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

'''
07_08_16
Last worked on 29_07_15, these controls add attrs on transform node which
drive nodes
(including "condition", "clamp" and "set range" nodes) in order to
interactively cycle through
transforms shape nodes and blend between shapes.
* This does not work on transform nodes

TIPS FOR USE;
--> To add blend cycle > make sure that all shape nodes are visible
--> To convert shaders to Lambert > select shaders and click button
--> First add shader transparency attr > Then setup shader transparency
attr (select transform and click button)

'''

clm1 = 250
clm2 =100
clm3 = 100
separation = 15
sepStyle2 = 'in'
redBG = (.6, .45, .45)
blueBG = (.4, .4, .7)
blueBGLt = (.5, .5, .6)

# helpers
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 getNodeListFromTransform(transformNode):
    # transformNode = mc.listRelatives(node, parent=True,
type='transform')[0]
```

```
    shapeNodeList = mc.listRelatives(transformNode, children=True, type =
'shape')

    #### get rid of mesh_Orig and make new list ##########
    newList = []
    checkString1 = "Orig"
    checkString2 = "polySurfaceShape"
    for node in shapeNodeList:
        if checkString1 in node or checkString2 in node:
            print '>>> removing ', node
        else:
            newList.append(node)

    shapeNodeList = orderList(newList)
    return shapeNodeList



class CycleShapeCntrls(object):

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

        self.trailLenMax = 5
        self.lambertDiffuse = 0.5
        self.shapeIndexStartValue = "0"

    def buildUI(self):
        separation = 15

        separator = mc.separator(style =sepStyle2, w=300, h=separation,
parent=self.parentLayout)
        mc.text(' ==================  cycle shape controls
==================', parent=self.parentLayout)

        separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        row6 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text("Add simple cycleShape to selected trnsfrm :  ",
align='right', parent=row6, w=clm1)
        addAttrsButton = mc.button(label = 'simple cycleShape',
command=self.addSimpleCycleShape, parent=row6, w=clm2,
backgroundColor=blueBGLt)

        row7 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text("Delete simpleCycle cntrl on selected trnsfrm :  ",
align='right', parent=row7, w=clm1)
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        addAttrsButton = mc.button(label = 'delete simpleCycle',
command=self.deleteSimpleCycleAttr, parent=row7, w=clm2,
backgroundColor=redBG)
```

```
        separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        row3 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Add Blend shape cntrl to selected trnsfrm :  ',
align='right', parent=row3, w=clm1)
        addAttrsButton = mc.button(label = 'add blend cntrl',
command=self.addBlendAttr, parent=row3, w=clm2, backgroundColor=blueBG)

        row4 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Delete blend cntrl on selected trnsfrm :  ',
align='right', parent=row4, w=clm1)
        addAttrsButton = mc.button(label = 'delete blend cntrl',
command=self.deleteBlendAttr, parent=row4, w=clm2, backgroundColor=redBG)

        separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        row1 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Add cycleShape cntrl to selected trnsfrm :  ',
align='right', parent=row1, w=clm1)
        addAttrsButton = mc.button(label = 'add cycleShape',
command=self.addCycleAttr, parent=row1, w=clm2, backgroundColor=blueBG)

        row2 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Delete cycle shape cntrl on selected trnsfrm :  ',
align='right', parent=row2, w=clm1)
        # separator = mc.separator(style ="none", w=300, h=separation,
parent=self.parentLayout)
        addAttrsButton = mc.button(label = 'delete cycleShape',
command=self.deleteCycleAttr, parent=row2, w=clm2, backgroundColor=redBG)




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

        row6 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('convert selected shaders :  ', align='right',
parent=row6, w=clm1)
        addAttrsButton = mc.button(label = 'convert to Lambert',
command=self.convertToLambert, parent=row6, w=clm2)

