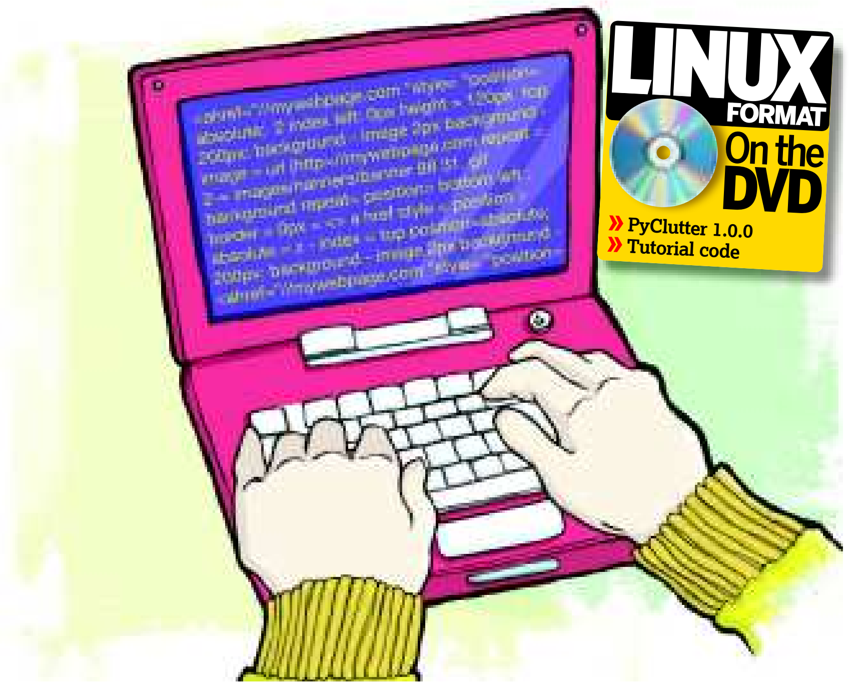
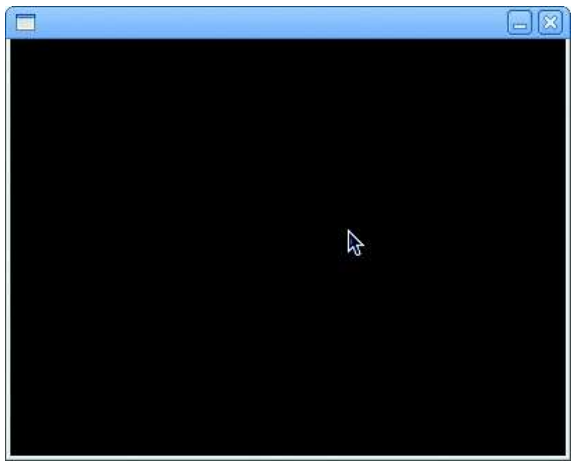
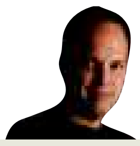
Clutter: Code a

Tidying up some code with Clutter, **Nick Veitch** takes you far from the command line into a new realm of technicolour graphical possibilities




› **It's a bit dark in here... Could be the promising start of a 3D adventure game, perhaps. Or your first *Clutter* effort!**


Our expert

Nick Veitch, as all of his colleagues will attest, is possibly the world's greatest living expert on clutter. He knows a bit about Clutter too.

We have built a lot of web-based wonder in our Python tutorial series so far, but only rarely have we touched on using a GUI to display stuff graphically to the user. One of the reasons for this is that, for the most part, GUI code gets very big very quickly, so the whole tutorial would be taken up by just drawing a panel and a few buttons on the screen.

We're going to take a break from being so user-unfriendly for a while, as for the next few months we're going to be building applications using the *PyClutter* library. If you don't know much about *Clutter*, check the boxout over the page. For the first tutorial we're going to build a small but useful little utility to get to grips with how *Clutter* and *PyClutter* work. As *Clutter* has until recently not been that widely used, there is a dearth of documentation and examples, so hopefully the code we will cover here will give you an idea of how we can use it practically within our Python web apps.

Our task this issue is to create an app that will show us the current network speeds for our internet connection. Yes, there are plenty of monitors out there, but this will be our own, and delivered in about 70 lines of simple code.

