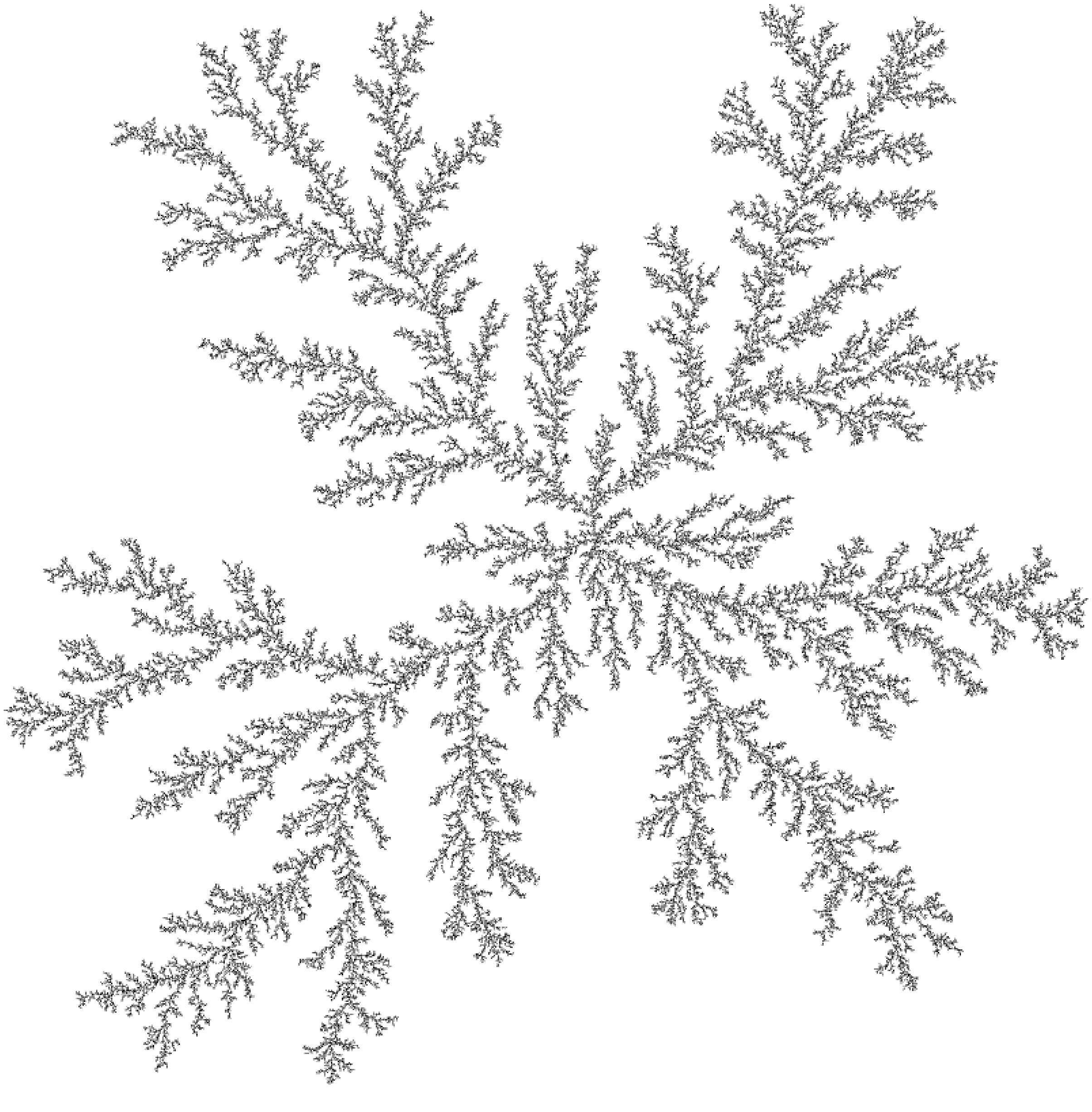
Diffusion Limited Aggregation



Author: André Offringa
andre@fmf.nl

Introduction

This article concerns a project about diffusion limited aggregation (or DLA). A DLA is a growing object consisting of particles that diffuse and aggregate to the object, much like how flocks of snow are created, and is simulated by a computer program. In such a simulation it is possible to determine some of the properties that DLA's have. In this report I'll tell about how I've implemented the simulation and I will try to discover some of the properties of DLA. Since I've made a somewhat non-trivial implementation, I will spent a great amount of this article on it.

The DLA-algorithm

A DLA consists of particles that attach to each other. How they attach is described here.
First of all, we start with a complete empty space and put one particle somewhere in it. Now, a second particle is 'fired' at an infinitely distance away from this particle. This new particle keeps moving randomly in the empty space until it collides with the already placed particle. At this moment, the particles attach to each other and are settled. Now a third particle is fired likewise, and the same procedure repeats itself over and over.

Implementation

Simplifications

To simulate the continuous algorithm on a discrete device like a computer with a finite amount of time, we need to make some changes to the algorithm. First of all; new particles aren't fired infinitely far away from the object, but rather are fired just outside the bounding circle of the object. The place on this circle is randomly chosen with a uniform distribution over the entire circle. Since the particle would have reached this circle anyhow, the outcome of the algorithm is not changed by this limitation (the time taken by the algorithm however *is* changed). Also, since we can't simulate a continuously moving particle, we simulate it by making discrete steps with a certain distance.

Moreover, to make things speed up a little bit more, when particles aren't interested in our neat DLA and start to drift away, we kill the particle at some distance and fire a new particle. This procedure actually changes the chances for attaching to the particles very, very slightly. This is because the chance for a new particle to get killed while reaching a certain particle in the DLA is for each particle in the DLA different. Therefore, there are certain positions which become a little bit harder to reach. Since we will see that particles aren't killed that often if we make the kill-distance large, we can do it nevertheless.

Another simplification we do to speed things up is to let particles 'jump' when they get outside the bounding circle of the DLA. Since, if the particle does not collide, the chances for the particle to reach some point on a circle around itself are uniformly spread (this is actually not true for small circles on a square lattice, as we will do later), therefore we can as well place it on a random position on the circle immediately.