        # text box with increment
        row7 = mc.rowLayout(numberOfColumns=4, parent=self.parentLayout)
        mc.text('shape index start :', width=clm2+clm3, parent=row7,
align='left')
        self.shapeIndexStart = mc.textField('shapeIndexStart',
parent=row7, width=clm1, \
                text='0', enterCommand=self.populateShapeIndexStart,
alwaysInvokeEnterCommandOnReturn=True)
```

```
        row5 = mc.rowLayout(numberOfColumns=2, columnWidth2=[(clm1),
(clm2)], parent=self.parentLayout)
        mc.text('Add shader trans attr to selected trnsfrm :  ',
align='right', parent=row5, w=clm1)
        addAttrsButton = mc.button(label = 'add Shader trans',
command=self.addTransparencyAttrs, parent=row5, w=clm2,
backgroundColor=blueBG)

    def populateShapeIndexStart(sef, *args):
        # shapeIndex = self.shapeIndexStartValue
        print "shapeIndex"


    def getColourConnection(self, surfaceShader):
        #check if node is connected to outColor if not get the colour as
a list
        if
mc.connectionInfo('{surfaceShader}.outColor'.format(surfaceShader=surface
Shader), isDestination=True):
            colorNode =
mc.listConnections('{surfaceShader}.outColor'.format(surfaceShader=surfac
eShader))[0]
            nodeType = mc.nodeType(colorNode)
            return colorNode, nodeType
        # else:
        #     # ???? dont know what this one does ???
        #     colourList =
mc.getAttr('{surfaceShader}.outColor'.format(surfaceShader=surfaceShader)
)[0]
        #     colourListType = type(colourList)
        #     return colourList, colourListType


    def convertToLambert(self, selection):
        # converts selected shaders to Lambert
        # print "convertToLambert"
        # get shader name
        selectedShaders = mc.ls(selection=True)
        for shader in selectedShaders:
            shaderType = mc.nodeType(shader)

            # check if its a surface shader
            if shaderType == "surfaceShader":
                # get the ramp
                colorNode, connectedNodeType =
self.getColourConnection(shader)
                if connectedNodeType == "ramp":
                    ramp = colorNode
                    # get the SG
                    # shadingGroup =
mc.listConnections('{shader}.outColor'.format(shader=shader))[0]
                    shadingGroupList =
mc.listConnections("{shader}.outColor".format(shader=shader),\
                         type="shadingEngine", exactType=True)
                    # print shadingGroup
                    # rename shader
```

```
                    mc.rename(shader,
"{shader}_old".format(shader=shader))
                    # create lambert
                    lambertNode = mc.shadingNode("lambert",
asShader=True, name=shader)
                    # connect ramp to color and ambient
                    mc.connectAttr("{ramp}.outColor".format(ramp=ramp),\

"{lambertNode}.color".format(lambertNode=lambertNode), force=True)
                    mc.connectAttr("{ramp}.outColor".format(ramp=ramp),\

"{lambertNode}.ambientColor".format(lambertNode=lambertNode), force=True)
                    # set default attr to 1

mc.setAttr("{lambertNode}.diffuse".format(lambertNode=lambertNode),
self.lambertDiffuse)

                    # connect to SG
                    for SG in shadingGroupList:

mc.connectAttr("{lambertNode}.outColor".format(lambertNode=lambertNode),\
                            "{SG}.surfaceShader".format(SG=SG),
force=True)


    def addTransparencyAttrs(self, *args):
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        if transform != None:
            self.addTransparencyEndAttr(transform)
            self.addTransparencyStartAttr(transform)
        else:
            print ">>> Transform node check failed"

        self.setUpShaderTrans(transform)
        mc.select(selection)

    def addTransparencyStartAttr(self, transform):
        shapeNodeList = getNodeListFromTransform(transform)
        numShapes = len(shapeNodeList)
        trailLenMax = numShapes
        # check if attr exists and add if it doesnt
        attrExistence = mc.attributeQuery('transparencyStart',
node=transform, exists=True)
        if attrExistence == False:
            mc.addAttr(transform, shortName='transparencyStart', min=0,
max=trailLenMax, storable=True, \
                    attributeType='double', keyable=True)
            print ">  transparencyStart attr added to ", transform
        else:
            print ">  transparencyStart attr ALREADY added to ",
transform

    def addTransparencyEndAttr(self, transform):
        shapeNodeList = getNodeListFromTransform(transform)
        numShapes = len(shapeNodeList)
        trailLenMax = numShapes
        # check if attr exists and add if it doesnt
```

