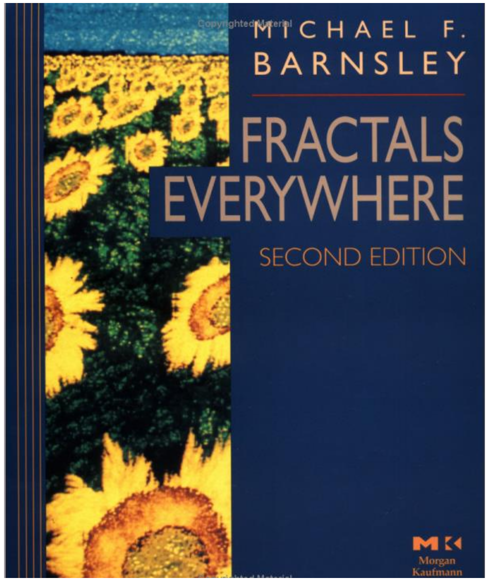
Hills = Clouds

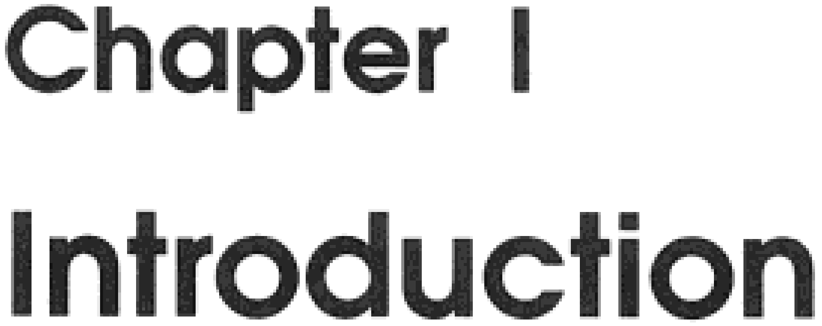
12 Fractals

Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

R. Mukundan (mukundan@canterbury.ac.nz)

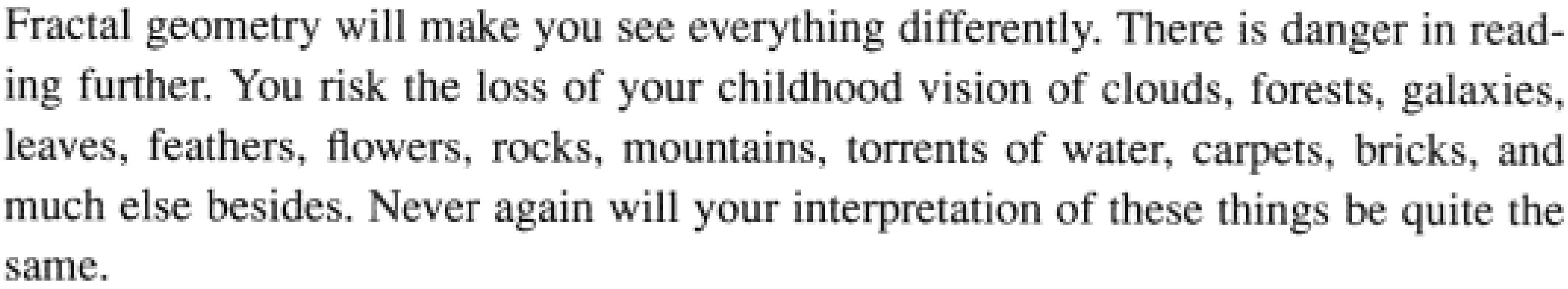
From the book, “Fractals Everywhere” by Michael Barnsley:





Chapter I

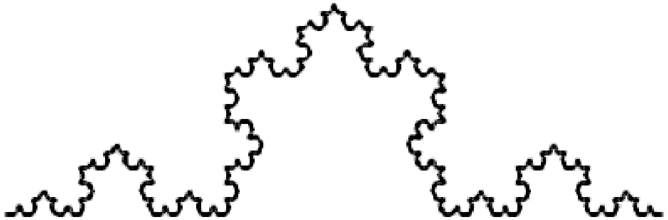
Introduction



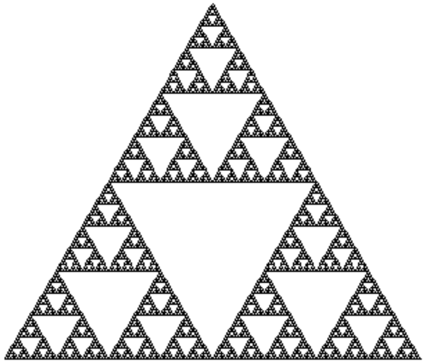
Fractal geometry will make you see everything differently. There is danger in reading further. You risk the loss of your childhood vision of clouds, forests, galaxies, leaves, feathers, flowers, rocks, mountains, torrents of water, carpets, bricks, and much else besides. Never again will your interpretation of these things be quite the same.

Fractals

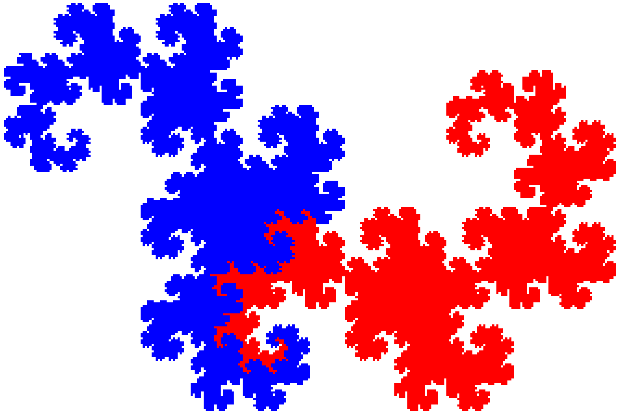
Fractals exhibit the property of *self similarity* –it is a structure that can repeat itself at every smaller scale to produce complex patterns.



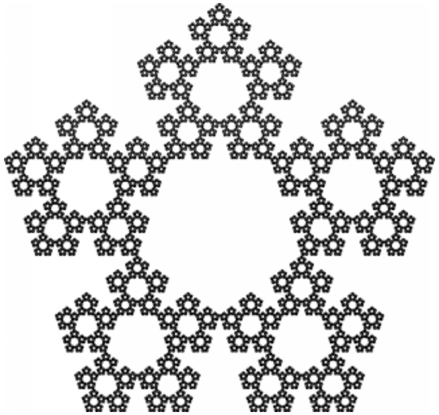
von Koch Curve



Sierpinski Triangle



The Dragon Curve



Sierpinski Pentagon

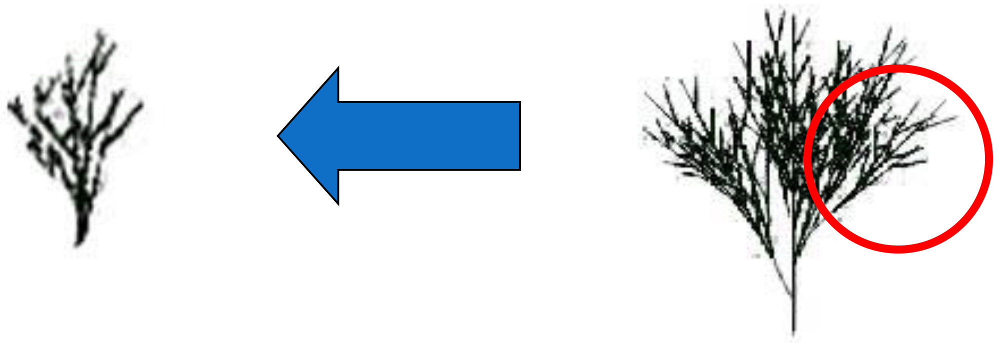
Fractals and Self-Similarity

- **Exact Self-similarity:**

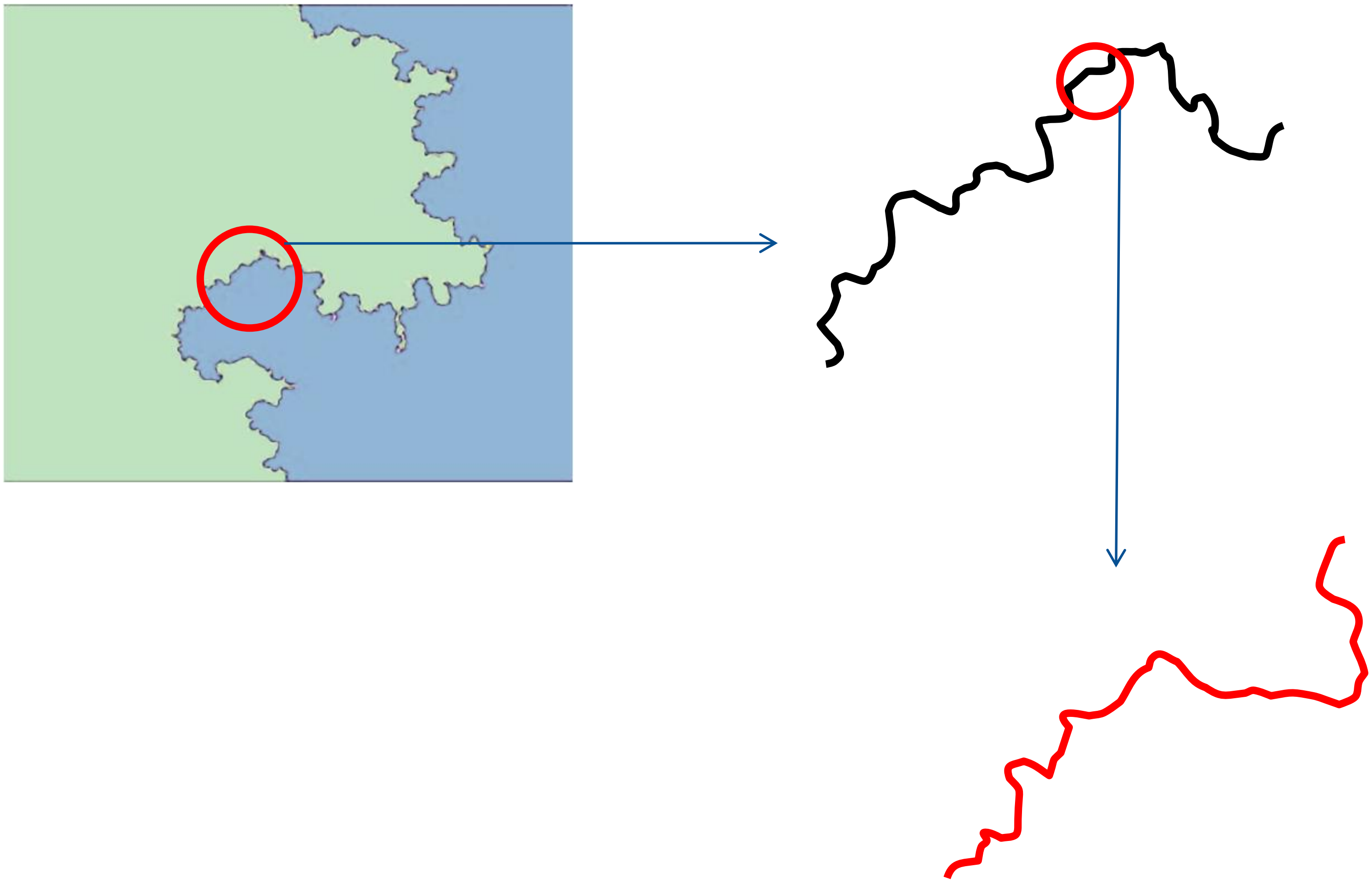
  Each small portion of the fractal can be viewed as a reduced-scale replica of the whole (except for a possible rotation and shift).

- **Statistical Self-similarity:**

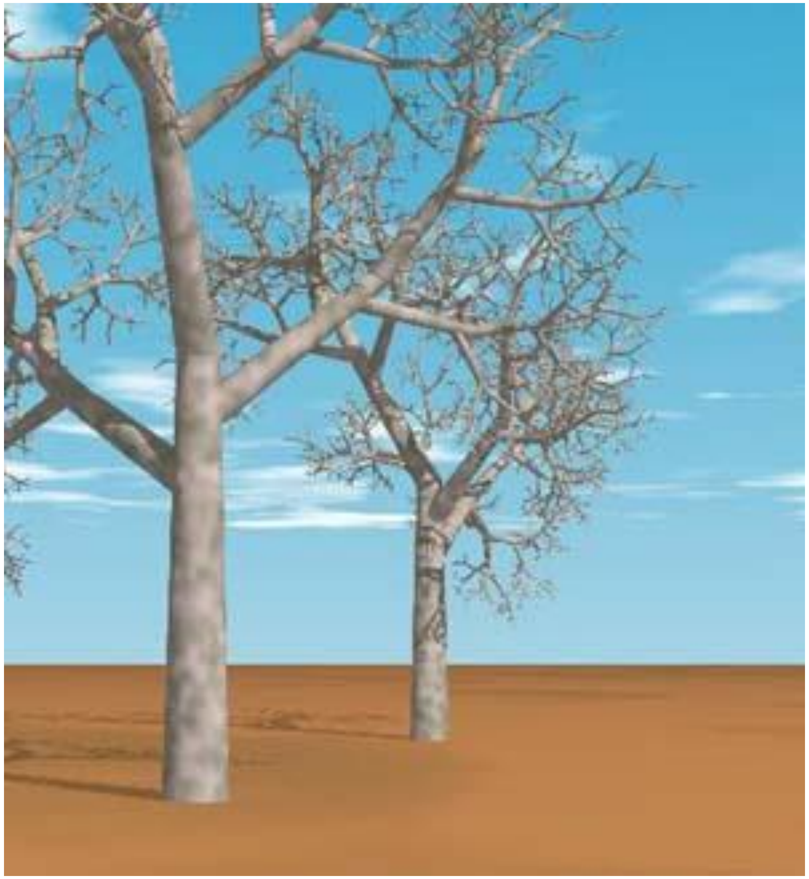
  The irregularities in the curve are the same on the average, no matter how many times the picture is enlarged.



