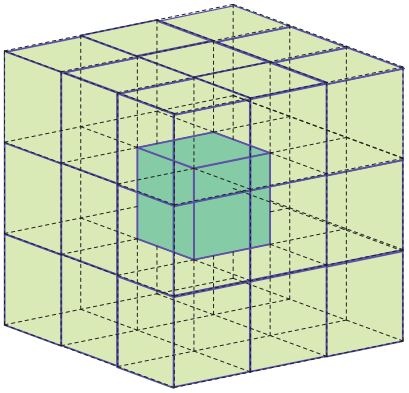


moore neighbourhood 3D

IF DEAD:
13, 14 , 17, 18 , 19 = birth

IF ALIVE:
13 - 26 = survive

FIGURE 01: Rules for the Cellular Automata used as the initial environment. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

CONTEXT

Initially, we were interested in basic self-organizing computation systems like cellular automata and their capacity of expressing architectural qualities.

Cellular Automata are systems that organize themselves by interacting in a local level and are able to produce complex global results and behaviours.

For our mid-term assignment we started investigating 2D cellular automaton structures to better understand the principles and their outcomes. We then moved to 3D cases to further explore the system. We particularly chose to work with the 'Moore Neighbourhood' and after experimenting with various rules we focused on the results of the 3D system with the following rules:

Every dead cell that has exactly 13, 14, 17, 18, 19 alive neighbours in his 'Moore Neighbourhood' changes its state to alive state. Every dead cell with other number of neighbours stays dead.

Every alive cell with 13 up to 26 alive neighbours in his 'Moore Neighbourhood' survives to the next step. Every alive cell with less neighbours eventually dies.

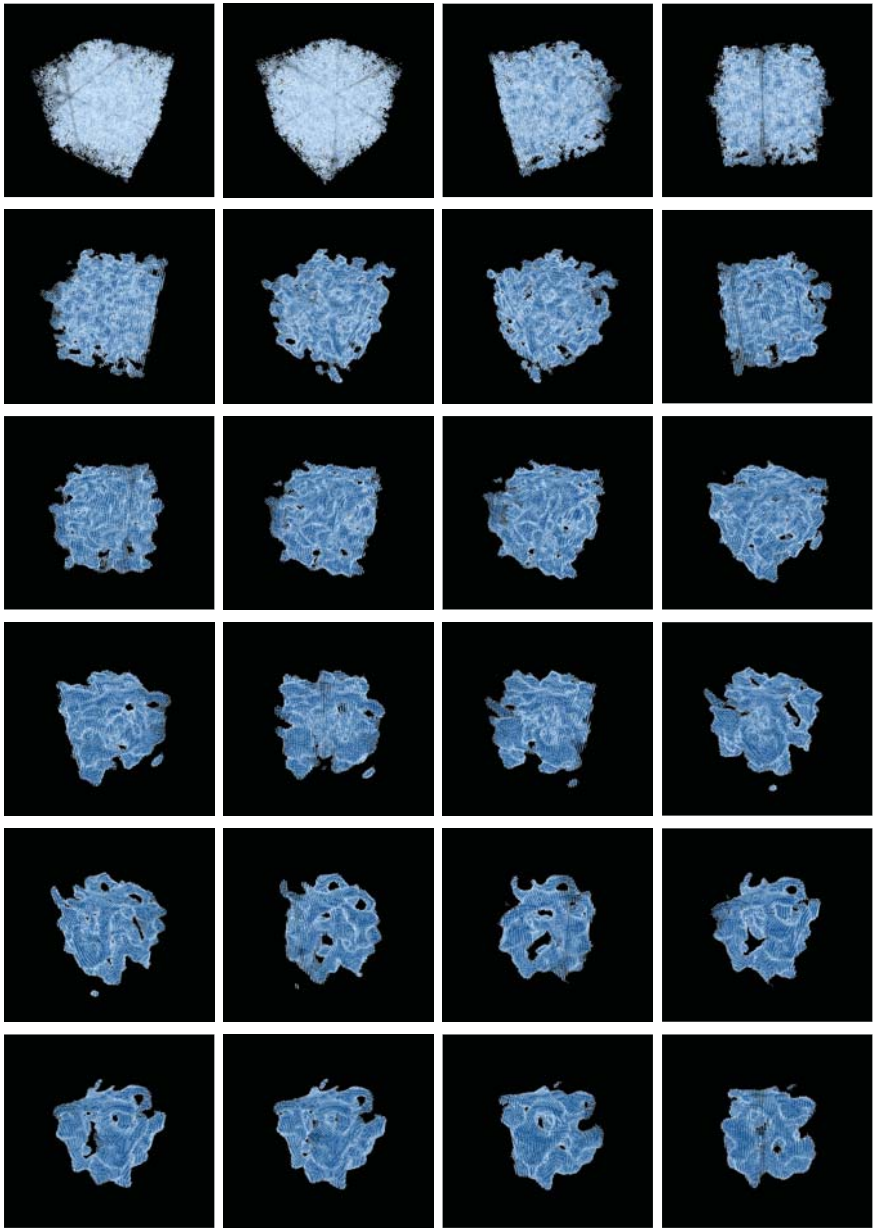


FIGURE 02: Stills from the Cellular Automaton process. Images were taken every 40 frames from an rotating amimation running on Processing (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

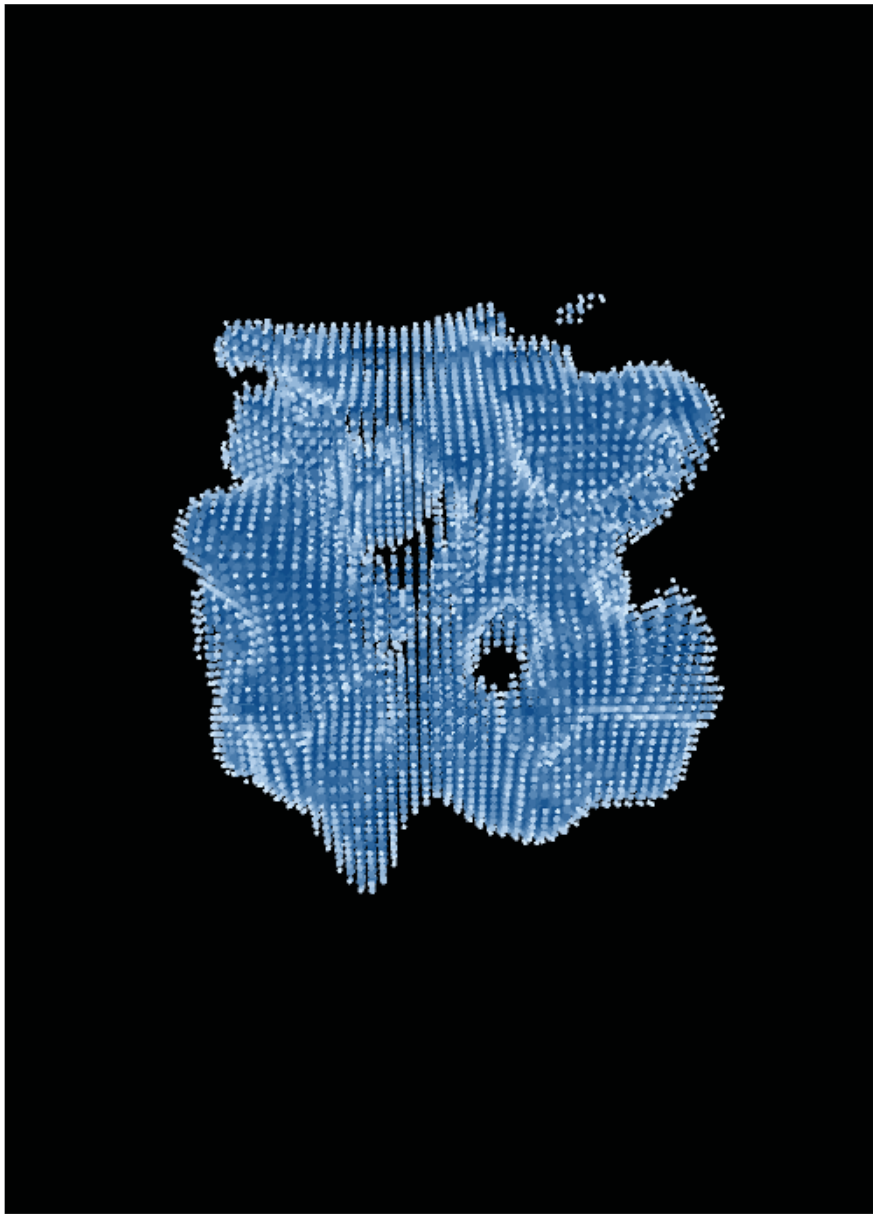


FIGURE 03: Still from the Cellular Automaton model. Final result after 16 iterations.
(*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

The series of models that we produced informed us about interesting spatial qualities that can emerge from self-organizing computational systems. At this point though, we had not embedded any further logic yet.

Strict and precise rules and computational logic produced emerging geometry. It was not possible though to yield more interesting results, able to be translated to other applications such as architectural ones.

Thus, we started getting interested in perfomance through computational logic. More specifically, we got really interested in taking advantage of the already produced substrate and regenerate another version of it using optimal material distribution.

We realize our interest is not purely in Cellular Automata models, but rather in finding a computational means of translating rules that could express performance criteria through local levels of interaction with the substrate. Therefore, we keep our CA model as a substrate that has an interesting geometry and we shift to Agent Based Systems to pursue our goal.

