

COMP3201 – Computer Graphics

Module 2: Transformations and Scene Creation

2.9 Fractals

2.9.1 Introduction

Although we can construct many geometric shapes using primitives (lines, triangles, polygons, etc.), this approach is not effective when we need to represent forms and shapes that occur in nature (for example, terrain, mountains, clouds, plants). However, the use of fractals in Computer Graphics allows us to create more realistic natural forms effectively.

One of the basic properties of fractal images is that of self-similarity, and it is this property that allows the efficient construction of such images using recursion. The images may be generated from a simple rule, but the resulting images can appear quite complex.

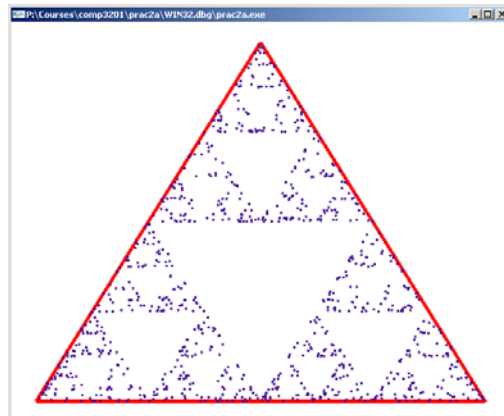
The word “fractal” was first used by Mandelbrot in 1975. A fractal is self-similar (for example, the Sierpinski triangle) and is independent of any scaling – that is, no matter how close we zoom in on such an image, the part we are looking at is similar to all others. Fractals are formed using an algorithm that is usually recursive, so it is well-suited to computer implementation.

Whereas Euclidean objects have a dimension that we understand easily (e.g. the triangle of dimension 2, and the tetrahedron of dimension 3), fractals have their own dimension that is fractional and represents the complexity of the fractal curve.

Some simple examples of fractal images include

- Sierpinski triangle (or gasket)
- Koch snowflake curve
- Julia set
- Mandelbrot set

The concept of self-similarity is readily observed in the Sierpinski triangle.

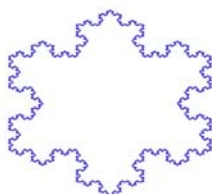


This triangle can be decomposed into 3 congruent triangles, where the smaller triangles are half the width and half the height of the original triangle. In other words, this triangle consists of three self-similar copies of itself, each of which has a magnification factor of 2. Looking more closely at the triangle, we can also see 9 self-similar copies which have a magnification factor of 4, and so on. In general, it is made up of 3^N self-similar copies, each with a magnification factor of 2^N .

The Koch snowflake curve is formed by dividing each line segment into thirds and replacing the middle third with two extra segments to form an equilateral triangle:



Applying this rule to an initial line segment yields the Koch curve; the snowflake is formed by applying this rule to each side of an equilateral triangle:



Each sub-segment of the curve is similar to every other. During generation of this curve, each line segment increases its length by a factor of $4/3$. To see what happens to the perimeter of the entire snowflake, let's assume that the length of the initial segment is 1 unit (so that the perimeter of the initial triangle is 3 units). Then, at the first subdivision, the length of each side of the triangle becomes $3 \times \frac{1}{3} + \frac{1}{3}$ (so the new perimeter is $3 \times \frac{4}{3}$). At the second subdivision, each of the 4 sub-segments becomes $3 \times \frac{1}{9} + \frac{1}{9}$, so that the perimeter is now $3 \times 4 \times \frac{4}{9}$ or $3 \times \left(\frac{4}{3}\right)^2$. In general, after n subdivisions, the perimeter will be $3 \times \left(\frac{4}{3}\right)^n$. The area of the snowflake also increases with each subdivision, but at a much slower rate. Consequently, the Koch snowflake has the interesting property of having finite area but with a perimeter that tends to infinity, as $n \rightarrow \infty$.

