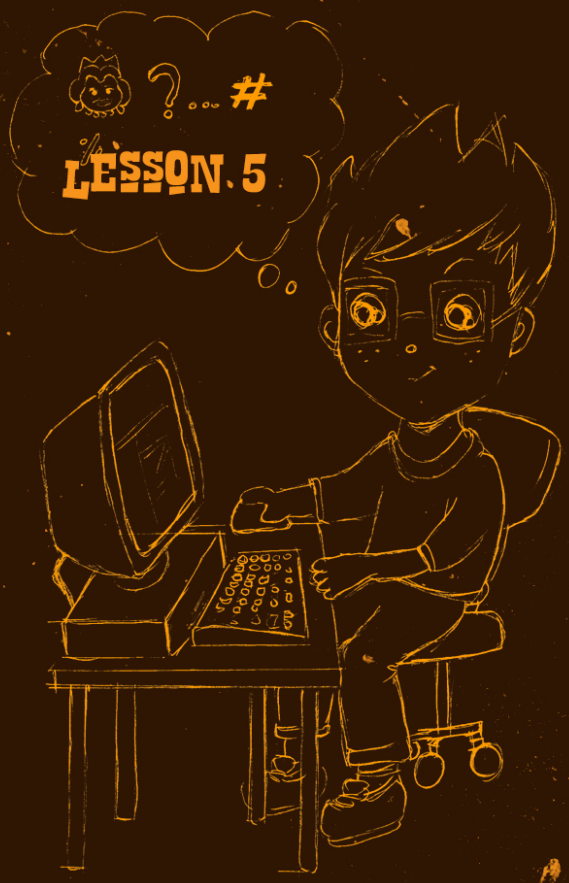
PYTHON SCRIPTING FOR SMART AND CURIOUS COMPOSITORS



Gianluca Dentici
www.gianlucadentici.com

LESSON 5 !!

Hey nuke-python gang! Welcome back to another funny lesson ! Probably the last before Christmas !
If you got up to this point you rock !!! Well we have just passed halloween and I couldn't help but carving my own Jack Skeleton! I really hope you guys made your own too and more over I hope you didn't forget to bin it after 4 days or by then your house will stink of rotten corpses, trust me!
So find here my big masterpiece! However if you made one inspired to our beloved film world please feel free to send it over, I will be delighted to admire it!

But now wrap all the things up and prepare for Christmas...no sorry, I meant get ready for another funny lesson**!**
Before starting with the code just let me say that many of you guys are sending their homeworks or sharing ideas over different way around ! This is absolutely fantastic ! At this point I'm sure you figured out there are many ways to get to the same results and those I'm showing you in my lessons aren't always the best or shortest. Actually on some occasion I could have been more concise but I thought it is better to write more clear lines to make you better understand all the steps! you could always shorten the things up once you get more confident with the scripting.
So that said let's begin with the homework solutions!

Assignment 1:

*create four branches of nodes with 1 Transform, 1 Grade and 1 Blur each and connect their inputs to a dot and their output to the A and B inputs of a merge n*ode.

```
transforms=[]
blurs=[]
finalsel=[]
dd=nuke.createNode("Dot", inpanel = False)
dd.setSelected(False)
```

```
for k in range (1,7):
   tr = nuke.createNode("Transform", inpanel = False)
   tr.setInput(0,dd)
   transforms.append(tr)
   br = nuke.createNode("Blur", inpanel = False)
   blurs.append(br)
   finalsel.extend([blurs,transforms])

for k in blurs:
   k.setSelected(True)

mm=nuke.createNode("Merge2", inpanel = False)
mm.setSelected(False)

newlist=transforms+blurs
newlist.extend([mm,dd])

for newlist in newlist:
     newlist.setSelected(True)

for sele in nuke.selectedNodes():
   nuke.autoplace(sele)
```

Ok let's go through this script. Basically I started by creating 3 empty lists, the first 2 will be filled with transforms and blurs whereas the latter will contain both lists. In fact by using the command **extend** I'm effectively adding both the lists into another one.

Well the code looks really long and sometime the difference with a good programming skill is getting to the same result by using less lines and a neat code as we mentioned before. In order to do so some practice is required but at the same time it is important to learn more commands.
In the next example I'll show you how we can spare a few lines by using the command **nuke.selectConnectedNodes()** which basically selects all connected nodes from the currently selected node.

```
lis=[]
lis2=[]
dd=nuke.createNode("Dot")
dd['selected'].setValue(False)
for k in range (0,4):
   tr = nuke.createNode("Transform", inpanel = False)
   lis.append(tr)
   tg = nuke.createNode("Grade", inpanel = False)
   tb = nuke.createNode("Blur", inpanel = False)
   lis2.append(tb)
```

```
for i in lis:
    i.setSelected(True)
    i.setInput(0,dd)
    i.setSelected(False)

for i in lis2:
    i.setSelected(True)

nuke.createNode("Merge2", inpanel = False)
scn=nuke.selectConnectedNodes()
for placing in nuke.selectedNodes():
    nuke.autoplace(placing)
nuke.selectAll()
nuke.invertSelection()
```

Wow looks quite nice actually but be careful on using this command, it is fine if you are creating a chunk of code and nodes as we are doing but if you execute it into a big script it might select all nodes upward,which is something not desiderable.
Besides there is something new that I'm adding over the last 2 lines. Basically I'm using those lines to deselect everything. So **nuke.selectAll()** selects everything within your script whilst on the following line we are inverting our selection by invoking **nuke.invertSelection()**, which basically returns nothing selected ! ;) Good to know ! you might want to use them for something else ;)

Assignment 2:

Write a code who asks the number of cards to create, then put them all into a new group, then connect its input to a scene node. In the end connects your group's output to a scanline render.

```
lis=[]
txt = nuke.getInput( 'how many cards ?', 'type value' )
for k in range (1,int(txt)):
    tr = nuke.createNode("Card", inpanel = False)
    pos= k * (int(txt)/5)
    tr['translate'].setValue([0,0,pos])
    tr['uniform_scale'].setValue(pos)
    tr['selected'].setValue(False)
    lis.append(tr) # here I'm adding new cards to my empty list

#now here I'm selecting all cards and then I'm creating a Scene node and connect
them all to it

for i in lis:
    i.setSelected(True)
    k=nuke.selectedNodes()
```

```
b = nuke.createNode('Scene')

lis.append(b) #here I'm appending the Scene node to the list

x=0
for i in k:
    b.setInput(x,i) #here I'm connecting the scene node to all cards
    i.setSelected(True) #let's select cards and scene nodes

nmg=nuke.makeGroup(show=False) #here we group all selected nodes and deselect.
nmg.setSelected(False)

for rem in lis:
    rem['name'].value()
    rem.setSelected(True)
    nuke.delete(rem)
srender = nuke.createNode('ScanlineRender')
srender.setInput(1,nmg)
```

