```
import maya.cmds as mc
from functools import partial
from operator import itemgetter

'''
14_07_14
do I need to order nodeList????
- set keys based on vis keys as well as start and duration

'''

clm1 = 250
clm2 =100
clm3 = 100
separation = 15
sepStyle2 = 'none'

def orderList(nodeList):
    extendedNodeList = []

    for node in nodeList:
        nodenameList = node.split('_')

        if nodenameList[-1] == '':
            nodeNumber = 0
        else:
            nodeNumber = int(nodenameList[-1])

        extendedNodeList.append([node,nodeNumber])

    newExtendedNodeList = sorted(extendedNodeList, key=itemgetter(1))
    print 'newExtendedNodeList == ', newExtendedNodeList

    newNodeList = []
    for item in newExtendedNodeList:
        newNodeList.append(item[0])

    print 'newNodeList == ', newNodeList
    return newNodeList

def getMaterialFromSG(shadingGroupList):
    materialList = []
    for shadingGroupName in shadingGroupList:
        shadingGroup = shadingGroupName
        material =
mc.listConnections('{shadingGroup}.surfaceShader'.format(shadingGroup=sha
dingGroup))[0]
        materialList.append(material)

    return materialList

class KeyShaderTransparency(object):

    def __init__(self, parentLayout, workOnTransformNodeStatus):
        self.parentLayout=parentLayout
        self.workOnTransformNodeStatus = workOnTransformNodeStatus

        self.workOnChildNodes = True
```

```
                ### dictionary to store material and associated ramp
        self.rampDict = dict()

        self.buildUI()

    def buildUI(self):
        print 'new PPShaderCtrls'
        separator = mc.separator(style =sepStyle2, w=300, h=separation,
parent=self.parentLayout)
        mc.text(' --- shader controls', parent=self.parentLayout)

        self.radioCollection = mc.radioCollection()
        RBlayout = mc.rowLayout(numberOfColumns=2,
parent=self.parentLayout)
        self.childrenRB = mc.radioButton(label='Work on child nodes',
parent=RBlayout, select=True,
onCommand=(partial(self.setWorkOnChildNodeStatus, True)))
        self.selectedNodesRB = mc.radioButton(label='Work on selected
nodes', parent=RBlayout,
onCommand=(partial(self.setWorkOnChildNodeStatus, False)))

        setupLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.text(label = ' -- setup shader network', width=clm1,
parent=setupLayout)
        mc.button(label = 'Setup network', width=clm2,
command=self.setupShaderNetwork, backgroundColor=(.6, .45, .45),
parent=setupLayout)

        ### trail layout
        trailLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.text('Start')
        self.startTF = mc.textField('startTF', parent=trailLayout,
width=25, text='10')
        mc.text('     Duration')
        self.durationTF = mc.textField('durationTF', parent=trailLayout,
width=25, text='10')

        transLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.button(label = 'Key shader transparency', width=clm1,
command=self.setTransKeys,\
            parent=transLayout)
        mc.button(label = 'Delete keys', width=clm2,
command=self.deleteTransKeys,\
            parent=transLayout)

    def getNodeList(self):
        ## returns list of transforms or of shapes
        ############ should this be ordered
??????????????????????????????