```
        attrExistence = mc.attributeQuery('transparencyEnd',
node=transform, exists=True)
        if attrExistence == False:
            mc.addAttr(transform, shortName='transparencyEnd', min=0,
max=trailLenMax, storable=True, \
                       attributeType='double', keyable=True)
            print ">  transparencyEnd attr added to ", transform
        else:
            print ">  transparencyEnd attr ALREADY added to ", transform


    def setUpShaderTrans(self, selection):
        print "setUpShaderTrans"
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        if transform != None:
            # NB. the following function returns an list ordered
ascending
            shapeNodeList = getNodeListFromTransform(transform)
            numShapes = len(shapeNodeList)

            index = 0
            for shape in shapeNodeList:

                # get shaders on shape
                shaderList = self.getShaderList(shape)
                # make sure theres a switch node connected
                for shader in shaderList:
                    switchNode = self.createAndConnectSwitch(shader)
                    self.addShapeToSwitch(shape, switchNode, index)
                    rampCntrlName =
"{transform}{shape}_transCntrl".format(transform=transform, shape=shape)
                    self.createAndConnectRamp(switchNode, index,
rampCntrlName)

                self.createTransparencyNodes(shape, index, transform)
                index += 1

    def createTransparencyNodes(self, shape, index, transform):
        # ammend to accomodate transStart and trans end
        # addMin;  addMax; clamp;  set range

        addMin = mc.shadingNode("addDoubleLinear", asUtility=True)
        mc.setAttr("{addMin}.input1".format(addMin=addMin), index)
        addMax = mc.shadingNode("addDoubleLinear", asUtility=True)
        mc.setAttr("{addMax}.input1".format(addMax=addMax), index+1)

        clamp = mc.shadingNode("clamp", asUtility=True)

        setRange = mc.shadingNode("setRange", asUtility=True)
        mc.setAttr("{setRange}.minX".format(setRange=setRange), 0)
        mc.setAttr("{setRange}.maxX".format(setRange=setRange), 1)

        # connect attrs

mc.connectAttr("{transform}.transparencyStart".format(transform=transform
),\
         "{addMin}.input2".format(addMin=addMin))
```

```
        mc.connectAttr("{addMin}.output".format(addMin=addMin),\
         "{clamp}.minR".format(clamp=clamp))
        mc.connectAttr("{addMin}.output".format(addMin=addMin),\
         "{setRange}.oldMinX".format(setRange=setRange))


mc.connectAttr("{transform}.transparencyEnd".format(transform=transform),
\
         "{addMax}.input2".format(addMax=addMax))
        mc.connectAttr("{addMax}.output".format(addMax=addMax),
            "{clamp}.maxR".format(clamp=clamp))
        mc.connectAttr("{addMax}.output".format(addMax=addMax),
            "{setRange}.oldMaxX".format(setRange=setRange))

        mc.connectAttr("{clamp}.outputR".format(clamp=clamp),
            "{setRange}.valueX".format(setRange=setRange))

mc.connectAttr("{transform}.shapeCycle".format(transform=transform),
            "{clamp}.inputR".format(clamp=clamp))