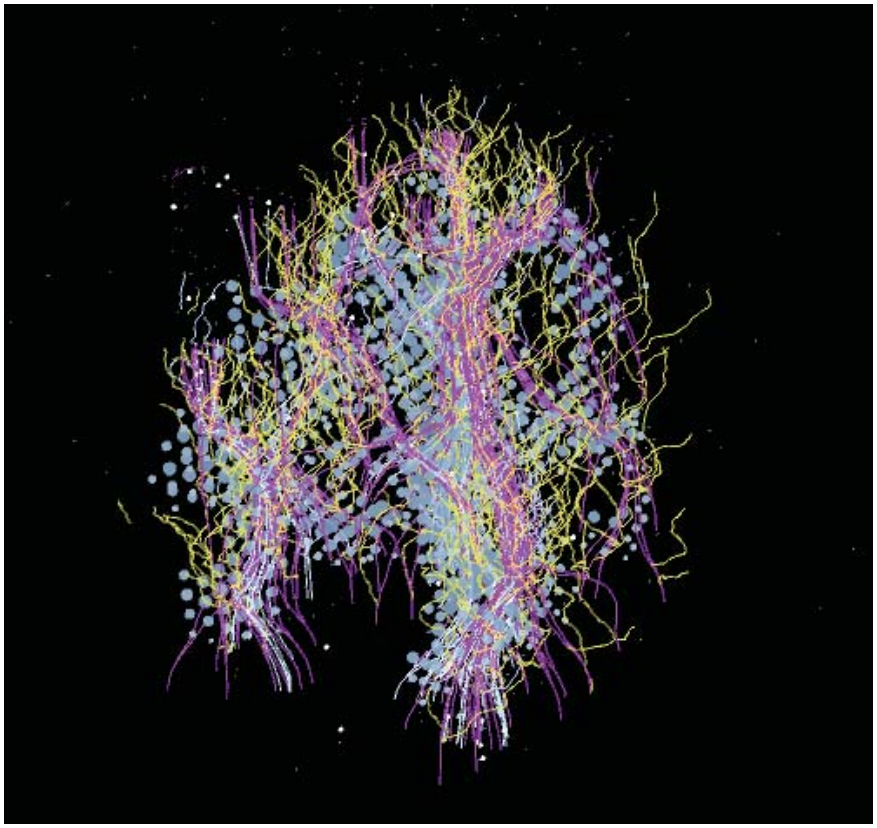


FIGURE 04: Still of animation after running though the whole environment. Initial agent size is 40.
(*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

CONCEPT OVERVIEW

An Agent Based System was incorporated to generate a network that can express optimal material distribution. Density concentration through flow of agents in the direction of the forces leads us to structural performance and material efficiency.

Thus, we let agents flow from the bottom of the substrate following a set of rules interacting locally with their environment and themselves being influenced by a global unitary force.

In the end we get a system in which the summation of the forces/vectors acting on the agents on a local level create a condition of an efficient material distribution. Regarding the process, every agent first senses its environment, it evaluates it and then interacts with it and with the rest of the agents. Thus, from local intelligence we deduct global emergent performance.

We can divide our behaviours into evaluation/sensory ones and steering/locomotion ones at one level and at another level into agent-environment interaction and agent-agent interaction.

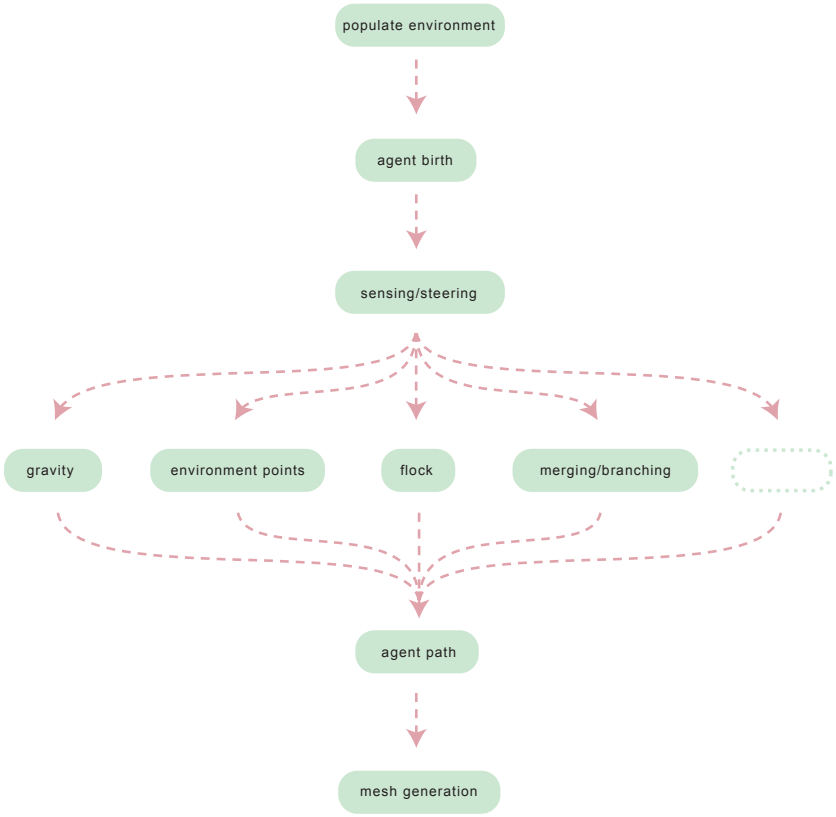


FIGURE 05: Pseudocode. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

FIGURE 06: Pseudocode highlighted. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

AGENTS BEHAVIOURS

At each step every agent first senses its environment. In reality this means that every agent senses if there is any point within its vision and if there is, it assigns a weight to each of these points according to the number of their neighbours. In this way the agents evaluate their environment and decide which way they should follow.

Agents sense their relation to other agents as well. Flock behaviour and a merging/branching behaviour are calibrated by this condition.

Eventually, the summation of all the behaviours (environment points, flock, merging/branching, gravity) define the velocity vector for each agent at each step.

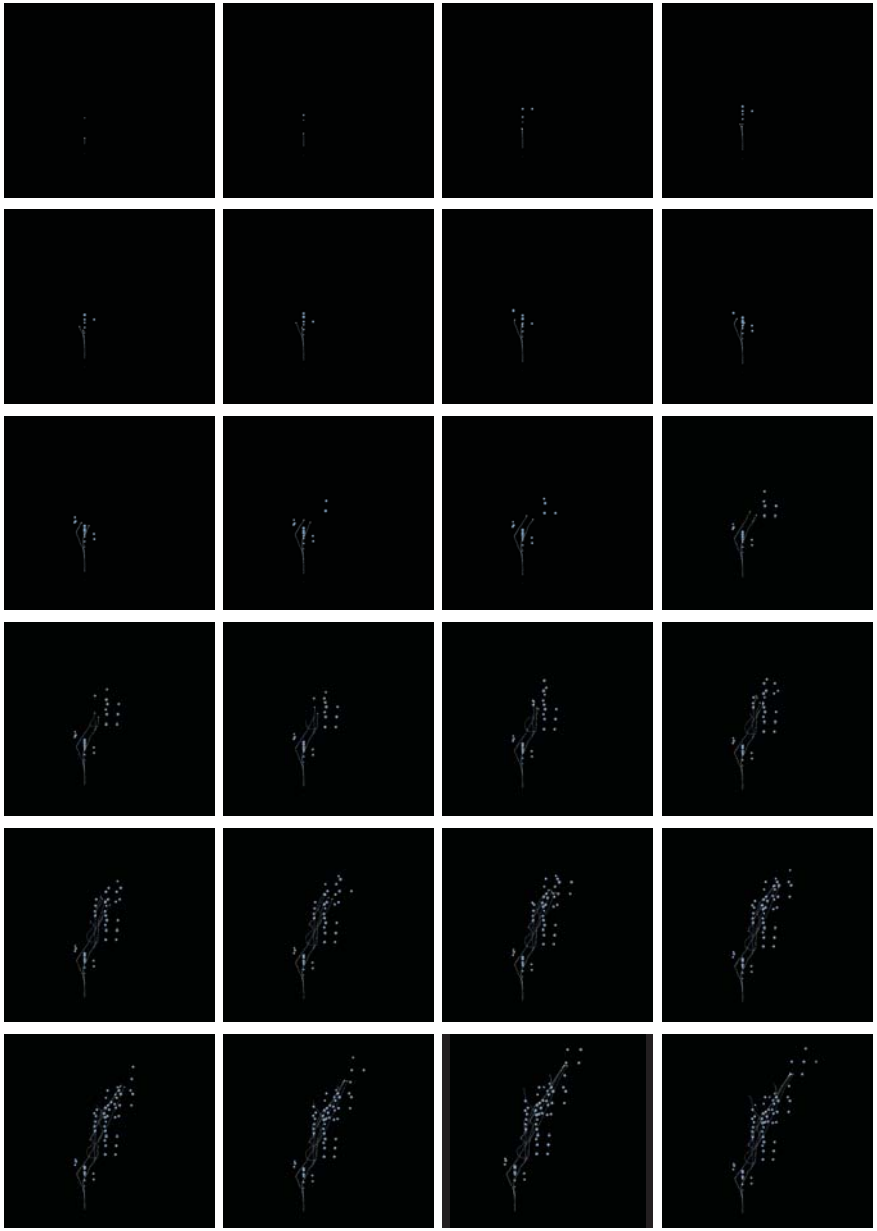
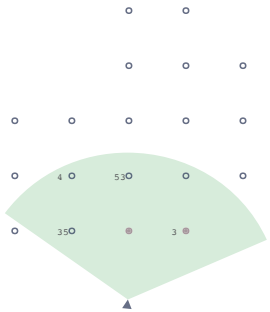
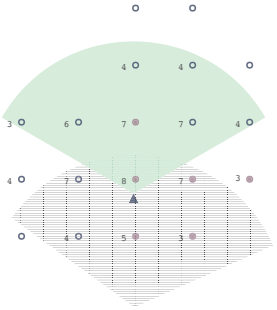


FIGURE 05: Weight assignment on the points of the environment. The initial agent size is 1. The craziness is disabled. Images were taken every 40 frames from a Processing animation (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)



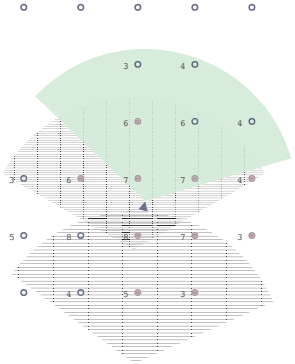
ENVIRONMENT EXPLORATION 01

First, the agent approaches the points and as soon as they get inside its vision it counts its neighbours, which eventually is the weight for each point.