Fractal Coastline



3D Fractals



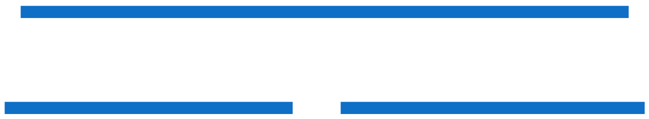
Fractals are used in computer modeling of irregular patterns and structures in nature (eg. trees, clouds, leaves, terrains).

Dimension

If a shape can be divided into *N* smaller copies of itself, each scaled down by a factor *s* given by

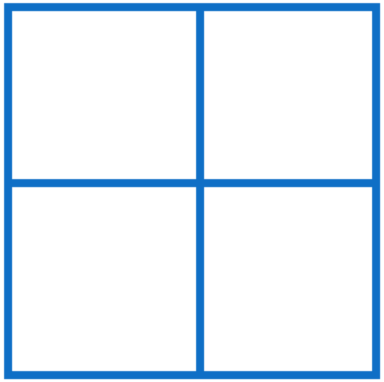
$$s = \frac{1}{N^{\frac{1}{D}}}$$

where *D* is a constant, then the curve is said to have a dimension *D*. Regular shapes such as lines, and squares have integral dimensions.



$N = 2, \; s = ½$

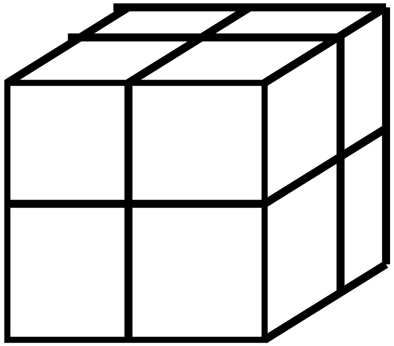
$\therefore D = 1$



$N = 4, \; s = ½$

$\therefore D = 2$

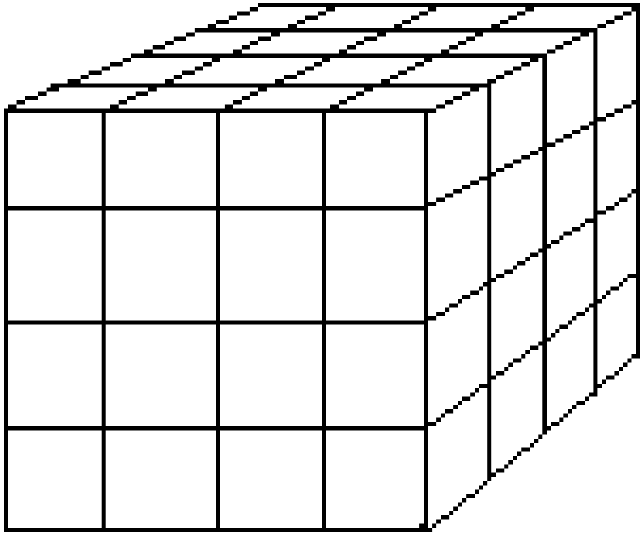
Dimension



$s = 1/2$

$N = 8$

$\therefore D = 3$



$s = 1/4$

$N = 64$

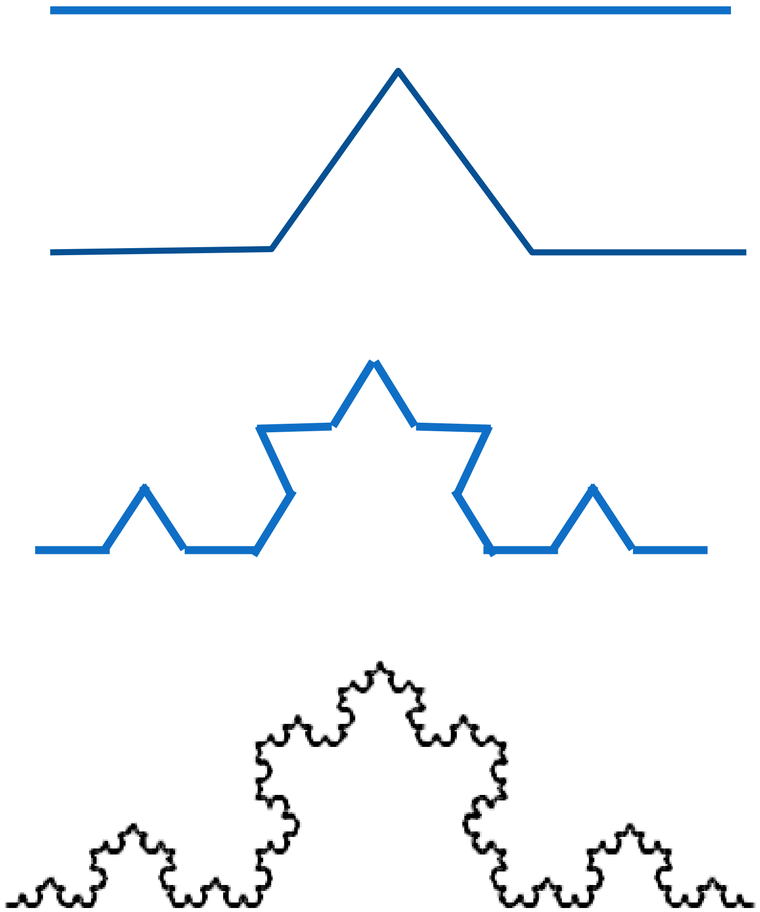
$\therefore D = 3$

Fractional Dimension – Eg.1

Fractal shapes have fractional dimensions!

The von-Koch Curve

(Snowflake Curve)



$$D = \frac{\log N}{\log\left(\frac{1}{s}\right)}$$

$$N = 4, \;\; s = 1/3$$

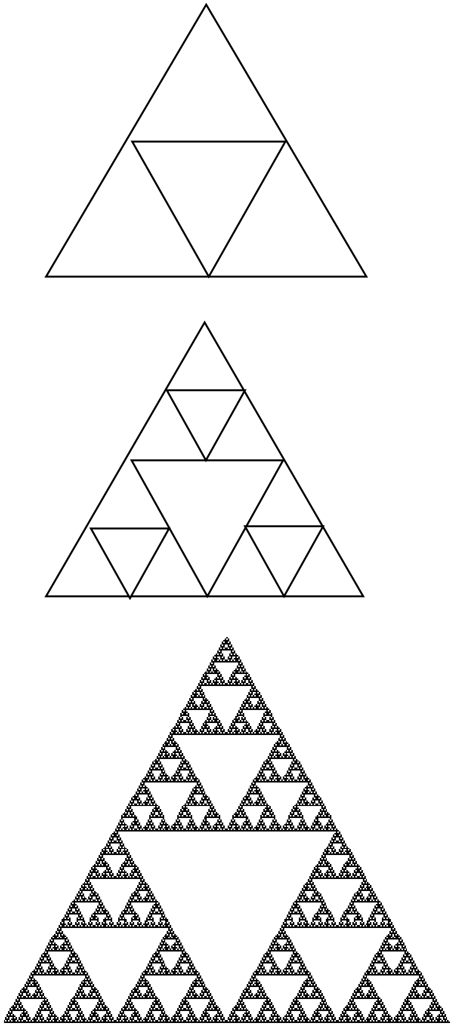
$$\therefore D = 1.26$$

Fractional Dimension – Eg. 2

The Sierpinski Triangle

$$D = \frac{\log N}{\log\left(\frac{1}{s}\right)}$$

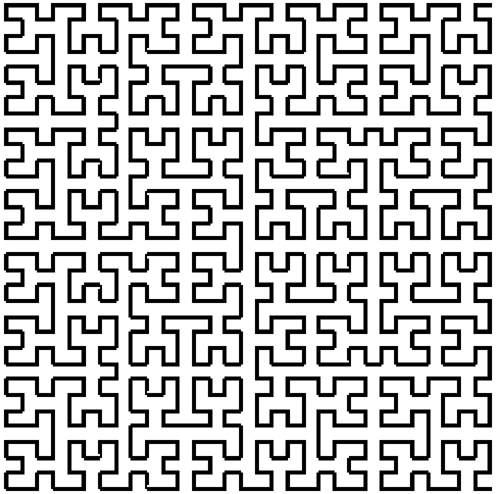
$$N = 3,\ \ s = ½$$
$$\therefore D = 1.584$$

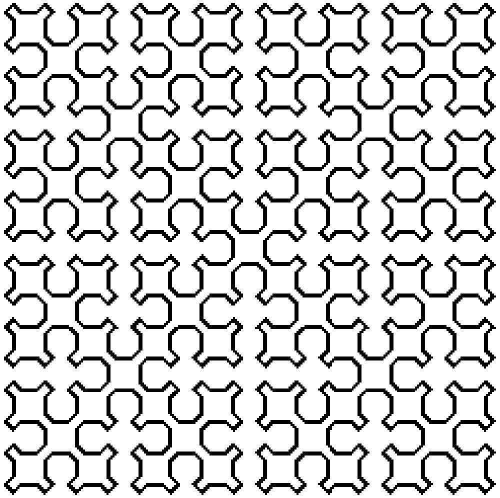


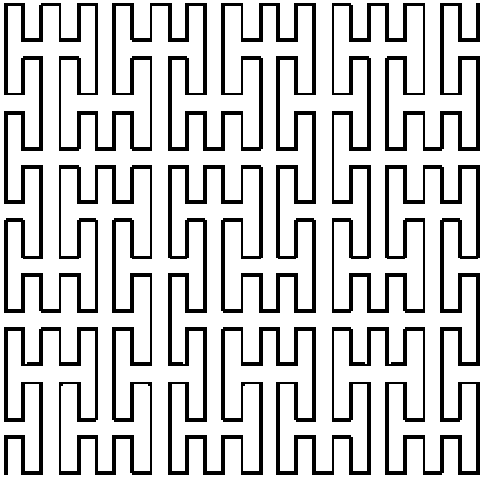
Space-Filling Curves

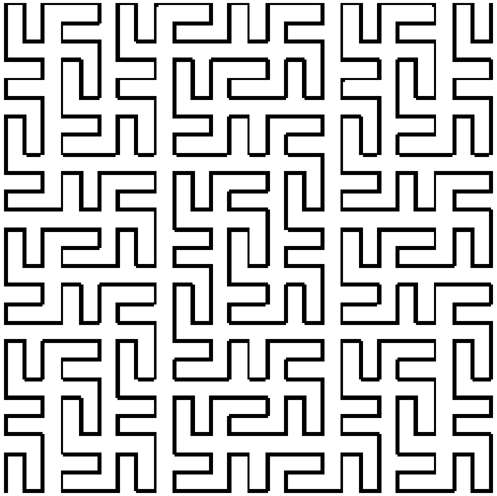
- There are fractal curves which completely fill up higher dimensional spaces such as squares or cubes.
- The space-filling curves are also known as Peano curves (Giuseppe Peano: 1858-1932).
- Space-filling curves in 2D have a fractal dimension 2.

