Homework 3

Computer Graphics

Francesco Palandra 1849712

December 7, 2021

Contents

1 Procedural Noises (easy) **3**
1.1 Voronoise 3
1.2 Smooth Voronoi 4
1.3 Cell Noise 5

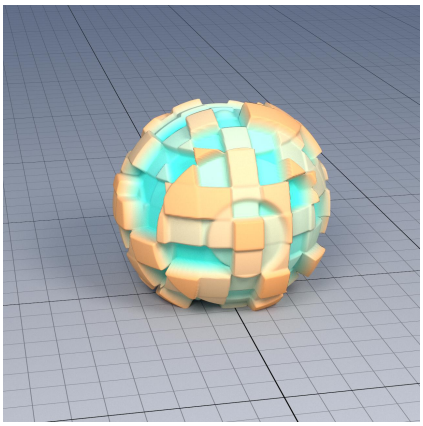
2 Sample Elimination (medium) **7**

3 Tree (hard) **9**

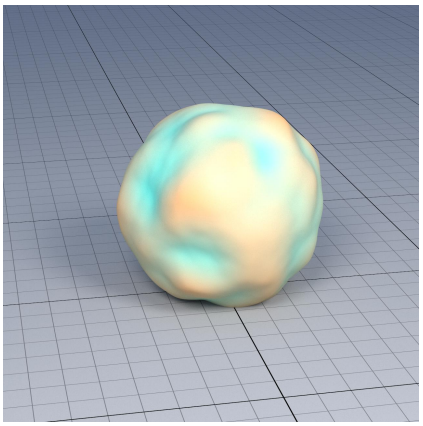
1 Procedural Noises
(easy)

1.1 Voronoise

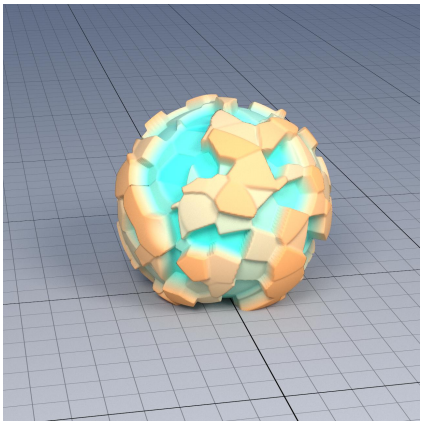
For Voronoi we splits the domain in a regular grid such that there's one feature point in each of the cells. That means that Voronoi patterns are based on a grid after all just like Noise, the difference being that while in Noise the feature originators are in the vertices of the grid, Voronoi has the feature generators jittered somewhere in the grid. Two parameters are introduced: u and v.
The first one controls the amount of jittering of the feature points, and v controls the metric. We can see as the u and v values varies we have different results, when we set v = 0 there is no blending and the grid is visible, whilst with u = 0 we obtain the grid effect. The code is from here, a change has been made to a value from 63 to 31 that was causing an overflow in a pow operation.