        # connect to ramp shader on shape
        rampName =
"{transform}{shape}_transCntrl".format(transform=transform, shape=shape)
        mc.connectAttr("{setRange}.outValueX".format(setRange=setRange),\
         "{rampName}.colorEntryList[0].colorR".format(rampName=rampName))
        mc.connectAttr("{setRange}.outValueX".format(setRange=setRange),\
         "{rampName}.colorEntryList[0].colorG".format(rampName=rampName))
        mc.connectAttr("{setRange}.outValueX".format(setRange=setRange),\
         "{rampName}.colorEntryList[0].colorB".format(rampName=rampName))


    def getShaderList(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 shaders from teh shading groups
            shaders = self.getShaderFromSG(shadingGroupList)
            #removeDuplicates
            shaderList  = list(set(shaders))
            print '>>> shaderList  ==  ',  shaderList

            return shaderList

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

        return shaderList
```

```
    def createAndConnectSwitch(self, shader):
        # connect triple switch to transparency is its not already
connected
        switchNodeList =
mc.listConnections("{shader}.transparency".format(shader=shader),
type="tripleShadingSwitch", exactType=True)
        if switchNodeList == None:
            # only make switch if needed
            switchNode = mc.shadingNode("tripleShadingSwitch",
asUtility=True)
        else:
            switchNode = switchNodeList[0]

        switchNodeType = mc.nodeType(switchNode)

        #### find out whether the shader is a surface shader
        shaderType = mc.nodeType(shader)
        if shaderType != 'surfaceShader':
            # ## connect switch to material if its not already connected
            if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{shader}.transparency'.format(shader=shader)):

mc.connectAttr('{switchNode}.output'.format(switchNode=switchNode),
'{shader}.transparency'.format(shader=shader), force=True)
        else:
            print "its a surface shader... this wont work"
        return switchNode

    def addShapeToSwitch(self, shape, switch, index):
        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} at index
{i}'.format(shape=shape, switch=switch, i=index)


    def createAndConnectRamp(self, switch, index, rampCntrlName):
        # make ramp if its not already connected
        rampList =
mc.listConnections("{switch}.input[{i}].inTriple".format(switch=switch,
i=index))
        if rampList == None:
            # check if node of rampName exists
            rampExists = mc.objExists(rampCntrlName)
            if rampExists == True:
                ramp = rampCntrlName
            else:
                # NB only one ramp per shape
                ramp = mc.shadingNode("ramp", asUtility=True,
name=rampCntrlName)

mc.removeMultiInstance("{ramp}.colorEntryList[1]".format(ramp=ramp),
b=True)
```

```
mc.removeMultiInstance("{ramp}.colorEntryList[2]".format(ramp=ramp),
b=True)

mc.setAttr("{ramp}.colorEntryList[0].color".format(ramp=ramp), 0, 0, 0)
        else:
            ramp = rampList[0]
        print "ramp is ", ramp

        # # check if ramp is connected at current index
        if
mc.isConnected('{ramp}.colorEntryList[0].color'.format(ramp=ramp),
"{switch}.input[{i}].inTriple".format(switch=switch, i=index)):
            print '>>> ramp node already connected to ', switch, "index
", index
        else:

mc.connectAttr('{ramp}.colorEntryList[0].color'.format(ramp=ramp),
'{switch}.input[{i}].inTriple'.format(switch=switch, i=index),
force=True)
            print '>>> {ramp} node just connected to '.format(ramp=ramp),
switch, "index ", index


    def transformCheck(self, selection):
        # check if selection is a single transform node
        if len(selection) != 1:
            print ">>> Please select ONE node"
        else:
            transform = mc.ls(selection=True)[0]

