Iterative Pattern Design via Decodes Python Scripts in Grasshopper

Hao Zheng [1] , Zhe Guo [2] , Yang Liang [3]

[1] University of Pennsylvania, Philadelphia, USA
zhhao@design.upenn.edu
[2] Tongji University, Shanghai, China
1732143@tongji.edu.cn
[3] Tongji University, Shanghai, China
liangyang@tongji.edu.cn

Abstract. With the rapid development of parametric design, Grasshopper, as a visual programming tool for architects, has been widely used. However, although Grasshopper is powerful for data processing, there is a weakness that the data only flows linearly from the first component to the last component, which means it's impossible to update the data iteratively by loop structure in native Grasshopper. So here, we introduce a Python based scripting plug-in Decodes, adding the function of loop construct into Grasshopper while integrating the basic graphical operations with faster mathematical matrix calculation. What's more, in order to bring Decodes into play as far as possible, four iterative patterns are researched and designed through Decodes scripting, demonstrating the strength and necessity of loop construct. The patterns include iterative subdivision patterns (center tiling and pinwheel tiling) and iterative growing patterns (semi-regular tiling and swarm behavior). Also, the core parts of their codes are revealed and deciphered in this article.

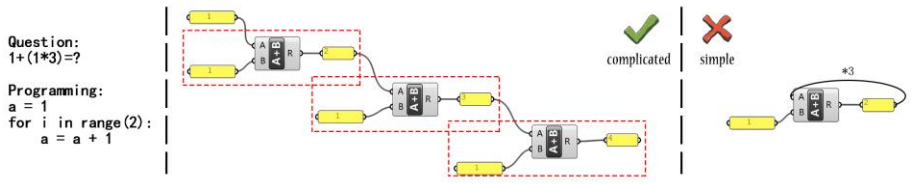
Keywords: Algorithmic design; Iterative pattern; Programming;

1 Grasshopper and its Data Flow

Grasshopper is a visual programming plug-in, developed by David Rutten in McNeel and running in Rhino, whose aim at first was to record and visualize the operation history in Rhino. Although Rhino does have a command called 'history' to help users to check the operation history, there is no parameters recorded after the operation. Users cannot see the history tree, and the parameters inputted when using this command cannot be modified once it is determined.

The invention of Grasshopper solved this problem. It records each operation history with a component (battery), which is visual, reusable, and modifiable. When we make connections between components, we are actually building a history tree, recording each operation, in order to view the contents of previous components.

But this linear modeling logic actually limits the development of Grasshopper, preventing it from evolving into a true programmable modeling tool. In Grasshopper, the data of the following components is strictly based on the data of the previous components. This relationship of dependency cannot be reversed, which means we cannot connect the output of a following component to the input of its previous component, because this will cause a logic error. So the data only flows linearly from the first component to the last component, which means it's impossible to update the data iteratively by loop structure in native Grasshopper (figure 1).


```
Question:
1+(1*3)=?

Programming:
a = 1
for i in range(2):
    a = a + 1
```

Fig. 1. Data Flow Rules in Grasshopper.