ENVIRONMENT EXPLORATION 02

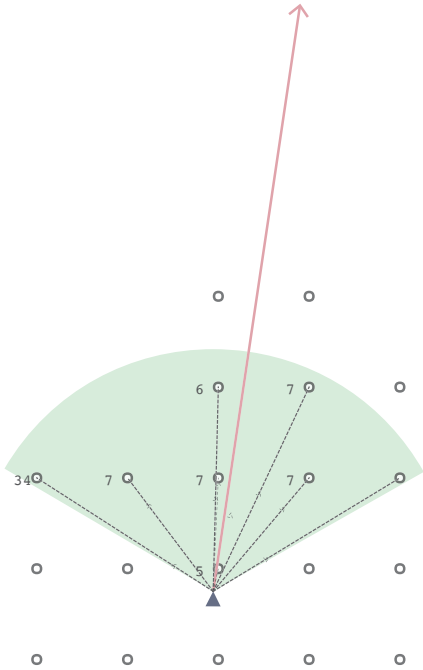
As the agent flows it learns more about the neighbourhood of each point. Thus, the weight for each point is not a static information but at each iteration it gets updated



ENVIRONMENT EXPLORATION 03

The agent might not always be able to identify the exact amount of neighbours for each point of the environment. So if a following agent discovers more information about the neighbourhood of a point the weight gets updated.

FIGURE 06, 07, 08: Weight assignment on the points of the environment. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

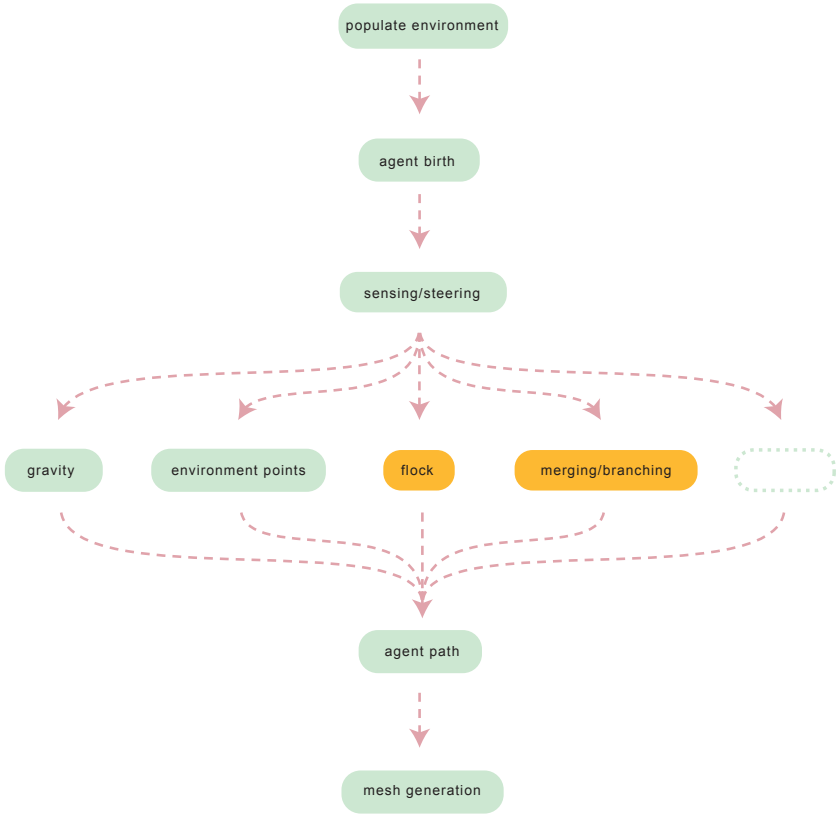


STEERING ACCORDING TO ENVIRONMENT

Practically, weight assignment is scaling the unitized vector between agent's position and environment point scaled by its weight.

After assigning weight to the points the agent explores, the agent adds the vectors of the environment points and finds a resultant.

FIGURE 09: Vector sum for the environment points and definition of the steering force. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)



AGENT-AGENT INTERACTION

Behaviours that define the agent-agent interaction are incorporated in order for the agents to exchange information about the progress of the structure.
At each step agents check their neighbourhood for other agents' activity. More specifically, they examine neighbouring agents positions (densities) and neighbouring agents directions. This implies the use of flocking behaviour to keep the agents informed about the activity of other agents inside their neighbourhood.

FIGURE 10: Pseudocode highlighted. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

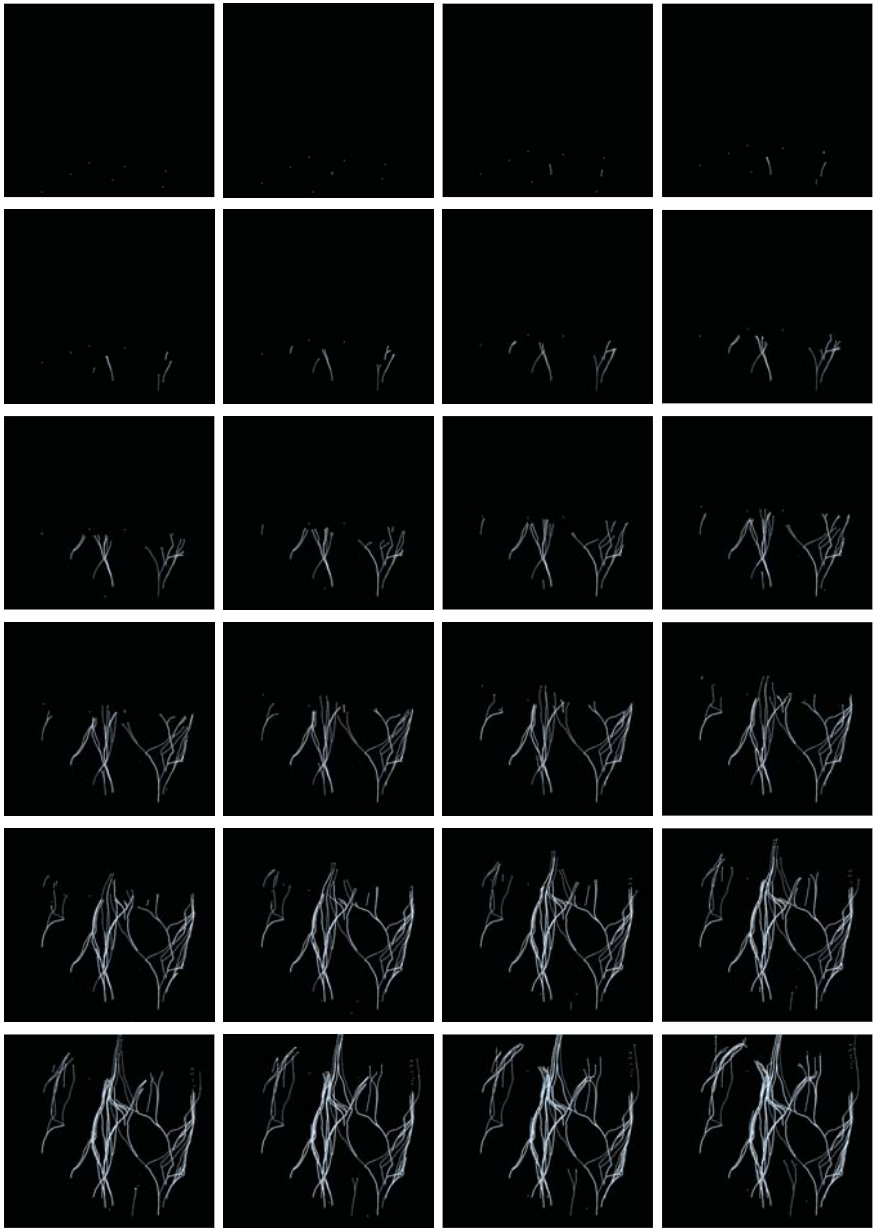
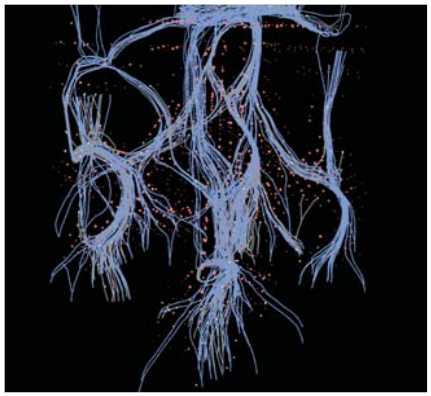


FIGURE 11: Stills from the animation showing all basic behaviours: gravity, target point attraction, flock, merging/branching. Craziness and environment points are disabled. Images were taken every 40 frames from a Processing animation
(*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)



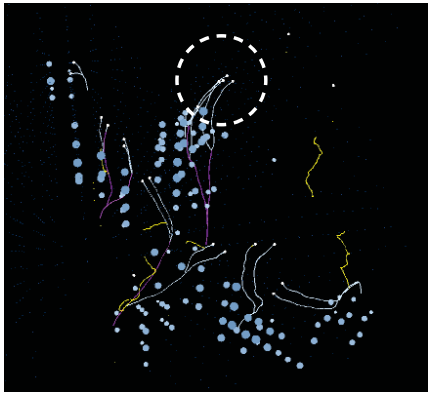
FLOCK

Flock behaviour is responsible for making the total swarm of agents perform a coherent flow avoiding irrational outcomes.
All three simple rules of flocking are adopted:

Cohesion (fig.a) is defined as the reaction of agents to steer towards average position of neighbours.

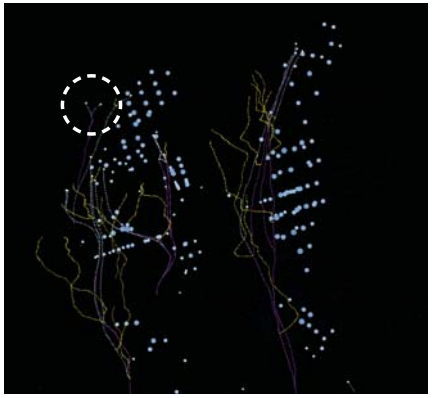
Separation (fig.2) is defined as the reaction of agents to avoid crowding neighbours(short repulsion).

Alignment (fig.3) is defined as the reaction of the agent to steer towards average heading of neighbours.



