Lecture 9

It is often natural to model objects as consisting of parts, and the parts themselves as consisting of subparts, etc. Think of a bicycle. It has 2 wheels, a frame, a seat, handle bars, etc. Each wheel consists of a tire, spokes, hub, and rim. The frame consists of several cylindrical parts. The seat consists of a frame and cushion and mount, etc. Another example is a person's body. The body consists of a head, a torso, and limbs.

It is useful to think of these hierarchies as having a tree structure.

A person's body might be a root node. Its two children might be upper body and lower body. Each of these "parts" would consist of subparts and hence child nodes. We can imagine a set of draw commands which define a call tree correspdonding the the drawing of parts and subparts. In the lecture, I sketched out such a call tree for the parts of the human body.

When drawing a body or any hierarchical object in OpenGL, one also needs to keep in mind a different kind of tree that describes the coordinate systems of the various objects and parts. The root of the tree might be the coordinate system of some object (or perhaps the world coordinate system). Then any parent-child pair in the tree represents the coordinate systems of two objects and the edge represents a mapping from one coordinate system to the other.

To draw all the parts of all the objects in a scene, we essentially need to traverse this tree as well. It is common to to keep track of the `GL_MODELVIEW` matrix, by using a stack. OpenGL provides `glPushMatrix()` and `glPopMatrix()` commands for doing so. Before we descend from parent to child, we push the current `GL_MODELVIEW` matrix onto the stack. When we return to the parent, we pop the stack to recall the `GL_MODELVIEW` matrix at the parent. I sketched out how to do that in the lecture slides as well.

In the rest of this lecture, I'll briefly describe two notions of object hierarchies that have been very useful and commonly used in computer graphics, namely fractals and L-systems.

Fractals

We want to make pictures of many kinds of objects. Some objects such as spheres and cylinders have smooth surfaces. However other objects in the world do not have smooth surfaces, and indeed what makes many objects visually interesting and characteristic is that they are *not* smooth. For example, a pile of mud, a broccoli, a rocky mountain, a sponge, a cloud, a lung or kidney, etc are certainly not smooth objects. Such objects have been modelled as *fractals.*

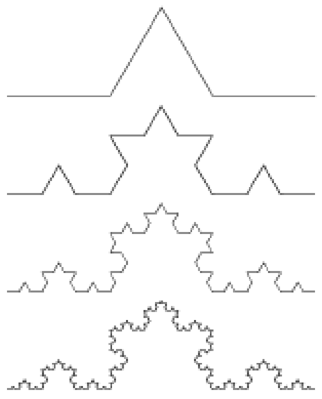
What are fractals? Fractals are rough (non-smooth) objects, so rough that it is impossible to define their length/ area/ volume in the usual way. A famous example is the coastline of some rocky island, such as England. How would you measure the length of the coastline? One idea is to use a measuring instrument and to count how many "steps" you need to take with this instrument to go once around the object. The length is step size multiplied by the number of steps. When we are dealing with a smooth curve, the length of the curve is the limit of step size times numsteps as stepsize goes to zero. You know this well from Calculus.

The problem with non-smooth objects is that the limit mentioned above doesn't exist. For example, take the case of a rocky coastline and start with 1 km steps. The total length of the coastline would be not be the same as if you used 1 m steps. The reason is that there would be indentations and protrusions in the coastline that occur at a scale of 1 m and that you would "step over" with your 1 km instrument. But the same problem arises if you were to take 1 cm steps. There would be little protrusions and indentations that the 1 m stick would step over but the 1 cm stick would not. For a smooth curve, the total length asymptotes as the step size becomes small because, "in the small" the coastline is approximately a straight curve. This is just not the case for rocky coastlines though.