Consider the process

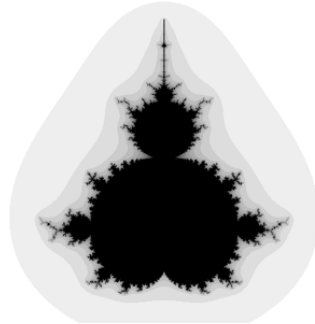
$$x_{n+1} = f(x_n) = x_n^2 + c$$

for complex numbers x_n . So, given x_0 , the series is constructed by computing

$$x_1 = x_0^2 + c, \quad x_2 = x_1^2 + c, \quad \dots$$

A Julia set, for a given complex number c , consists of the set of all initial values x_0 such that the iterative scheme $x_{n+1} = x_n^2 + c$ does not diverge.

Closely related to this is the Mandelbrot set. Without loss of generality, we can take $x_0 = 0$. Then the Mandelbrot set is the set of complex numbers c such that the iterative process $x_{n+1} = x_n^2 + c$ does not diverge. To represent this visually, the complex points c are coloured according to the number of iterations required for the iterative scheme to reach the value X (some large value). Another interesting visual representation occurs when analysing which complex c will give rise to periodic solutions (that is, when there exists N such that $x_N = 0 = x_0$). The following figure (Bourke (1991)) shows a graphical representation of a Mandelbrot set.



The fractal dimension of a fractal image is a measure of its complexity, and is attributed to Hausdorff. Intuitively a square has dimension 2 (it is in a plane), while a cube has dimension 3. This figure can also be derived by noting that a square, for example, can be broken into N^2 self-similar squares, each of which has a magnification factor of N . A cube can be broken into N^3 self-similar cubes, each with a magnification factor N . The following rule (where \ln is the natural logarithm) gives the dimension of the object:

$$\begin{aligned} \dim \text{ square} &= \frac{\ln(\# \text{ selfsimilar objects})}{\ln(\text{magnification factor})} \\ &= \frac{\ln(N^2)}{\ln N} = 2 \frac{\ln N}{\ln N} = 2 \end{aligned}$$

$$\dim \text{ cube} = \frac{\ln(N^3)}{\ln N} = 3.$$

This same rule can be used to define the fractal dimension of a self-similar image:

$$\text{fractal dim} = \frac{\ln(\# \text{ selfsimilar objects})}{\ln(\text{magnification factor})}.$$

So the fractal dimension of the Sierpinski triangle is

$$\frac{\ln 3^N}{\ln 2^N} = \frac{\ln 3}{\ln 2} \approx 1.58.$$

The fractal dimension of the Koch snowflake curve is

$$\frac{\ln 4}{\ln 3} \approx \frac{1.3863}{1.0986} \approx 1.58.$$

2.9.2 Fractals and Recursion in Computer Graphics

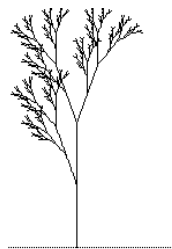
Fractal algorithms are often used for generating realistic landscapes; random deviations are added to the surface on a decreasing scale in the same way that a fractal curve increases in complexity during its construction. The fractal dimension of the algorithm corresponds to how irregular the surface may appear, and for landscapes a fractal dimension of about 2.2 is often used (Barnsley et al (1988)).

Fractals have also been used successfully in the generation of clouds. In this case, for realism, a larger fractal dimension of approximately 3.3 has been used (Barnsley).

Coastlines, on the other hand, need finer detail and hence a fractal algorithm of dimension approximately 1.2 would be appropriate (Barnsley et al.).

Mountains can also be made to look more realistic by incorporating fractal distortions. Because the model can be generated from a rule rather than storing large quantities of vertex and height data, the construction of such a model is computationally efficient.

Another technique used in Computer Graphics is that of plant growth. For example, L-systems (Lindenmayer) are defined by specifying a set of rewriting rules that are then applied to “grow” a plant. L-systems are particularly useful for creating realistic plants as they can handle the branching structure of plants. The following figure shows a plant grown by an L-system (Bourke (1991)).



