A Parallel DLA Fractal Generator?

Abstract

In this document, we will describe the process of creating a parallel DLA (diffusion-limited aggregation) fractal generator algorithm using the Delphi programming language. We begin with a definition of the DLA fractal and construct a simple (non-parallel) algorithm that will serve as the prototype for a DLA fractal generator. Then we will make the algorithm parallel. A complete parallel DLA fractal generator application with GUI is designed, and the performance gain due to its parallelism will be investigated.

Table of Contents

Introduction

Diffusion-limited aggregation (DLA) is a model physical process where particles of some species are undergoing random walk until they hit a solid object (in general, an aggregate of similar particles); at such an instance, they stop and become a part of the aggregate. The aggregates formed after many such instances are called DLA fractals (or, Brownian trees), since the corresponding point sets resemble mathematical fractals. The DLA model is applicable to many chemical and physical systems, such as systems with electrodeposition and mineral deposits.[1]

DLA fractals are easily generated algorithmically. In this document, we will describe a simple algorithm for the creation of a two-dimensional DLA fractal. We will also implement a parallel version of it in the Delphi programming language.

The DLA Model

The algorithm we will use is very simple. As our basic data structure, we will use an ordinary square raster image ('pixmap', or 'bitmap'), of dimensions $d \times d$, say. Each pixel represents a possible particle position, and a pixel is black if a particle is located at the site and white otherwise. Initially every pixel is white, except for an initial 'seed' from which our aggregate can form. Then we iterate one particle at a time. We place the particle at some random location inside the bitmap, and let it perform a random walk on the bitmap. More precisely, if the particle is at (x, y) at one step, then it will move to either $(x - 1, y)$, $(x + 1, y)$, $(x, y - 1)$, or $(x, y + 1)$ in the next step, with equal probability. Two special things can happen at the new location: (1) The new location might actually be invalid, that is, *outside* the bitmap. If this is the case, we simply start all over again with a new particle. (2) The new location might have a neighbouring pixel that is black. In this fortunate case, the particle will stick to its new location, and so we colour this pixel black. Then we start all over again with a new particle.

The most typical example of a DLA fractal, perhaps, is the case where the seed is a single particle (or small and roughly circular cluster) in the centre of the container, which is a *circular disk*. Indeed, this is what one might expect to see in an electrochemical experiment performed in a petri dish or a similar container. This is also the main type of DLA fractal that we will investigate. Thus, we let the initial bitmap be completely white except for a black pixel at $(\lfloor d/2 \rfloor, \lfloor d/2 \rfloor)$. Now, a bitmap is intrinsically rectangular (and, in our case, square), and so we will have to impose a circular boundary manually. Specifically, we will imagine that the boundary is the *inscribed* circle

$$C_{\mathrm{bdry}}: \quad (x - \lfloor d/2 \rfloor)^2 + (y - \lfloor d/2 \rfloor)^2 = \lfloor d/2 \rfloor^2.$$

When we insert a new particle in the system, we will place it at a random point on C_{bdry}, that is, at

$$(x, y) = (\langle \lfloor d/2 \rfloor + \lfloor d/2 \rfloor \cos\theta \rangle, \langle \lfloor d/2 \rfloor + \lfloor d/2 \rfloor \sin\theta \rangle)$$

where $\theta \in [0, 2\pi[$ is picked at random and angular brackets denote rounding to the nearest integer. In addition, we will consider the particle as 'gone' if it wanders across this imaginary circular boundary; then we will start on a new particle.

[1] For more background on DLA, please refer to the Wikipedia article at http://en.wikipedia.org/wiki/Diffusion-limited_aggregation.

By its very nature, a DLA simulator is a very computer-intensive algorithm, since the particles tend to wander for a very long time in white regions of the bitmap. However, there is a very simple way to improve the performance significantly. Indeed, at any step, let r be the maximum distance from the central point (the initial seed) to any black pixel in the bitmap, and let

$$C_r: \quad (x - \lfloor d/2 \rfloor)^2 + (y - \lfloor d/2 \rfloor)^2 = (r + \epsilon)^2$$

where ϵ is any (small) positive number. We could introduce a new particle at a random point on C_r instead of at a random point on C_{bdry}. Then the distance that the newly inserted particle needs to wander before it (possibly) hits a black pixel is much smaller, and so, on average, a smaller number of steps is required before the particle either escapes the scene or hits a black pixel. Thus, this *greatly* improves performance! To implement this performance improvement, we could use

$$(x, y) = (\langle \lfloor d/2 \rfloor + (r + \epsilon) \cos\theta \rangle, \langle \lfloor d/2 \rfloor + (r + \epsilon) \sin\theta \rangle)$$

as our initial particle position, but for practical reasons we will instead use

$$(x, y) = (\langle d/2 + R\cos\theta \rangle, \langle d/2 + R\sin\theta \rangle)$$

where, at any time,

$$R = \max(r, 6).$$

Thus, if $r < 6$, we will use a circle of radius 6. This is more practical since the grid of pixels is discrete, and so we cannot really consider any exact circles on the bitmap.[2] However, this 'varying-radius' optimization will alter the overall shape of the fractal, and make it less symmetric (why?).

Implementation

Although we could use a raw

```
FBitmap: array of array of TColor;
```

as our basic bitmap data structure, we will use a slightly more fancy (but still very simple) class:

```
interface

type
  PBitmap = ^TBitmap;
  TBitmap = class(TObject)
  public
    type
      TPixel = type TColor;
    const
      BLACK = TPixel($00000000);
      WHITE = TPixel($FFFFFFFF); // SIC! It is important that the
                                 // reserved byte is also $FF.
  private
    FBitmap: array of array of TPixel;
    function GetHeight: integer;
    function GetWidth: integer;
    procedure SetHeight(const Value: integer);
    procedure SetWidth(const Value: integer);
```

[2] We could probably choose a slightly smaller number than 6, but then we need to think about the extreme cases. Now we are sure that no rounding errors will cause us any problems.

```
    function GetPixel(X, Y: integer): TPixel;
    procedure SetPixel(X, Y: integer; const Value: TPixel);
  public
    procedure FillWhite;
    procedure FillBlack;
    procedure SetSize(const AWidth, AHeight: integer);
    function PixelExists(const APixel: TPoint): boolean; overload;
    function PixelExists(const AX, AY: integer): boolean; overload;
    function CreateGDIBitmap: Graphics.TBitmap;
    property Pixels[X, Y: integer]: TPixel read GetPixel write SetPixel;
    property Width: integer read GetWidth write SetWidth;
    property Height: integer read GetHeight write SetHeight;
  end;

implementation

{ TBitmap }

function TBitmap.CreateGDIBitmap: Graphics.TBitmap;
var
  y: Integer;
begin
  Assert(sizeof(TPixel) = 4);
  result := Graphics.TBitmap.Create;
  result.SetSize(Width, Height);
  result.PixelFormat := pf32bit;
  for y := 0 to Height - 1 do
    Move(FBitmap[y, 0], result.ScanLine[y]^, Width * sizeof(TPixel));
end;

procedure TBitmap.FillWhite;
var
  y: Integer;
begin
  for y := 0 to Height - 1 do
    FillChar(FBitmap[y, 0], Width * sizeof(TPixel), $FF);
end;

procedure TBitmap.FillBlack;
var
  y: Integer;
begin
  for y := 0 to Height - 1 do
    FillChar(FBitmap[y, 0], Width * sizeof(TPixel), $00);
end;

function TBitmap.GetHeight: integer;
begin
  result := length(FBitmap);
end;

function TBitmap.GetPixel(X, Y: integer): TPixel;
begin
  result := FBitmap[Y, X];
end;

function TBitmap.GetWidth: integer;
begin
```

```
  if FBitmap <> nil then
    result := length(FBitmap[0])
  else
    result := 0;
end;

function TBitmap.PixelExists(const APixel: TPoint): boolean;
begin
  result := InRange(APixel.X, 0, GetWidth - 1) and
    InRange(APixel.Y, 0, GetHeight - 1);
end;

function TBitmap.PixelExists(const AX, AY: integer): boolean;
begin
  result := InRange(AX, 0, GetWidth - 1) and InRange(AY, 0, GetHeight - 1);
end;

procedure TBitmap.SetHeight(const Value: integer);
begin
  SetLength(FBitmap, Value, GetWidth);
end;

procedure TBitmap.SetPixel(X, Y: integer; const Value: TPixel);
begin
  FBitmap[Y, X] := Value;
end;

procedure TBitmap.SetSize(const AWidth, AHeight: integer);
begin
  SetLength(FBitmap, AHeight, AWidth);
end;

procedure TBitmap.SetWidth(const Value: integer);
begin
  SetLength(FBitmap, GetHeight, Value);
end;
```

Notice in particular the function **CreateGDIBitmap** that creates a Windows bitmap object from our bitmap data. The reason why we demand that the reserved byte in **TBitmap.WHITE** be $FF is that it is then possible to create an entirely white bitmap using **FillWhite**. Of course, we could use this **FillWhite** procedure regardless of the value of **TBitmap.WHITE**, but later on, when we investigate whether a pixel is white or black, we try **<> TBitmap.WHITE** rather than **= TBitmap.BLACK**, and so the white 4-colour in the bitmap needs to be identical to the constant **TBitmap.WHITE**. Indeed, if constant used the 'ordinary' 32-bit white colour $00FFFFFF, then every pixel would be considered to be occupied. But why do we not want to test **<> TBitmap.WHITE** instead of **= TBitmap.BLACK**? The reason is that we want to allow occupied pixels to have *any* colour (except for **TBitmap.WHITE**), not only black.

We will also introduce a couple of self-explanatory helper functions:

```
function RandomAngle: real; inline;
begin
  result := 2*Pi*Random;
end;

```

```
procedure DoRandomStep(var APoint: TPoint); inline;
begin
  case RandomRange(0, 4) of
    0: inc(APoint.X);
    1: dec(APoint.X);
    2: inc(APoint.Y);
    3: dec(APoint.Y);
  end;
end;
```

Since we might expect that our final version of the DLA generator might take a large number of parameters, we will store all parameters inside a single record. The code below is a straightforward implementation of our algorithm discussed above:

```
interface

type
  TDLACreateSettings = record
    Size: integer;
    MaxNumMoleculesAdsorbed,
    MaxNumMoleculesUsed: Int64;
    Iterations,
    UsedMolecules,
    AdsorbedMolecules: Int64;
    Radius,
    DurationInSecs: real;
  end;

function CreateDLAFractal(var Settings: TDLACreateSettings): TBitmap;

implementation

function CreateDLAFractal(var Settings: TDLACreateSettings): TBitmap;
var
  cx, cy: integer;
  pnt: TPoint;
  sintheta, costheta: extended;

  procedure SetInitialPos;
  begin
    SinCos(RandomAngle, sintheta, costheta);
    pnt.X := Round(cx + settings.Radius*costheta);
    pnt.Y := Round(cy + settings.Radius*sintheta);
  end;

  function WithinCircle: boolean;
  begin
    result := Hypot(pnt.X - cx, pnt.Y - cy) < settings.radius + 6;
  end;

  function ShouldAdsorb(ABitmap: TBitmap; AX, AY: integer): boolean;
  var
    i: Integer;
    j: Integer;
  begin
    result := false;
    if not ABitmap.PixelExists(AX, AY) then Exit;
    for i := -1 to 1 do
      for j := -1 to 1 do
        if ABitmap.PixelExists(AX + i, AY + j) and (ABitmap.Pixels[AX + i, AY + j]
<> TBitmap.WHITE) then
          Exit(true);
  end;
```

```
50
51 var
52   c1, c2, f: Int64;
53
54 begin
55   QueryPerformanceCounter(c1);
56   QueryPerformanceFrequency(f);
57
58   result := TBitmap.Create(nil);
59   result.SetSize(Settings.Size, Settings.Size);
60   result.FillWhite;
61
62   cx := Settings.Size div 2;
63   cy := Settings.Size div 2;
64
65   result.Pixels[cx, cy] := TBitmap.BLACK;
66
67   settings.Iterations := 0;
68   settings.UsedMolecules := 0;
69   settings.AdsorbedMolecules := 0;
70   settings.Radius := 6;
71   while (settings.AdsorbedMolecules < Settings.MaxNumMoleculesAdsorbed) and
72     (settings.UsedMolecules < Settings.MaxNumMoleculesUsed) do
73   begin
74     SetInitialPos;
75     while WithinCircle do
76     begin
77       if ShouldAdsorb(result, pnt.X, pnt.Y) then
78       begin
79         result.Pixels[pnt.X, pnt.Y] := TBitmap.BLACK;
80         inc(settings.AdsorbedMolecules);
81         settings.Radius := Max(settings.Radius, hypot(pnt.X - cx, pnt.Y - cy));
82         break;
83       end;
84       inc(settings.Iterations);
85       DoRandomStep(pnt);
86     end;
87     inc(settings.UsedMolecules);
88   end;
89
90   QueryPerformanceCounter(c2);
91   settings.DurationInSecs := (c2-c1) / f;
92
93 end;
```

Notice the addition of the constant 6 on line 34. If it were not for this addition, the result of the function **WithinCircle** might be false every time we insert a new particle on the circle itself; numerical fuzz might determine the outcome. Hence, in practice, it is important that the *insertion circle* (the circle on which we insert new particles) has a slightly smaller radius than the *killing circle* (the circle that we do not allow particles to pass through). The members of the **TDLACreateSettings** record are rather self-explanatory. The input parameters are

- **Size:** The width and height (in pixels) of the square bitmap.
- **MaxNumMoleculesAdsorbed:** The maximum number of particles adsorbed. The resulting bitmap will not contain more black pixels than this number. This parameter measures (roughly) the number of particles found on the resulting bitmap. If you do not want to set any limit on the number of molecules adsorbed, then set this parameter to its maximum value $2^{63} - 1$).