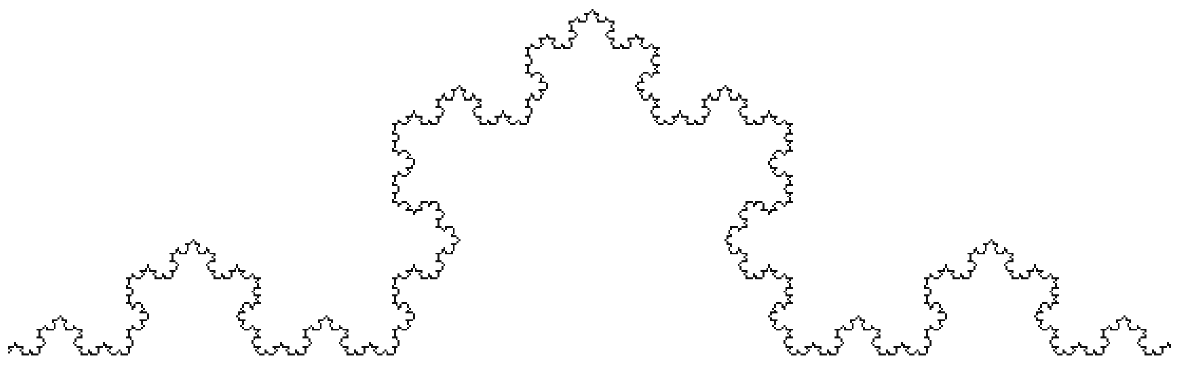
Fractals have been studied by mathematicians for many decades. They became much more popular with the work of Benoit Mandelbrot who pointed out how often real world objects and phenomena have fractal-like behavior. Mandelbrot introduced both new determininstic (non-random) fractals e.g. the Mandelbrot set, and also discussed random fractals like Brownian motion. Here I will touch on some of the basics of these fascinating objects and give some examples.

Koch Curve

Consider a line segment of length 1. Suppose replace it with four line segments of length 1/3 each, as in the figure below. Then we replace each of these line segments of length 1/3 each, by four line segments of length length $\frac{1}{3^2}$ each. This gives a total length of $(\frac{4}{3})^2$. If we repeat this recursively n times, we get a curve of total length $(\frac{4}{3})^n$ composed out of 4^n line segments, each of length $\frac{1}{3^n}$. The figure below shows the construction after 1,2,3,4 steps.



We expand the drawing (by rescaling by a factor of 3) to see the reconstruction after 5 steps.



The length of the curve goes to ∞ as $n \to \infty$. The limiting curve is called the *Koch Curve*.

The Koch curve is said to be *self-similar*. If you compare the whole Koch curve to the part of the Koch curve that is generated by any single one of the four line segments of length $\frac{1}{3}$ (at step 1), then you get the same curve except that it is scaled by a factor of $\frac{1}{3}$ (and then translated and perhaps rotated).

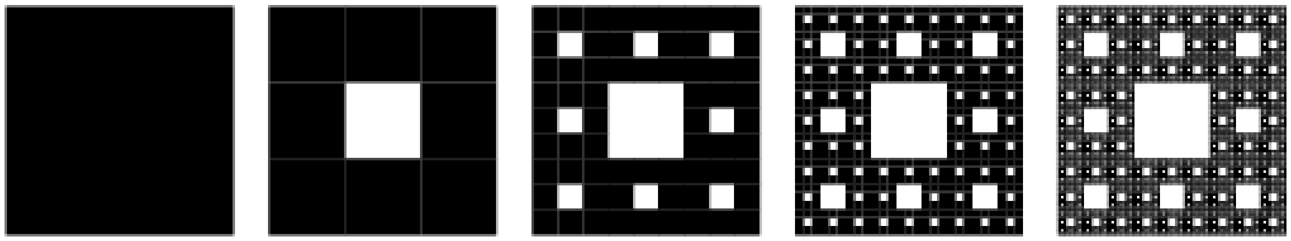
We can write a recursive function for drawing a Koch curve in OpenGL as follows:

```
def koch(i):
    if i == 0
        drawline()
    else if i > 0
        glPushMatrix()
        glScalef(1.0/3,1.0/3,1.0/3)
        koch(i-1)
        glTranslatef(1.0, 0.0, 0.0)
        glRotatef(60, 0.0, 0.0, 1.0)
        koch(i-1)
        glTranslatef(1.0, 0.0, 0.0)
        glRotatef(-60, 0.0, 0.0, 1.0)
        glRotatef(-60, 0.0, 0.0, 1.0)
        koch(i-1)
        glTranslatef(1.0, 0.0, 0.0)
        glRotatef(60, 0.0, 0.0, 1.0)
        koch(i-1)
        glPopMatrix()
```