However We can simplify and shorten this code by using the command **selectConnectedNodes()** one time again:

```
lis=[]
txt = nuke.getInput( 'how many cards ?', 'type value' )
for k in range (1,int(txt)):
    tr = nuke.createNode("Card", inpanel = False)
    pos= k * (int(txt)/5)
    tr['translate'].setValue([0,0,pos])
    tr['uniform_scale'].setValue(pos)
    tr['selected'].setValue(False)
    lis.append(tr) #we are adding new cards to our empty list

#now we are selecting all card nodes,then we create a scene node and connect them
all to it
for i in lis:
     i.setSelected(True)
     k=nuke.selectedNodes()
b = nuke.createNode('Scene')

scn=nuke.selectConnectedNodes()
alnodes=nuke.selectedNodes()
nmg=nuke.makeGroup(show=False) #here we group all selected nodes and deselect.

for ee in alnodes:
```

```
    nuke.delete(ee)

srender = nuke.createNode('ScanlineRender', inpanel = False)
srender.setInput(1,nmg)
```

Assignment 3:

Create an array of nodes using the range function and connect their inputs to a Dot. then change the label name for all of them and assign random colors and increasing blur values.

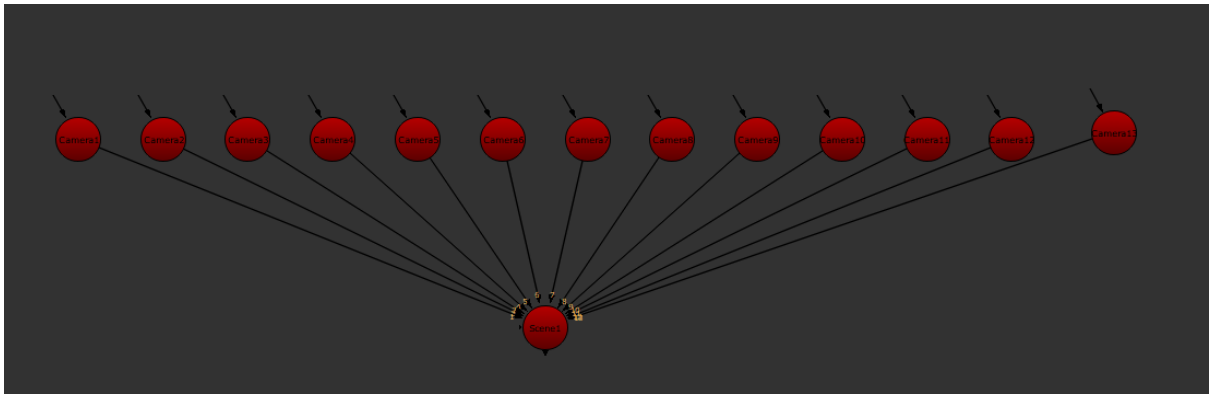
```
import random
dd=nuke.createNode('Dot')
ra=10
for ex in range(1,ra):
    bl=nuke.createNode('Blur', inpanel=False)
    bl['size'].setValue(ex * 25)
    bl['label'].setValue("Hello")
    colo=(ex*ra)*random.randrange(60000,560000,50000)
    bl['tile_color'].setValue(colo)
    bl['selected'].setValue(True)
    x=0
    bl.setInput(x,dd)
```

Obviously I've been using a range of colors' values to feed up the random function.

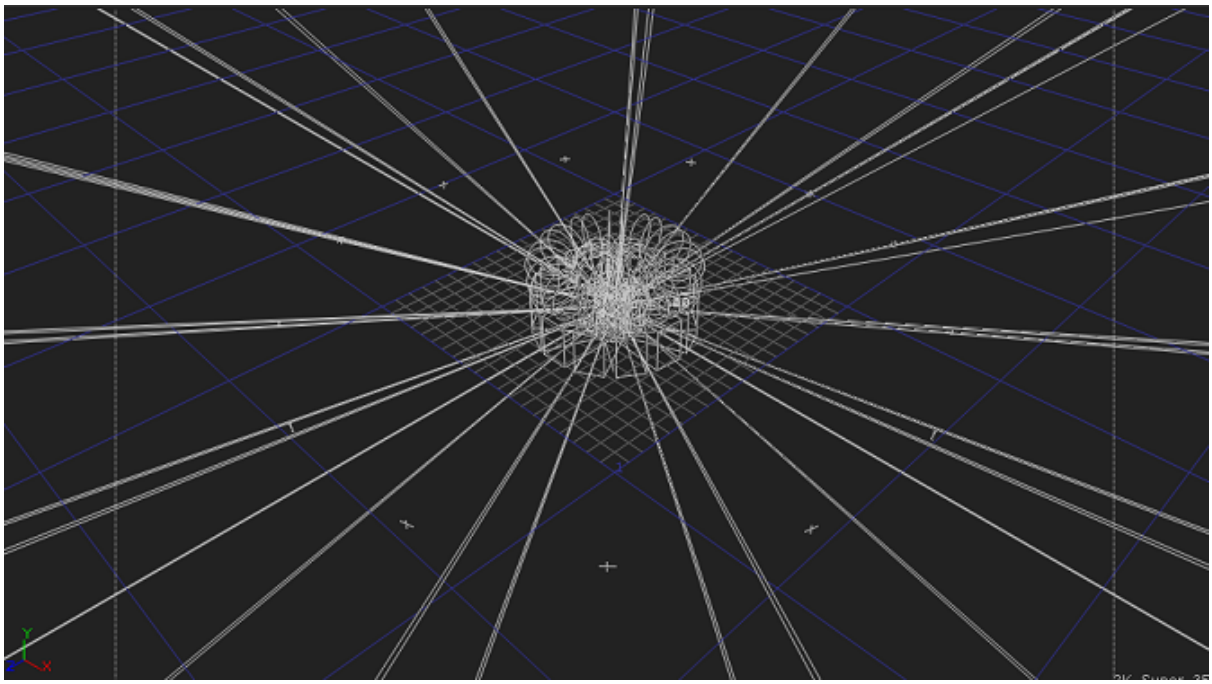
Ok!! homeworks wrapped! Now let's carry on and try something else: we want to build an array of cameras, like a proper rig that we might use for some projection work. What a buzz! The idea behind this is using the **range** command for the cameras creation itself and rotations. So take a look at the following lines and execute it:

```
amount=13
for s in range(amount):
    a= nuke.nodes.Camera()
    rot= s * 28
    a['rotate'].setValue([0,rot,0])
    a['selected'].setValue(True)
    a['haperture'].setValue(24.576)
    a['focal'].setValue(50)
    k=nuke.selectedNodes()
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)
```

..and this is what you get ..



And look at the 3D space:



Basically I have created 13 cameras with a focal length of 50mm and each one has an increasing rotation of 28° around its Y axis to cover the entire 360° panorama.
Python-wise it is similar to what we have done on previous examples; the rotation value for each camera comes out from the multiplication of 30 degrees by the increasing values of the **range** command.
Wow it works !