The first thing you need to get to grips with in *Clutter* is the basic terminology. Unlike other GUI toolkits, which usually define objects like windows or panel, *Clutter* refers to the visual area as a 'stage'. To continue the analogy, objects that appear on (or actually, in, but it sounds weird to say it) the stage are called 'actors'. It makes more sense when you start coding it, and the names don't seem so strange after a while. The thing about the actors is that they have more properties than a standard widget because they actually exist in a 3D environment, rather than a 2D one.

All the world's a stage

Anyway, enough hyperbabble – it will make more sense when we write some code. Open up your standard Python environment (mine is a *Bash* shell, but you can use some of those fancy ones if you like), and let's create our very first *Clutter* script...

```
>>> import clutter
>>> stage = clutter.Stage()
>>> stage.set_size(500,300)
>>> red=clutter.Color(255,0,0,255)
>>> black=clutter.Color(0,0,0,255)
>>> stage.set_color(black)
>>> stage.show_all()
```

When you're done, click in the Close gadget on the window that opened. I know it didn't do anything amazing, but it does have the potential to! Let's take a look at what just happened. The first line obviously loaded the *Clutter* module. In turn, *Clutter* opens a few more modules itself – back-end stuff that links into display libraries to be able to put things on the screen. Next up we created a stage object. the stage is like a viewport – an area where your actor objects can play.

Setting the attributes is as simple as calling some methods for the stage class, in this case a size and a colour. The parameters for the size method are x and y dimensions,

» **Last month** We made the web respect our authority with OAuth.

system monitor

and the colour is taken from the **clutter.Color** object (which takes values for RGB and alpha). As with other GUI toolkits, we should cause the object to be shown before any of it is drawn on the screen, which is what the final command does.

But what of our actors, the objects that we want to show on the screen? Let's add some text objects:

```
>>> a=clutter.Text()
>>> a.set_font_name("Sans 30")
>>> a.set_colour(red)
>>> a.set_text ("Hello World!")
>>> a.set_position(130,100)
>>> stage.add(a)
```

Now we've added a text object, our first actor. Hopefully it will be fairly clear what the methods are doing – picking a font, a colour, setting the text string and positioning it on the stage. The final call in the code example adds the actor to the stage, and until this point, you won't be able to see it. Now that it's there though, you can continue to play around with it - try setting it to a different position or adding new colours.

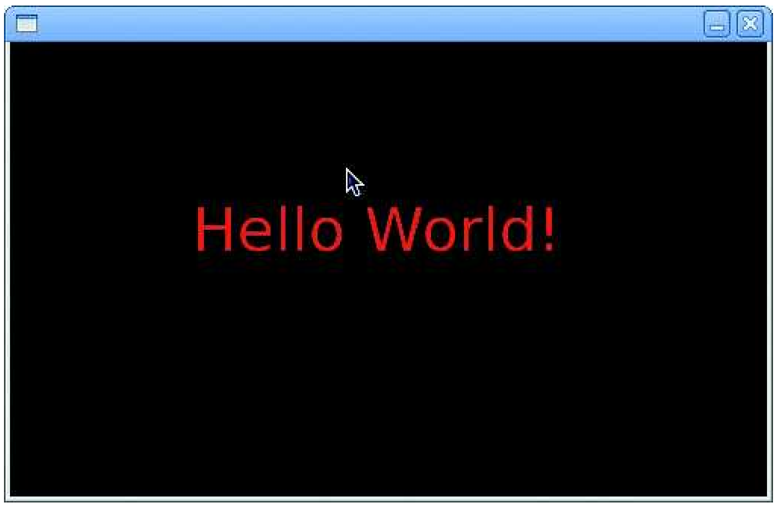
As I mentioned earlier, the *PyClutter* documentation is scanty, but we can gain some solace in the fact that Python has good introspection. Try typing in **dir (a)** at this point to see the methods and attributes available for this object.

Our next step is to build a running script, but there's something we haven't covered yet: for all the *Clutter* magic to work properly, we should turn control of the application over to the **clutter.main()** function, but we don't want to do that without some way to exit the program. In such situations, Python will catch Ctrl+C interrupts, so we will have no way of quitting. The answer is to provide some keyboard events.

When the stage window is active, *Clutter* will receive signals for keypresses. All we need to do is provide a callback function that will process that event, and if the correct key has been pressed, quit out of the main loop. You could also assign other actions to some keys, like changing the colour of the stage for example.

```
>>> def parseKeyPress(self, event):
...         if event.keyval == clutter.keysyms.q:
...                 clutter.main_quit()
...         elif event.keyval == clutter.keysyms.r:
...                 self.set_color(red)
...
>>> stage.connect('key-press-event', parseKeyPress)
>>> clutter.main()
```

When run in the interactive Python shell, the **quit** function will not quit Python itself, or even destroy the application; it will just return control to the Python shell. In the case of a running script though, calling the **clutter.main_quit()** method will effectively end the application, or at least the *Clutter* part of it.