Sierpinski carpet

Here's another interesting example. Begin with a unit square, and partition it into nine subsquares, i.e. a 3×3 grid of squares, each of area $\frac{1}{9}$. We then delete thecentral subsquare, leaving a remaining area of $\frac{8}{9}$. For each of these eight remaining subsquares, we again partition it into nine sub-squares and delete the central one. The total area remaining is now $(\frac{8}{9})^2$. We proceed recursively n times, and we are left with a total area of $(\frac{8}{9})^n$. This area goes to 0 as $n \to \infty$.

Notice there are lots of points that are never deleted. For example, the boundary of any of the subsquares is never deleted since we only delete interiors. Each subsquare at step $n \geq 1$ has perimeter (boundary length) $4(\frac{1}{3})^n$ and there are 8^{n-1} such subsquares. Thus the total perimeter (length) of the boundary of the deleted subsquares is $8^{n-1}4(\frac{1}{3})^n$ which goes to ∞ as $n \to \infty$.

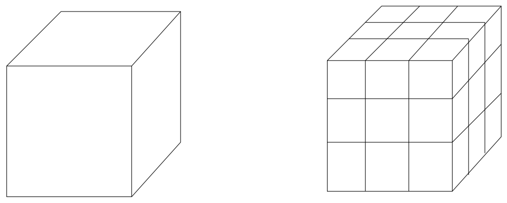


The limiting shape is called the *Sierpinski carpet.*

Both the Koch Curve and Sierpinski carpet are infinitely "long" (if we try to measure them in 1D, as curves) but have infinitesimal "area" (if we try to measure them in 2D, as areas). Both of these seem to have dimension somewhere between 1 and 2. Intuitively, the Sierpinski carpet consists of more points than the Koch curve, but it is not yet clear how to quantify this.

Sierpinski Cube

The Sierpinski cube is the 3D version of the Sierpinski carpet. We partition a unit cube, rather than a square. Partition the unit cube into 27 subcubes ($3 \times 3 \times 3$) each of volume $\frac{1}{27}$. [**Here I give a slightly different version than the one in the lecture.**] In this version, I delete only the interior of the central subcube. This leaves a total volume $\frac{26}{27}$. For each of these remaining 26 subcubes, again partition it into 27 subcubes (now each of volume $(\frac{1}{27})^2$) and delete the central subcube from each of these. The total remaining volume is now $(\frac{26}{27})^2$. If we repeat this n times, the remaining volume is $(\frac{26}{27})^n$, which goes to 0 as $n \to \infty$.



Fractal dimension

What do we mean by dimension anyhow? One simple idea is that we are referring to the rate at which the size of a set grows as we *scale* it. Suppose we are talking about subsets of $\Re^3$. We are familiar with talking about points (0D), curves (1D), surfaces (2D), and volumes (3D). But what do we mean by nD ?

If we take a line segment of length 1, and we scale it by a factor, say $S = 2$, then get a new line segment such that we require $C = 2$ copies of the original line segment to cover the newly scaled line segment. What about area? Suppose we have a unit square. If we scale it by $S = 2$, then get a new square of area S^2 and so we need $C = S^2$ of the original squares to cover the new square. The same idea holds for volume. If we have a unit cube and we scale it by $S = 2$, then we get a new cube of volume S^3, which means that we need $C = S^3$ unit cubes to cover this new cube.

