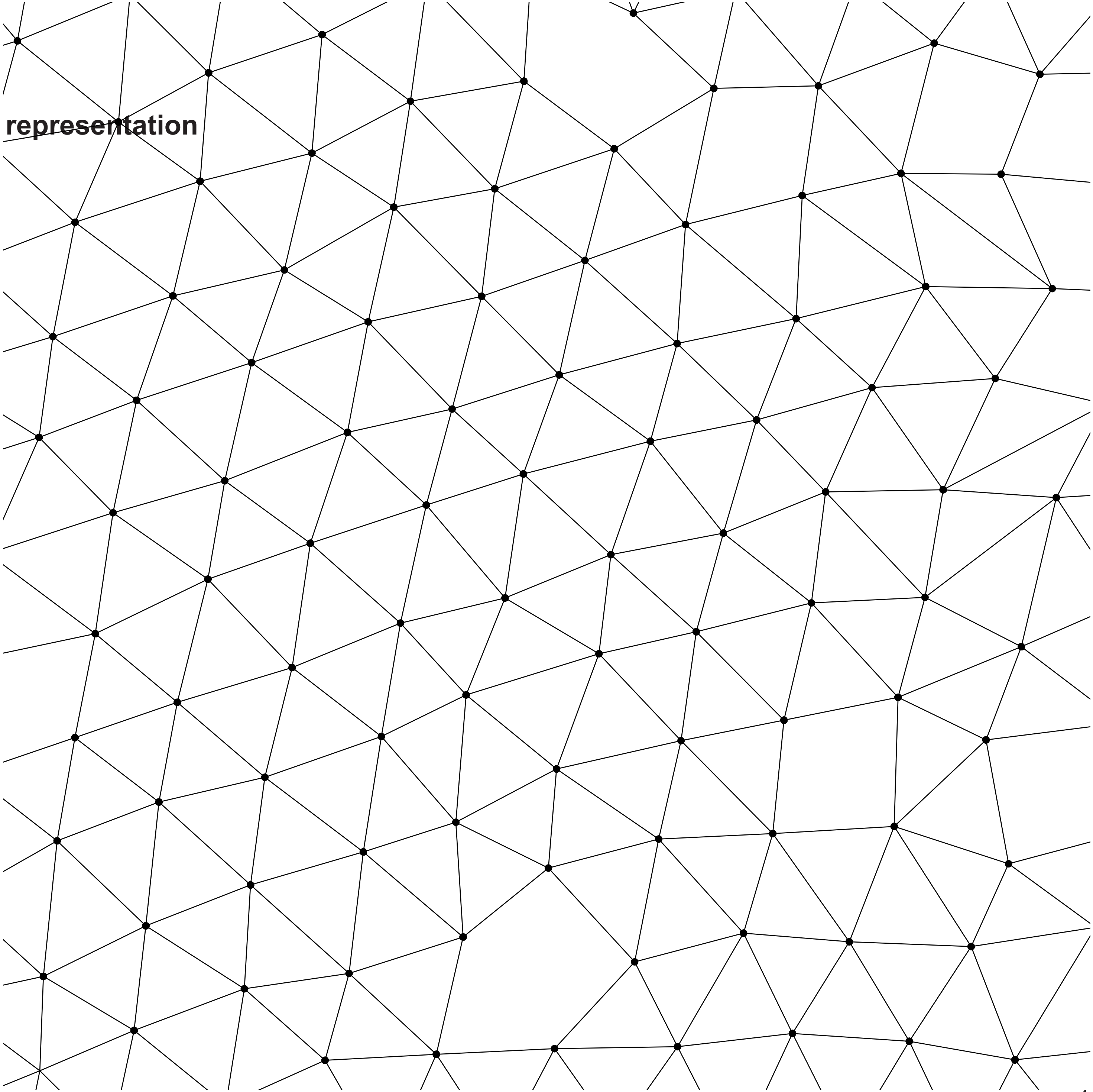
Chapter 1
Rethinking representation



The introduction sets out the initial position of text as design representation. Fundamentally the proposition is that Chomsky's dictum – that finite syntax and lexicon can nevertheless generate an infinite number of useful (well formed) structures – can be applied to artificial languages, and that texts can be written in those languages to generate architectural objects, taken to mean 'well formed' configurations of space and form. This is the generative algorithm and the idea is that a generative algorithm is a description of the object just as much as the measurement and analysis of the object, the illustration of the object and the fact of its embodiment in the world.

The position here is that the text we are looking at, being an artificial language, usually depends for its embodiment on some hardware – the engineering product of the turing machine – and this hardware affords some species of representation, from simple graphics all the way up to programmable hardware, 3d printing and immersive virtual worlds. But this aspect is simply an unfolding of the underlying algorithm, which is still the original representation. It would be possible to orchestrate 300 human beings to obey instructions and so run the computations in their bodies (by, in the cases below moving small balls about in a coordinated manner) – it doesn't matter, it's the algorithm that counts.

Some simple texts

As a very first shot, take the example of representing some simple geometric shapes and volumes like the circle the spheroid and other 3d polyhedra, not using geometry, but small programs written in a dialect of LOGO (a venerable AI language defined by Seymour Papert. whose history is elaborated in the next section)

Triangles and Circles

For the 2D case, this can be verified with a simple experiment using a program with a large number of points in 2D space, initially randomly sprinkled over the plane.

Give each point a rule
"search through all the other points and find the nearest one to yourself"
"Then: turn towards this nearest point and back off a little bit"

all the points do this simultaneously.

Of course the problem is that in backing away from your nearest neighbour you may inadvertently come too close to someone else, but that's ok because then you just turn around and back away from them. Remember that everybody is doing this at the same time.

To demonstrate how this works we can teach these rules to a computer using the Netlogo language which provides a mechanism for setting up parallel computations very simply (the points are described using "turtles" –little autonomous computer programs all of whom obey the program set out below)

```
to repel
ask turtles
[
    set closest-turtle   min-one-of other turtles
      [distance  myself]
    set heading towards closest-turtle
    back 1
]
end
```

To understand this piece of code, first notice that the whole thing is wrapped up in the clause:

To repel

Do something

End

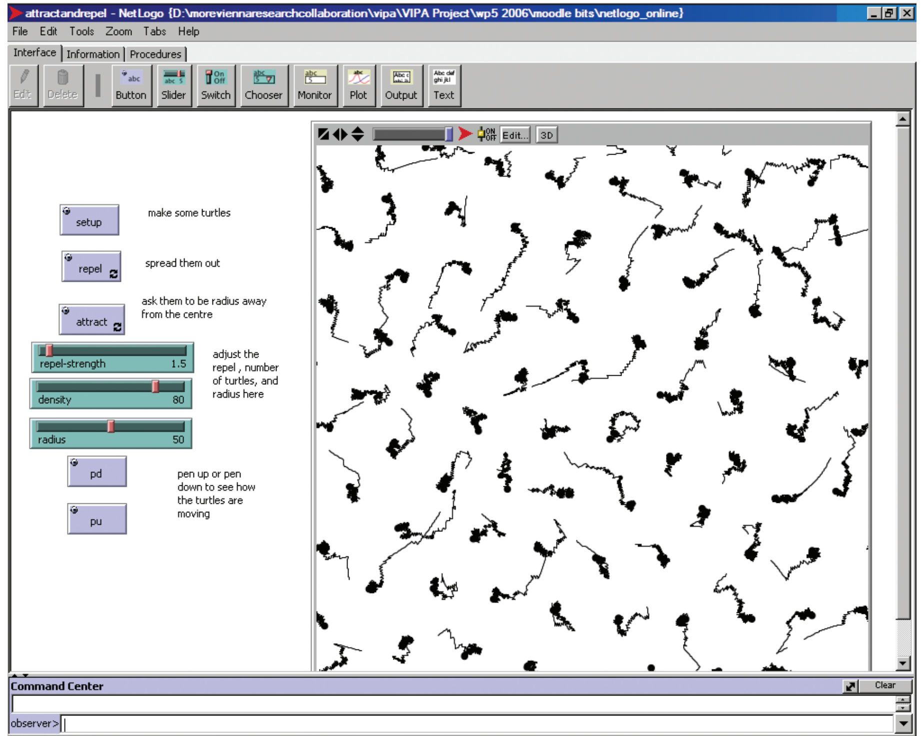
This is because we are defining how *to do something* for the computer so here we are setting out how *to repel.* The stuff between the word TO and the word END is the actual code. Then comes the phrase *ask turtles.* Who, you might ask, is doing this asking? The turtles are the points in space, they are really a lot of tiny abstract computers, and the global overall observer is in this statement sending out a message to all the turtles to run the program enclosed in the square brackets [ ] which is the three sentences:

```
1)  set closest-turtle min-one-of other turtles
    [distance  myself]
2)  set heading towards closest-turtle
3)  back 1
```

The turtles are being told
"Dear turtles, I would like to ask you to look through all the other turtles to find the one whose distance away is at a minimum"
then they must remember which turtle this is by storing its reference in the name '*closest-turtle'*

Now the turtles are told
"Set your heading so that you are pointing towards this 'closest-turtle', and back off one step"