The value of **24.576** at the line 7 is the nuke's standard **HAperture** camera's value but I did set it again just in case; Obviously it must be changed if you are willing to use a camera with a different sensor size. …..But the question is: where I could take all those values ?
well in order to figure it out we need to dig into some math and we are going to do with the next example.
This is going to be a little more demanding as we are not only talking about Python but also photography fundamentals, so make sure you are well concentrated! Eventually it is not a bad idea on learning something more than just python :-)

Let's say we are in an ideal scenario where we took multiple photos we want to project over a big sphere to create a nice environment panorama. Clearly we assume that we are taking enough pictures to cover 360 degrees with constant rotation's steeps before each photo click.
Once we downloaded the photos we load them into Nuke, we undistort them and now we are ready to create our projection rig.
At this point we have to write a few Python lines to do the main math and automation and let python creating all the cameras we need. Obviously we have to specify the focal length and the sensor size of the camera we have been using and thus we need a specific calculation:

HFOV = (2*atan(sensorwidth/(2*focallenght))*(180/PI)

Where HFOV is the Horizontal Field of view in degrees from our camera lens.

This is really useful because by handling this value we figure out how many cameras we need to create in Nuke and we can automatically set the stepping value for the rotation of each camera. This calculation might be also an invaluable tool on our preparation for set shootings because we can figure out how many pictures we might need to take and the rotation steps for the camera. For example if we shoot with a Canon 5D MK II/III that has a sensor width of 36mm and we are using a lens of 50mm the step angle will be 39.59 (let's say 40), hence if we divide 360 by 40 it returns 9. This means that we will need 9 photos to cover the entire panorama. Later on if we set these values into a Python program it will build exactly the same rig and situation.

Nice uh ? Sooner or later I will also cook a special Nuke calculator for calculating many photographics stuff, but for now let's concentrate on our topic.

Ok now that we know how the math works, let's put it into the code:

```
from __future__ import division
import math
FL=nuke.getInput('focal length', 'type your focal length')
sen_width=nuke.getInput('Sensor Width', 'type your sensor width')
FOV=2*math.atan(float(sen_width)/(2*int(FL)))*(180/math.pi)
print math.ceil(FOV) #approximates the result to the returning value

cam_num=math.floor(360/FOV)

for s in range(int(cam_num)):
    a= nuke.nodes.Camera()
    rot= s * float(FOV)
    a['rotate'].setValue([0,rot,0])
    a['selected'].setValue(True)
    a['haperture'].setValue(float(sen_width))
    a['focal'].setValue(int(FL))
    k=nuke.selectedNodes()
```

```
b = nuke.createNode('Scene')
x=0
for i in k:
    b.setInput(x,i)

nuke.message('I have created '+ str(cam_num) +' cameras' + ' with a rotation factor of
' + str(FOV) + ' degrees')
```

What the **f..ç°?^ç°##[@]** is the first line ? ok the point is we are going to use a formula with many math operations and one of them, the simple division, seems to be very tough to run for the current Python releases because has an ambiguous meaning for numerical arguments. Python guys will fix it on future versions and Nuke will benefit too, but for now we need a way around. The most recommended procedure you can find by browsing online is the use of a specific module, promisingly called **__future__** ,from which we are importing the **division** set.
Then, just in case, let's import the "**math**" module too; this comes with other math commands for our convenience. On most cases you wouldn't need to import it at all but we are doing it here as a precaution.

Then I've created 2 user's inputs where the user is asked for typing shooting datas like the focal length end the sensor size.

tip:*Once we will be familiar with Panels we will be able to set all these infos into a single interface, but for now let's do it the way we already know.*

On line 5 we are typing our math; please note that I specified whether some values are **float** or **int**, in fact for the sensor size I can accommodate a decimal value, whereas for the focal length I prefer to allow integers values only.

Math.atan is the specific command to call the trigonometry operation for the arctangent, whilst **math.ceil** is a specific operator who rounds the number to the closer integer value.
In the end the **math.pi** returns the mathematical constant of Pi.
At line 7 we have some more math used to calculate the number of cameras and the **floor** command that rounds off to the next higher value.
So the calculation to find the camera's number is 360 divided by the FOV value.

Next step is the proper cameras creation that is going to be identical with the previous example, we just need to change the variables with those we have been creating.

Last thing, I added a **nuke.message** showing the user numeric values for the new cameras.

Ok now that you have all your cameras created you will need to create your projections for each of them and link each image on the right order, obviously you have to create the sphere you will be projecting to. This is something you could achieve with some code adds, why don't do it on your next assignments ? ;)

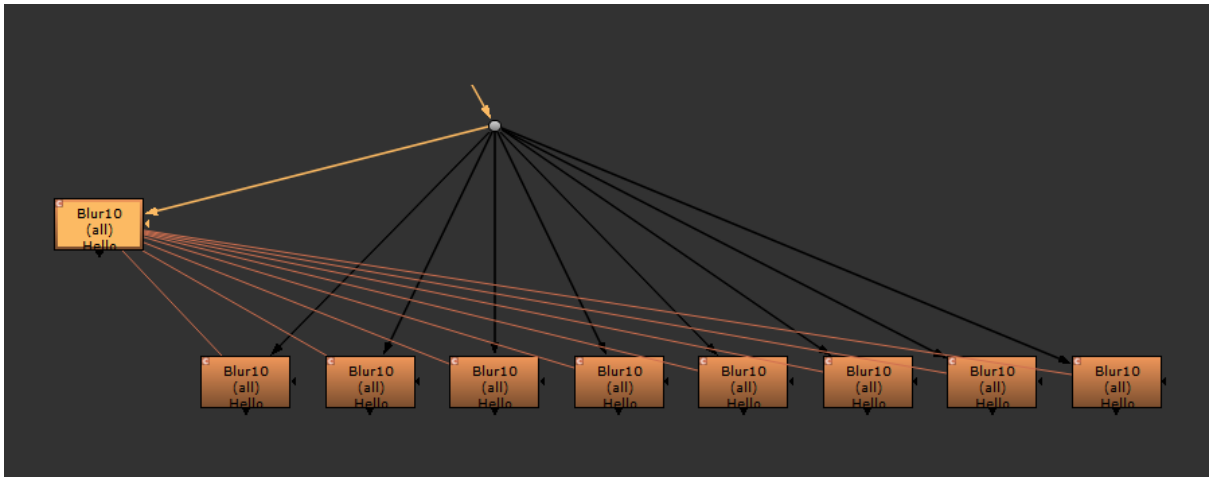
Well this script works really well in an ideal world where you have been really good on

taking very precise photos with you camera but a good idea would be to introduce some kind of correction tool in order to allow for some error or overlapping issues. But this is something you could experiment on your own, what do you reckon ? let's put it off for now.

