```
import maya.cmds as mc
import maya.OpenMaya as om
from functools import partial
'''
02_07_14
key shader transparency for trail effect
- base the keys on vis keyFrames on the transform or shape node
- make it work for lambert shaders and surface shaders
'''

# def keyMeshVisibility(meshName='meshName', meshNumber=0, keySpacing=4,
visTrail=4):
#     frameNumber = int(meshNumber)

#     #set 0 vis at frame 1
#     mc.setKeyframe( "{meshName}.v".format(meshName = meshName), time =
0, value = 0)

#     # #set first keyframe to visible
#     firstKey = frameNumber * keySpacing
#     mc.setKeyframe( "{meshName}.v".format(meshName = meshName),
time=firstKey, value=1)

#     #set secondKey to off
#     secondKey = firstKey + keySpacing + visTrail
#     mc.setKeyframe( "{meshName}.v".format(meshName = meshName), time =
secondKey, value = 0)

#     return meshNumber

def getMaterialFromSG(shadingGroupName):
    shadingGroup = shadingGroupName
    material =
mc.listConnections('{shadingGroup}.surfaceShader'.format(shadingGroup=sha
dingGroup))
    return material

# class CreateBlendSahpes(object):

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

#     def buildUI(self):
#         mc.text('-- Select transform node to create blend nodes on
shapes'\
#             , parent=self.parentLayout)

#         mc.button(label = 'Create blend anim tween selected nodes',
width=200, command=self.createBlends,\
#             parent=self.parentLayout)
#                 ### feedback text
#         # self.feedbackText = mc.text('  Feedback:',
parent=self.parentLayout)

#     def createBlends(self, *args):
#         ###### get the shapes as a list
```

```
#         # shapeNodeList = mc.ls(orderedSelection=True, shapes=True,
dag=True, objectsOnly=True)
#         transformNode = mc.ls(selection=True)
#         shapeNodeList = mc.listRelatives(transformNode, children=True,
type = 'mesh')
#         print shapeNodeList
#         # get rid of the first mesh entry
#         nodeList = shapeNodeList.remove(shapeNodeList[0])
#         # print nodeList
#         # nodeList = shapeNodeList


#         i = 0
#         for index in range(len(shapeNodeList)-1):
#             blendTarget = shapeNodeList[i+1]
#             print blendTarget
#             node = shapeNodeList[i]
#             print node

#             #go to frame 0
#             mc.currentTime(0)
#             ## delete the first vis key
#             mc.cutKey( node, time=(0,1), attribute='v', option="keys")
#             mc.cutKey( blendTarget, time=(0,1), attribute='v',
option="keys")

#             mc.select(blendTarget, replace=True)
#             mc.select(node, add=True)
#             #create the blend node and key its value
#             blendNode = mc.blendShape( origin='local', tc=0)[0]

#             ### get vis keys
#             keyframes = mc.keyframe('{node}.v'.format(node=node),
query=True)

#             ## set blend key 1
#             key1 = keyframes[0]
#
mc.setKeyframe('{blendNode}.{blendTarget}'.format(blendNode=blendNode,
blendTarget=blendTarget),\
#                 time = key1, value = 0)
#             key2 = keyframes[1]
#
mc.setKeyframe('{blendNode}.{blendTarget}'.format(blendNode=blendNode,
blendTarget=blendTarget),\
#                 time = key2, value = 1)

#             ### put the first vis keyframe back on the target  and teh
node
#             mc.setKeyframe('{node}.v'.format(node=node), time = 0,
value = 0)
#
mc.setKeyframe('{blendTarget}.v'.format(blendTarget=blendTarget), time =
0, value = 0)

#             i+=1

# class CreateBlendSahpesTweenMeshNodes(object):
```

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

#         self.workOnChildNodes = True
#         self.blendOriginIsWorld = True

#         self.buildUI()

#     def buildUI(self):
#         clm1 = 125
#         clm2 = 75
#         clm3 = 100
#         separation = 15

#         mc.text('', 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)))

#         inputLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
#         mc.text('    Key spacing')
#         self.keySpacingTF = mc.textField('keySpacingTF',
parent=inputLayout, width=25, text='10')
#         mc.text('    Trail size')
#         self.visTrailTF = mc.textField('visTrailTF',
parent=inputLayout, width=25, text='10')