MERGING

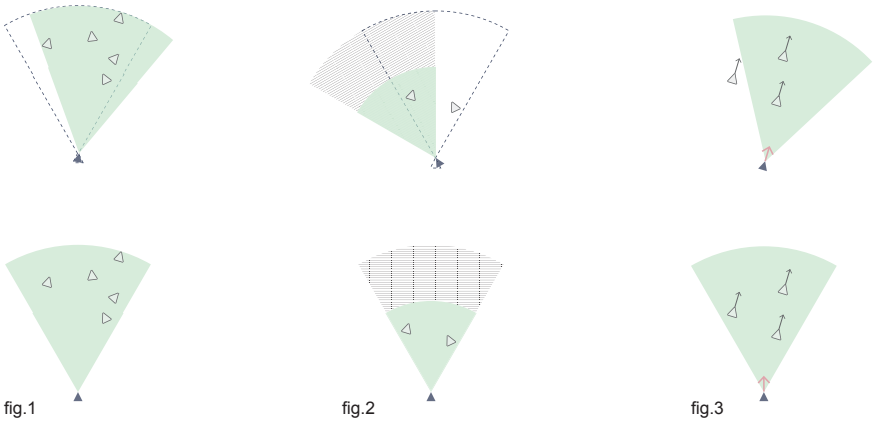
Merging is defined as a behaviour that fuses two agents into one.
This happens in two stages: when two agents come closer than a specific small distance (fig.4), separation is deactivated and they are allowed to come really close (fig.5) and at second stage if they are over a certain age, one of them dies, when their distance gets really small that they overlap (fig.6).

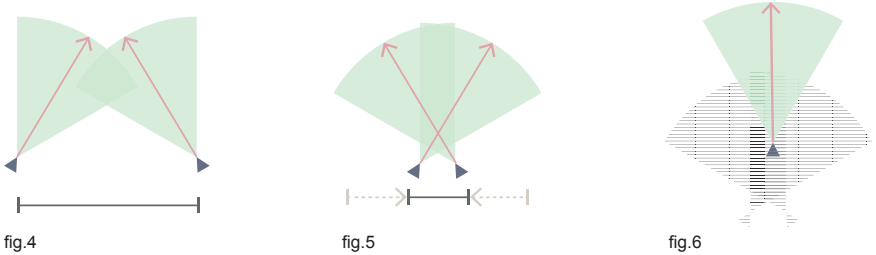


BRANCHING

Branching allows for density control in sparse areas of the environment.
If an agent detects no other agent in the neighbourhood in a distance smaller that the so called branching range (fig.7), it gives birth to another agent by duplicating himself (fig.8), as long as it is over a certain age, thus able to reproduce itself (fig.9).

FIGURE 12, 13, 14: Flocking, Merging, Branching behaviours. Pictures frome Processing animation (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)





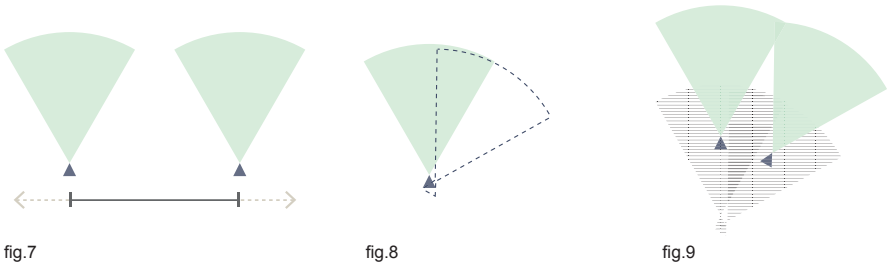


FIGURE 15: Diagrammatic demostrations of flocking, merging, branching behaviours of the agents. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

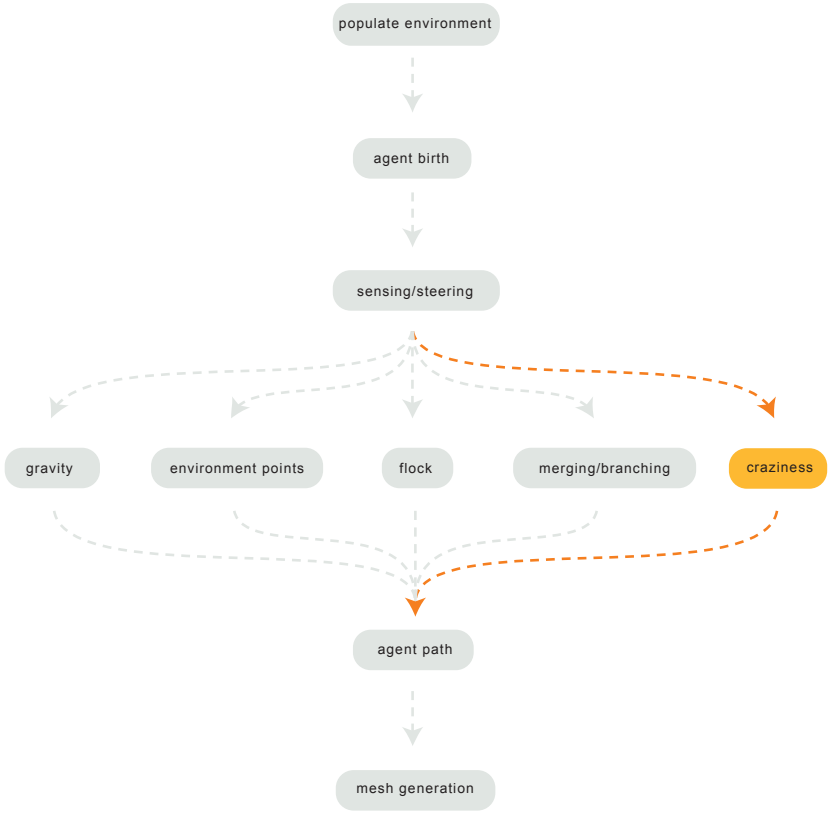


FIGURE 16: Pseudocode highlighted. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

CRAZINESS

Up to this point we had been producing structures that follow strict rules. These rules produce a global behaviour of the structure by controlling the local interactions under the influence of one unitary force. The global unitary force suggests the forces' flow towards a certain direction and it could be gravity. In this way we obtain a rational structure that is produced through summing all the local forces generated by interactions of the agents with the environment and with each other and the global unitary force which is the ultimate goal of their motion.

In this way, we end up with certain areas of high concentration of material and other areas that are mostly hollow. At this point we came to the conclusion that this could be a primary structure. This primary structure only occupies a certain percentage of the total environment, which is mostly about 30% and returns information about 50% of the total environment.

We then got really interested in making the algorithm capable of creating at the same time a soft layer around the primary structure; a secondary structure. In this way we manage to explore up to 90% of the environment, return information about it and create a soft layer around the primary structure.

In this behaviour the agent enters a state where the global unitary force, the flocking behaviours and the merging branching behaviours are not affecting it anymore. Regarding the environment points, the agent flattens their values and all the points then have the same values, changing the way the agent perceives its environment. This might suggest a random walk but this is not the case. When the agent goes into the 'crazy' state, it already has a certain velocity vector, and thus inertia. so it will never go backwards. It will float more freely towards its velocity vector and die after a certain age. or when it reaches the void over the environment.

This layer is created by the very same agents that create the primary structure. In every iteration every agent has a very small probability of shifting to the 'crazy' state, and soon it will die.

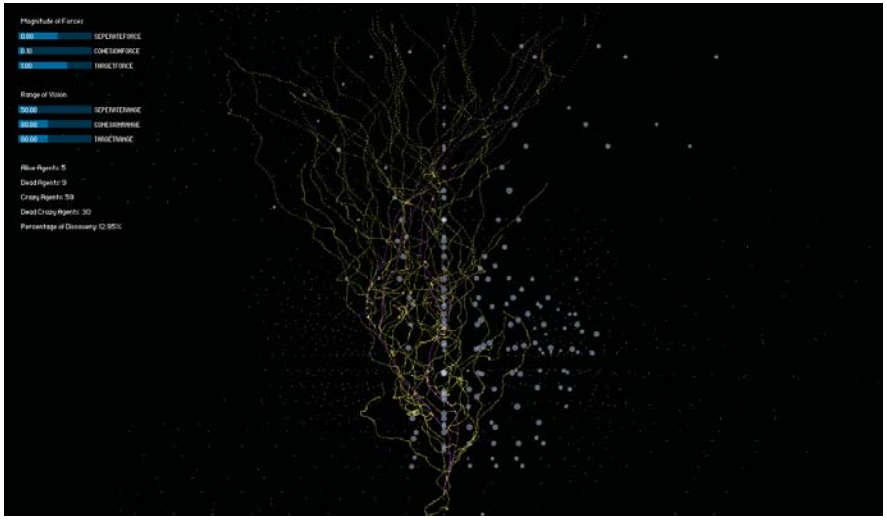
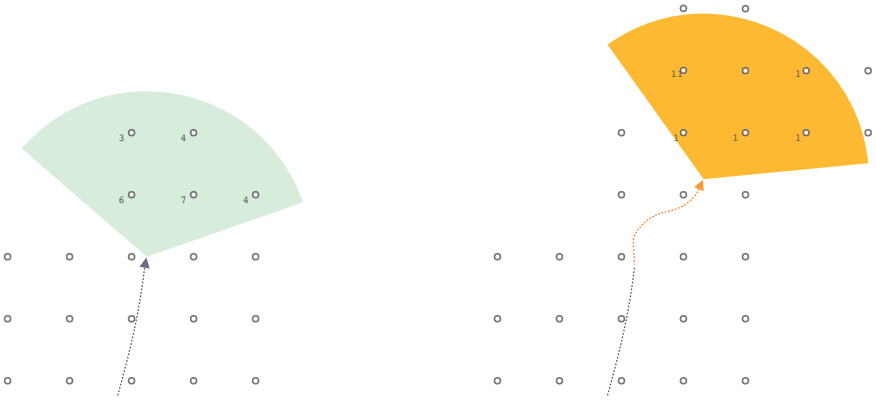


FIGURE 17: Crazy behaviour highlighted with yellow. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)



The density of the secondary structure is defined by a probability percentage that is predefined from the beginning. Practically, this means that every agent at each iteration rolls a dice and if the event is inside a certain probability space the agent shifts into the soft layer state called ‘crazy’.

In this state the agent becomes performs a freer motion which mostly relies on two factors: its inertia and the existance of environment points surrounding it.

That means that as long as there are points around it it moves towards an average direction trying to stay inside its environment, starting with its last velocity vector direction (inertia).

This is acheived by eliminating the inflluence of a global unitary force and the rest of the influence of other agents. As the environment point weight is the same for all of the points anymore, the ‘crazy’ agent floats inside this environment towards an average direction until it reaches a certain age and dies.