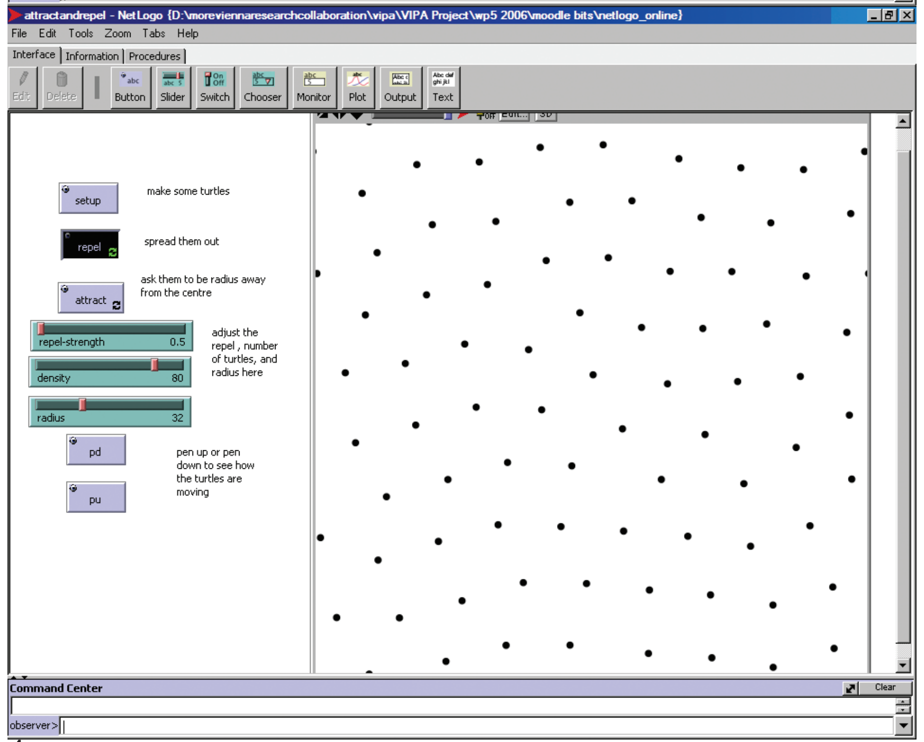
Interestingly we also have to tell the computer to address the 'other' turtles as in the human language description . If we just asked all the turtles this would include myself (the one doing the ASKing and we would get a value of zero and try to walk away from ourselves - not a good idea. This is a good example (the first of many) of how we have to
S.P.E.L.L. I.T. O.U.T. for these supremely pedantic machines.



In the left hand image above, the trails of the turtles are shown moving from the initial random sprinkling to the triangular grid. It takes about 500 steps for the system to settle down, and it can be observed that the turtles quite quickly find a suitable position and then stay there (the trails don't stretch very far, and rarely cross).

These and many other examples if programming in the book are based on NetLogo. This language is a descendent of StarLogo which in turn was a parallel implementation of Logo described in the next chapter, which itself was a development of Lisp (see chapter 3).

A good introduction to this language is Michel Resnic's "Turtles Termites and TrafficJams" (MIT press 1995) which came out with Starlogo



The turtles settle down to triangular least effort configuration

Opposite page a version of the program running with links shown in the tessellation and radial rings modes. None of these patterns lasts for long, like all dynamic systems the moment can be captured, but is gone and lost for ever by the ceaseless jiggling of the agents.

Emergent Tessellations

With a suitable repel strength the points all settle down in a triangular pattern because whenever they diverge from this grid they are in an unstable situation and will always fall back into the triangular lattice. The point to note is that these wiggles are not in the algorithm (all it states is the backing off principle outlined above). What would one expect from such an algorithm? At first sight perhaps just aimless wandering; however it does in fact settle down as if pulled into alignment by some "force" not implied by the two lines of code. This is an example of "emergence" – the idea that the program, by operating continuously in parallel engenders a higher order observation, which could be characterised as a simple demonstration of the principle that the triangular lattice is the least cost minimum energy equilibrium point for a 2d tessellation, with each point equidistant to 6 others. Here is out first example also of an algorithm which possessed epistemic independence of the model (in this case the code of the repel algorithm) from the structural output of its being run. (the stable triangular tessellation)

Distributed representation

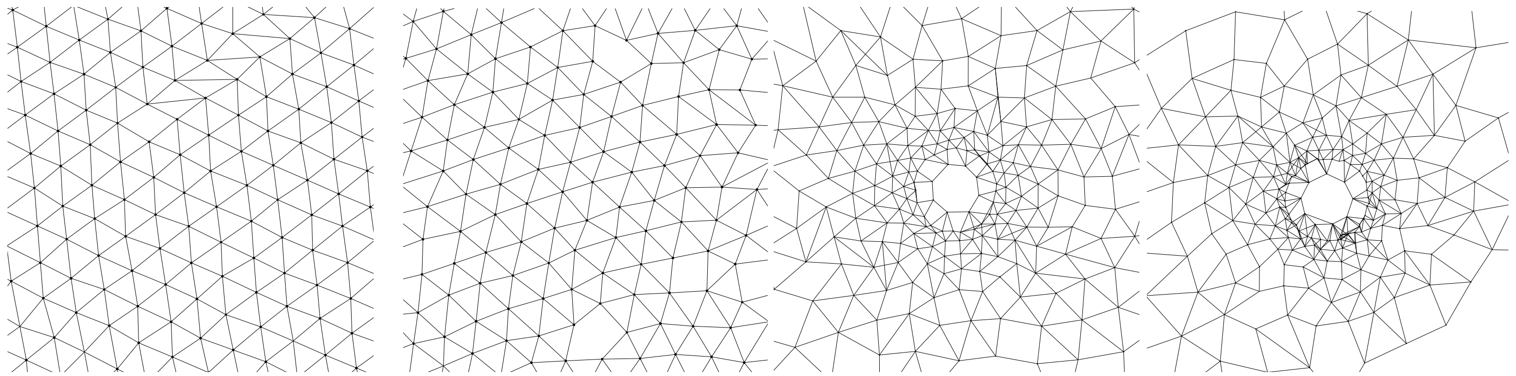
This is also the first example of many that illustrates the notion of distributed representation. The way the algorithm works is to embed the rules to be **simultaneously followed** in EACH turtle. Each turtle (small autonomous computational entity) is running the little program described above with its own decision making - who is nearest to MYSELF - and behaves independently of the other little computers - I turn this THIS WAY and back off. The repel algorithm is the only available description we can find in this system, everything else is just general scheduling events and general start stop for the whole simulation and this representation is present in EVERY turtle. The turtles can interact with each other and have some limited observational powers, for instance they can 'feel' the nearest turtle and take appropriate action, but they don't know about the triangular tessellation since that can only be observed by the global observer - in this cas the person (you) running the simulation on your computer. This distinction between different levels of observer is a key aspect of distributed representation, and will crop up many times in the following pages. It is vital with distributed representation models that there is some feedback present between these little autonomous programs, if each one took no notice of its neighbours then nothing would happen. This is evident in the cellular automata shown next and the canonical 'pondslime' algorithm' introduced last in this chapter.

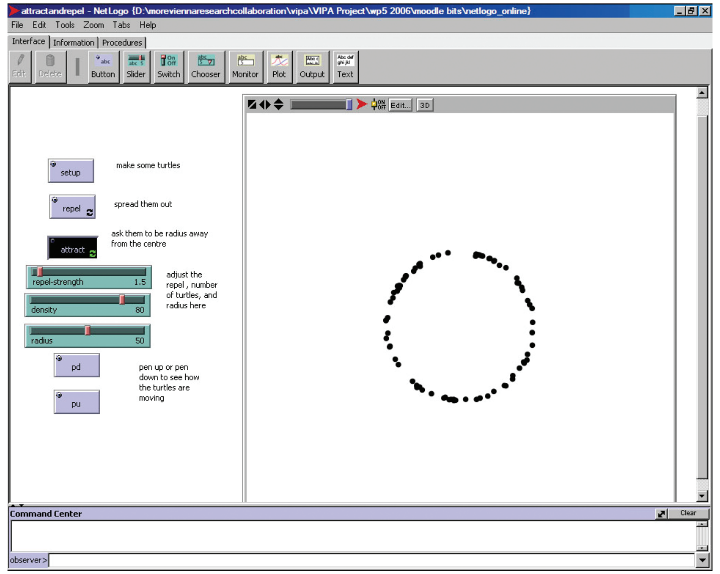
It is instructive to compare this bottom up small program with the conventional recipe for a triangular tessellation. Of course there are many ways of describing how to draw such a pattern, but using a simple wallpaper approach you might say:

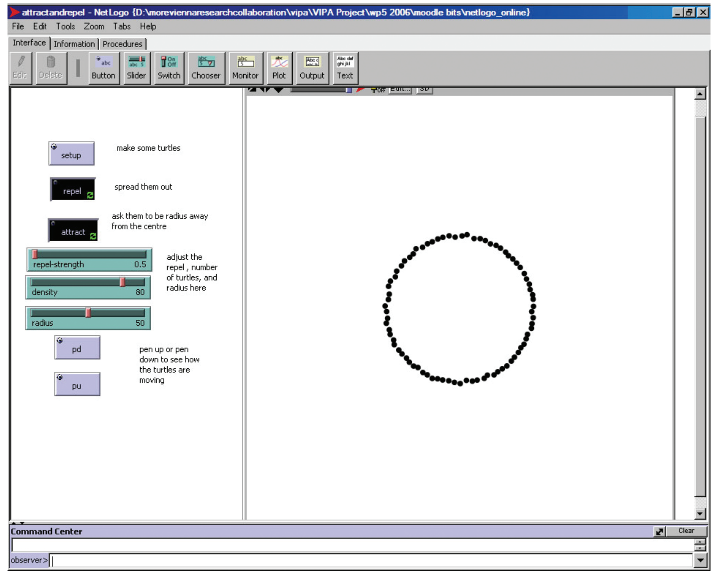
Wallpaper algorithm

```
Set out a line of dots at a spacing of 1
Duplicate this line with an offset of 0.5 in the X direction
and the square root of 0.75 in the y direction.
Do this as many times as you like.
```

The square root of 0.75 is the height of an equilateral triangle of side 1 derived from Pythagoras (where $height^2 + 0.5^2 = 1^2$; so h = √1-0.25), which evaluates to approximately 0.86602540378443864 676372317075294. This is not a very attractive number and seems to suggest that this algorithm is not capturing the real description of the underlying dynamics, but just mechanically constructing a top down and rather clumsy measurement of the outcome. This distinction should be remembered when simulations and modelling are discussed elsewhere, as it forms part of the argument in favour of the "short description" encoded in the generative rule rather than the "long description" involved in traditional geometry.