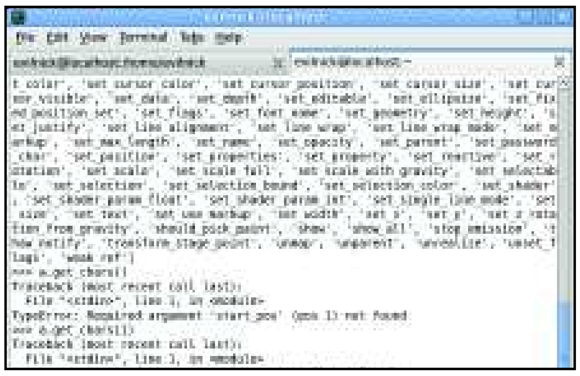
› **The traditional first app, though rather daringly we have left out the comma.**

Time to monitor something

Right, now we have the interface sorted out, how are we going to build an amazing bandwidth monitor? We first need to find out the speed of the network traffic. Whenever I am confronted with a question about some piece of system statistics, I always go and ask my old friend, **proc**. Yes, the »

A note about versions

The *Clutter* library, and consequently the Python module that uses the *Clutter* library, has been updated recently to version 1.0. Normally updates may cause a few inconsistencies between old versions and new versions of software, but in this case there are fundamental differences between the code of versions before and after 0.9. The *PyClutter* module and the *Clutter* library should be available in your distro's repository, but when you install it, make sure you have a version 0.9 (preferably 1.0) or above, otherwise I can guarantee you that none of the code in this tutorial will work. If you think that's a faff, you ought to try writing a tutorial and then discovering the whole library changes...



```
t_color', 'set_cursor_color', 'set_cursor_position', 'set_cursor_size', 'set_cur
sor_visible', 'set_data', 'set_depth', 'set_editable', 'set_ellipsize', 'set_fix
ed_position_set', 'set_flags', 'set_font_name', 'set_geometry', 'set_height', 's
et_justify', 'set_line_alignment', 'set_line_wrap', 'set_line_wrap_mode', 'set_m
arkup', 'set_max_length', 'set_name', 'set_opacity', 'set_parent', 'set_password
_char', 'set_position', 'set_properties', 'set_property', 'set_reactive', 'set_r
otation', 'set_scale', 'set_scale_full', 'set_scale_with_gravity', 'set_selectab
le', 'set_selection', 'set_selection_bound', 'set_selection_color', 'set_shader'
, 'set_shader_param_float', 'set_shader_param_int', 'set_single_line_mode', 'set
_size', 'set_text', 'set_use_markup', 'set_width', 'set_x', 'set_y', 'set_z_rota
tion_from_gravity', 'should_pick_paint', 'show', 'show_all', 'stop_emission', 't
haw_notify', 'transform_stage_point', 'unmap', 'unparent', 'unrealize', 'unset_f
lags', 'weak_ref']
>>> a.get_chars()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Required argument 'start_pos' (pos 1) not found
>>> a.get_chars(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

› **Messing around in the interactive shell is a quick, safe way to finding out about *Clutter* objects and methods.**

» **If you missed last issue** Call 0870 837 4773 or +44 1858 438795.

Quick tip

Keeping track of versions can be a nightmare, but most modules store their version number in **<modulename>.__version__**. Not only is this useful for you to check, but your applications can check for a compatible version before they try and do anything tricky.

/proc pseudo filesystem is the repository of everything you ever needed to know about a running Linux box. **proc** is a huge sprawling mess of files, but the one we want is ***/proc/net/dev***. This lists all the network devices, and reading the file will give you statistics on bytes in and out, packets, dropped packets, errors and so on. The only thing we are interested in are the bytes sent and the bytes received. I know that the number there is a total, and we wanted a speed, but behold the power of **proc** – just open the file again and the magic numbers will have changed. Now, I hope I am not going too fast for you, but simple arithmetic should not be beyond us. If we poll the file every second and subtract the old number from the new number, everything should be fine.

All we really have to do is build a little function that will read in the file, parse it for the information we want, and compute the deltas. Before we leave we will save the old number so we can subtract it next time. Here's how the function should look, more or less:

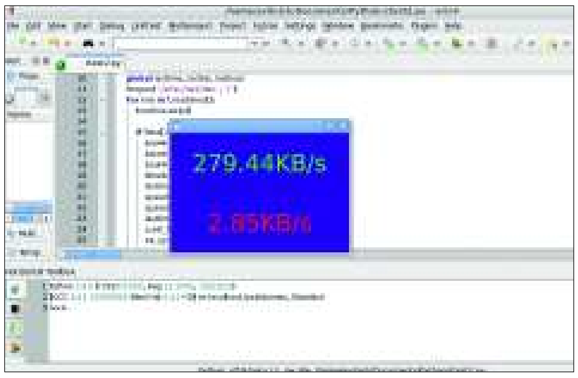
```
devfile=open(‘/proc/net/dev’,’r’)
for line in devfile.readlines():
    line=line.strip()
    if (line[:4] == ‘eth0’):
        line=line[5:].split()
        print line[0],  line[8]
```

Hopefully, this will make some sense to you without me needing to draw diagrams. We read in the file and iterate through the lines, looking for the one that begins **eth0:** – it is necessary to strip the line before searching because the output is padded by an amount to make the tables line up. When we have the correct line, we take of the interface part and split the string up, so we have each of the numbers as part of a list. The counts for bytes in and out happen to be at the 0 and 8 positions in this list. Here we have just printed them out – you can type in the code and see what it gives you. All that needs to be added to that is to convert the strings to integers and store them so we can keep a track of what's going on.

Maths is your friend

The more detail-oriented of you might question whether we take into account the length of time it takes this snippet of code to run. If you want to time this code, go ahead – on my development system it takes 0.0001 seconds to run. In case you're interested, a complete command line app would look something like this:

```
import time
lasttime=1
lastin=0
```



› **Numbers. Coloured numbers. That change. And monitor things. I call this a pretty good start.**

```
lastout=0
def getspeed():
    x=open(‘/proc/net/dev’,’r’)
    for line in x.readlines():
        line=line.strip()
        if (line[:4] == ‘eth0’):
            line=line[5:].split()
            bin=int(line[0])
            bout=int(line[8])
    return (bin, bout)

while True :
    z= getspeed()
    timedelta=time.time()-lasttime
    lasttime=time.time()
    sin=(float(z[0]-lastin))/(1024*timedelta)
    sout=(float(z[1]-lastout))/(1024*timedelta)
    print sin, sout
    lastin=z[0]
    lastout=z[1]
    time.sleep(5)
```

This incorporates a timing function to more accurately calculate the speeds, but bear in mind that we're only talking about a couple of milliseconds, so it doesn't make a lot of difference. It is useful however, if we ever want to alter the timing period elsewhere in the software.

Now what we have to do is to incorporate this functionality into our *Clutter* application. We could just stick the loop at the end of our program and fail to ever call the main *Clutter* loop. We can still update the actor objects whenever we like, but this would be a Bad Thing. The nicer way to do it is to give liberty, autonomy and freedom back to the actors, but make use of an animation timeline to control their text.

Timelines are covered in slightly more detail in the box over the page, but to give you a brief summary, a timeline is just a timer that counts to some value and then emits the programmatic equivalent of a beep – a signal. The signal can be caught and fed to a callback, and as well as itself, you can supply other parameters to the call. For our purposes, we can make the timer call a function that will test the network speed and update our two actors.

The timeline is an object unto itself, but when we execute the connection between the timeline and the callback function, we can pass along our text actor objects too, so the callback function will be able to change them directly. Note that if you're going for more complicated behaviours, this

Why should I care about Clutter?

Clutter is a GPL graphics and GUI library that was originally developed by the OpenedHand team. It was later sold to Intel, which is committed to further development and deployment.

The great thing about *Clutter* is that it's a simple, fast and powerful way to deliver 3D or 2D graphics on a number of platforms. The back-end is essentially OpenGL, but by using the *Clutter* library developers can take advantage of a fast, efficient and friendly way to develop graphically rich apps without messing around with more technical aspects of the OpenGL libraries.

Clutter also forms an integral part of Moblin, the latest attempt to deliver a lightweight but powerful graphical version of Linux to run on mobile devices. Moblin is primarily aimed at Intel Atom based devices, although it will run on other hardware.

doesn't preclude you from having other timers too – you could set one up to change the colour of the objects every second if you wanted, and it needn't interfere with the timeline we have already created. Timelines can be used like threads in a multithreaded app – they aren't quite as flexible, but they are easier to manage and they it easier to deal with animated objects, because you can separate the business of animating the object from the other interactions it has.