        ## BUT do I ever need a list of transforms???????????????????
        if self.workOnChildNodes == True:
            groupNode = mc.ls(selection=True)
            if self.workOnTransformNodeStatus == True:
                tranformNodeList = mc.listRelatives(groupNode,
children=True, type = 'transform')
```

```
                nodeList = tranformNodeList
                print 'nodeList is list of transforms and == ', nodeList

            elif self.workOnTransformNodeStatus == False:
                shapeNodeList = mc.listRelatives(groupNode,
children=True, type = 'shape')
                nodeList = shapeNodeList
                print 'nodeList is list of shape nodes and == ', nodeList

        else:
            nodeList = mc.ls(selection=True)
            print 'nodeList == ', nodeList

        return nodeList

    def getSwitchListFromNode(self, shapeNode):
        ############ does this need to get a list of switches? one for
each material
        switchList = []
        #get the SG on the mesh
        # shadingGroup = mc.listConnections(shapeNode,
type='shadingEngine')
        # get list of materials on the shape
        materialList = self.getMaterialList(shapeNode)
        print ' in getSwitchListFromNode() materialList ==  ',
materialList
        if materialList:
            for material in materialList:

                materialType = mc.nodeType(material)

                #####  get teh switch node     #############if its a
surface shader
                if materialType == 'surfaceShader':
                    ### check if it has anything attached to transparency
R
                    if
mc.connectionInfo('{material}.outTransparencyR'.format(material=material)
, isDestination=True):
                        # print 'yep'
                        switchNode =
mc.listConnections('{material}.outTransparencyR'.format(material=material
))[0]
                        print switchNode, ' connected to transparency R'

                    elif
mc.connectionInfo('{material}.outTransparencyG'.format(material=material)
, isDestination=True):
                        switchNode =
mc.listConnections('{material}.outTransparencyG'.format(material=material
))[0]
                        print switchNode, ' connected to
outTransparencyG'

                    elif
mc.connectionInfo('{material}.outTransparencyB'.format(material=material)
, isDestination=True):
```

```
                        switchNode =
mc.listConnections('{material}.outTransparencyB'.format(material=material
))[0]
                        print switchNode, ' connected to
outTransparencyB'
                    else:
                        switchNode = None
                        print '>>>> No single switch node connected to SS
material  ', material

                                                    # #if its a surface
shader
                if materialType != 'surfaceShader':
                    ### check if it has anything attached to transparency
R
                    if
mc.connectionInfo('{material}.transparencyR'.format(material=material),
isDestination=True):
                        # print 'yep'
                        switchNode =
mc.listConnections('{material}.transparencyR'.format(material=material))[
0]
                        print switchNode, ' connected to transparency R'

                    elif
mc.connectionInfo('{material}.transparencyG'.format(material=material),
isDestination=True):
                        switchNode =
mc.listConnections('{material}.transparencyG'.format(material=material))[
0]
                        print switchNode, ' connected to transparency G'

                    elif
mc.connectionInfo('{material}.transparencyB'.format(material=material),
isDestination=True):
                        switchNode =
mc.listConnections('{material}.transparencyB'.format(material=material))[
0]
                        print switchNode, ' connected to transparency B'
                    else:
                        switchNode = None
                        print '>>>> No single switch node connected to
', material

                switchList.append(switchNode)
        return switchList

    def setTransKeys(self, *args):
        ######### gina what if nodelist returns a list of
transforms???????????????????
        unsortedNodeList = self.getNodeList()
        ########## sort the list
        nodeList = orderList(unsortedNodeList)
        ########### NB I need to get the material

        index = 0
        for node in nodeList:
            # get the shape node
```

```
            # node = nodeList[index]
            shapeNode = self.getShapeFromNode(node)

            switchNodes = self.getSwitchListFromNode(shapeNode)
            ########### NB I need to get the material

            nodeName = node
            for switchNode in switchNodes:
                self.keyTransAttrs(switchNode, nodeName, index)

            index += 1


    def deleteTransKeys(self, *args):
        nodeList = self.getNodeList()

        for index in range(len(nodeList)):
            # get the shape node
            node = nodeList[index]
            shapeNode = self.getShapeFromNode(node)

            switchNodes = self.getSwitchListFromNode(shapeNode)

            for switchNode in switchNodes:
                ##cut the keys

mc.cutKey('{switchNode}.mesh{meshNumber}_trans'.format(switchNode=switchN
ode, meshNumber=index))
                ##set node vis to 1

mc.setAttr('{switchNode}.mesh{meshNumber}_trans'.format(switchNode=switch
Node, meshNumber=index), 0)

    def getShapeFromNode(self, node):
        if self.workOnTransformNodeStatus == True:
            shapeNode = mc.listRelatives(node, children=True, type =
'shape')[0]
        else:
            shapeNode = node

        return shapeNode

    def getMaterialList(self, shapeNode):
        #get the SG's on the mesh
        shadingGroupList = mc.listConnections(shapeNode,
type='shadingEngine')
        if shadingGroupList != None:
            #remove duplicates
            shadingGroups = list(set(shadingGroupList))
            ## get the materials from teh shading groups
            materials = getMaterialFromSG(shadingGroupList)
            #removeDuplicates
            materialList  = list(set(materials))
            print '>>> materialList  ==  ',  materialList

            return materialList

    def getSingleSwitch(self, material):
```

```
############# gina is this only getting it for a surface shader????
