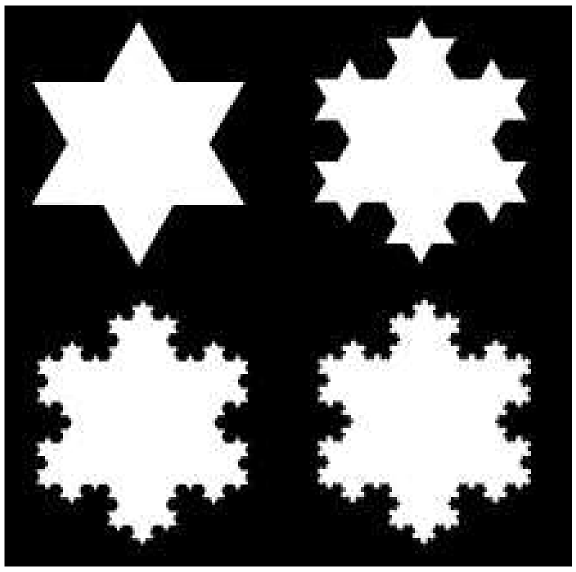
Python: Draw

Nick Veitch puts his maths head on to combine Pythagoras' theorem, Python, Clutter and Cogls to produce beautiful fluffy Koch snowflakes.



› **This shows the first, second, third and ninth iteration of a Koch snowflake. After that, it's hard to see the difference.**

Our expert

Nick Veitch
The Veitch family motto is, "Our fame spreads through our code". Or something like that.

Previously in this series, we've played with actors and stages, and used the power of additional libraries such as *Gstreamer* and *Cairo* to create more objects and animate them until they were very sorry. It's time now to turn our attention to actors once again, but this time we're not going to limit ourselves to the meagre rectangles and text that *Clutter* provides for us – we're going to generate our own. In order to do this, we're going to need to make use of some of the primitive methods for manipulating the underlying GL objects – it's time to play with Cogls.

Just before we do that though, we need to know what handsome shape our actor will take. Frankly, basic Euclidean shapes are a little dull, if useful, so let's create something a bit more interesting – a Koch snowflake!

Let it snow

A Koch snowflake, or Koch curve, is a particular type of fractal. Since fractals are procedural drawings for the most part, they adapt readily to being drawn by computers, and I'm sure that many computer science lessons have been spent trying to draw similar items in a few lines of Basic, Pascal or whatever redundant language they teach kids these days (it was Algol and Fortran in my day).

The basic concept is simple. First up, draw an equilateral triangle. Then for each side, create a further equilateral triangle that's one-third of the length of the existing side and place it in the middle. Repeat the previous step until you get bored. See the image above for some of the shapes that you can create after a few iterations.

For some reason, the usual way of solving this problem is to use a recursive algorithm that calls itself. While these may seem clever and neat, it's far from the ideal solution to the problem. Aside from being a bit tricky to understand, it's incredibly wasteful. You may also come across some hard recursion limits in Python, because it balks at the idea of adding another function call to the stack. By default, Python will only allow 1,000 levels of recursion, and while it's possible to set some system variables to extend this, certain platforms do have a hard limit.

For our Koch algorithm, we'll take a less flashy, but more processor-friendly and reliable approach (which will become important for our object later). Quite simply, we start with a

Things you'll need

Obviously, before you start you'll need Python and the Python *Clutter* module. Both are readily available in your distro repositories, assuming you're running a distro that has been updated in the last year or so. It's usually safer to get them from there, but you can check out the latest source for *Clutter* at **www.clutter-project.org**.

» **Last month** We used *Clutter* to put buttons on their best behaviour.