```
import clutter
import time
lasttime=1
lastbin=0
lastbout=0
black =clutter.Color(0,0,0,255)
red = clutter.Color(255, 0, 0, 255)
green =clutter.Color(0,255,0,255)
blue =clutter.Color(0,0,255,255)
def updatespeed(t, a, b):
    global lasttime, lastbin, lastbout
    f=open('/proc/net/dev','r')
    for line in f.readlines():
        line=line.strip()
                if (line[:4] == 'eth0'):
            line=line[5:].split()
            bin=int(line[0])
            bout=int(line[8])
            timedelta=time.time()-lasttime
            lasttime=time.time()
            speedin=round((bin-lastbin)/(1024*timedelta), 2)
            speedout=round((bout-lastbout)/(1024*timedelta), 2)
            lastbin, lastbout = bin,  bout
            a.set_text(str(speedin)+'KB/s')
            xx, yy=a.get_size()
            a.set_position(int((300-xx)/2),int((100-yy)/2) )
            b.set_text(str(speedout)+'KB/s')
            xx, yy=b.get_size()
            b.set_position(int((300-xx)/2),int((100-yy)/2)+100 )
def parseKeyPress(self, event):
    # Parses the keyboard
    #As this is called by the stage object
    if event.keyval == clutter.keysyms.q:
        #if the user pressed "q" quit the test
        clutter.main_quit()
    elif event.keyval == clutter.keysyms.r:
        #if the user pressed "r" make the object red
        self.set_color(red)
    elif event.keyval == clutter.keysyms.g:
        #if the user pressed "g" make the object green
        self.set_color(green)
    elif event.keyval == clutter.keysyms.b:
        #if the user pressed "b" make the object blue
        self.set_color(blue)
    elif event.keyval == clutter.keysyms.Up:
        #up-arrow = make the object black
        self.set_color(black)
    print 'event processed',  event.keyval
stage = clutter.Stage()
stage.set_size(300,200)
stage.set_color(blue)
stage.connect('key-press-event', parseKeyPress)
intext=clutter.Text()
intext.set_font_name("Sans 30")
```

It's all about timing

The *Clutter* library uses objects called timelines to do practically everything that needs to be done while an application is running. The timeline is the heartbeat of your script, and makes sure that everything at least makes a good attempt at running together.

Timelines are used extensively for controlling animations and effects within *Clutter*, but you can also use them as your own interrupts to call routines every so often. It does this by emitting signals for events such as **started**, **next-frame**, **completed** and so on. Each of these signals can be bound to a callback function to control something else.

Here is a short example you can type into a Python shell:

```
>>> import clutter
>>> t=clutter.Timeline()
>>> t.set_duration(2000)
>>> t.set_loop(True)
>>> def ping(caller):
...         print caller
...
>>> t.connect('completed',ping)
9L
>>> t.start()
>>> <clutter.Timeline object at
0xb779639c (ClutterTimeline at
0x95b9860)>
```

Hopefully the methods of the timeline object should be easy to follow. The duration is set as a number of milliseconds. The timeline is then set to loop. Here we have created a simple function called **ping**, which just prints out the parameter it was called with. next, we connect the **completed** emitted signal to the **ping** function and start the timeline running. Without any further interaction, the **ping** function will now be called every two seconds, as the timeline completes, until you kill the Python shell.

```
intext.set_color(green)
stage.add(intext)
outtext=clutter.Text()
outtext.set_font_name("Sans 30")
outtext.set_color(red)
stage.add(outtext)
stage.show_all()
t=clutter.Timeline()
t.set_duration(5000)
t.set_loop(True)
t.connect('completed', updatespeed, intext, outtext)
t.start()
clutter.main()
```

Here we've brought together all the elements we have explored in this tutorial. We have created a stage, populated it with actors, and then used the timeline objects in *Clutter* to make them update themselves at our whim. But so far we have only scratched the surface of *Clutter*'s graphical capabilities. We haven't even learned about behaviours or animations yet, never mind the alpha channel effects. Please trust us that we will be including these in our next project. LXF

› The main clutter website at www.clutter-project.org doesn't have much help for Python users, but there is lots of background info and plenty of C documentation.

» Next month We'll build an animated feed-reader and make it look brilliant.