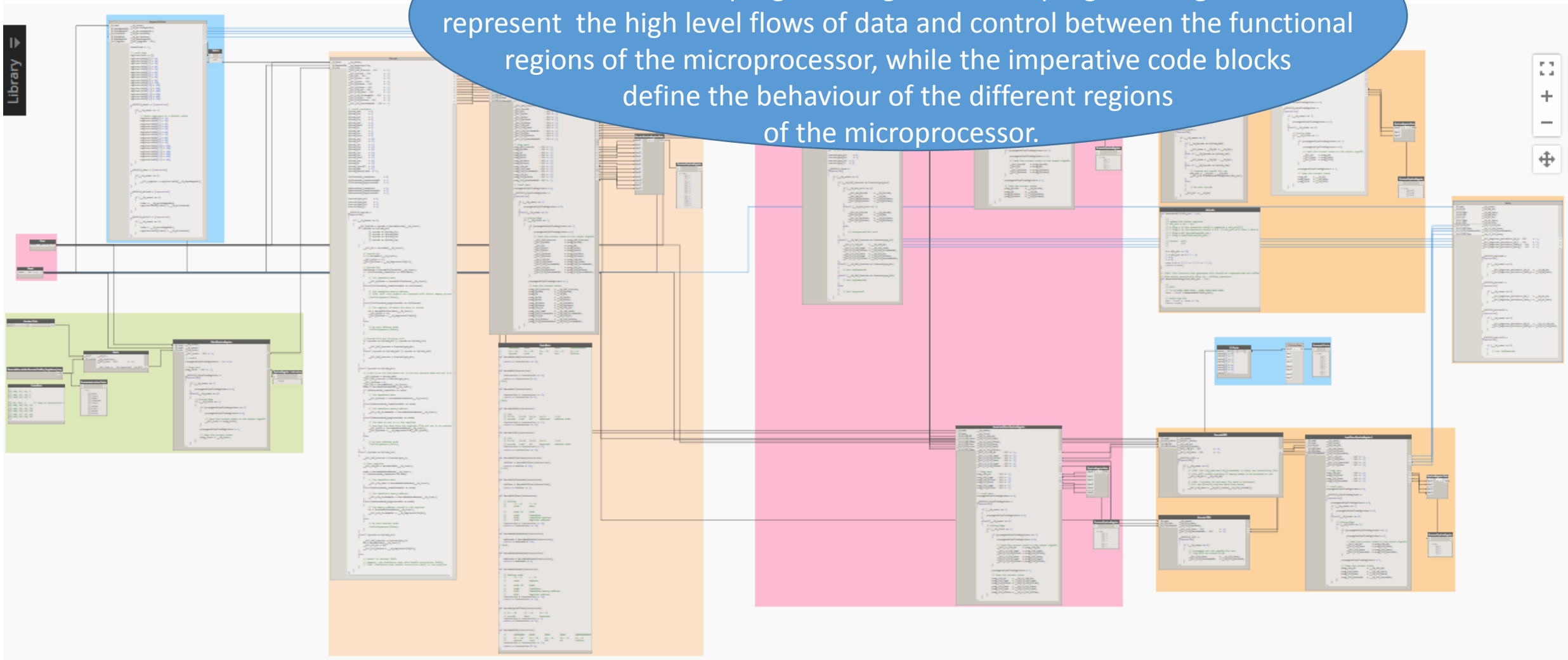
Building are composed of collections of components. This example demonstrates the value of being able to directly operate on collections.





OceanView Building, San Francisco. Images courtesy of Foster+Partners

This is a completely different type of example and demonstrates the value of being able to combine visual and text based programming. The visual programming is used to represent the high level flows of data and control between the functional regions of the microprocessor, while the imperative code blocks define the behaviour of the different regions of the microprocessor.



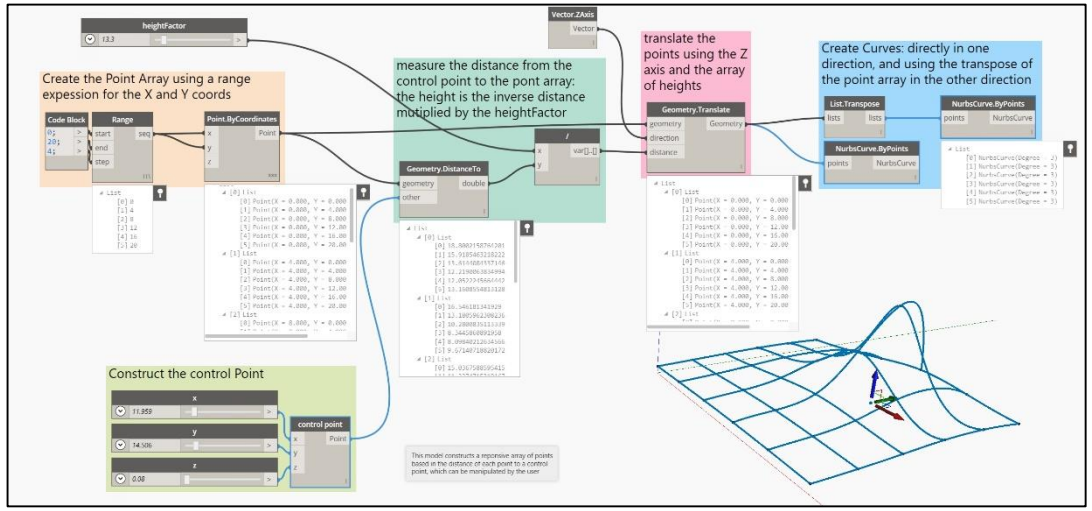
Using the hybrid visual and text based programing to model the MIPS microprocessor pipeline

overview of the design and implementation

data flow

imperative

visual



text based

Data Flow Code

```
heightFactor  pointArray = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<2>, 0
controlPoint  heights = heightFactor/pointArray.DistanceTo(controlPoint);
              translatedPointArray = pointArray.Translate(Vector.ZAxis()
              transposedPointArray = List.Transpose(translatedPointArr
              nurbsCurve1 = NurbsCurve.ByPoints(translatedPointArra
              nurbsCurve2 = NurbsCurve.ByPoints(transposedPointAr
```

In DesignScript, we have implemented visual and text based data flow programming and text based imperative programming, but not visual imperative programming…
This cell is empty.

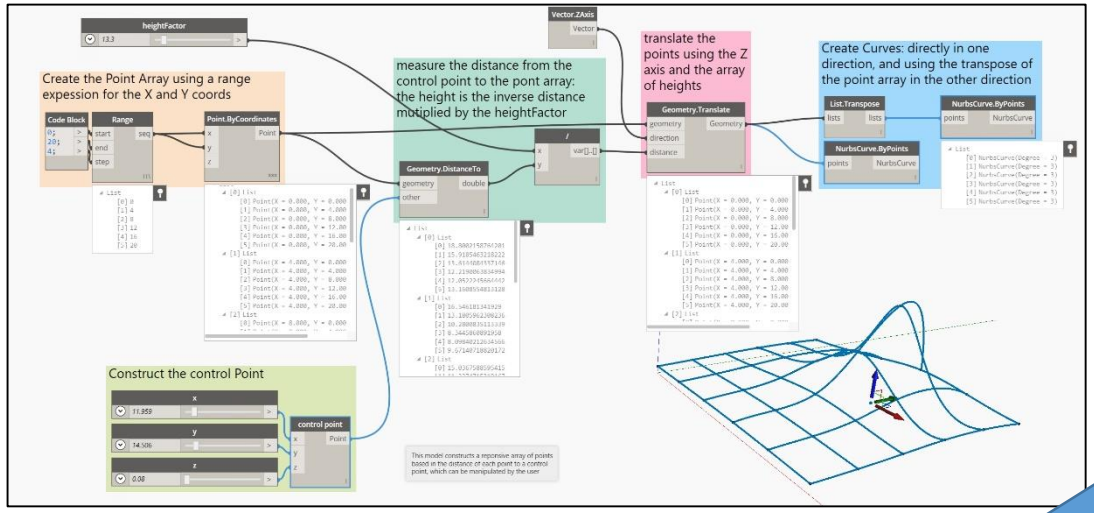
Imperative Code

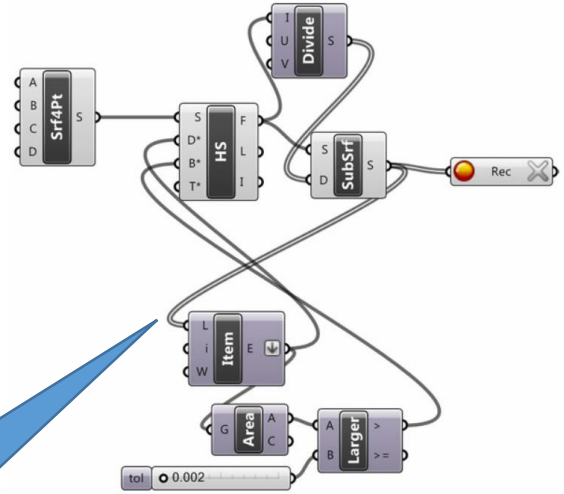
```
heightFactor heightFactor;
controlPoint controlPoint;
             pointArray  ={};
             translatedPointArray ={};
             transposedPointArray = {};
             nurbsCurve1 ={};
             nurbsCurve2 ={};
             [Imperative]
             {
                 xRange = (0..20..4);
                 yRange = (0..20..4);

                 xCount = Count(xRange);
                 yCount = Count(yRange);

                 for (i in 0..(xCount-1))
                 {
                     for (j in 0..(yCount-1))
                     {
                         pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
                     }
                 }

                 for (i in 0..(xCount-1))
                 {
                     for (j in 0..(yCount-1))
                     {
                         height = heightFactor/pointArray[i][j].DistanceTo(controlPoint);
                         translatedPointArray[i][j] = pointArray[i][j].Translate(Vector.ZAxis(
                     }
                     nurbsCurve1[i] = NurbsCurve.ByPoints(translatedPointArray[i]);
                 }

                 transposedPointArray = List.Transpose(translatedPointArray);
                 for (i in 0..(yCount-1))
                 {
                     nurbsCurve2[i] = NurbsCurve.ByPoints(transposedPointArray[i]);
                 }
             };
```

data flow

imperative

visual





text based

```
Data Flow Code
heightFactor  pointArray = Point.ByCoordinates((0..20..4)<1>, (0..20..
controlPoint  heights = heightFactor/pointArray.DistanceTo(controlP
              translatedPointArray = pointArray.Translate(Vector
              transposedPointArray = List.Transpose(translate
              nurbsCurve1 = NurbsCurve.ByPoints(translate
              nurbsCurve2 = NurbsCurve.ByPoints(transp
```