Space-Filling Curves

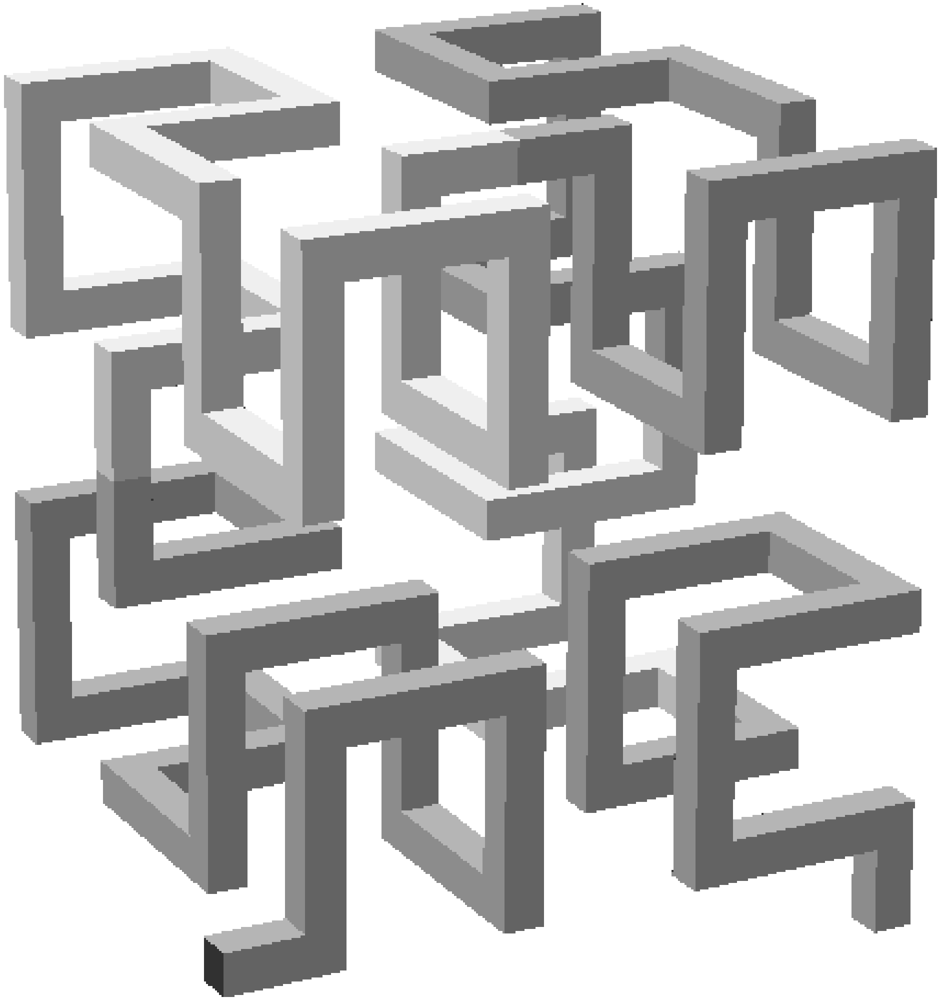








Space-Filling Curves in 3D



Generating Fractals

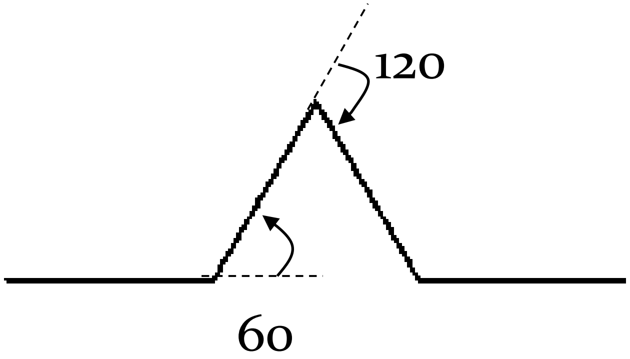
- Iterative/recursive subdivision techniques.
- Grammar based systems (L-systems).
  - Suitable for turtle graphics/vector devices.
- Iterated Functions Systems (IFS).
  - Suitable for raster devices.

Grammar-Based Models

- The grammar-based model for generating simple fractal curves was originally proposed by Lindenmayer, and is therefore known as L-systems.
- L-systems are particularly suited for rendering line drawings of fractal curves using turtle graphics.
- An L-system typically uses the following set of symbols:
  - F   Move forward one unit in the current direction.
  - +   Turn right through an angle *A*.
  - – Turn left through an angle *A*.

L-Systems

F–F++F–F (Angle = 60 degs)

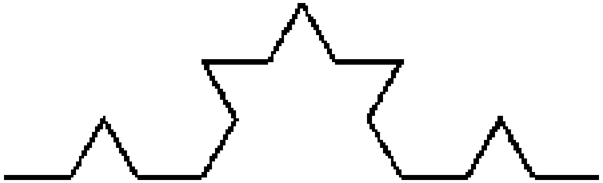


First order Koch Curve

String Production Rule:

F → F–F++F–F

$\underline{\text{F–F++F–F}}\text{–}\underline{\text{F–F++F–F}}\text{++}\underline{\text{F–F++F–F}}\text{–}\underline{\text{F–F++F–F}}$



Second order Koch Curve

L-Systems

- The starting string is called the “atom” (sometimes denoted by the symbol S). The atom also represents zeroth-order curve.

Koch Curve:

S → F

F → F−F++F−F

(*A*=60 degs.)

Zero order Koch Curve

- An L-System can also contain the variables X and Y, with their own productions (rules). These variables are used only in deriving a string, and they do not have any associated functions while drawing the curve.

L-Systems

The Dragon Curve:

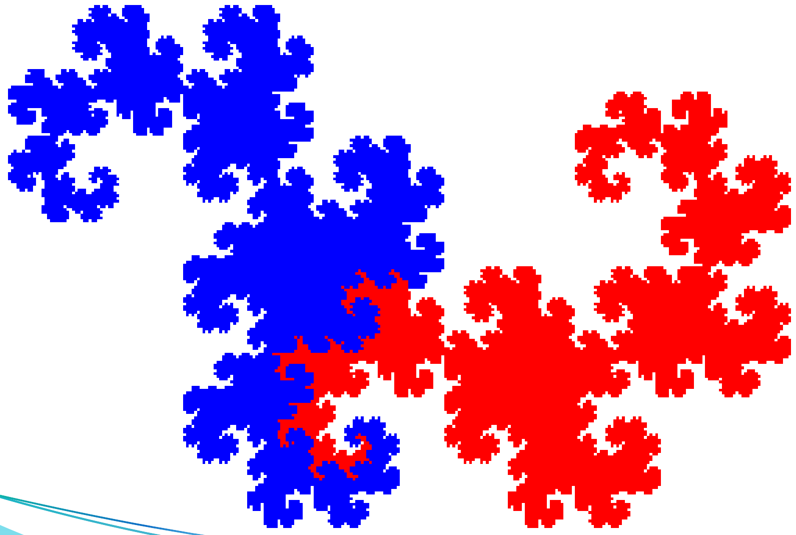
S → FX

F → F

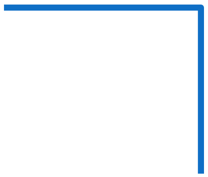
X → X+YF+

Y → –FX –Y

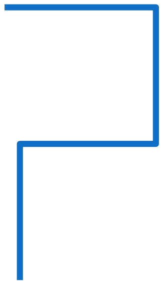
(*A* = 90 degs.)



Zero order dragon
FX



First order dragon
FX+YF+



Second order dragon
FX+YF++ –FX –YF+

L-System (Code)

```
void produceString(char *st, int order)
{
    for( ; *st; st++)
    switch(*st)
    {
      case '+': dir -= angle; break;  // right turn
      case '-': dir += angle; break;  // left turn
      case 'F': if(order > 0)
            produceString(f_str, order-1);
            else forward(1, 1); break;
      case 'X': if(order > 0)
            produceString(x_str, order - 1); break;
      case 'Y': if(order > 0)
            produceString(y_str, order - 1);
    }
}
```

Generalized Grammars

- The grammar rules in L-systems can be further generalized to provide the capability of drawing branchlike figures, rather than just continuous curves.
- The symbol [is used to store the current state of the turtle (position and direction) in a stack for later use.
- The symbol] is used to perform a pop operation on the stack to restore the turtle's state to a previously stored value.

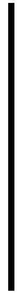
Generalized Grammars

Fractal bush:

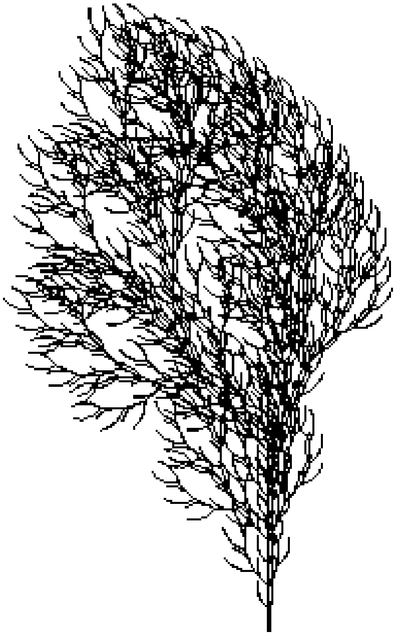
S → F

F → FF–[–F+F+F]+[+F–F–F]

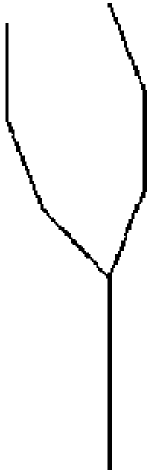
(*A* = 22 degs.)



Zero order bush



Fourth order bush
(with 90 deg. rotation)



First order bush

Iterated Function Systems

An iterated function system (IFS) is a collection of *N* affine transformations T_i , *i* =1,2,…*N*, which are applied iteratively to an image to produce a fractal image. Each affine map T_i is defined using six parameters as

$T_i = \{m_{11}, m_{12}, m_{21}, m_{22}, m_{13}, m_{23}\}$,

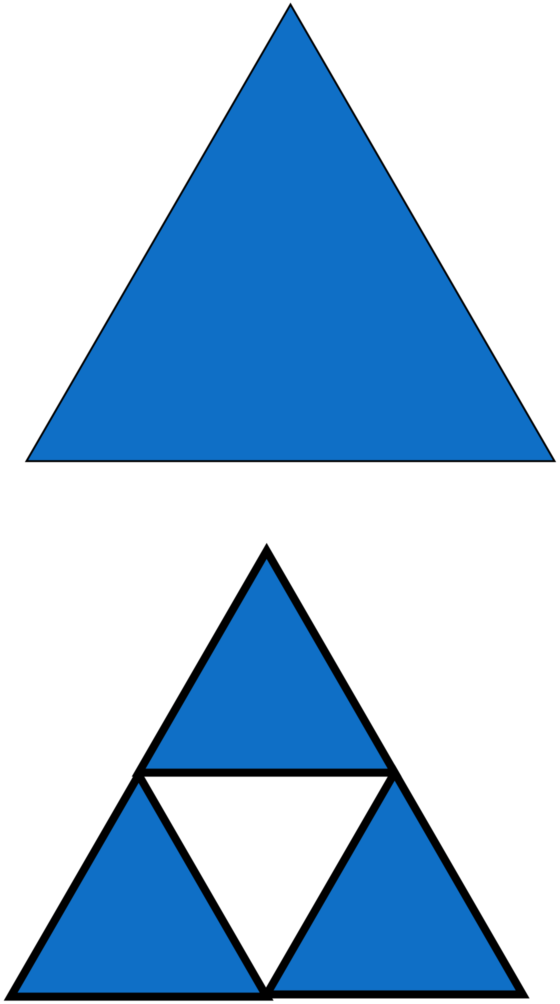
which represent a 2D transformation:

$$x' = m_{11}x + m_{12}y + m_{13}$$

$$y' = m_{21}x + m_{22}y + m_{23}$$

IFS: Example

The Sierpinski triangle is generated by taking three copies of the triangle in the previous step, uniformly scaling by a factor of 0.5, and finally translating the positions of the three triangles by vectors (0, 0), (0.5, 0), (0.25, 0.5) respectively.



$T_1 = \{0.5, 0.0, 0.0, 0.5, 0.0, 0.0\}$

$T_2 = \{0.5, 0.0, 0.0, 0.5, 0.5, 0.0\}$

$T_3 = \{0.5, 0.0, 0.0, 0.5, 0.25, 0.5\}$

IFS: Pseudo Code

```
drawFract(Array points, int iteration)
{
  if(iteration==0) draw(points);
   else{
       for(int i=0; i<N; i++){
           transform(i, points, newpoints);
           drawFract(newpoints, iteration-1);
       }
    }
}
```

Each level of recursion produces a new object, which finally converges to a unique image called the *attractor* of the IFS.

IFS: Shortcomings

- Requires many recursive calls, each with some computational overhead.

- Each iteration creates *N* copies of the previous figure. If *n* denotes the number of points in the previous iteration, the current iteration will generate *N.n* new points. Thus the k^{th} will generate $N^k n$ points. The program thus requires a huge memory allocation to store all points.

Random Iteration Algorithm

In each iteration, an affine map is chosen at random with the assigned probability, and is used to transform a point *P* to a new point. The new point is drawn and becomes the point to be transformed in the next iteration.

Affine map for a random iteration algorithm:

$T_i = \{m_{11}, m_{12}, m_{21}, m_{22}, m_{13}, m_{23}\},\ \ p_i$

$i = 1,2,\ldots,N.$

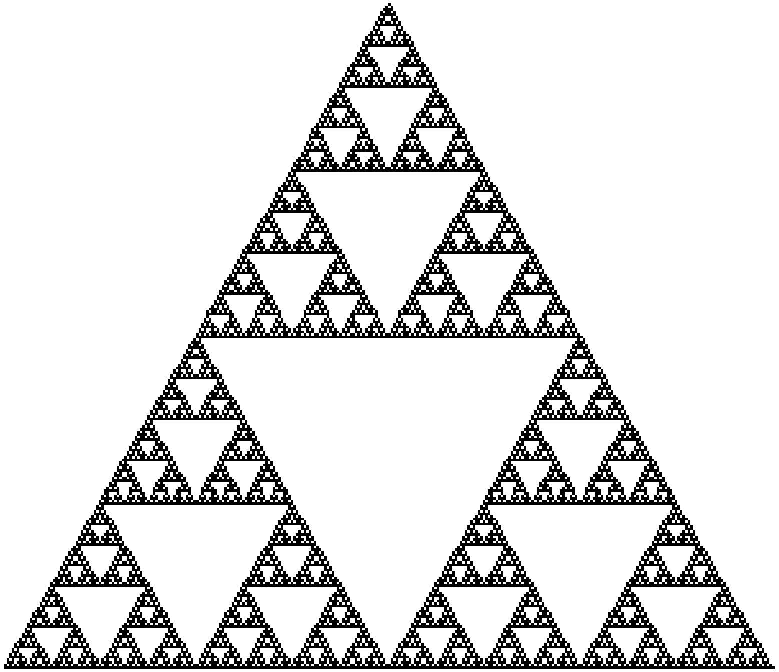
Random Iteration Algorithm

- Starting from a point P_0, the iterations through this system produces a sequence of points P_0, P_1, P_2, P_3, … which is called the *orbit* of the system starting from the point P_0. The set of all *points* produces the the *attractor* of the IFS.
- The random iteration algorithm replaces recursion (in the deterministic IFS) with iteration, and the whole image is generated as an orbit of a single point.
- Suitable for raster devices.