A simple example of an L-system rewriting rule is the string

$$F \rightarrow F - F ++F - F$$

where F means to move forward a distance of 1, and $+$ (or $-$) indicates a right (or left) turn through some angle (e.g. 60 degrees). This particular rewriting rule will actually generate the Koch curve described in the previous subsection.

L-system images are made up of many lines. An alternative approach is that of IFS (or Iterated Function Systems), based on polygons. Each polygon undergoes a transformation (scaling, rotation, and/or translation) that results in a new polygon.

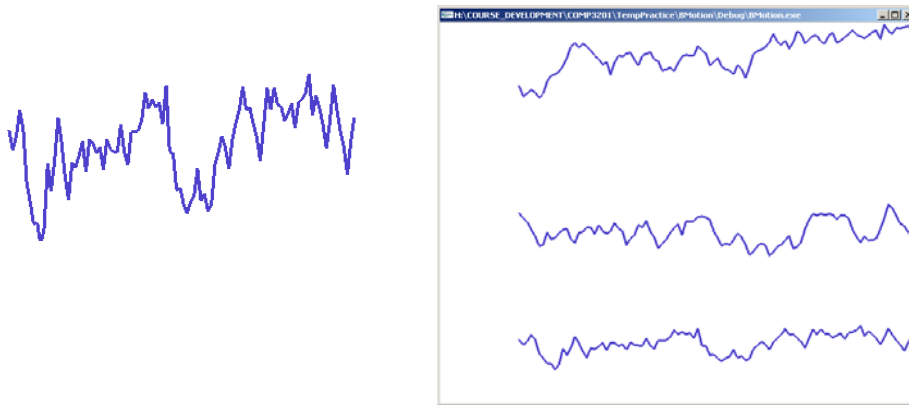
In concluding this subsection, note that generating the data that defines a fractal image is only one small part of creating a realistic scene. It will be necessary to apply appropriate colours and shading to complete the effect.

2.9.3 Fractal Algorithms

There are two main types of fractal image – deterministic and random. Deterministic fractals are those that are made up of scaled-down self-similar copies (e.g. Sierpinski, Koch). Random fractals are ones that use some element of randomness in their generation, and these are the ones that are used to simulate landscapes, coastlines, etc.

2.9.3.1 Brownian Motion

Brownian Motion (a term used to describe the random behaviour of particles in a liquid) is fundamental to random fractals. The simulation of Brownian Motion (using Gaussian Normal random variables) can be used to approximate curves and surfaces. The following figure shows a number of simulations of Brownian motion.



Note that by specifying a seed for the random number generator at the start of the algorithm, the curves above can be re-generated again.

Brownian Motion is a process $W(t)$ that has zero mean and normally distributed increments. For a series of time steps $0 < t_0 < t_1 < \dots < t_n$, the increments $W(t_1) - W(t_0), W(t_2) - W(t_1), \dots$ are independent. Also, the variance satisfies

$$\text{Var}(W(t) - W(s)) = |t - s|.$$

In other words, for some t and $h > 0$, $W(t+h) - W(t)$ is normally distributed with mean 0 and variance h .

To construct such a process (which we can then use when generating mountains and terrain, for example) we need to be able to generate (Gaussian) $N(0, \sigma^2)$ random variables (that is, random variables that are normally distributed with mean 0 and variance σ^2). However, if the (pseudo) random number generator available produces random variables that are uniformly distributed (for example, **rand()**), then some extra calculations are required. There are several different algorithms for generating normal random variables from uniform random variables (for example, the Box-

Muller and Polar-Marsaglia methods); however, for our requirements here, we will use a simple algorithm that will approximate normal random variables. This algorithm just sums 12 uniform random numbers (in the interval [0, 1]) (which then have a mean value of 6 and a variance of σ^2) and adjusts the result to have a mean of 0 and the appropriate scaled variance:

```
rn = 0.0;
for (i = 0; i<12; i++) {
    rn += rand() / ( (float) RAND_MAX );
}
rn = (rn - 6) *  $\sigma$  ;
```

The variable `rn` is then an approximate normal random variable.