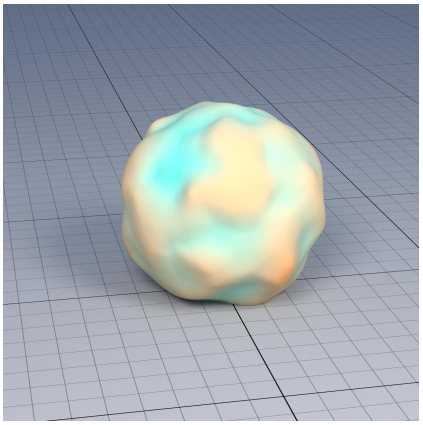
(a) Cell Noise u = 0 v = 0



(b) Noise u = 0 v = 1



(c) Voronoi u = 1 v = 0



(d) Voronoise u = 1 v = 1

Figure 1: Voronoise with different parameters

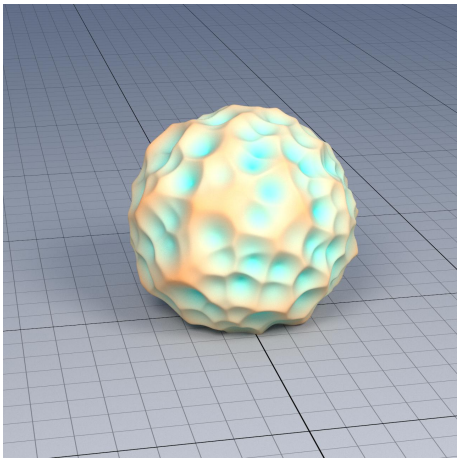
Algorithm 1 Voronoise algorithm

Input: $point\ x,\ int\ u, v$
Output: $flaotf$

```
p ← floor(x)
f ← fract(x)

k ← 1 + 31 · (1 − v)^4
va ← 0.0
wt ← 0.0

for (i = −2...2) do
    for (j = −2...2) do
        for (n = −2...2) do
            g ← {i, j, n}
            o ← hash(p + g) · {u, u, u, 1}
            r ← g − f + o.xyz
            d ← dot(r, r)
            w ← hash(p + g) · {u, u, u, 1}
            va ± w · o.w
            wt ± w
        end for
    end for
end for

return va/wt
```

1.2 Smooth Voronoi

In voronoi function the domain is partitioned in a grid, the cell in which the current shading point x lays is determined, the set of 3x3 cells centered in it is scanned, a pseudo-random point is generated in side each of these 9 cells, and the distance from x to the closest such point is recorded. In this implementation [here] does not generate the points in "domain" space, but in "cell" space: once the integer and fractional parts of the shading point are extracted and therefore the cell in which we are working identified, all we care about is what happens around this cell, meaning we can drop all the integer part of our coordinates away all together, saving many precision bits.



Algorithm 2 Smooth Voronoise algorithm

Input: *point x*
Output: $flaotf$

```
p ← floor(x)
f ← fract(x)

res ← 0.0

for (i = −1...1) do
    for (j = −1...1) do
        for (n = −1...1) do
            b ← {i, j, n}
            o ← hash(p + g) · {u, u, u, 1}
            r ← b − f + hash(p + b)
            d ← dot(r, r)
        end for
    end for
end for

return (1/res)^(1/16)
```

1.3 Cell Noise

One simple approach to estimate the real distance would be to take the function F2-F1 of the voronoi function, and evaluate its gradient, then divide F2-F1 by the modulo of the gradient. We first detect which cell contains the closest point to our shading point x, then do the neighbor search centered at that cell instead of the cell that contains x. The algorithm can be found on this link.