Now let's jump on something less demanding ;)
Over one of our previous scripts with the Blurs' nodes creation we have been creating a lovely array of Blur nodes with increasing values, but now let's assume we want to make the blur nodes cloned from a first-created one, so no more increasing values, just clones. We could go for many solutions like for example setting expressions for each node or, more effectively, we could simply use the command **nuke.forceClone()** along with some other stuff we already know:

```
import random
dd=nuke.createNode('Dot')
ra=10
for ex in range(1,ra):
    bl=nuke.createNode('Blur', inpanel=False)

    bl['label'].setValue("Hello")
    bl['selected'].setValue(True)
    x=0
    bl.setInput(x,dd)

dd.setSelected(True);
nuke.selectConnectedNodes()

for x in nuke.selectedNodes():
    nuke.forceClone()
```

After manually spreading the nodes in the DAG the result should be looking like this:

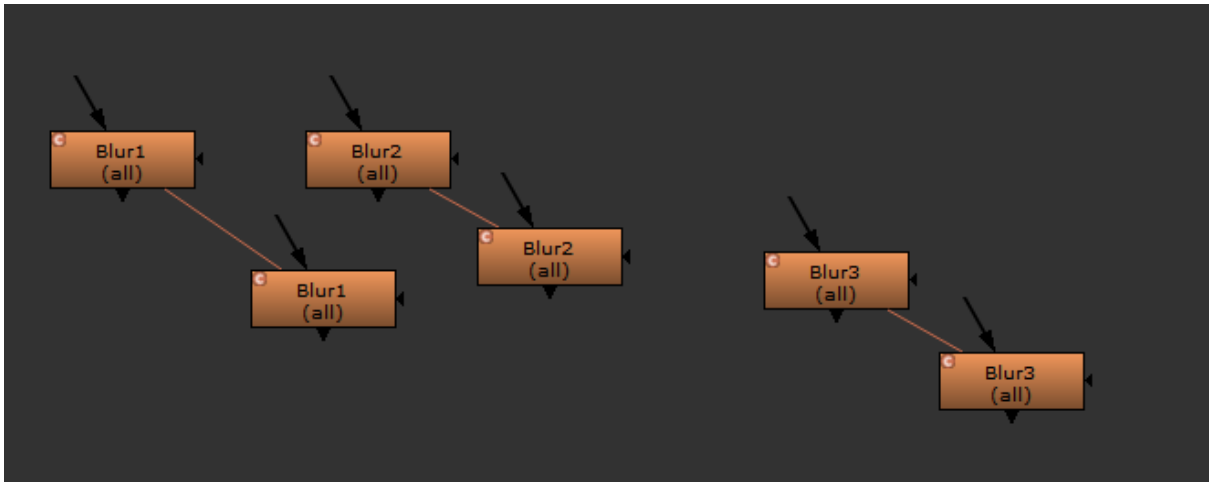


I'm sure you figured out that I removed the blur size line because since we are cloning a node it didn't make sense any more !

Sweet! But now the question is how could I possibly simply clone a node or nodes, well there is a good command called **nuke.cloneSelected()**
Let's make it work!
Just manually create 3 blur nodes in the node graph and keep them all selected. So now in our script editor we just type:

```
nuke.cloneSelected()
```

Once executed we will have a clone for each selected node as in the figure:

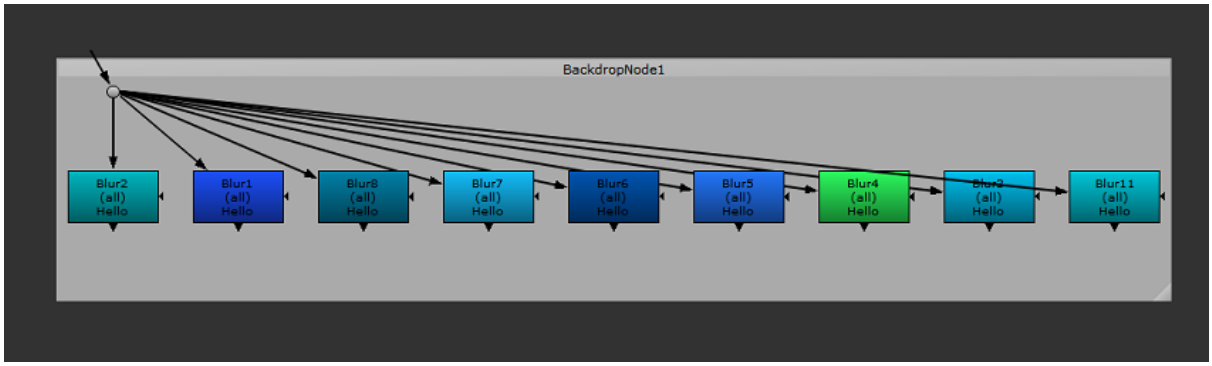


Nothing crazy but effective actually !

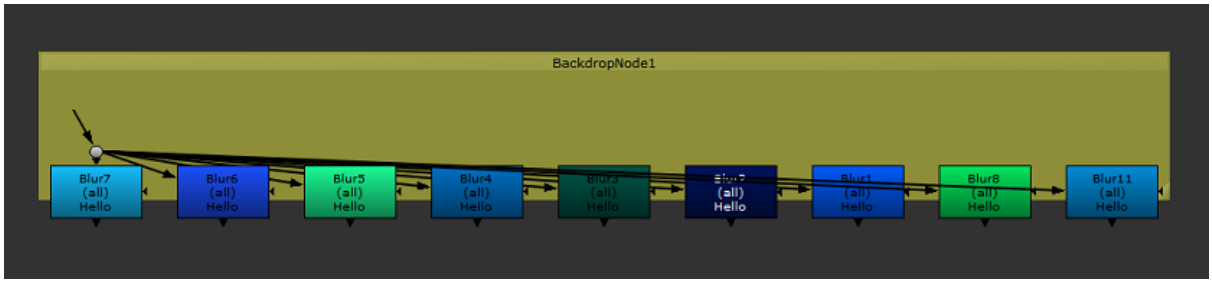
Ok now that we succeed on creating all these nodes why not placing a backdrop to frame them all? We can do it automatically with a few changes on our last script and using the last theory we have seen.

```
import random
dd=nuke.createNode('Dot')
ra=10
for ex in range(1,ra):
    bl=nuke.createNode('Blur', inpanel=False)
    bl['size'].setValue(ex * 25)
    bl['label'].setValue("Hello")
    colo=(ex*ra)*random.randrange(60000,560000,50000)
    bl['tile_color'].setValue(colo)
    x=0
    bl.setInput(x,dd)
dd.setSelected(True)
scn=nuke.selectConnectedNodes()
abd=nukescripts.autoBackdrop()
abd.knob("bdheight").setValue(211)
```

So we have something like this:



Basically over the second last line I'm using the command **nukescripts.autoBackdrop()** which in fact creates a backdrop node and places it just behind the nodes. But there's more! On the last line I'm setting the value for the backdrop height. Most of the time you don't need it, but this time if we don't our template will be looking like this:



As much as the height I could specify the width by using the command **bdwidth**, but in this case my nodes fit well and I don't need it.
At this point you might be wondering why I've also created a new variable **abd** for the **autoBackdrop**. Well the reason is I might want to vary some parameter of the backdrop, let's say its label for instance:

```
import random
dd=nuke.createNode('Dot')
ra=10
for ex in range(1,ra):
    bl=nuke.createNode('Blur', inpanel=False)
    bl['size'].setValue(ex * 25)
    bl['label'].setValue("Hello")
    colo=(ex*ra)*random.randrange(60000,560000,50000)
    bl['tile_color'].setValue(colo)
    x=0
```

```
    bl.setInput(x,dd)
dd.setSelected(True)
scn=nuke.selectConnectedNodes()
abd=nukescripts.autoBackdrop()
abd.knob("bdheight").setValue(211)
abd['label'].setValue("My blur nodes")
```

Wow I'm really having a nice time with these backdrops game!
However I'm utterly sure that during your daily comp practice you might have your lovely scripts populated with any sort of backdrops' colors. Now sometime it looks like too much like Harlequin...you know the famous italian character and mask ?