2.9.3.2 Fractional Brownian Motion

The curves in the preceding subsection have been generated using one-dimensional Brownian motion, but by introducing fractional Brownian motion, the roughness of the curve can be adapted depending on what is being modelled.

Recall that the Brownian motion process has independent normally distributed increments, and that the variance satisfies

$$\text{Var}(W(t) - W(s)) = |t - s|.$$

If this property is generalised so that

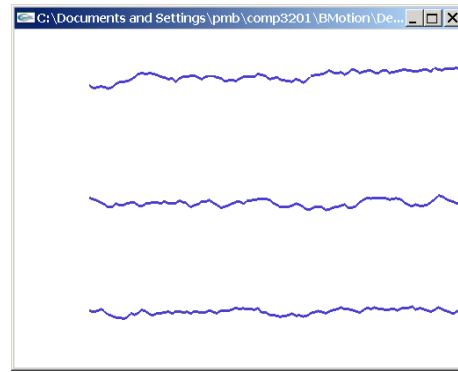
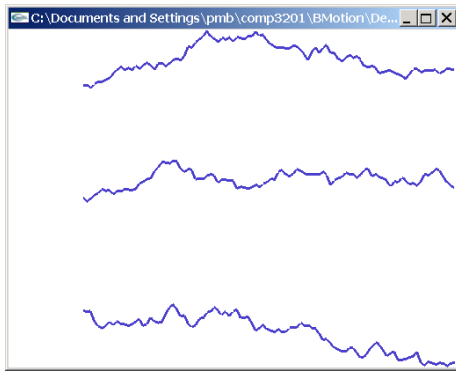
$$\text{Var}(W(t) - W(s)) = |t - s|^{2k},$$

where $0 < k < 1$, the process is called Fractional Brownian Motion (fBM). Simple Brownian motion corresponds to $k = \frac{1}{2}$. Smaller values of k will yield Brownian motion curves that oscillate more erratically, while $\frac{1}{2} < k < 1$ results in smoother curves. It turns out that the fractional Brownian motion curves have a fractal dimension of $2-k$, and so the roughness or smoothness of the curve corresponds directly to the factor k .

To generate fBM curves, the random samples are obtained as before except that `rn` is scaled by

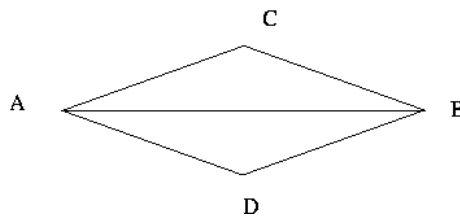
$$\left(\frac{1}{2}\right)^k \sqrt{1 - 2^{2k-2}} \sigma.$$

The following figures show three sample paths of fractional Brownian motion for $k = 0.25$ (left) and $k = 0.75$ (right).



2.9.3.3 Midpoint Displacement

Midpoint Displacement is another way of approximating Brownian motion. The algorithm discussed here is one that adds distortion to a line segment, by using random midpoint displacement. Basically the midpoint of the line segment AB is perturbed by an amount d_0 which is sampled from the Normal distribution with mean 0 and variance $\frac{\sigma^2}{2}$. If $d_0 > 0$, the perturbed point is at C, otherwise it is at D.



This algorithm is executed recursively on each successive sub-segment (e.g. on segments AC and CB , using d_1 , and then on ensuing sub-segments using d_2, d_3 etc.).