FIGURE 18: Diagrammatic demostration of how the agent perceives its environment weight in each case: normal(left), crazy(right). (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

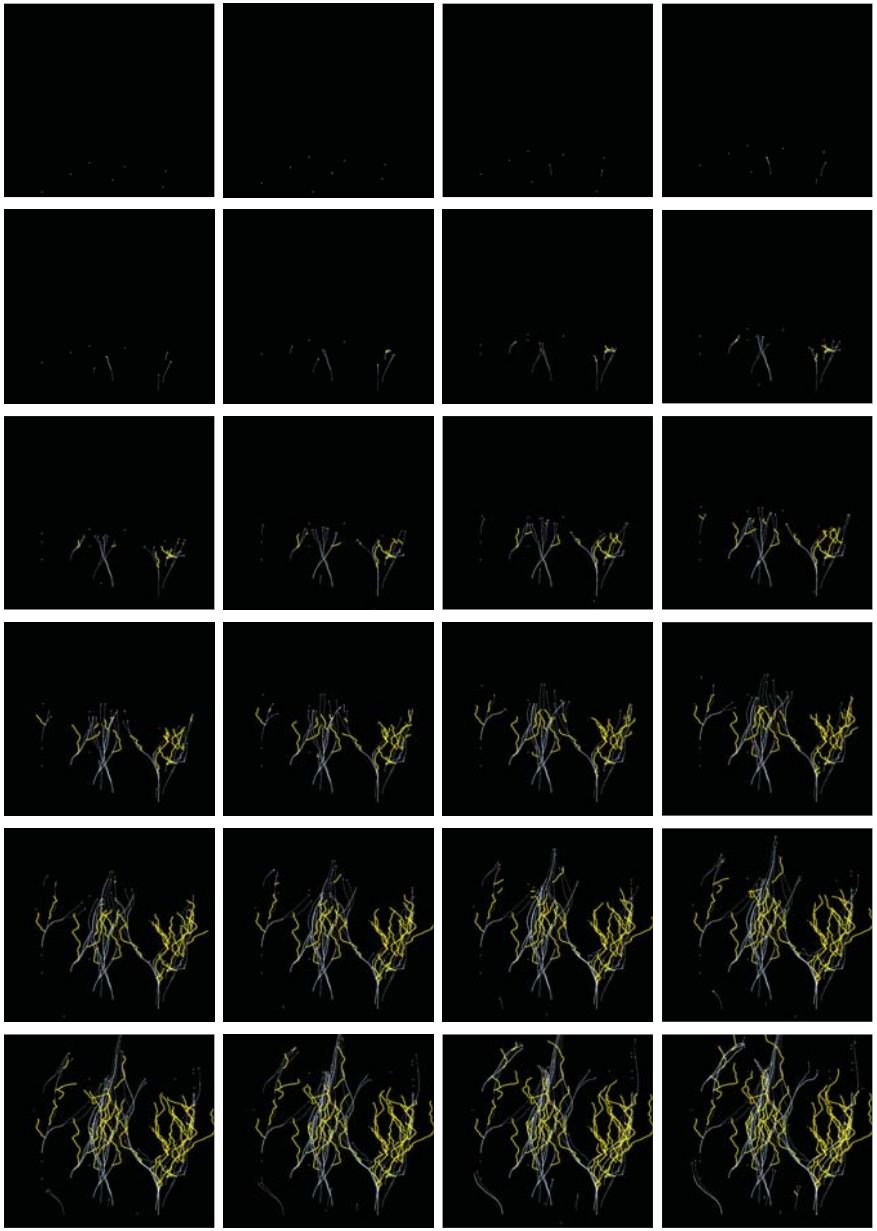
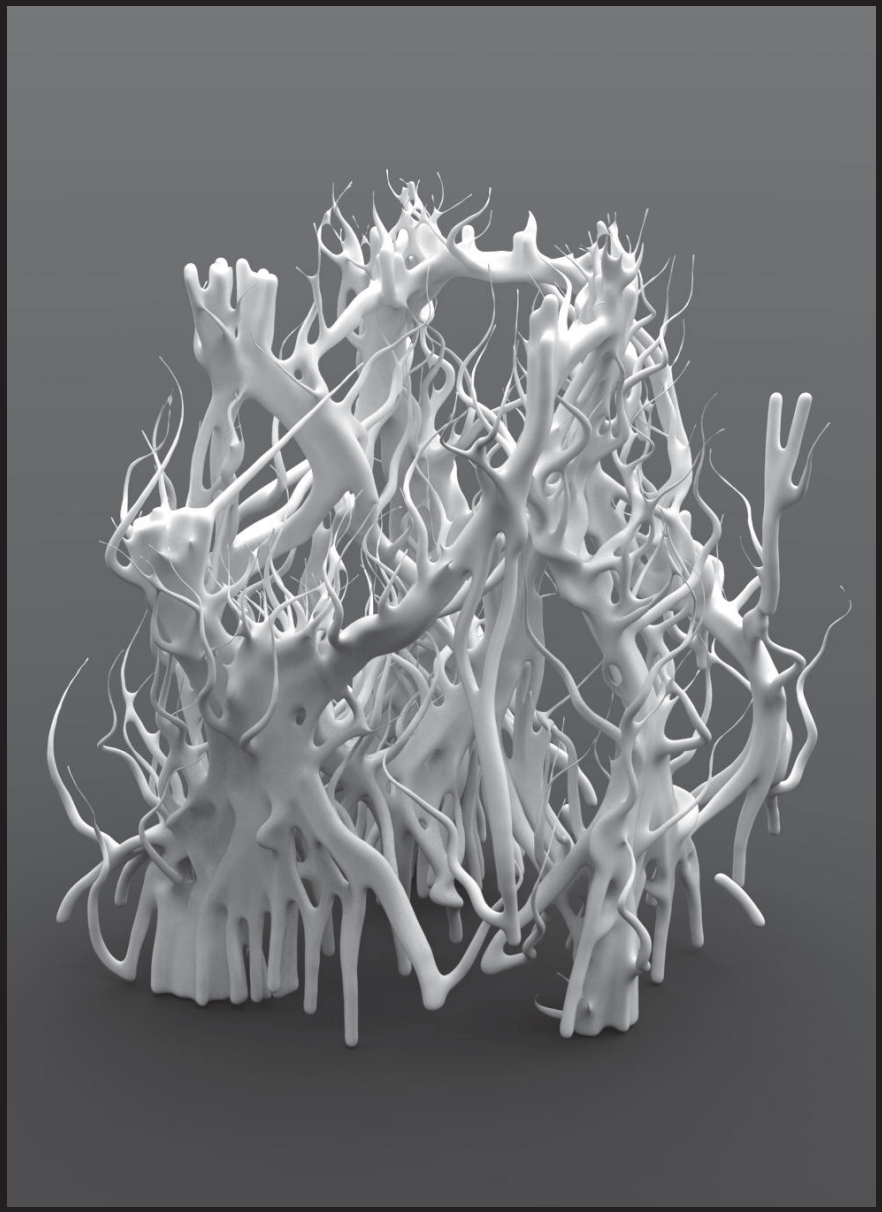


FIGURE 19: Crazy behaviour highlighted with yellow. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)



CONCLUSION

In this project we investigated how computational systems can become performative under architectural criteria and how they can be manipulated to express architectural qualities.

We started exploring computational systems before understanding the importance of meeting certain performance criteria. We managed to grasp the notion of emergence and we achieved to work with it as purely geometrical expression.

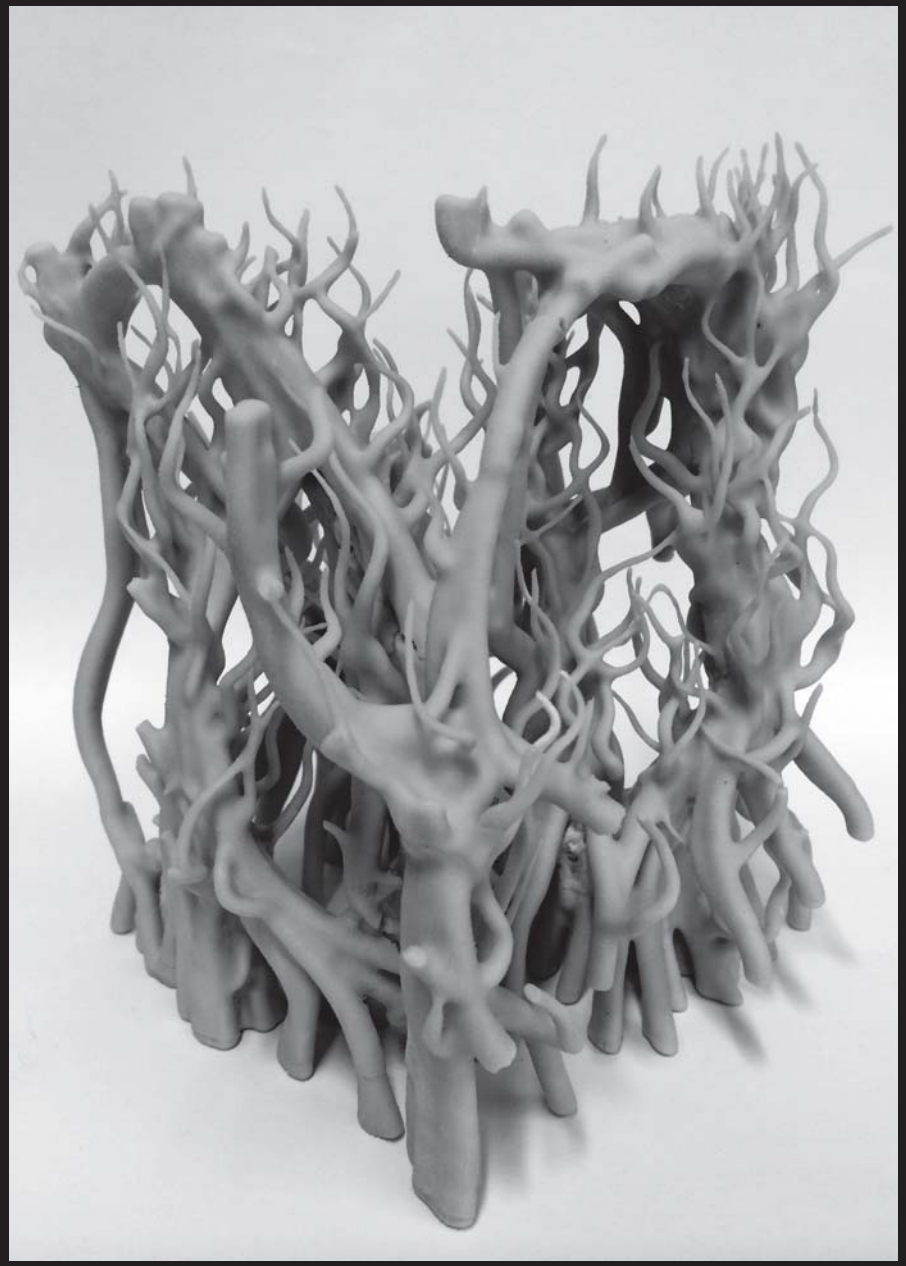
We got really interested in manipulating emergence by injecting more performative criteria in the process or changing the nature of certain criteria already used to act more performative.