I wrote two programs which implemented the algorithm with the above simplifications. In the first one, another simplification was made; particles were only able to move on a square lattice. Therefore, besides that a particle cannot make a step in any direction, it can only make a step in horizontal or vertical direction, and optionally it might move diagonally. This simplification makes implementing the algorithm much easier, and will result in a better performance. In the second program, which is discussed after the square lattice program, particles are off-lattice and *can* move in any direction. We will see that the square-lattice simplification changes the shape of the lattice.

Implementation of the square lattice DLA

Both implementations were written in the C++ computer language. Like in the previous project ("The Chirikov map") I used the library SDL to do the graphics for me. I will only mention the most interesting code or algorithms here, the program altogether can be find in the appendices.

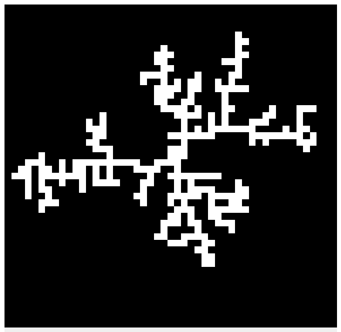
The data structure is fairly simple in the square lattice case; it suffices to have a two-dimensional array of integers of size *n x n*, where *n* is the maximum size that we want to reach. If we do not know the size on forehand, we can dynamically grow the array as needed. The integers represents each position on the square lattice. Each integer is initialized to zero, except for the element in the array which represent the point in the middle. This point is initialized to the number '*minValue*'. The value of each integer is incremented when a particles 'ends' at the position that integer represents. When the integer has reached '*minValue*', the particle on this position will actually become part of the DLA and new particles can attach to this particle.

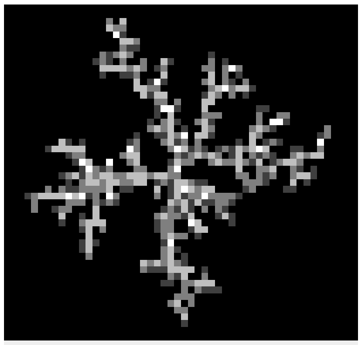
In a normal DLA, the value *minValue* will be one, which will cause particles to attach to the DLA the first time they collide. However, if we give *minValue* some higher number, the randomness of the algorithm will be 'averaged', and we can see the properties that the algorithm brings without having to generate huge DLA's.

The function in which a new point is proposed and which contains the DLA algorithm is `DLALattice::CalculatePoint(..)`, which can be found just below. It is made in such a way that the steps can be made with high performance; floating point calculations are avoided and used as few times as possible and square roots are avoided by taking the square of the radii. The last interesting thing this method implements is the use of a 'sticky value'. This value determines how much chance there is for a particle to stick to the DLA. This parameter, which normally will have a value of 1 (=particles will always stick, the value is actually pre-multiplied by `RAND_MAX` to avoid a floating point variable), can be used to change the shape of the DLA as well.

```
void DLALattice::CalculatePoint(int &x, int &y) const {
  GetRandomPointOnCircle(x, y, centerX, centerY, 1.1*(double) (radiusMax+5));
  int radiusKillSquared = radiusKill * radiusKill;
  int radiusJumpSquared = radiusJump * radiusJump;
  bool stick=false;
  do {
    while(!HasNeighbour(x,y)) {
      int choice = random() / (RAND_MAX / 4);
      switch(choice) {
      case 0: x++; break;
      case 1: y++; break;
      case 2: x--; break;
      case 3: y--; break;
      }
      int r = (x-centerX)*(x-centerX) + (y-centerY)*(y-centerY);
      while(r>radiusJumpSquared) {
        if(r>radiusKillSquared) {
          // Kill
          GetRandomPointOnCircle(x, y, centerX, centerY, 1.1*(double)
(radiusMax+5));
        } else {
          // Jump
          GetRandomPointOnCircle(x, y, x, y, sqrt(r)-radiusMax);
        }
        r = (x-centerX)*(x-centerX) + (y-centerY)*(y-centerY);
      }
    }
    stick = (!useSticky) || random()<sticky;
  } while(!stick);
}
```

Though most of the rest of the program is trivial, I would like to point on other thing out: since DLA's might become of size greater than the screen, I figured it would be nice to be able to shrink the data of the DLA. I therefore wrote a simple resampling routine, that either enlarges or shrinks the lattice to a preferred size. The shrinking resampling routine works by averaging the “color” (which is either completely white or completely black) of multiple pixels. The result is very nice, since it allows pixels to become gray, and by this gives us more detail using the same amount of pixels on the screen. Below are two images, the left being a 50x50 DLA visualized in a 50x50 image. The right one is a 100x100 DLA, visualized on the same 50x50 grid using the resample method that combines several pixels to an average gray value.


DLA without gray values

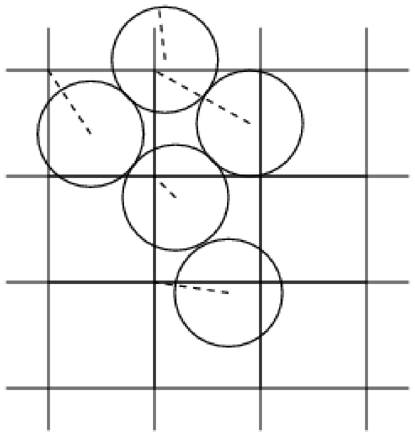

DLA with gray values

Implementation of the off-lattice DLA

The data structure that holds an off-lattice DLA is less trivial, compared to the square lattice case. In this case, we do not assume particles are square-shaped, but will represent each of the particles as a circle. Also, we will lose the simplification that particles could move only horizontally or vertically. Particles are moving freely through our area now, making steps in any direction.

A simple method for implementing the circle collection

We cannot simple store each particle in one element of the *n x n* matrix as we did in the square lattice case. However, with a minor change to the matrix, we can. Since we know that each particle, represented as a circle, is of constant diameter, say *d*, we can divide our area in squares of *d x d* and we know that each square can hold one circle as a maximum, since the circles are at least *2d* away from each other. If we store each square in an element in the *n x n* matrix, we're settled.


Mapping of round particles to a grid

To see if two circles intersect, which is done continuously when we fire a particle, we can check the square in which the new particle would be placed. If there is a circle in this square, we're sure that they will collide. If there isn't we check whether there are circles in one of the nine neighboring squares. If so, we check for each circle whether the distance between the centers of the new and existing particles is less then *2d*. If so, we know that they collide. If the circle intersects with neither a circle in its square, nor intersects with a circle in the nine neighbors, we can be sure that the circle does not intersect with any of the circles in our collection of circles.