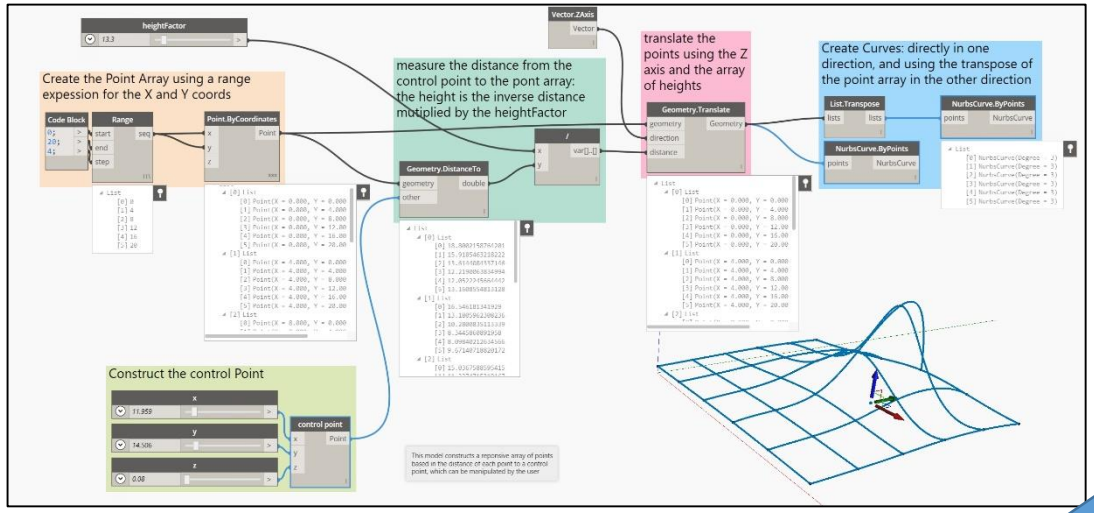
```
Imperative Code
heightFactor  heightFactor;
controlPoint  controlPoint;
              pointArray  ={};
              translatedPointArray ={};
              transposedPointArray = {};
              nurbsCurve1 ={};
              nurbsCurve2 ={};
              [Imperative]
              {
                  xRange = (0..20..4);
                  yRange = (0..20..4);

                  xCount = Count(xRange);
                  yCount = Count(yRange);

                  for (i in 0..(xCount-1))
                  {
                      for (j in 0..(yCount-1))
                      {
                          pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
                      }
                  }

                  for (i in 0..(xCount-1))
                  {
                      for (j in 0..(yCount-1))
                      {
                          height = heightFactor/pointArray[i][j].DistanceTo(controlPoint);
                          translatedPointArray[i][j] = pointArray[i][j].Translate(Vector.ZAxis(
                      }
                      nurbsCurve1[i] = NurbsCurve.ByPoints(translatedPointArray[i]);
                  }

                  transposedPointArray = List.Transpose(translatedPointArray);
                  for (i in 0..(yCount-1))
                  {
                      nurbsCurve2[i] = NurbsCurve.ByPoints(transposedPointArray[i]);
                  }
              };
```

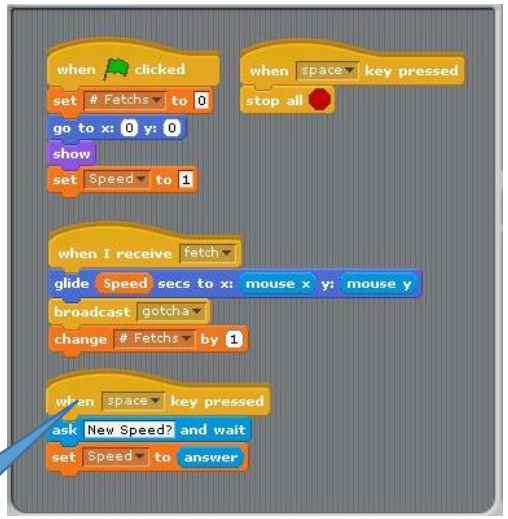
There have been attempts
to retrofit aspects of imperative programming into
visual data flow systems, but again we may be falling into the
trap of making visual programming too complex and unreadable.
The effort to explain how this work might be better spent
teaching a regular text based imperative
language

data flow

imperative

visual





text based

Data Flow Code

```
heightFactor  pointArray = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<
controlPoint  heights = heightFactor/pointArray.DistanceTo(controlPoint
              translatedPointArray = pointArray.Translate(Vector.ZA          eights);
              transposedPointArray = List.Transpose(translatedPo
              nurbsCurve1 = NurbsCurve.ByPoints(translatedPoi
              nurbsCurve2 = NurbsCurve.ByPoints(transposed
```

Other system use
the ‘jig-saw’ puzzle visual approach for imperative programming.

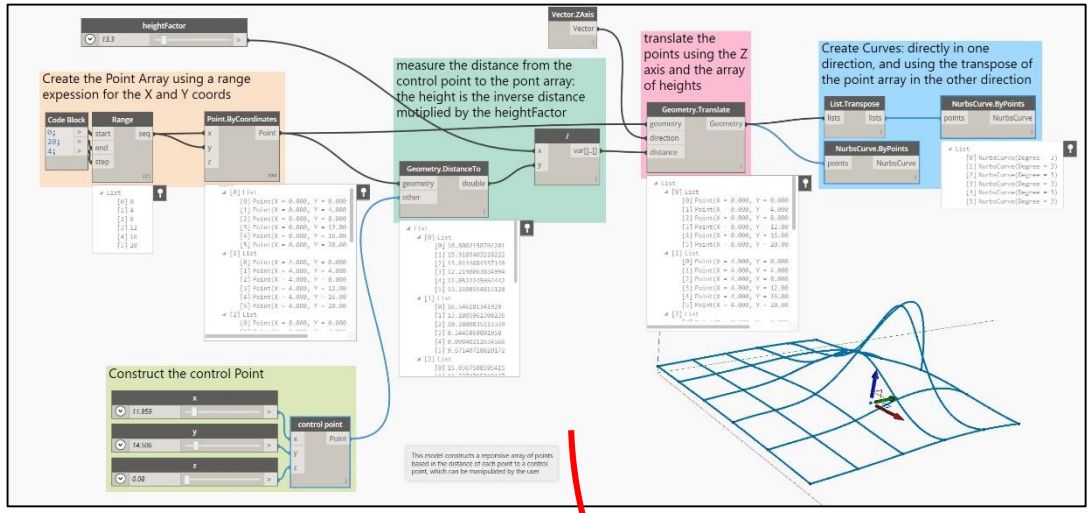
Imperative Code

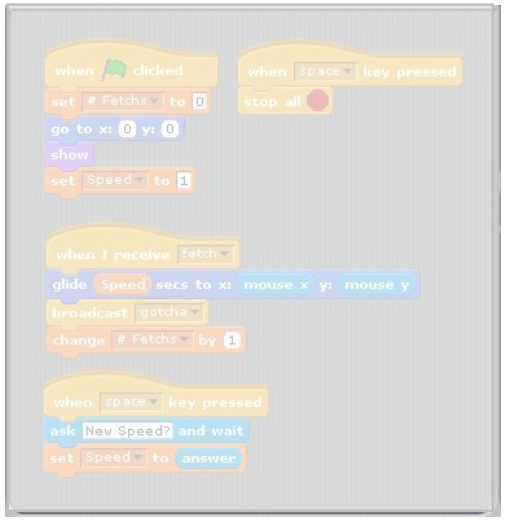
```
heightFactor heightFactor;
controlPoint controlPoint;
             pointArray  ={};
             translatedPointArray ={};
             transposedPointArray = {};
             nurbsCurve1 ={};
             nurbsCurve2 ={};
             [Imperative]
             {
                 xRange = (0..20..4);
                 yRange = (0..20..4);

                 xCount = Count(xRange);
                 yCount = Count(yRange);

                 for (i in 0..(xCount-1))
                 {
                     for (j in 0..(yCount-1))
                     {
                         pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
                     }
                 }

                 for (i in 0..(xCount-1))
                 {
                     for (j in 0..(yCount-1))
                     {
                         height = heightFactor/pointArray[i][j].DistanceTo(controlPoint);
                         translatedPointArray[i][j] = pointArray[i][j].Translate(Vector.ZAxis(
                     }
                     nurbsCurve1[i] = NurbsCurve.ByPoints(translatedPointArray[i]);
                 }

                 transposedPointArray = List.Transpose(translatedPointArray);
                 for (i in 0..(yCount-1))
                 {
                     nurbsCurve2[i] = NurbsCurve.ByPoints(transposedPointArray[i]);
                 }
             };
```

data flow

imperative

visual





text based

Data Flow Code

```
heightFactor  pointArray = Point.ByCoordinates((0..20..4)<1>, (0..20..4)<2>, 0);
controlPoint  heights = heightFactor/pointArray.DistanceTo(controlPoint);
              translatedPointArray = pointArray.Translate(Vector.ZAxis(), heights);
              transposedPointArray = List.Transpose(translatedPointArray);
              nurbsCurve1 = NurbsCurve.ByPoints(translatedPointArray );
              nurbsCurve2 = NurbsCurve.ByPoints(transposedPointArray);
```