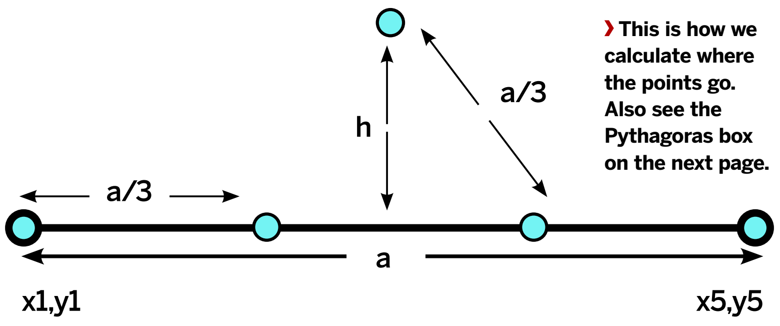
Koch snow

LINUX FORMAT On the DVD » Tutorial code

list of three points, which happen, more or less, to make an equilateral triangle. We'll then create a loop that works its way through this list, and adds three points in between each pair on the list (not forgetting the pair that includes the first and last point). So, each time the loop is processed, it adds an extra level to the fractal. It's easy, and it also only takes about half as long as a recursive solution:

```
def generatekoch(depth=4):

  sqrtof3=1.7320508075688772

  pointlist=[(0,50),(75,180),(150,50)] # an equilateral triangle
more or less
  for i in range(depth):
    newlist=[]
    for p in range(len(pointlist)):
      x1=pointlist[p][0]
      y1=pointlist[p][1]
      if p==len(pointlist)-1:
        x5=pointlist[0][0]
        y5=pointlist[0][1]
      else:
        x5=pointlist[p+1][0]
        y5=pointlist[p+1][1]
      dx=x5-x1
      dy=y5-y1
      x2=x1+(dx/3.0)
      y2=y1+(dy/3.0)
      x4=x1+(2*dx/3.0)
      y4=y1+(2*dy/3.0)
      x3=(x1+x5)/2 + (sqrtof3 * (y1-y5))/6 #see diagram
      y3=(y1+y5)/2 + (sqrtof3 * (x5-x1))/6

      newlist.append((x1,y1))
      newlist.append((x2,y2))
      newlist.append((x3,y3))
      newlist.append((x4,y4))
      #point 5 is already in the list
    pointlist = newlist
  return pointlist
if __name__ == "__main__":
  list=generatekoch(3)
  print list
```

Inside the loop, the values correspond to the five points along the new line. The first and last are the ones we fetch from the list; the other three we have to work out. The second and fourth are one-third and two-thirds of the distance along the line between the initial pair, so those are easy enough to work out. The third point is the apex of the new triangle we've drawn, which is a little trickier. Fortunately, Pythagorean equations for equilateral triangles collapse quite nicely, so all that we need to calculate this is the square root of 3, which we can borrow from the *math* library (or you could just write



› **This is how we calculate where the points go. Also see the Pythagoras box on the next page.**

in a decent approximation to save time). If you want to think about the maths, just remember that an equilateral triangle is two right-angled triangles back to back. See the diagram above to help understand the concept in detail.

We used a **for** loop here rather than use the list as an iterator, because it's easier if you want to work with two values from the list. We write out a new list of points rather than inserting values into the old one because, apart from anything else, it rather mucks up the loop counter.

There's a small caveat to this generation of snow – the maths relies on the original list being in a clockwise point order, otherwise it reads the shape inside out (which nevertheless produces an interesting shape).

Making actors

Now we know what we're going to draw, we can start building our actor. *Clutter* has a metaclass for actors, which is an object template we can use. This means that without filling anything in, if we base a new class on **clutter.Actor**, it will inherit a range of methods and properties.

Clutter objects themselves are derived from *gobject*, which are a part of the Gnome Foundation's *GLib* library (not to be confused with *Glibc*), which is a large library of cross-platform data structures. This is important later, because we'll need to know a few bits of *GLib* to make our code work. For now though, let's just build a simple triangle actor. Open up a terminal and type **python** to run Python in interactive mode, then enter the following (or if you're lazy, copy and paste from the listing files on the **LXFDVD**)

```
>>> import gobject
>>> import clutter
>>> from clutter import cogl
>>> class Triangle (clutter.Actor):
...     def __init__ (self):
...         clutter.Actor.__init__(self)
...         self._color = clutter.Color(255,255,255,255)
...     def do_paint (self):
...         (x1, y1, x2, y2) = self.get_allocation_box()
...         width=x2-x1
...         height=y2-y1
```

»

```
...         cogl.path_move_to(width / 2, 0)
...         cogl.path_line_to(width, height)
...         cogl.path_line_to(0, height)
...         cogl.path_line_to(width / 2, 0)
...         cogl.path_close()
...         cogl.set_source_color(self._color)
...         cogl.path_fill()
...
>>> gobject.type_register(Triangle)
<class '__main__.Triangle'>
>>>
```

As well as the usual *Clutter* library, we've also imported **gobject**, and specifically, the **cogl** library. The latter is simply to shorten the namespace (instead of writing **clutter.cogl.path_move_to** we can omit the first **clutter**). *Gobject* is necessary, not only for when we want to add properties and signals, but also for registering the object type, which is a necessary part of the *Clutter* setup. You can see we did this immediately after making our class – it needs to be done before we make any **Triangle** elements.

In the class itself, we've defined an **__init__** method, as is usual. The **Actor** metaclass has an **init** method of its own, but we're overwriting that to add our own functionality (in this case, merely setting up a colour variable). However, we can still call the default **__init__** method by making a specific call to it, which will set up the normal *Clutter*-type things that we don't want to be bothered with.