IFELSE is an example of one of the key concepts of any programming language, the ability to get the computer to ask a question about which of a number of things to process. known as a 'conditional statement' it has many forms, but in this language in this situation we use the phrase 'ifelse'

This construct has to decide which of two possible routes to take in the flow of the program.

Cheesy illustration: *If standing at a fork in the road , with the possiblity of going left or right you need some way of evaluating the choices open to you. so there you are, what do you do ? It happens you have a note from your aunt in your pocket, you take it out and it says:*

{"when reaching a fork in the road, if its after lunch turn left, else turn right"}

It is clearly just after lunch, so you take the left turn. Problem resolved. (the left turn takes you to the tea rooms obviously)

In the script of `attract` the note from your aunt is asking "if your distance to the centre less than radius then take a step back otherwise step forwards"

The general notion of IFELSE is that you ask a question, then on the basis of the TRUTH or otherwise of the statement you choose between two possibilities

IF <something is true> ***THEN*** DOTHIS
ELSE DOTHAT

thats why its called ifelse
formally
ifelse (conditional expression)
[thing to do if true]
[thing to do if false]

Extending the model – drawing circles with turtles

The following examples are based on the Papert paradigm of allowing the geometry to emerge from the algorithm rather than being imposed from outside. In this case the geometry is based on the circle, which is then extended to cover more complex geometries such as the voronoi (emergent tessellation). These are "illustrations of consensus" because the bit you can see (the figures below) is the emergent result of all the components of the system (turtles mostly) finally reaching some agreement about where to be. The phrase begs the question as to what the turtles are being asked to agree about, and what architectural idea might be involved. Generally the task is to distribute themselves with respect to two conflicting pressures – that of the group based on some higher order pattern, and that of the individual.

Papert points out that the equations

Xcirc = originX + Radius cos (angle)
Ycirc = originY + Radius sin (angle)

Do not capture any useful information about circles, whereas we can write a small program in LOGO to get one turtle to walk in a circle by telling it to go forward and left a bit. (see chapter 2 for background on Seymor Papert)

```
To circle
      Repeat 36
      Forward 1
      Turn Left 10
End repeat
End circle
```

requires only English and a familiarity with walking.

As Resnic points out in 'Turtles Termites and Traffic jams' with parallel computation we can propose another implementation of the circle using not just one turtle but many of them. The algorithm is based on the characterisation of a circle as being:
An array of points all at the same distance from another common point

To do this with turtles we :

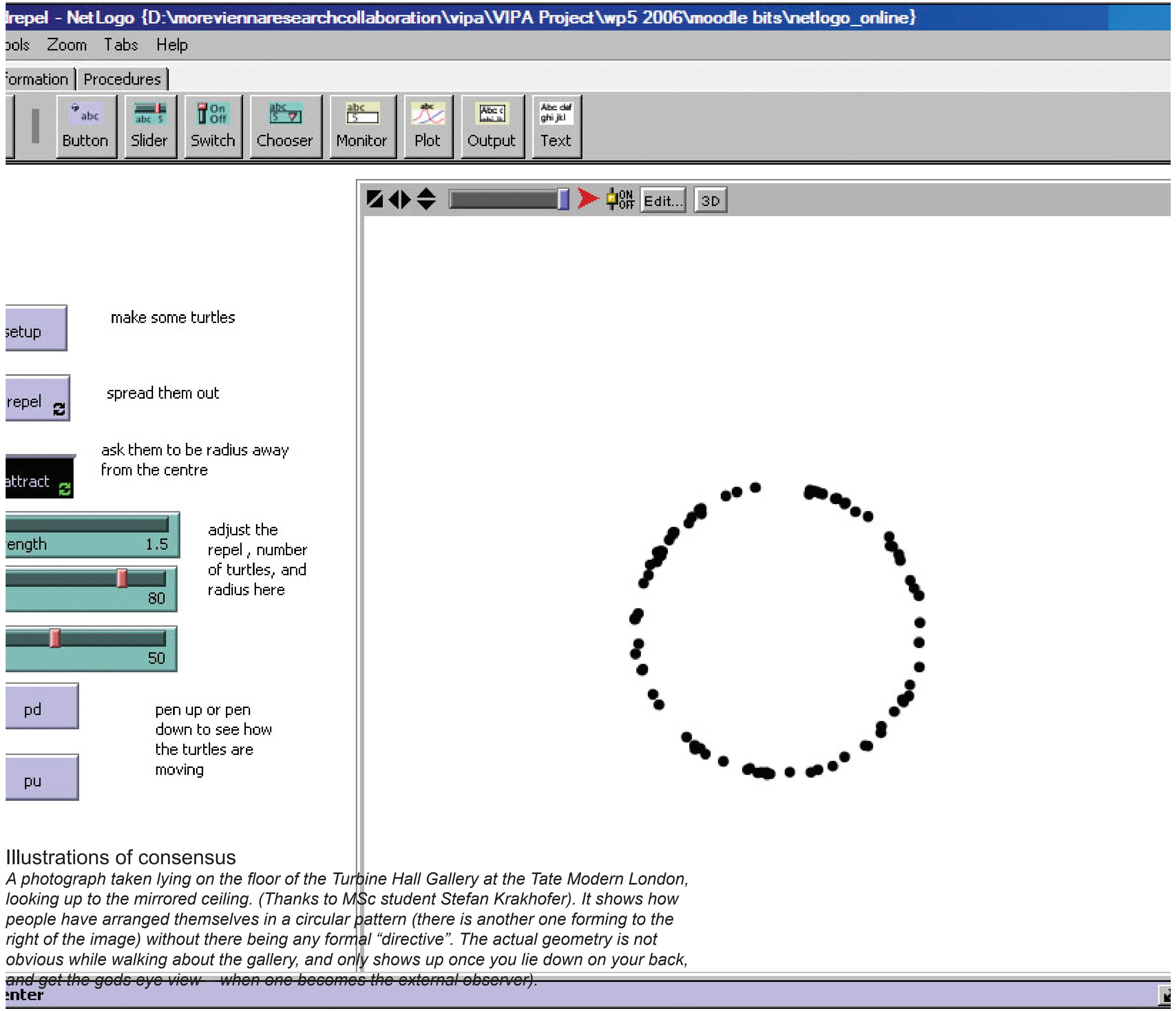
- create a lot of turtles at random
- get each turtle to turn towards the centre of the circle
- get each turtle to measure the distance between itself and this centre point
- if this distance is less than the desired radius then take a step back (because you're too near)
- if it is greater then take a step forward (because you're too far away)
- Go on doing this for ever.

This procedure can be written in netlogo as so:

```
to attract
      ask turtles
      [
            set heading towardsxy 0 0
            ifelse ((distancexy 0 0 ) < radius)
                  [bk 1]
                  [fd 1]
      ]
end
```

Notice that nowhere in the procedure is it given where the turtles are to walk to, they just walk back and forth. In fact the "circle" is only apparent to the human observer, and while we look at it, it shimmers into being rather than being constructed carefully.
The result is a ring of turtles defining a circle. In fact there is one more thing to do because just using this process will result in an uneven circle with gaps in because the turtles start off randomly and gather in random spacings around the circumference. How can we get the turtles to spread themselves out? – the answer is to do the repel procedure we have already looked at. This version backs of not 1 unit but a variable amount controlled by a "slider" on the interface.

```
to repel
     ask turtles
      [
      set closest-turtle min-one-of other turtles
[distance myself]
      set heading towards closest-turtle
      bk repel-strength
       ]
end
```