            # check if its a transform node
            nodeType = mc.nodeType(transform)
            if nodeType != "transform":
                print ">>> Please select a transform node"
            else:
                return transform


    def addVisTrailAttr(self, transform, numShapes):
        trailLenMax = numShapes
        # trailLenMax = self.trailLenMax
        print "addVisTrailAttr"
            # check if attr exists and add if it doesnt
        attrExistence = mc.attributeQuery('visibilityTrail',
node=transform, exists=True)
        if attrExistence == False:
            mc.addAttr(transform, shortName='visibilityTrail', min=0,
max=trailLenMax, storable=True, \
                    attributeType='double', keyable=True)
            print ">  visibilityTrail attr added to ", transform
        else:
            print ">  visibilityTrail attr ALREADY added to ", transform

    def addVisPreludeAttr(self, transform, numShapes):
        trailLenMax = numShapes
            # check if attr exists and add if it doesnt
```

```
        attrExistence = mc.attributeQuery('visibilityPrelude',
node=transform, exists=True)
        if attrExistence == False:
            mc.addAttr(transform, shortName='visibilityPrelude', min=0,
max=trailLenMax, storable=True, \
                    attributeType='double', keyable=True)
            print ">  visibilityTrail attr added to ", transform
        else:
            print ">  visibilityTrail attr ALREADY added to ", transform

    def deleteVisTrailAttr(self, selection):
        print "deleteVisTrailAttr"


    def deleteCycleAttr(self, *args):
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        if transform != None:
            attrExistence = mc.attributeQuery('shapeCycle',
node=transform, exists=True)
            if attrExistence == True:
                # delete connected condition nodes if they exist
                conditionNodeList = mc.listConnections(transform,
exactType=True, type="condition")
                # print conditionNodeList
                for node in conditionNodeList:
                    mc.delete(node)
                    print ">>   condition node ", node, " deleted"


mc.deleteAttr("{transform}.shapeCycle".format(transform=transform))
                print ">>   ShapeCycle attr deleted"


            attrExistence2 = mc.attributeQuery('visibilityTrail',
node=transform, exists=True)
            if attrExistence2 == True:

mc.deleteAttr("{transform}.visibilityTrail".format(transform=transform))
                print ">>   visibilityTrail attr deleted"

            attrExistence2 = mc.attributeQuery('visibilityPrelude',
node=transform, exists=True)
            if attrExistence2 == True:

mc.deleteAttr("{transform}.visibilityPrelude".format(transform=transform)
)
                print ">>   visibilityPrelude attr deleted"

            # delete mult div node and set shape vis to 1
            shapeNodeList = getNodeListFromTransform(transform)
            for shape in shapeNodeList:
                # delete multi divide also
                multDivNodeList = mc.listConnections(shape,
exactType=True, type="multiplyDivide")
                for node in multDivNodeList:
                    mc.delete(node)
                    print ">>   multiplyDivide node ", node, " deleted"
```

```
                mc.setAttr("{shape}.visibility".format(shape=shape), 1)

    def addCycleAttr(self, *args):
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        # print "transform value == ", transform
        if transform != None:
            # NB. the following function returns an list ordered
ascending
            shapeNodeList = getNodeListFromTransform(transform)
            numShapes = len(shapeNodeList)

            # start by adding visibility trail attr
            self.addVisPreludeAttr(transform, numShapes)
            self.addVisTrailAttr(transform, numShapes)

            # check if attr exists and add if it doesnt
            attrExistence = mc.attributeQuery('shapeCycle',
node=transform, exists=True)
            if attrExistence == False:
                # use long data type for simple cycle shape ##########
                # mc.addAttr(transform, shortName='shapeCycle', min=0,
max=(numShapes-1), storable=True, \
                #         attributeType='long', keyable=True)
                mc.addAttr(transform, shortName='shapeCycle', min=0,
max=(numShapes-1), storable=True, \
                        attributeType='double', keyable=True)

                print ">  Cycle attr added to ", transform
            else:
                print ">  Cycle attr ALREADY added to ", transform

            # connect cycleAttr
            self.connectCycleAttr(transform, shapeNodeList)

        else:
            print ">>> Transform node check failed"
        mc.select(selection)