Random Iteration Algorithm

```
drawFract()
{
Point p={0,0};
k = 0;
  while(k<50000){     //Lots of points!
      r =  random();
      i = chooseAffine(r);
      p = transform(i, p);
      draw(p);
      k++;
   }
}
```

Random IFS: Example



T_1 = {0.5, 0.0, 0.0, 0.5, 0.0, 0.0}; 0.33

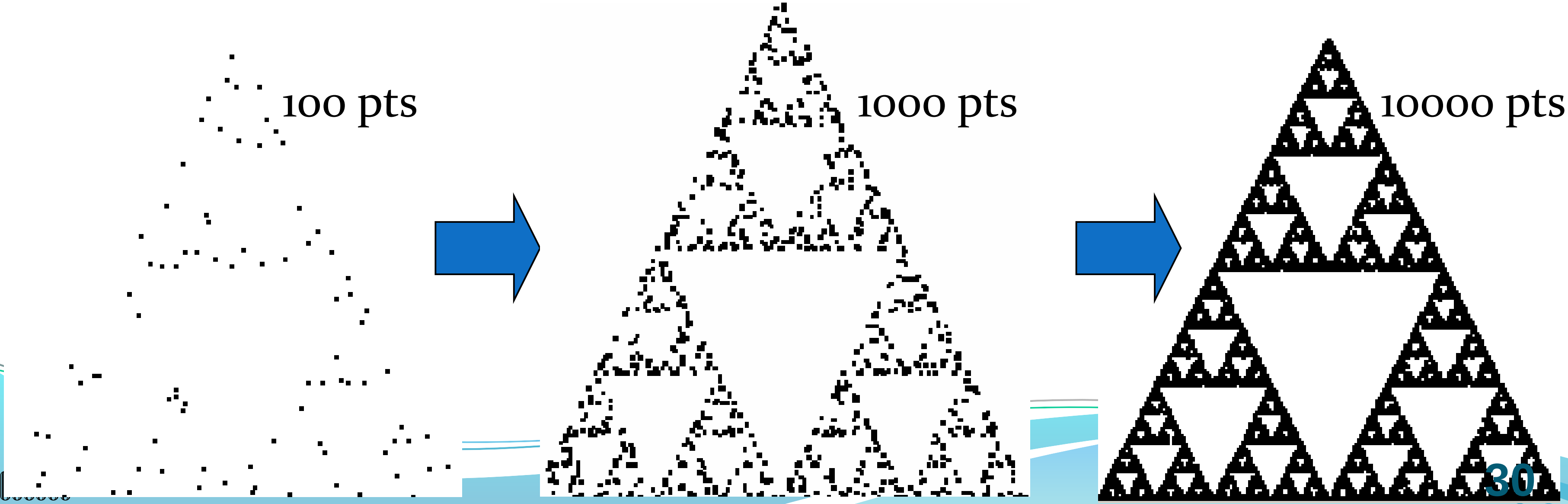
T_2 = {0.5, 0.0, 0.0, 0.5, 0.5, 0.0}; 0.33

T_3 = {0.5, 0.0, 0.0, 0.5, 0.25, 0.5}; 0.33

Random IFS

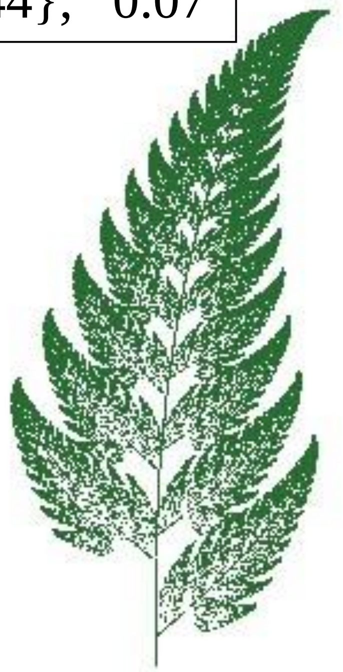
The fundamental properties of random-IFS:

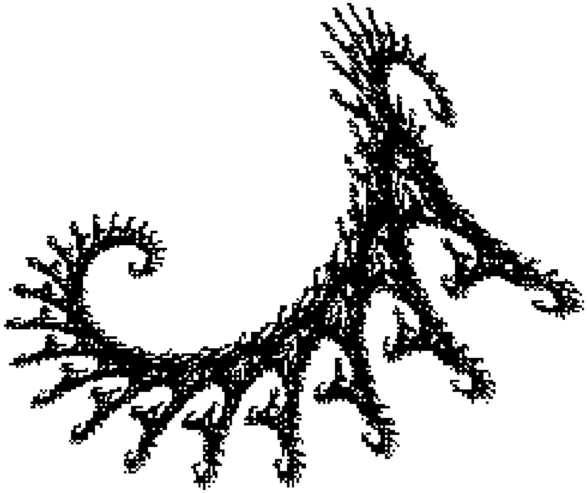
1. The orbit of a point for a given IFS converges to an attractor.
2. All points in an attractor produce the same figure.
3. Any point in an attractor, when transformed using a function in an IFS, gives another point in the same attractor.



Random IFS: Examples

T_1 = {0.0, 0.0, 0.0, 0.16, 0.0, 0.0}; 0.01
T_2 = {0.85, 0.04, – 0.04, 0.85, 0.0, 1.6}; 0.85
T_3 = {0.2, – 0.26, 0.23, 0.22, 0.0, 1.6}; 0.07
T_4 = {– 0.15, 0.28, 0.26, 0.24, 0.0, 0.44}; 0.07





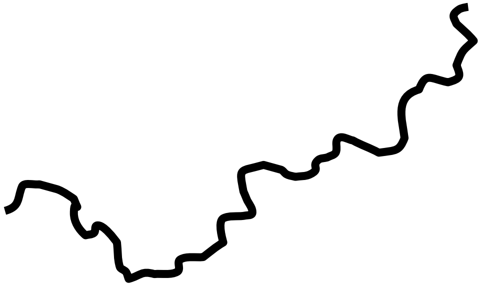
T_1 = {0.82~, 0.28~, –0.21~, 0.86~, –1.8~, –0.11~}; 0.787
T_2 = {0.08~, 0.52~, –0.46~, –0.37~, 0.78~, 8.09~}; 0.212

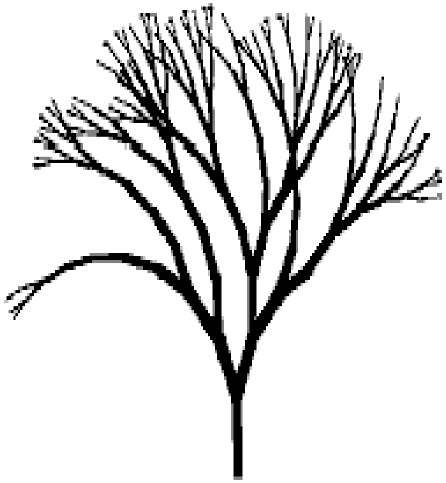
Random Fractals

- Natural objects do not contain identical scaled down copies within themselves and so are not exact fractals.
- Practically every example observed involves what appears to be some element of randomness, perhaps due to the interactions of very many small parts of the process.
- Almost all algorithms for generating fractal landscapes effectively add random irregularities to the surface at smaller and smaller scales.

Random Fractals

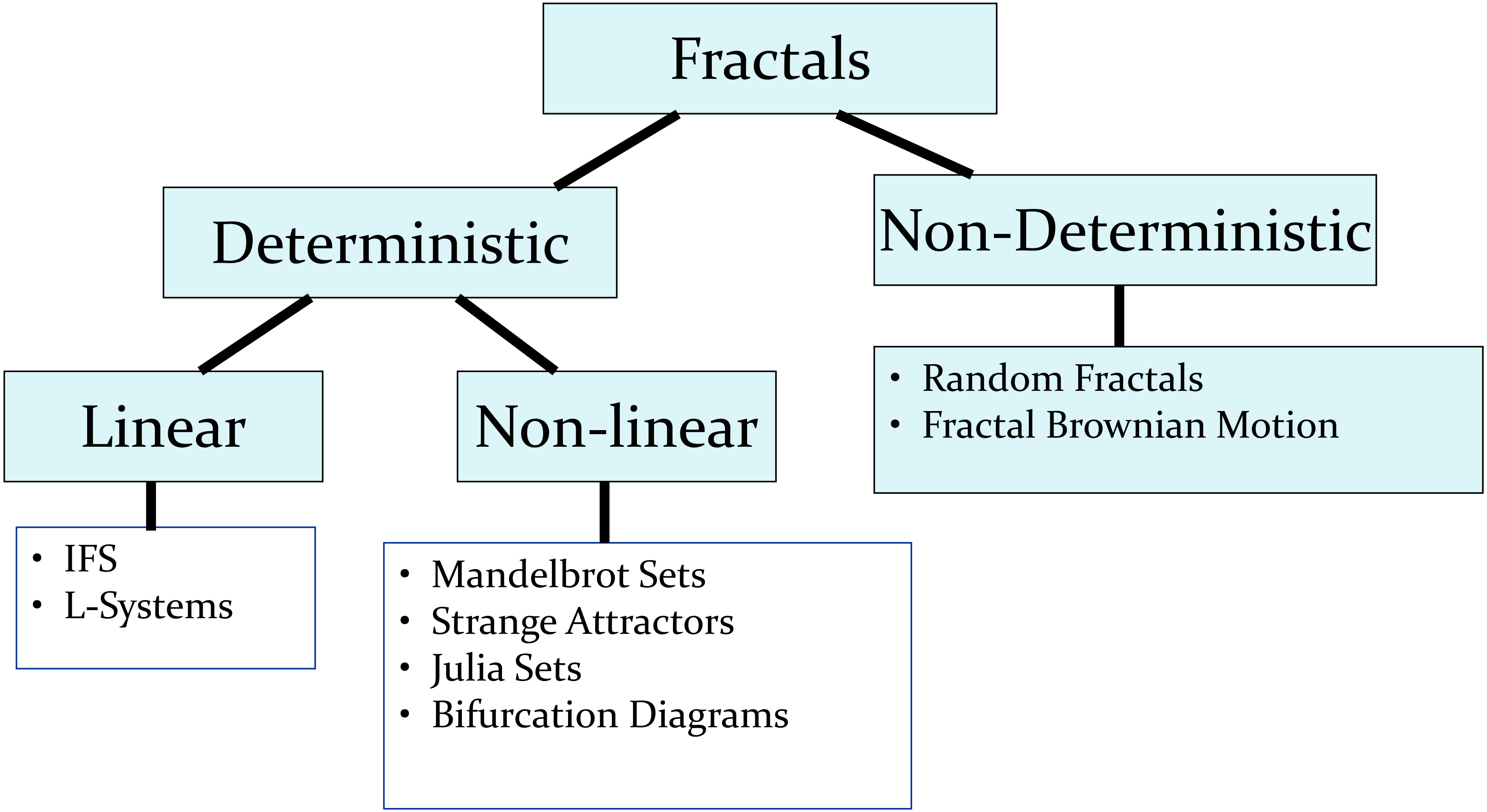
- Random fractals are
  - randomly generated curves that exhibit self-similarity, or
  - deterministic fractals modified using random variables while drawing line segments.
- Random fractals are used to model many "naturalistic" shapes such as trees, clouds, and mountains.





$F \rightarrow FF-[-F+F+F]+[+F-F-F]$

Classes of Fractals



Fractals

Deterministic

Non-Deterministic

Linear

Non-linear

- IFS
- L-Systems

- Mandelbrot Sets
- Strange Attractors
- Julia Sets
- Bifurcation Diagrams

- Random Fractals
- Fractal Brownian Motion

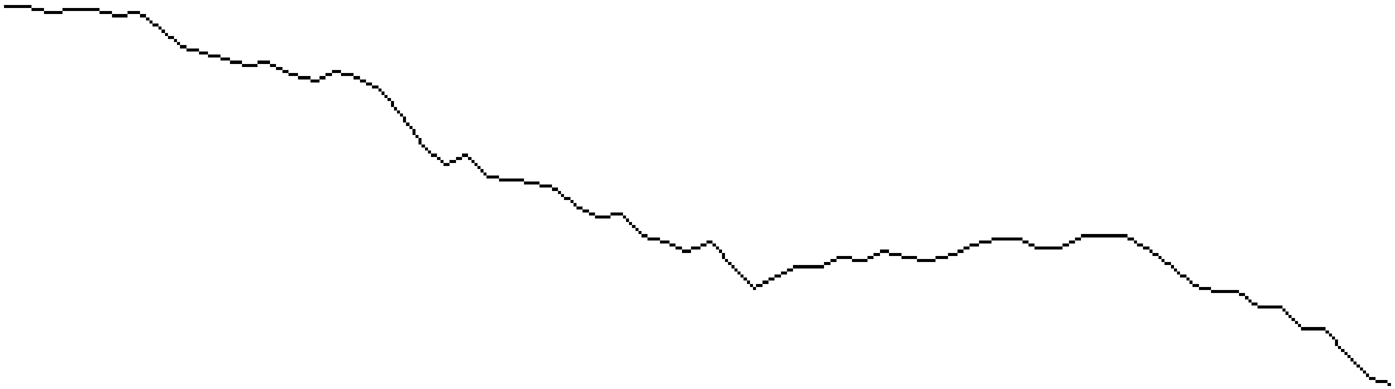
Fractional Brownian Motion (fBm)

- Fractional Brownian motion is also known as the "Random Walk Process." It basically consists of steps in a random direction and with a step-length that has some characteristic value.