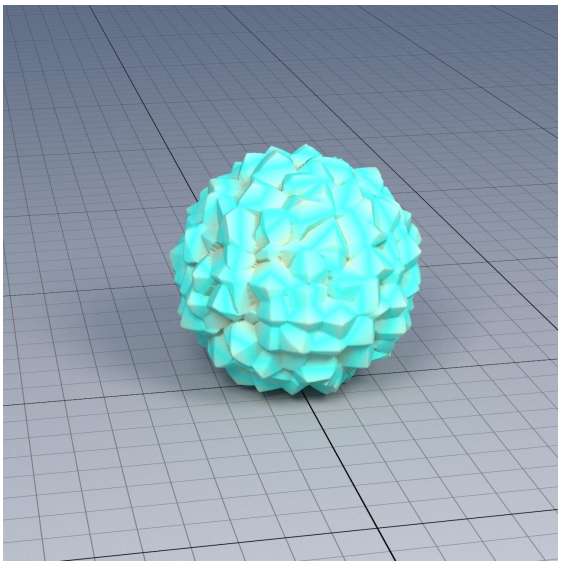


Figure 2: Cell Shading

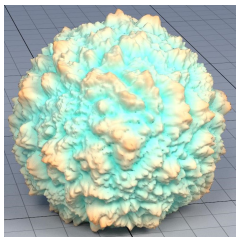
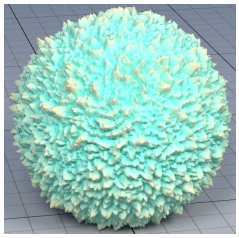
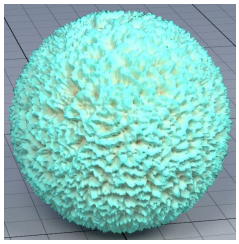
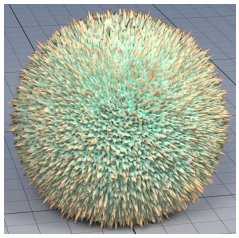
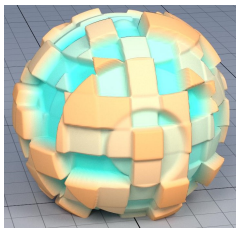
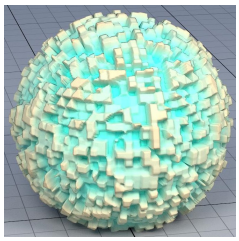
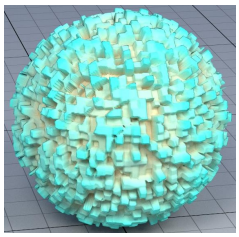
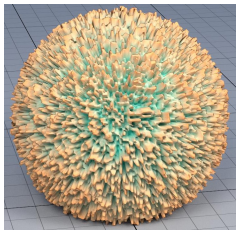
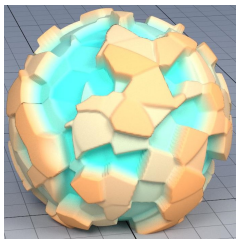
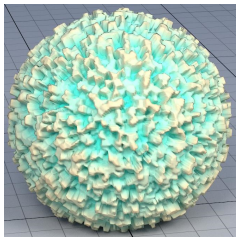
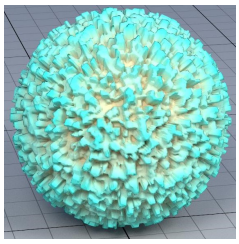
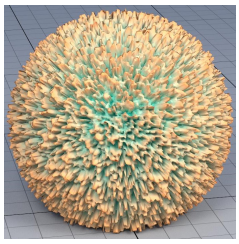
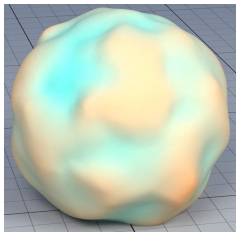
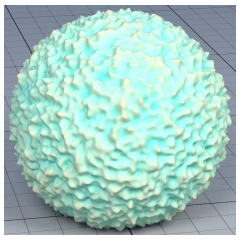
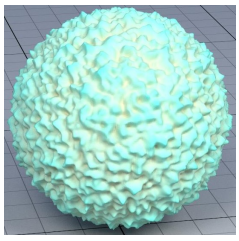
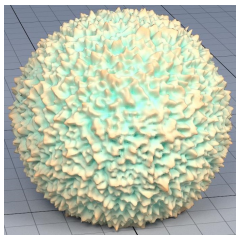
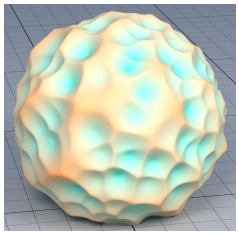
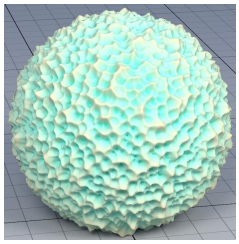
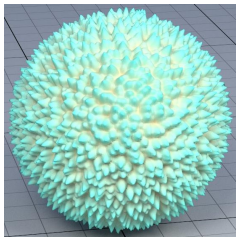
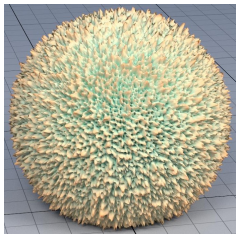
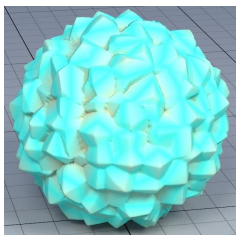
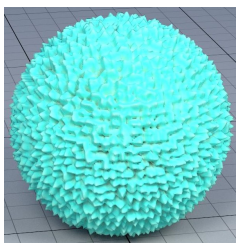
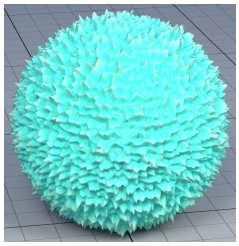
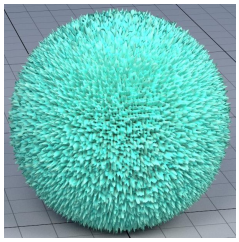
	Large	Normal	Inverted	Small
Turbulence				
Cell Noise				
Voronoi				
Voronoise				
Smooth Voronoi				
Cell Noise 2				