Imperative Code

```
heightFactor heightFactor;
controlPoint controlPoint;
             pointArray  ={};
             translatedPointArray ={};
             transposedPointArray = {};
             nurbsCurve1 ={};
             nurbsCurve2 ={};
             [Imperative]
             {
                 xRange = (0..20..4);
                 yRange = (0..20..4);

                 xCount = Count(xRange);
                 yCount = Count(yRange);

                 for (i in 0..(xCount-1))
                 {
                     for (j in 0..(yCount-1))
                     {
                         pointArray[i][j] = Point.ByCoordinates(xRange[i], yRange[j], 0);
                     }
                 }

                 for (i in 0..(xCount-1))
                 {
                     for (j in 0..(yCount-1))
                     {
                         height = heightFactor/pointArray[i][j].DistanceTo(controlPoint);
                         translatedPointArray[i][j] = pointArray[i][j].Translate(Vector.ZAxis(
                     }
                     nurbsCurve1[i] = NurbsCurve.ByPoints(translatedPointArray[i]);
                 }

                 transposedPointArray = List.Transpose(translatedPointArray);
                 for (i in 0..(yCount-1))
                 {
                     nurbsCurve2[i] = NurbsCurve.ByPoints(transposedPointArray[i]);
                 }
             };
```

Our target users are not school students, but professionals who happen to be novice end-user programmers. Our sense, is that by the time the users have progressed from ‘node to code’ they will not want to go back to a restricted ‘jig-saw’ puzzle approach

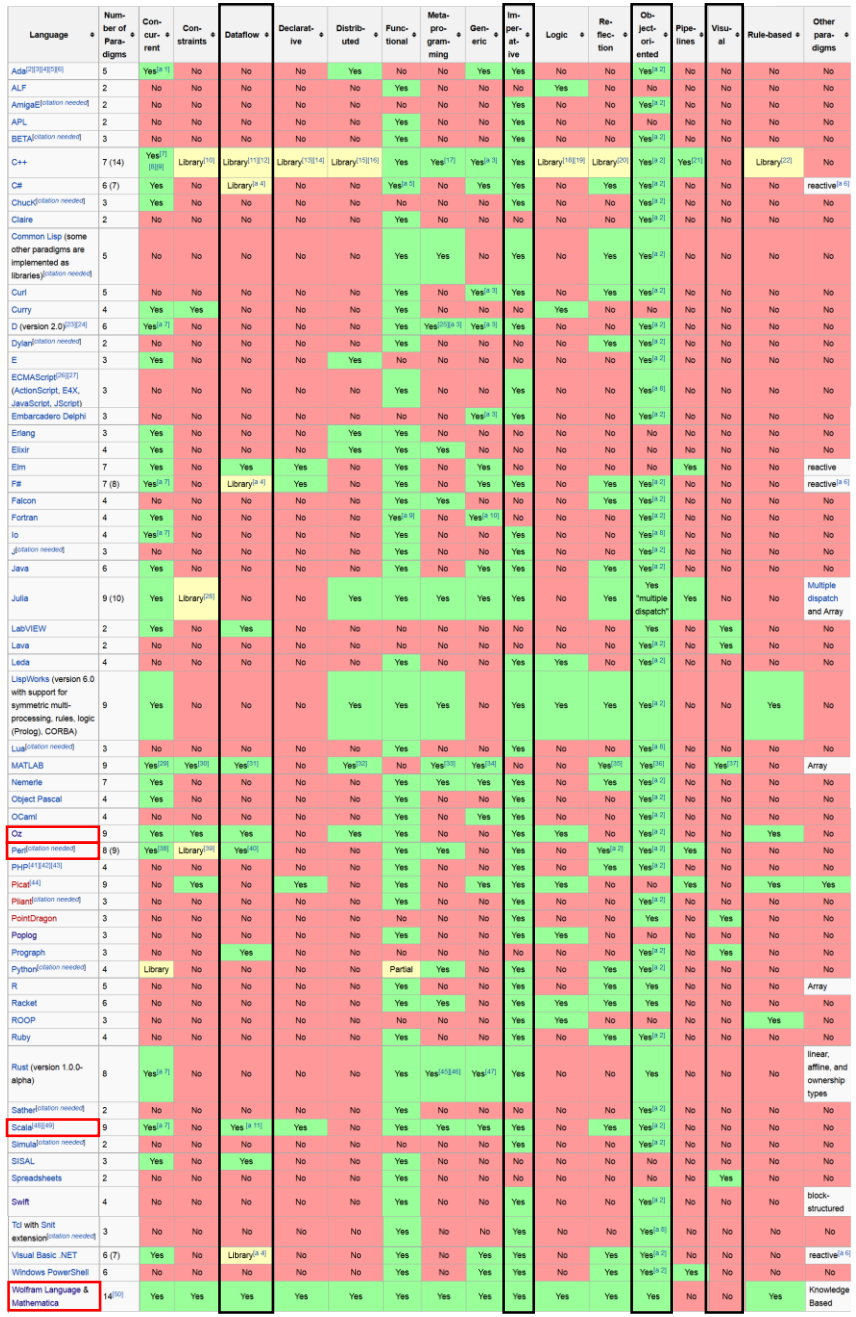
Language	Number of Paradigms	Concurrent	Constraints	Dataflow	Declarative	Distributed	Functional	Metaprogramming	Generic	Imperative	Logic	Reflection	Object oriented	Pipelines	Visual	Rule-based	Other paradigms
Ada[23][24][25][26]	5	Yes[a 1]	No	No	No	Yes	No	No	Yes	Yes	No	No	Yes[a 2]	No	No	No	No
ALF	2	No	No	No	No	No	Yes	No	No	No	Yes	No	No	No	No	No	No
AmigaE[citation needed]	2	No	No	No	No	No	No	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
APL	2	No	No	No	No	No	Yes	No	No	Yes	No	No	No	No	No	No	No
BETA[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
C++	7 (14)	Yes[7][8][9]	Library[10]	Library[11][12]	Library[13][14]	Library[15][16]	Yes	Yes[17]	Yes[a 3]	Yes	Library[18][19]	Library[20]	Yes[a 2]	Yes[21]	No	Library[22]	No
C#	6 (7)	Yes	No	Library[a 4]	No	No	Yes[a 5]	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	reactive[a 6]
Chuck[citation needed]	3	Yes	No	No	No	No	No	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
Claire	2	No	No	No	No	No	Yes	No	No	No	No	No	Yes[a 2]	No	No	No	No
Common Lisp (some other paradigms are implemented as libraries)[citation needed]	5	No	No	No	No	No	Yes	Yes	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
Curl	5	No	No	No	No	No	Yes	No	Yes[a 3]	Yes	No	Yes	Yes[a 2]	No	No	No	No
Curry	4	Yes	Yes	No	No	No	Yes	No	No	No	Yes	No	No	No	No	No	No
D (version 2.0)[23][24]	6	Yes[a 7]	No	No	No	No	Yes	Yes[25][a 8]	Yes[a 3]	Yes	No	No	Yes[a 2]	No	No	No	No
Dylan[citation needed]	2	No	No	No	No	No	Yes	No	No	No	No	Yes	Yes[a 2]	No	No	No	No
E	3	Yes	No	No	No	Yes	No	No	No	No	No	No	Yes[a 2]	No	No	No	No
ECMAScript[26][27] (ActionScript, E4X, JavaScript, JScript)	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 9]	No	No	No	No
Embarcadero Delphi	3	No	No	No	No	No	No	No	Yes[a 3]	Yes	No	No	Yes[a 2]	No	No	No	No
Erlang	3	Yes	No	No	No	Yes	Yes	No	No	No	No	No	No	No	No	No	No
Elixir	4	Yes	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No
Elm	7	Yes	No	Yes	Yes	No	Yes	No	Yes	No	No	No	No	Yes	No	No	reactive
F#	7 (8)	Yes[a 7]	No	Library[a 4]	Yes	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	reactive[a 6]
Falcon	4	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes[a 2]	No	No	No	No
Fortran	4	Yes	No	No	No	No	Yes[a 9]	No	Yes[a 10]	No	No	No	Yes[a 2]	No	No	No	No
Io	4	Yes[a 7]	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 8]	No	No	No	No
J[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
Java	6	Yes	No	No	No	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	No
Julia	9 (10)	Yes	Library[28]	No	No	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes "multiple dispatch"	Yes	No	No	Multiple dispatch and Array
LabVIEW	2	Yes	No	Yes	No	No	No	No	No	No	No	No	Yes	No	Yes	No	No
Lava	2	No	No	No	No	No	No	No	No	No	No	No	Yes[a 2]	No	Yes	No	No
Leda	4	No	No	No	No	No	Yes	No	No	Yes	Yes	No	Yes[a 2]	No	No	No	No
LispWorks (version 6.0 with support for symmetric multi-processing, rules, logic (Prolog), CORBA)	9	Yes	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes[a 2]	No	No	Yes	No
Lua[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 8]	No	No	No	No
MATLAB	9	Yes[29]	Yes[30]	Yes[31]	No	Yes[32]	No	Yes[33]	Yes[34]	No	No	Yes[35]	Yes[36]	No	Yes[37]	No	Array
Nemerle	7	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	No
Object Pascal	4	Yes	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
OCaml	4	No	No	No	No	No	Yes	No	Yes	Yes	No	No	Yes[a 2]	No	No	No	No
Oz	9	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	No	Yes[a 2]	No	No	Yes	No
Perl[citation needed]	8 (9)	Yes[38]	Library[39]	Yes[40]	No	No	Yes	Yes	No	Yes	No	Yes[a 2]	Yes[a 2]	Yes	No	No	No
PHP[41][42][43]	4	No	No	No	No	No	Yes	No	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
Picaf[44]	9	No	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes
Pliant[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
PointDragon	3	No	No	No	No	No	No	No	No	Yes	No	No	Yes	No	Yes	No	No
Poplog	3	No	No	No	No	No	Yes	No	No	Yes	Yes	No	No	No	No	No	No
Prograph	3	No	No	Yes	No	No	No	No	No	No	No	No	Yes[a 2]	No	Yes	No	No
Python[citation needed]	4	Library	No	No	No	No	Partial	Yes	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
R	5	No	No	No	No	No	Yes	No	No	Yes	No	Yes	Yes	No	No	No	Array
Racket	6	No	No	No	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No	No
ROOP	3	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	Yes	No
Ruby	4	No	No	No	No	No	Yes	No	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
Rust (version 1.0.0-alpha)	8	Yes[a 7]	No	No	No	No	Yes	Yes[45][46]	Yes[47]	Yes	No	No	Yes	No	No	No	linear, affine, and ownership types
Sather[citation needed]	2	No	No	No	No	No	Yes	No	No	No	No	No	Yes[a 2]	No	No	No	No
Scala[48][49]	9	Yes[a 7]	No	Yes [a 11]	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	No
Simula[citation needed]	2	No	No	No	No	No	No	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
SISAL	3	Yes	No	Yes	No	No	Yes	No	No	No	No	No	No	No	No	No	No
Spreadsheets	2	No	No	No	No	No	Yes	No	No	No	No	No	No	No	Yes	No	No
Swift	4	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	block-structured
Tcl with Snit extension[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 8]	No	No	No	No
Visual Basic .NET	6 (7)	Yes	No	Library[a 4]	No	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	reactive[a 6]
Windows PowerShell	6	No	No	No	No	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	Yes	No	No	No
Wolfram Language & Mathematica	14[50]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Knowledge Based