- **MaxNumMoleculesUsed:** The maximum number of particles used (also counting those that escape and do not adsorb). This parameter measures (roughly) the runtime of the algorithm. If you do not want to set any limit on the number of molecules used, then set this parameter to its maximum value $2^{63} - 1$).

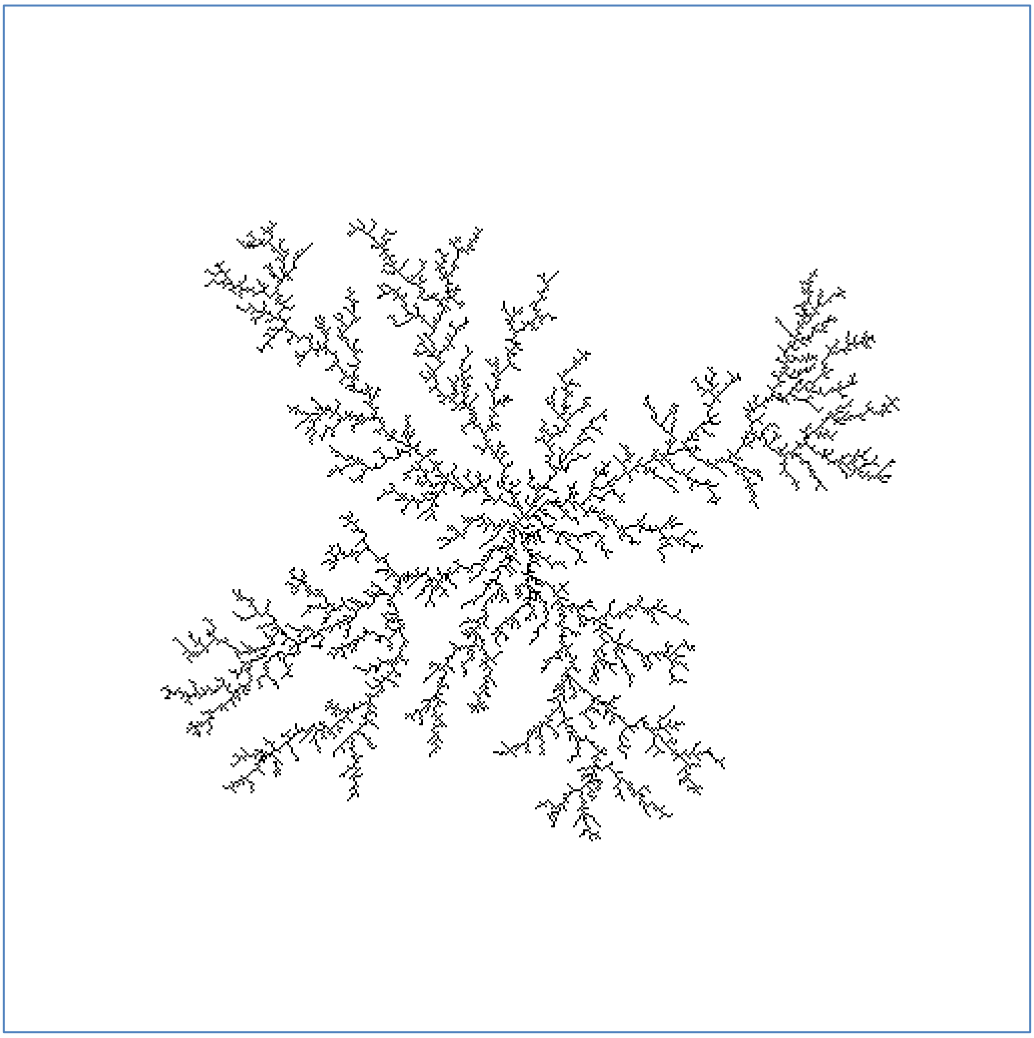
The output parameters are

- **Iterations:** The number of steps used in the simulation (every particle contributes with a lot of steps).
- **UsedMolecules:** The number of particles that was used in the simulation (also counting those that escaped).
- **AdsorbedMolecules:** The number of particles that was adsorbed.
- **Radius:** The maximum distance from the centre of the bitmap to a black pixel (or, the number 6, whatever is greatest).
- **DurationInSecs:** The runtime of the algorithm in seconds.

Testing

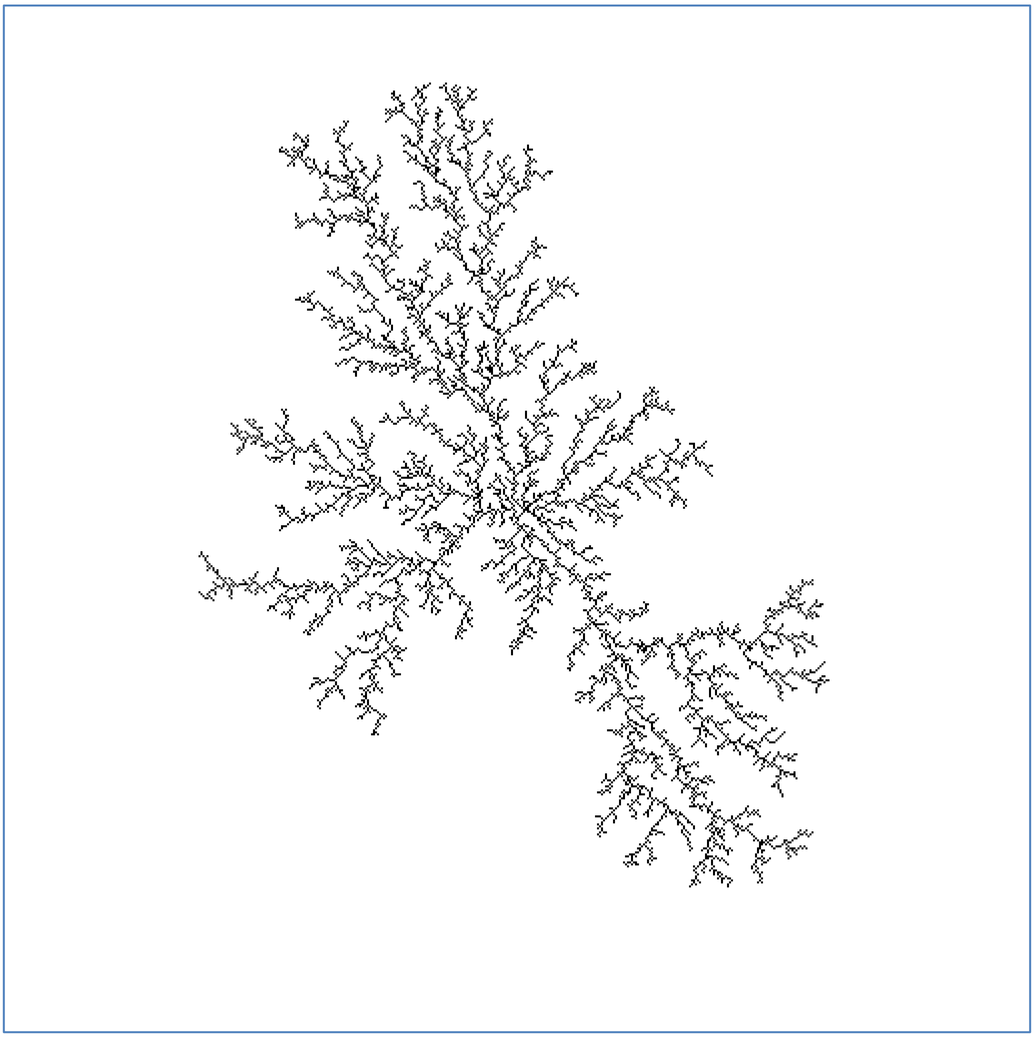
Let us test our function with a size of 256 and a maximum number of adsorbed particles of 2048. To get a feeling for the variation of the output, we will run three simulations. The resulting bitmaps and the corresponding output parameters are given below.

DLA1



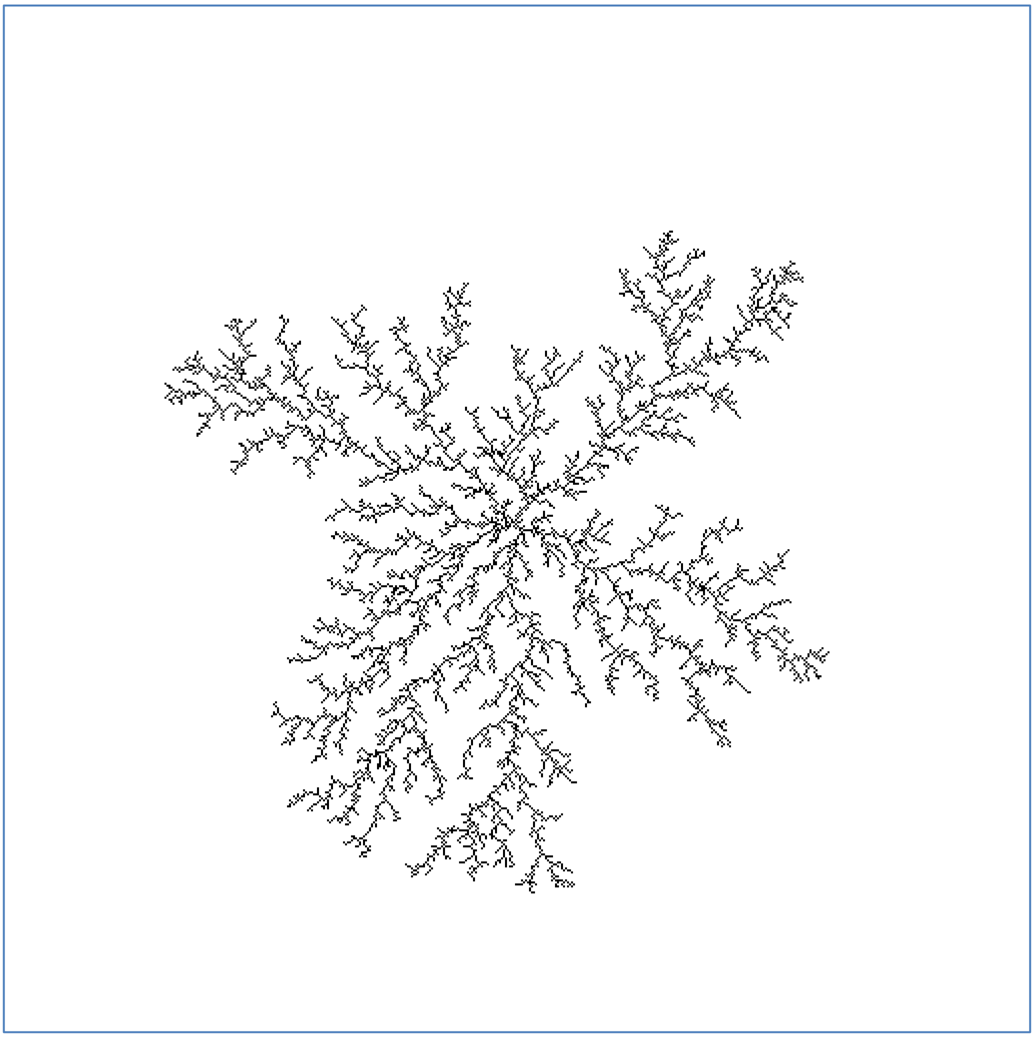
Width	512 px
Height	512 px
Iterations	7584296
Particles used	24015
Particles adsorbed	8192
Radius	199.02
Duration	1.10 s
∴ Iterations/sec	6894815 s^{-1}

DLA2



Width	512 px
Height	512 px
Iterations	15144283
Particles used	31403
Particles adsorbed	8192
Radius	221.84
Duration	2.23 s
∴ Iterations/sec	6791158 s^{-1}