        #### find out whether the shader is a surface shader
        shaderType = mc.nodeType(material)

        if shaderType != 'surfaceShader':
            #         # ## connect switch to material if its not already
connected
            # if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyR'.format(material=material)):
            #     # print
'{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyR'.format(material=material)
            #
mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyR'.format(material=material), force=True)

            # if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyG'.format(material=material)):
            #
mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyG'.format(material=material), force=True)

            # if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyB'.format(material=material)):
            #
mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyB'.format(material=material), force=True)
                        ### check for single switch and create it of
there is none
            if
mc.connectionInfo('{material}.transparencyR'.format(material=material),
isDestination=True):
                singleSwitch =
mc.listConnections('{material}.transparencyR'.format(material=material))[
0]

            elif
mc.connectionInfo('{material}.transparencyG'.format(material=material),
isDestination=True):
                singleSwitch =
mc.listConnections('{material}.transparencyG'.format(material=material))[
0]
                # print singleSwitch, ' connected to transparencyG'

            elif
mc.connectionInfo('{material}.transparencyB'.format(material=material),
isDestination=True):
                singleSwitch =
mc.listConnections('{material}.transparencyB'.format(material=material))[
0]

                print singleSwitch, ' [in getSingleSwitch()]  connected
to transparencyB'

            else:
```

```
                #create a single switch node
                singleSwitch = mc.shadingNode('singleShadingSwitch',
asUtility=True)


        if shaderType == 'surfaceShader':
            ### check for single switch and create it of there is none
            if
mc.connectionInfo('{material}.outTransparencyR'.format(material=material)
, isDestination=True):
                singleSwitch =
mc.listConnections('{material}.outTransparencyR'.format(material=material
))[0]
                # print singleSwitch, ' connected to outTransparencyR'

            elif
mc.connectionInfo('{material}.outTransparencyG'.format(material=material)
, isDestination=True):
                singleSwitch =
mc.listConnections('{material}.outTransparencyG'.format(material=material
))[0]
                # print singleSwitch, ' connected to outTransparencyG'

            elif
mc.connectionInfo('{material}.outTransparencyB'.format(material=material)
, isDestination=True):
                singleSwitch =
mc.listConnections('{material}.outTransparencyB'.format(material=material
))[0]

                print singleSwitch, ' connected to outTransparencyB'

            else:
                #create a single switch node
                singleSwitch = mc.shadingNode('singleShadingSwitch',
asUtility=True)

        # print 'getSingleSwitch() >>> singleSwitch is ', singleSwitch
        return singleSwitch

    def getColourConnection(self, material):
        #check if node is connected to outColor if not get the colour as
a list
        if
mc.connectionInfo('{material}.outColor'.format(material=material),
isDestination=True):
            colorNode =
mc.listConnections('{material}.outColor'.format(material=material))[0]
            nodeType = mc.nodeType(colorNode)

            return colorNode, nodeType

        else:
            colourList =
mc.getAttr('{material}.outColor'.format(material=material))[0]
            colourListType = type(colourList)

            return colourList, colourListType
```

```
    def addShapeToSwitch(self, shape, switch, index):
        # print 'addShapeToSwitch() >>> switch is ', switch

        if mc.isConnected('{shape}.instObjGroups[0]'.format(shape=shape),
'{switch}.input[{i}].inShape'.format(switch=switch, i=index)):
            print '>>> shape node already added'
        else:

mc.connectAttr('{shape}.instObjGroups[0]'.format(shape=shape),
'{switch}.input[{i}].inShape'.format(switch=switch, i=index))
            print '>>> {shape} just been added to
{switch}'.format(shape=shape, switch=switch, i=index)

    def makeAndConnectBlend(self, shapeNode, material, singleSwitch,
attrName, tripleSwitch, index):
        ### make a blend node
        blendNode = mc.shadingNode('blendColors', asUtility=True, \
            name
