7_loops

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

Cecil Balmond

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

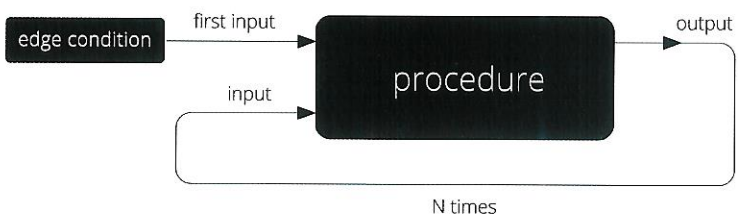


FIGURE 7.1
A recursion starts with an *edge condition* (first input) that generates, by a specific procedure, the first output. The procedure repeats itself N times and, at every step, it generates an output that becomes a new input.

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

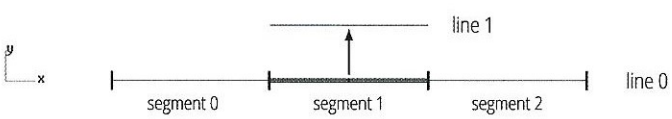
defined recursively by the procedure: $F_n = F_{n-1} + F_{n-2}$. Once the start values or *edge conditions* $F_0 = 0$ and $F_1=1$ are defined the sequence can be calculated.

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

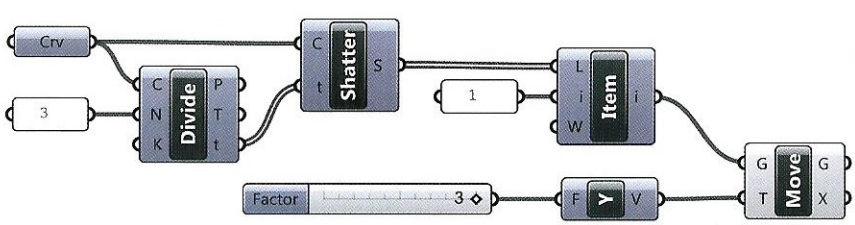
Recursions can be used to achieve ruled complexity, a condition of natural geometries at all scales. The following image shows a geometric configuration based on a simple recursion.

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

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



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



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

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

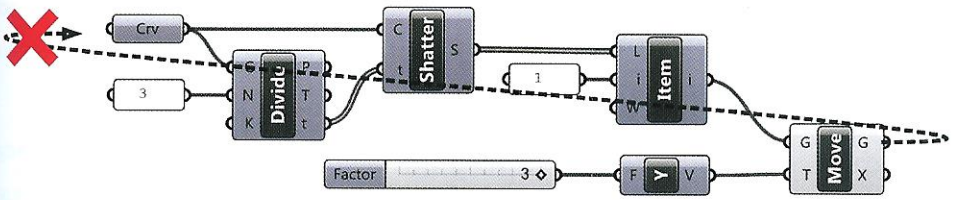
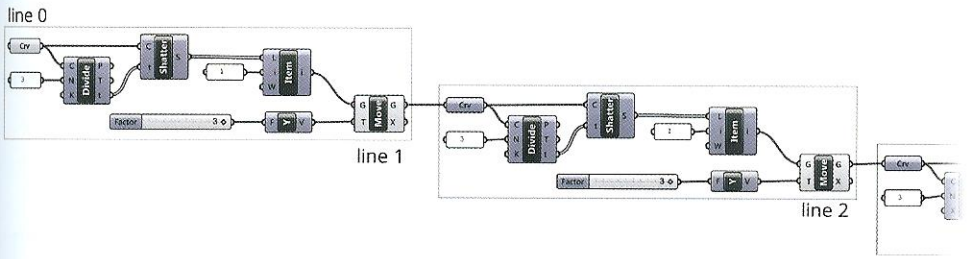


FIGURE 7.2
Loops are not allowed in Grasshopper because of the "left-to-right" connection logic.