What we eventually were interested in developing was an apparatus capable of recognizing and evaluating its environment real-time with optimal material distribution as its ultimate goal.

We choose Agent Based Systems considering it as an efficient and economic computational system acting at a local level to establish a global outcome.

This apparatus receives a substrate as input, it injects agents in it, and the agents try to find their way to acheive optimal paths for material efficiency. The trails then are meshed establishing a structure.

As a first attempt this system minimizes the use of material to achieve resistance to self load. As future investigation, we would seriously consider implying more structural criteria that create a generative system that can have a more integrative behaviour towards several types of loads under the scope of material efficiency.



Code by Leonard Balas, Jingcheng Chen, Nikolaos Xenos

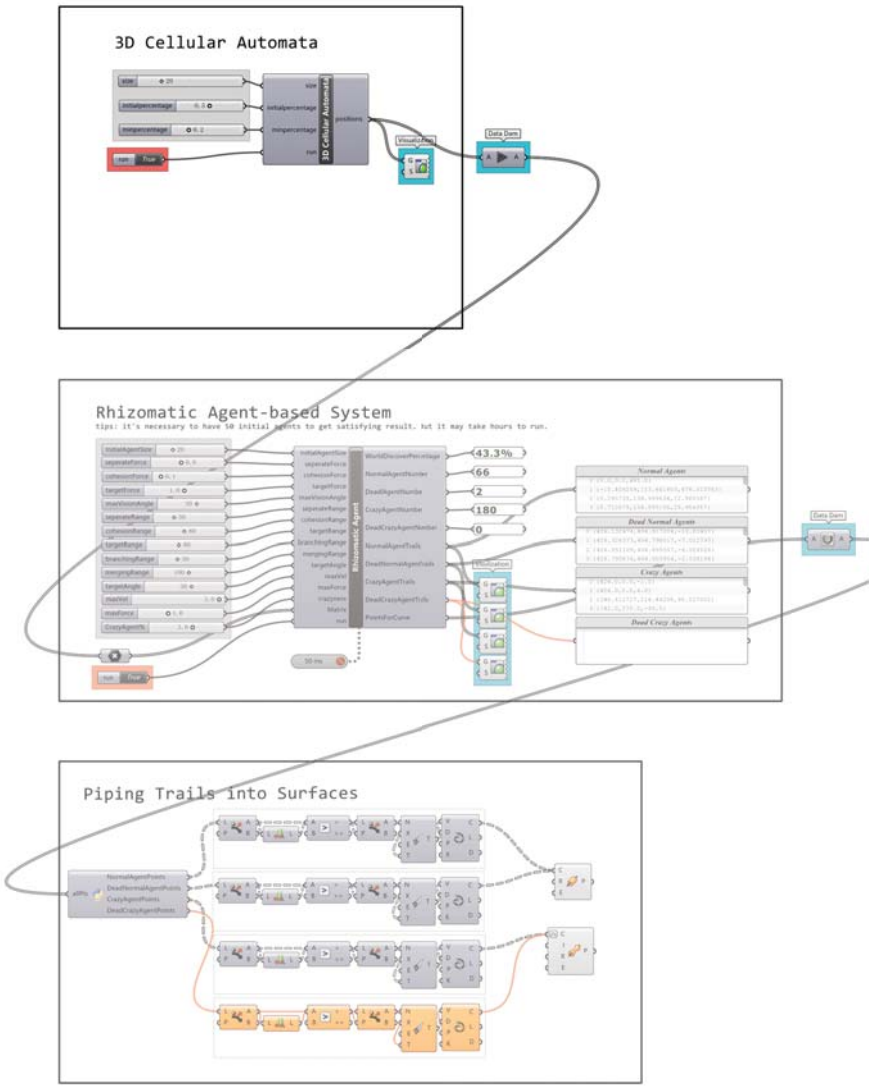


FIGURE 21: Grasshopper definition highlighted. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

```
import random
import rhinoscriptsyntax as rs
import scriptcontext as sc
import time
import Rhino.Geometry as rg

# parameters
world_x = size
world_y = size
world_z = size
minworldsize = world_x*world_y*world_z*minpercentage
flatternworld = []
entities = []
PtListSteps = []
# create cell class
class cell:
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
        self.pos = [x,y,z]
        self.currentstate = None
        self.nextstate = 0
        self.neighbours = None
        self.weight = 0
    # count neightbours in the space
    def countneighbours(self):
        count = [0] * 26
        # top layer
        if (self.z + 1 < world_z):
            if (self.x - 1 >= 0) and (self.y + 1 < world_y):
                count[0] = world[self.x - 1][self.y + 1][self.z + 1].currentstate
            if (self.y + 1 < world_y):
                count[1] = world[self.x][self.y + 1][self.z + 1].currentstate
            if (self.x + 1 < world_x) and (self.y + 1 < world_y):
                count[2] = world[self.x + 1][self.y + 1][self.z + 1].currentstate
            if (self.x - 1 >= 0):
                count[3] = world[self.x - 1][self.y][self.z + 1].currentstate
            count[4] = world[self.x][self.y][self.z + 1].currentstate
            if (self.x + 1 < world_x):
                count[5] = world[self.x + 1][self.y][self.z + 1].currentstate
            if (self.x - 1 >= 0) and (self.y - 1 >= 0):
                count[6] = world[self.x - 1][self.y - 1][self.z + 1].currentstate
            if (self.y - 1 >= 0):
                count[7] = world[self.x ][self.y - 1][self.z + 1].currentstate
            if (self.x + 1 < world_x) and (self.y - 1 >= 0):
                count[8] = world[self.x + 1][self.y - 1][self.z + 1].currentstate
        # middle layer
        if (self.x - 1 >= 0) and (self.y + 1 < world_y):
            count[9] = world[self.x - 1][self.y + 1][self.z].currentstate
        if (self.y + 1 < world_y):
            count[10] = world[self.x][self.y + 1][self.z].currentstate
        if (self.x + 1 < world_x) and (self.y + 1 < world_y):
            count[11] = world[self.x + 1][self.y + 1][self.z ].currentstate
        if (self.x - 1 >= 0):
            count[12] = world[self.x - 1][self.y][self.z].currentstate
        if (self.x + 1 < world_x):
            count[13] = world[self.x + 1][self.y][self.z].currentstate
        if (self.x - 1 >= 0) and (self.y - 1 >= 0):
            count[14] = world[self.x - 1][self.y - 1][self.z].currentstate
```

```
    if (self.y - 1 >= 0):
        count[15] = world[self.x ][self.y - 1][self.z].currentstate
    if (self.x + 1 < world_x) and (self.y - 1 >= 0):
        count[16] = world[self.x + 1][self.y - 1][self.z].currentstate
    # bottom layer
    if self.z - 1 >= 0:
        if (self.x - 1 >= 0) and (self.y + 1 < world_y):
            count[17] = world[self.x - 1][self.y + 1][self.z - 1].currentstate
        if (self.y + 1 < world_y):
            count[18] = world[self.x][self.y + 1][self.z - 1].currentstate
        if (self.x + 1 < world_x) and (self.y + 1 < world_y):
            count[19] = world[self.x + 1][self.y + 1][self.z - 1].currentstate
        if (self.x - 1 >= 0):
            count[20] = world[self.x - 1][self.y][self.z - 1].currentstate
        count[21] = world[self.x][self.y][self.z - 1].currentstate
        if (self.x + 1 < world_x):
            count[22] = world[self.x + 1][self.y][self.z - 1].currentstate
        if (self.x - 1 >= 0) and (self.y - 1 >= 0):
            count[23] = world[self.x - 1][self.y - 1][self.z - 1].currentstate
        if (self.y - 1 >= 0):
            count[24] = world[self.x ][self.y - 1][self.z - 1].currentstate
        if (self.x + 1 < world_x) and (self.y - 1 >= 0):
            count[25] = world[self.x + 1][self.y - 1][self.z - 1].currentstate

    self.neighbours = sum(count)
# eof
def calweight(self):
    if self.neighbours in range(13,17) :
        self.weight = 4
    if self.neighbours in range(17,20) :
        self.weight = 3
    if self.neighbours in range(20,23) :
        self.weight = 2
    if self.neighbours in range(23,27) :
        self.weight = 1
#eof
# decide state of cells for next phase
def decideLife(self):
    if self.currentstate == 0 :
        if self.neighbours in [13,14,17,18,19] :
            self.nextstate = 1
        else:
            self.nextstate = 0
    if self.currentstate == 1 :
        if self.neighbours in range(13,27) :
            self.nextstate = 1
        else:
            self.nextstate =
# check the current state
def checkstate(self):
    self.countneighbours()
    self.decideLife()
# update the current state
def updatestate(self):
    self.calweight()
    self.currentstate = self.nextstate
```

```
# create a world with random lives
def creatworld(x,y,z,percentageoflives):
    global world
    world = [[[0 for q in xrange(x)] for w in xrange(y)] for e in xrange(z)]
    for i in range(x):
        for j in range(y):
            for k in range(z):
                r = random.random()
                c = cell(i,j,k)
                if r < percentageoflives:
                    c.currentstate = 1
                else:
                    c.currentstate = 0
                world[i][j][k] = c
                flatternworld.append(c)
#check the survival size of the world
def worldSize():
    Size = 0
    for onecell in flatternworld:
        Size = Size + onecell.currentstate
    return Size
# check the world
def checkworld():
    for onecell in flatternworld:
        onecell.checkstate()
# clean the world
def cleanworld():
    rs.DeleteObjects(entities)
    del entities[:]
# update the world
def updateworld():
    for onecell in flatternworld:
        onecell.updatestate()
def recordworld():
    pts = []
    weights = []
    neighbours = []
    for onecell in flatternworld:
        if onecell.currentstate == 1:
            pts.append(rg.Point3d(onecell.pos[0],onecell.pos[1],onecell.pos[2]))
            weights.append(onecell.weight)
            neighbours.append(onecell.neighbours)
    return pts,weights,neighbours


creatworld(world_x,world_y,world_z,initialpercentage)

while (worldSize() > minworldsize) and run:
    sc.escape_test()

    checkworld()
    cleanworld()
    updateworld()

all = recordworld()

positions = all[0]
weights = all[1]
neighbours = all[2]

positions = [i*int(600/size) for i in positions]
```