DLA3

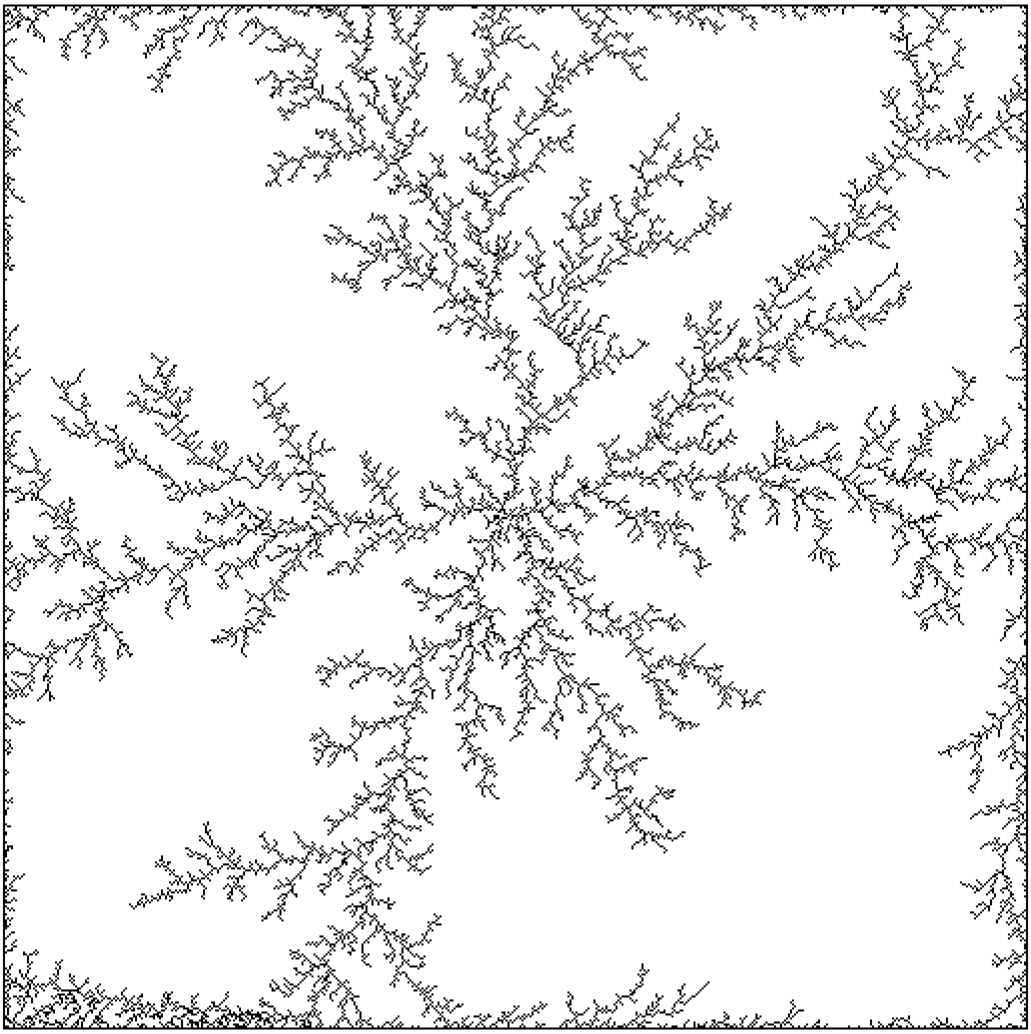


Width	512 px
Height	512 px
Iterations	5976270
Particles used	21909
Particles adsorbed	8192
Radius	188.32
Duration	0.88 s
∴ Iterations/sec	6791216 s^{-1}

It is evident from the tables above that it does take quite some time to compute even DLA fractals of moderate size. But it is also evident that this is not due to slow computers; indeed, the number of iterations per second is almost seven million! Consequently, DLA simulation is an inherently time-consuming task. This motivates our attempt to make a parallel version of the algorithm. Indeed, a typical high-end consumer computer has eight logical processors, and so one would expect a parallel algorithm to be up to eight times faster (but definitely less due to coordination work, of course).

Let us increase the number of adsorbed atoms to get a feeling for the qualitative features of the high-particle limit. With 81920 particles, we obtain the following result:

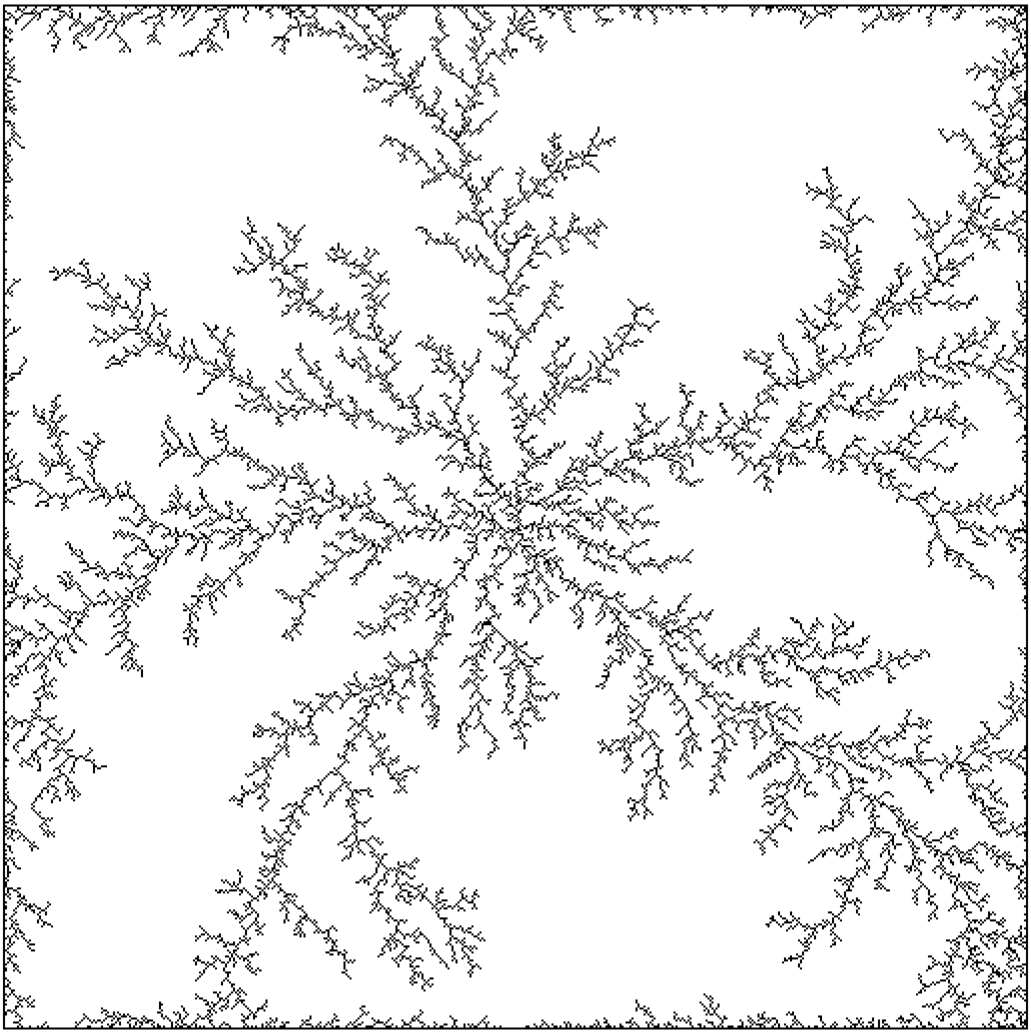
DLA4



Width	512 px
Height	512 px
Iterations	725989450
Particles used	744446
Particles adsorbed	81920
Radius	362.04
Duration	44.97 s
∴ Iterations/sec	16143861 s^{-1}

In this case, we reach the boundary of the bitmap. At some point the radius hit the special value $\frac{512}{2}$; at this time, the inserting circle was inscribed in the square. Then the fractal continued to grow until the radius hit the upper limit $\frac{512}{2}\sqrt{2} = 362.038 \ldots$. At this time, the square bitmap was inscribed in the circle (and not the other way around!), and the fractal began to grow along the edges of the bit-map. (Indeed, it cannot grow outside the bitmap.) As soon as the fractal had formed a string along the rim of the bitmap, the interior of the fractal couldn't grow further since every new particle was inserted outside the fractal. Indeed, if we increase the number of adsorbed atoms by a factor of four, the result is almost identical in terms of pixel density and qualitative features:

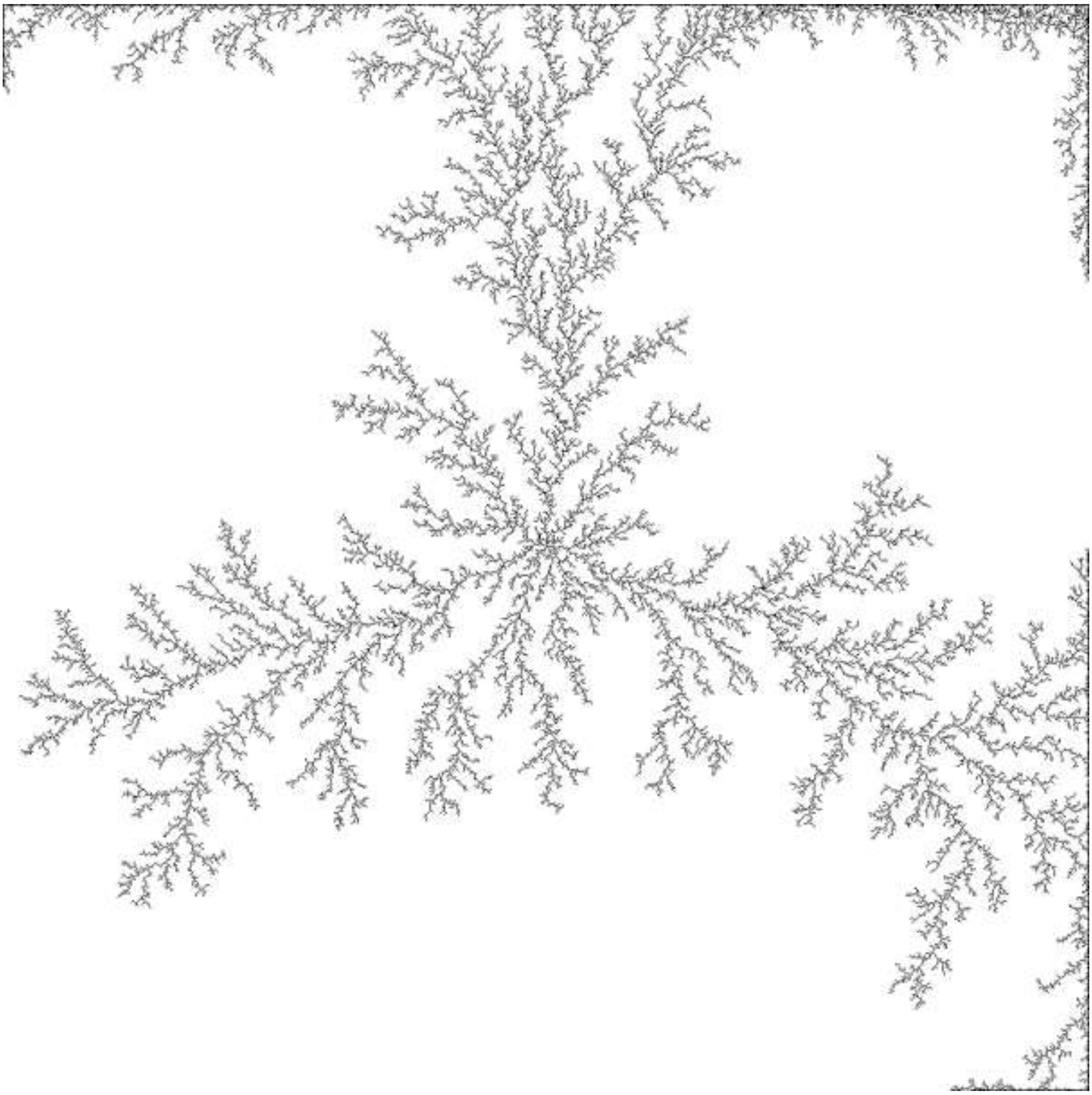
DLA5



Width	512 px
Height	512 px
Iterations	2560584849
Particles used	2856319
Particles adsorbed	327680
Radius	362.04
Duration	130.05 s
∴ Iterations/sec	19689233 s^{-1}

It is also not surprising that the average number of iterations per second increases as soon as a bounding rim is formed. We will now turn to the task of making our algorithm parallel. As a suitable benchmark we use a fractal of size 1024 and 81920 adsorbed particles. Using our current implementation of the algorithm, we obtain the following result.

DLA6



Width	1024 px
Height	1024 px
Iterations	5813014766
Particles used	1831032
Particles adsorbed	81920
Radius	723.37
Duration	499.49 s (8.3 min)
∴ Iterations/sec	11637900 s^{-1}

Notice that it took about eight minutes[3] to complete the simulation, and we are certainly interested in computing even bigger fractals. Thus our endeavour is well-motivated. Our (ideal) goal is to reduce

[3] A second simulation with the same input parameters took twelve minutes to complete. Every DLA simulation is unique.

the computation time of a fractal of this size to almost one eighth, that is, to about one minute, on an eight-core processor.

Making the Algorithm Parallel

The output parameters, especially the members **settings.Iterations** and **settings.AtomsUsed** indicate that a very large number of iterations is performed for each particle. Indeed, the ratios are

$$316, \quad 482, \quad 273, \quad 975, \quad 896, \quad 3175, \quad 3789$$

where the last ratio is from the simulation mentioned in footnote 3 on page 16. Hence, the ratio is large, and seems to increase with the size of the fractal. In particular, this means that the ratio increases with the need for performance improvements! This is very fortunate, because the approach we will employ, namely, letting each thread iterate a single particle at a time, works best when most of the work is done per particle.