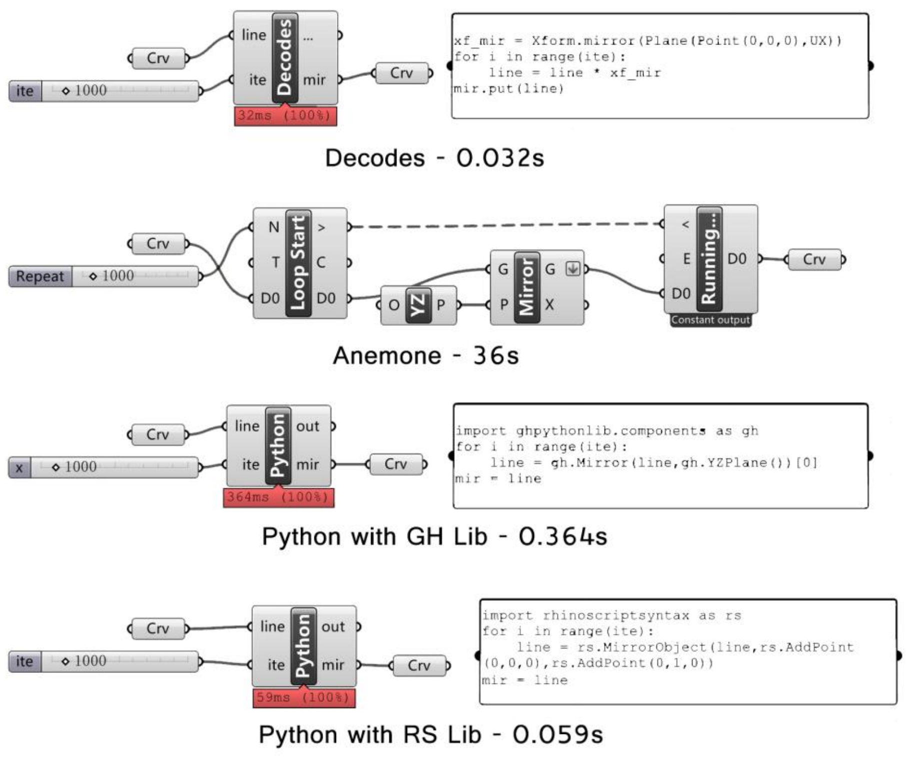
2 Decodes Scripts in Grasshopper

To make loop structure possible in Grasshopper, Decodes, a Python based programming language invented by [1] is introduced into Grasshopper. The syntax of loop construct in Decodes is the same as that in Python, while mathematical matrix operations are developed for transforming geometries, and the basic graphical operations in Rhino and Grasshopper are integrated into Decodes.

2.1 Looping Tools Comparison

Other than Decodes, [2] also developed a Grasshopper plug-in Anemone for constructing loops, and they applied the idea of this looping structure with the Evolutionary Structural Optimization, similar with the genetic algorithm tool developed by [3], to proceed data in modeling. Also, loop structure can be built in Python, with the geometric operation library imported from Grasshopper components or Rhino commands.

Figure 2 shows the comparison of the processing speed of the four loop construct tools, Decodes, Anemone, Python with Grasshopper Library, and Python with Rhino Script Library. After telling the scripts to mirror a giving curve 1000 times, Decodes performs fastest by only costing 0.032 seconds to complete the loop, while the Grasshopper components in Anemone spends around 36 seconds, calling the mirror component in Grasshopper multiple times, which is very time consuming. When importing Grasshopper components as a library into Python script, the process still takes 0.364 seconds, which is ten times slower than Decodes. This proves that the geometric operations in Grasshopper is slower than the mathematical matrix operations in Decodes. When using Rhino Script Library in Python to mirror the curve, it runs 0.059 seconds, just one time slower than Decodes.



```
xf_mir = Xform.mirror(Plane(Point(0,0,0),UX))
for i in range(ite):
    line = line * xf_mir
mir.put(line)
```

```
import ghpythonlib.components as gh
for i in range(ite):
    line = gh.Mirror(line,gh.YZPlane())[0]
mir = line
```

```
import rhinoscriptsyntax as rs
for i in range(ite):
    line = rs.MirrorObject(line,rs.AddPoint
(0,0,0),rs.AddPoint(0,1,0))
mir = line
```

Fig. 2. Comparison of the Processing Speed of Decodes, Anemone, Python with Grasshopper Library, and Python with Rhino Script Library.

So, among these four looping tools, Decodes are most efficient, since it allows the usage of Xform operation, transforming geometries by matrix calculation rather than relying on Grasshopper or Rhino commands.

2.2 Loop Construct of Decodes Scripts in Grasshopper

In Decodes, there are three ways to construct a loop. First, the simplest method is called 'for loop' (table 1). For example, in line 1, a variable 'a' is defined and equals to 1. Then in line 2, a variable 'i' is defined and the value will be given as an integer from 0 to 2, and each time when the value of 'i' changes, the contents in line 3 (and following lines starting with 4 spaces) will be executed. So after the loop, the value of 'a' equals to 1+(0+1+2)=4. 'For loop' is the most widely used loop structure, since it's very easy to understand and the loop can be stopped by counts.

```
1.  a = 1
2.  for i in range(2):
3.      a = a + i
```

Table 1. ‘For loop’ example.