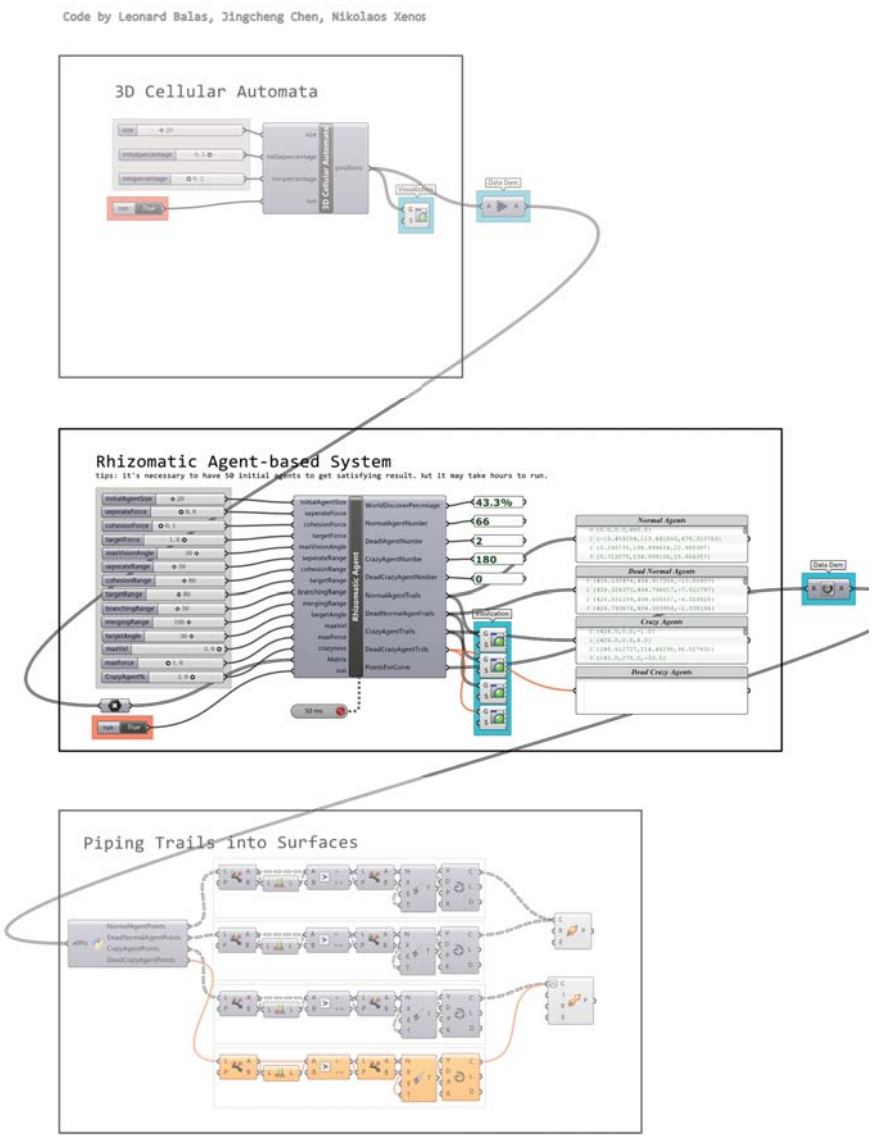


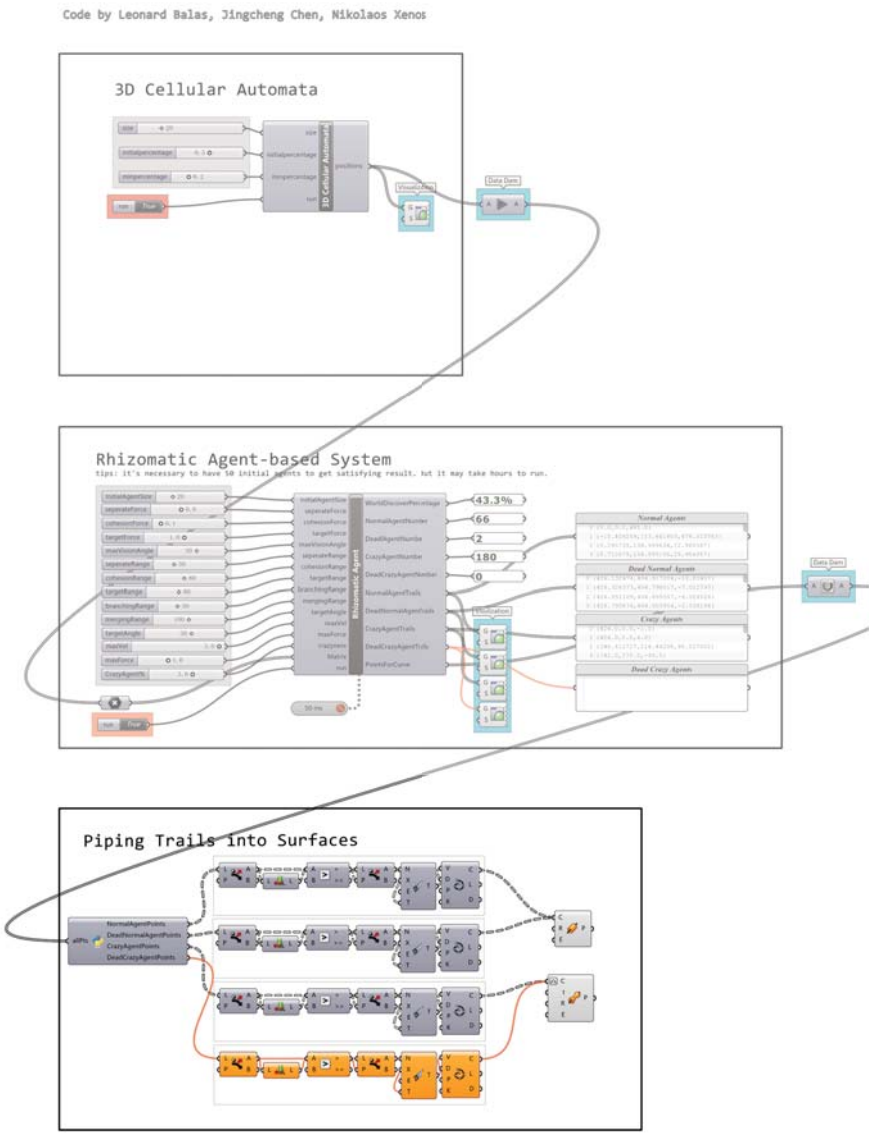
FIGURE 22: Grasshopper definition highlighted. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

```
import rhinoscriptsyntax as rs
import Rhino.Geometry as rg
import random as rnd
import math

widthX,widthY,widthZ = 570,540,570
step = 1000
# ====================Start of Define Classes==================
# Define space point class
class Point:
    def __init__(self, pos):
        self.pos = pos
        self.neighbours = 0
# Define dead crazy agent
class DeadCrazyAgent:
    def __init__(self, l,trails):
        self.l = l
        self.trails = trails
    def render(self):
        pt = rs.AddPoints(self.trails)
        return pt
# Define crazy agent
class CrazyAgent:
    def __init__(self, l,v,a):
        self.l = l
        self.v = rg.Point3d(0,0,0.5)
        self.a = a
        self.trails = []
        self.lifespan = 200
    def update(self):
        self.targetpoints()
        self.v = rs.VectorAdd(self.v,self.a)
        self.v = limit(self.v,maxVel)
        self.l = rs.VectorAdd(self.l,self.v)
        self.a = rg.Point3d(0,0,0)
        self.lifespan =  self.lifespan - 1
        if self.lifespan < 0:
            self.dead()
    def border(self):
        if self.l[0] > widthX : self.l[0] = 0
        if self.l[0] < 0 : self.l[0] = widthX
        if self.l[1] > widthY : self.l[1] = 0
        if self.l[1] < 0 : self.l[1] = widthY
        if self.l[2] > widthZ : self.l[2] = 0
        if self.l[2] < 0 : self.l[2] = widthZ
    def trail(self):
        self.trails.append(self.l)
    def dead(self):
        a = DeadCrazyAgent(self.l,self.trails)
        deadcrazyagents.append(a)
        crazyagents.remove(self)
    def targetpoints(self):
        sum = rg.Point3d(0,0,0)
        count = 0
        localPoints = []
        for i in points:
            celdist = rs.Distance(i.pos,self.l)
            if celdist < targetRange:
                localPoints.append(i)
                count += 1
```

```
        if count > 0:
            self.trail()
            calNeighbours(localPoints)
            self.a = rg.Point3d(rnd.uniform(-0.5,0.5),rnd.uniform(-0.5,0.5),rnd.uni-
form(0,0.1))

    def render(self):
        pt = rs.AddPoints(self.trails)
        return pt
# Define dead agent
class DeadAgent:
    def __init__(self, l,trails):
        self.l = l
        self.trails = trails
    def render(self):
        pt = rs.AddPoints(self.trails)
        return pt
# Define normal flow agent
class NormalAgent:
    def __init__(self, l):
        self.l = l
        self.v = rg.Point3d(0,0,0.01)
        self.g = rg.Point3d(0,0,3)
        self.a = rg.Point3d(0,0,0)
        self.merge = False
        self.branch = True
        self.step = 0
        self.trails = []
        self.age = len(self.trails)
        self.flock = False
    def update(self):
        if self.flock == True:
            self.separation()
            self.trail()
        self.branching()
        self.cohesion()
        self.targetpoints()
        self.merging()
        self.floordeath()
        self.age = len(self.trails)

        self.a = rs.VectorAdd(self.a,self.g)
        self.v = rs.VectorAdd(self.v,self.a)
        self.v = limit(self.v,maxVel)
        self.l = rs.VectorAdd(self.l,self.v)
        self.a = rg.Point3d(0,0,0)
    def dead(self):
        a = DeadAgent(self.l,self.trails)
        deadagents.append(a)
        agents.remove(self)
        if random(10) < 3 : self.nextPopulation()
    def nextPopulation(self):
        n = rg.Point3d(self.l[0],self.l[1],-100)
        flow2 = NormalAgent(n)
        agents.append(flow2)
    def floordeath(self):
        if self.l[2] >= widthZ: self.dead()
    def cohesion(self):
        sum = rg.Point3d(0,0,0)
        count = 0
        for i in agents:
            distance = rs.Distance(i.l,self.l)
            if distance < cohesionRange and distance > 0 :
                vision = rs.VectorAngle(i.v,self.v)
                if vision <= maxVisionAngle:
                    sum = rs.VectorAdd(sum,i.l)
                    count += 1
        if count>0:
            sum = rs.VectorScale(sum,1/float(count))
            sum = rs.VectorSubtract(sum,self.l)
            sum = limit(sum,maxForce)
            sum = rs.VectorScale(sum,cohesionForce)
            self.a = rs.VectorAdd(self.a,sum)
    def separation(self):
        if self.merge == False or self.age < 100:
            sum = rg.Point3d(0,0,0)
            count = 0
            for i in agents:
                distance = rs.Distance(i.l,self.l)
                if distance < seperateRange and distance > 0 :
                    vision = rs.VectorAngle(i.v,self.v)
                    if vision <= maxVisionAngle:
                        diff = rs.VectorSubtract(self.l,i.l)
                        diff = rs.VectorScale(diff,10/distance)
                        sum = rs.VectorAdd(sum,diff)
                        count += 1
            if count>0:
                sum = rs.VectorScale(sum,1/float(count))
                sum = limit(sum,maxForce)
                sum = rs.VectorScale(sum,seperateForce)
                self.a = rs.VectorAdd(self.a,sum)
    def targetpoints(self):
        sum = rg.Point3d(0,0,0)
        count = 0
        localPoints = []
        for i in points:
            celdist = rs.Distance(i.pos,self.l)
            if celdist < targetRange:
                targetvec = rs.VectorSubtract(i.pos,self.l)
                localPoints.append(i)
                count += 1
        if count > 0:
            calNeighbours(localPoints)
            for p in localPoints:
                targetvec = rs.VectorSubtract(p.pos,self.l)
                targetang = rs.VectorAngle(targetvec,self.v)
                if targetang <= targetAngle:
                    weight = self.calWeight(p)
                    targetvec = rs.VectorSubtract(p.pos,self.l)
                    targetvec = rs.VectorScale(targetvec,weight)
                    sum = rs.VectorAdd(sum,targetvec)
                    self.flock = True
                    sum = rs.VectorScale(sum,1/float(count))
                    sum = limit(sum,maxForce)
                    sum = rs.VectorScale(sum,targetForce)
                    self.a = rs.VectorAdd(self.a,sum)
        else:
            self.flock = False
```

```
    def calWeight(self,p):
        if p.neighbours in range(0,5): return 5
        if p.neighbours in range(5,10): return 4
        if p.neighbours in range(10,15): return 3
        if p.neighbours in range(15,20): return 2
        if p.neighbours in range(20,26): return 1