If we store the particles like described, we have approximately the same advantages and disadvantages as in the square lattice case. The advantages are:

- $O(1)$ worst-case time complexity for testing whether a specified circle intersects with one in the collection (ie., it can be done in constant time)
- $O(1)$ worst-case time complexity for inserting a new circle
- Fairly simple implementation
- It takes at most 9 intersection tests to see whether a specific circle intersects with one in the collection

Limitations of this data structure include:

- Diameter of the circles has to be constant
- We cannot store intersecting circles in our data structure
- We have to declare the size of the grid on forehand, or change the size of the grid when it can not hold its circles any longer
- Iterating all circles takes $O(n^2)$ time, where *n* is the width of the grid. Since not all squares will contain a circle, this will usually mean we spent more time than necessary. In the case of our DLA for example, if we already know that our DLA has a dimension of 1.5, the DLA will be of diameter $m^{(2/3)}$ (*m* is mass of the DLA). Since our grid has to be at least of this size and iterating will cost quadratic more, the iteration will cost $O(m^{(4/3)})$
- The same calculation holds for the worst-case space requirement: this will also be of $O(m^{(4/3)})$ in the case of our DLA.
- To achieve the data structure's space requirement, we assume that we know that the circles will be stored within a certain bounding box.

Since each newly fired particle is being tested for intersection with all other particles until it actually intersects and is attached to the particle, the efficiency of testing the intersects *seems* to set the performance of our program. Is this true?

The distance that each particle travels before hitting another particle grows, because of the growing diameter of the bounding circle on which new particles are fired. Because of our jump and killing strategy, particles tent to move towards the DLA. Later we will see that reaching the DLA costs approximately $O(m)$ steps[1]. Therefore, the insertion operation can cost $O(m)$ time without changing the time complexity of the entire algorithm.
Also, if we do not perform a circle-iterate operation more than once every $m^{(1/3)}$ insertions, the iteration will not influence the algorithm. A circle-iteration operation occurs when we save the circles to a file or when we enlarge the grid. Therefore, neither should we do more than once every $m^{(1/3)}$ particles. This is not a problem though, since if we make the grid two times larger in horizontal and vertical direction every time the DLA is about to overgrow the grid, we perform the operation once per 0,63m on average, because the diameter of the bounding circle of the DLA will only be doubled every 0,63m particles (and once every 0,63m insertions is less often than once every $m^{(1/3)}$ insertions, for all m>2 approximately). Therefore, the test for intersection does indeed set the performance for our program.

It is interesting to note that, if we would have increased the size of the grid every c particles, where c is a constant, we would affect our time-complexity, since performing an $O(m^{(4/3)})$ operation every c insertions implies that inserting will cost $O(m^{(4/3)})$, by which the insertion will cost more than the number of steps we have to take each insertion.
Note that the above calculations are true in the square lattice case as well.

We can conclude from this that the above data structure is almost ideal for our problem. The only thing that is not so ideal is the fact that we need $O(m^{(4/3)})$ space. Considering the fact that simulating DLA's containing one billion particles, which will probably be far beyond the maximum amount of particles we can simulate considering the time this is going to cost, and considering the fact that a DLA of one billion particles will cost $(10\text{^}9)^{(4/3)} = 10^{12} \approx 1$ terrabyte of space – and this is still more or less technically possible, this is not our first problem.

In short: this data structure is ideal for our problem. I didn't use it however... Why not?? Well, I preferred something... ehm.. more difficult. I tried to implement the circle collection in such a way that *some* of the previously called disadvantages disappeared, without influencing the time complexity... much. The data structure and intersection algorithm I used will actually perform better in *some* situations In the upcoming paragraph I will describe this algorithm.

1 Since the direction of the steps are based on random numbers, this is not a worst case performance, but rather an average performance. In worst case scenario, the algorithm will never end.

A less simple method for implementing the circle collection

In our second method we will use a completely different technique. In this implementation, the circles are sorted and stored in an AVL-tree. To do so, we need an ordering in which circles that are closely together remain closely together. This is 'more or less' achieved by ordering the circles lexicographically first on ring and then on angle.

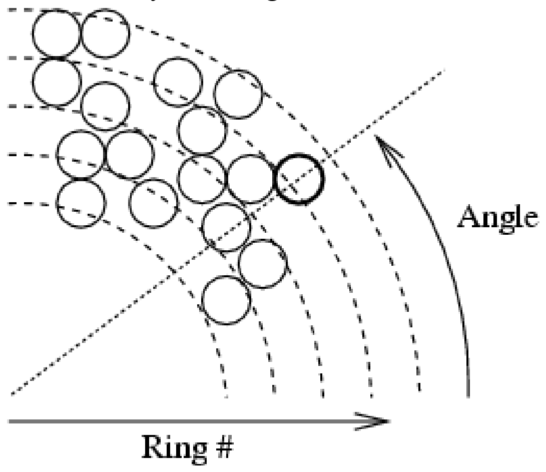
A ring is an area which contain all points that have a distance l to the origin, with the constraint that:

$$d \times ring \leq l < d \times (ring + 1)$$

With:

d	– the diameter of the circles stored in the collection
$ring$	– the number of the ring
l	– the distance of a point to the origin

(To avoid confusion: we will only call the circles we store actually 'circles' and the circles that separate rings are the ring limits)



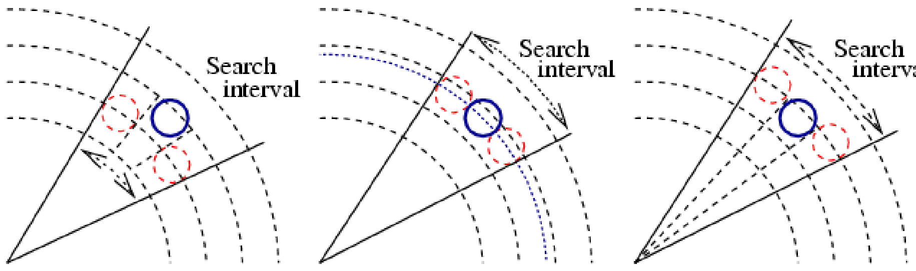
Principal of the ring and angle of a circle

Thus, a circle with center point in the origin is in the first ring (number zero) and a circle with center point a distance of 1*d* away from the origin is in the second ring (number one). We define the angle of a circle as the angle between a horizontal line through the origin and a line that intersects the circle through its center point. Now we say that circle **A** > circle **B** if and only if:

- ring(**A**) > ring(**B**) or;
- if ring(**A**)=ring(**B**), then angle(**A**) > angle(**B**).