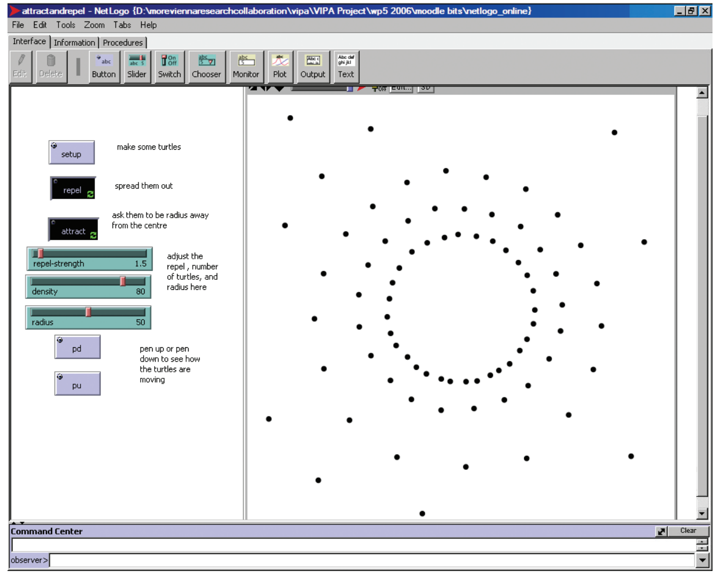
Illustrations of consensus

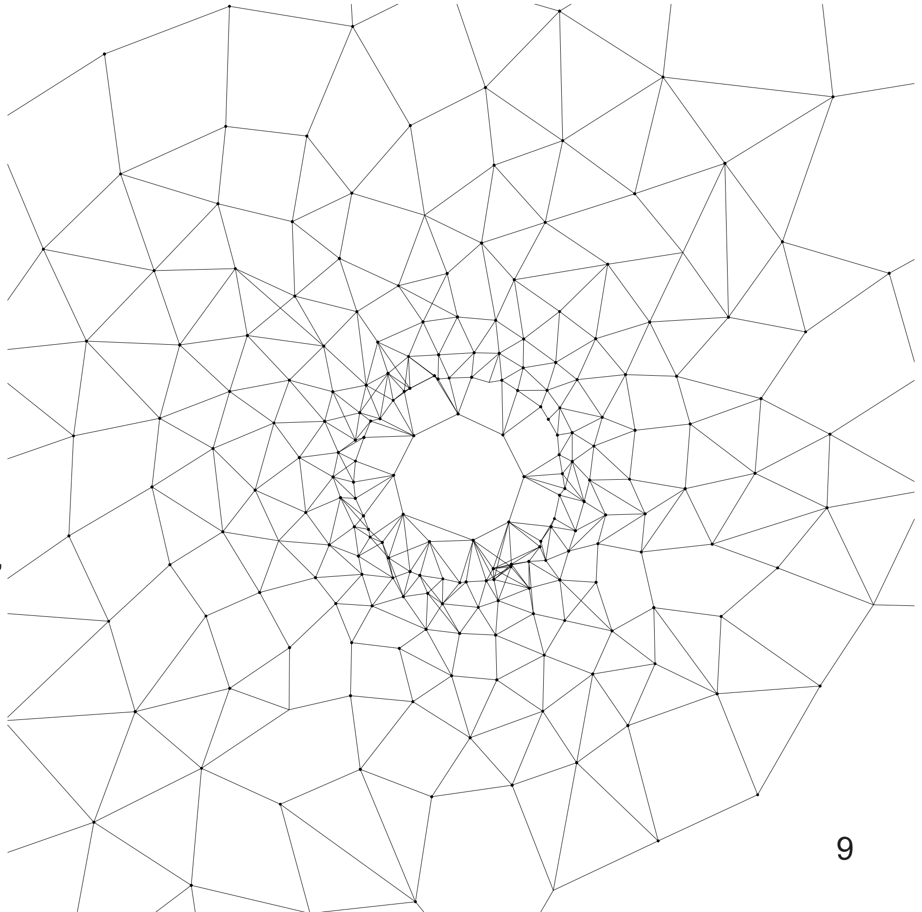
A photograph taken lying on the floor of the Turbine Hall Gallery at the Tate Modern London, looking up to the mirrored ceiling. (Thanks to MSc student Stefan Krakhofer). It shows how people have arranged themselves in a circular pattern (there is another one forming to the right of the image) without there being any formal "directive". The actual geometry is not obvious while walking about the gallery, and only shows up once you lie down on your back, and get the gods eye view – when one becomes the external observer).

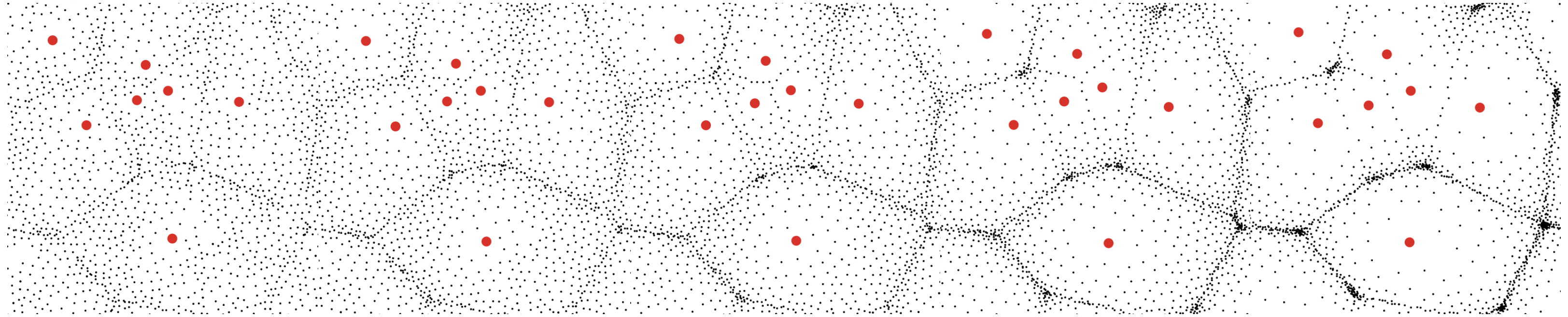
These two procedures use two references to globally defined values which affect the system being simulated, called 'radius' and 'repel strength'. These named values are referred to as variables (because they can contain numbers that vary of course). In Netlogo you can set the variables through the user interface by using sliders.

You might say that this isn't a 'real' circle, but just a messy thing that is a bit circular. But, like the triangular tesselation example the classical definition of pi as the ratio of the circumference divided by the diameter is famously unresolvable. In fact the expansion of pi can be used as the basis for generating a random sequence as it is impossible to predict the next number in the sequence by any means other than continuing to iterate the division sum. In other word in our universe circles cannot be identified with whole numbers, every measurement of a circular thing is inevitably a compromise, only resolved by its eventual instantiation into an array of bricks, pieces of steel etc. So repel and attract (which only use simple adds and no funny ratios) seem more fundamental descriptions, generating the funny ratios out of the process rather than squashing them in by force.
With these two variables, 'repel' and 'attract' form a useful test bed for experiments. There is a relationship between the values of the variables such that, if you make the radius very small then you of course make a smaller circle, then if you make the repel strength quite large, then depending on the number of turtles (another variable) the turtles will find it impossible for all of them to comfortably fit on the circumference. The actual result is quite surprising, as it leads to a series of well formed rings of turtles at ever increasing distances from the nucleus. In many ways this could be seen as a model of bohrs model of the atom, since the radius is the overall energy of the atom and the repulsion force is the energy level of an electron. (this is intended only as an illustration of the possible explanatory power of these simple models and not a claim to deep physical truth!). What is undeniable is that, instead of a general fuzzy ring of turtles from radius outwards, they only inhabit particular rings, which again is not in the model. The text of the algorithm does not include an explicit reference to annular ringyness, but only one circle.

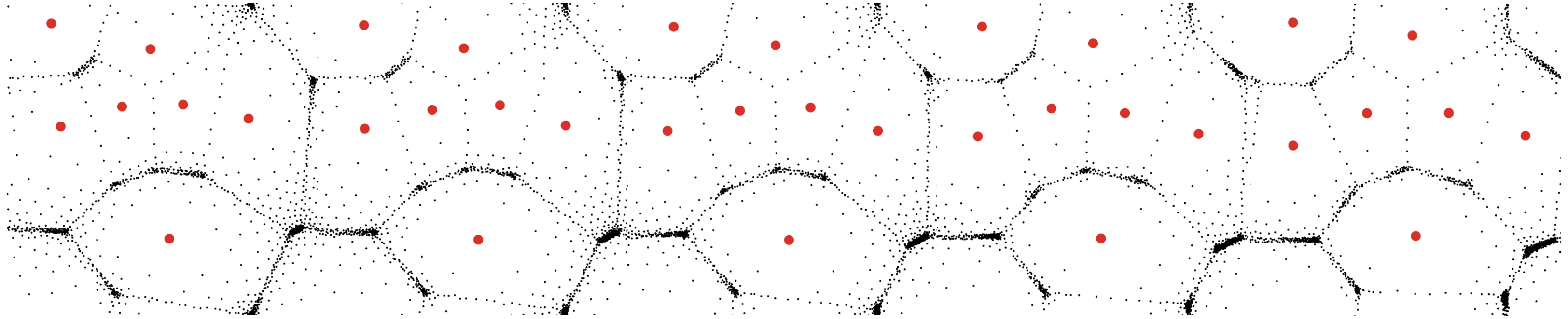
Given the high level of abstraction we can begin to model more complex shapes and spatial organisations than individual geometric objects without having to do much extra coding.

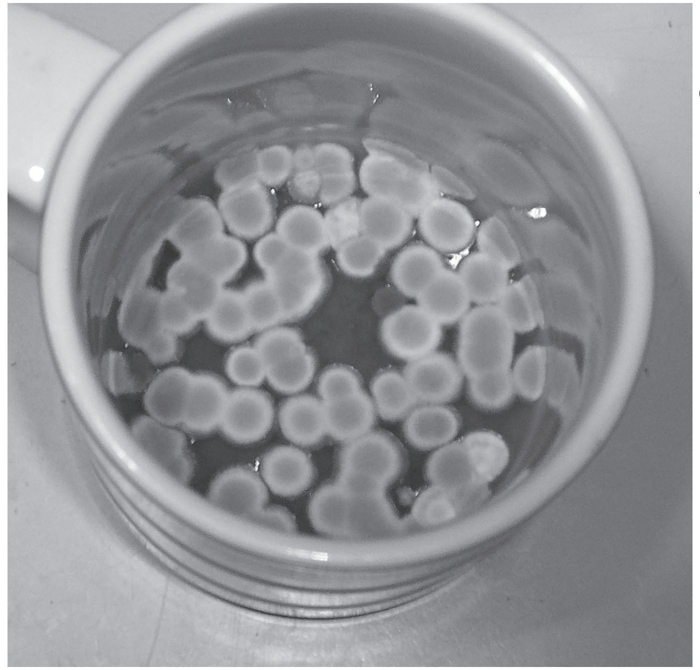






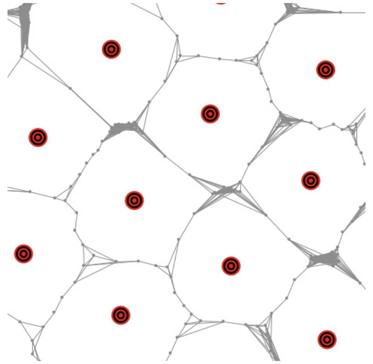
The simulation begins with the two kinds of turtles sprijnkled randomly about 'normal' turtles (little)and 'target' ones (big).Slowly the smaller normal turtles retreat to the given radius distance in the attcat procedure, gathering on the boundaries in ever greater numbers. they cant go near other targets but end up in a position which is as far away from all the nearest targets

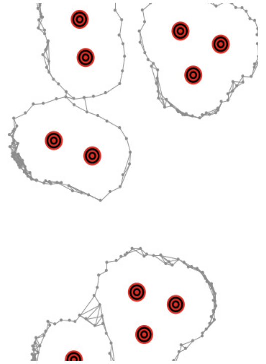




If the program models the process to be represented rather than the graphics of the outcome it is likely to be a better , shorter model.