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

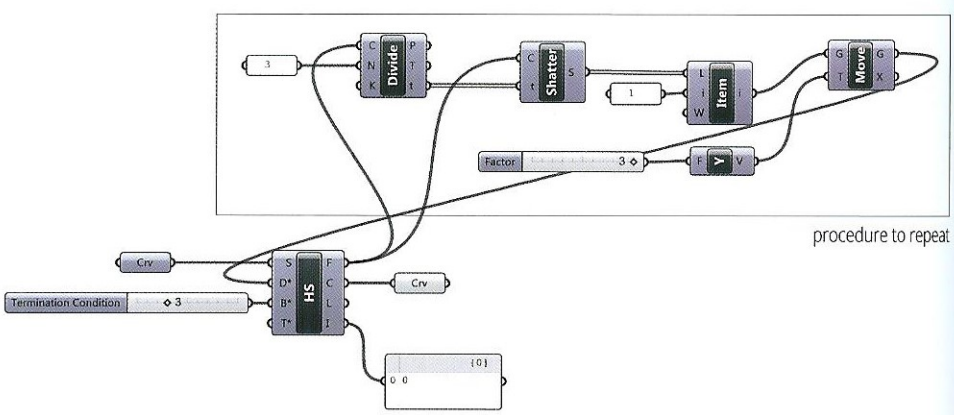


7.1 Loops in Grasshopper: HoopSnake component

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

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

The procedure is composed in the following definition:

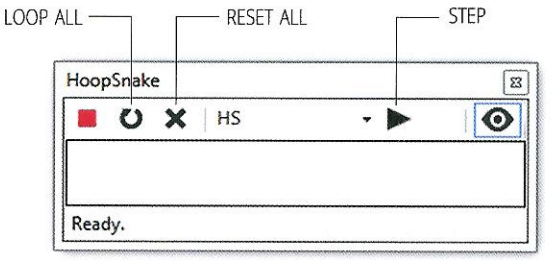


NOTE 18

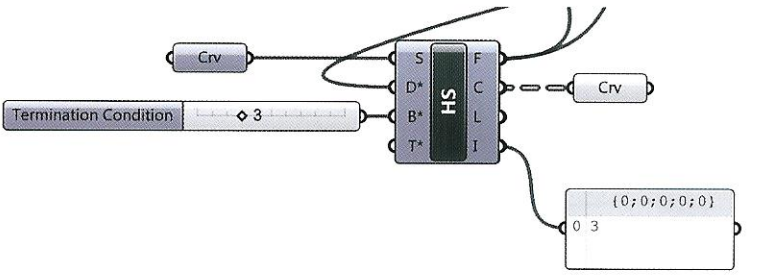
http://yconst.com/software/hoopsnake/.

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

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



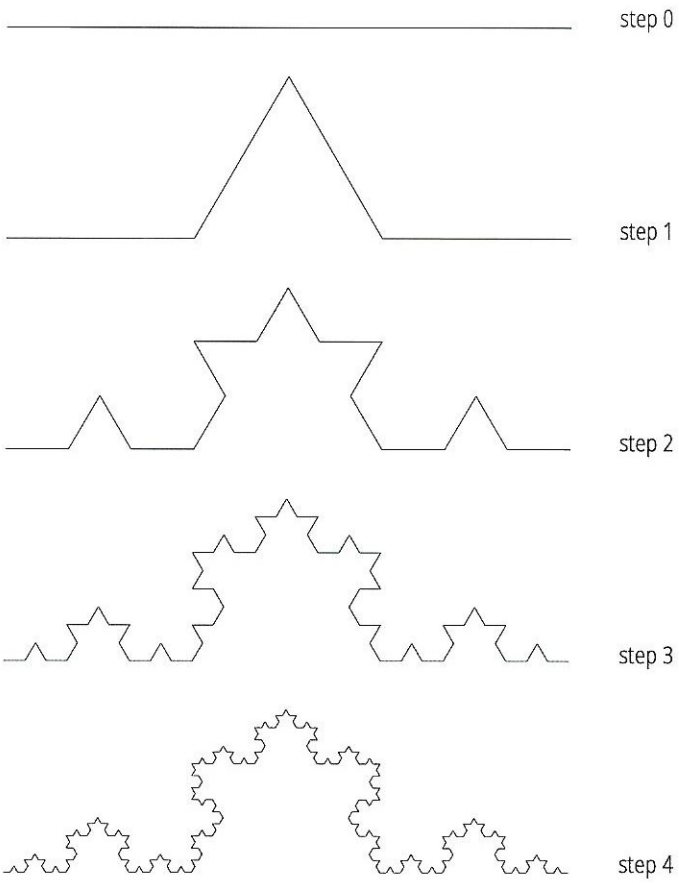
The output (I) provides the iteration count, when *Auto Loop All* button is run the recursion terminates when the counter is equal to the value set in B*.