In this ordering, circles that are in the same disc and are close to each other, will be ordered close to each other. However, circles that are in different rings are not ordered close to each other. Therefore, when we search for an intersection, much like we had to search in 9 neighboring squares in the previous method, in this method we have to search in three rings; in the ring that the circle we test is in and in the next and previous ring.

So, in which interval do we seek for intersections? We start seeking through our ordered list of circles at an angle that is calculated by moving the distance of two times the circle diameters along a certain ring limit. For the ending angle the same principal holds, but the ring limit is followed in the other direction. The following picture visualize the interval for each ring that we seek through:



Note that the angles are not the same for each ring. It is quite simple to express the angle using the geometric functions (*arc*-)*tan*, *sin* and *cos*. It is also possible to express it analytically *without* the use of these functions, but this yields an enormous formula. I used the geometric functions therefore.
Note also that there are some exceptions around the origin and around a zero angle.

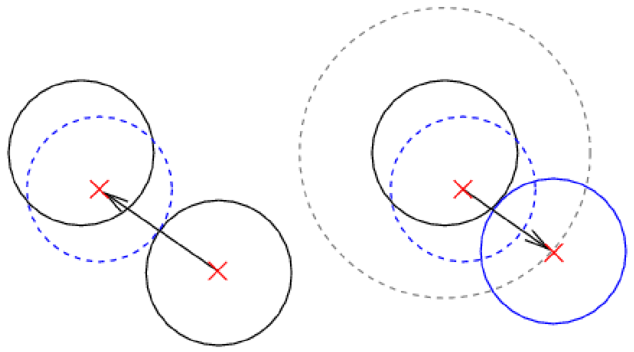
We now have enough information to store the circles in an ALV-tree. Implementing an ALV-tree was less trivial then I though (especially since I've implemented the tree avoiding any recursion in the insertion and seek routines – for which I have no good reason though. It might be a little bit more efficient). The resulting algorithm has the following properties:

- $O(\log n)$ worst-case time complexity for testing whether two particles intersect (this is caused by the fact that we have to traverse the tree with height $\log n$ from root to bottom to find the first circle that is higher than the start angle)
- $O(\log n)$ worst-case time complexity for inserting a new particle
- It takes at most 9 intersection tests to see whether a specified circle intersects with one in the collection
- Space requirement is $O(n)$
- Iterating all circles takes $O(n)$
- In contrast to the first algorithm, we do not have to assume that the circles will be stored within a certain bounding circle: we can insert a particle in the collection at a long distance from the origin without affecting properties of the collection.
- In contrast to the first algorithm, we can store intersecting circles in our data structure. However, if we do so, the 9 intersection tests-maximum is not hold any longer.
- Remark that using this method we can also make an intersection graph of n equal-sized circles in $O(n \log n + i)$ (where i is the number of intersections) expected time.

So we gained in the space requirements, but lost in the intersection test. We also gained a little in generalization: comparing the two algorithms yield that the second method is a little bit more general, especially by the fact that we can add circles at arbitrary distance from the origin. We do not need this for our DLA simulation, though.

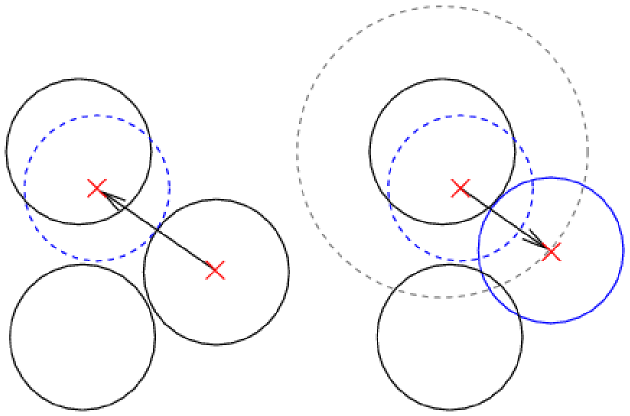
<u>Intersection handling</u>

So now we have the data structure, but what do we do with the knowledge that two particles intersect? Remember that we let it make discrete steps for simplification, but we are simulating a continuously moving particle. A particle can therefore never step on another particle, but only touch a particle and attach to it. The solution is simple: if we make a step and find an intersection, we move the particle back into the direction where it came from until it only touches the other particle.



The above image shows how this position is calculated: we first draw a line L between the old position and the new position (on which it intersects) of the particle. Then we draw a circle C that has twice the diameter of an ordinary circle around the intersecting circle. We now place the particle on the intersection of L and C.

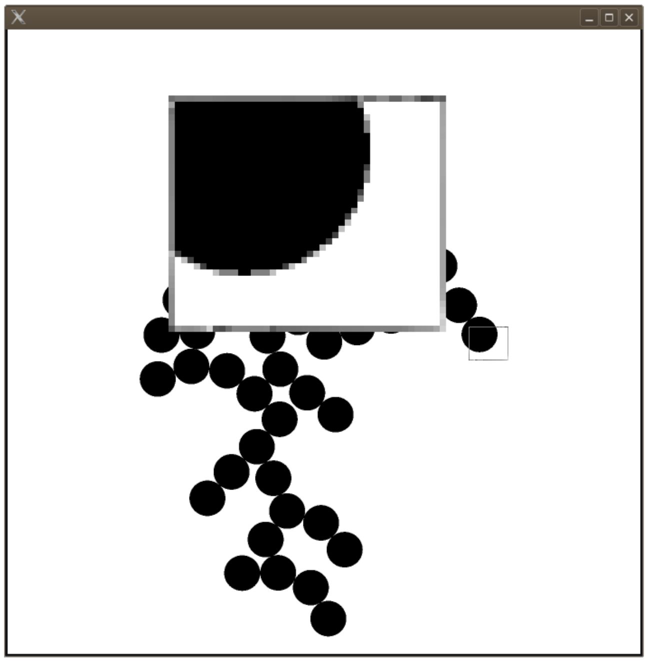
There is one troubling factor though; on our new position, the circle might once again intersect with yet another circle, as illustrated in this picture:



Since it seemed that this does not occur that much, I solved this problem by just placing the particle back to the position it came from and make a new step.