Image of mould growing in a coffee cup showing agglomeration of disc like elements into a voronoi like mat





Extending the model - drawing bubbles

A more complex outcome that we can achieve with only small modifications is the emergent voronoi diagram (dirichelet tessellations). Voronoi diagrams are conventionally calculated using computational geometry. A voronoi diagram is a pattern which describes the minimal energy pathways between a set of points. Looking at such a diagram we can see that each initial point is separated from its immediate neighbours by being enclosed in a polygon, face joining the polygons of all its neighbours.

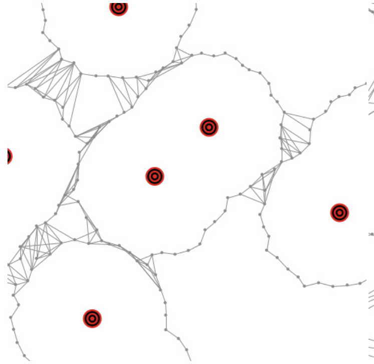
Taking the two procedures attract and repel we can make a small modification to the attract one, so that instead of turtles being attracted to the constant location 0 0 they are interested instead in another of the turtles acting as a "target". So we can make two kinds of turtles – normal ones and targets. Both the normal turtles and the target turtles obey the repel rule, but the attract rule only applies to normal turtles, who try to stay at a particular radius from the target turtles.

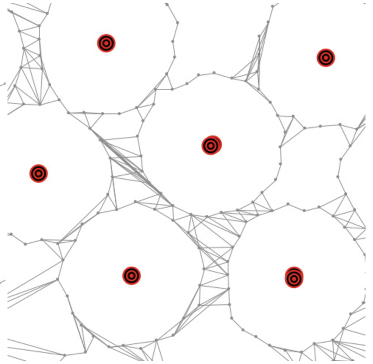
```
to attract
   locals [targets]
      ask turtles
      [
         set targets turtles with [target = true]
         set closest-turtle min-one-of other
targets [distance myself]
         set heading  towards closest-turtle
         ifelse ((distance closest-turtle ) <
radius) [bk 1] [fd 1]
      ]
end
```

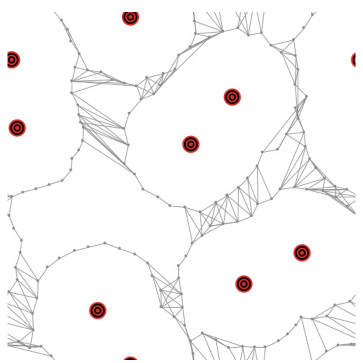
Emergent spatial tessellation of minimal path polygons

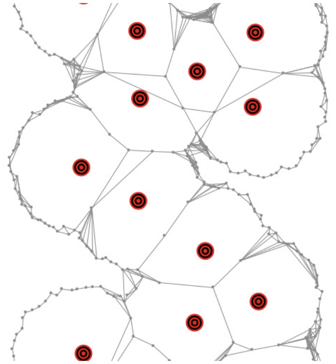
In this series(left to right) a very large number of turtles slowly retreat from the stationary targets (larger dots) to form the boundaries of the *voronoi tessellation*. This is an example of an emergent self organised structure, where the algorithm goes with the flow of the problem to be solved, namely draw the equidistant boundaries given the initial distribution of points. The answer emerges naturally from the very simple process described above.
The difference between the code for drawing a circle and the code for drawing a voronoi diagram using the traditional "computational geometry" approach is huge, the two trig functions described earlier have to be expanded to many pages of code dealing with complex maths and elaborate sorting and scheduling procedures in order to define the polygons, whereas the step from circle to voronoi using the attract and repel procedures is simply to have two kinds of turtles and a lot more of them!

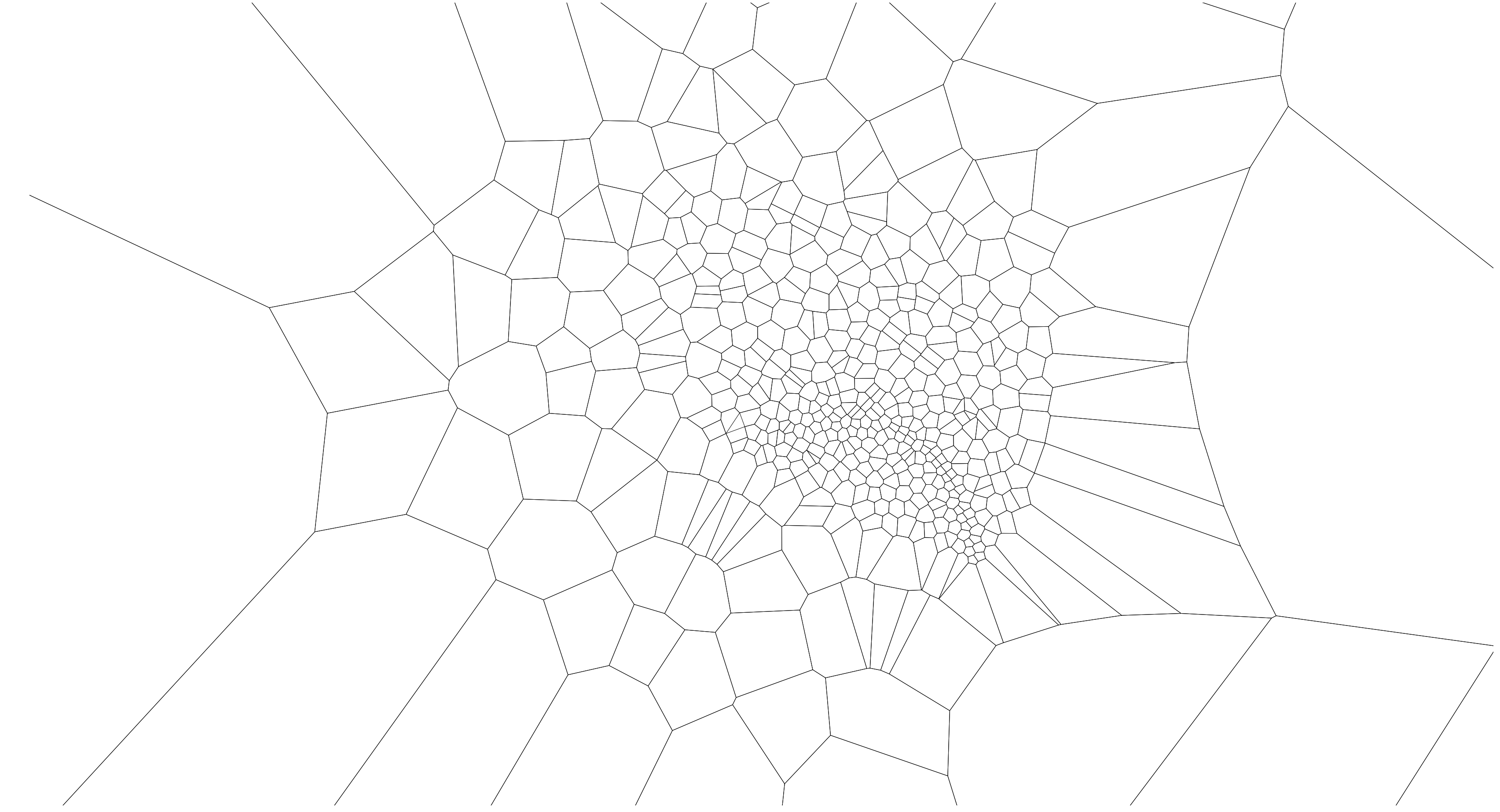
All this is intended to illustrate the fundamental point about how representational methods can change when we use the turing machine to generate form. as we shall see in the next section the complexity of the emergent forms can be much higher than defining them in purely geometric ways. With these two texts we can represent a huge range of objects, and interestingly the representation hardly has to change at all to accommodate the 3rd dimension.