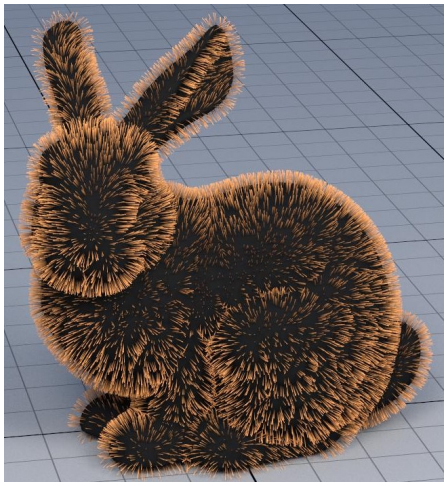
Table 1: Table of figures 2

2 Sample Elimination
(medium)

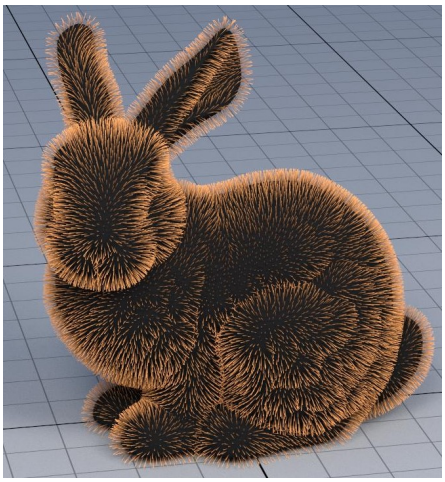
Sample Elimination is a method used to generate random position on a Poisson distribution for a better representation of natural object. paper]
Firstly we generate random point on the shape and we assign on each one a weight that depends on their neighbors within a certain radius $2 \cdot r_{max}$:

$$w_i = \sum_{neighborj} \left(1 - \frac{\hat{d_{i,j}}}{2 \cdot r_{max}}\right)^{\alpha} \quad \hat{d}_{i,j} = min(d_{i,j}, 2 \cdot r_{max}) \quad r_{max} = \sqrt[3]{\frac{A_3}{4 \cdot \sqrt{2} \cdot N}} \tag{1}$$

Then, a while cycle delete the point with the highest value of w until a desired number of points are left. Every time a point is eliminated from the shape, we have to remove his contribution of the weight of his neighbors.
The results shows a more natural disposition of the hair around the shapes.