Drawing

The last step we need to implement is drawing the image to the screen. We can chose to draw all circles one by one, but this will take $O(n)$ time, with quite a high constant factor, since drawing a circle involves some calculations. This drawback is only visible with huge DLA's though. However, another draw-back is that antialias, a method which makes circles more smooth by adding gray pixel on the boundaries of the circles, is not very easy to do with this method. The method I used is a slow one, but more constant when n increases, and allows antialias. Also, it is very easy to implement.



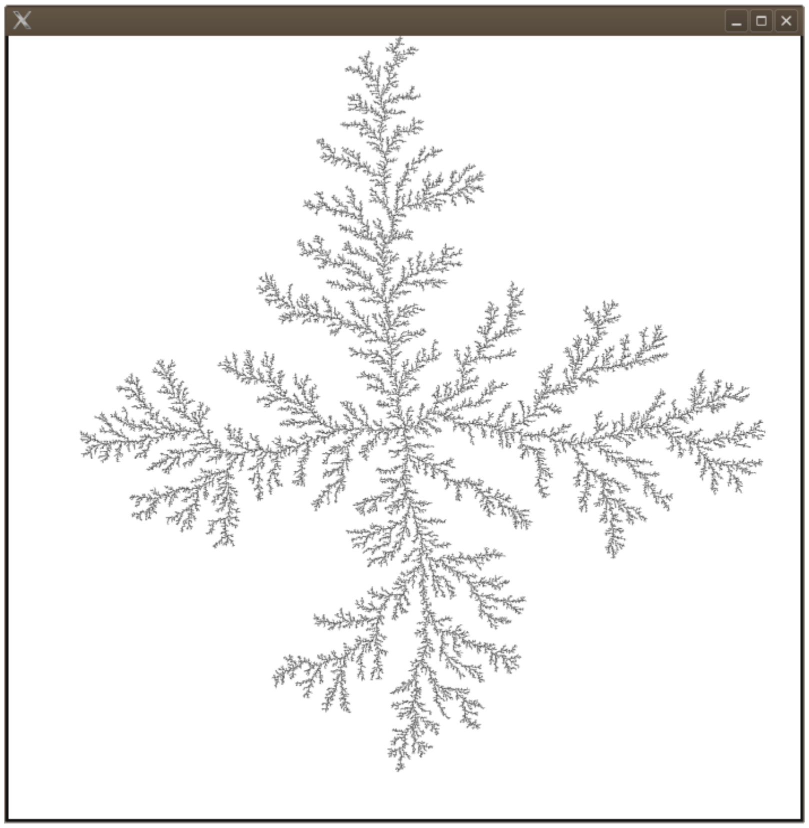
Part of a circle zoomed in, demonstrating antialias

We iterate over all pixels in the screen and divide each pixel in a x a sections, where a is the antialias depth. For each section we check whether the center point of the section is inside a circle. If so, we increase the brightness of the pixel with 100/(a x a) percent. This way, a pixel that is only half inside a circle, will become 50% white (thus gray).

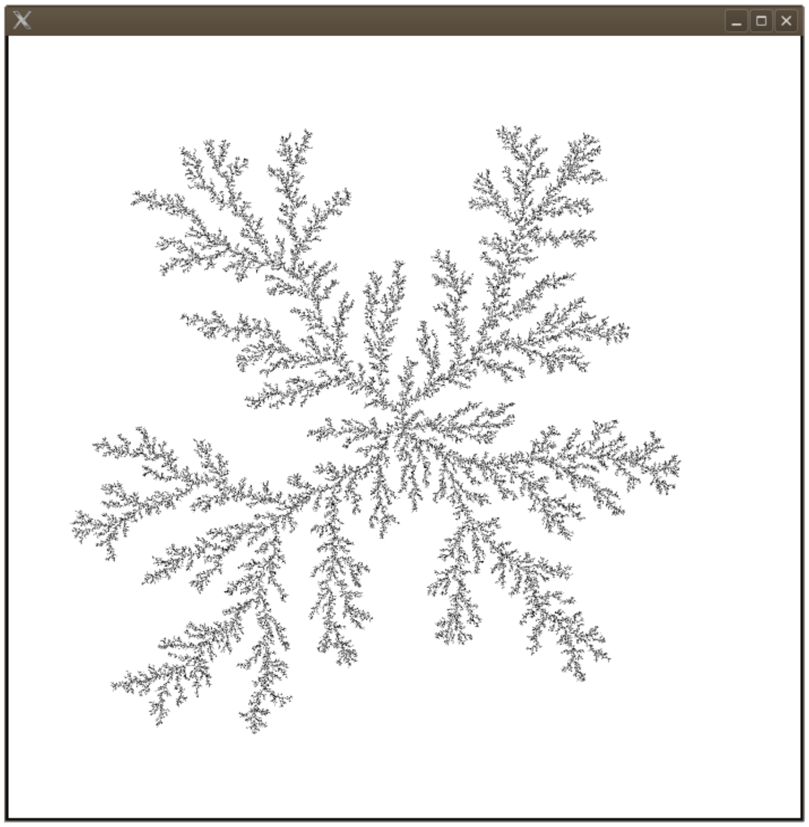
This algorithm is O(a x a x *width* x *height* x *log n*). However, since *a, width* and *height* remain constant, and only n is increased. Theoretically, the algorithm is therefore of $O(\log n)$. This works very well in practice: though drawing a DLA with mass 40 takes already 3 seconds, drawing a DLA with mass 140.000 takes only 2 additionally seconds.

Results

To the right you can see two images. The first one is generated with the square lattice method and the second one is generated with the off-lattice method.