7.2 Fractals

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

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



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

NOTE 19

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

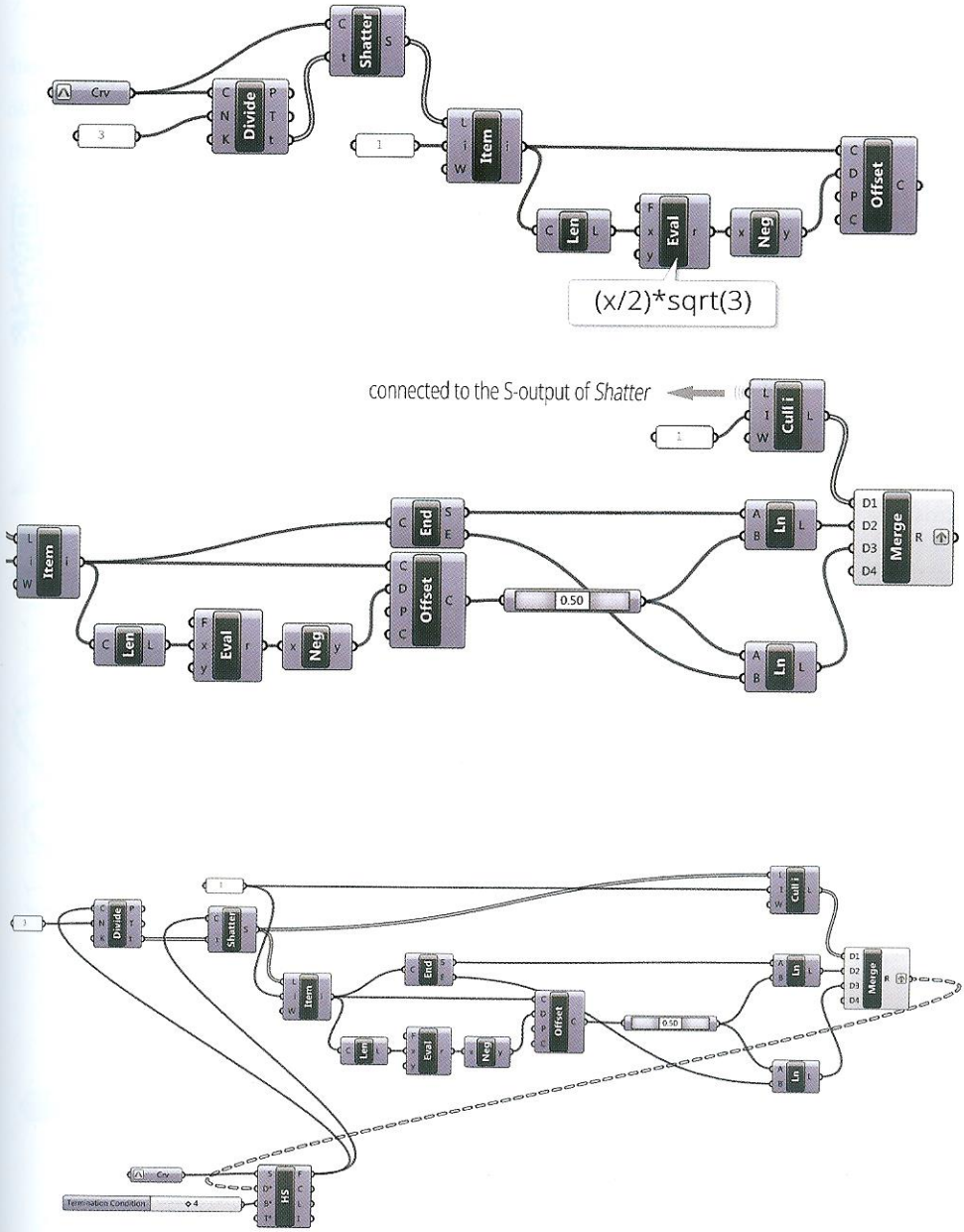


FIGURE 7.3

Koch curve algorithm developed using HoopSnake.