# simple cycleShape
    def addSimpleCycleShape(self, *args):
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        # print "transform value == ", transform
        if transform != None:
            # NB. the following function returns an list ordered
ascending
            shapeNodeList = getNodeListFromTransform(transform)
            numShapes = len(shapeNodeList)
            # check if attr exists and add if it doesnt
            attrExistence = mc.attributeQuery('simpleShapeCycle',
node=transform, exists=True)
            if attrExistence == False:
                # use long data type for simple cycle shape ##########
                mc.addAttr(transform, shortName='simpleShapeCycle',
min=0, max=(numShapes-1), storable=True, \
```

```
                        attributeType='long', keyable=True)
                print ">  simpleShapeCycle attr added to ", transform
            else:
                print ">  simpleShapeCycle attr ALREADY added to ",
transform

            # connect cycleAttr
            self.connectSimpleCycleAttr(transform, shapeNodeList)

        else:
            print ">>> Transform node check failed"
        mc.select(selection)

    def deleteSimpleCycleAttr(self, *args):
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        if transform != None:
            attrExistence = mc.attributeQuery('simpleShapeCycle',
node=transform, exists=True)
            if attrExistence == True:
                # delete connected condition nodes if they exist
                conditionNodeList = mc.listConnections(transform,
exactType=True, type="condition")
                # print conditionNodeList
                for node in conditionNodeList:
                    mc.delete(node)
                    print ">>   condition node ", node, " deleted"


mc.deleteAttr("{transform}.simpleShapeCycle".format(transform=transform))
                print ">>   ShapeCycle attr deleted"

            # set shape vis to 1
            shapeNodeList = getNodeListFromTransform(transform)
            for shape in shapeNodeList:
                mc.setAttr("{shape}.visibility".format(shape=shape), 1)


    def connectSimpleCycleAttr(self, transform, shapeNodeList):
        numShapes = len(shapeNodeList)
        shapeIndex = 0
        for shape in shapeNodeList:
            # use the following for simple cycle shape
            conditionNodeName = self.createConditionNode(transform,
shape, numShapes, shapeIndex)
            shapeIndex += 1


    def connectCycleAttr(self, transform, shapeNodeList):
        numShapes = len(shapeNodeList)
        shapeIndex = 0
        for shape in shapeNodeList:
            # use the following for simple cycle shape
            # conditionNodeName = self.createConditionNode(transform,
shape, numShapes, shapeIndex)
            # create more complex node setup
            self.createCycleShapeNodes(transform, shape, numShapes,
shapeIndex)
```

```
            shapeIndex += 1


    def createCycleShapeNodes(self, transform, shape, numShapes,
shapeIndex):
        add = mc.createNode("addDoubleLinear")
        conditionNodeMax = mc.createNode("condition", name="Max")
        conditionNodeMin = mc.createNode("condition", name="Min")
        multDiv = mc.createNode("multiplyDivide")

        # add stuff for Vis Prelude
        multDivPrelude = mc.createNode("multiplyDivide")
        addPrelude = mc.createNode("addDoubleLinear")

        # set attrs
        mc.setAttr("{add}.input1".format(add=add), shapeIndex + 1)

mc.setAttr("{conditionNodeMax}.colorIfTrueR".format(conditionNodeMax=cond
itionNodeMax), 1)

mc.setAttr("{conditionNodeMax}.colorIfFalseR".format(conditionNodeMax=con
ditionNodeMax), 0)

mc.setAttr("{conditionNodeMax}.operation".format(conditionNodeMax=conditi
onNodeMax), 4)

mc.setAttr("{conditionNodeMin}.colorIfTrueR".format(conditionNodeMin=cond
itionNodeMin), 0)

mc.setAttr("{conditionNodeMin}.colorIfFalseR".format(conditionNodeMin=con
ditionNodeMin), 1)

mc.setAttr("{conditionNodeMin}.operation".format(conditionNodeMin=conditi
onNodeMin), 4)
        # theh following is now connected to vis Prelude attr
        #
mc.setAttr("{conditionNodeMin}.secondTerm".format(conditionNodeMin=condit
ionNodeMin), shapeIndex)