We briefly compared DesignScript with other multi-paradigm languages.

Of all the multi-paradigm languages listed on https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

only Oz, Higher Order Perl and Scala [via Akka] support data flow, imperative and object oriented programming

and none support data flow, imperative object oriented and visual programming.



Language	Number of Paradigms	Concurrent	Constraints	Dataflow	Declarative	Distributed	Functional	Metaprogramming	Generic	Imperative	Logic	Reflection	Object oriented	Pipelines	Visual	Rule-based	Other paradigms
Ada[23][24][25][26]	5	Yes[a 1]	No	No	No	Yes	No	No	Yes	Yes	No	No	Yes[a 2]	No	No	No	No
ALF	2	No	No	No	No	No	Yes	No	No	No	Yes	No	No	No	No	No	No
AmigaE[citation needed]	2	No	No	No	No	No	No	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
APL	2	No	No	No	No	No	Yes	No	No	Yes	No	No	No	No	No	No	No
BETA[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
C++	7 (14)	Yes[7] [8][9]	Library[10]	Library[11][12]	Library[13][14]	Library[15][16]	Yes	Yes[17]	Yes[a 3]	Yes	Library[18][19]	Library[20]	Yes[a 2]	Yes[21]	No	Library[22]	No
C#	6 (7)	Yes	No	Library[a 4]	No	No	Yes[a 5]	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	reactive[a 6]
Chuck[citation needed]	3	Yes	No	No	No	No	No	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
Claire	2	No	No	No	No	No	Yes	No	No	No	No	No	Yes[a 2]	No	No	No	No
Common Lisp (some other paradigms are implemented as libraries)[citation needed]	5	No	No	No	No	No	Yes	Yes	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
Curl	5	No	No	No	No	No	Yes	No	Yes[a 3]	Yes	No	Yes	Yes[a 2]	No	No	No	No
Curry	4	Yes	Yes	No	No	No	Yes	No	No	No	Yes	No	No	No	No	No	No
D (version 2.0)[23][24]	6	Yes[a 7]	No	No	No	No	Yes	Yes[25][a 8]	Yes[a 3]	Yes	No	No	Yes[a 2]	No	No	No	No
Dylan[citation needed]	2	No	No	No	No	No	Yes	No	No	No	No	Yes	Yes[a 2]	No	No	No	No
E	3	Yes	No	No	No	Yes	No	No	No	No	No	No	Yes[a 2]	No	No	No	No
ECMAScript[26][27] (ActionScript, E4X, JavaScript, JScript)	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 9]	No	No	No	No
Embarcadero Delphi	3	No	No	No	No	No	No	No	Yes[a 3]	Yes	No	No	Yes[a 2]	No	No	No	No
Erlang	3	Yes	No	No	No	Yes	Yes	No	No	No	No	No	No	No	No	No	No
Elixir	4	Yes	No	No	No	Yes	Yes	Yes	No	No	No	No	No	No	No	No	No
Elm	7	Yes	No	Yes	Yes	No	Yes	No	Yes	No	No	No	No	Yes	No	No	reactive
F#	7 (8)	Yes[a 7]	No	Library[a 4]	Yes	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	reactive[a 6]
Falcon	4	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes[a 2]	No	No	No	No
Fortran	4	Yes	No	No	No	No	Yes[a 9]	No	Yes[a 10]	No	No	No	Yes[a 2]	No	No	No	No
Io	4	Yes[a 7]	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 8]	No	No	No	No
J[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
Java	6	Yes	No	No	No	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	No
Julia	9 (10)	Yes	Library[28]	No	No	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes "multiple dispatch"	Yes	No	No	Multiple dispatch and Array
LabVIEW	2	Yes	No	Yes	No	No	No	No	No	No	No	No	Yes	No	Yes	No	No
Lava	2	No	No	No	No	No	No	No	No	No	No	No	Yes[a 2]	No	Yes	No	No
Leda	4	No	No	No	No	No	Yes	No	No	Yes	Yes	No	Yes[a 2]	No	No	No	No
LispWorks (version 6.0 with support for symmetric multi-processing, rules, logic (Prolog), CORBA)	9	Yes	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes[a 2]	No	No	Yes	No
Lua[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 8]	No	No	No	No
MATLAB	9	Yes[29]	Yes[30]	Yes[31]	No	Yes[32]	No	Yes[33]	Yes[34]	No	No	Yes[35]	Yes[36]	No	Yes[37]	No	Array
Nemerle	7	Yes	No	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	No
Object Pascal	4	Yes	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
OCaml	4	No	No	No	No	No	Yes	No	Yes	Yes	No	No	Yes[a 2]	No	No	No	No
Oz	9	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	No	Yes[a 2]	No	No	Yes	No
Perl[citation needed]	8 (9)	Yes[38]	Library[39]	Yes[40]	No	No	Yes	Yes	No	Yes	No	No	Yes[a 2]	Yes[a 2]	Yes	No	No
PHP[41][42][43]	4	No	No	No	No	No	Yes	No	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
Picat[44]	9	No	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No	No	Yes	No	Yes	Yes
Pliant[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
PointDragon	3	No	No	No	No	No	No	No	No	Yes	No	No	Yes	No	Yes	No	No
Poplog	3	No	No	No	No	No	Yes	No	No	Yes	Yes	No	No	No	No	No	No
Prograph	3	No	No	Yes	No	No	No	No	No	No	No	No	Yes[a 2]	No	Yes	No	No
Python[citation needed]	4	Library	No	No	No	No	Partial	Yes	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
R	5	No	No	No	No	No	Yes	No	No	Yes	No	Yes	Yes	No	No	No	Array
Racket	6	No	No	No	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No	No	No
ROOP	3	No	No	No	No	No	No	No	No	Yes	Yes	No	No	No	No	Yes	No
Ruby	4	No	No	No	No	No	Yes	No	No	Yes	No	Yes	Yes[a 2]	No	No	No	No
Rust (version 1.0.0-alpha)	8	Yes[a 7]	No	No	No	No	Yes	Yes[45][46]	Yes[47]	Yes	No	No	Yes	No	No	No	linear, affine, and ownership types
Sather[citation needed]	2	No	No	No	No	No	Yes	No	No	No	No	No	Yes[a 2]	No	No	No	No
Scala[48][49]	9	Yes[a 7]	No	Yes [a 11]	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	No
Simula[citation needed]	2	No	No	No	No	No	No	No	No	Yes	No	No	Yes[a 2]	No	No	No	No
SISAL	3	Yes	No	Yes	No	No	Yes	No	No	No	No	No	No	No	No	No	No
Spreadsheets	2	No	No	No	No	No	Yes	No	No	No	No	No	No	No	Yes	No	No
Swift	4	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 2]	No	No	No	block-structured
Tcl with Snit extension[citation needed]	3	No	No	No	No	No	Yes	No	No	Yes	No	No	Yes[a 8]	No	No	No	No
Visual Basic .NET	6 (7)	Yes	No	Library[a 4]	No	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	No	No	No	reactive[a 6]
Windows PowerShell	6	No	No	No	No	No	Yes	No	Yes	Yes	No	Yes	Yes[a 2]	Yes	No	No	No
Wolfram Language & Mathematica	14[50]	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Knowledge Based

Of all the multi-paradigm languages listed on
https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

only Oz, Higher Order Perl and Scala [via Akka] support data flow, imperative and object oriented programming

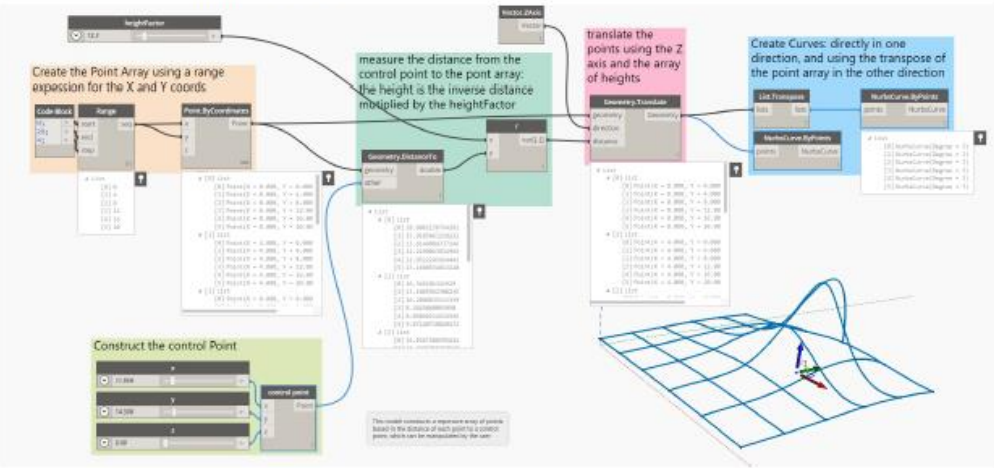
and none support data flow, imperative object oriented and visual programming.