Code

```
type
  PDLACreateSettings = ^TDLACreateSettings;
  TDLACreateSettings = record                         // New definition!
    Size: integer;
    ThreadCount: integer;                             // New member
    MaxNumMoleculesAdsorbed,
    MaxNumMoleculesUsed: integer;                     // New type
                                                      // Member 'iterations' removed
    UsedMolecules,
    AdsorbedMolecules: integer;                       // New type
    Radius,
    DurationInSecs: real;
    RadiusCriticalSection: TRTLCriticalSection;       // New member
  end;

type
  PDLAData = ^TDLAData;
  TDLAData = record
    Settings: PDLACreateSettings;
    Bitmap: PBitmap;
    cx, cy: integer;
  end;

function RandomWalker(Parameter: Pointer): integer;
var
  pnt: TPoint;
  sintheta, costheta: extended;
  cx, cy: integer;

  Settings: PDLACreateSettings;
  Bitmap: PBitmap;

  rCache: real;

  function WithinCircle: boolean;
  begin
    result := Hypot(pnt.X - cx, pnt.Y - cy) < rCache + 6;
  end;

  function ShouldAdsorb: boolean;
  var
    i: Integer;
    j: Integer;
```

```
  begin
    result := false;
    if not Bitmap.PixelExists(pnt.X, pnt.Y) then Exit;
    for i := -1 to 1 do
      for j := -1 to 1 do
        if Bitmap.PixelExists(pnt.X + i, pnt.Y + j) and
          (Bitmap.Pixels[pnt.X + i, pnt.Y + j] <> TBitmap.WHITE) then
          Exit(true);
  end;

begin
  Settings := PDLAData(Parameter).Settings;
  Bitmap := PDLAData(Parameter).Bitmap;

  cx := PDLAData(Parameter).cx;
  cy := PDLAData(Parameter).cy;

  while (Settings.AdsorbedMolecules < Settings.MaxNumMoleculesAdsorbed) and
    (Settings.UsedMolecules < Settings.MaxNumMoleculesUsed) do
  begin
    InterlockedIncrement(Settings.UsedMolecules);
    SinCos(RandomAngle, sintheta, costheta);
    EnterCriticalSection(Settings.RadiusCriticalSection);
      rCache := Settings.Radius;
    LeaveCriticalSection(Settings.RadiusCriticalSection);
    pnt.X := Round(cx + rCache*costheta);
    pnt.Y := Round(cy + rCache*sintheta);
    while WithinCircle do
    begin
      if ShouldAdsorb then
      begin
        Bitmap.Pixels[pnt.X, pnt.Y] := TBitmap.BLACK;
        InterlockedIncrement(Settings.AdsorbedMolecules);
        EnterCriticalSection(Settings.RadiusCriticalSection);
          Settings.Radius := Max(Settings.Radius, hypot(pnt.X - cx, pnt.Y - cy));
        LeaveCriticalSection(Settings.RadiusCriticalSection);
        break;
      end;
      DoRandomStep(pnt);
    end;
  end;

  result := 0;

end;

function CreateDLAFractal(var Settings: TDLACreateSettings): TBitmap;
var
  data: TDLAData;
  c1, c2, f: Int64;
  dummy: cardinal;
  i: Integer;
  threads: array of THandle;
begin
  QueryPerformanceCounter(c1);
  QueryPerformanceFrequency(f);

  result := TBitmap.Create;
  result.SetSize(Settings.Size, Settings.Size);
  result.FillWhite;

  Settings.UsedMolecules := 0;
  Settings.AdsorbedMolecules := 0;
  Settings.Radius := 6;

  data.Settings := @Settings;
```

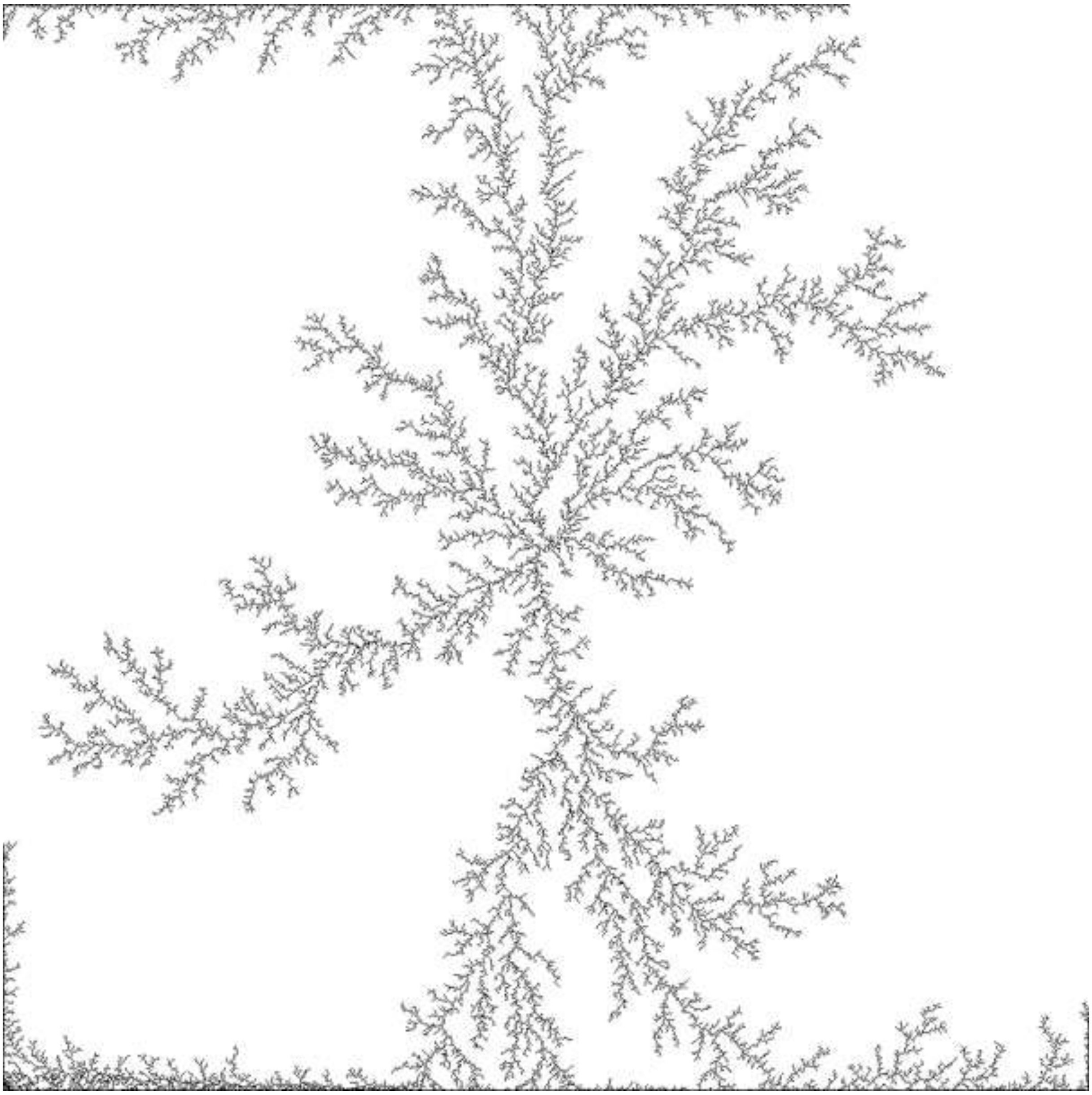
```
  data.Bitmap := @result;
  data.cx := Settings.Size div 2;
  data.cy := Settings.Size div 2;

  result.Pixels[data.cx, data.cy] := TBitmap.BLACK;

  InitializeCriticalSection(Settings.RadiusCriticalSection);

  SetLength(threads, Settings.ThreadCount);
  for i := 0 to Settings.ThreadCount - 1 do
    threads[i] := BeginThread(nil, 0, @RandomWalker, @Data, 0, dummy);

  if WaitForMultipleObjects(Settings.ThreadCount, @threads[0], true, INFINITE) = WA
IT_FAILED then
    RaiseLastOSError;

  DeleteCriticalSection(Settings.RadiusCriticalSection);

  QueryPerformanceCounter(c2);
  Settings.DurationInSecs := (c2-c1) / f;

end;
```

A number of remarks are in order. First, we have removed the **Iterations** member of **TDLACreateSettings**. This is done for performance reasons. Indeed, **Iterations** is to be increased every iteration, that is, in the inner loops. In addition, we are not particularly interested in this number (other than for algorithm-theoretical inquiries). We have also replaced every **inc** compiler function by the Windows API function **InterlockedIncrement** in order to make the thread function safe(r) to run in parallel. Since this function only works for 32-bit integers, and the 64-bit equivalent **InterlockedIncrement64** might be incompatible with older versions of Windows (in particular, Windows XP), we have changed the types of **UsedMolecules** and **AdsorbedMolecules** to 32-bit integers. (This is reasonable, but it would *not* have been reasonable for **Iterations**, and so this is yet another reason to remove that member.) Since (as far as I know) there is no **Interlocked***function that works with floating-point numbers, I use critical sections when I read and write the **Radius** member. Apparently, I use *no* protection when I read/write the pixels of the bitmap. Although I could use a critical section for this shared resource, that would likely remove the benefits of having a parallel algorithm in the first place, for the pixels are read in each iteration. However, the outcome of the program is probably not changed much by this ‘sloppiness’.

Testing

We try our new parallel implementation using our benchmarking settings with 1024 pixels and 81920 adsorbed particles.

DLA7



Width	1024 px
Height	1024 px
Iterations	N/A
Particles used	1733712
Particles adsorbed	81921
Radius	724.08
Duration	124.64 s (2.1 min)
∴ Iterations/sec	N/A

Recall that the single-threaded implementation used about 500 seconds, and so the new value of 124 seconds suggests is a significant improvement. However, since each fractal is unique, it is impossible to say anything by comparing *one* single simulation using each implementation. Therefore, we will now turn to some more accurate benchmarking.

Performance Testing

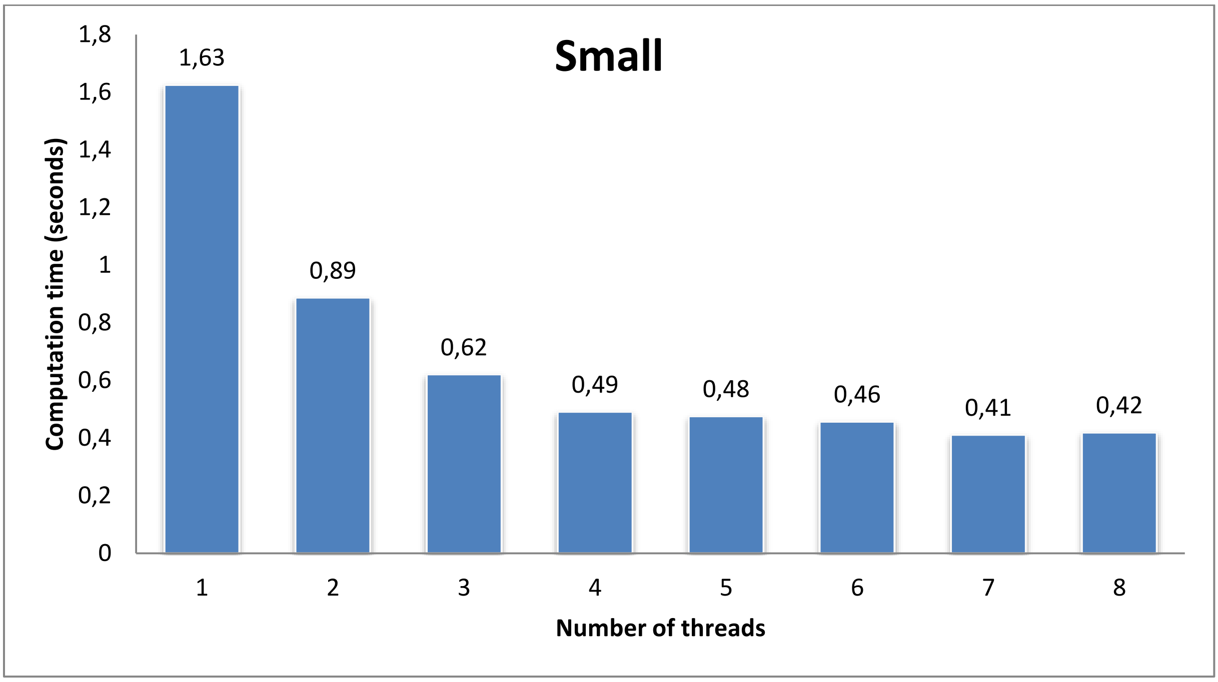
In this section, we will investigate how the performance depends upon the number of threads used. We will fix some initial parameters and perform a large number of simulations using these initial parameters for each number of threads, and then determine the average computation time versus thread count. We will use the following simple code:

```
procedure TmainFrm.btnBenchmarkClick(Sender: TObject);
var
  settings: TDLACreateSettings;
  results: array[1..8] of real;
  i: Integer;
  j: Integer;
const
  RUNS = ?;
begin
  settings.Size := ?;
  settings.MaxNumMoleculesAdsorbed := ?;
  settings.MaxNumMoleculesUsed := MaxInt;

  mOutput.Clear;

  for i := low(results) to high(results) do
  begin
    settings.ThreadCount := i;
    results[i] := 0;
    for j := 0 to RUNS - 1 do
    begin
      CreateDLAFractal(settings).Free;
      results[i] := results[i] + (1 / RUNS) * settings.DurationInSecs;
    end;
  end;

  for i := low(results) to high(results) do
    mOutput.Text := mOutput.Text + FloatToStr(results[i]) + #9;
end;
```

We will run three simulations with different initial parameters, according to the table below. Roughly, the simulations correspond to typical ‘small’, ‘medium’, and ‘large’ fractals.