In general, if have an object that is of unit size and is of dimension D, and we scale this object by $S > 1$, then we need how many copies to cover the newly scaled object? We need

$$C = S^D$$

objects of unit size (measured in that dimension). Taking the log of both sides, we get

$$D \equiv \frac{\log C}{\log S}$$

Verify for yourself that this definition holds for a line segment, square, and cube. Now, let's use this observation as a *definition* of dimension and apply it to the objects introduced above.

What is the dimension of the Koch curve? If one starts with a Koch curve, and makes $C = 4$ copies, one can piece them together to make a scaled Koch curve where the scale factor is $S = 3$. Thus, according to the above definition, the dimension of the Koch curve is $D = \frac{\log 4}{\log 3} \approx 1.26$. Note that the dimension of the Koch curve is strictly between 1 and 2 which is consistent with the idea that its dimension is greater than that of a line segment and less than that of a square.

What about the Sierpinski carpet? If we make $C = 8$ copies of the Sierpinskicarpet described above, then we can piece them into a 3×3 grid (with the center one missing) and this gives a scaled Sierpinski carpet such that the scale factor is $S = 3$. Thus, the dimension of the Sierpinski carpet $D = \frac{\log 8}{\log 3} \approx 1.88$. Note that the dimension of the Sierpinski carpet is greater than the dimension of the Koch curve, and that both have dimension strictly between 1 and 2.

For the 3D version of the Sierpinski carpet, we can take $C = 26$ copies and make a 3D Carpet that is a scale version where the scale factor is $s = 3$. Thus the dimension is $D = \frac{\log 26}{\log 3} \approx 2.96$. Notice that this is very close to 3, but still less than 3.

Random fractals e.g. Midpoint displacement method

Just as a sphere and and a square are too symmetric to be for modelling smooth objects, the objects we have just defined are too symmetric to be used to model natural complicated objects like coastlines, rock faces, broccoli, and clouds. To model these, we need to destroy the symmetry. We can do so by adding random perturbations. Here are some basic ideas of how to do so.

Suppose we take a 1-d function $f(x)$ with endpoints $f(0) = a$ and $f(1) = b$. We can make a line joining these two endpoints. The midpoint has $f(\frac{1}{2}) = (a + b)/2$. If we add a random amount to this midpoint value, we get $f(\frac{1}{2}) = \frac{a+b}{2} + \delta$, where δ is chosen from a suitable probability density function. For example, it might be normal (Gaussian) with mean zero and standard deviation σ.

We then repeat this "midpoint displacement" for the two line segments from $(0, f(0))$ to $(\frac{1}{2}, f(\frac{1}{2}))$ and from $(\frac{1}{2}, f(\frac{1}{2}))$ to $(1, f(1))$. We partition each line segment into two halves, giving us midpoints $x = 1/4$ and $x = 3/4$. We then add a random perturbation δ to each, chosen from a suitable probability density function. And so on... At step n, we have 2^n line segments. We can repeat this recursively, adding more and more detail as we wish.

How should we choose probability density functions to generate the δ values? There is no single answer to this. It depends on how rough we wish the curve to be. If we perturb only by a little, then the curve will be close to smooth. If we perturb by alot, then the curve will be very rough.

On the next page is Python code for midpoint displacement algorithm. It takes two parameters. One is a `roughness` parameter which is a number between 0 and 1. The other is the standard deviation of the displacement, which has a normal distribution – see Python `numpy.random.normal`. (If you don't know what a normal distribution is, you probably want to look it up.) At each level of the recursion the standard deviation is reduced by some factor, namely the `roughness` parameter. If you use a roughness parameter of $\frac{1}{\sqrt{2}}$ then the curve you get has fractal-like properties, namely statistical self similarity. (To state this claim more carefully and to prove it is true requires mathematics beyond this course.)

```
def  midpointDisplacement(a, roughness, stdev):
   size = len(a)
   newStd = stdev * roughness
   if (size <= 2):
       return a
   else:
       middle = size / 2
       a[middle] = (a[0] + a[size-1])/2 +random.normal(0, newStd)
       a[0:middle+1]  = midpointDisplacement(a[0:middle+1], roughness, newStd)
       a[middle:size] = midpointDisplacement(a[middle:size], roughness, newStd)
   return a
```

The slides show several examples of curves generated with different roughnesses. [At the time of this writing, I am still trying to decide whether to give you something on this material in A2 or A3. I may edit this later once this issue has been settled.]