(a) Input



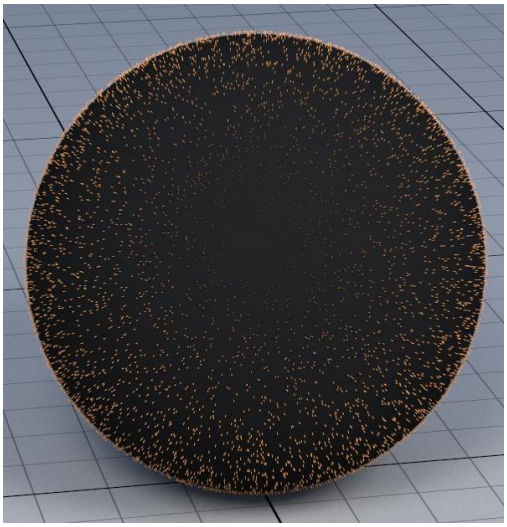
(b) Sample Elimination

Figure 3: Sample Elimination on 20000 random points

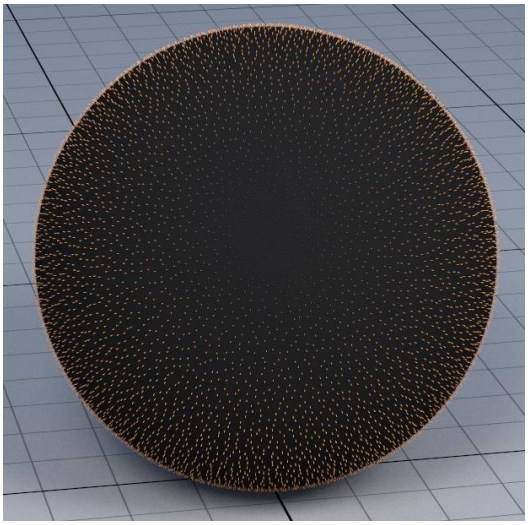
Algorithm 3 Sample Elimination algorithm

Input: *shape s*, *int num*
Output: *shape s′*

```
shape ← sample_shape(num · 5)
r_max ← ∛(A_3 / (4·√2·N))
calculate_weight(shape)
while (shape.positions.size() > num) do
    max ← shape.positions.pop_max()
    for (max's neighbors n) do
        n_i.weight −= (1 − d_{i,max} / (2·r_max))^α
    end for
end while
return shape
```

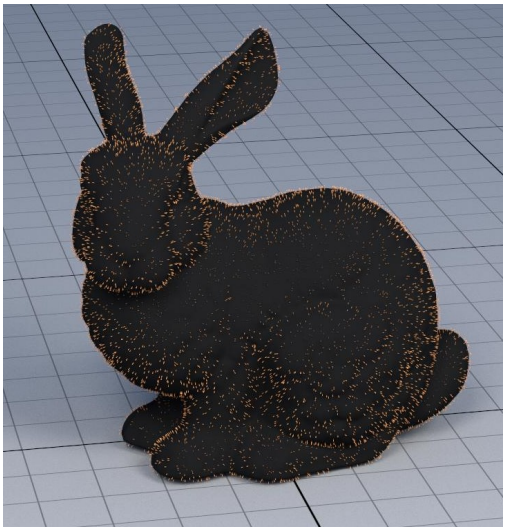


(a) Input

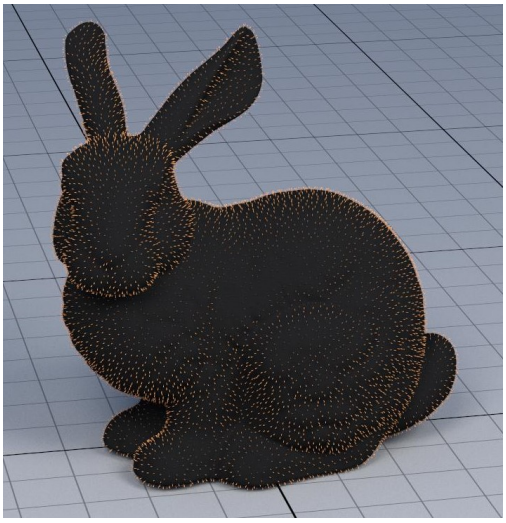


(b) Sample Elimination

Figure 4: Sample Elimination on 10000 random points



(a) Input



(b) Sample Elimination

Figure 5: Sample Elimination on 20000 random points

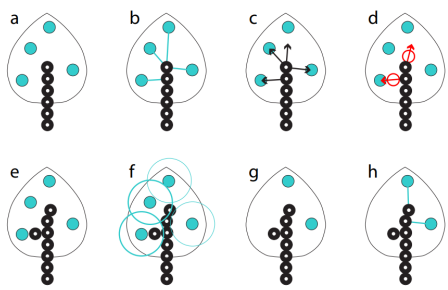
3 Tree
(hard)

For the implementation of the trees we followed method illustrated in this paper.