It can be shown that the displacements d_i should be sampled as follows:

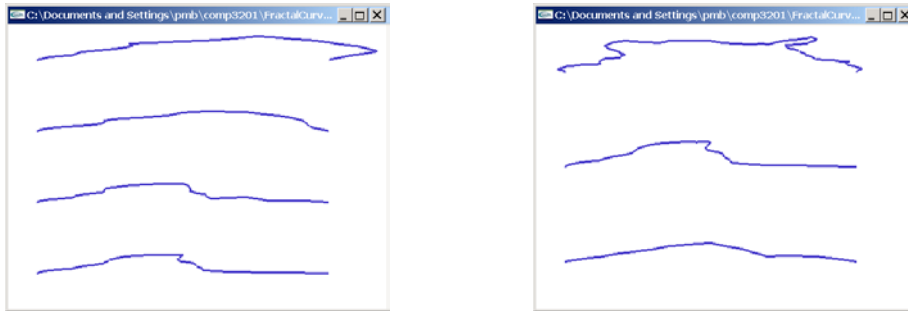
$$\begin{aligned} d_0 &\approx N(0, \frac{1}{2} \sigma^2) \\ d_1 &\approx N(0, \frac{1}{4} \sigma^2) \\ d_2 &\approx N(0, \frac{1}{8} \sigma^2) \\ d_n &\approx N(0, \frac{1}{2^{n+1}} \sigma^2). \end{aligned}$$

This means that the $N(0, 1)$ random variable should be scaled by $\Delta_n = \sigma \times (\frac{1}{2})^{n+1/2}$ for the n^{th} level of recursion.

Such a recursion will continue until a stopping criterion is satisfied. For example, it could continue until the new segment length is smaller than epsilon (a preset small number), or it may continue until a certain number of levels of recursion have been applied.

The figure on the left shows modified line segments with epsilon set to (from top to bottom) 1.0, 0.75, 0.2 and 0.01, while the figure on the right shows the effect of

varying the fractal dimension of the curve from 1.25 (top), to 1.0 (middle) and to 0.75 (bottom) while keeping epsilon set to 0.01.



The following pseudo-code indicates how this procedure could be implemented recursively to generate random distortion on a line segment (for example, a coastline); the line segment from A to B will be distorted to traverse ACB by calculating the point C on a line perpendicular to AB passing through the midpoint of AB .

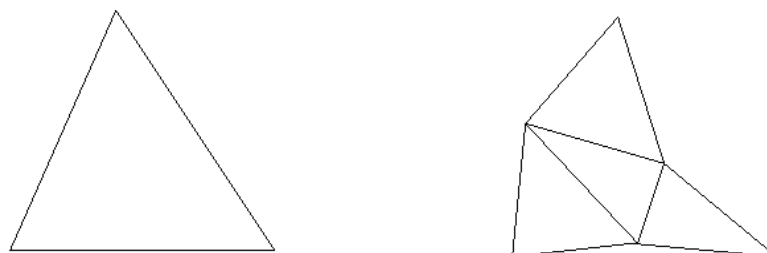
```

FractalCurve(A, B, sigma)
  Compute xIncr = A.x - B.x
  Compute yIncr = A.y - B.y
  If (xIncr * xIncr + yIncr * yIncr < epsilon)
    drawLine(A, B)
  Else {
    delta_n = sigma * pow(0.5, (n+1)/2)
    Generate rn * delta_n
    Compute C.x = 0.5*(A.x+B.x) - rn*(B.y-A.y)
    Compute C.y = 0.5*(A.y+B.y) + rn*(B.x-A.x)
    FractalCurve(A, C, delta_n)
    FractalCurve(C, B, delta_n)
  }

```

The midpoint displacement technique can also be used on 3D shapes, to generate mountains (for example). Suppose the mountain has been approximated by a tetrahedron. Then, in turn, each face of the tetrahedron can be modified by displacing the midpoint of each side of the triangle using the midpoint displacement algorithm. At each level of recursion, four new triangles are created. The roughness or smoothness of the distortion is again controlled by the variance of the random numbers.

The following figure shows how the front face of the tetrahedron may be distorted by perturbing the midpoints of the sides of the triangle to form four new triangles.