However, the stop signal of a loop can also be regarded as a Boolean variable. In the case of ‘for loop’, when there are no more unassigned ‘i’ values, a hidden Boolean variable will be defined as ‘True’ (or ‘False’) to stop the loop. So another loop structure called ‘while loop’ (table 2), which is more flexible, can be used to replace ‘for loop’. For example, in line 3, if the inequality (i<=2) is true, the contents in line 4 and 5 (and following lines starting with 4 spaces) will be executed, then the program will run from line 3 again until the inequality turns to be false. In this case, after the loop, the variable ‘a’ also equals to 1+(0+1+2)=4, but the variable ‘i’ equals to 3, since 3 is the smallest integer larger than 2. What’s more, the stop signal in ‘while loop’ can be other types of expressions, such as (a==4).

```
1.  a = 1
2.  i = 0
3.  while (i <= 2):
4.      a = a + i
5.      i = i + 1
```

Table 2. ‘While loop’ example.

While ‘for loop’ and ‘while loop’ construct the basic loop structure in Decodes, there is another way called ‘recursion’ to construct a loop (table 3). For example, in line 1, a function called ‘plus’ is defined, and the contents from line 2 to line 6 will be executed once the function ‘plus’ is called in the program. In line 9, the function ‘plus’ is called in the first time, and in line 5, the function is called again while executing the function itself. So the program will run the function again with updated variable ‘a’ and ‘i’, until (i<=2) equals to ‘False’. In recursion structure, a complicated problem is deconstructed into small problems, whose solutions are same but variables are different. It actually achieves a loop by calling itself multiple times instead of looping the script directly. In some applications, the recursion structure is easier to understand by showing the current and future goals in the same function.

```
1.  def plus(a,i):
2.      a = a + i
3.      i = i + 1
4.      if (i<=2):
5.          a = plus(a,i)
6.      return a
7.  a = 1
8.  i = 0
9.  a = plus(a,i)
```

Table 3. ‘Recursion’ example.

In conclusion, Decodes provides choices to process data in loops, including both mathematical and geometrical data in Grasshopper. Figure 3 shows the work flow of the three loop structures. The factor for which structure to choose depends on the aims and applications of the program.

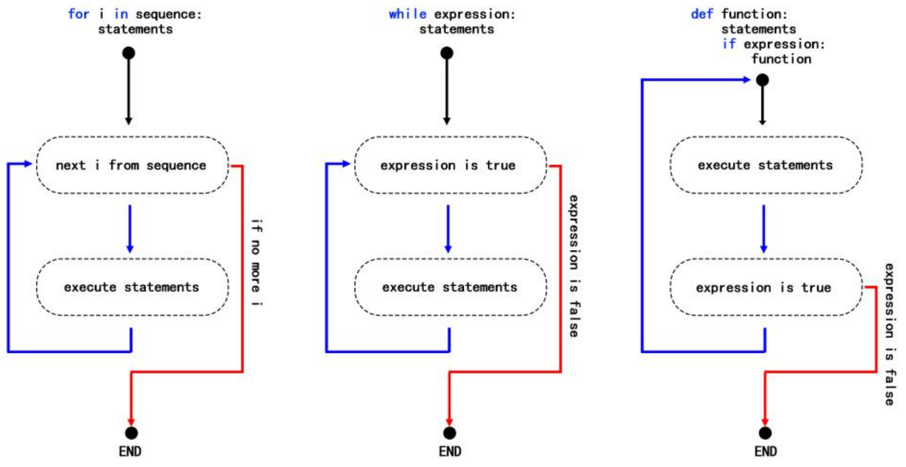


Fig. 3. Work Flow of the Three Loop Structures.

3 Iterative Subdivision Patterns

With Decodes, the powerful loop-build tool in Grasshopper, now it's possible to describe iterative designs through scripting. Iterative patterns as the basic representatives were researched and modelled.

3.1 Center Tiling

First, a subdivision rule called 'center tiling' was invented (figure 4). A rectangle is subdivided into 4 triangles, then for each triangle, the central point (point 3) is found and point 4, 5, and 6 are the nearest points from the three sides to the central point. After drawing lines between points as figure 4 shows, a triangle is further subdivided into 6 triangles, and each triangle is sent to the process for the next round of subdivision.