Name	Size	Max number of adsorbed particles	Number of runs
Small	512	8192	100
Medium	TBD	TBD	TBD
Large	TBD	TBD	TBD

The results are displayed in the diagrams below.



TBW

Additional Features

Now that we have written a parallel implementation of the DLA algorithm, we can use this to build a fully-featured DLA simulator application. The first thing we need to do is to implement visual feedback as the fractal grows. Essentially, the user should see the entire growth as an animation. The natural way to accomplish this is to draw the current state of the bitmap at regular intervals (say, 30 times a second). We will also allow a new mode of simulation. Instead of specifying the maximum number of used or adsorbed particles in the input settings, it should be possible to start a simulation with no predefined end. Instead, the user can stop, and resume, the simulation at any time.

In addition, the user should be able to specify his own seed. It could be a single point (not necessarily at the centre of the bitmap), a line, a circle, any curve (such as a rabbit), any solid area, or, most generally, any point set. But then one question arises: if we now abandon the petri-dish approach, should we still make use of the insertion and killing circles? If so, how do we determine their centres and radii? Indeed, if the initial seed is the top-left pixel, then it would not make much sense to let the insertion set be the circle inscribed in the bitmap; it would make even less sense to use this circle (or a slightly larger concentric circle) as the killing circle (why?). There are several solutions to this problem, and we will use a rather simple one. Besides being simple, this solution has the benefit that it will work with almost any initial seed, and that it includes the previous method as a special case. We allow four different modes:

- **Fast Circular Mode:** This is the mode that we have been using up to now. The insertion and killing sets are circles centred at the centre of the bitmap. The radius of the insertion circle is always the radius of the fractal, that is, the maximum distance to any black pixel from the centre of the bitmap (but no lower than 6). The killing circle is a slightly larger, but concentric, circle.

- **True Circular Mode:** The same but without the 'varying-radius' performance trick (that, admittedly, *does* have an effect on the outcome). The insertion set is the inscribed circle, and the killing set is a slightly larger concentric circle.
- **Uniform Mode:** The insertion set is the entire bitmap and the killing set is the (rectangular) boundary of the bitmap.
- **Boundary Mode:** The insertion set is the rectangle obtained by slightly shrinking the boundary of the bitmap towards the centre (the same amount for each edge). The killing set is the boundary of the bitmap.

Above, as usual, 'slightly' means 'six pixels'. Finally, we will add one last feature. As stated above, the typical example of a DLA fractal is one grown in a petri dish. This corresponds to our 'circular mode' insertion and killing sets. This works well as long as the fractal does not grow to the end of the bitmap. This is trivial. The 'problem' is that the appearance of an overgrown circular DLA fractal gives the impression of a square container (cf DLA4, DLA5, and DLA6). If we modify our **ShouldAdsorb** function as to return false outside the inscribed circle, then the fractal will instead start to grow along the (imaginary) rim of the petri dish. We will introduce a new input parameter, **CircularCompensator**, that is true iff this modification is applied.

Now, let us do some code. I found it natural to derive a new class based on the bitmap class to hold a DLA fractal.

```
interface

type
  TDLAMode = (dlaCircularTrue, dlaCircularFast, dlaUniform, dlaBoundary);

  PDLAFractal = ^TDLAFractal;
  TDLAFractal = class(TBitmap)
  private
    FMode: TDLAMode;
    FCircularCompensator: boolean;
    FThreadCount: integer;
    FCachedCX,
    FCachedCY: integer;
    FThreads: array of THandle;
    FCachedRadius: real;
    FCachedParticleCount: integer;
    FCachedRadiusOfInscribedCircle: integer;
    FRadiusCriticalSection: TRTLCriticalSection;
    FTerminatorCriticalSection: TRTLCriticalSection;
    FTerminate: boolean;
    FRunning: boolean;
    function GetCX: integer;
    function GetCY: integer;
    procedure FixWhite;
    function GetRadiusOfInscribedCircle: integer;
  public
    procedure LoadFromFile(const FileName: string);
    procedure SaveToFile(const FileName: string);
    procedure DrawPointAtCentre;
    procedure DrawRandomPoints(const ANumber: integer);
    procedure DrawCircle(const ARadius: integer);
    procedure DrawBoundary;
```

```
    procedure DrawLineAtMiddle;
    procedure DrawLineAtBottom;
    function GetRadius: real;
    function GetParticleCount: integer;
    procedure Simulate;
    procedure Pause;
    constructor Create;
    destructor Destroy; override;
    property Mode: TDLAMode read FMode write FMode default dlaCircularFast;
    property CircularCompensator: boolean read FCircularCompensator write FCircular
Compensator default false;
    property ThreadCount: integer read FThreadCount write FThreadCount default 1;
    property Running: boolean read FRunning;
    property Radius: real read GetRadius;
    property ParticleCount: integer read GetParticleCount;
    property Width;
    property Height;
  end;

implementation

{ TDLAFractal }

constructor TDLAFractal.Create;
begin
  inherited;
  FMode := dlaCircularFast;
  FCircularCompensator := false;
  FThreadCount := 1;
  FCachedRadius := 6;
  FCachedParticleCount := 0;
  InitializeCriticalSection(FTerminatorCriticalSection);
  InitializeCriticalSection(FRadiusCriticalSection);
  FTerminate := false;
  FRunning := false;
end;

destructor TDLAFractal.Destroy;
begin
  DeleteCriticalSection(FRadiusCriticalSection);
  DeleteCriticalSection(FTerminatorCriticalSection);
  inherited;
end;

procedure TDLAFractal.DrawBoundary;
var
  i: Integer;
begin
  for i := 0 to Width - 1 do
  begin
    Pixels[i, 0] := TBitmap.BLACK;
    Pixels[i, Height - 1] := TBitmap.BLACK;
  end;
  for i := 0 to Height - 1 do
  begin
    Pixels[0, i] := TBitmap.BLACK;
    Pixels[Width - 1, i] := TBitmap.BLACK;
  end;
```

```
end;

procedure TDLAFractal.DrawCircle(const ARadius: integer);
var
  cx, cy: integer;
  x, y: integer;
  cosphi, sinphi: extended;
  t: Integer;
begin
  cx := GetCX;
  cy := GetCY;
  for t := 0 to ceil(2*Pi*ARadius - 1) do
  begin
    SinCos(t / ARadius, sinphi, cosphi);
    x := Round(cx + ARadius*cosphi);
    y := Round(cy + ARadius*sinphi);
    if PixelExists(x, y) then
      Pixels[x, y] := TBitmap.BLACK;
  end;

end;

procedure TDLAFractal.DrawLineAtBottom;
var
  i: Integer;
begin
  for i := 0 to Width - 1 do
    Pixels[i, Height - 1] := TBitmap.BLACK;
end;

procedure TDLAFractal.DrawLineAtMiddle;
var
  i: Integer;
  cy: integer;
begin
  cy := GetCY;
  for i := 0 to Width - 1 do
    Pixels[i, cy] := TBitmap.BLACK;
end;

procedure TDLAFractal.DrawPointAtCentre;
begin
  if not Assigned(FBitmap) then Exit;
  Pixels[GetCX, GetCY] := TBitmap.BLACK;
end;

procedure TDLAFractal.DrawRandomPoints(const ANumber: integer);
var
  i: Integer;
begin
  if not Assigned(FBitmap) then Exit;
  for i := 0 to ANumber - 1 do
    Pixels[Random(Width), Random(Height)] := TBitmap.BLACK;
end;

function TDLAFractal.GetCX: integer;
begin
  result := Width div 2;
```

```
  FCachedCX := result;
end;

function TDLAFractal.GetCY: integer;
begin
  result := Height div 2;
  FCachedCY := result;
end;

function TDLAFractal.GetParticleCount: integer;
var
  y: Integer;
  x: Integer;
begin
  result := 0;
  for y := 0 to Height - 1 do
    for x := 0 to Width - 1 do
      if Pixels[x, y] <> TBitmap.WHITE then
        inc(result);
  FCachedParticleCount := result;
end;

function TDLAFractal.GetRadius: real;
var
  cx, cy: integer;
  y: Integer;
  x: Integer;
begin
  cx := GetCX;
  cy := GetCY;
  result := 6; // SIC!
  for y := 0 to Height - 1 do
    for x := 0 to Width - 1 do
      if Pixels[x, y] <> TBitmap.WHITE then
        result := Max(result, hypot(x - cx, y - cy));
  FCachedRadius := result;
end;

function TDLAFractal.GetRadiusOfInscribedCircle: integer;
begin
  result := Min(Width, Height) div 2;
  FCachedRadiusOfInscribedCircle := result;
end;

procedure TDLAFractal.FixWhite;
var
  y: Integer;
  x: Integer;
begin
  // Ugly but necessary.
  for y := 0 to Height - 1 do
    for x := 0 to Width - 1 do
      if (Pixels[x, y] and $00FFFFFF) = $FFFFFF then
        Pixels[x, y] := TBitmap.WHITE;

end;

procedure TDLAFractal.LoadFromFile(const FileName: string);
```

```
var
  y: integer;
begin
  Assert(sizeof(TPixel) = 4);
  with Graphics.TBitmap.Create do
    try
      LoadFromFile(FileName);
      Self.SetSize(Width, Height);
      PixelFormat := pf32bit;
      for y := 0 to Height - 1 do
        Move(Scanline[y]^, FBitmap[y, 0], Width * sizeof(TPixel));
    finally
      Free;
    end;
  FixWhite;
end;

procedure TDLAFractal.Pause;
begin
  EnterCriticalSection(FTerminatorCriticalSection);
  FTerminate := true;
  LeaveCriticalSection(FTerminatorCriticalSection);
  if WaitForMultipleObjects(FThreadCount, @FThreads[0], true, INFINITE) = WAIT_FAIL
ED then
    RaiseLastOSError;
  FTerminate := false;
  FRunning := false;
end;

function DLARandomWalker(Parameter: Pointer): integer;
var
  pnt: TPoint;
  sintheta, costheta: extended;
  cx, cy: integer;

  DLAFractal: TDLAFractal;

  rCache: real;

  function WithinRegion: boolean;
  begin
    case DLAFractal.Mode of
      dlaCircularTrue:
        result := Hypot(pnt.X - cx, pnt.Y -
 cy) < DLAFractal.FCachedRadiusOfInscribedCircle + 6;
      dlaCircularFast:
        result := Hypot(pnt.X - cx, pnt.Y - cy) < rCache + 6;
      dlaUniform:
        result := DLAFractal.PixelExists(pnt);
      dlaBoundary:
        result := DLAFractal.PixelExists(pnt);
    end;
  end;

  function WithinInscribedCircle: boolean;
  begin
    result := Hypot(pnt.X - cx, pnt.Y -
 cy) < DLAFractal.FCachedRadiusOfInscribedCircle;
```

```
  end;

  function ShouldAdsorb: boolean;
  var
    i: Integer;
    j: Integer;
  begin
    result := false;
    if not DLAFractal.PixelExists(pnt.X, pnt.Y) then Exit;
    if DLAFractal.CircularCompensator and not WithinInscribedCircle then Exit;
    for i := -1 to 1 do
      for j := -1 to 1 do
        if DLAFractal.PixelExists(pnt.X + i, pnt.Y + j) and
          (DLAFractal.Pixels[pnt.X + i, pnt.Y + j] <> TBitmap.WHITE) then
          Exit(true);
  end;

  function IsTerminated: boolean;
  begin
    EnterCriticalSection(DLAFractal.FTerminatorCriticalSection);
    result := DLAFractal.FTerminate;
    LeaveCriticalSection(DLAFractal.FTerminatorCriticalSection);
  end;

begin
  DLAFractal := TDLAFractal(Parameter);

  cx := DLAFractal.FCachedCX;
  cy := DLAFractal.FCachedCY;

  while not IsTerminated do
  begin

    // Set initial position
    case DLAFractal.Mode of
      dlaCircularTrue:
        begin
          SinCos(RandomAngle, sintheta, costheta);
          pnt.X := Round(cx + DLAFractal.FCachedRadiusOfInscribedCircle*costheta);
          pnt.Y := Round(cy + DLAFractal.FCachedRadiusOfInscribedCircle*sintheta);
        end;
      dlaCircularFast:
        begin
          SinCos(RandomAngle, sintheta, costheta);
          EnterCriticalSection(DLAFractal.FRadiusCriticalSection);
            rCache := DLAFractal.FCachedRadius;
          LeaveCriticalSection(DLAFractal.FRadiusCriticalSection);
          pnt.X := Round(cx + rCache*costheta);
          pnt.Y := Round(cy + rCache*sintheta);
        end;
      dlaUniform:
        begin
          pnt.X := Random(DLAFractal.Width);
          pnt.Y := Random(DLAFractal.Height);
        end;
      dlaBoundary:
        begin
          case Random(4) of
```