There was not time to consider other aspects, but we note this in passing, for example…..

There are important issues concerning the evaluation of usability of these systems. Please see:

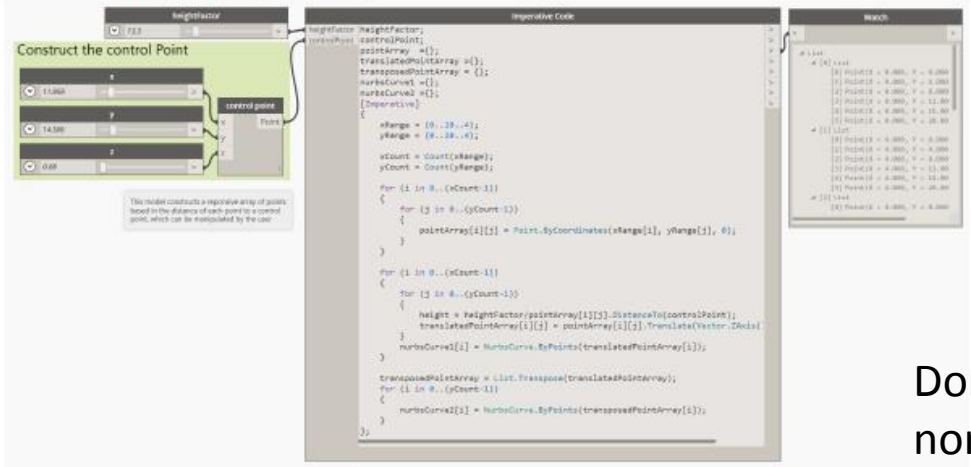
Robert Aish and Sean Hanna (2017) “Comparative evaluation of parametric design systems for teaching design computation”, paper submitted to the forthcoming coming special issue on Parametric Design System, Design Studies.

Visual Data Flow programing

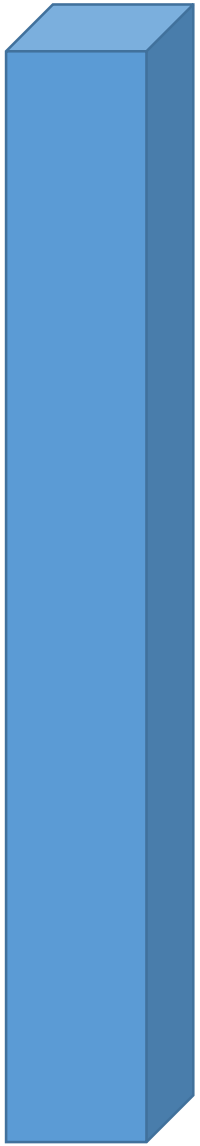


Text based Data Flow programing

Text based Imperative programing

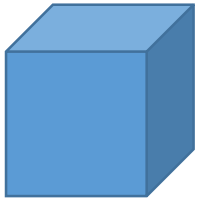


More generally, what is the take-up within professional architectural practices of the domain specific programming tools? Of the 'high-end' practices, we have reports that while most architects are still using standard design and modelling applications [which don't require any programming expertise], a small but increasing number are using visual data flow programming, supported by smaller proportion using scripting and general purpose programming tools.....



Domain Specific, non-computational

15%



Visual data flow programming

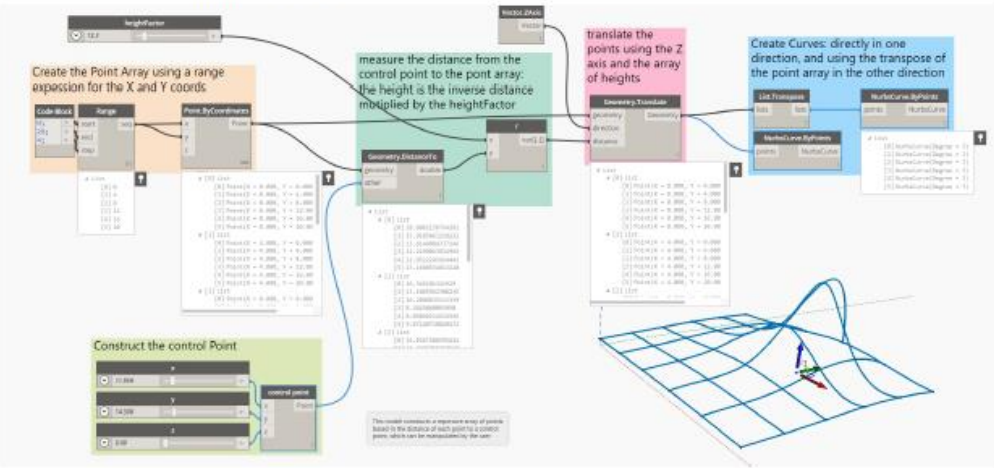
5%

Scripting

1.5%

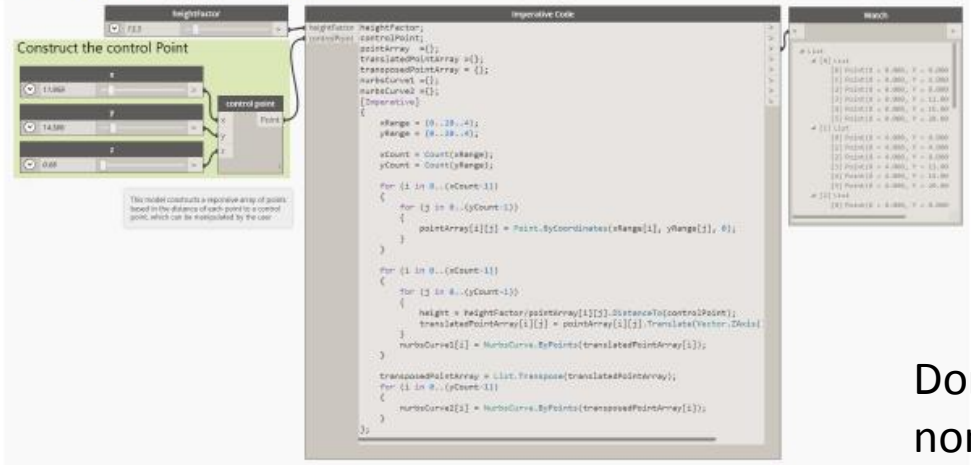
Programming with general purpose languages