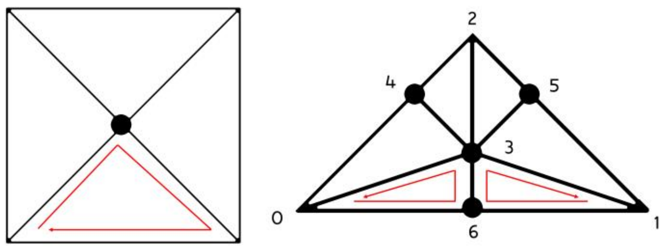


Fig. 4. Subdivision Rule of Center Tiling.

The core part of the Decodes script shows the basic loop structure from line 9 to line 12 (table 4). For each iteration, the triangles in the variable ‘faces’ are extracted and sent to the function ‘trisub_cen’ from line 1 to line 8. Then the function will calculate the positions of the points and return the subdivided triangles. Last, the variable ‘faces’ will be updated by the new triangles and be ready for the next round of the loop.

```
1.  def trisub_cen(fac):
2.      subfaces = []
3.      cen = face_cen(fac)
4.      for seg in fac:
5.          subpoi = seg.near(cen)[0]
6.          subfaces.append((Segment(seg.spt,subpoi),Segment(subpoi,cen),Segment(cen,seg.spt)))
7.          subfaces.append((Segment(seg.ept,subpoi),Segment(subpoi,cen),Segment(cen,seg.ept)))
8.      return subfaces
9.  for n in range(iter-1):
10.     subfaces = []
11.     for fac in faces:
12.         subfaces.extend(trisub_cen(fac))
```

Table 4. Decodes Script of Center Tiling.

Figure 5 shows the results of center tiling with different looping times. With the further subdivision, the pattern will be more complex. And it’s easy to adjust the looping times to achieve different patterns.

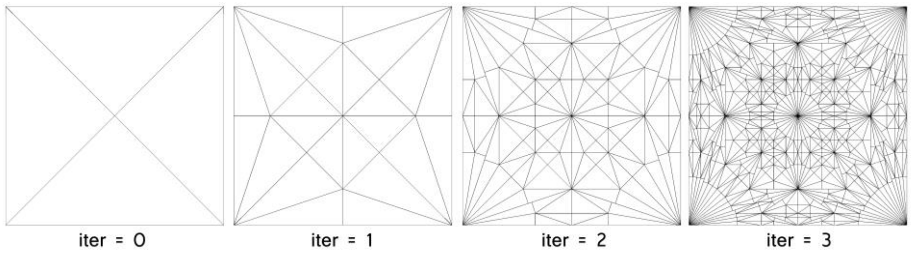


Fig. 5. Results of Center Tiling.

3.2 Pinwheel Tiling

Next, a pattern called pinwheel tiling that was invented by [4] was selected as the second example. Figure 6 shows the overall pattern of pinwheel tiling. The basic geometry is a triangle, whose three sides keep a ratio of 1:2:√5. After marking the three vertexes as point 0, 1, and 2, four more points (point 3, 4, 5, and 6) are added, so that the five new triangles shown in figure 5 will have the same side radio as the original triangle, which can be processed into the next round of subdivision.

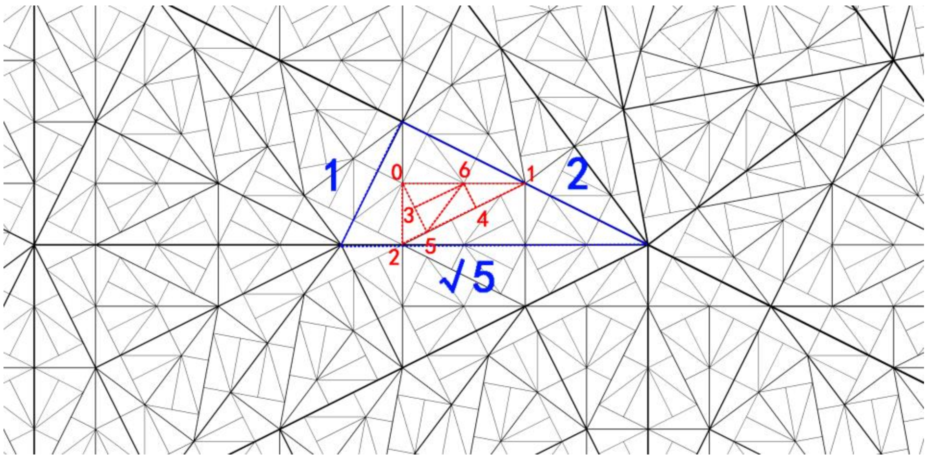


Fig. 6. Subdivision Rule of Pinwheel Tiling.