='{shapeNode}{material}Blend'.format(shapeNode=shapeNode,material=materia
l))
        ## set colour1 to black
        mc.setAttr('{blendNode}.color1'.format(blendNode=blendNode),
0,0,0)

        #### get the dictValueRampNode from the dictionary
        colour2 = self.rampDict.get(material)
        #### get colour2Type
        colour2Type = type(colour2)

        if colour2Type == tuple:
            # set the Blend colour2 to the colour
            mc.setAttr('{blendNode}.color2'.format(blendNode=blendNode),
colour2[0], colour2[1], colour2[2])


        if colour2Type == unicode:
            # connect node to colour2
            mc.connectAttr('{colour2}.outColor'.format(colour2=colour2),
'{blendNode}.color2'.format(blendNode=blendNode), force=True)

        # connect switch node trans attr to the blender

mc.connectAttr('{singleSwitch}.{attrName}'.format(singleSwitch=singleSwit
ch, attrName=attrName),
'{blendNode}.blender'.format(blendNode=blendNode))
        ### connect blend to triple
        mc.connectAttr('{blendNode}.output'.format(blendNode=blendNode),
\

'{tripleSwitch}.input[{index}].inTriple'.format(tripleSwitch=tripleSwitch
, index=index), force=True)

        return blendNode

    def makeAndConnectTripleSwitch(self, material):
        tripleSwitch = mc.shadingNode('tripleShadingSwitch',
asUtility=True)
```

```
        ## set triple switch default Colour
        #### get the dictValueRampNode from the dictionary
        colour2 = self.rampDict.get(material)
        #### get colour2Type
        colour2Type = type(colour2)

        if colour2Type == tuple:
            # set the Blend colour2 to the colour

mc.setAttr('{tripleSwitch}.default'.format(tripleSwitch=tripleSwitch),
colour2[0], colour2[1], colour2[2])

        if colour2Type == unicode:
            # connect node to colour2
            mc.connectAttr('{colour2}.outColor'.format(colour2=colour2),
'{tripleSwitch}.default'.format(tripleSwitch=tripleSwitch), force=True)

        return tripleSwitch

    def setupShaderNetwork(self, *args):
        ### gina sort the node list ##########################
        unsortedNodeList = self.getNodeList()
        nodeList = orderList(unsortedNodeList)

        index = 0
        for node in nodeList:
            # get the shape node and list of material on the node
            shapeNode = self.getShapeFromNode(node)
            # print '>>> shapeNode is ', shapeNode
            # get list of materials on the shape
            materialList = self.getMaterialList(shapeNode)
            # print index, 'time though nodeList loop'
            # print '>>> materialList == ', materialList

            if materialList:
                for material in materialList:
                    # print ' >> material == ', material
                    ### set the triple switch, ramp and blend to none
                    tripleSwitch = None
                    colourList = None

                    ## get or create a single switch for this material
                    singleSwitch = self.getSingleSwitch(material)
                    # print 'doIt() >>> singleSwitch is ', singleSwitch
                    ## add attr to single switch
                    attrName =
self.createTransAttrs(switchNode=singleSwitch, nodeName=node  ,
index=index)
                    ### connectShape to singleSwitch
addShapeToSwitch(shape, switch, index)
                    self.addShapeToSwitch(shapeNode, singleSwitch, index)
                    ### connect the switch to the shader transparency
                    self.connectSwitchNodeToMaterial(material,
singleSwitch)

                    ### check if material type is a surface shader
                    materialType = mc.nodeType(material)
```

```
                    if materialType == 'surfaceShader':
                        ## get the node connected to colour; returns
list, rampLTnode, or triple
                        colourConnection =
self.getColourConnection(material)[0]
                        colourConnectionType =
self.getColourConnection(material)[1]
                        # print '>>> colourConnection == ',
colourConnection, ' >> colourConnectionType == ', colourConnectionType

                        if colourConnectionType == 'tripleShadingSwitch':
                            tripleSwitch = colourConnection
                        elif colourConnectionType == 'tuple':
                            colourList = colourConnection
                            # add colourList to the dictionary
                            self.rampDict[material] = colourList
                        else:
                            colourRamp = colourConnection
                            # add colourRamp to the dictionary