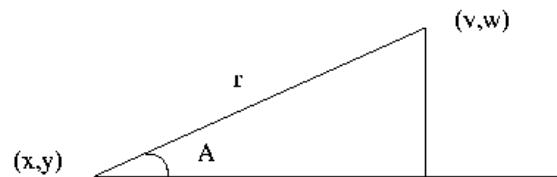
Note that it is important that the resulting objects are formed correctly so that inappropriate transformations (e.g. folding inwards) does not arise.

This algorithm can also be used to generate a fractal terrain, by starting with a flat mesh base of polygons (for example, rectangles). Then each rectangle can be modified by subdividing it into smaller rectangles with the modified vertices now representing a height.

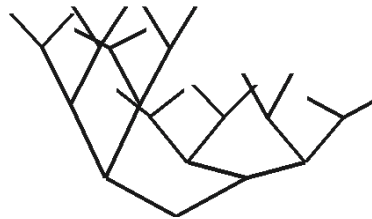
2.9.3.4 Fractal Plants

The final subsection of these notes on Fractals discusses the generation of fractal plants, using an L-system recursive algorithm. In 2D, (flat) plant generation is very straightforward. A rule can be constructed that defines when to draw a segment and when to turn a certain angle. To extend this to 3D, we just allow turns in more than one plane, and also we will need to draw a solid shape for each plant segment rather than just a line as in the 2D case.

Consider the 2D case first. We need to be able to draw the line (of length r) from the point (x,y) to (v,w) where we have turned through an angle A degrees:



From the figure, we can see that $v = x + r \cos A$, $w = y + r \sin A$. Whereas in the Koch snowflake construction, the angle A is fixed, we can introduce a random element to our plant construction to have the branches leaving the main stem at varying angles. For example, the plant could resemble the following:



Also, for realism, the length of the branches should be reduced, the further away the branch is from the base of the plant.

Pseudo-code to construct such a plant to recursion level n would resemble the following (where Push and Pop have been included to make use of matrix stacks for the transformations):

```
drawPlant2D(A, length, n)
    Calculate new coordinates newx, newy
    PushMatrix
        Draw line from (0,0) to (newx, newy)
    PopMatrix
```

```

PushMatrix
    Draw line from (0,0) to (-newx, newy)
PopMatrix

if (n != 0){
    PushMatrix
        Move to (newx, newy)
        drawPlant2D(rand-angle, 0.9*length, n-1)
    PopMatrix
    PushMatrix
        Move to (-newx, newy)
        drawPlant2d(rand-angle, 0.9*length, n-1)
    PopMatrix
}

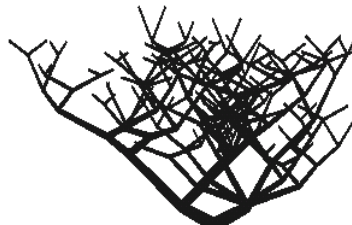
```

To extend this to the 3D case, we will introduce a rotation of B degrees around the y -axis in addition to the rotation of A degrees around z . The rotation matrix is

$$\begin{aligned}
 R &= R_z(A) R_y(B) \\
 &= \begin{pmatrix} \cos A & -\sin A & 0 & 0 \\ \sin A & \cos A & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos B & 0 & \sin B & 0 \\ 0 & 1 & 0 & 0 \\ -\sin B & 0 & \cos B & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
 &= \begin{pmatrix} \cos A \cos B & -\sin A & \cos A \sin B & 0 \\ \sin A \cos B & \cos A & \sin A \sin B & 0 \\ -\sin B & 0 & \cos B & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.
 \end{aligned}$$

So the (already-translated) point on the z -axis $(0, 0, r)^T$ is mapped to $(r \cos A \sin B, r \sin A \sin B, r \cos B)^T$.

The following figure gives an example of a 3D plant (for example, as required in Assignment 3):



References:

Barnsley, M.F., Devaney, R.L., Mandelbrot, B.B., Peitgen, H.-O., Saupe, D. and Voss, R.F. (1988): *The Science of Fractal Images*, Springer-Verlag, New York.

Bourke, P. (1991): <http://astronomy.swin.edu.au/~pbourke/fractals/fracintro/>