The loop structure of pinwheel tiling is similar to that of center tiling, in which the original geometry will be subdivided into several new geometries, and then the new geometries will be put into the list to wait for further subdivision. In the Decodes script (table 5), the variable 'tiles' from line 10 to line 15 stores the information of the triangles, which was created from the previous round and will be processed in the current round. The class 'PinwheelTile' from line 1 to line 9 defines a variable group to represent the information of a triangle, its variable 'data' gives relative coordinates of point 3, 4, 5, and 6, and its function 'inflate' works to generate the subdivided triangles according to the rule of pinwheel tiling, and return the new triangles. So by looping this script, the variable 'tiles' is updated as the final result.

```
1.  class PinwheelTile():
2.      def __init__(self,pts):
3.          self.pts = [pts[0],pts[1],pts[2]]
4.          data = [[0.1,0.4],[0.6,0.4],[0.2,0.8],[0.5,0]]
5.          for i in range(4):
6.              self.pts.append(pts[0]+Vec(pts[0],pts[1]).normaliz
    ed(Segment(pts[0],pts[1]).length*data[i][0])+Vec(pts[0],pts[2]
    ).normalized(Segment(pts[0],pts[2]).length*data[i][1]))
7.          self.segs = [Segment(self.pts[0],self.pts[1]),Segment(
    self.pts[1],self.pts[2]),Segment(self.pts[2],self.pts[0])]
8.      def inflate(self):
9.          return [PinwheelTile([self.pts[3],self.pts[6],self.pts
    [5]]),PinwheelTile([self.pts[4],self.pts[5],self.pts[6]]),Pinw
    heelTile([self.pts[3],self.pts[6],self.pts[0]]),PinwheelTile([
    self.pts[5],self.pts[0],self.pts[2]]),PinwheelTile([self.pts[4
    ],self.pts[1],self.pts[6]])]
10. for n in range(iter):
11.     nxt_tiles = []
12.     for i in range(len(tiles)):
13.         temp = tiles[i].inflate()
```

```
        nxt_tiles.extend(temp)
    tiles = nxt_tiles
```

Table 5. Decodes Script of Pinwheel Tiling.

As the variable ‘iter’ increases, the original triangle is further subdivided (figure 7). By inputting different weights, patterns in different iterations will have different line widths, showing clearer relationships between iterations.

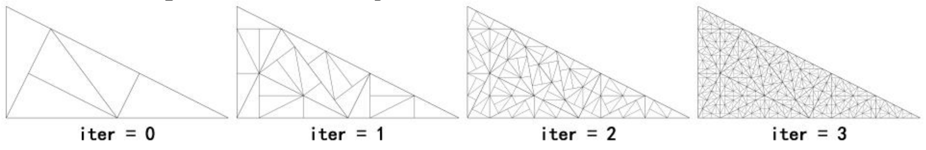


Fig. 7. Results of Pinwheel Tiling.

4 Iterative Growing Patterns

In addition to the subdivision pattern, which regards the original geometry as an overall boundary, growing pattern processes a geometry as a unit, and copy it to fill in the space, which can also be achieved through loop structure in Decodes scripts.

4.1 Semi-regular Tiling

A paving pattern called semi-regular tiling was first described by [5], which means two tiles meet at a common edge and the same pattern of polygons surrounds every vertex. Figure 8 shows three possible semi-regular patterns, which are represented by a series of numbers, showing the edge numbers of surrounding polygons. However, since the length of edges are the same in all the polygons, semi-regular tiling can also be regarded as a result of multiple mirror transformations of a single unit geometry, which consists of several lines with the same start point but different end points (blue marks in figure 8). So after drawing the unit geometry based on the edge numbers, the whole tiling can be grown by mirroring the unit geometry to get the new geometries (yellow marks in figure 8), and then further mirroring the new geometries to get the geometries in the next generation.