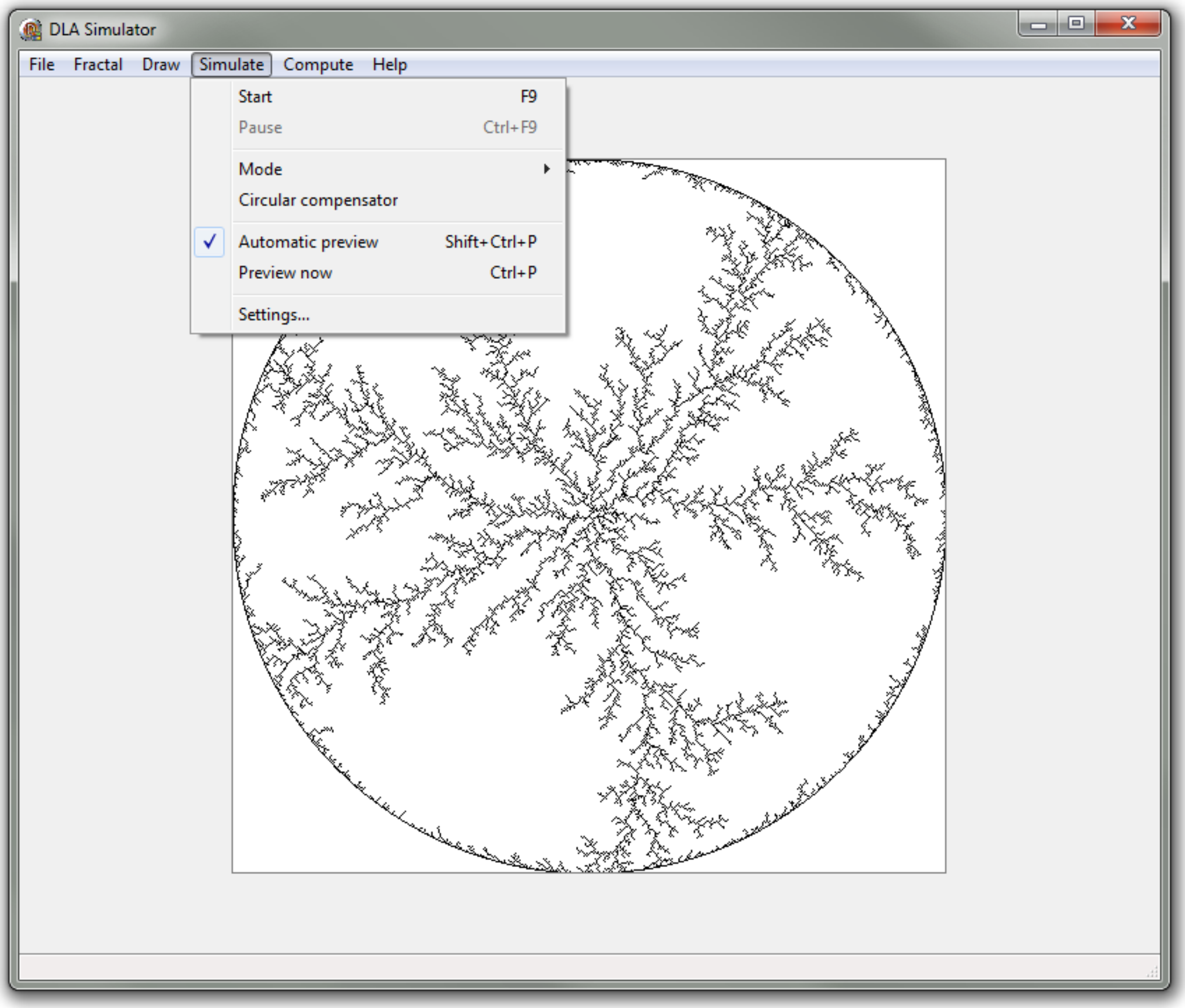
```
          0: pnt := Point(Random(DLAFractal.Width), 6);
          1: pnt := Point(DLAFractal.Width - 1 - 6, Random(DLAFractal.Height));
          2: pnt := Point(Random(DLAFractal.Width), DLAFractal.Height - 6);
          3: pnt := Point(6, Random(DLAFractal.Height));
        end;
      end;
    end;

    // Walk
    while WithinRegion do
    begin
      if ShouldAdsorb then
      begin
        DLAFractal.Pixels[pnt.X, pnt.Y] := TBitmap.BLACK;
        InterlockedIncrement(DLAFractal.FCachedParticleCount);
        EnterCriticalSection(DLAFractal.FRadiusCriticalSection);
          DLAFractal.FCachedRadius := Max(DLAFractal.FCachedRadius,
            hypot(pnt.X - cx, pnt.Y - cy));
        LeaveCriticalSection(DLAFractal.FRadiusCriticalSection);
        break;
      end;
      DoRandomStep(pnt);
    end;

  end;

  result := 0;
end;

procedure TDLAFractal.SaveToFile(const FileName: string);
begin
  with CreateGDIBitmap do
    try
      SaveToFile(FileName);
    finally
      Free;
    end;
end;

procedure TDLAFractal.Simulate;
var
  i: integer;
  dummy: cardinal;
begin

  GetRadiusOfInscribedCircle;

  GetRadius; //
  GetCX;     // update the cached values
  GetCY;     //

  FTerminate := false;
  SetLength(FThreads, FThreadCount);
  for i := 0 to FThreadCount - 1 do
    FThreads[i] := BeginThread(nil, 0, @DLARandomWalker, Self, 0, dummy);

  FRunning := true;
end;
```

The reader should be able to understand the code without any further explanations, but let us highlight some of its features. Notice in particular the helper functions **DrawPointAtCentre**, **DrawRandomPoints**, **DrawCircle**, **DrawBoundary**, **DrawLineAtMiddle**, and **DrawLineAtBottom** that can be used to create different kinds of seeds. Clearly, they can be combined in any order (and they commute!). By means of **LoadFromFile**, any bitmap can be used as seed, and, in addition, one can always load a previously computed fractal to continue its grown with the same (or, perhaps, new) parameters. The programmer can start a simulation using **Simulate** and pause an on-going simulation using **Pause**. The **ThreadCount** property can be used to specify the number of threads used whilesimulating. At any time, a preview can be obtained by calling **CreateGDIBitmap** inherited from **TBitmap**. (And, yes, this seems to work without any special precautions such as critical sections.)

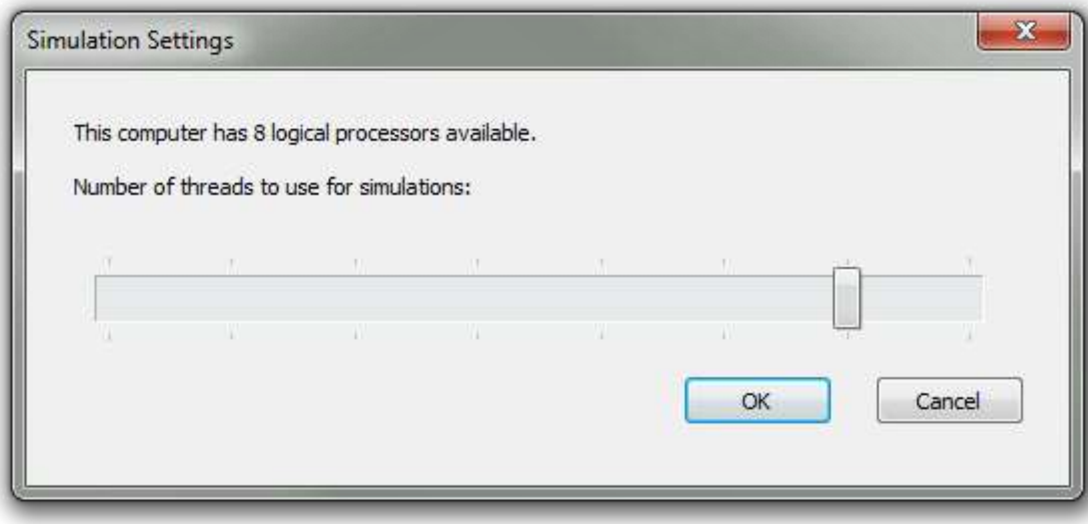
Notice that the previewing is (likely) done by the GUI thread. Assume that this is done with 30 FP, and assume that the computer has N logical processors. If $N \geq 2$ and you run the simulation with a single thread, then the previewing will not slow down the simulation, even if the bitmap is big and the previewing eats up most of the CPU time of the GUI thread. If there is no previewing, one reserve $N-1$ threads for the simulation, and the computer will still be perfectly responsive (unless you are running any other demanding processes). If the bitmap is large and you previewing is enabled at a high frame rate, you might wish to reserve only $N-2$ treads to the simulation. Then one of the remaining logical processors can run the GUI thread (with the previewing), and you still have one logical processor left. Of course, if you want to compute a ridiculously large DLA fractal overnight, you use N threads and disable automatic previewing, unless Windows Media Center is scheduled to record some DVB video stream, in which case you use $N-1$ threads, just to be safe.

GUI

Below is a screenshot from a sample application utilising our algorithm.