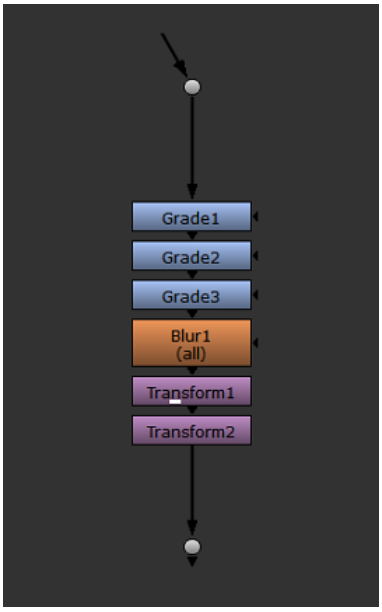
For more information on Harlequin please visit: https://en.wikipedia.org/wiki/Harlequin

I could remember more than once I opened a colleague's script and I was kind of getting tanned with the amount of bright red and yellow backdrops he had into his scripts. So once you got protected with a proper sun lotion you are ready to change all backdrops color at once to something less flashy like a standard grey, are you ?
This is very easy to do:

```
for kk in nuke.allNodes('BackdropNode'):
    kk['tile_color'].setValue(1128481791)
```

Done ! It is a very minor thing but I thought it worth the case to spend a couple of lines on it ! ...You might add into your personal tools in Nuke !
Yet...I'm really sure that you would be able to ask the user choosing a color for templates by use a **getInput** function, right ? Think about it !

Mmm very interesting indeed! Now let's try to put more things together!
Let's assume of having a script with a few nodes we want to group, like blurs, grades etc, like in the figure:

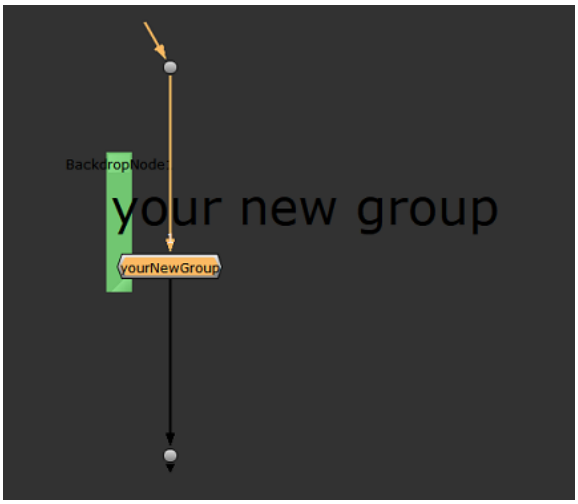


Once the grouping is done we would add a backdrop behind it and a label… so far so good ! nothing impossible for us!! Can you remember the lesson when we covered how to create groups, extracting them etc? Hopefully you do. Now for our current goal actually we want selected nodes being collapsed into a group and then kept it connected into the tree. So this is going to be different because instead of using the **makeGroup** command we will be using **nuke.collapseToGroup.**

Let's have a look:

```
sel=nuke.selectedNodes()
ciao=nuke.collapseToGroup(show=False)
ciao.setSelected(True)
ciao['name'].setValue('yourNewGroup')
abd=nukescripts.autoBackdrop()
abd['label'].setValue('your new group')
```

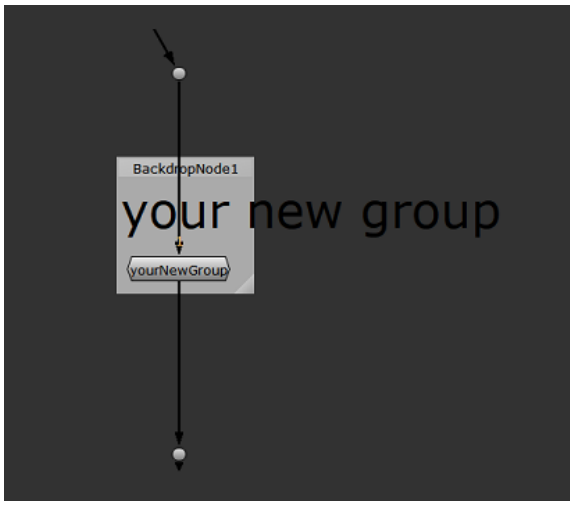
And this is what you get:



Undoubtedly interesting! The group collapsing command works like a charm but what's that ridiculous slim backdrop? It is thin like a cheese slice!

…..and it doesn't align properly behind the group node. So we really need to fix its size. So let's simply add a line with a pixel value for the width, in this case 111.

```
sel=nuke.selectedNodes()
ciao=nuke.collapseToGroup(show=False)
ciao.setSelected(True)
ciao['name'].setValue('yourNewGroup')
abd=nukescripts.autoBackdrop()
abd.knob("bdwidth").setValue(111)
abd['label'].setValue('your new group')
```

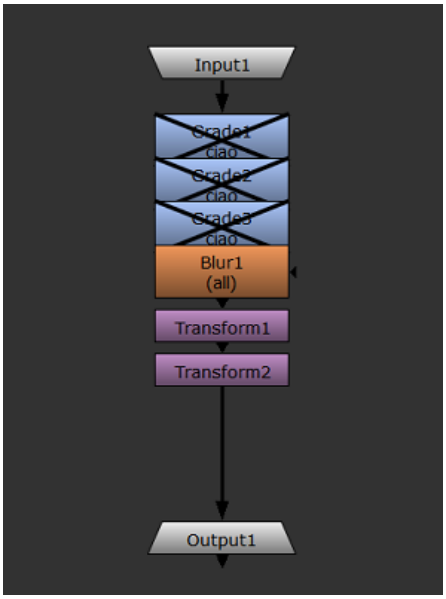


Wow!