                            self.rampDict[material] = colourRamp

                        ### create a tripleSwitch if there is not one
                        if tripleSwitch == None:
                            tripleSwitch =
self.makeAndConnectTripleSwitch(material)
                            ## connect the switch node to teh shader
colour
                            self.connectSwitchNodeToMaterial(material,
tripleSwitch)

                        ## connectShape to tripleSwitch
                        self.addShapeToSwitch(shapeNode, tripleSwitch,
index)

                        #### make blend, and connect to triple switch,
etc
                        blendNode = self.makeAndConnectBlend(shapeNode,
material, singleSwitch, attrName, tripleSwitch, index)

            index += 1
        # print '>>>> rampDict >>>> ', self.rampDict

    def createTransAttrs(self, switchNode, nodeName, index):
        #### this adds an attribute to teh switch node to then key the
vis
        switchNode = switchNode
        meshNumber = index

        attrName = 'mesh{meshNumber}_trans'.format(meshNumber=meshNumber)

        #check if node has the attr
        if mc.attributeQuery(attrName, node=switchNode, exists=True) ==
True:
            print 'attr exists'
        else:
            mc.addAttr(switchNode, longName=attrName,
attributeType='float', keyable=True, maxValue=1.0, minValue=0.0)
```

```
        #### connect the attr
        if not
mc.isConnected('{switchNode}.{attrName}'.format(switchNode=switchNode,
attrName=attrName), \

'{switchNode}.input[{i}].inSingle'.format(switchNode=switchNode,
i=meshNumber)):

mc.connectAttr('{switchNode}.{attrName}'.format(switchNode=switchNode,
attrName=attrName), \

'{switchNode}.input[{i}].inSingle'.format(switchNode=switchNode,
i=meshNumber))

        return attrName

    def keyTransAttrs(self, switchNode, nodeName, index):
        nodeName = nodeName
        meshNumber = index
        attrName = 'mesh{meshNumber}_trans'.format(meshNumber=meshNumber)

        onKey, offKey = self.getTransFrameNumbers(nodeName)
        ### key trans attrs
        mc.setKeyframe(
"{switchNode}.{attrName}".format(switchNode=switchNode,
attrName=attrName), time=onKey, value=0)
        mc.setKeyframe(
"{switchNode}.{attrName}".format(switchNode=switchNode,
attrName=attrName), time=offKey, value=1)


    def getTransFrameNumbers(self, nodeName):
        clipToVisOffTime = True

        ######### these frame numbers should be based on Vis keys on the
mesh
        start = int(mc.textField(self.startTF, query=True, text=True))
        duration = int(mc.textField(self.durationTF, query=True,
text=True))

        ### get a list of the vis keys on the node
        visKeys = mc.keyframe('{nodeName}.v'.format(nodeName=nodeName),
query=True)

        index = 0
        for keyTime in visKeys:
            value =  (mc.keyframe('{nodeName}'.format(nodeName=nodeName),
at='v',t=(keyTime, keyTime), q=True, eval=True))[0]
            # print 'value = ', value
            if value == 1.0:
                visKeyOnTime = keyTime
                visKeyOffTime = visKeys[index+1]
            index += 1

        # ## set blend keyframes
        if visKeyOffTime != False:
            # blendKey0 = visKeyOffTime - (blendDuration + 1)
            key1 = visKeyOnTime + start
```

```
        #
mc.setKeyframe('{blendNode}.{blendTarget}'.format(blendNode=blendNode,
blendTarget=blendTarget),\
        #         time = blendKey0, value = 0)

            if clipToVisOffTime == True:
                print 'clipToVisOffTime == ', clipToVisOffTime

                if int(key1 + duration) > int(visKeyOffTime):
                    print '>>>> clipping duration'
                    key2 = visKeyOffTime
                else:
                    key2 = key1 + duration
            else:
                key2 = key1 + duration

        print 'key1, key2 ', key1, key2
        return key1, key2

    def setWorkOnChildNodeStatus(self, status, *args):
        self.workOnChildNodes = status


    def connectSwitchNodeToMaterial(self, material, switchNode):
        # shapeNode = shapeNode
        material = material
        switchNode = switchNode
        switchNodeType = mc.nodeType(switchNode)

        #### find out whether the shader is a surface shader
        shaderType = mc.nodeType(material)

        if shaderType != 'surfaceShader':
                    # ## connect switch to material if its not already
connected
            if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyR'.format(material=material)):
                # print
'{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyR'.format(material=material)