Square-lattice DLA with mass 243857 and dimension 1.605

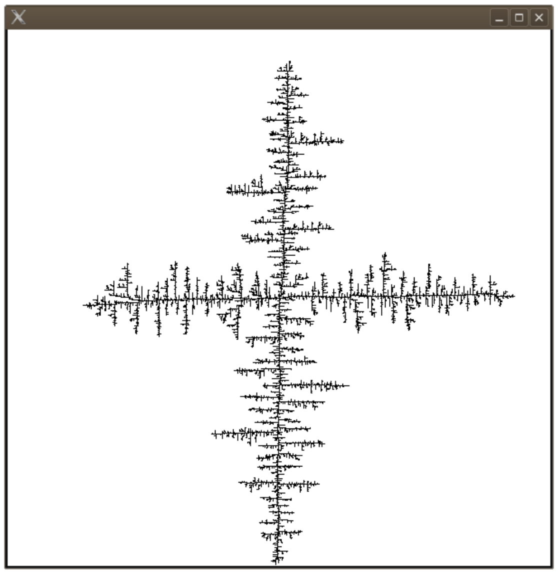
The square lattice method image clearly has a different structure, compared to the off-lattice one. The main branches of the square lattice tend towards following the squares either horizontally or vertically. In the off-lattice method, branches grow in any direction. Therefore, it seems that building the DLA on a square lattice is not a very good simplification.


Off-lattice DLA with mass 141000 and dimension 1.633

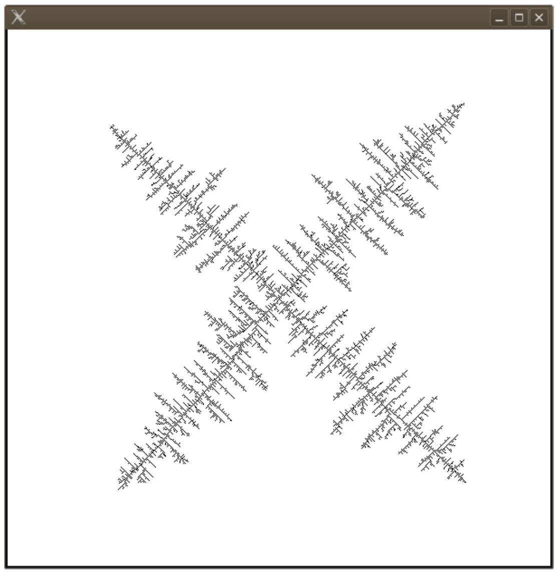
The dimension of the figures is not very differing though; in all simulations, the dimension reaches a value of approximately 1.6, but because of the randomness of the algorithms, the dimension keeps varying. I was hoping that creating a DLA of a tremendous size would allow me to determine the true dimension with some decimals accuracy, but if we calculate the dimension using the bounding circle and by counting the particles in it, the dimension does not converge. If, however, we draw a circle in the figure, say half of the size of the bounding circle, and count only the particles that lie within this circle, the dimension becomes more constant.

When we use this more sophisticated method for calculating the dimension, the square lattice DLA converges to a dimension of 1.65 and the off-lattice DLA converges to a dimension of 1.66.

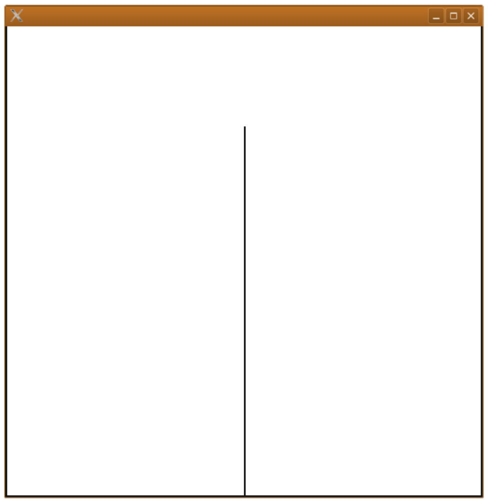
When we enlarge the effect of the square lattice by setting *minValue* to 6 (described in the methods section), so that a pixel only attaches to the DLA if its run ended six time on that position, we'll see that all branches tend to grow straight either in vertical or in horizontal direction (see figure on the right).


Square-lattice method (minValue=6, mass=12250, dimension =1.50, part dimension=1.60)

The more sophisticated dimension calculation for this figure seems to converge to a value between 1.60 and 1.65, but I have not simulated DLA's with a mass as high as in the previous example (ie., this is just an estimate – this because the previous figures both took a couple of days to calculate).


Square-lattice method with minValue=6, allowing particles to connect diagonally (mass = 8762, all dimension = 1.44, part dimension = 1.53)

In the image on the left we see what happens if we use eight connectivity instead of four connectivity (*minValue* remains 6). This means that we allow particles to attach to each other in horizontal, vertical and diagonal direction. The dimension of this figure is quite lower as in the previous figure; the sophisticated method yields 1.53 in this case. This is intuitively not so strange, since we allow particles to connect to each other when in fact they are still farther away from each other. The density would decrease therefore, and thus the dimension decreases as well.



Next thing we do is attaching the particle to the DLA on a position that has the highest chance of being the next position. We can do this by setting *minValue* to infinity, or more practical, a high number, and after each time that we attach a particle, we reset all integers in the matrix on which no particle is placed to zero. Using this method, we get the fabulous picture on the right. The program reports a dimension of one for this figure, which is indeed generally true for a straight line. If we calculate the dimension using the bounding circle (instead of some smaller sub-circle), we get a slight error: a dimension of 0.98. This is due to the fact that the line is not centered in the figure (and the bounding circle *is*), and reaches the bounding circle only on one side of the bounding circle.

If we change the connectivity to eight-connected, we always get a diagonal line.

Time complexity

In this chapter we look at the time-complexity of the methods.

We already saw that the inner loop of the algorithm performs particle steps until we reach another particle. It is actually quite hard to proof how many steps are necessary on average: we have to take into account that the bounding circle grows and the particles are fired from another place, but this is partly compensated by the fact the DLA itself grows as well. To make things even more complex, we kill particles and make jumps if they drift away to far.