7.2.1 Further Study: Practical Fractals

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

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

2

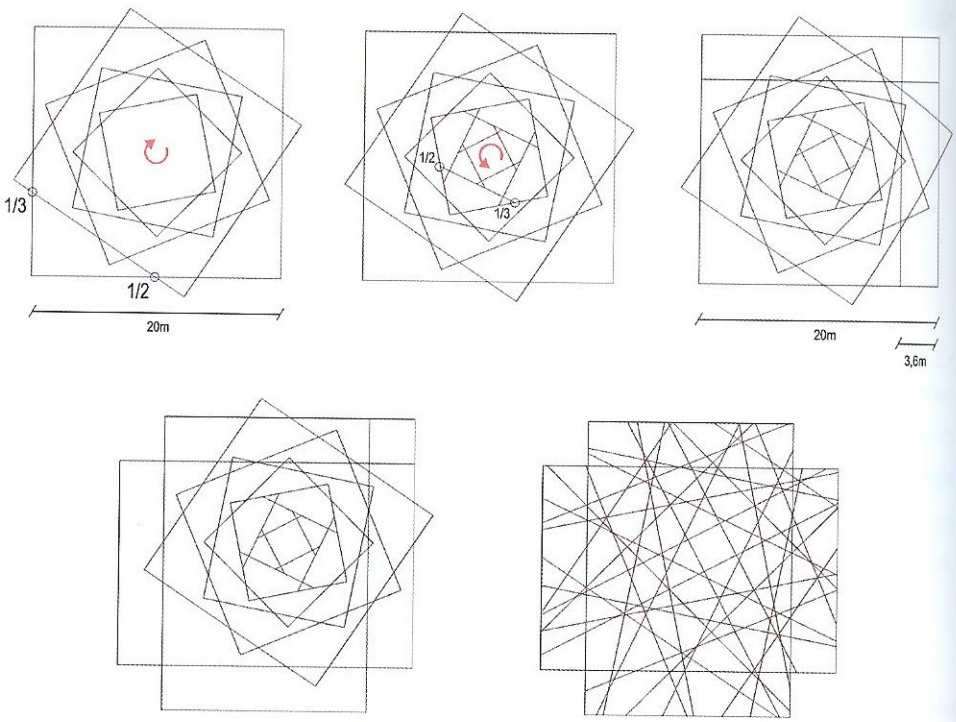


FIGURE 7.4
Serpentine Pavilion 2002, schemes of iteration.

7.2.2 Tridimensional fractals

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

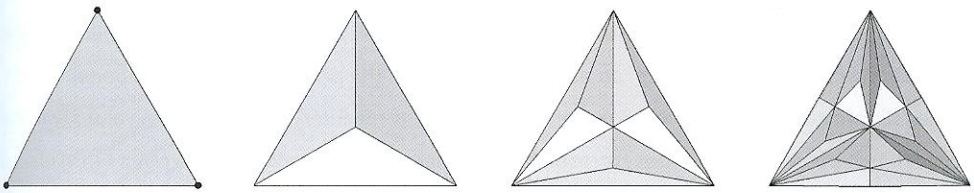
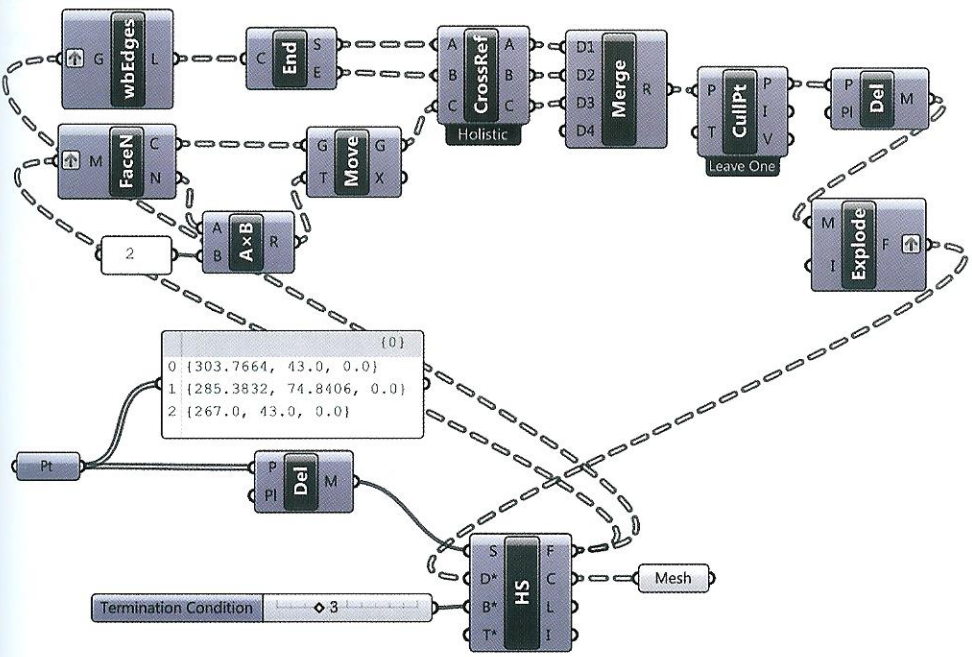