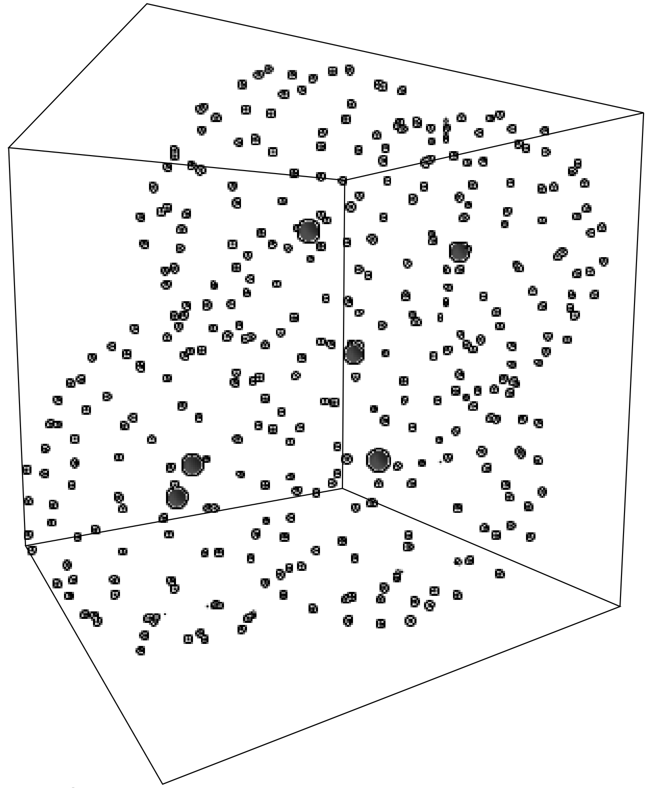
Voronoi by computational Geometry. this was generated as part of an experiment in recursive voronoi diagrams where each generation provides the seed points for the next diagram

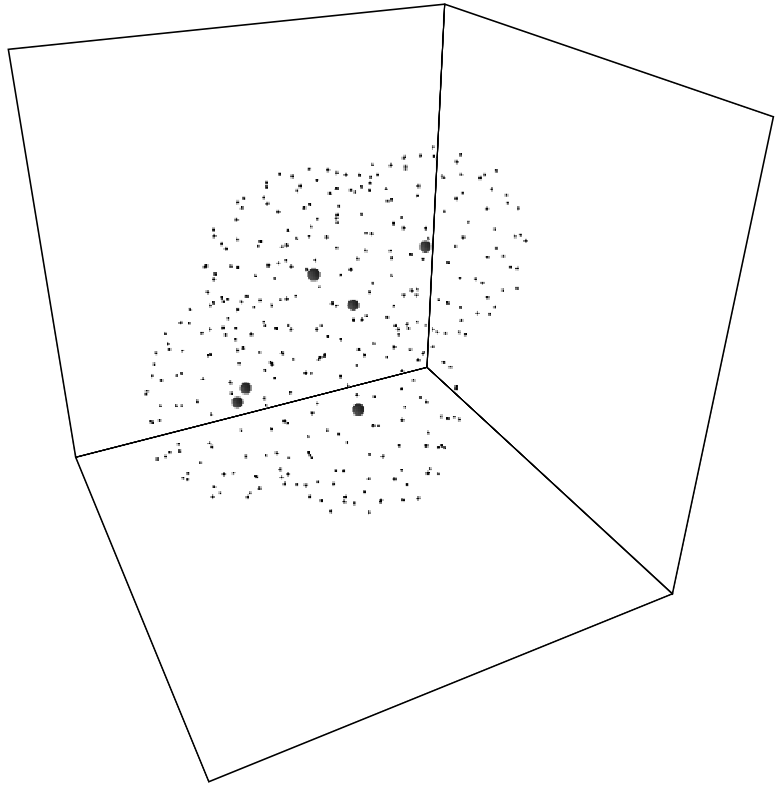

```
Attribute VB_Name = "Voronoibits"

    '------------------------------------
-- changing datastructure to hold indeces
into originalpoints
               '--------------------------
------------   rather   than   points
11.6.03---------------------
      ' defining the cells of the voronoi
diagram
    ' working 26 june 03

    Const pi = 3.1415926535
    Const yspace = 0
    Const xspace = 1


    Type pointedge
     pos As point         'position of in-
tersection
       Bedge(2)  As  Integer  'indeces  into
boundary array where intersection occurs
    End Type


    Type intersectStuff
    outnode As point
   outnodeid As Integer  'index into vertex
array for voronoi cell
    beforeinter As pointedge
    afterinter As pointedge
    End Type


    Const VERYSLOW = 0.7
    Type mypoint
        x As Double
        y As Double
        z As Double
        spacetype As Integer
        kuller As Integer
    End Type

     Type pair 'to tie the triangle nos to
the sorted angles
        value As Double
        index As Integer
    End Type

    Type delaunay
        p1 As Integer
        p2 As Integer
        p3 As Integer
         circcentre As mypoint        ' the
coordinates of the centre of the circle by
3 pts constructed by this point
         circrad As Double            ' the
radius of this circle
    End Type

    Type cell
        item() As Integer
        tot As Integer
        area As Double
        id As Long
        spacetype As Integer
        jump As Boolean
        kuller As Integer
    End Type

    Public pts As Integer
    Public numtriangles As Integer
    Public originalpoints() As mypoint
    Public triangles() As delaunay
    Public cells() As cell
    Public neighbour() As cell

    Public cyclesmax As Long
    Public cycles As Long


  Sub voronoi(d As Integer)
    ReDim cells(1 To pts) As cell
    ReDim neighbour(1 To pts) As cell
      Dim  i  As  Integer,  j  As  Integer,  k
As Integer


    For i = 1 To pts
   cells(i).spacetype = originalpoints(i).
spacetype ' having been set in teatime
      cells(i).kuller = originalpoints(i).
kuller
    Next i


    cycles = 0
    numtriangles = 0
    'cyclesmax = pts ^ 3

    For i = 1 To pts
        For j = i + 1 To pts
            For k = j + 1 To pts
              ' the triangles array is popu-
lated in the sub drawcircle - sorry !!
                    drawcircle_ifnone_inside
i, j, k, pts
                  cycles = cycles + 1
                  'counterform.count_Click
            Next k
        Next j
    Next i

     collectcells (0)       'define data for
all voronoi cells
    neighcells (0)        'define

  End Sub
    Sub collectcells(d As Integer) ' popu-
lates array cells with lists of all the
vertex incident triangles of a point
      Dim  v  As  Integer,  N  As  Integer,  t
As Integer

     For v = 1 To pts ' go through all the
original points
         N = 0
          ReDim cells(v).item(1 To 1)
            ' drawpoint originalpoints(V),
acGreen, 2
            ' ThisDrawing.Regen acAllView-
ports

              For t = 1 To numtriangles 'go
through all triangles
                 If triangles(t).p1 = v Or
triangles(t).p2 = v Or triangles(t).p3 =
v Then
              N = N + 1 '' T is index into a
tri sharing a vertex with originalcells(V)
                  ReDim Preserve cells(v).
item(1 To N)
                 cells(v).item(N) = t
                 cells(v).tot = N
              End If
           Next t
           sortbyangle v, cells(v)
    Next v
    End Sub
    Function centre_gravity(this As delau-
nay) As mypoint
      Dim tx As Double, ty As Double, tz
As Double
           tx   =   (originalpoints(this.
p1).x   +   originalpoints(this.p2).x   +
originalpoints(this.p3).x) / 3
           ty   =   (originalpoints(this.
p1).y   +   originalpoints(this.p2).y   +
originalpoints(this.p3).y) / 3
    tz = 0

    centre_gravity.x = tx
    centre_gravity.y = ty
    centre_gravity.z = tz

    End Function

    Sub sortbyangle(index As Integer, this
As cell)
     Dim angles() As pair, i As Integer, O
As mypoint, CG As mypoint
    ReDim angles(1 To this.tot) As pair
    O = originalpoints(index)
        For i = 1 To this.tot
        CG = centre_gravity(triangles(this.
item(i)))
            angles(i).value = getangle(O,
CG)
            angles(i).index = this.item(i)
        Next i
    bubblesort angles, this.tot
        For i = 1 To this.tot
            this.item(i) = angles(i).index
        Next i

    End Sub
     Sub bubblesort(s() As pair, N As In-
teger)
       Dim index As Integer, c As Integer,
swap As Integer, temp As pair

Do
    swap = False
    For c = 1 To N - 1

           If s(c).value > s(c + 1).value
Then
            temp = s(c)
            s(c) = s(c + 1)
            s(c + 1) = temp
            swap = True
        End If

    Next c
Loop Until (swap = False)

End Sub
Function getangle(st As mypoint, fin As my-
point) As Double

Dim q As Integer, head As Double, add As
Double
Dim xd As Double, yd As Double, r As Dou-
ble
' calculate quadrant
If fin.x > st.x Then
 If fin.y > st.y Then
    q = 1
  Else
    q = 2
 End If
  Else
   If fin.y < st.y Then
    q = 3
  Else
    q = 4
  End If
 End If


 Select Case q

 Case 1
    xd = fin.x - st.x
    yd = fin.y - st.y
    If xd = 0 Then
        r = pi / 2
    Else
        r = yd / xd
    End If
    add = 0
 Case 2
    yd = st.y - fin.y
    xd = fin.x - st.x
    add = 270
    If yd = 0 Then
        r = pi / 2
    Else
        r = xd / yd
    End If
 Case 3

    xd = st.x - fin.x
    yd = st.y - fin.y
    If xd = 0 Then
        r = pi / 2
    Else
        r = yd / xd
    End If
    add = 180
 Case 4
    xd = st.x - fin.x
    yd = fin.y - st.y
    If yd = 0 Then
        r = pi / 2
    Else
        r = xd / yd
   End If
    add = 90
 End Select

 If xd = 0 Then
    getangle = 90 + add
 Else
    getangle = ((Atn(r) / pi) * 180) + add
 End If


End Function
    Sub neighcells(d As Integer)

    Dim v As Integer, N As Integer, nbs As
Integer, cp As Integer

    For v = 1 To pts
   nbs = 0                             'go
through the item list for this cell (based
on vertex V)
          For  cp  =  1  To  cells(v).tot  -  1
'the indeces into array cells
                N  =  matchupcells(cells(v).
item(cp),  cells(v).item(cp  +  1),  v)  'two
points on the voronoi region
            If N > 0 Then
                nbs = nbs + 1
              ReDim Preserve neighbour(v).
item(1 To nbs)
                     neighbour(v).item(nbs)
= N
                neighbour(v).tot = nbs
            End If
        Next cp
    Next v
    End Sub

Function matchupcells(p1 As Integer, p2 As
Integer, current As Integer) As Integer

' find a cell (in array cells)which shares
an edge p1 - p2 with this cell (current)
Dim m As Integer, v As Integer, cp As In-
teger

matchupcells = 0

        For v = 1 To pts
          If v <> current Then 'dont look
at you own list
            m = 0

          'a voronoi region can only share
two verteces ( one edge) with any other
            'but since the edges are organ-
ised anti clockwise, the neighbouring cell
              'will be going the other way.
so here we just look for two matches hope
thats ok?
                For cp = 1 To cells(v).tot
'run through vertex list for this cell
                    If cells(v).item(cp) =
p1 Then m = m + 1
                    If cells(v).item(cp) =
p2 Then m = m + 1
                Next cp
                If m = 2 Then
                    matchupcells = v
                   Exit For       'dont go
on looking once found a match
                End If
            End If
        Next v
    End Function

     Sub drawcircle_ifnone_inside(i As In-
teger, j As Integer, k As Integer, pts As
Integer)
    Dim testcircle As delaunay

    testcircle.p1 = i
    testcircle.p2 = j
    testcircle.p3 = k
    circbythreepts testcircle
    If Not inside(testcircle, pts) Then
         'drawpoint testcircle.circcentre,
acYellow, testcircle.circrad
        numtriangles = numtriangles + 1
          ReDim  Preserve  triangles(1  To
numtriangles)
           triangles(numtriangles) = test-
circle
    End If

    End Sub

     Function inside(this As delaunay, pts
As Integer) As Integer
     ' are there any points closer to the
centre of this circle than the radius

    inside = False
     Dim  i  As  Integer,  dd  As  Double,  cr
As Double
    For i = 1 To pts
    'ignore points that are on this circle
          If i <> this.p1 And i <> this.p2
And i <> this.p3 Then
              dd = distance(this.circcentre,
originalpoints(i))
              cr = this.circrad
                 If (dd < cr) Then
                    inside = True
                    Exit For
                 End If
          End If
     Next i
    End Function
    Sub circbythreepts(this As delaunay)

    Dim a As Double, b As Double, c As Dou-
ble, k As Double, h As Double, r As Double,
d As Double, e As Double, f As Double
    Dim pos As mypoint
     Dim k1 As Double, k2 As Double, h1 As
Double, h2 As Double

       a  =  originalpoints(this.p1).x:  b  =
originalpoints(this.p1).y
       c  =  originalpoints(this.p2).x:  d  =
originalpoints(this.p2).y
       e  =  originalpoints(this.p3).x:  f  =
originalpoints(this.p3).y


     'three points (a,b), (c,d), (e,f)
      'k  =  ((a²+b²)(e-c)  +  (c²+d²)(a-e)  +
(e²+f²)(c-a)) / (2(b(e-c)+d(a-e)+f(c-a)))
    k1  =  (((a  ^  2)  +  (b  ^  2))  *  (e  -  c))  +
(((c  ^  2)  +  (d  ^  2))  *  (a  -  e))  +  (((e  ^  2)
+  (f  ^  2))  *  (c  -  a))
   k2  =  (2  *  ((b  *  (e  -  c))  +  (d  *  (a  -  e))
+  (f  *  (c  -  a))))

    k = k1 / k2

      'h  =  ((a²+b²)(f-d)  +  (c²+d²)(b-f)  +
(e²+f²)(d-b)) / (2(a(f-d)+c(b-f)+e(d-b)))
    h1  =  (((a  ^  2)  +  (b  ^  2))  *  (f  -  d))  +
(((c  ^  2)  +  (d  ^  2))  *  (b  -  f))  +  (((e  ^  2)
+  (f  ^  2))  *  (d  -  b))
   h2  =  (2  *  (((a  *  (f  -  d))  +  (c  *  (b  -  f))
+  (e  *  (d  -  b)))))
    h = h1 / h2

    'the circle center is (h,k) with radius;
r² = (a-h)² + (b-k)²
    r = Sqr((a - h) ^ 2 + (b - k) ^ 2)

    pos.x = h: pos.y = k: pos.z = 0
    ''drawpoint pos, acYellow, r
    this.circcentre = pos
    this.circrad = r

    End Sub

    Sub convert(b As mypoint, f As mypoint,
start() As Double, finish() As Double)

    start(0) = b.x
    start(1) = b.y
    start(2) = b.z
    finish(0) = f.x
    finish(1) = f.y
    finish(2) = f.z
    End Sub

     Function findcenter(pts As Integer) As
mypoint
    Dim xt As Double, yt As Double

    xt = 0
    yt = 0

        For i = 1 To pts
            xt = xt + originalpoints(i).x
            yt = yt + originalpoints(i).y
        Next i

        findcenter.x = xt / pts
        findcenter.y = yt / pts
        findcenter.z = 0

    End Function



     Sub Draw_Line(b As mypoint, f As my-
point, c As Integer)
    Dim lineobj As AcadLine
    Dim mLineObj As AcadMLine
     Dim start(0 To 2) As Double, finish(0
To 2) As Double

    convert b, f, start, finish

     Set lineobj = ThisDrawing.ModelSpace.
AddLine(start, finish)

    lineobj.color = c
    lineobj.Layer = "delaunay"
    'lineobj.Update


    End Sub
    Sub drawpoly(this As cell)
    Dim tri As delaunay
    Dim plineObj As AcadLWPolyline
'changed to lw polyline so only duets of
coords not trios
        Dim thepoly(0) As AcadEntity 'thing
to use in addregion
           Dim boundary As Variant 'assign
with addregion
          Dim boundy() As AcadRegion 'thing
you redim
         Dim acell As AcadRegion
          Dim numtri As Integer, thepoints()
```