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyR'.format(material=material), force=True)

            if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyG'.format(material=material)):

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyG'.format(material=material), force=True)

            if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyB'.format(material=material)):
```

```
mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.transparencyB'.format(material=material), force=True)

        if shaderType == 'surfaceShader':
            if switchNodeType == 'singleShadingSwitch':
                # ## connect switch to material if its not already
connected
                if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparencyR'.format(material=material)):
                    # print 'is not connected'

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparencyR'.format(material=material), force=True)
                    # print 'NOW tr connected'

                if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparencyG'.format(material=material)):
                    # print 'is not connected'

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparencyG'.format(material=material), force=True)

                if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparencyB'.format(material=material)):
                    # print 'is not connected'

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparencyB'.format(material=material), force=True)

            if switchNodeType == 'tripleShadingSwitch':
                #if its a triple connect it to the colour
                if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outColor'.format(material=material)):
                    # print '{switchNode}.output  is NOT
connected'format(switchNode=switchNode)

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outColor'.format(material=material), force=True)

        # return switchNode


class TextureCntrls(object):

    def __init__(self, parentLayout):
        self.parentLayout = parentLayout


        self.buildUI()

    def buildUI(self):
        # separation = 15
```

```
        separator = mc.separator(style =sepStyle2, w=300, h=separation,
parent=self.parentLayout)
        mc.text(' --- texture (ramp) controls', parent=self.parentLayout)
        row1 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)]  , parent=self.parentLayout)
        mc.text('project texture of selected SS :  ', align='right',
parent=row1, w=clm1)
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        addAttrsButton = mc.button(label = 'add projector',
command=self.addProjector, parent=row1, w=clm2)

        row2 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)]  , parent=self.parentLayout)
        mc.text('set ramp type on selected SS :  ', align='right',
parent=row2, w=clm1)
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        addAttrsButton = mc.button(label = 'set ramp type',
command=self.setRampType, parent=row2, w=clm2)

    def addProjector(self, *args):
        ### get the shader
        shaderList = mc.ls(selection=True, materials=True)
        ### get the base ramp
        for shader in shaderList:
            ## get the baseRamp node attached to color
            baseRampList =
mc.listConnections("{shader}.outColor".format(shader=shader),
type='ramp')
            print 'shader is ', shader
            print 'baseRampList is ', baseRampList
            if baseRampList != None:
                baseRamp = baseRampList[0]

                ### make 2d placement for ramp
                placement2D = mc.shadingNode('place2dTexture',
asUtility=True)
                #hook them up

mc.connectAttr('{placement2D}.outUV'.format(placement2D=placement2D), \
                    '{baseRamp}.uvCoord'.format(baseRamp=baseRamp),
force=True)

mc.connectAttr('{placement2D}.outUV'.format(placement2D=placement2D),
                 '{baseRamp}.uvFilterSize'.format(baseRamp=baseRamp),
force=True)

                #make a projector
                projector = mc.shadingNode('projection', asUtility=True)
                ## make a placement for projector
                placement3D = mc.shadingNode('place3dTexture',
asUtility=True)
                #hook them up

mc.connectAttr('{placement3D}.worldInverseMatrix[0]'.format(placement3D=p
lacement3D), \
```

```
'{projector}.placementMatrix'.format(projector=projector), force=True)

                ##connect ramp to projector

mc.connectAttr('{baseRamp}.outColor'.format(baseRamp=baseRamp), \
                 '{projector}.image'.format(projector=projector),
force=True)

                ### connect image to SS outColour