FIGURE 7.5
A tridimensional fractal obtained by three iterations of a recursive algorithm.



7.3 Loops in Grasshopper: Loop component

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

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

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

NOTE 20

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

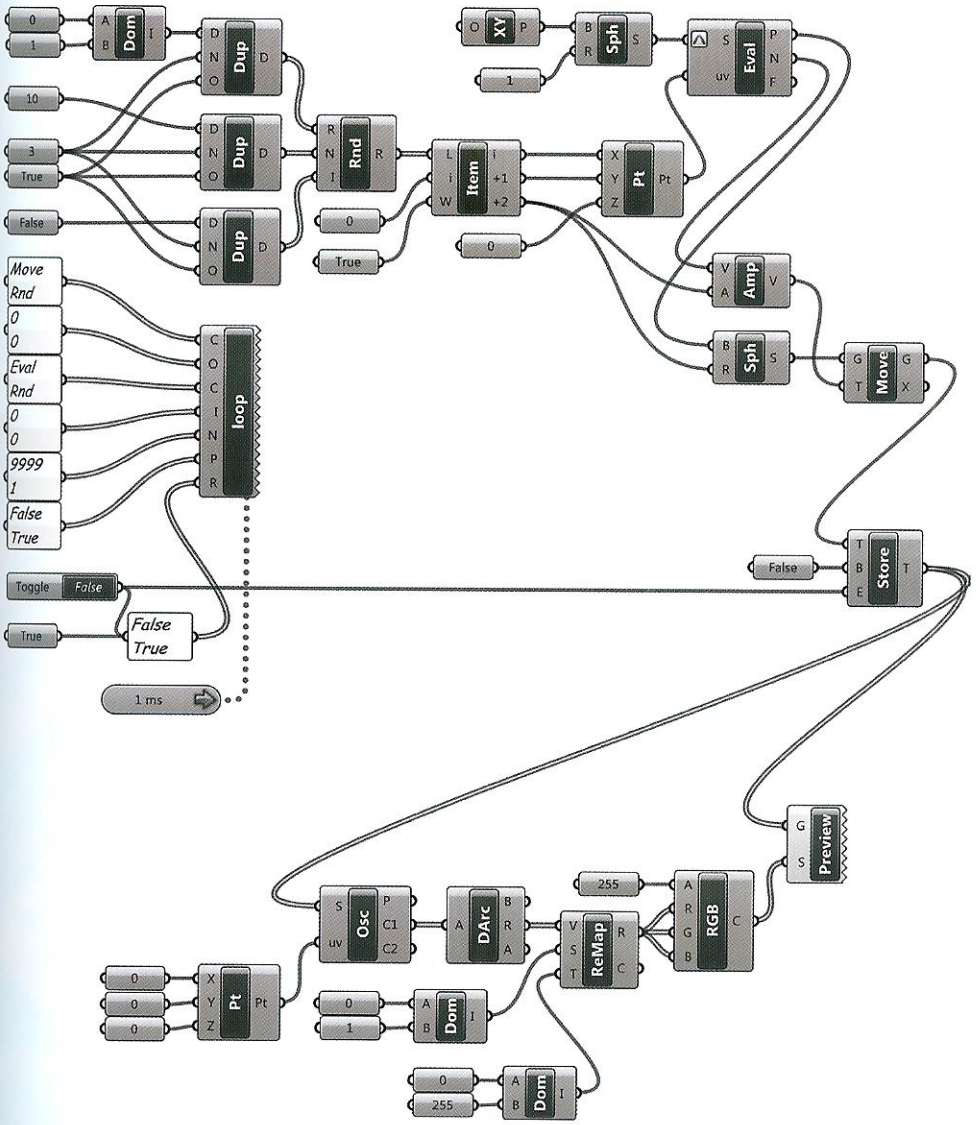
http://antonioturiello.blogspot.com/