So at this point it might be very useful to create a Python script that actually gets into a selected group and do something, for instance adding labels to grade nodes only and eventually disabling them. In order to do it we need to learn how to get into groups first. So after the previous lines of code type the following and execute it.

```
groupNode = nuke.toNode('yourNewGroup')
groupNode.begin()

for i in nuke.allNodes('Grade'):
    i.knob('label').setValue('ciao'); i.knob('disable').setValue(1)

groupNode.end()
```

As evident with these lines the commands **begins** and **end** are those responsible for showing us into the group domain. Once got in we can use all the syntax we already know to perform any kind of operation.
So now if you peek inside your group it should be looking something like this:

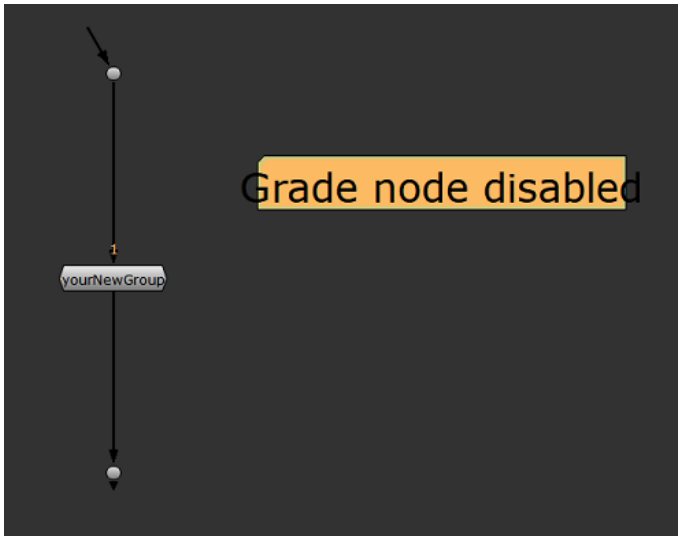


Now try to experiment on your own by adding further operations and commands! Sounds nice, doesn't it ?

Now on our previous example we have seen how to build and connect a group and we stuck a backdrop just behind it, so what if instead of doing it we place a stickyNote to its right with a description within? Like a short description on what's going on into the group ?
Well in order to do it we should retrieve group's position in the DAG space otherwise the stickiNote itself might appear near our granny's chest of drawers in Kentucky. It is quite easy actually as we have a specific command for it, let's take a look at this code:

```
sel=nuke.selectedNodes()
ciao=nuke.collapseToGroup(show=False)
ciao.setSelected(True)
ciao['name'].setValue('yourNewGroup')
groupNode = nuke.toNode('yourNewGroup')
groupNode.begin()

for i in nuke.allNodes('Grade'):
```

```
    i.knob('label').setValue('ciao'); i.knob('disable').setValue(1)

groupNode.end()

grpos=groupNode['xpos'].getValue()
stNode=nuke.createNode("StickyNote", inpanel = False)
stNode['label'].setValue('Grade node disabled')
stNode['note_font_size'].setValue(33)
stNode['xpos'].setValue(grpos+150)
```

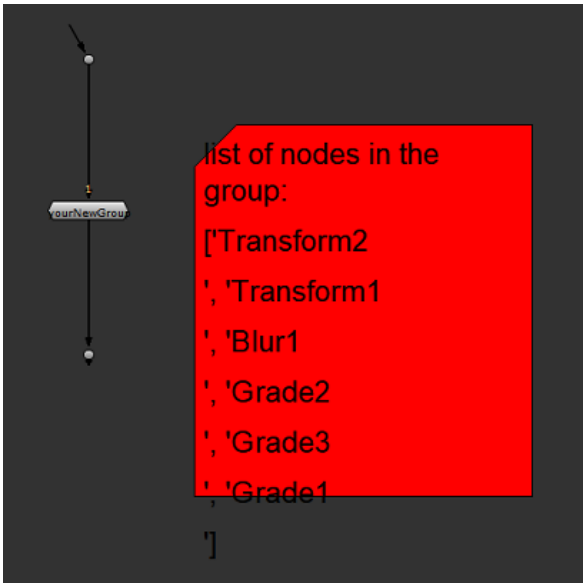


Over the first highlighted line I get the group's x value by using the **xpos** command, whereas over the last line I'm just assigning the value to the stickyNote and I'm adding a further 150 pixels just to make sure it sits properly on the right side of the group.
Awesome ! Your granny is safe!
Another good idea would be writing something else into the stickyNote, like a list of the nodes that have been collapsed into the group! What a nice idea ! let's do it !
First of all let's select the nodes we want to callapse in a group, then execute this:

```
sel=nuke.selectedNodes()
lis=[]
for s in sel:
  n = s['name'].value()
  lis.append(n+'<p>')

ciao=nuke.collapseToGroup(show=False)
ciao.setSelected(True)
ciao['name'].setValue('yourNewGroup')
groupNode = nuke.toNode('yourNewGroup')
groupNode.begin()

for i in nuke.allNodes('Grade'):
```

```
    i.knob('label').setValue('ciao'); i.knob('disable').setValue(1)

groupNode.end()

grpos=groupNode['xpos'].getValue()
stNode=nuke.createNode("StickyNote", inpanel = False)
stNode['label'].setValue('list of nodes in the group:'+'<p>'+str(lis))
stNode['tile_color'].setValue(4278190335)
stNode['note_font_size'].setValue(33)
stNode['xpos'].setValue(grpos+150)
```

...and this is what you get !



...On future lessons we will learn how to make it more neat without any commas, quotes and square brackets ;)
Have you noticed that I'm using the html command **<p>** to start a new line? How cool is it?

Awesome ! how many things we are learning in this lesson ! so now everybody knows how to manually opening a group and see what's inside, but what if we want to do it programmatically ?

It is very easy, we just need to use a function called **nuke.showDag(n)** , so let's select our group and then execute the following:

```
for n in nuke.selectedNodes():
    print n.name()
    nuke.showDag(n)
```

Gorgeous!
Now let's leave Groups topic and let's focus some more time over stickyNotes.

We are now about to create something interesting, a stickyNote which includes a list of all read nodes file's names ! So in order to test it open up one of your nice comp scripts and run the following into the script editor:

```
lis=[]
sel = nuke.allNodes('Read')
for hh in sel:
    Ftype = hh['file'].getValue()
    Fname = os.path.basename(Ftype)+'<P>'
    lis.append(Fname)
stNode=nuke.createNode("StickyNote", inpanel = False)
stNode['label'].setValue('list of nodes in the group:'+'<P>'+str(lis))
```