- A key feature to fBm is that if you zoom in on any part of the function you will produce a similar random walk in the zoomed in part.

- A statistically self-similar fractal.

A 2D Terrain



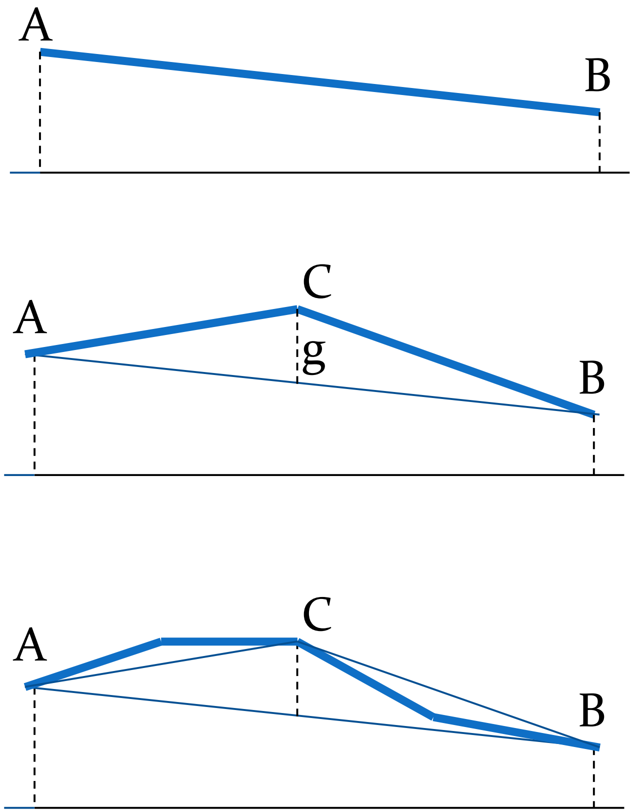
The above terrain is generated from a straight line (1D) segment, repeatedly subdividing each segment to create smaller segments

Midpoint Displacement Algorithm (2D)

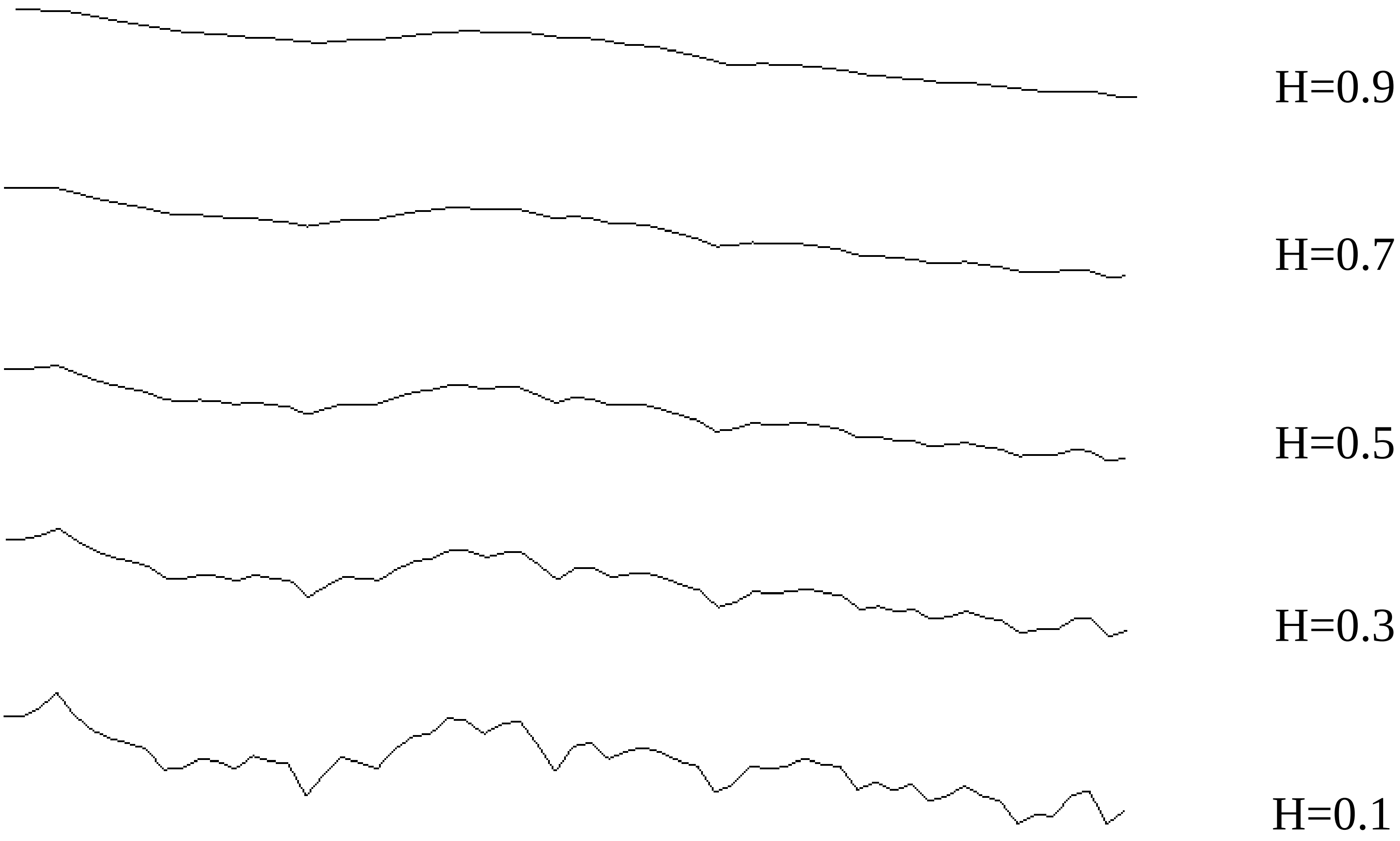
- Subdivide a line segment AB into two equal parts, and displace the midpoint C by an amount “g”, where “g” is Gaussian random value with zero mean (allowing negative values) and standard deviation *s*.

  $y_C = (y_A + y_B)/2 + g$

- Subdivide each segment AC and CB by displacing their midpoints by an amount g with standard deviation $s(1/2)^H$. (H is a constant between 0 and 1).
- In general, at each new level of subdivision, introduce a scale factor $(1/2)^H$. The “roughness” of the terrain depends on the parameter H (called the Hurst parameter)



Varying H



Brownian Motion

- A curve with parameter *H* in the range $0 < H < 1$ is called *fractional Brownian motion* (fBm).
- Brownian motion in one dimension constitutes the simplest random fractal, and also it is at the heart of many generalizations in fractal modeling.
- The parameter H describes the roughness of the function. The roughness increases as H decreases.
- A Brownian motion can also be characterized by its spectral density function S(*f*), where

$S(f) = \frac{1}{f^{\beta}}$ and, $\beta = 2H+1$

fBm: Fractal Dimension

- The fractal dimension D of the above curves (generated using a one dimensional random function X(t)) is given by the formula

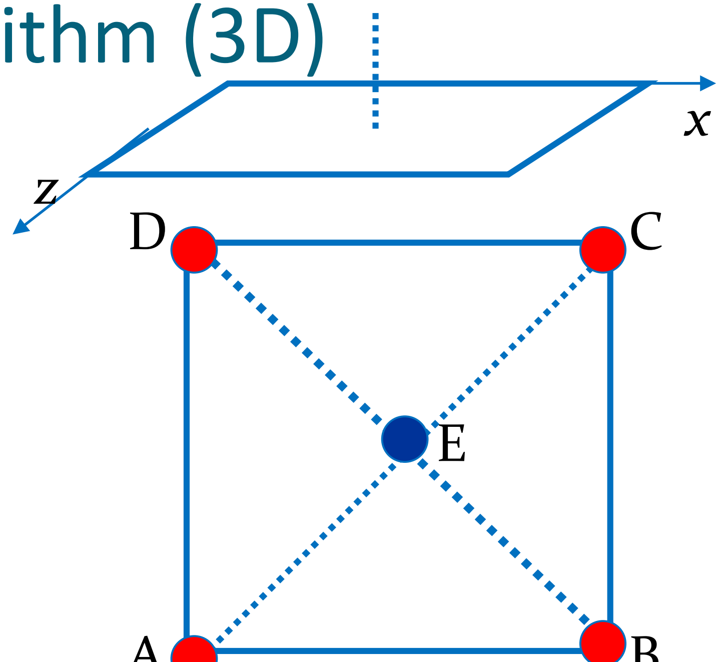
$$D = 2 - H.$$

- If we use a two parameter random function X(u,v) to generate 3D terrains, the fractal dimension D is given by

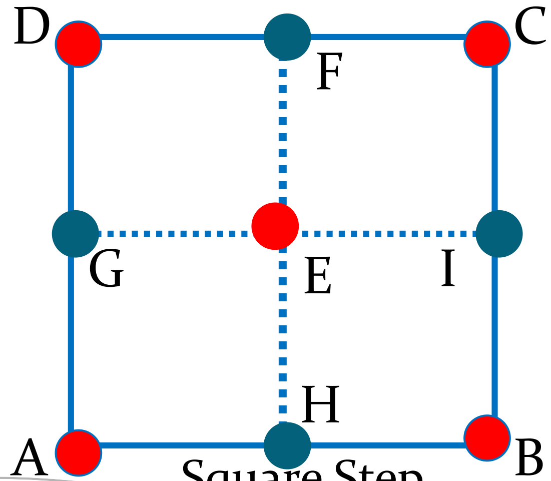
$$D = 3 - H.$$

Diamond-Square Algorithm (3D)

- Initialization: Assign height values to the 4 vertices A, B, C, D of a square grid.
- Diamond Step: Displace the midpoint E by a Gaussian random variable “g” with mean 0, and standard deivation *s*.
  $y_E = (y_A + y_B + y_C + y_D)/4 + g$
- Square Step: Displace the midpoints of each “diamond” created in the previous step (F, G, H, I), using a Gaussian variable “g” with the *same* standard deviation *s*.
- Displace the midpoint of each “square” created in the previous step, after scaling the SD of “g” by a factor $(1/2)^H$.
- Repeat the square step with the *same* SD.



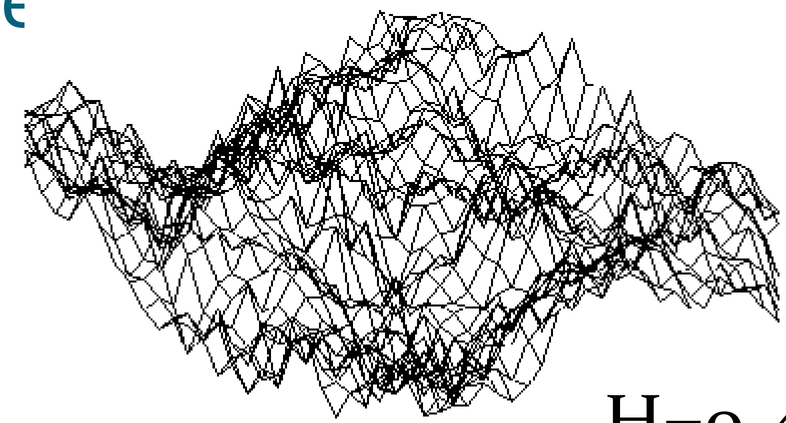
Diamond Step



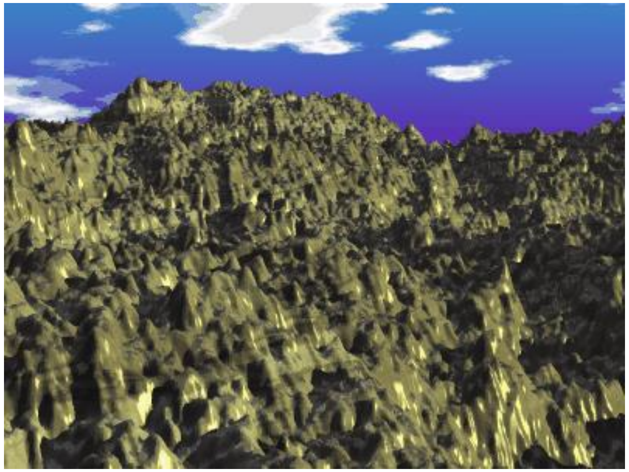
Square Step

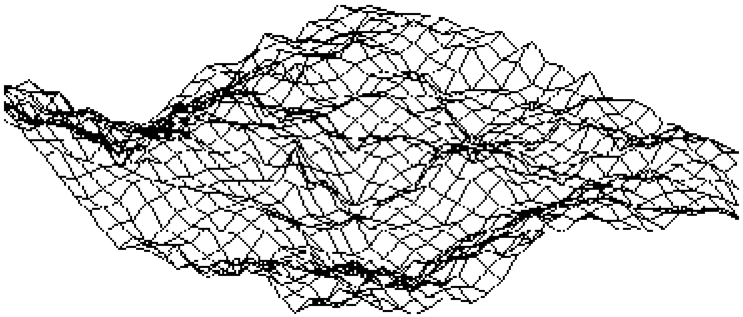
Diamond-Square

The above two steps are repeated for each of the smaller squares, after scaling the standard deviation of g by $(1/2)^H$.