The **paint** method is the important one, and one that uses the Cogl functions. Each actor object has a **paint** method, which is called whenever the object needs to be drawn. This method is called by *Clutter* itself, and may need to be called numerous times in the course of, for example, an animation.

The drawing commands are pretty easy to understand. Imagine you have a pen – you need to move it to the position you want to start at, then draw the path to various points. The **path_close** method joins up the first and last points to complete a shape, which is necessary if you want to fill it. There are lots of extra drawing commands (most have relative and absolute versions) and you can check out the documentation for the primitives on the main *Clutter* website here **http://clutter-project.org/docs/cogl/stable/cogl-Primitives.html**. Of course, this is the C documentation, but it's easy enough to see how most of the methods work.

Magic painting

The only other magic trick in this code is at the beginning of the **paint** method. The call to **get_allocation_box** uses one of the inherited **Actor** methods to fetch the drawing size of the actor, which returns two points giving the limit of the drawable area. You don't have to worry about the size of the object at the moment – whenever you call an actor's **set_size()** method, the various *Clutter* internals will take care of updating the size of the actor, and the drawable area will change accordingly.

We can test our triangles now, by doing the usual setup of a stage and adding the objects:

```
>>> stage=clutter.Stage()
>>> stage.set_size(400,400)
>>> t=Triangle()
>>> t.set_size(50,50)
>>> stage.add(t)
>>> stage.set_color(clutter.Color(0,0,0,255))
>>> stage.show_all()
>>> tt=Triangle()
>>> tt.set_size(100,100)
>>> tt.set_position(200,200)
>>> stage.add(tt)
```

So, we can now make triangles and even animate them:

```
>>> tt.animate(clutter.EASE_IN_QUAD,2000,'y',0)
<clutter.Animation object at 0x97b334c (ClutterAnimation at
0x98a1990)>
```

But all is not as it seems. Try changing the colour of your triangle, in this way:

```
>>> tt.set_color(clutter.Color(255,255,0,255))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Triangle' object has no attribute 'set_color'
```

Inheritance tax

Not all of the functionality of a standard actor is inherited. Unlike the built-in **rectangle** object, we have no method for setting the colour of the **Triangle** object we made, unless we add that to our class.

```
    def set_color (self, color):
        self._color = color
```

This snippet would obviously have to be part of the main class, and in this instance, it accepts a standard **clutter.Color()** object, although you could change this. So, adapting this to our Koch snowflake shape, we would get something like this (note that the Koch generator, shown elsewhere, has been removed for brevity):

```
import gobject
import clutter
from clutter import cogl
class Koch (clutter.Actor):
    """
    Koch snowflake Actor
    has extra property '_iterations', to control depth of
generated fractal
    """
    __gtype_name__ = 'Koch'
    def __init__ (self):
        clutter.Actor.__init__(self)
        self._color = clutter.Color(255,255,255,255)
```

Pythagoras' theorem

Pythagoras proved that, for a right-angled triangle, the square of the hypotenuse is equal to the sum of the squares of the other two sides. By hypotenuse, he means the longest side – the one opposite the right angle. By dint of it being an equilateral triangle, the length of the base side is half that of the hypotenuse, which will help greatly.

Say, in our case, the height is y, and the length of the hypotenuse is x. This gives us:

$$y^2 + (x/2)^2 = x^2$$
$$y^2 = x^2 - (x/2)^2$$
$$y^2 = x^2 - x^2/4$$
$$y^2 = 3x^2/4$$
$$4y^2 = 3x^2$$

Then take the square root of both sides:

$$2y = (\sqrt{3})x$$
$$y = ((\sqrt{3})x)/2$$

» **Never miss another issue** Subscribe to the #1 source for Linux on p66.