        # set attr for vis prelude

mc.setAttr("{multDivPrelude}.input2X".format(multDivPrelude=multDivPrelud
e), -1)
        mc.setAttr("{addPrelude}.input2".format(addPrelude=addPrelude),
shapeIndex)

        # connect attrs

mc.connectAttr("{transform}.visibilityTrail".format(transform=transform),
\
            "{add}.input2".format(add=add),  force=True)
        mc.connectAttr("{add}.output".format(add=add), \

"{conditionNodeMax}.secondTerm".format(conditionNodeMax=conditionNodeMax)
,  force=True)

mc.connectAttr("{conditionNodeMax}.outColorR".format(conditionNodeMax=con
ditionNodeMax), \
```

```
            "{multDiv}.input2X".format(multDiv=multDiv),  force=True)

mc.connectAttr("{transform}.shapeCycle".format(transform=transform), \

"{conditionNodeMax}.firstTerm".format(conditionNodeMax=conditionNodeMax),
force=True)

        mc.connectAttr("{multDiv}.outputX".format(multDiv=multDiv), \
            "{shape}.visibility".format(shape=shape),  force=True)

mc.connectAttr("{conditionNodeMin}.outColorR".format(conditionNodeMin=con
ditionNodeMin), \
            "{multDiv}.input1X".format(multDiv=multDiv),  force=True)

mc.connectAttr("{transform}.shapeCycle".format(transform=transform), \

"{conditionNodeMin}.firstTerm".format(conditionNodeMin=conditionNodeMin),
force=True)
        # connect attr for vis prelude

mc.connectAttr("{transform}.visibilityPrelude".format(transform=transform
), \

"{multDivPrelude}.input1X".format(multDivPrelude=multDivPrelude),
force=True)

mc.connectAttr("{multDivPrelude}.outputX".format(multDivPrelude=multDivPr
elude), \
            "{addPrelude}.input1".format(addPrelude=addPrelude),
force=True)

mc.connectAttr("{addPrelude}.output".format(addPrelude=addPrelude), \

"{conditionNodeMin}.secondTerm".format(conditionNodeMin=conditionNodeMin)
, force=True)


    def createConditionNode(self, transform, shape, numShapes,
shapeNumber):
        #  NB. this one is for simple cycle shape with no vis trail
capability
        # precaution if condition node exists and Maya renames
        preferredname = "condNode{shape}".format(shape=shape)
        conditionNodeName = mc.createNode("condition", name =
preferredname)
        # print ">>  created ConditionNode  ", conditionNodeName

        # connect first term

mc.connectAttr("{transform}.simpleShapeCycle".format(transform=transform)
, \

"{conditionNodeName}.firstTerm".format(conditionNodeName=conditionNodeNam
e),  force=True)
        # print ">>  connected ConditionNode  ", transform, " to",
conditionNodeName
        # connect to shapeVis
```

```
mc.connectAttr("{conditionNodeName}.outColorR".format(conditionNodeName=c
onditionNodeName),\
            "{shape}.visibility".format(shape=shape), force=True)

        # add shapeNum attr to condition node with default value of
shapeNum
        mc.addAttr(conditionNodeName, longName="shapeNum",
attributeType="long", defaultValue=shapeNumber, keyable=True)
        # set second Term to shapeNumber

mc.setAttr("{conditionNodeName}.secondTerm".format(conditionNodeName=cond
itionNodeName), shapeNumber)


mc.setAttr("{conditionNodeName}.colorIfTrueR".format(conditionNodeName=co
nditionNodeName), 1)

mc.setAttr("{conditionNodeName}.colorIfFalseR".format(conditionNodeName=c
onditionNodeName), 0)

        return conditionNodeName



        # BLEND setup

    def deleteBlendAttr(self, *args):
        print "delete blend attr"