mc.connectAttr('{projector}.image'.format(projector=projector), \
                    '{shader}.outColor'.format(shader=shader),
force=True)


            else:
                print 'cant find base ramp'


    def setRampType(self, *args):
        shaderList = mc.ls(selection=True, materials=True)
        ### get the base ramp
        for shader in shaderList:
            ## get the baseRamp node attached to color
            projectorList =
mc.listConnections("{shader}.outColor".format(shader=shader),
type='projection')
            print 'shader is ', shader
            print 'projectorList is ', projectorList


            if projectorList != None:
                projector = projectorList[0]

                baseRampList =
mc.listConnections("{projector}.image".format(projector=projector),
type='ramp')
                if baseRampList != None:
                    baseRamp = baseRampList[0]

                    ###set ramp type to; Vramp = 0,  tartan = 8,
circular=4

mc.setAttr("{baseRamp}.type".format(baseRamp=baseRamp), 0)
        ###set ramp interp to linear
                    #
mc.setAttr("{baseRamp}.interpolation".format(baseRamp=baseRamp), 1)
        ###set ramp interp to exponential down
                    #
mc.setAttr("{baseRamp}.interpolation".format(baseRamp=baseRamp), 3)

class ColourTweakCntrls(object):

    def __init__(self, parentLayout):
        self.parentLayout = parentLayout

        self.buildUI()
```

```
    def buildUI(self):
        separation = 15

        separator = mc.separator(style =sepStyle2, w=300, h=separation,
parent=self.parentLayout)
        mc.text(' --- colour tweak controls', parent=self.parentLayout)

        row1 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Add Tweak attr to selected SurfaceShader :  ',
align='right', parent=row1, w=clm1)
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        addAttrsButton = mc.button(label = 'add tweak attrs',
command=self.addTweakAttrs, parent=row1, w=clm2)

    def addTweakAttrs(self, *args):
        # print "adding controls for cycling through the colour tweaks"
        #get the SScell shaders
        shaderList = mc.ls(selection=True, materials=True)

        for shader in shaderList:

            ## get the ramp node attached to color
            baseRampList =
mc.listConnections("{shader}.outColor".format(shader=shader),
type='ramp')
            print 'shader is ', shader
            print 'baseRampList is ', baseRampList

            if baseRampList != None:
                baseRamp = baseRampList[0]
                #### get the numEntries
                numEntries =
mc.getAttr('{baseRamp}.colorEntryList'.format(baseRamp=baseRamp),
size=True)
                print baseRamp, numEntries

                #### make sure that there are more than 1 colour entry
                if numEntries == 1:
                    print 'only one colorEntryList in ramp; no colour
tweaks'
                    return numEntries

                else:

                    #### add tweak attr to the shader
                    attrExistence = mc.attributeQuery('colourTweak',
node=shader, exists=True)
                    if attrExistence == False:
                        print 'adding
','{shader}.colourTweak".format(shader=shader)
                        mc.addAttr(shader, shortName='colourTweak',
min=0, max=(numEntries-1), storable=True, \
                            attributeType='double', keyable=True)

                    #### work on each colorEntry in turn
```

```
                    colorIndex = 1
                    for entry in range(numEntries-1):
                        print 'entry is ', entry
                        print ' colorIndex is ', colorIndex
                        print 'numEntries is ', numEntries
                        print 'shader is ', shader

                        self.createNodeNetwork(numEntries, colorIndex,
shader)
                        colorIndex += 1


    def createNodeNetwork(self, numEntries, colorIndex, shader):
        cellIndexList =  shader.split('_')
        cellIndex = cellIndexList[1]
        print 'cellIndex is ', cellIndex

        ## NB the following is complicated by the fact of the reverse
node
        numOfTweaks = numEntries - 1
        tweakRange = 1 - (0.001 * numOfTweaks)
        tweakNum = colorIndex
        endPos = 0.001 * tweakNum
        startPos = endPos + tweakRange

        # print 'numEntries is ', numEntries
        # print 'shader is ', shader
        # print 'colorIndex is ', colorIndex
        # print 'cellIndex is ', cellIndex

        # print 'numOfTweaks is ', numOfTweaks
        # print 'tweakRange is ', tweakRange
        # print 'tweakNum is ', tweakNum