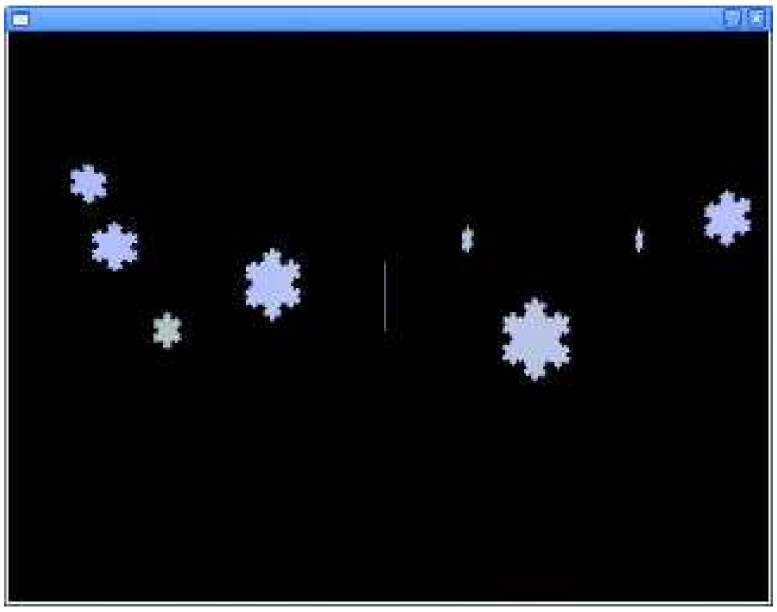
```
        self._iterations = 2
        self._points=[(0,0),(0,0),(0,0)]
    def generatekoch(self,dimension):
        ### already explained elsewhere
        return pointlist

    def set_color (self, color):
        self._color = color
    def __paint_shape (self, paint_color):
        pointlist=self._points
        cogl.path_move_to(pointlist[0][0], pointlist[0][1])
        for point in pointlist:
            cogl.path_line_to(point[0], point[1])
        cogl.path_close()
        cogl.set_source_color(paint_color)
        cogl.path_fill()
    def do_paint (self):
        paint_color = self._color
        real_alpha = self.get_paint_opacity() * paint_color.alpha /
255
        paint_color.alpha = real_alpha
        self.__paint_shape(paint_color)

    def set_size (self,width,height):
        clutter.Actor.set_size(self,width,height)
        dimension=float(min(width,height))
        self._points=self.generatekoch(dimension)

    def set_iterations (self,number):
        self._iterations=number
        (x,y) = self.get_size()
        dimension = min(x,y)
        self._points=self.generatekoch(dimension)
        self.do_paint()
gobject.type_register(Koch)
```

As you can probably see here, we store the list of points as a property – it would get painfully slow if you had to generate an eighth-level shape every time you needed to paint it, especially considering it may need to be painted many times a second! The generation is called whenever the size or the number of iterations is changed. This means that the points will usually generate twice when you set up an object, assuming you change the default number of iterations. Unfortunately, this is unavoidable, unless you want the 'depth' of the fractal only to take effect when the size is changed.

We've overwritten the **set_size** method of the **Actor** class to make sure our generated points reflect the size of the object, but it's still important to call the parent **set_size** method to ensure buffer allocations and such are updated.

To demonstrate our new shapes, here's a simple sample generator to test your objects with:

```
import clutter, random
from clutterKoch import Koch
stage = clutter.Stage()
stage.set_size(640, 480)
stage.set_color(clutter.Color(0,0,0,255))
stage.connect('destroy', clutter.main_quit)
for i in range(10):
    s = Koch()
    x=random.randint(20,90)
    s.set_size(x, x)
    s.set_iterations(6)
    s.set_color(clutter.Color(200,200,random.
randint(200,255),255))
    z=random.randint(0+x,640-x)
    zz=random.randint(x,x+200)
    s.set_position(z,-zz)
    stage.add(s)
    s.animate(clutter.EASE_IN_QUAD, 5000,'y',x+random.
randint(480,550),'rotation-angle-y',random.randint(180,720))
stage.show()
clutter.main()
```


› **Here comes the snow again – you can generate as many flakes as you like, whatever the weather, with your super soaraway *Linux Format* code.**

As long as your Actor file (in this case **clutterKoch.py**) is in the same directory, you can run this and generate random shapes. As you can see, we can animate and rotate our creations. The points don't need to be regenerated for this, because they're GL objects at this point, so the graphics card takes care of drawing them in the right place.

Taking it further

Cogls aren't just useful for drawing shapes. You can change many aspects of the display using this interface to OpenGL, even to the extent of generating your own shaders to use. There's more documentation on the various abilities of Cogls at the *Clutter* website. However, as we mentioned earlier, this is intended for C programmers, so you'll have to spend some time experimenting to get things working in Python. Have a look at **http://clutter-project.org/docs/cogl/stable**.

The other thing we've been remiss in is setting up our **gobject** properly. Essentially, all we've done here is the bare minimum to get the Actor to work. To be nice players with the system, we should register the *Gobject* properties of our object, and we might even want to set up some signals for it.

Gobject is great, but it's a little complicated and long-winded, so unfortunately there's no space to explain it fully here. The *PyGTK* documentation has lots of useful information on *Gobjects* though, so if you're interested, it's well worth checking out. Head over to **www.pygtk.org/docs/pygobject** to read more. LXF

» **Next month** A bonanza double-sized tutorial on building a complete app.