Visual Data Flow programing

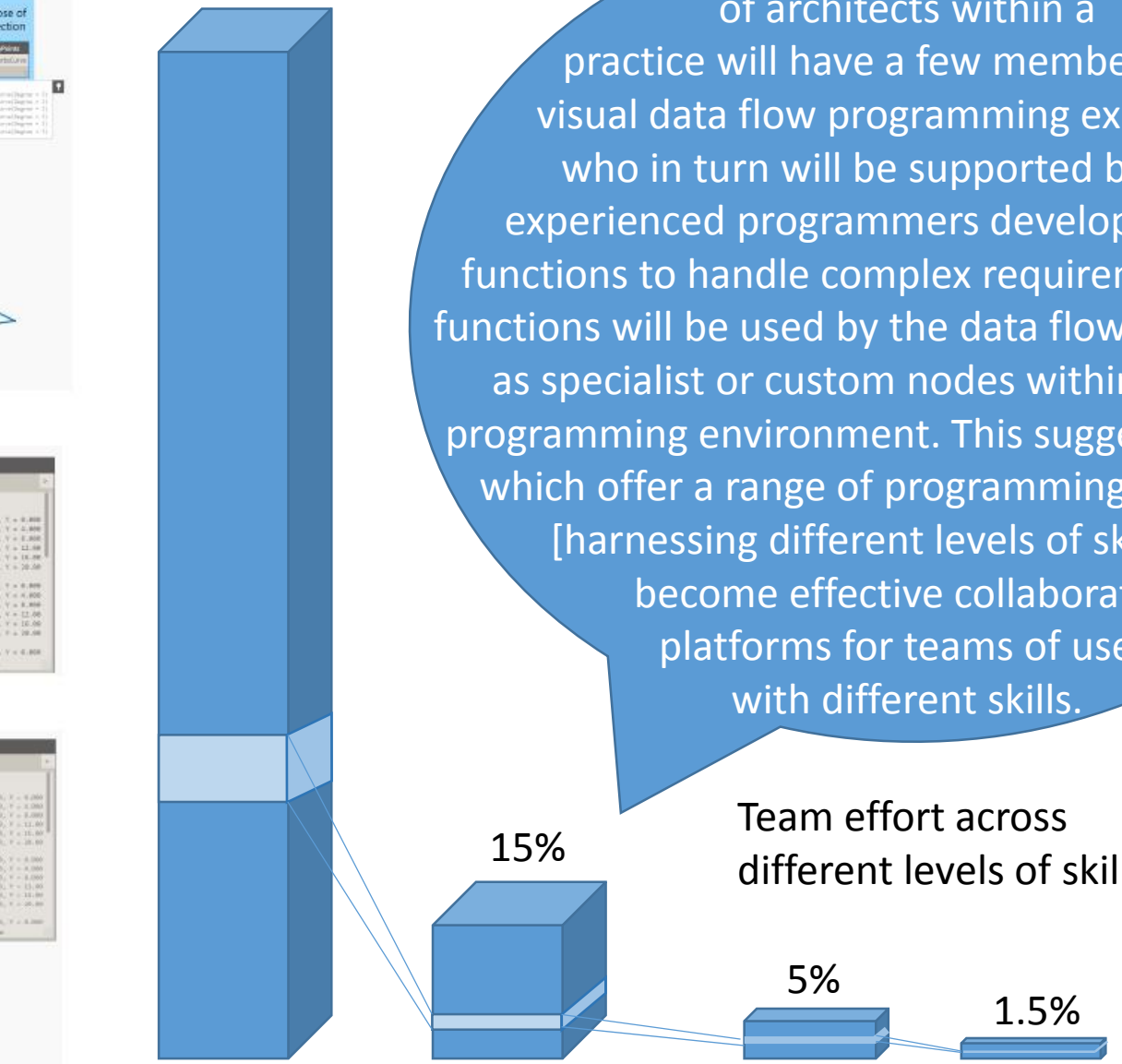


Text based Data Flow programing

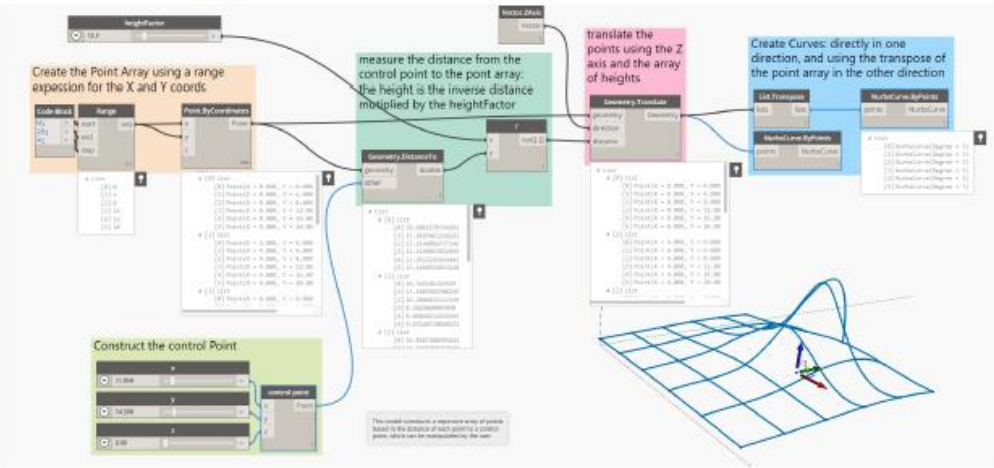
Text based Imperative programing



Often a team of architects within a practice will have a few members with visual data flow programming experience who in turn will be supported by more experienced programmers developing special functions to handle complex requirements. These functions will be used by the data flow programmers as specialist or custom nodes within the visual programming environment. This suggests that tools which offer a range of programming techniques [harnessing different levels of skills] can become effective collaboration platforms for teams of users with different skills.