We first generate n random points around a shape, in particular in this code we use a distribution that results in a sphere with a little optimization on the final positions of the points [generating 3D tree]. In this code there is also the possibility to generate random point around an ellipsoid (there is a parameter for the shape and the lines of the code are commented). The trunk is now built starting from the ground on the same position of the center of the sphere, and then iteratively the tree grows.
On each iteration we calculate the points around the current branch within an attraction range. If no points are found a new branch is created with the same position of the parent, otherwise we connect each attracted point with the nearest branch.
Then for each branch with attraction points we calculate the direction of the son:

Figure 6: Tree with 800 points

$$\hat{n} = \frac{\vec{n}}{\|\vec{n}\|} \quad where\ \vec{n} = \sum_{s\ \in\ S(v)} \frac{s-v}{\|s-v\|} \tag{2}$$



and push it into the queue. Following the implementation of the paper we added also a killer range that set "inactive" the points of the sphere away from the branches by a factor of d_k. Lastly we draw on the scene the tree using a cylinder, on every iteration the thickness of the branch is reduced of a factor that depends also on the number of ramification.

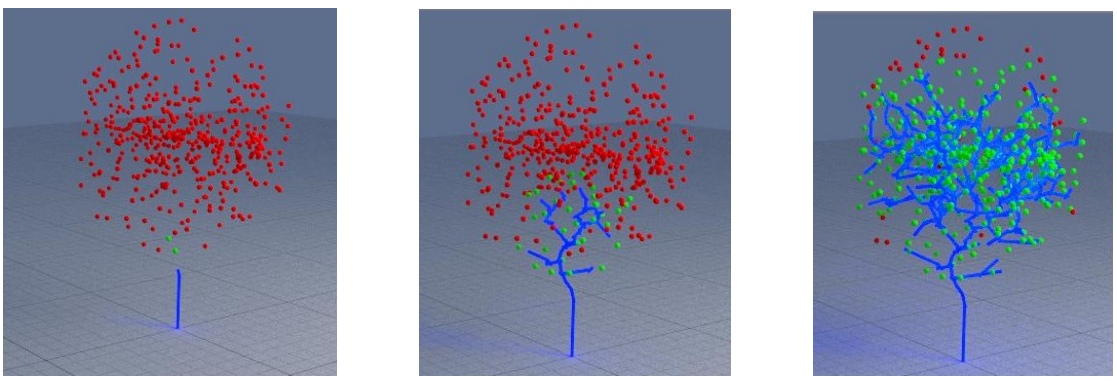


Figure 7: Application of the algorithm [4]

Algorithm 4 Tree Generation algorithm

Input: *scene s*, *tparams p*
Output: *scene* s'

```
leaves ← generatePoints(params.num)

parent.end ← { 0, 0, 0}
parent.direction ← { 0, 1, 0}

while (!leaves.empty()) do
    attr_points ← find_neighbors(parent.end, p.att_range)
    branches ← connect(tree, p.attr_points)

    for (branch in branches) do
        son.start ← parent.end

        for (point in branch.attractors) do
            direction += (p_i − son.start) / ‖p_i − son.start‖
        end for

        if (branch.attractors.empty()) then
            son.direction ← parent.direction
        else
            son.direction ← (direction / ‖direction‖) · rand3f · p.randomness
        end if

        son.thick ← parent.thick − 0.0001 · k
        son.end ← son.start + son.direction · p.step
        tree.push_back(son)
    end for
    parent ← tree[passo++]
end while
return scene
```