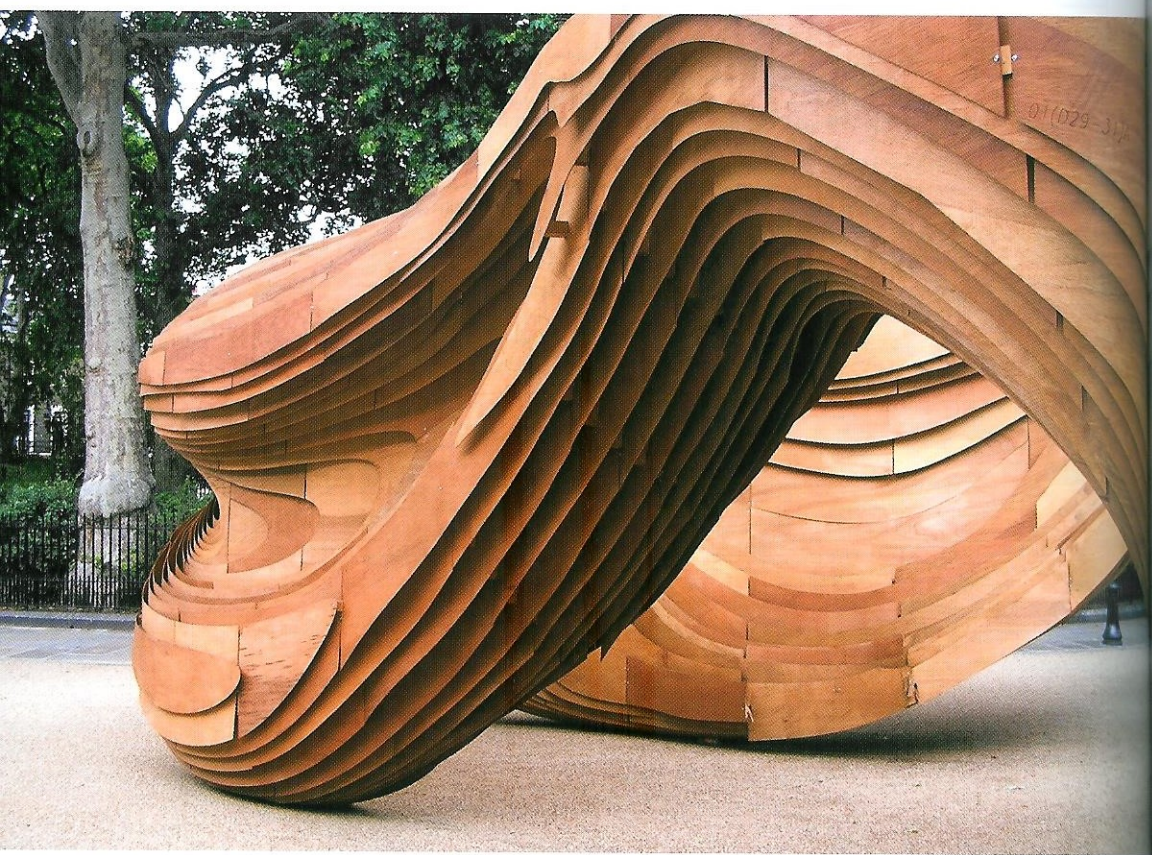
NOTE 21

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

NOTE 22

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


0
1
Dom
10
3
True
False
Dup
Rnd
0
True
XY
Sph
Eval
Item
Pt
Amp
Move
Move
Rnd
0
0
Eval
Rnd
0
0
9999
1
False
True
loop
Toggle
False
True
False
True
1 ms
False
Store
Preview
Osc
DArc
ReMap
255
RGB
0
0
0
Pt
0
1
Dom
0
255
Dom



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