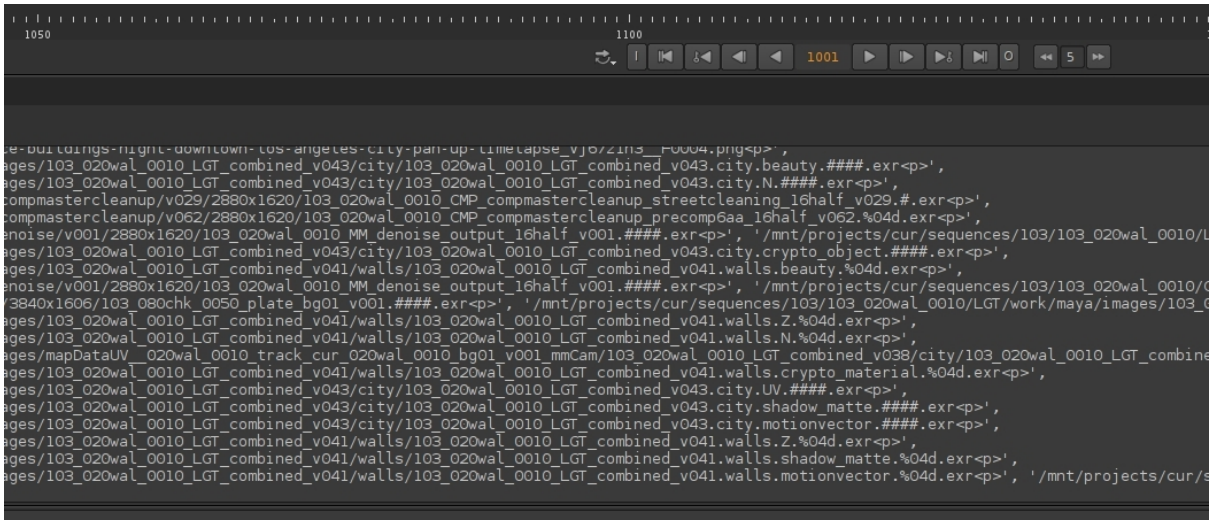
Yet in this case I've been using the html **<p>** statement to start a new line.

Well this script is quite interesting because just think at the possibilities you might have when, one day not too far away, we will introduces more advanced stuff on conditions ! Just imagine one day when you could be able to write something to compare Read nodes between scripts and doing stuff ! wow sounds exciting !

There is more! We might want to print entire paths for all our read's nodes ! For now I'm just

printing it into the script editor, but by now you perfectly know how to do something more with it !

```
lis=[]
for i in nuke.allNodes('Read'):
    nomi=nuke.filename(i)
    lis.append(nomi+'<p>')
print lis
```



And now has come the time to make a huge step ahead and going over the next big topic: Conditions!!!!!
It took 5 lessons to come up but I have done it intentionally because I wanted to make you more familiar with basic commands and loops before going any further, otherwise you could mess things up and got overwhelmed with too many notions at the same time.
By using conditions you will figure out how to simplify some of our previous scripts or make them even more effective by adding other parallel processes etc.

But as by now usual with all our lessons, let's take a break, grab some all-buttered cookies, sit back, relax and take a look at this 3D street' s art video !
...but for opening it up let's use a nuke command from our script window and type:

nuke.tcl('start', 'https://www.youtube.com/watch?v=a8tngtNgXl4**')**

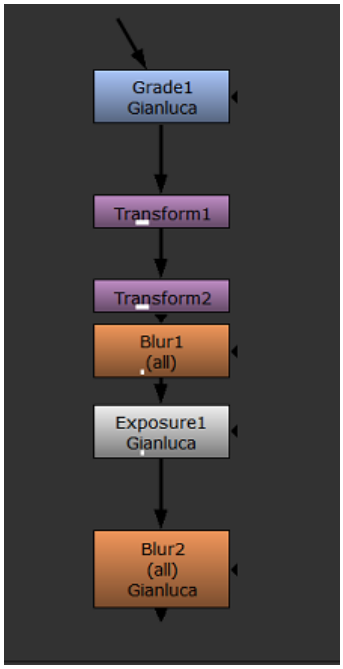
Wow! You have just learned something new !

Ok it is finally time to give it up to:

Mr. if

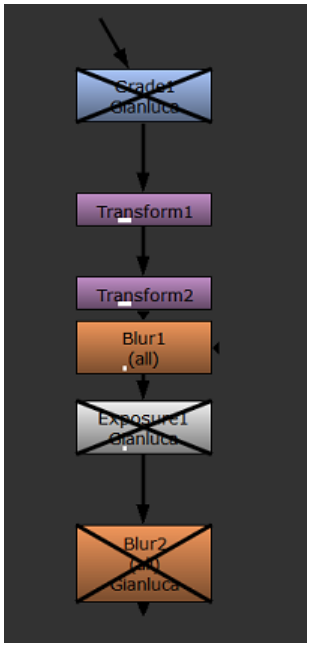
And so here we are! Well the "**if**" command and in general all condition operators are really important in all programming environments regardless of the language you are jamming on!
So building great programs and very effective pipelines or compositing tools with Python implies learning these operators very well.
You guys already know that I like to start with very easy scripting to introduce new stuff and this time won't be different!

So now let's assume we have a few nodes like grades, blurs, transforms etc. but just some of them have been labeled "Gianluca" ...



….and you just want to disable them from your script, just execute:

```
for n in nuke.allNodes():
    if n['label'].value() == 'Gianluca':
        n['disable'].setValue(True)
```



Ok first thing to notice is the indentation! As much as all other functions and commands the "**if**" must have its own indentation if you put under a "**for**" loop. As a matter of fact the "**if**" itself actually requires additional indentation when you type further commands to execute if the condition is true! and don't forget che colons at the end of it !
So the first line is obvious, the second line is where our new command is coming around, but to your by now expert eyes it shouldn't looks so weird!
Basically it is taking the value into the Label and it is creating a condition saying that if the word "Gianluca" is present, then it switches all these nodes off. On the contrary nothing will be affected.

That's amazing !! now let's try something else, let's assume we want to change the font size for all your write nodes:

```
for a in nuke.allNodes():
        if "Write" in a['name'].value():
                a['note_font_size'].setValue(60)
```

Basically on the second line we are telling Python that we are looking for nodes that have the word "Write" into their name and if it is the case, it will change their font size to 60.
Easy peasy ! try to make some experiment by using any other commands you already know!

Now here comes something interesting: let's assume we want to set our Read nodes, like the way they work when we are looking at them out of the framerange. By default it is set to "hold" but we want to turn it to "black" basically by choosing one of the dropdown menu choices. Very easy actually ! we already know how to do it, but we just want to perform this change over all read nodes in the script if the "class" of our node is a Read.

```
for a in nuke.allNodes():
    if a.Class()=='Read':
      a['before'].setValue('black')
      a['after'].setValue('black')
```

So the command **Class** is very important as we can use it to perform

One last example that might be interesting and more effective. Let's assume we have both grades and ColorCorrect nodes into our script and we want to change the gamma values for Grade nodes only, to 3.