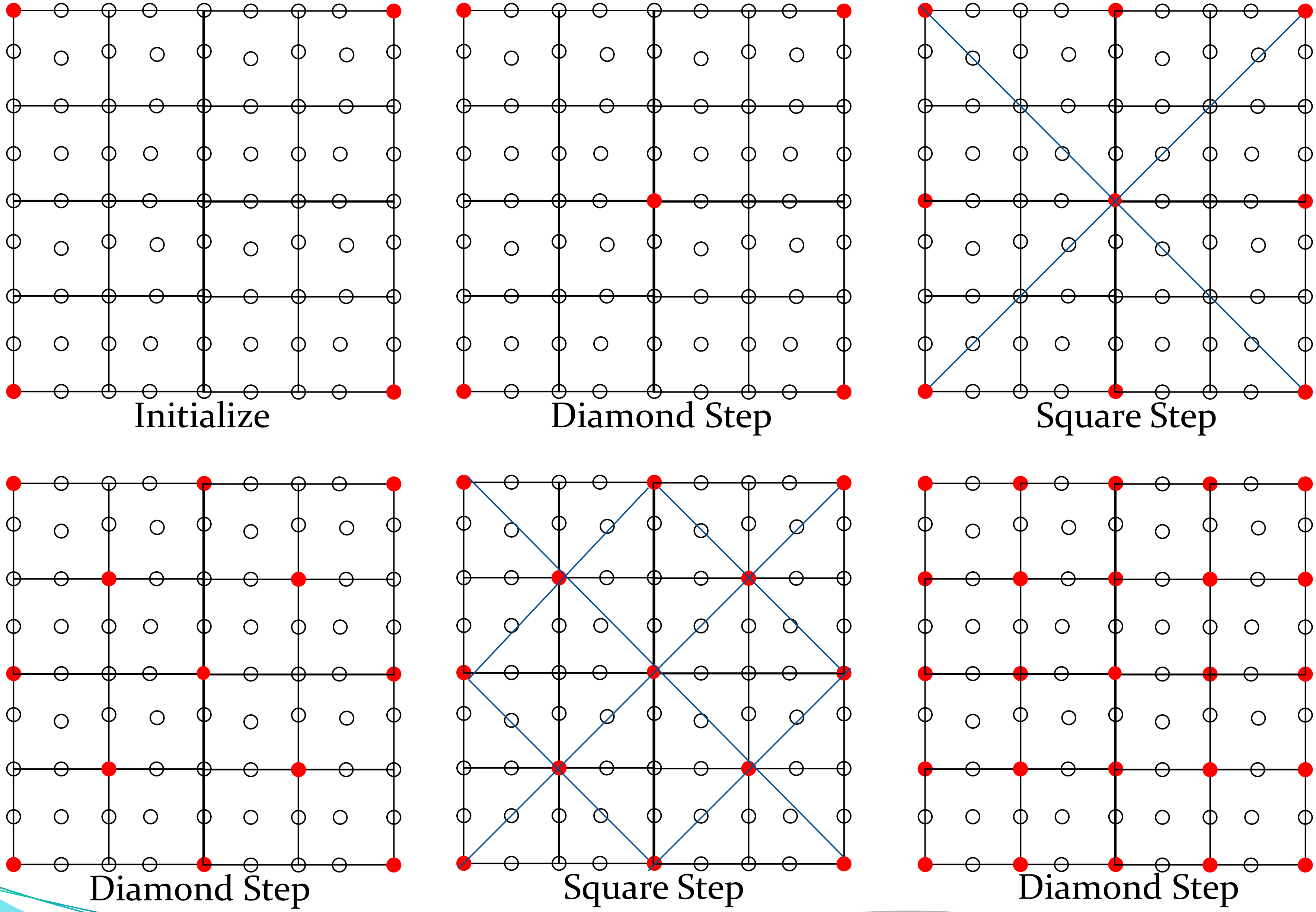


H=0.4





H=0.8



Initialize

Diamond Step

Square Step

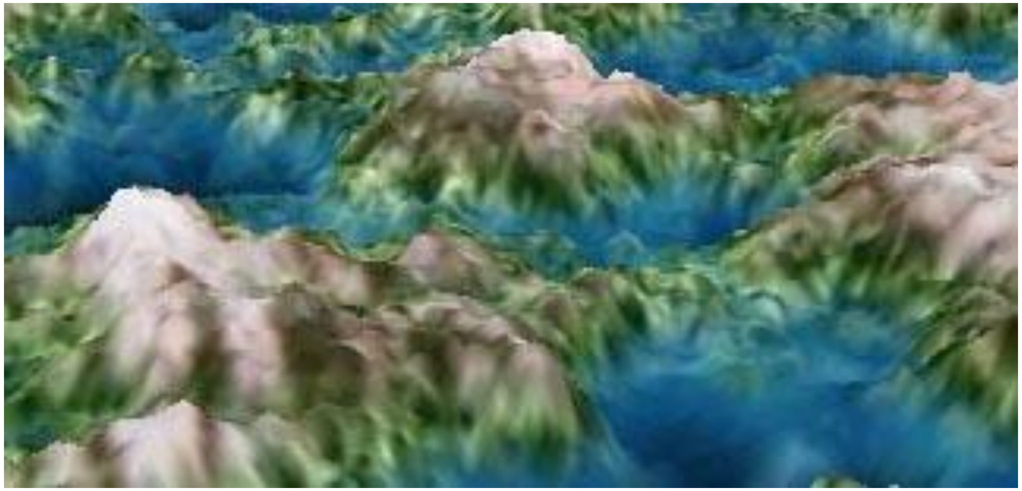
Diamond Step

Square Step

Diamond Step

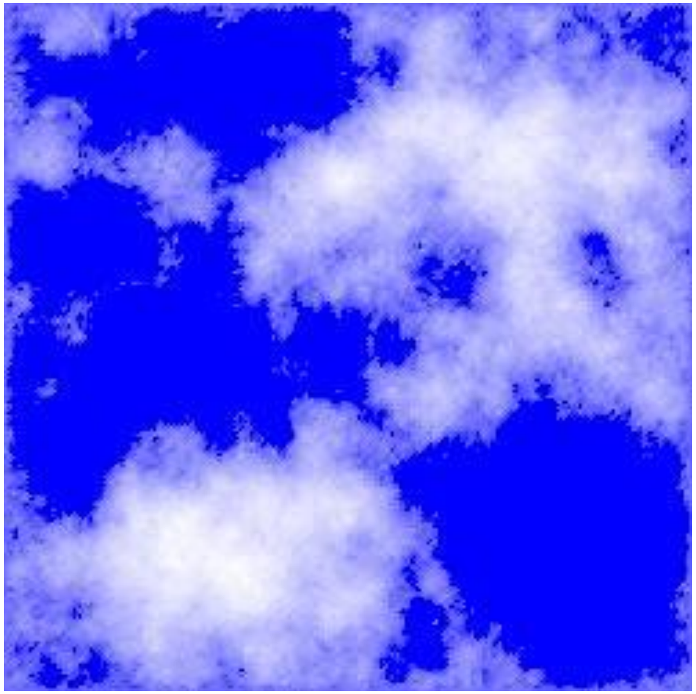
Diamond-Square Algorithm

- The diamond-square algorithm can be easily implemented as a recursive procedure to generate a terrain model from a uniformly subdivided grid of size 2^n+1 x 2^n+1.
- Select H in the range 0.5 – 0.8
- Use height based texturing.



Height Maps

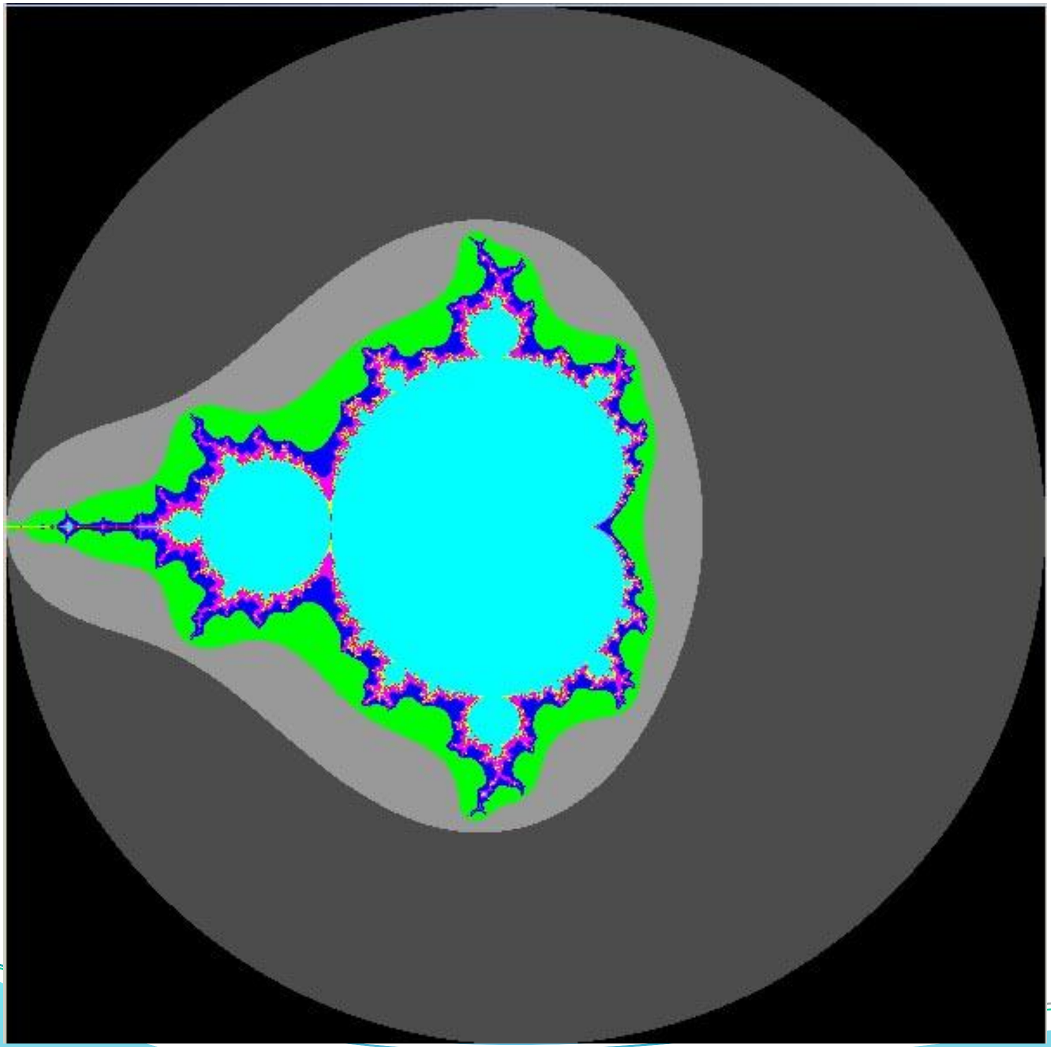
The 2D height map obtained using the diamond-square algorithm can be used to generate fractal clouds. Use the y value to generate opacity.