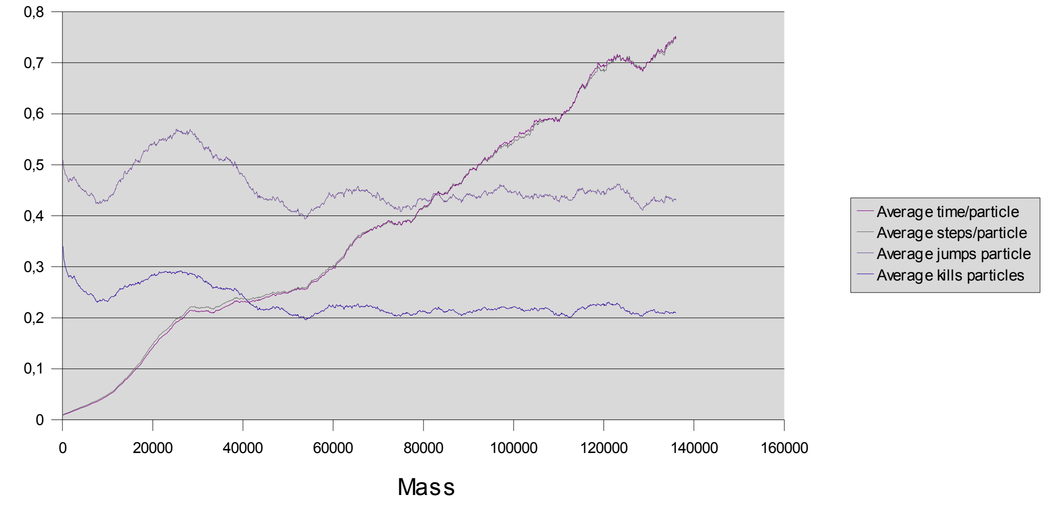
Since our distance to the center of the DLA increases, and since particles will on average have to cross a distance equal to its distance to the center, the average distance to cross is equal to the distance of the start position to the center of the DLA. Since the distance to the jump-position is also linearly, we know that having to step a distance linear to the bounding circle of the DLA can be expected.

However, we want to express our complexity in the mass of the DLA and not in the diameter of the bounding circle, thus we have to convert between these. Since we know that the dimension of the DLA is on average at least 1.5, the radius of the DLA will linearly follow the equation $m^{(2/3)}$.

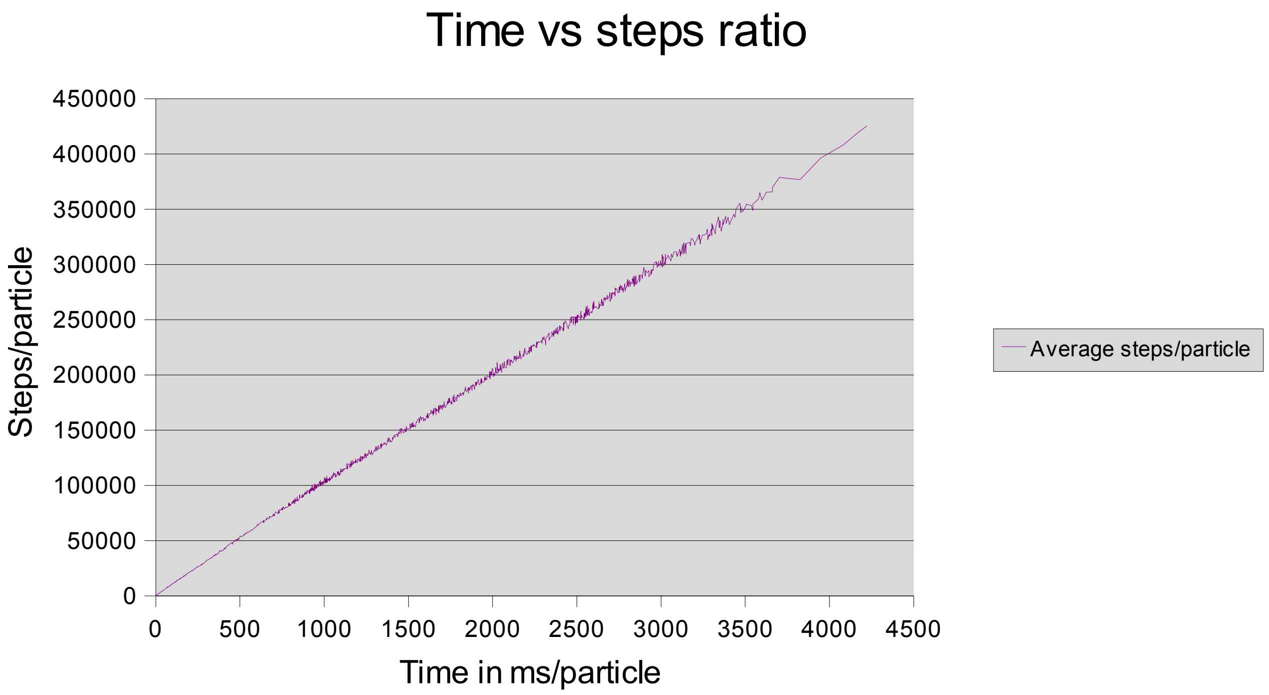
Now, if we note that for a particle the chance to increase its distance to its start position is around 50% per step, we can roughly estimate the average steps needed to cross a distance *d* with $2^d/d$ steps, since the chance for a particle to reach the distance *d* in *d* steps is $(0.5)^d$. Discovering this is actually quite a shock, since this means that the complexity of DLA is non-polynomial.

If we combine the found values above we get a complexity of $O(2^{m^{2/3}})$. It is a very pessimistic calculation though, since the chance that a particle will travel on average to a place with a distance the same as that from the center of the DLA to the start position of the particle is not very likely: it well probably connect to one of the branches that has grown in his direction. The branches to which the particle is most likely to connect have a length that is roughly equal to the diameter of the bounding circle. If we make sure that a particle starts jumping after traveling a *constant* amount of steps in the wrong direction instead of a number of steps that is linearly to the radius of the bounding circle as well, the number of steps would become constant.

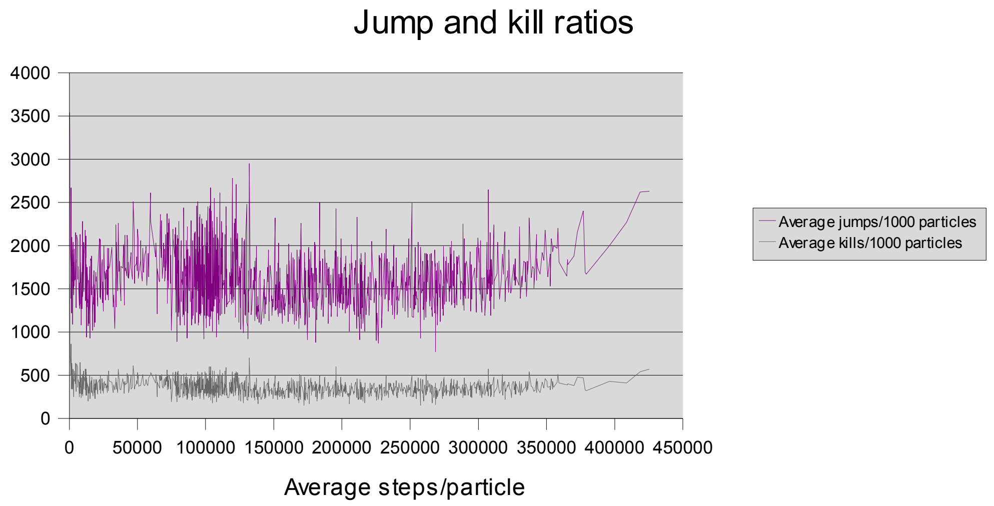
The results though show that this isn't true either; the number of steps seem to increase linearly with the mass of the DLA. The number of kills and jumps *are* constant though. This would mean that on average a particle would travel log(*m*) distance. I can't explain this.