The code on this page can be contrasted with the short snippet on the previous page. both are doing essentially the same thing - generating the minimal path tessellation known as a Voronoi Diagram. However one is written in NetLogo as parallel process of dynamic systems of turtles, the other in BASIC as an exercise in computational geometry. (code by the author). not only is the BASIC enormously longer, but it is also much more restrictive in that it doesn't allow for easy manipulation of the underlying generating points or alterations of the dynamics of the particles. the only advantage this approach has over the emergent version is that the defined polygons are explicitly defined by ordered line segments whereas the images taken from the agent based examples would need a little post processing to define them.

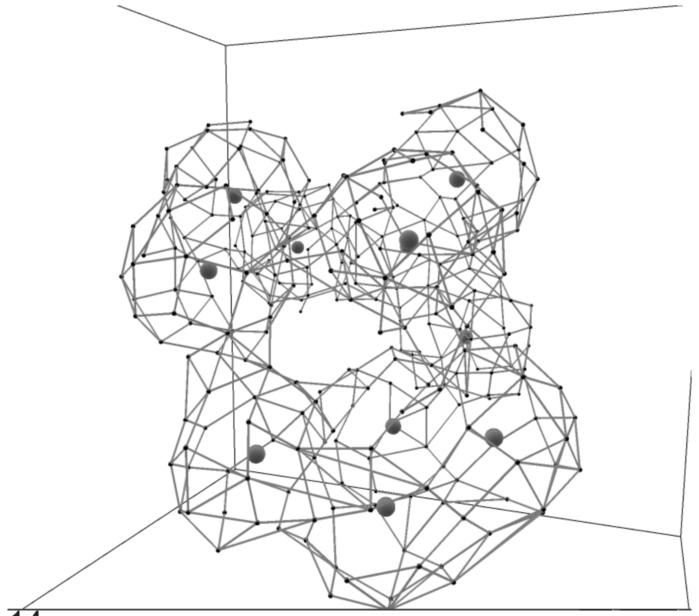
BASIC is a very old programming language used in many windows applications to automate operations. See chapter 3 for a discussion of the badness of BASIC

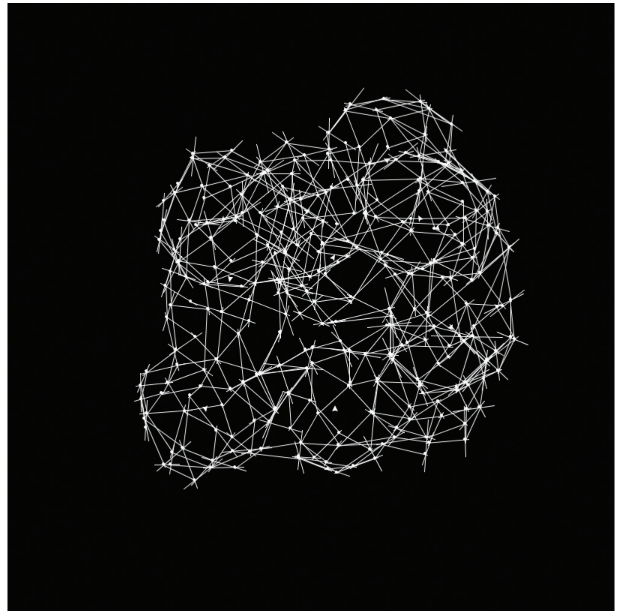
```
As Double, TPC As Integer
    numtri = this.tot * 2 - 1
    ReDim thepoints(numtri + 2) As Double
    TPC = 0

    ' loop through all the items getting the
coordinates of the circlcentres that are
     ' inside the elements of the thetri-
angles array

    For i = 1 To this.tot
          thepoints(TPC) = triangles(this.
item(i)).circcentre.x
           TPC = TPC + 1
          thepoints(TPC) = triangles(this.
item(i)).circcentre.y
           TPC = TPC + 1
        ' thepoints(TPC) = triangles(this.
item(i)).circcentre.z
          ' TPC = TPC + 1
    Next i
   thepoints(TPC) = thepoints(0)
    TPC = TPC + 1: thepoints(TPC) = thep-
oints(1)
    'TPC = TPC + 1: thepoints(TPC) = thep-
oints(2)

    If TPC > 3 Then
      On Error Resume Next      'got crash
on huge poly
         Set plineObj = ThisDrawing.Model-
Space.AddLightWeightPolyline(thepoints)
         If plineObj.area > 0 Then

         Set acell = makeregion(plineObj)

         On Error Resume Next
              acell.Boolean acIntersec-
tion, bound
            this.area = acell.area
               this.id = acell.ObjectID
'changed to acell
            If this.spacetype = 1 Then
            acell.color = this.kuller
            Else
            acell.color = acWhite
            End If

           ' acell.Update
              ' ThisDrawing.Regen acAc-
tiveViewport
            makeboundaryregion 0

        End If

    End If

    End Sub


    Sub drawcircle(x As Variant, y As Vari-
ant, kuller As Integer, size As Integer)
    Dim p(2) As Double, circ As AcadCircle
    p(0) = x: p(1) = y: p(2) = 0
      Set  circ  =  ThisDrawing.ModelSpace.
AddCircle(p, size)
    circ.color = kuller
   ' circ.Update

    End Sub
     Function random(bn As Double, tn As
Double) As Double

          random = ((tn - bn + 1) * Rnd
+ bn)

    End Function


     Function distance(startp As mypoint,
endp As mypoint) As Double
    Dim xd As Double, yd As Double
    xd = startp.x - endp.x
    yd = startp.y - endp.y
    distance = Sqr(xd * xd + yd * yd)
    End Function


    Sub drawpoint(pos As mypoint, c As In-
teger, r As Double)
          ' This example creates a point in
model space.
        Dim circleObj As AcadCircle
        Dim location(0 To 2) As Double
        location(0) = pos.x
        location(1) = pos.y
        location(2) = pos.z
        ' Create the point
          Set circleObj = ThisDrawing.Mod-
elSpace.AddCircle(location, r)
        circleObj.color = c
        'ZoomAll
    End Sub
```