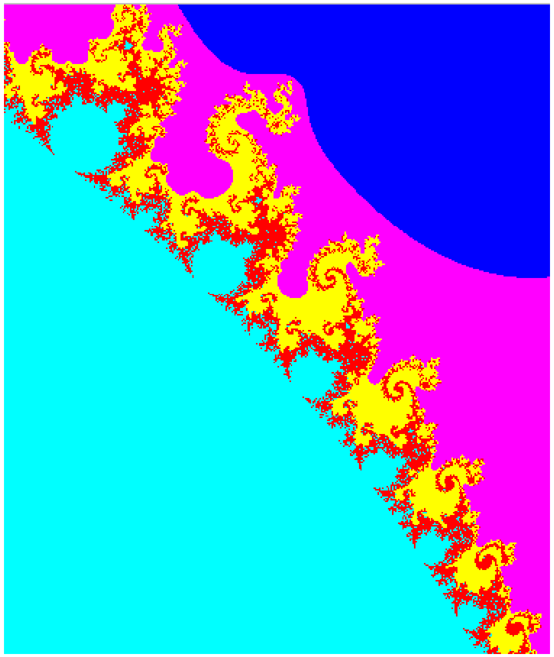


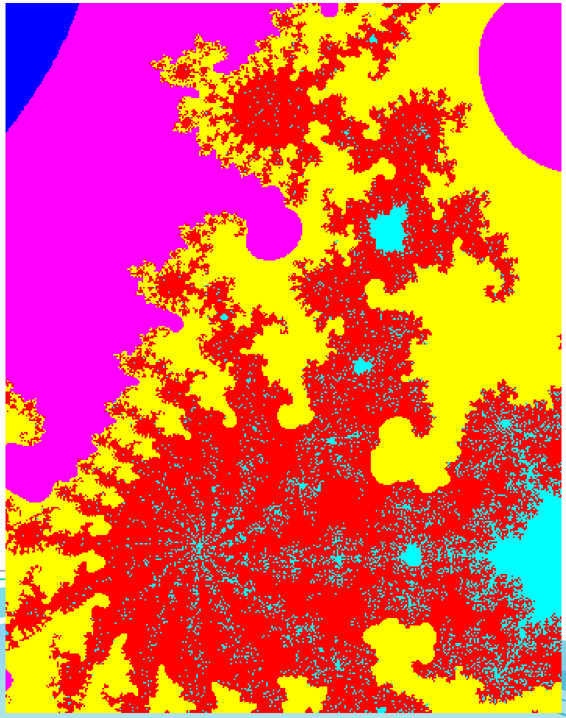
The Mandelbrot Set

- The border of a Mandelbrot set is an astoundingly complicated fractal.
- A Mandelbrot set's complexity can be explored by zooming in on a portion of the border. In theory, the zooming can be repeated forever – the border is "infinitely complex"
- Much of the fascination of the Mandelbrot set stems from the fact that an extremely simple formula gives rise to an object of such great complexity.

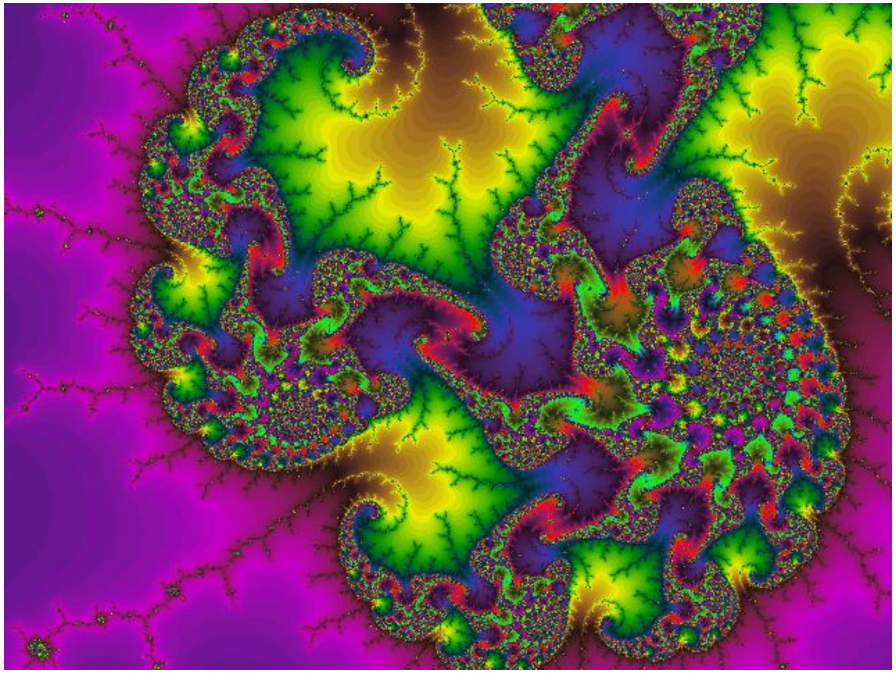
The Mandelbrot Bug

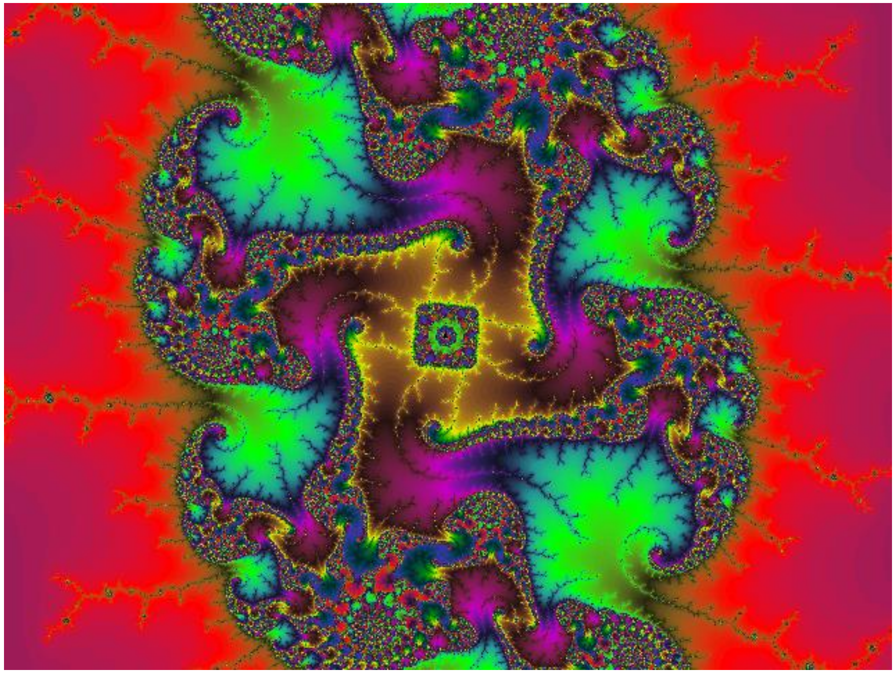




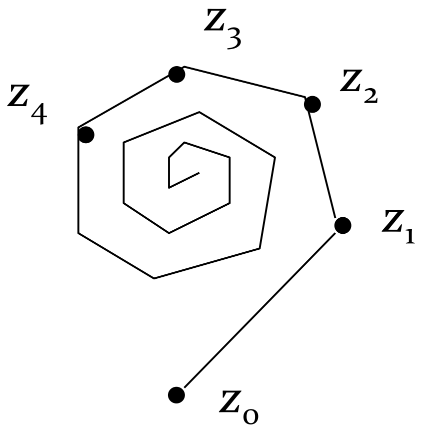


Mandelbrot Set at Higher Magnifications





Non-linear Dynamics



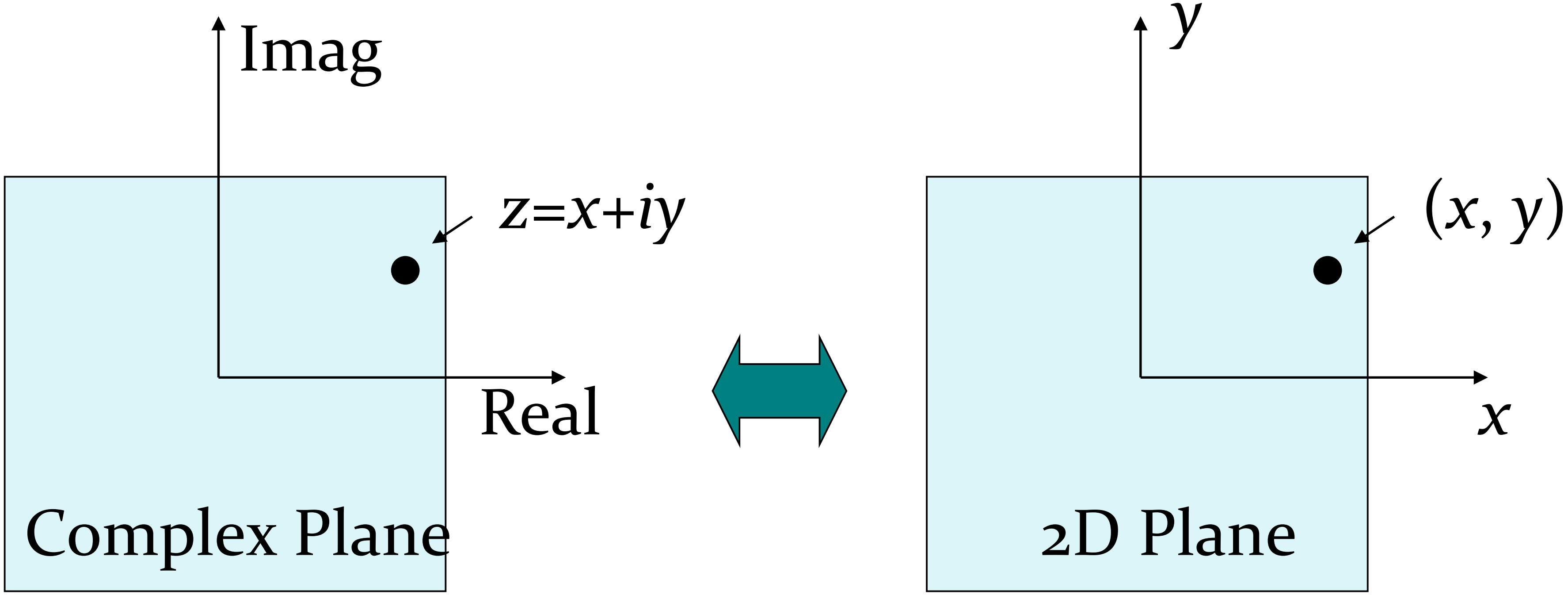
- Both Mandelbrot and Julia sets are generated by a non-linear equation of the form:

  $z = z_0$ (Initial condition)

  $z \leftarrow z^2 + c$ (Update equation)

- The sequence of *z* obtained using z_0 as the initial condition is called the orbit of z_0. The orbit obviously depends on the value of *c* also.
- Mandelbrot set is generated using orbits of 0 (i.e., $z_0 = 0$), for different values of *c*.
- A Julia set is obtained by considering different orbits for a fixed value of *c*.

2D-Points and Complex Numbers



Imag

$z = x + iy$

Real

Complex Plane

y

(x, y)

x

2D Plane

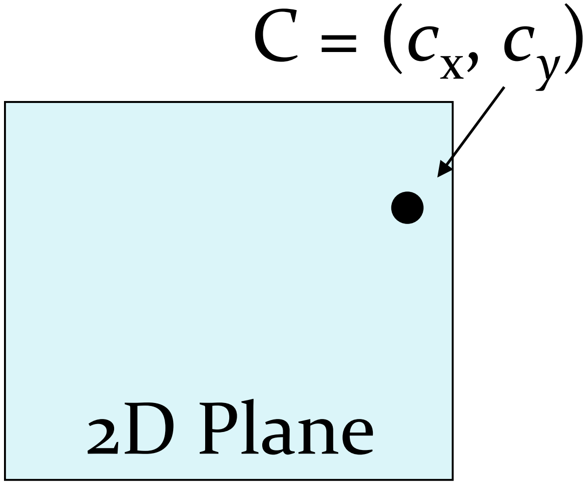
$$(x_1 + i\,y_1) + (x_2 + i\,y_2) \;=\; (x_1 + x_2) + \; i\,(y_1 + y_2)$$

$$(x_1 + i\,y_1)\,(x_2 + i\,y_2) \;=\; (x_1x_2 - y_1y_2) + i\,(x_2y_1 + x_1y_2)$$

If $z = (x + i\,y)$, then $z^2 = (x^2 - y^2) + i\,2xy$
and, $|z|^2 = x^2 + y^2$. $\quad (i = sqrt(-1))$.

The Mandelbrot Set

- Consider each point on the complex plane as representing the value c in the non-linear sequence of complex numbers $z \leftarrow z^2 + c$
- Thus each point c on the 2D plane produces an orbit of 0.
- $x \leftarrow x^2 - y^2 + c_x$
- $y \leftarrow 2xy + c_y$



- Assign a color value to this point depending on how fast or slow the orbit diverges.