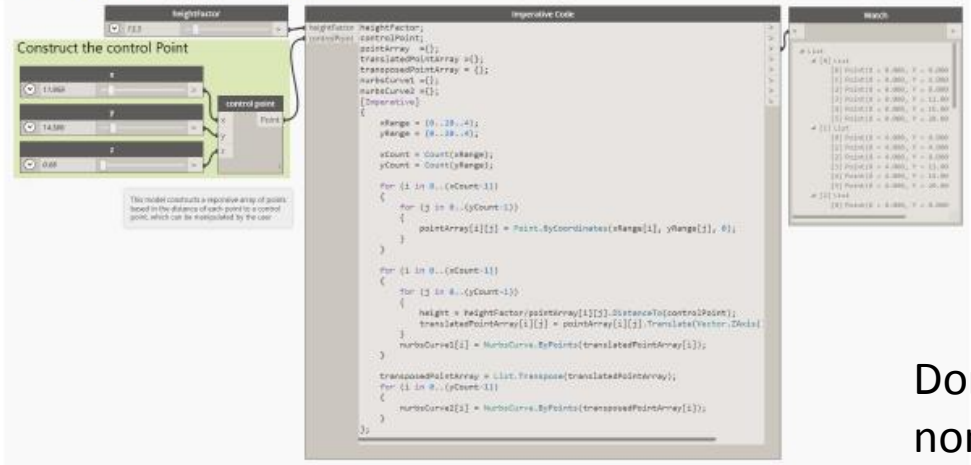


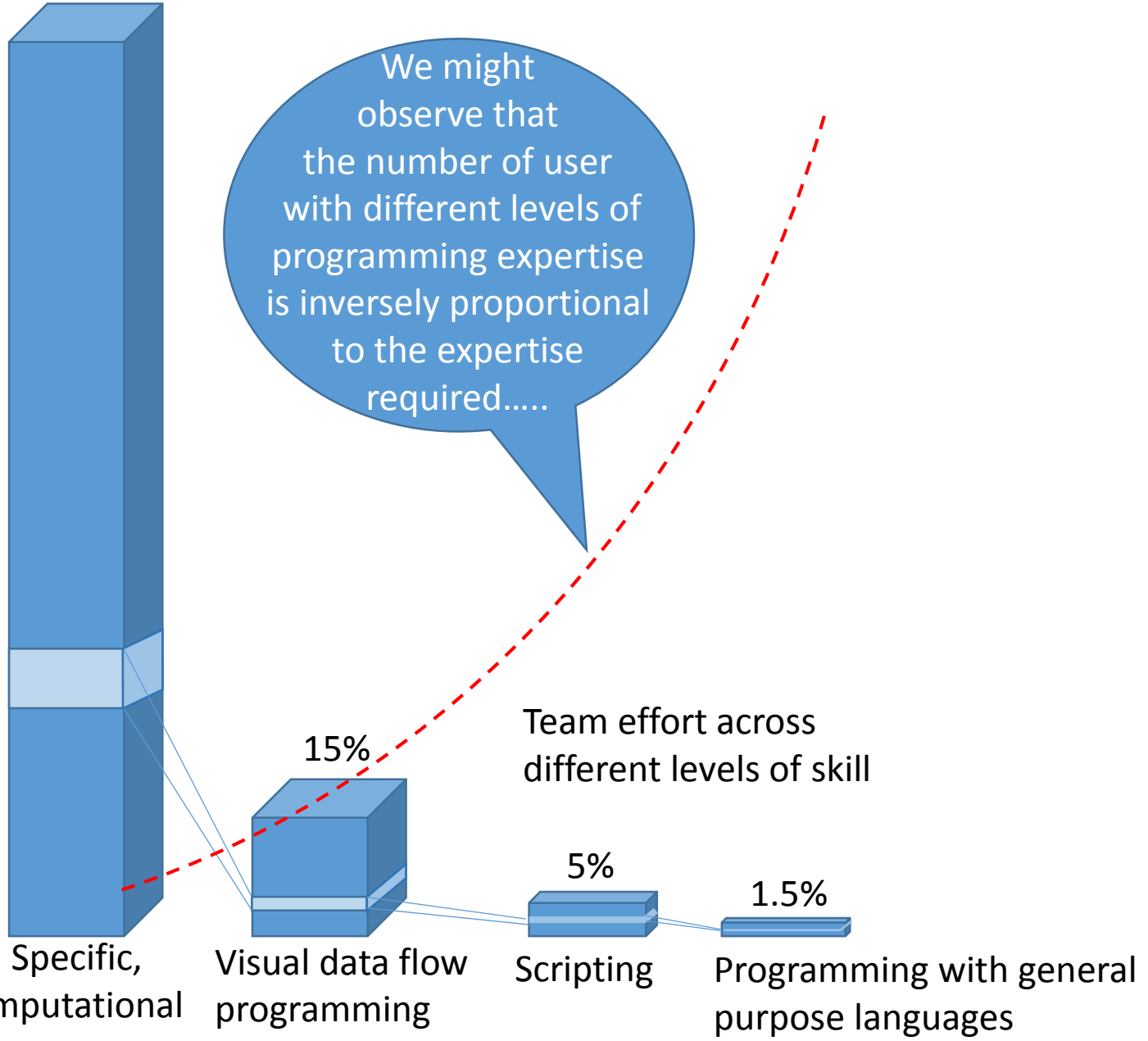
Visual Data Flow programing



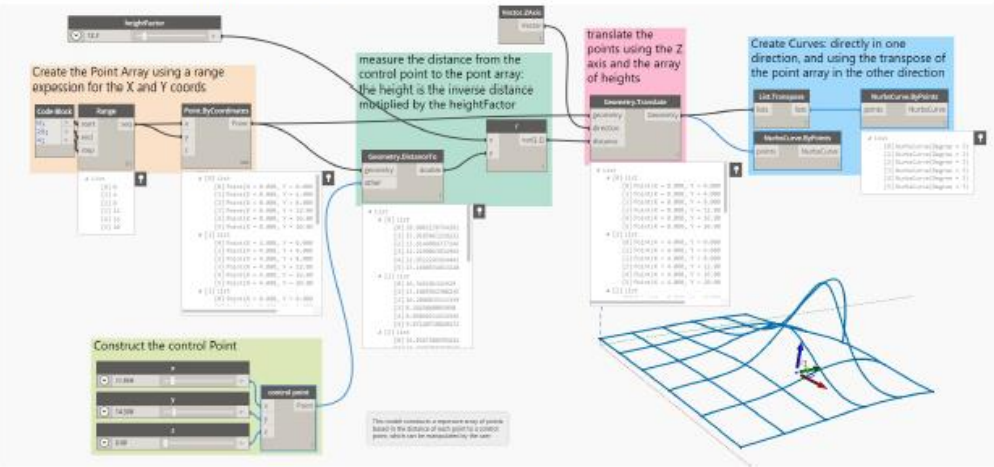
Text based Data Flow programing

Text based Imperative programing



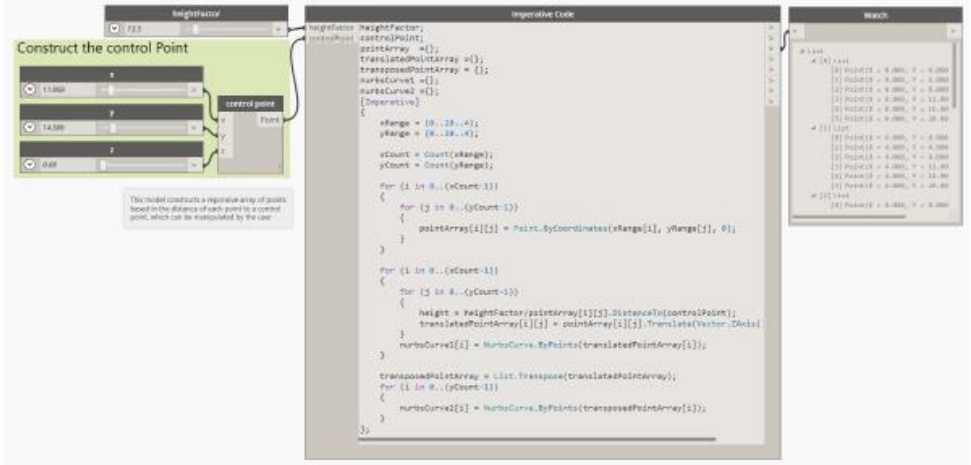


Visual Data Flow programing



Text based Data Flow programing

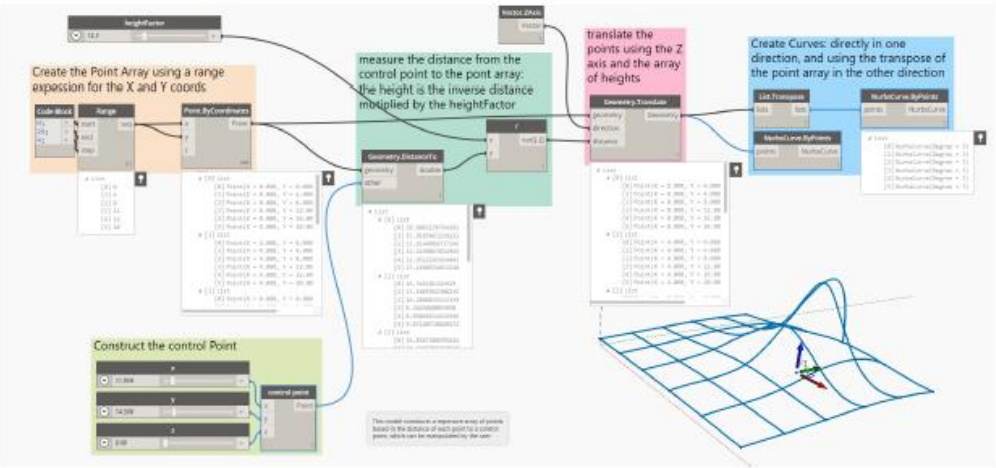
Text based Imperative programing



We conclude with some ‘take home’ messages…..

In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual Data Flow programing



Text based Data Flow programing

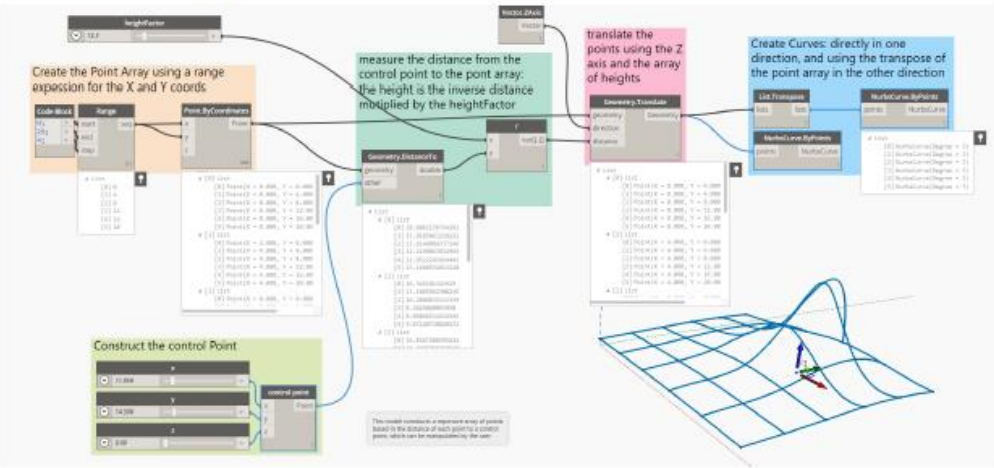
Text based Imperative programing

We conclude with some ‘take home’ messages…..

In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

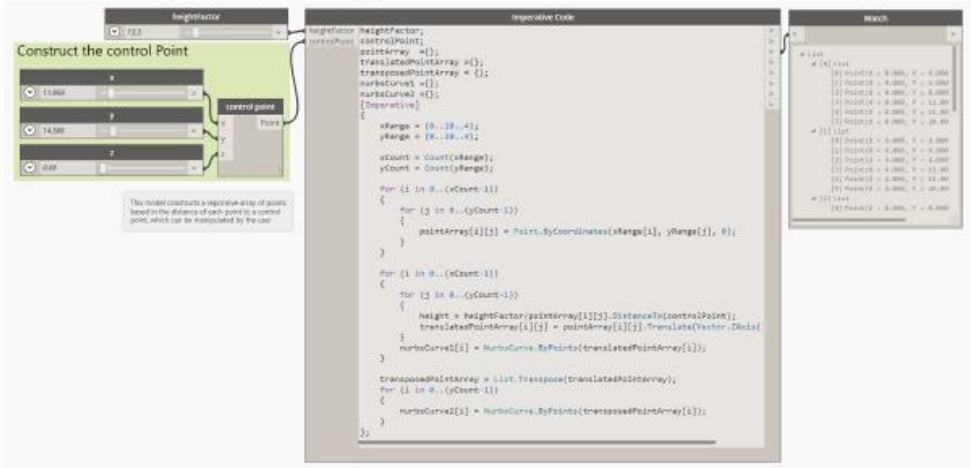
Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

Visual Data Flow programing



Text based Data Flow programing

Text based Imperative programing



In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

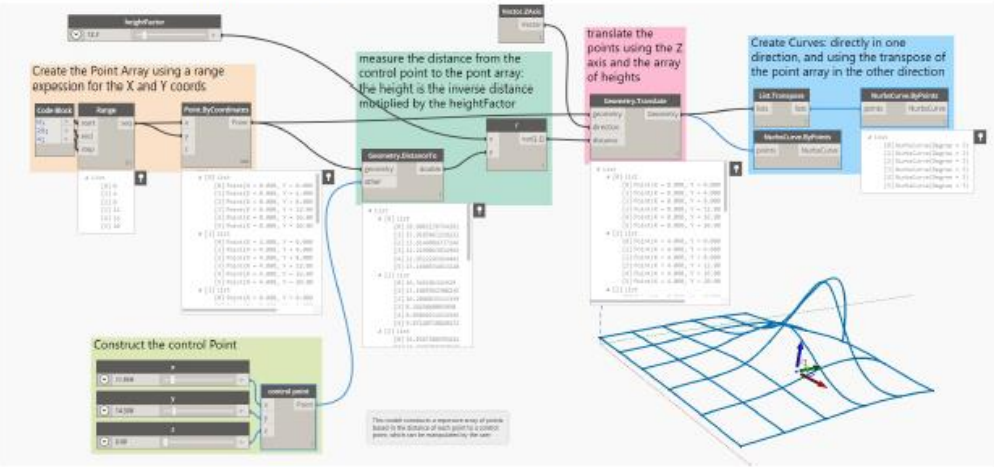
Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis in ‘Epigrams on Programming’.