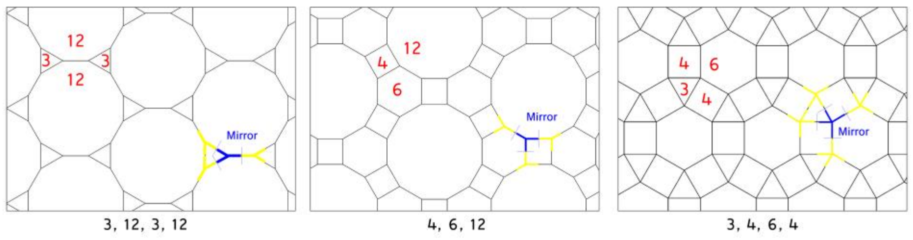


Fig. 8. Growing Rule of Semi-regular Tiling.

In the Decodes script (table 6), line 1 constructs a basic loop to set data in different generation, while line 2 builds a second loop to assign the directions of mirroring. The variable ‘xf_mir’ in line 4 is a matrix, representing the mathematical calculation of mirroring, which is very easy to construct through the ‘Xform’ class in Decodes. By updating the variable ‘list’, new geometries will be added and ready for the next generation.

```
1.  for k in range(gen):
2.      for seg in list[k]:
3.          sublist = []
4.          xf_mir = Xform.mirror(Plane(seg.ept,seg.vec))
5.          for segb in list[k]:
6.              sublist.append(segb*xf_mir)
7.          list.append(sublist)
```

Table 6. Decodes Script of Semi-regular Tiling.

Figure 9 shows the pattern of (3, 12, 3, 12) tiling in different generations. Since the mirror operation may cause a mirroring back problem, which means the geometries in next generation may coincide with the geometries in the previous generation, removing the overlapped geometries regularly helps to reduce the program’s computation load.

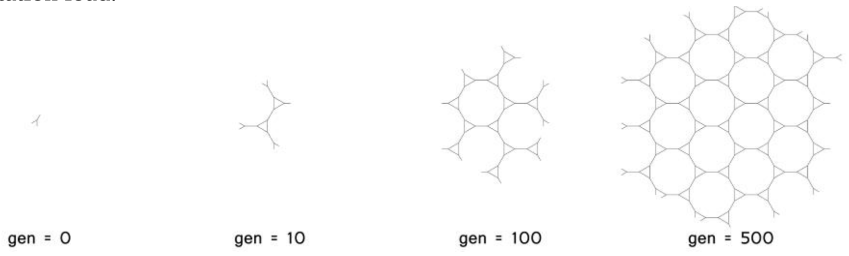


Fig. 9. Results of Semi-regular Tiling.

4.2 Swarm Behavior

Other than tilings and patterns with clear mathematical expressions, loop structure can also apply to customized and complex rules to create geometries. Swarm behavior, firstly simulated through computer programs by [6], follows three rules (figure 10) that, a bird is,

(1) flying away to avoid collisions with other birds (separation);
(2) flying close to remain close to other birds (cohesion);
(3) flying in the same direction as other birds (alignment).

Under the guidance of these three rules, each bird will have a unique flying path, whose current position point is updated by the previous position points of all the paths. And all the paths together draw a 3D/2D pattern, which is also defined based on the initial positions and flying directions of birds.

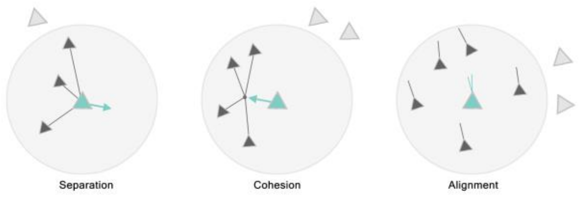


Fig. 10. Growing Rule of Swarm Behavior.

The basic loop structure from line 17 to line 19 is very simple, in which the program assigns the task of calculating the bird's position and flying direction of next generation to the function 'step' from line 1 to line 16 (table 7). In the function 'step', the variable 'vec' defines the direction and speed as a vector, and it will be updated from line 4 to line 15 according to the three rules, which means that, for every other bird, whose distance to the current bird is,

(1) smaller than 2 units, a vector pushes the bird away will be added into 'vec';

(2) larger than 2 units but smaller than 4 units, a vector drags the bird back will be added into 'vec';

(3) smaller than 6 units, a vector from the current bird toward the central point of the positions of all this kind of birds will be added into 'vec'.