    def givebirth(self):
        flow = NormalAgent(rs.VectorAdd(self.l,rg.Point3d(0.5,0.5,0.5)))
        a = rnd.uniform(0,math.pi)
        flow.v = rg.Point3d(rnd.uniform(-2,2),rnd.uniform(-2,2),2)
        flow.merge = False
        agents.append(flow)
    def branching(self):
        counter = 0
        for i in agents:
            distance = rs.Distance(i.l,self.l)
            if distance > 0 and distance < branchingRange: counter += 1
        if counter == 0 and self.age > 25:
            self.givebirth()
    def merging(self):
        count = 0
        count2 = 0
        for i in agents:
            distance = rs.Distance(i.l,self.l)
            if distance < mergingRange and distance > 0 and self.age > 100: count += 1
            if distance <10 and self.age > 100 and (self.l[2] < i.l[2]): count2 +=1
        if count > 1:
            self.merge = True
        if count2 > 0:
            self.dead()
            self.givebirth()
    def trail(self):
        self.trails.append(self.l)
    def render(self):
        pt = rs.AddPoints(self.trails)
        return pt

# ====================End of Define Classes==================

# limit the max vector
def limit(vec,maxlen):
    l = rs.VectorLength(vec)
    if l > maxlen:
        vec = rs.VectorUnitize(vec)
        vec = rs.VectorScale(vec,maxlen)
    return vec
# calculate the discovery percentage
def discoverpercentage():
    y = 0
    for i in points:
        if i.neighbours > 0: y += 1
    per = y/len(points)
    return round(per,4)
# populate agents at beginning
def regularpopulate():
    agentlen = int(math.sqrt(InitialAgentSize))
    for i in range(agentlen):
        for j in range(agentlen):
            x = int(widthX/agentlen) * i
            y = int(widthY/agentlen) * j
            l = rg.Point3d(x, y, -100)
            flow1 = NormalAgent(l)
            agents.append(flow1)
# calculate the neighbours of points
def calNeighbours(localPoints):
    global step
    for i in localPoints:
        count = 0
        for j in localPoints:
            if i != j:
                dis = rs.Distance(i.pos,j.pos)
                step = min(step,dis)
                if  dis < step*2: count += 1
    if i.neighbours < count:
        i.neighbours = count
# change normal agents to crazy agents in a certain percentage
def SomeoneGoesCrazy():
    for i in agents:
        if rnd.uniform(0,100) < crazyness:
            crazy = CrazyAgent(i.l,i.v,i.a)
            crazyagents.append(crazy)
def process():
    normal = []
    deadnormal = []
    crazy = []
    deadcrazy = []
    allInfo = [[],[],[],[]]
    if len(agents) > 0:
        for i in agents:
            normal.append(i.l)
            if len(i.trails) > 1:
                allInfo[0].append(i.trails)
                normal.extend(i.trails)
    if len(deadagents) > 0:
        for i in deadagents:
            if len(i.trails) > 1:
                allInfo[1].append(i.trails)
                deadnormal.extend(i.trails)
    if len(crazyagents) > 0:
        for i in crazyagents:
            crazy.append(i.l)
            if len(i.trails) > 1:
                allInfo[2].append(i.trails)
                crazy.extend(i.trails)
    if len(deadcrazyagents) > 0:
        for i in deadcrazyagents:
            if len(i.trails) > 1:
                allInfo[3].append(i.trails)
                deadcrazy.extend(i.trails)
    return normal,deadnormal,crazy,deadcrazy,allInfo
if not run:
    global points,agents, deadagents, crazyagents, deadcrazyagents, NormalAgentList,
    DeadAgentList, CrazyAgentList, DeadCrazyAgentList
    points = []
    agents, deadagents, crazyagents, deadcrazyagents = [], [], [], []
    NormalAgentList, DeadAgentList, CrazyAgentList, DeadCrazyAgentList = [], [], [], []
    for i in range(len(Matrix)):
        p = Point(Matrix[i])
        points.append(p)
```

Code by Leonard Balas, Jingcheng Chen, Nikolaos Xenos




```
    regularpopulate()
else:
    for i in agents:
        i.update()
    for m in crazyagents:
        m.update()
    SomeoneGoesCrazy()

show = process()
NormalAgentTrails = show[0]
DeadNormalAgentTrails = show [1]
CrazyAgentTrails = show[2]
DeadCrazyAgentTrails = show[3]
PointsForCurve = show[4]
WorldDiscoverPercentage = str(discoverpercentage()*100) + “%”
NormalAgentNumber = len(agents)
DeadlAgentNumber = len(deadagents)
CrazyAgentNumber = len(crazyagents)
DeadCrazyAgentNumber = len(deadcrazyagents)
```

```
import rhinoscriptsyntax as rs

import Rhino.Geometry as rg
from clr import AddReference as addr
addr(“Grasshopper”)

from System import Object
from Grasshopper import DataTree
from Grasshopper.Kernel.Data import GH_Path


def raggedListToDataTree(raggedList):
    rl = raggedList
    result = DataTree[object]()
    for i in range(len(rl)):
        temp = []
        for j in range(len(rl[i])):
            temp.append(rl[i][j])
        #print i, “ - “,temp
        path = GH_Path(i)
        result.AddRange(temp, path)
    return result

def dataTreeToList(aTree):
    theList = []
    for i in range(aTree.BranchCount ):
        thisListPart = []
        thisBranch = aTree.Branch(i)
        for j in range(len(thisBranch)):
            thisListPart.append( thisBranch[j] )
        theList.append(thisListPart)
    return theList

NormalAgentPoints = raggedListToDataTree(allPts[0])
DeadNormalAgentPoints = raggedListToDataTree(allPts[1])
CrazyAgentPoints = raggedListToDataTree(allPts[2])
DeadCrazyAgentPoints = raggedListToDataTree(allPts[3])
```

FIGURE 23: Grasshopper definition highlighted. (*Source:* Léonard Balas, Jingcheng Chen, Nikolaos Xenos)

References

Fourie, P. and Groenwold, A. (2002). The particle swarm optimization algorithm in size and shape optimization. Structural and Multidisciplinary Optimization, 23(4), pp.259-267.

Hensel, M., Menges, A. and Weinstock, M. (2010). Emergent technologies and design. Oxon [England]: Routledge.

Hodge, A., Berta, G., Doussan, C., Merchan, F. and Crespi, M. (2009). Plant root growth, architecture and function. Plant Soil, 321(1-2), pp.153-187.

Leach, N. (2009). Digital cities. Chichester: John Wiley & Sons.

Menges, A. (2012). Material computation. Hoboken, N.J.: Wiley.

Particle Swarm Optimization: Tutorial. (2016). [online] Swarmintelligence.org. Available at: http://www.swarmintelligence.org/tutorials.php [Accessed 4 Mar. 2016].

Shiffman, D. (2012). The nature of the code. [S.l.]: D. Shiffman.

Smith, S. and De Smet, I. (2012). Root system architecture: insights from Arabidopsis and cereal crops. Philosophical Transactions of the Royal Society B: Biological Sciences, 367(1595), pp.1441-1452.