L-systems

Earlier this lecture I gave recursive code for drawing a Koch curve. Let's write the code in a slightly different way, where we describe how to generate the Koch curve by replacing each line with a four smaller lines in a particular arrangement.

Let [and] denote `glPushMatrix` and `glPopMatrix`, respectively. Let **L** refer to the drawing of a unit line in the x direction, and let **T** denote a unit translation in the x direction. Let **S** be scaling by a factor $\frac{1}{3}$. Let **R** be a rotation by 60 degrees counterclockwise and let **R′** be a rotation 60 degrees clockwise. The first step of approximating a Koch curve is just to draw a line of length 1, so the string is just **L**. In the second step, the string L is replaced as follows: In the third step, each occurence of **L** in the second step is replaced (recursively) by the same substitution, etc.

$$\mathbf{K} \rightarrow \; [\; \mathbf{S\ K\ T\ R\ K\ T\ R'\ R'\ K\ T\ R\ K}\;]$$

Note that this recursion, as written, doesn't have a base case. Later this lecture we will add notation to indicate a base case.

This idea of recursively replacing symbols with strings will be familiar to most of you. It is reminiscent of formal grammars that are used to define programming languages e.g. parse trees. Here we will not be interested in *parsing* such strings, however. Rather we will be interested only in the simpler problem of *generating* such strings.

This idea of generating a shape by recursive substitution was developed by Astrid Lindemayer, and the method has come to be known as L-systems. Lindemayer was interested in growth and form of biological structures, in particular, plants. The idea is that the parts of a plant (stems, leaves, flowers, branches, etc) at any given time is described by a string that has been generated by a sequence of "productions". As the plant grows over time, some parts grow, some parts create new parts, some parts die. To describe such growth formally, some symbols in the string are replaced by new symbols, and other symbols disappear.

L-systems began to be used in computer graphics in the 1980's. Download the book "The Algorithmic Beauty of Plants" by Przemyslaw Prusinkiewicz[1] and Astrid Lindemayer. The book was published in 1990 and is now out of print. It is available online at `http://algorithmicbotany.org/papers/#abop` Have a look through some of the examples, in particular, Chapter 1 (p. 25) of that book.

For those of you who are familiar with formal grammars: an L-system is defined by a set of symbols (alphabet) $\mathcal{A}$ which includes a special symbol $a \in \mathcal{A}$ called an *axiom* or starting symbol. There is also a set of *production rules*

$$P : \mathcal{A} \rightarrow \mathcal{A}^*$$

where $\mathcal{A}^*$ is the set of strings made up of symbols from $\mathcal{A}$ which may include the empty string. A specific production $p \in P$ is written

$$p : a \rightarrow \chi$$

where $\chi \in \mathcal{A}^*$. If no production is explicitly specified for a symbol a, then we assume the identity production

$$p : a \rightarrow a \ .$$

While L-systems are similar to formal grammars, there is a key difference: with L-systems the productions are applied to each symbol in the string *in parallel*. With formal grammars, only one production is applied at each step (and the ordering in which productions are applied is often important for the analysis). Also, we use L-systems to generate sequences, not to analyze (parse) them.