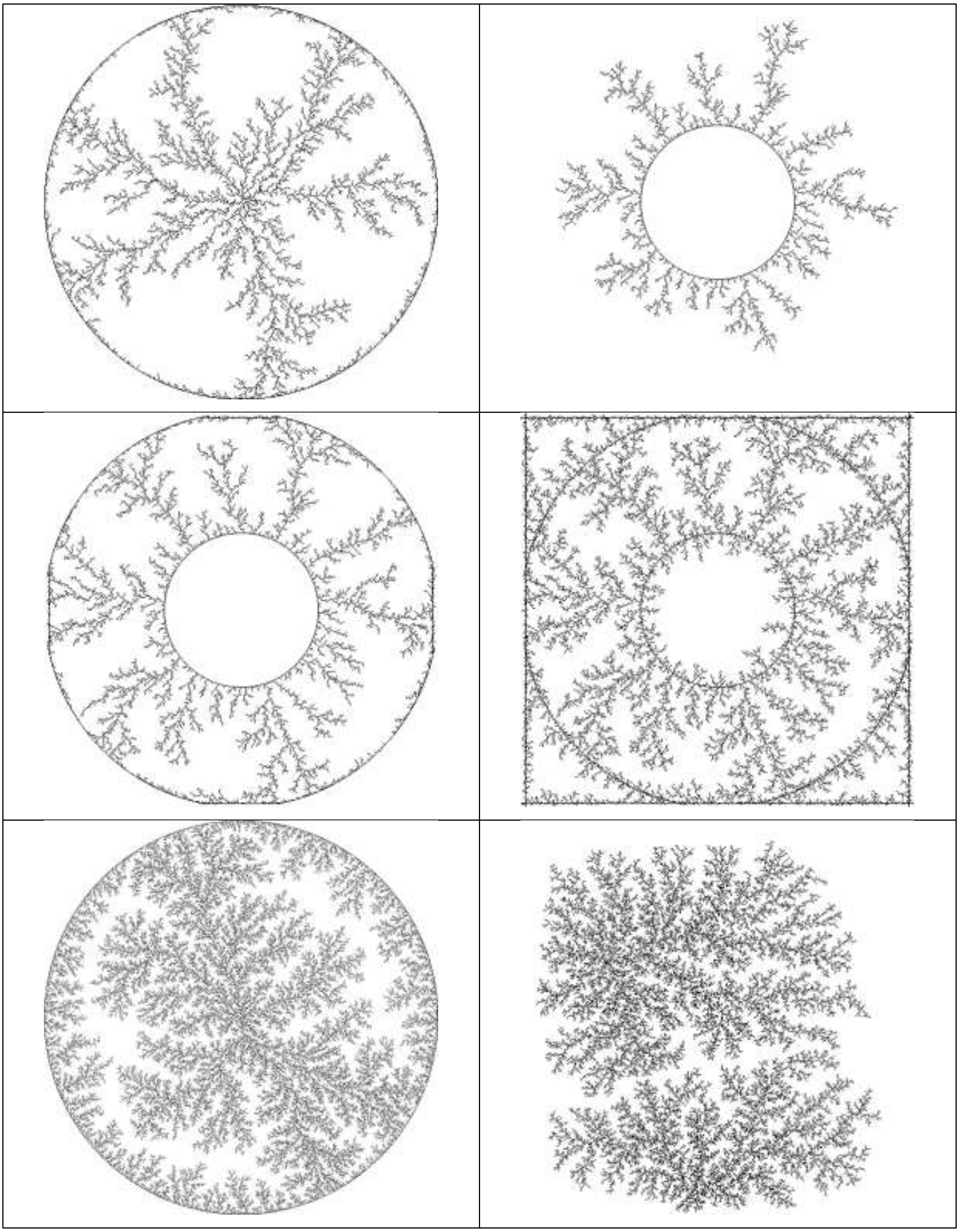


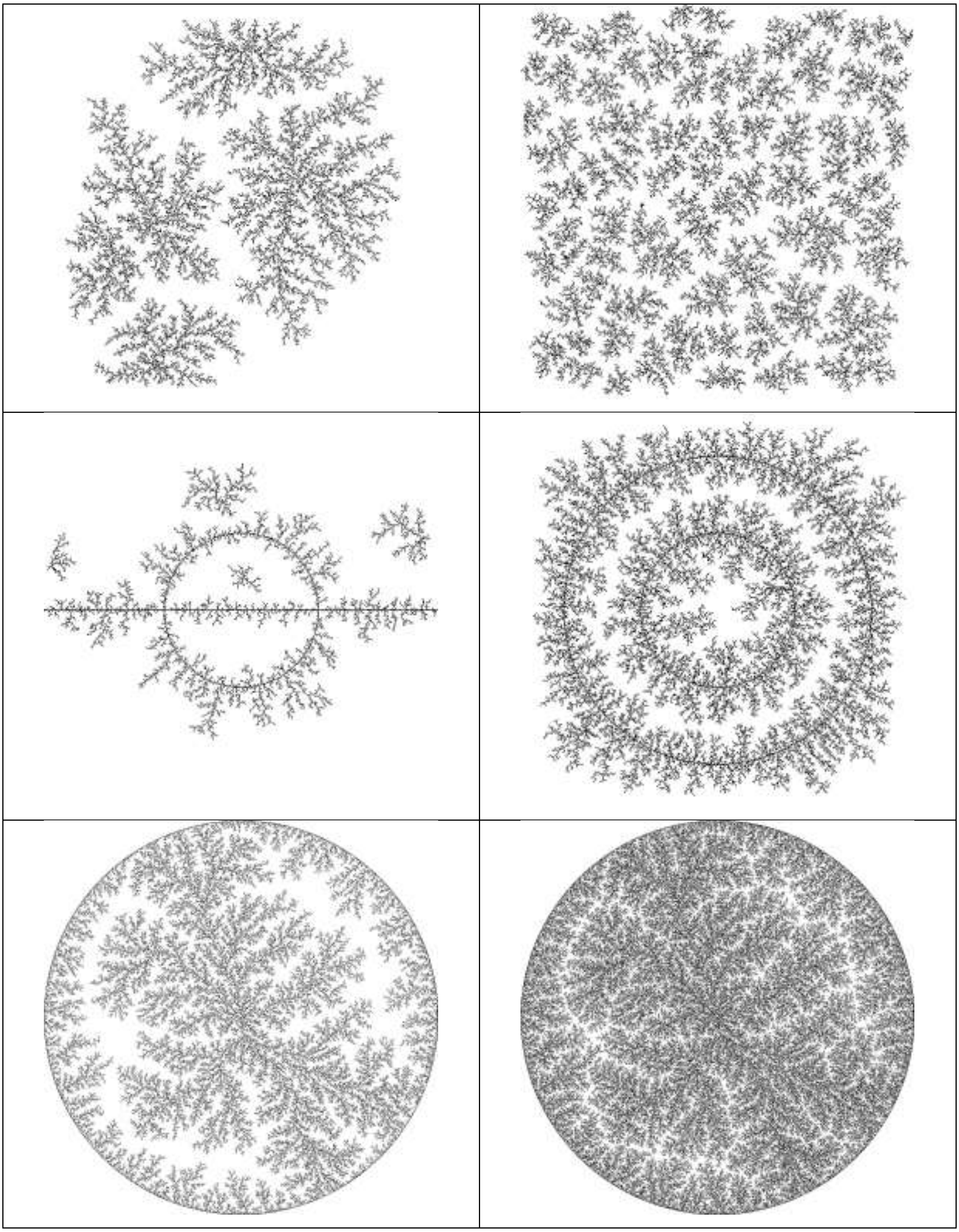
The settings dialog:

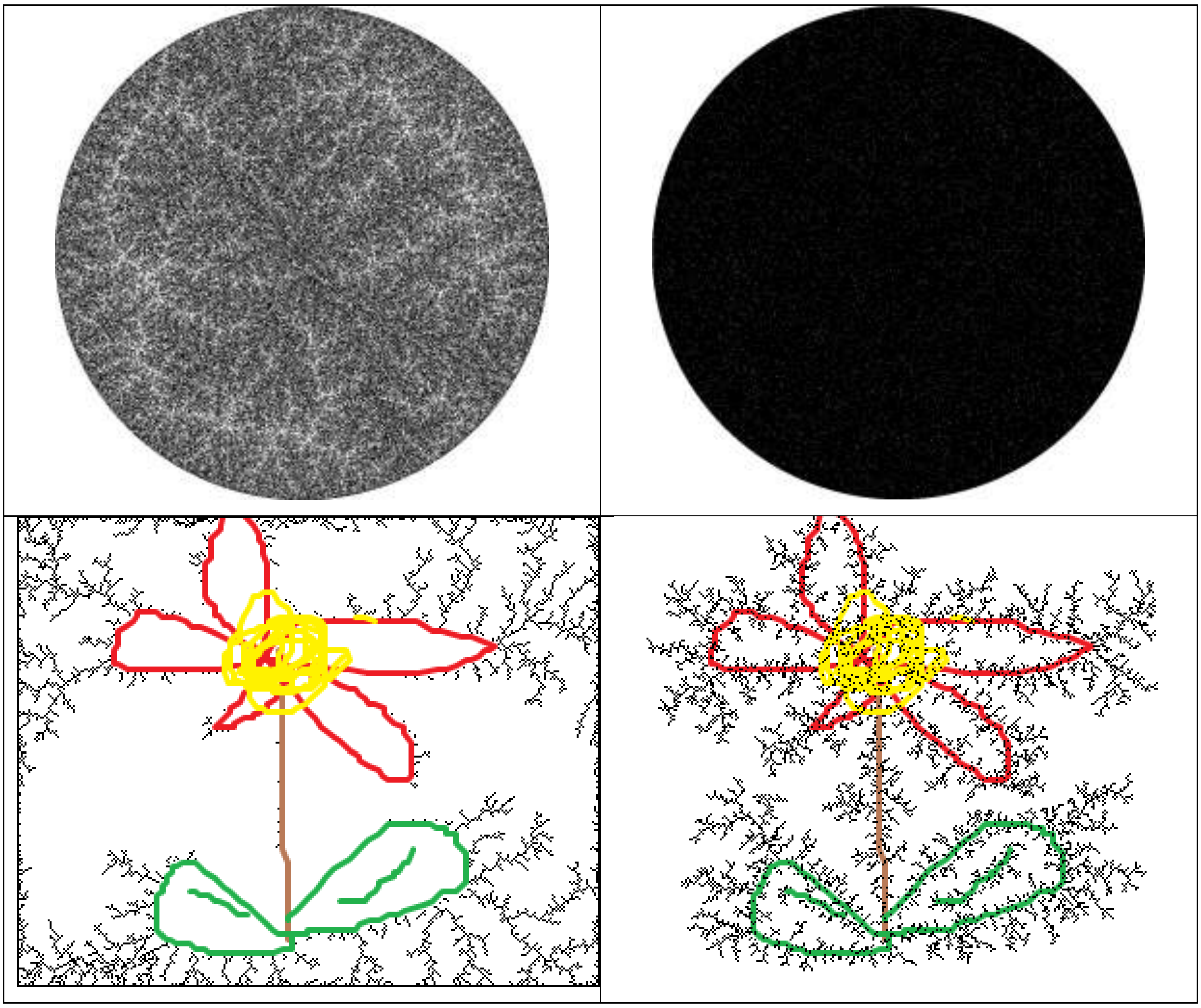


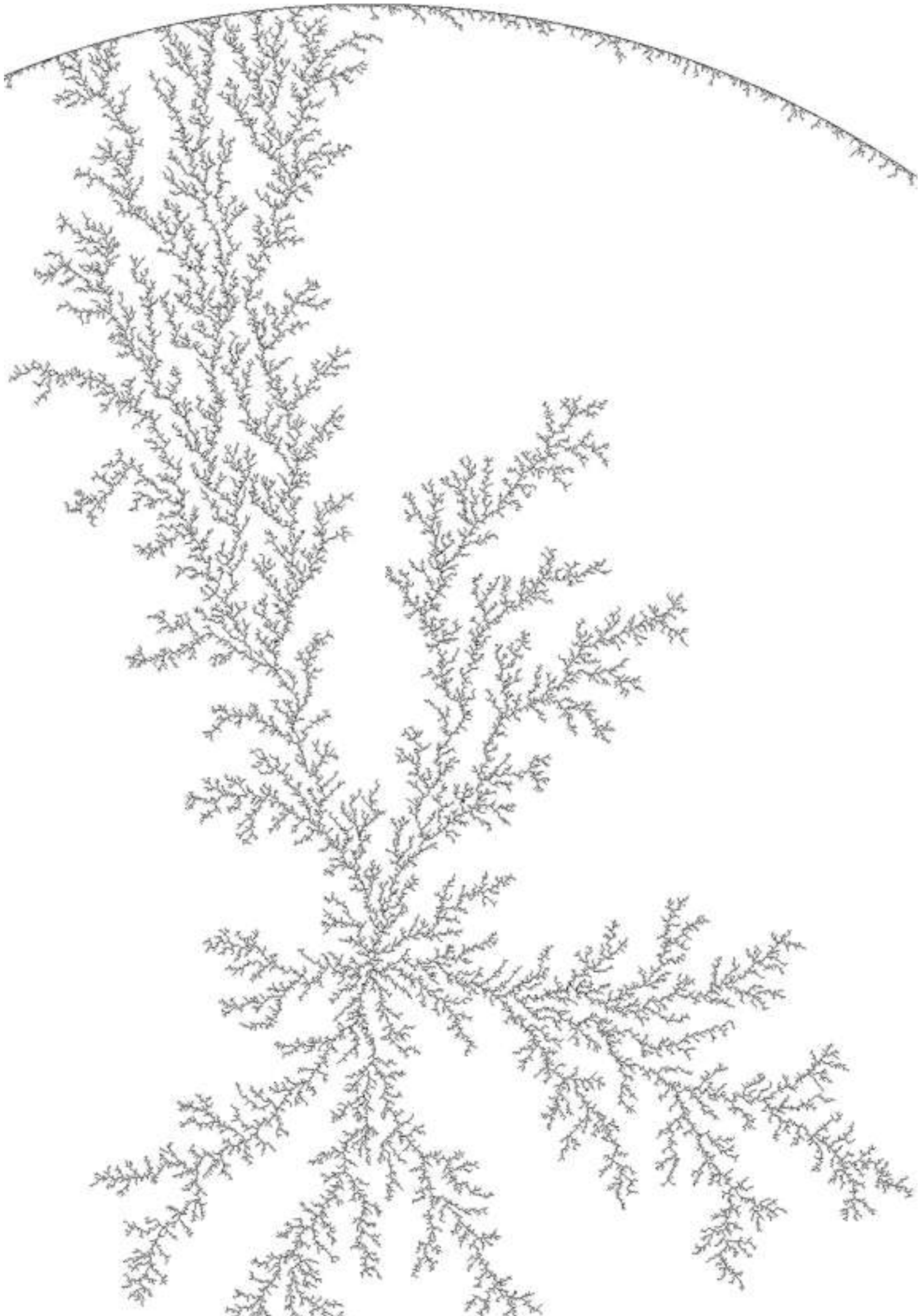
Gallery

Let us end up with a gallery of DLA fractals generated by the program. We leave it as an exercise to the reader to figure out the procedure used to create each of these.