This suggests a positive motivation for an end-user to learn programming

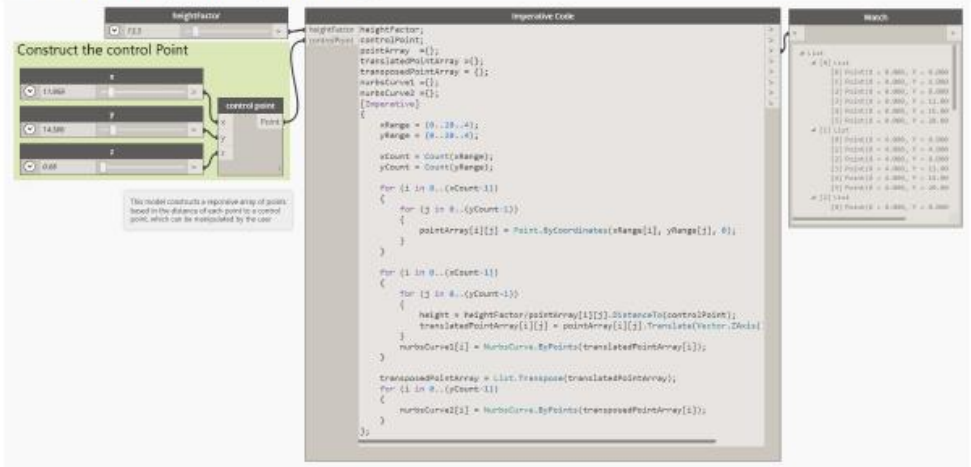
DesignScript @ Domain Specific Modelling 2016

Visual Data Flow programing



Text based Data Flow programing

Text based Imperative programing



In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

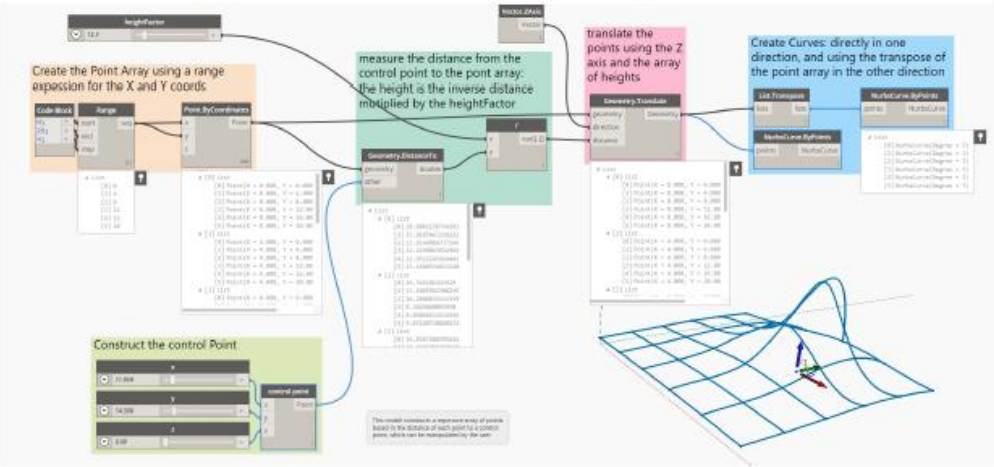
“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis 4 in ‘Epigrams on Programming’.

“The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.” Green and Blackwell.

One the other hand there are challenges

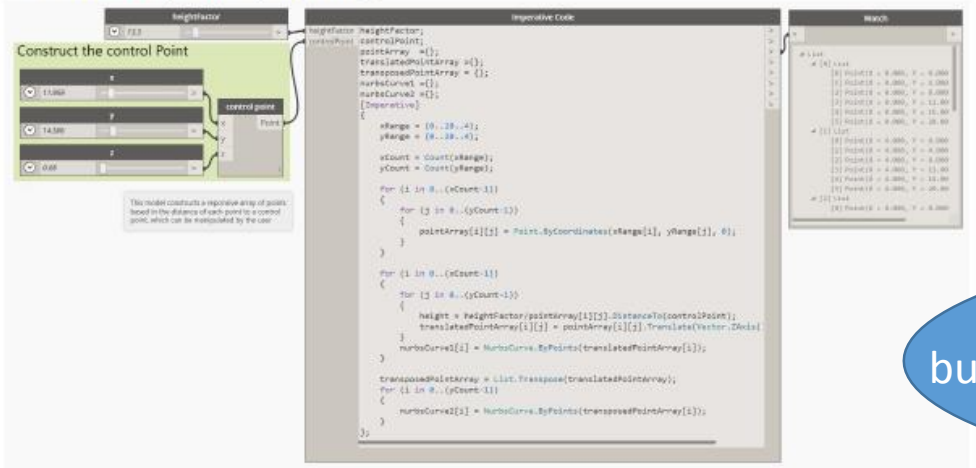
DesignScript @ Domain Specific Modelling 2016

Visual Data Flow programing



Text based Data Flow programing

Text based Imperative programing



In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis 4 in ‘Epigrams on Programming’.

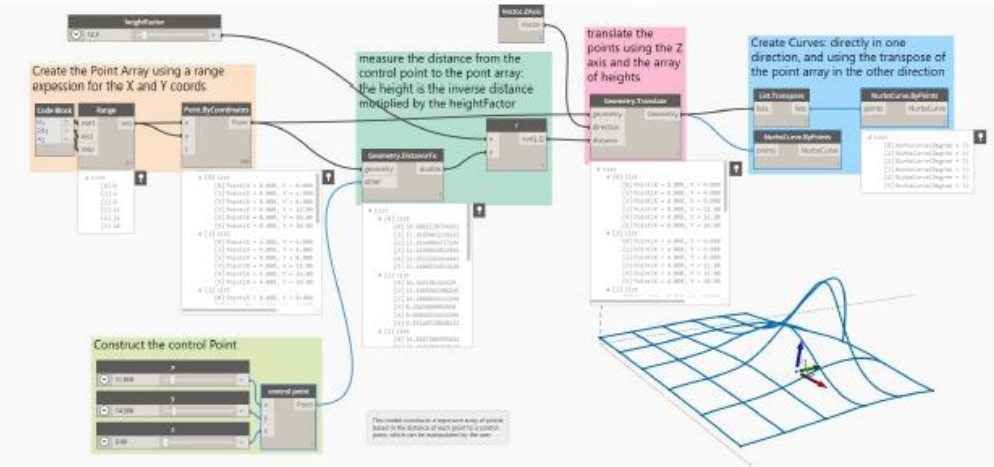
“The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.” Green and Blackwell.

A domain specific application should not to avoid abstractions, but avoid abstractions becoming a barrier. The same abstraction may sometimes be a barrier; at other times the key idea that ‘changes the way you think’.

A successful domain specific language has to weave this incredibly difficult path between supporting abstractions but not forcing their use… We have to wait for the users to recognise that there might be a ‘better way’ and then have that ‘better way’ waiting in the wings

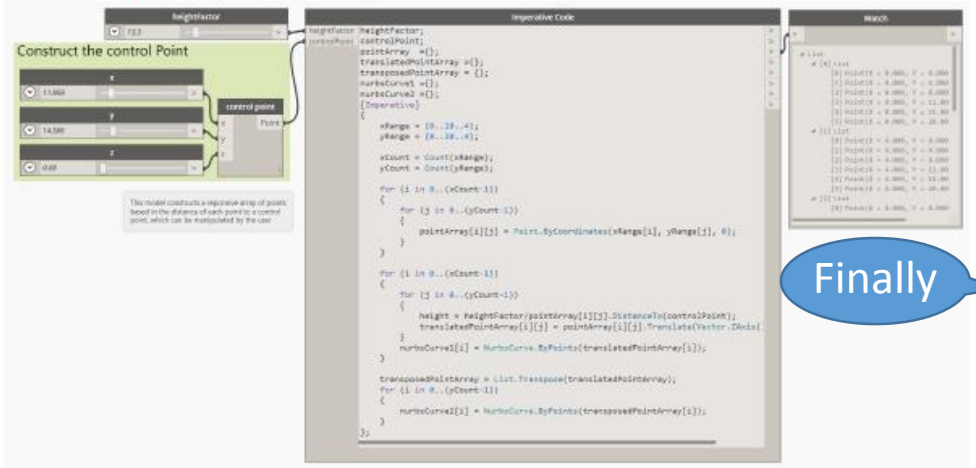
DesignScript @ Domain Specific Modelling 2016

Visual Data Flow programing



Text based Data Flow programing

Text based Imperative programing



In this application domain architects are expected to be exploratory, but also to manage the complexity which results from this exploration.

Visual data flow programming attracts an initial use with small exploratory designs, but does not scale to complex real world projects. To combine exploration and complexity users have to be helped to progress beyond visual programming, but there are challenges:

“A programming language that doesn’t change the way you think is not worth learning.” Alan Perlis 4 in ‘Epigrams on Programming’.

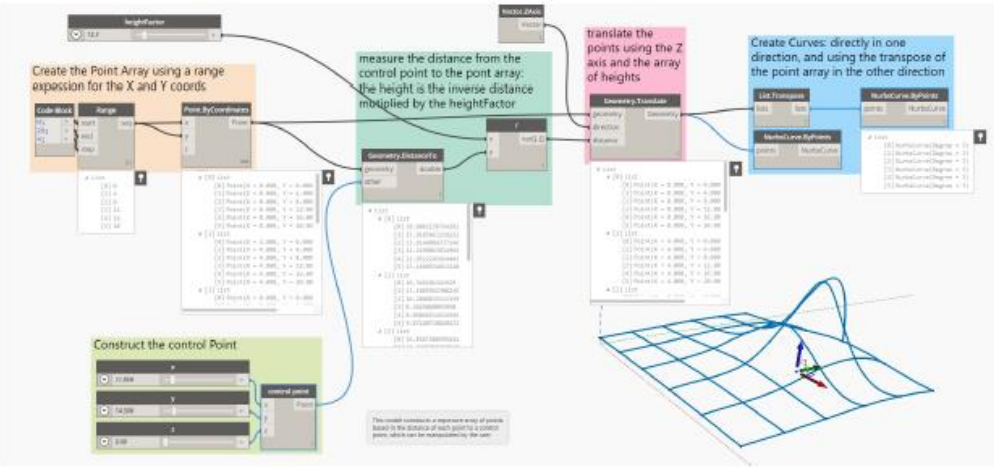
“The abstraction barrier is determined by the minimum number of new abstractions that must be mastered before using the system.” Green and Blackwell.

A domain specific application should not to avoid abstractions, but avoid abstractions becoming a barrier. The same abstraction may sometimes be a barrier; at other times the key idea that ‘changes the way you think’.

Finally

A domain specific computing system will only be successful if it is more than domain specific and introduces the user to more general purpose computing ideas and their application.

Visual Data Flow programing



Text based Data Flow programing

Text based Imperative programing

DesignScript @ Domain Specific Modelling 2016

Visit www.designscript.io

Questions