Note that the numbers on the vertical axis do not represent anything; all values have been normalized and averaged. The relation between jumps/kills is also not correct.



On this chart, we can see that my invented algorithm is doing very well. We can make 100 intersection tests (test whether a given circle intersects with any circle in the collection) per second during the start, and this remains so even when we've reached a mass in which each particle takes 450.000 steps (this is the case at mass 140.000). This would indicate that our intersection test is $O(1)$, which is not true. It seems that traversing down the tree, which costs $O({}^2\log)$ and ${}^2\log(140.000)\approx 17$ operations, is still an insignificant operation compared to calculating whether two circles intersect. That's also why the square lattice case is about 100 times faster; it requires hardly any floating point operations.



This last figure shows the exact relation (without normalization) between steps made and the amount of kills and jumps are made. A particle is killed once every two particles, and about two jumps are made every particle. This seems reasonable.

Conclusions

First of all, we can conclude that simulating an (off-lattice) DLA with a square lattice method is quite a brutal simplification: the dimension as well as the structure of the DLA is changed. The dimension doesn't change significantly, yet the structure does – as we have also seen when we enlarge the effect of the square lattice by using the *minValue* principal. The *minValue* noise reduction technique is therefore a good method to make certain properties more visible. If we don't want these side effects, we can implementing an off-lattice method. This is however more complicated and much less efficient. By evaluation of the results we can conclude that the simplifications made for the off-lattice DLA don't change the DLA too much.

How efficient my implementation is in comparison to other methods is hard to say. For DLA's the size of which I have created, the implementation was performing rather well. The logarithmic time complexity did not make any difference, however, since for each intersection some additional floating point calculations are made, the formerly described method would probably have been two or three times faster. That method would also require more memory though. Another conclusion based on the experiments is that constructing an DLA seems to be possible in $O(n^2)$.

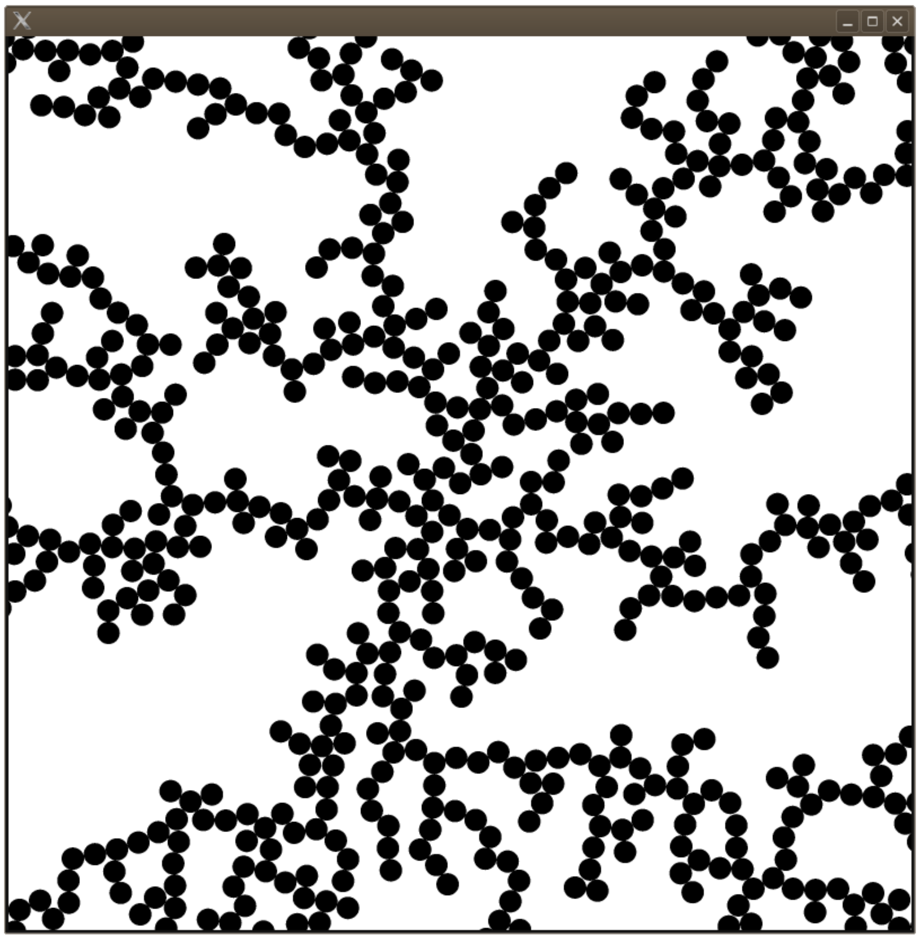
Other algorithms that could have been used are kd-trees with fractional cascading or priority search trees, which both can be constructed in $O(n\log n)$. Intersection founding can be done in $O(\log n)$. Though, both data structures are at least as complicated as my implementation, as well as that fractional cascading requires $\log n$ times extra memory.

Appendices

With this report come two projects with source files that generated the DLA-pictures for this report; a square lattice implementation and an off-lattice implementation. Each project can be compiled with a script called *c*. To compile the files correctly, the SDL library has to be installed.
Both projects come with some additional executables:

- noiseReduction – shows the principal of the *minValue* problem.
- show – shows saved dla's
- consoleIterator – creates a dla without graphical output and saves it
- offLatticeMain – starts to create an off-lattice DLA and shows progress on the screen
- zoom – zooms in on a generated DLA

All programs have only been tested on Linux.


Result of zoom