        # print 'endPos is ', endPos, 'startPos is ', startPos

        setRangeMin = (numOfTweaks * 0.001) - (0.001 * tweakNum)
        setRangeMax = setRangeMin + tweakRange

        clampName = 'clampTweak{i}_SS{cellIndex}'.format(i=colorIndex,
cellIndex=cellIndex)
        clampMin = colorIndex - 1
        clampMax = colorIndex
        setRangeName =
'setRangeTweak{i}_SS{cellIndex}'.format(i=colorIndex,
cellIndex=cellIndex)

        ### create clamp node and set values
        clampNode = mc.shadingNode('clamp', asUtility=True, name =
clampName)
        mc.setAttr("{clampNode}.minR".format(clampNode=clampNode),
clampMin)
        mc.setAttr("{clampNode}.maxR".format(clampNode=clampNode),
clampMax)

        # setRangeNode = mc.shadingNode('setRange', asUtility=True, name
= setRangeName)
```

```
        #
mc.setAttr("{setRangeNode}.minX".format(setRangeNode=setRangeNode),
setRangeMin)
        #
mc.setAttr("{setRangeNode}.maxX".format(setRangeNode=setRangeNode),
setRangeMax)

        setRangeNode = mc.shadingNode('setRange', asUtility=True, name =
setRangeName)

mc.setAttr("{setRangeNode}.minX".format(setRangeNode=setRangeNode),
setRangeMin)

mc.setAttr("{setRangeNode}.maxX".format(setRangeNode=setRangeNode),
setRangeMax)
        #### old min and old max

mc.setAttr("{setRangeNode}.oldMinX".format(setRangeNode=setRangeNode),
clampMin)

mc.setAttr("{setRangeNode}.oldMaxX".format(setRangeNode=setRangeNode),
clampMax)




        ##create reverse node
        reverseNode = mc.shadingNode('reverse', asUtility=True)

        ###### connect Nodes
        attrExistence = mc.attributeQuery('colourTweak', node=shader,
exists=True)
        if attrExistence == True:

            # shader into clamp
            mc.connectAttr('{shader}.colourTweak'.format(shader=shader),\
                '{clampNode}.input.inputR'.format(clampNode=clampNode),
force=True)


mc.connectAttr('{clampNode}.output.outputR'.format(clampNode=clampNode),
\

'{setRangeNode}.value.valueX'.format(setRangeNode=setRangeNode),
force=True)


mc.connectAttr('{setRangeNode}.outValue.outValueX'.format(setRangeNode=se
tRangeNode),\

'{reverseNode}.input.inputX'.format(reverseNode=reverseNode), force=True)

            #reverse to pos

mc.connectAttr('{reverseNode}.output.outputX'.format(reverseNode=reverseN
ode),\
```

```
'rampCell_{cellIndex}.colorEntryList[{colorIndex}].position'\
                .format(cellIndex=cellIndex, colorIndex=colorIndex),
force=True)

class SetCameraBGcolour(object):
    # sets camera bg to ramp texture if it exists
    def __init__(self, parentLayout):
        self.parentLayout = parentLayout
        self.buildUI()

    def buildUI(self):
        # separation = 15

        separator = mc.separator(style =sepStyle2, w=300, h=separation,
parent=self.parentLayout)
        mc.text(' --- set camera background colour',
parent=self.parentLayout)

        row1 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Set camera BG to viewport BG colour :  ', align='right',
parent=row1, w=clm1)
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        setCamBGButton = mc.button(label = 'set cam BG',
command=self.setcamBG, parent=row1, w=clm2)

    def setcamBG(self, *args):
        ###check that ramp exists make it if it doesnt
        viewportBGramp = 'viewportBGcolour'

        if mc.objExists(viewportBGramp) == False:
            viewportBGramp = mc.shadingNode("ramp",asTexture=True, name =
'viewportBGcolour')
                    #### if theres more than two entries delete the third
            numEntries =
mc.getAttr('{baseRamp}.colorEntryList'.format(baseRamp=baseRamp),
size=True)
            if numEntries >= 3:

mc.removeMultiInstance('viewportBGcolour.colorEntryList[2]')

        #### get the ramp colour (I guess its just an average for now
        colour = mc.getAttr('viewportBGcolour.outColor')[0]
        print 'setting cam BG colour to', colour
        #get list of cameras
        cameras = mc.ls(type = 'camera', long = True)
        #iterate over cameras
        for camera in cameras:
            ########## with image planes
            # imagePlane = mc.createNode('imagePlane')
            # mc.setAttr
("{imagePlane}.type".format(imagePlane=imagePlane), 1)
            # mc.connectAttr ( 'BGcolour.outColor',
'{imagePlane}.sourceTexture'.format(imagePlane=imagePlane, force=True))
```

```
            #
mc.connectAttr('{imagePlane}.message'.format(imagePlane=imagePlane),'{cam
era}.imagePlane[0]'.format(camera = camera))

            ###### using cam BG colour....NB is a flat colour
            mc.setAttr("{camera}.backgroundColor".format(camera=camera),
colour[0], colour[1], colour[2], type='double3')
```