Then in line 16, the effect of 'vec' will be given to the bird and update the position point. Finally, by connecting all previous position points for every bird, a paths drawing will be generated as the final pattern. Through adjusting the distance ranges and the weights for the three rules, or randomly setting the initial parameters, different final patterns can be achieved.

```
1.  def step(self,EBird):
2.      vec = Vec(0,0,0)
3.      cenpts = []
4.      for i in range(len(EBird)):
5.          dis = self.pos.distance(EBird[i].pos)
6.          if dis<2:
7.              vec += Vec(EBird[i].pos,self.pos).normalized(0.25)
8.          elif dis<4:
9.              vec += Vec(self.pos,EBird[i].pos).normalized(0.25)
10.         if dis<6:
11.             cenpts.append(EBird[i].pos)
12.     if not len(cenpts)==0:
13.         cenpts.append(self.pos)
14.         cenpt = Point.centroid([pt for pt in cenpts])
15.         vec += Vec(self.pos,cenpt).normalized(0.25)
16.     self.steer(vec)
17. for i in range(gen):
18.     for bird in flock:
19.         bird.step(flock)
```

Table 7. Decodes Script of Swarm Behavior.

Figure 11 shows the growing of swarm behavior pattern in different generations. With more birds being assigned in the system, the more complex pattern will be generated.

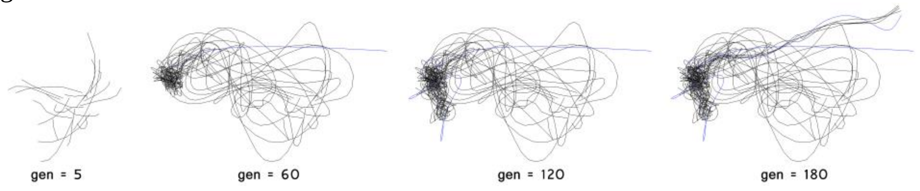


Fig. 11. Results of Swarm Behavior.

5 Conclusion

Decodes is a powerful scripting tool, which remedies the defect of lacking loop structure in Grasshopper, by adding the function of loop construct while integrating the basic graphical operations. Iterative patterns and other designs, which are described through loop structure, can be achieved via the data processing in Decodes and the geometrical modelling in Grasshopper. The patterns generated by Decodes have the potential to become a fa çade design as figure 12 shows.

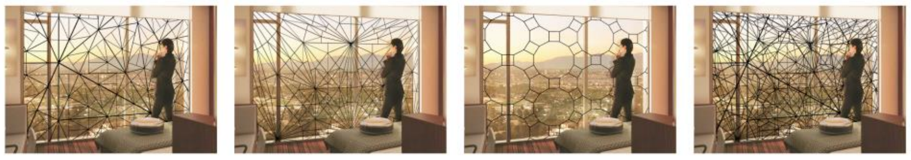


Fig. 12. Applications of Decodes Patterns in Fa çade Design.

Acknowledgement

This research was done as a series of exercises for course ARCH 229, Special Topics in Digital Design Theories and Methods, taught by Prof. Kyle Steinfeld at University of California, Berkeley.

References

1. Steinfeld, K. and J. Ko, *Decodes – A Platform-Independent Computational Geometry Environment*, in *the 18th International Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA 2013)*. 2013: Singapore.
2. Zwierzycki, M., *Ewolucyjne narzędzia cyfrowe w formowaniu struktur przestrzennych.* Archivolta, 2013(2): p. 54--61.
3. Rutten, D., *Galapagos: on the logic and limitations of generic solvers.* Architectural Design, 2013. **83**(2): p. 132-135.

4. Radin, C., *The pinwheel tilings of the plane.* Annals of Mathematics, 1994. **139**(3): p. 661-702.
5. Williams, R., *The geometrical foundation of natural structure*. 1979: Dover New York.
6. Reynolds, C.W. *Flocks, herds and schools: A distributed behavioral model*. in *ACM SIGGRAPH computer graphics*. 1987. ACM.