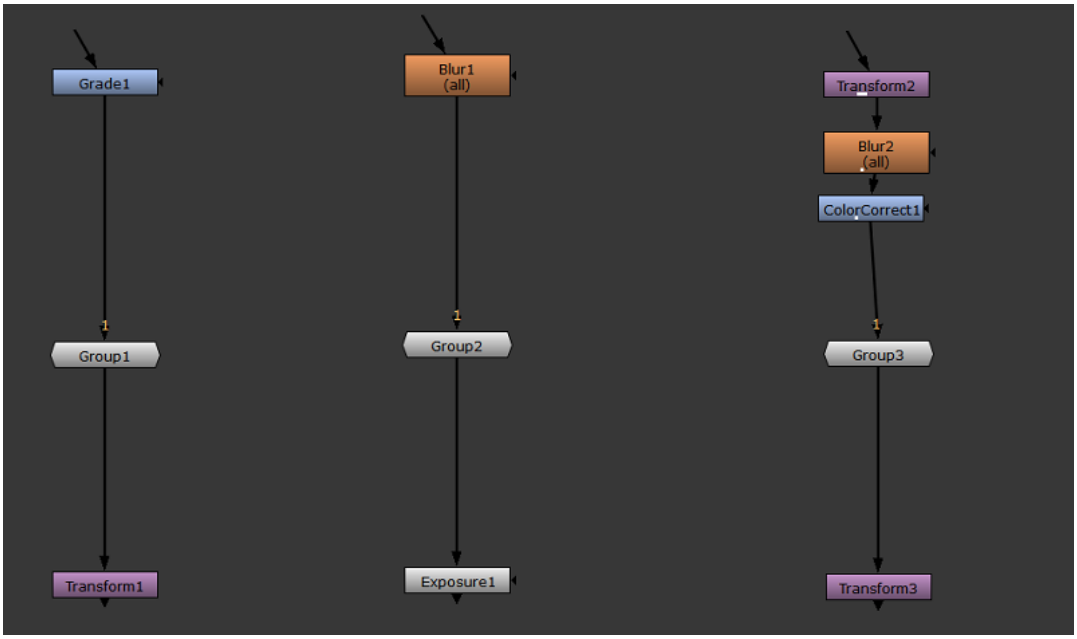
```
for a in nuke.allNodes():
    if not a.Class()=='ColorCorrect':
      a['gamma'].setValue(3)
```

Basically I can afford to call all nodes because I know that the only 2 nuke's nodes that have gamma values are the Grade and the ColorCorrect, and then I'm telling him **NOT** to act on ColorCorrect nodes, just on Grades.
Obviously you have to be careful if you think on doing something like this on other kind of nodes, it might not be safe, but it was just an example to show how a negative condition works on Python.

Ok we are not going any further with this lesson because you might get mad with all these notions at once ! ...but I will be giving you a few funny assignments so you can enjoy it during Christmas time instead of filling your stomach with tons of fat food !

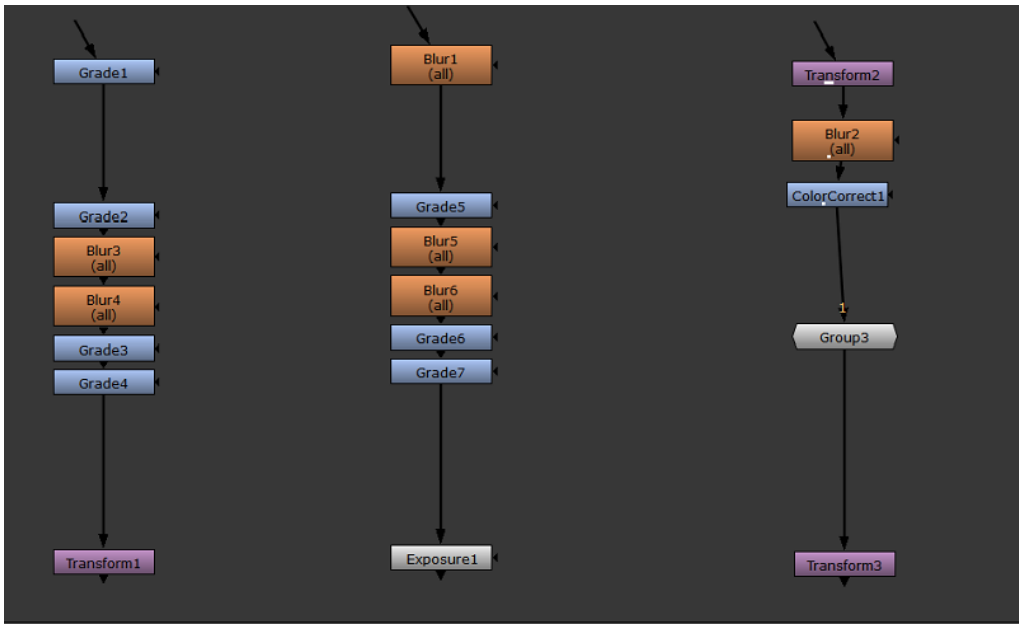
Assignment 1:

Let's assume we have a few groups into our script and we want our Python code to expand just those who has a particular class of nodes within. Let's say a Grade node.
So we could have something like have this :



There are clearly 3 groups in the image, one of them doesn't have the Grade node into it (the first from the right).

And then once the nodes are expanded it shoukd be looking like this:

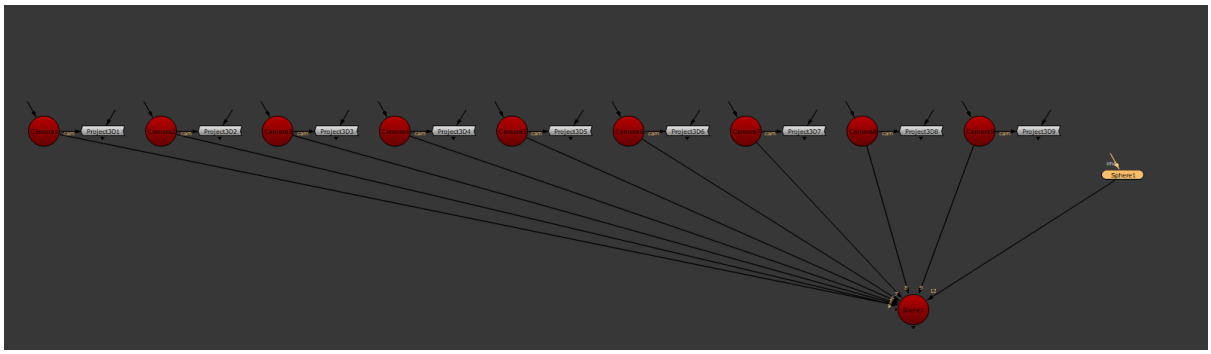


So just the groups with grades nodes will be expanded. Enjoy !

Assignment 2:

Grab the projection rig we have covered in this lesson and improve it by creating a sphere and connecting Project3D nodes to each camera, then if you have some more spare time try to move these projections' nodes more upwards so the script is more clean.

Should be looking something like this:



Assignment 3:

Let's assume we have a big script where for some reason we have been using a few strong defocuses or something similar like a PGBokeh. We want to create a script asking the user what's the maximum amount of defocusing value after which it will triggering the $Gui disable expression for all those nodes. This is sometimes useful when you are dealing with someone else's shot and they told you to make it smoother.

And that's all folks ! enjoy the Christmas the new year's celebrations and don't forget a good wish for you friend Gianluca !

HAPPY CHRISTMAS GUYS !!

Created in Maya-rendered in Arnold