From rings of points to spherical clouds. Below, using a link turtle to join the dots





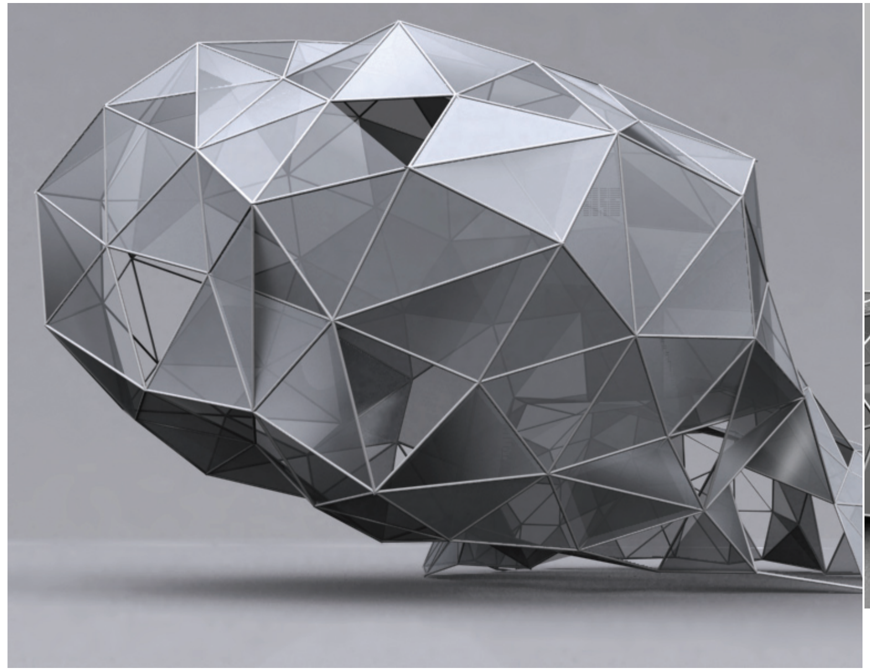
Moving into the 3rd dimension

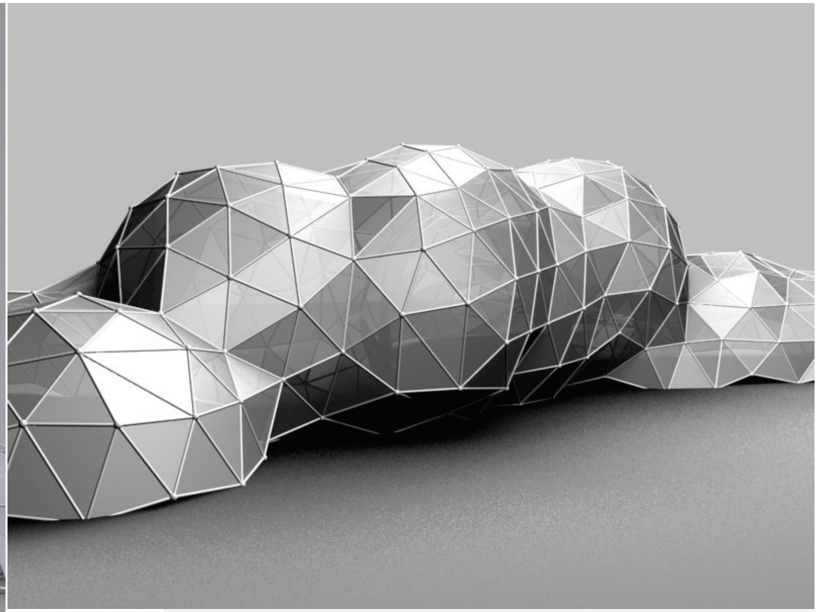
The code below is pretty much the same as before (there are a few differences due to the 3d version of the language being a revision behind the 2d version but we can ignore those) apart from that the only difference is the use of the word *pitch* as well as *heading*, which allows the turtles to point towards things in 3d space

```
to attract

ask nodes
[
      set closest-turtle min-one-of targets with other targets  [distance  myself]
      set heading  towards-nowrap closest-turtle
      set pitch towards-pitch-nowrap closest-turtle
      ifelse ((distance closest-turtle ) < radius) [bk 1] [fd 1]
]

end

to repel-nodes

ask nodes

    [
    set closest-turtle min-one-of nodes with other nodes  [    distance  myself]

    set heading towards-nowrap closest-turtle
    set pitch towards-pitch-nowrap closest-turtle
    bk repel-strength
   ]
end
```

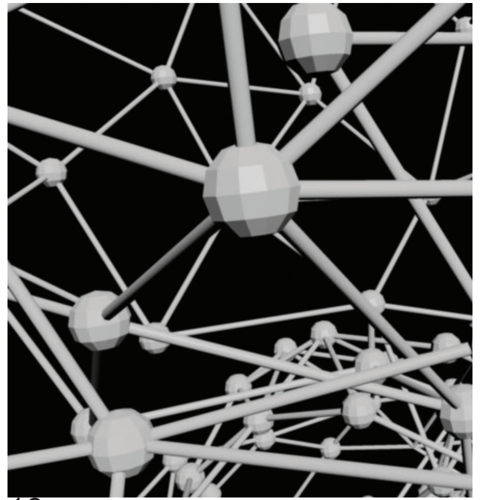
When running these simulations another thing that distinguishes this approach from geometry is apparent - rather than in the top down computational approach where a lot of works goes on until the “solution” in presented to you in one fell swoop, here the emergent organisation occurs as a visible process that sometimes has to be teased along with small tweaks of attract and repel values. Sometimes the whole thing descends into a chaotic muddle and cannot be retrieved without stopping and starting again. The algorithm for stitching the turtles together with line shaped turtles is typical of the bottom up approach.

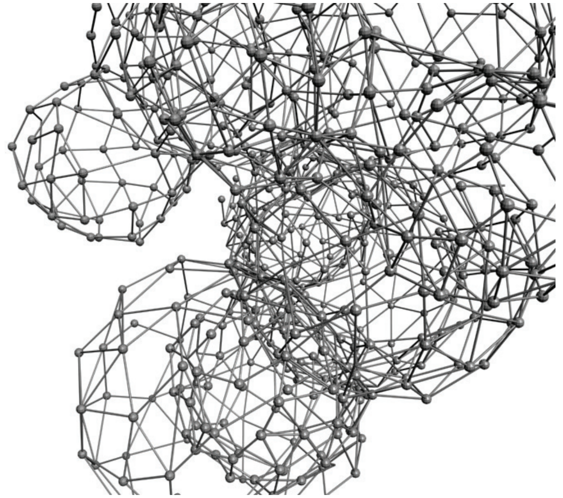
Once the closest turtle has been found we ask each node to create a link with it. the “link” turtle is a special feature of netlogo which behaves intelligently in that if the target node is already connected then this is not attempted again. in he course of a run the ‘nearest turtle’ will change so it is necessary to clear out existing links - this is easily accomplished with ‘*clear-links*’ (a special button - not shown - is needed for this)





further processing to develop the emergent distributions into varieties of forms






```
ask nodes

    [
    set closest-turtle min-one-of  other nodes [distance  myself]
       set heading towards-nowrap closest-turtle
       set pitch towards-pitch-nowrap closest-turtle
       bk repel-strength
       create-link-with closest-turtle

    ]
end
```

One might ask why this simple algorithm doesn't lead to links which cross the middle of the emerging spheroid, but remember that the attract and repel procedures have a habit of making sure that everyone's nearest neighbour is to be found on the 'shell'. Where several spheroids meet (as in the images facing) a certain amount of negotiation takes place with things jiggling about until most people are happy. The important point here is that no more code has to be written, this is an emergent outcome of the process provided for free by the dynamics of the system.

After everything has settled down (the 'emergent consensus' proposed at the start of this chapter) the self organised turtle configurations can be exported to other packages for further processing. In the images shown left the turtle coordinates are read into Autocad using a small visual basic script, and spheres and cylinders are drawn between the points of the nodes and links. Further processing to tile up the mesh and rendering can be achieved with your favourite CAD package.