The Orbit of 0

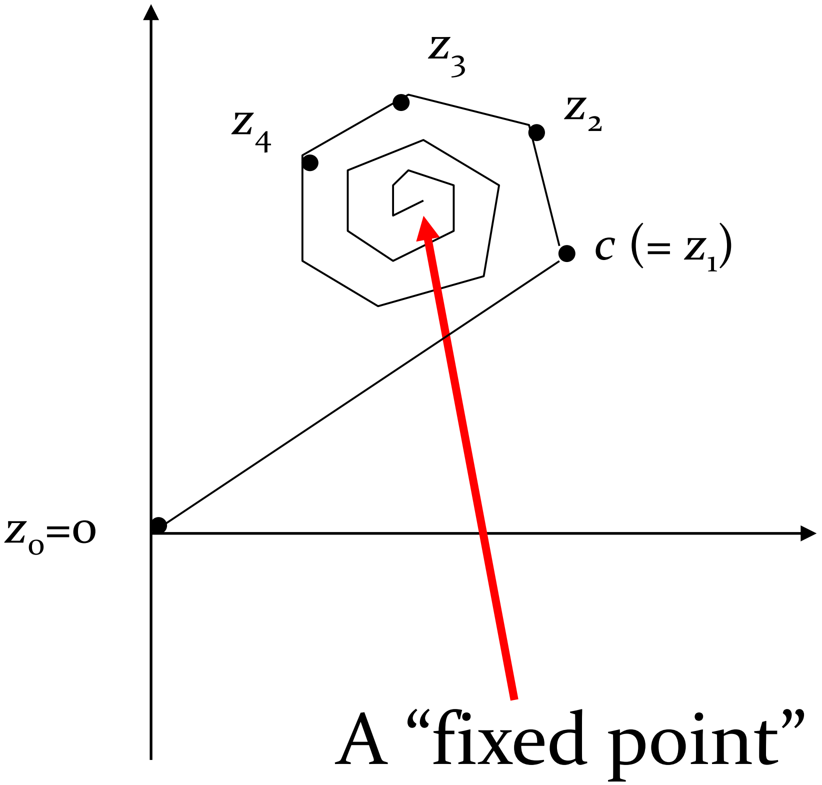
$z_0 = 0$

$z_1 = z^2 + c = c$

$z_2 = z^2 + c = c^2 + c$

$z_3 = z^2 + c = (c^2 + c)^2 + c$

$z_4 = z^2 + c = ((c^2 + c)^2 + c)^2 + c$



A “fixed point”

Orbits of 0

$C = (0.2, 0.05)$

z_1 →	0.2	0.05
z_2 →	0.2375	0.07
	0.251506	0.08325
⋮	0.256325	0.091876
	0.257261	0.0971
	0.256755	0.09996
	0.255931	0.101331
	0.255233	0.101867
	0.254767	0.102
	0.254502	0.101972
	0.254373	0.101904
	0.254321	0.101843
	0.254307	0.101802

↓

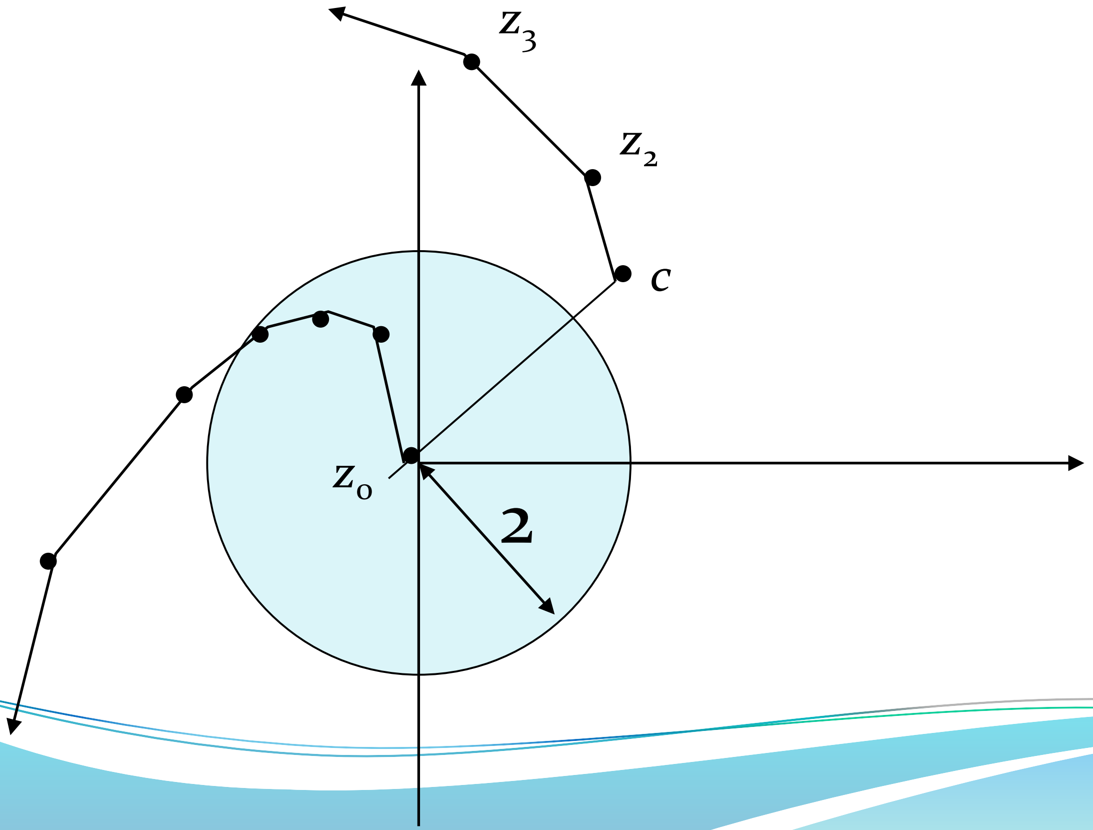
Fixed Point

$C = (0.4, 0.1)$

0.4	0.1
0.55	0.18
0.6701	0.298
0.76023	0.49938
0.72857	0.859287
0.19244	1.352101
-1.39114	0.620397
1.950385	-1.62612
1.559735	-6.24312
-36.1438	-19.3752
931.3766	1400.691
-1094472	2609141
-5.6E+12	-5.7E+12

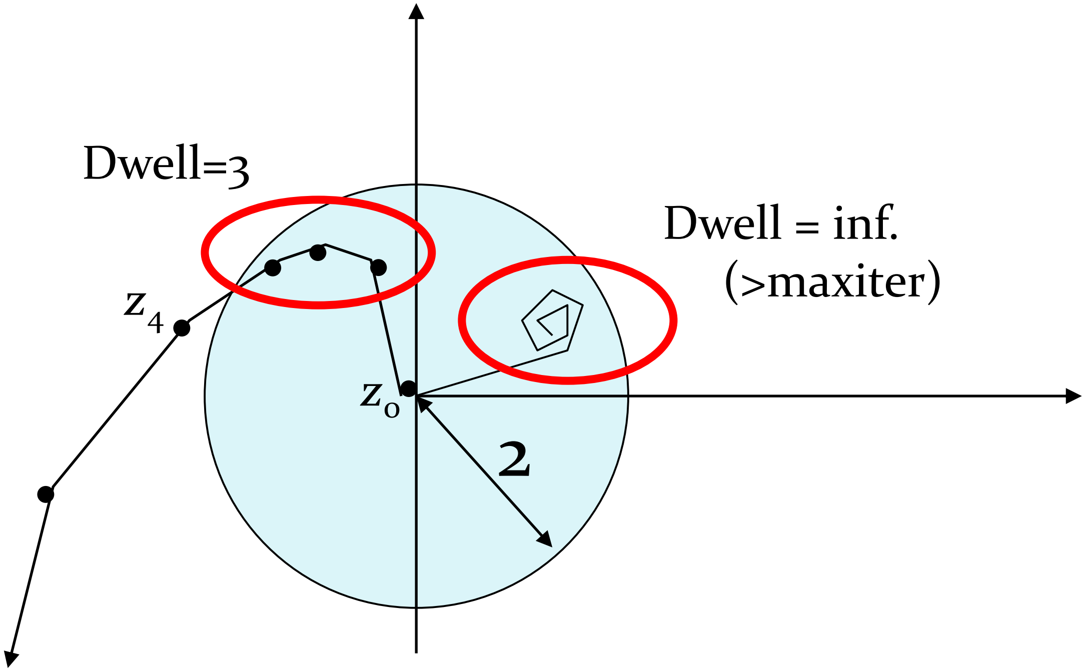
Properties of Orbits

- If the complex number *c* has a magnitude greater than 2, then it generates a divergent orbit.
- If the sequence attains a value whose magnitude is greater than 2.0, then the orbit diverges.



The Dwell of an Orbit

- The number of iterations a sequence takes to exceed the magnitude 2, is called the **dwell** of the orbit.
- For convergent orbits, the dwell is infinite.
- For each point *c*, give a color value depending on the dwell of the orbit it generates.



Mandelbrot Set - Code

```
for(cx=xmin; cx<xmax; cx+=step){
  for(cy=ymin; cy<ymax; cy+=step){
       x=0; y=0;
       for(iter=0; iter<maxiter; iter++){
            xnew = x*x- y*y + cx;
            ynew = 2*x*y + cy;
            a = xnew*xnew + ynew*ynew;
            if((a>4.0)||(iter==maxiter-1)){
                 setColor(iter);
                 glVertex2d(cx,cy);
                 break;
            }
            x=xnew; y=ynew;
       }
  }
}
```

Mandelbrot Set - Code

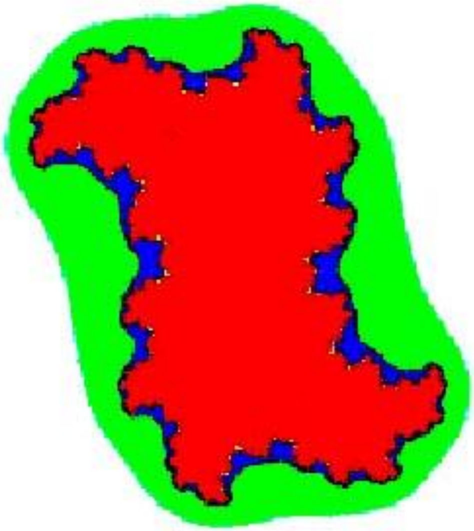
```
void setColor(int iter){
       if(iter<2)    glColor3f(0.0,0.0,0.0);
  else if(iter<4)    glColor3f(0.3,0.3,0.3);
   else if(iter<6)    glColor3f(0.6,0.6,0.6);
  else if(iter<10)   glColor3f(0.0,1.0,0.0);
  else if(iter<16)   glColor3f(0.0,0.0,1.0);
  else if(iter<32)   glColor3f(1.0,0.0,1.0);
  else if(iter<64)   glColor3f(1.0,1.0,0.0);
  else if(iter<128)  glColor3f(1.0,0.0,0.0);
  else               glColor3f(0.0,1.0,1.0);
}
```

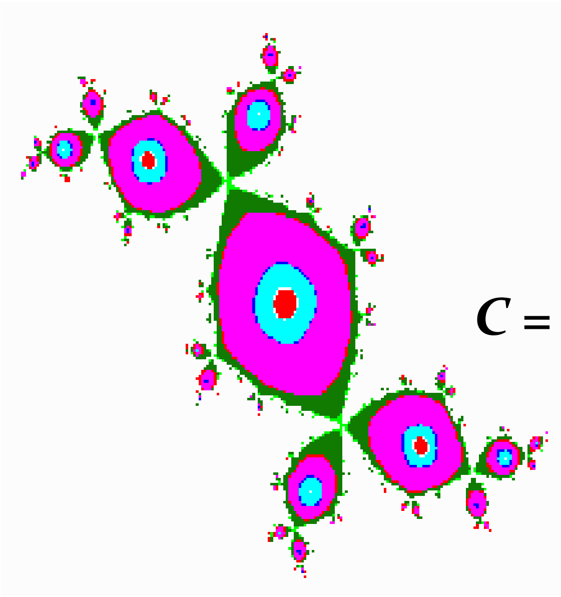
The Julia Set

- A Julia Set is also computed using the same non-linear mapping $z \leftarrow z^2 + c$, but we keep c constant and examine what happens for different starting points Z_0.
- The Julia Set thus depicts the dwell of orbits of different points for a fixed value of c.
- In a Julia Set, each point represents the starting value z_0 of an orbit.
- Color values are assigned as in the previous case (Mandelbrot set) based on the dwell of each orbit.

Julia Set: Examples

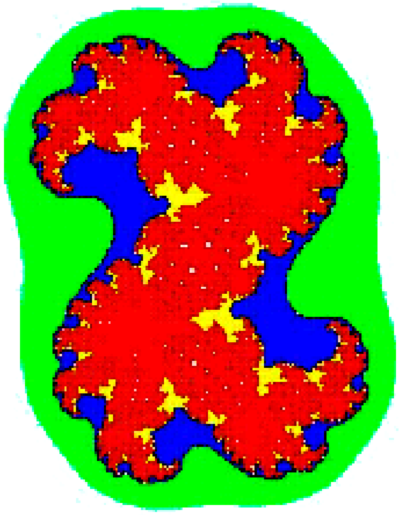
$C = 0.1 + i\,0.5$

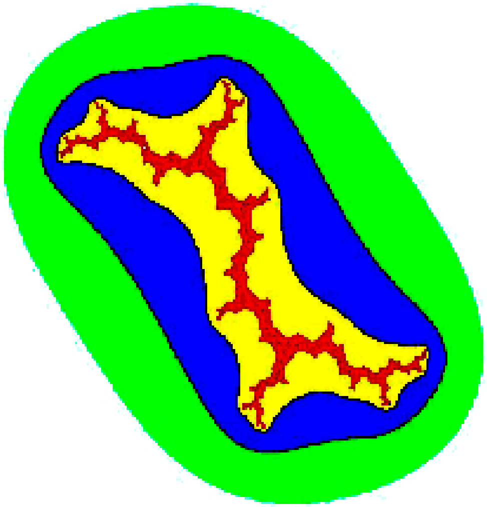




$C = -0.125 + i\,0.75$

$C = 0.38 + i\,0.25$





$C = i$

Julia Set - Code

```
for(x0=xmin; x0<xmax; x0+=step){
  for(y0=ymin; y0<ymax; y0+=step){
     x=x0; y=y0;
     for(iter=0; iter<maxiter; iter++){
          xnew = x*x- y*y + cx;
          ynew = 2*x*y + cy;
          a = xnew*xnew + ynew*ynew;
          if((a>4.0)||(iter==maxiter-1)){
               setColor(iter);
               glVertex2d(i,j);
               break;
          }
          x=xnew; y=ynew;
     }
  }
}
```