Source Code of GUI

```
unit mainWin;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, ImageViewer, ExtCtrls, StdCtrls, Menus, ActnList, ComCtrls, DLASim,
  Themes, RejbrandCommon, ShellAPI, Clipbrd;

const MaxSize = 1048576;

type
  TmainFrm = class(TForm)
    ImageViewer: TImageViewer;
    ActionList: TActionList;
    aNew: TAction;
    MainMenu: TMainMenu;
    Fractal1: TMenuItem;
    New1: TMenuItem;
    aExit: TAction;
    Exit1: TMenuItem;
    N1: TMenuItem;
    aDrawPointInCentre: TAction;
    aDrawCircle: TAction;
    aDrawLineInMiddle: TAction;
    aDrawLineAtBottom: TAction;
    aDrawBoundary: TAction;
    Draw1: TMenuItem;
    Pointincentre1: TMenuItem;
    Linebottom1: TMenuItem;
    Linemiddle1: TMenuItem;
    Boundary1: TMenuItem;
    Circle1: TMenuItem;
    aSimulate: TAction;
    Simulate1: TMenuItem;
    aSimulate1: TMenuItem;
    aStop: TAction;
    Pause1: TMenuItem;
    aSave: TAction;
    Save1: TMenuItem;
    aSetSize: TAction;
    Fractal2: TMenuItem;
    Setsize1: TMenuItem;
    StatusBar1: TStatusBar;
    N2: TMenuItem;
    aClear: TAction;
    Clear1: TMenuItem;
    aSettings: TAction;
    Settings1: TMenuItem;
    N3: TMenuItem;
    aLoadBitmap: TAction;
    Open1: TMenuItem;
    AutoUpdater: TTimer;
    aAutoPreview: TAction;
    Autopreview1: TMenuItem;
    N4: TMenuItem;
    aPreviewNow: TAction;
    Previewnow1: TMenuItem;
    Help1: TMenuItem;
    aAbout: TAction;
    About1: TMenuItem;
    aModeCircularTrue: TAction;
    aModeUniform: TAction;
```

```
    aModeBoundary: TAction;
    aModeTop: TAction;
    N5: TMenuItem;
    Mode1: TMenuItem;
    Circular1: TMenuItem;
    Uniform1: TMenuItem;
    Boundary2: TMenuItem;
    op1: TMenuItem;
    aCircularCompensator: TAction;
    CircularCompensator1: TMenuItem;
    aCopy: TAction;
    Copy1: TMenuItem;
    N6: TMenuItem;
    Compute1: TMenuItem;
    aDrawRandomPoints: TAction;
    Randompoints1: TMenuItem;
    aModeCircularFast: TAction;
    Circularfast1: TMenuItem;
    aStatistics: TAction;
    Statistics1: TMenuItem;
    procedure btnSaveBMClick(Sender: TObject);
    procedure aExitExecute(Sender: TObject);
    procedure aNewExecute(Sender: TObject);
    procedure aSetSizeExecute(Sender: TObject);
    procedure aClearExecute(Sender: TObject);
    procedure aDrawPointInCentreExecute(Sender: TObject);
    procedure aDrawCircleExecute(Sender: TObject);
    procedure aDrawLineInMiddleExecute(Sender: TObject);
    procedure aDrawLineAtBottomExecute(Sender: TObject);
    procedure aDrawBoundaryExecute(Sender: TObject);
    procedure aSimulateExecute(Sender: TObject);
    procedure aStopExecute(Sender: TObject);
    procedure UpdateImage;
    procedure aSimulateUpdate(Sender: TObject);
    procedure aStopUpdate(Sender: TObject);
    procedure aSaveExecute(Sender: TObject);
    procedure NotWhileRunning(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
    procedure aSettingsExecute(Sender: TObject);
    procedure aLoadBitmapExecute(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure AutoUpdaterTimer(Sender: TObject);
    procedure aAutoPreviewExecute(Sender: TObject);
    procedure aPreviewNowExecute(Sender: TObject);
    procedure aAboutExecute(Sender: TObject);
    procedure aModeCircularTrueExecute(Sender: TObject);
    procedure aModeUniformExecute(Sender: TObject);
    procedure aModeBoundaryExecute(Sender: TObject);
    procedure aModeCircularTrueUpdate(Sender: TObject);
    procedure aModeUniformUpdate(Sender: TObject);
    procedure aModeBoundaryUpdate(Sender: TObject);
    procedure aCircularCompensatorUpdate(Sender: TObject);
    procedure aCircularCompensatorExecute(Sender: TObject);
    procedure aCopyExecute(Sender: TObject);
    procedure aDrawRandomPointsExecute(Sender: TObject);
    procedure aModeCircularFastExecute(Sender: TObject);
    procedure aModeCircularFastUpdate(Sender: TObject);
    procedure aStatisticsExecute(Sender: TObject);
  private
    procedure SetStatus(const Status: string); overload;
    procedure SetStatus; overload;
    procedure TaskDialogHyperLinkClicked(Sender: TObject);
    { Private declarations }
  public
    { Public declarations }
    DLAFractal: TDLAFractal;
```

```
  end;

var
  mainFrm: TmainFrm;

implementation

uses SuperDialog, settingsWin;

{$R *.dfm}

procedure TmainFrm.TaskDialogHyperLinkClicked(Sender: TObject);
begin
  if Sender is TTaskDialog then
    with Sender as TTaskDialog do
      ShellExecute(Handle, 'open', PChar(URL), nil, nil, SW_SHOWNORMAL);
end;

procedure TmainFrm.aAboutExecute(Sender: TObject);
begin
  if (Win32MajorVersion >= 6) and ThemeServices.ThemesEnabled then
    with TTaskDialog.Create(self) do
      try
        Caption := 'About DLA Simulator';
        Title := 'DLA Simulator';
        CommonButtons := [tcbClose];
        Text := 'File Version: ' + GetFileVer(Application.ExeName) + #13#10#13#10'C
opyright © 2012 Andreas Rejbrand'#13#10#13#10'<a href="http://english.rejbrand.se">
http://english.rejbrand.se</a>';
        Flags := [tfUseHiconMain, tfEnableHyperlinks];
        CustomMainIcon := Application.Icon;
        OnHyperlinkClicked := TaskDialogHyperlinkClicked;
        Execute;
      finally
        Free;
      end
  else
    // Windows XP Compatibility Code
    MessageBox(Handle, PChar('File Version: ' + GetFileVer(Application.ExeName) + #
13#10#13#10 + 'Copyright © 2012 Andreas Rejbrand' + #13#10#13#10 + 'http://english.
rejbrand.se' + #13#10#13#10 + 'DLA Simulator is running in Windows XP Compatibility
 Mode.'), PChar('DLA Simulator'), MB_ICONINFORMATION);
end;

procedure TmainFrm.aAutoPreviewExecute(Sender: TObject);
begin
  {SIC!};
end;

procedure TmainFrm.aCircularCompensatorExecute(Sender: TObject);
begin
  DLAFractal.CircularCompensator := not DLAFractal.CircularCompensator;
end;

procedure TmainFrm.aCircularCompensatorUpdate(Sender: TObject);
begin
  aCircularCompensator.Checked := DLAFractal.CircularCompensator or (DLAFractal.Mod
e = dlaCircularTrue);
  aCircularCompensator.Enabled := (DLAFractal.Mode <> dlaCircularTrue) and not DLAF
ractal.Running;
end;

procedure TmainFrm.aClearExecute(Sender: TObject);
begin
  DLAFractal.FillWhite;
  UpdateImage;
```

```
end;

procedure TmainFrm.aCopyExecute(Sender: TObject);
begin
  Clipboard.Assign(ImageViewer.Bitmap);
end;

procedure TmainFrm.aDrawBoundaryExecute(Sender: TObject);
begin
  DLAFractal.DrawBoundary;
  UpdateImage;
end;

procedure TmainFrm.aDrawCircleExecute(Sender: TObject);
var
  r: integer;
begin
  r := 100;
  if TMultiInputBox.NumInputBox(Self, 'Draw Circle',
    'Please enter the radius of the circle:', 0, MaxSize, r) then
  begin
    DLAFractal.DrawCircle(r);
    UpdateImage;
  end;
end;

procedure TmainFrm.aDrawLineAtBottomExecute(Sender: TObject);
begin
  DLAFractal.DrawLineAtBottom;
  UpdateImage;
end;

procedure TmainFrm.aDrawLineInMiddleExecute(Sender: TObject);
begin
  DLAFractal.DrawLineAtMiddle;
  UpdateImage;
end;

procedure TmainFrm.aDrawPointInCentreExecute(Sender: TObject);
begin
  DLAFractal.DrawPointAtCentre;
  UpdateImage;
end;

procedure TmainFrm.aDrawRandomPointsExecute(Sender: TObject);
var
  n: integer;
begin
  n := 5;
  if TMultiInputBox.NumInputBox(Self, 'Draw Points',
    'Please enter the number of points to draw:', 0, MaxSize, n) then
  begin
    DLAFractal.DrawRandomPoints(n);
    UpdateImage;
  end;
end;

procedure TmainFrm.aExitExecute(Sender: TObject);
begin
  Close;
end;

procedure TmainFrm.aLoadBitmapExecute(Sender: TObject);
begin
  with TOpenDialog.Create(nil) do
    try
```

```
      Filter := '32-bit Windows Bitmap|*.bmp';
      if Execute then
      begin
        DLAFractal.LoadFromFile(FileName);
        UpdateImage;
      end;
    finally
      Free;
    end;
end;

procedure TmainFrm.aModeBoundaryExecute(Sender: TObject);
begin
  DLAFractal.Mode := dlaBoundary;
end;

procedure TmainFrm.aModeBoundaryUpdate(Sender: TObject);
begin
  aModeBoundary.Checked := DLAFractal.Mode = dlaBoundary;
  aModeBoundary.Enabled := not DLAFractal.Running;
end;

procedure TmainFrm.aModeCircularFastExecute(Sender: TObject);
begin
  DLAFractal.Mode := dlaCircularFast;
end;

procedure TmainFrm.aModeCircularFastUpdate(Sender: TObject);
begin
  aModeCircularFast.Checked := DLAFractal.Mode = dlaCircularFast;
  aModeCircularFast.Enabled := not DLAFractal.Running;
end;

procedure TmainFrm.aModeCircularTrueExecute(Sender: TObject);
begin
  DLAFractal.Mode := dlaCircularTrue;
  DLAFractal.CircularCompensator := false;
end;

procedure TmainFrm.aModeCircularTrueUpdate(Sender: TObject);
begin
  aModeCircularTrue.Checked := DLAFractal.Mode = dlaCircularTrue;
  aModeCircularTrue.Enabled := not DLAFractal.Running;
end;

procedure TmainFrm.aModeUniformExecute(Sender: TObject);
begin
  DLAFractal.Mode := dlaUniform;
end;

procedure TmainFrm.aModeUniformUpdate(Sender: TObject);
begin
  aModeUniform.Checked := DLAFractal.Mode = dlaUniform;
  aModeUniform.Enabled := not DLAFractal.Running;
end;

procedure TmainFrm.aNewExecute(Sender: TObject);
begin
  DLAFractal.Free;
  DLAFractal := TDLAFractal.Create;
  DLAFractal.SetSize(512, 512);
  DLAFractal.FillWhite;
  UpdateImage;
end;

procedure TmainFrm.aPreviewNowExecute(Sender: TObject);
```

```
begin
  UpdateImage;
end;

procedure TmainFrm.aSaveExecute(Sender: TObject);
begin
  with TSaveDialog.Create(nil) do
    try
      Filter := '32-bit Windows Bitmap|*.bmp';
      if Execute then
        DLAFractal.SaveToFile(FileName);
    finally
      Free;
    end;
end;

procedure TmainFrm.aSetSizeExecute(Sender: TObject);
var
  w, h: integer;
begin
  w := DLAFractal.Width;
  h := DLAFractal.Height;
  if TMultiInputBox.NumInputBox(Self, 'Image Size',
    'Please enter the width of the image:', 6, MaxSize, w) then
    if TMultiInputBox.NumInputBox(Self, 'Image Size',
      'Please enter the height of the image:', 6, MaxSize, h) then
    begin
      DLAFractal.SetSize(w, h);
      DLAFractal.FillWhite;
      UpdateImage;
    end;
end;

procedure TmainFrm.aSettingsExecute(Sender: TObject);
begin
  with TsettingsFrm.Create(nil) do
    try
      ShowModal;
    finally
      Free;
    end;
end;

procedure TmainFrm.aSimulateExecute(Sender: TObject);
begin
  DLAFractal.Simulate;
  SetStatus('Simulating');
  AutoUpdater.Enabled := true;
end;

procedure TmainFrm.aSimulateUpdate(Sender: TObject);
begin
  aSimulate.Enabled := not DLAFractal.Running;
end;

procedure TmainFrm.aStatisticsExecute(Sender: TObject);
begin
  with DLAFractal do
    MessageBox(Handle,
      PChar(Format('Width: %d'#13#10'Height: %d'#13#10'Radius: %f'#13#10 +
        'Number of particles: %d', [Width, Height, Radius, ParticleCount])),
      PChar('Statistics'), MB_ICONINFORMATION);
end;

procedure TmainFrm.aStopExecute(Sender: TObject);
begin
```

```
  AutoUpdater.Enabled := false;
  DLAFractal.Pause;
  UpdateImage;
  SetStatus;
end;

procedure TmainFrm.aStopUpdate(Sender: TObject);
begin
  aStop.Enabled := DLAFractal.Running;
end;

procedure TmainFrm.AutoUpdaterTimer(Sender: TObject);
begin
  if aAutoPreview.Checked then UpdateImage;
end;

procedure TmainFrm.btnSaveBMClick(Sender: TObject);
begin
  with TSaveDialog.Create(nil) do
    try
      Filter := '32-bit Windows Bitmap|*.bmp';
      if Execute then
        ImageViewer.Bitmap.SaveToFile(FileName);
    finally
      Free;
    end;
end;

procedure TmainFrm.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  CanClose := true;
  if DLAFractal.Running then
  begin
    CanClose := MessageBox(Handle, 'Do you want to terminate the ongoing simulation
 and close the application?',
      'Simulation in progress', MB_ICONQUESTION or MB_YESNO) = ID_YES;
    if CanClose then DLAFractal.Pause;
  end;
end;

procedure TmainFrm.FormCreate(Sender: TObject);
begin
  DLAFractal := TDLAFractal.Create;
  DLAFractal.SetSize(512, 512);
  DLAFractal.FillWhite;
  DLAFractal.DrawPointAtCentre;
  UpdateImage;
end;

procedure TmainFrm.NotWhileRunning(Sender: TObject);
begin
  if Sender is TAction then
    with Sender as TAction do
      Enabled := not DLAFractal.Running;
end;

procedure TmainFrm.SetStatus;
begin
  Caption := 'DLA Simulator';
end;

procedure TmainFrm.UpdateImage;
begin
  ImageViewer.Bitmap.Free;
  ImageViewer.Bitmap := DLAFractal.CreateGDIBitmap;
end;
```

```

procedure TmainFrm.SetStatus(const Status: string);
begin
  Caption := Format('DLA Simulator [%s]', [Status]);
end;

end.
```

See the attached ZIP file for full source code, including DPR and DFM files.

Appendix A: The Effect of the 'Varying-Radius' Approximation

If you haven't already concluded what effect the 'varying-radius' approximation have on the resulting bitmaps, perhaps the following samples will give you a hint.

True Circular Mode	Fast Circular Mode