        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        if transform != None:

            # go to frame 1
            mc.currentTime(1)
            # get shapes and iterate through deleting history
            shapeNodeList = getNodeListFromTransform(transform)
            for shape in shapeNodeList:
                mc.delete(constructionHistory=True)



            attrExistence = mc.attributeQuery('BlendCycle',
node=transform, exists=True)
            if attrExistence == True:
                # delete connected clamp nodes if they exist
                clampNodeList = mc.listConnections(transform,
exactType=True, type="clamp")
                for clampNode in clampNodeList:
                    # get set range node
                    setRange = mc.listConnections(clampNode,
exactType=True, type="setRange")
                    # delete set range node
                    mc.delete(setRange)
                    print ">>   ", setRange, " deleted"
                    # delete clamp node
```

```
                    mc.delete(clampNode)
                    print ">>   ", clampNode, " deleted"


mc.deleteAttr("{transform}.BlendCycle".format(transform=transform))
                print ">>   BlendCycle attr deleted"

    def addBlendAttr(self, *args):
        selection = mc.ls(selection=True)
        transform = self.transformCheck(selection)
        # print "transform value == ", transform
        if transform != None:
            # NB. the following function returns an list ordered
ascending
            shapeNodeList = getNodeListFromTransform(transform)
            numShapes = len(shapeNodeList)
            # check if attr exists and add if it doesnt
            attrExistence = mc.attributeQuery('BlendCycle',
node=transform, exists=True)
            if attrExistence == False:
                mc.addAttr(transform, shortName='BlendCycle', min=0,
max=(numShapes-1), storable=True, \
                           attributeType='double', keyable=True)
                print ">  Blend attr added to ", transform
            else:
                print ">  Blend attr ALREADY added to ", transform

            blendNode = self.createBlendNode(transform, shapeNodeList)

        else:
            print ">>> Blend node check failed"

        mc.select(selection)

    def createBlendNode(self, transform, shapeNodeList):
        print "shapeNodeList  ", shapeNodeList
        # create Blend nodes
        index = 0
        for x in range(0, (len(shapeNodeList)-1)):
            weightShape = shapeNodeList[index+1]

            mc.select(shapeNodeList[index+1])
            mc.select(shapeNodeList[index], add=True)
            blendNode = mc.blendShape(origin="world",
topologyCheck=False)[0]
            print "created blend ", blendNode

            # create clamp
            clamp = mc.shadingNode("clamp", asUtility=True)
            # create set range
            setRange = mc.shadingNode("setRange", asUtility=True)
            print "created clamp ", clamp
            print "created setRange ", setRange

            # connect attrs

mc.connectAttr("{transform}.BlendCycle".format(transform=transform),\
```

```
                "{clamp}.inputR".format(clamp=clamp), force=True)

            mc.connectAttr("{clamp}.outputR".format(clamp=clamp), \
                "{setRange}.valueX".format(setRange=setRange),
force=True)

mc.connectAttr("{setRange}.outValueX".format(setRange=setRange),\
             "{blendNode}.{weightShape}".format(blendNode=blendNode,
weightShape=weightShape), force=True)

            # set attrs
            # minVal = index
            minVal = index -1
            # maxVal = index + 1
            maxVal = index
            mc.setAttr("{clamp}.minR".format(clamp=clamp), minVal)
            mc.setAttr("{clamp}.maxR".format(clamp=clamp), maxVal)

            mc.setAttr("{setRange}.oldMinX".format(setRange=setRange),
minVal)
            mc.setAttr("{setRange}.oldMaxX".format(setRange=setRange),
maxVal)
            mc.setAttr("{setRange}.minX".format(setRange=setRange), 0)
            # mc.setAttr("{setRange}.maxX".format(setRange=setRange), 2)
            mc.setAttr("{setRange}.maxX".format(setRange=setRange), 1)

            index += 1
```