Example: plant

Here is an example for drawing a cartoon plant. This example was modified slightly from the example given in the book Algorithm Beauty of Plants (p. 25, Fig. 1.24d). The recursion can be written:

$$\mathbf{P} \rightarrow \ [\ \mathbf{S\ D\ T}\ [\ \mathbf{R\ P}\ ]\ \mathbf{D\ T}\ [\ \mathbf{R'\ P}\ ]\ \mathbf{R\ P}\ ]$$

and the base case is:

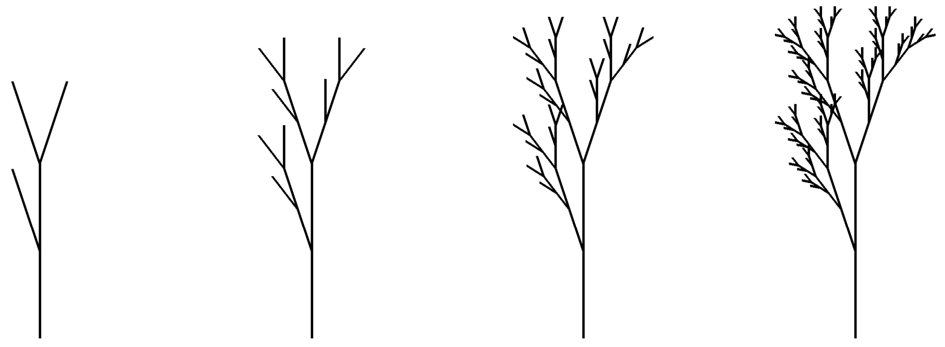
$$\mathbf{P} \rightarrow \mathbf{D}$$

Here

- **S** is a scaling by 0.5 in x and y
- **R** is a counterclockwise rotation by 30 degrees
- **T** is a unit length translation in y

Shown below are four stages of these productions. I have not bothered to include the zeroth stage, in which there is a single vertical line.

[1]Professor Prusinkiewicz is at the University of Calgary and here is his lab website `http://algorithmicbotany.org/`



Parametric L-systems

In the above examples, the symbols had parameters associated with them. For example, the rotations had a particular angle, the translation have a direction, scaling has a scalar. Since the symbols are just functions calls, these parameters would act just as function parameters. In the context of OpenGL, this is just obvious – we're talking about the parameters of the rotate, translate, and scaling functions. Another example of a parameter is the level for the recursion.

You can add the parameters in the obvious way, e.g $\mathbf{S}_x(\frac{1}{3})$ or $\mathbf{R}_z(30)$. Indeed the only reason I didn't include that notation originaly was to keep it simple!

Here is how you could write the conditions of the recursion more explicitly:

$$\begin{aligned} p_1 &: \mathbf{P}(n) : n > 1 \rightarrow [\ \mathbf{S}\ \mathbf{D}\ \mathbf{T}\ [\ \mathbf{R}\ \mathbf{P}(n-1)\ ]\ \mathbf{D}\ \mathbf{T}\ [\ \mathbf{R}'\ \mathbf{P}(n-1)\ ]\ \mathbf{R}\ \mathbf{P}(n-1)\ ] \\ p_2 &: \mathbf{P}(n) : n = 0 \rightarrow \mathbf{D} \end{aligned}$$

Open L-systems

Another powerful technique is to allow the productions to depend on global variables. This allows plants to interact with each other and with the environment. For example, suppose we consider the roots of a plant growing into the ground. We could define a 3D array `water(x,y,z)` that represents the amount of water at each point in the soil. As the roots grow, the plant absorbs water from the soil. At each time step, a root growth production could occur if there is a sufficient amount of water present in the grid cell closest to the tip of the root. Moreover, as the root grows, it could decrease the amount of water in that grid cell.

Another idea is to allow a branches to grow only if they receive a sufficient amount of light from the sky. At each time step, you could cast a set of rays toward the sky and count the percentage of rays that do not intersect branches or leaves above. If the number of rays that "see the sky" is sufficiently large, then you could allow the growth production to occur; otherwise, not.

Finally, in the lecture, I also mentioned probabilistic L-systems where productions could occur or not, based on the outcome of some random event (e.g. a coin flip). See the book Algorithmic Beauty of Plants for examples of this, and many more examples too.