#         visLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
#         mc.button(label = 'Create vis keys', width=clm1,
command=self.keyMeshVis,\
#             parent=visLayout)
#         mc.button(label = 'Delete keys', width=clm2,
command=self.deleteVisKeys,\
#             parent=visLayout)

#         separator = mc.separator(style ="none", w=300, h=10,
parent=self.parentLayout)

#         ### blend origin controls
#         self.BlendRadioCollection = mc.radioCollection()
#         RBblendLayout = mc.rowLayout(numberOfColumns=2,
parent=self.parentLayout)
#         self.worldRB = mc.radioButton(label='Blend origin = world',
parent=RBblendLayout, select=True,
onCommand=(partial(self.setBlendOriginToWorld, True)))
#         self.selectedNodesRB = mc.radioButton(label='Blend origin =
relative', parent=RBblendLayout,
onCommand=(partial(self.setBlendOriginToWorld, False)))
```

```
#         blendAlignLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
#         mc.text('Blend align')
#         self.blendAlignTF = mc.textField('blendAlignTF',
parent=blendAlignLayout, width=40, text='start')
#         mc.text('Blend duration')
#         self.blendDurationTF = mc.textField('blendDurationTF',
parent=blendAlignLayout, width=25, text='10')


#         blendLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
#         mc.button(label = 'Create blend nodes', width=clm1,
command=self.createBlends,\
#             parent=blendLayout)
#         mc.button(label = 'Delete history', width=clm2,
command=self.deleteHistory,\
#             parent=blendLayout)

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

#     def setBlendOriginToWorld(self, status, *args):
#         self.blendOriginIsWorld = status
#         print 'blendOriginIsWorld status = ', self.blendOriginIsWorld

#     def deleteVisKeys(self, *args):
#         #deletes visibility keys on selected Node and set vis to 1
#         if self.workOnChildNodes == True:
#             groupNode = mc.ls(selection=True)
#             tranformNodeList = mc.listRelatives(groupNode,
children=True, type = 'transform')
#             nodeList = tranformNodeList
#         else:
#             nodeList = mc.ls(selection=True)

#         i = 0
#         for index in range(len(nodeList)):
#             node = nodeList[i]

#             #check if there is keys on vis and delete if yes
#             visKeys = mc.keyframe('{node}.v'.format(node=node),
query=True)
#             if visKeys != None:
#                 print 'deleting existing keys'
#                 mc.cutKey('{node}'.format(node=node),
attribute='visibility', clear=True)

#             ##set node vis to 1
#             mc.setAttr('{node}.v'.format(node=node), 1)

#             i += 1

#     def deleteHistory(self, *args):
#         #deletes visibility keys on selected Node
#         if self.workOnChildNodes == True:
#             groupNode = mc.ls(selection=True)
```

```
#             tranformNodeList = mc.listRelatives(groupNode,
children=True, type = 'transform')
#             nodeList = tranformNodeList

#         else:
#             nodeList = mc.ls(selection=True)

#         i = 0
#         for index in range(len(nodeList)):
#             node = nodeList[i]

#             mc.delete('{node}'.format(node=node),
constructionHistory=True)

#             i += 1

#     def keyMeshVis(self, *args):
#         ### if there are vis keys delete them

#         keySpacing = int(mc.textField(self.keySpacingTF, query=True,
text=True))
#         visTrail = int(mc.textField(self.visTrailTF, query=True,
text=True))

#         # print 'visKeySpacing is ', keySpacing

#         if self.workOnChildNodes == True:
#             groupNode = mc.ls(selection=True)
#             tranformNodeList = mc.listRelatives(groupNode,
children=True, type = 'transform')
#             nodeList = tranformNodeList

#         else:
#             nodeList = mc.ls(selection=True)


#         # Loop to set keys on model vis
#         i = 0
#         for index in range(len(nodeList)):
#             node = nodeList[i]

#             #check if there is keys on vis and delete if yes
#             visKeys = mc.keyframe('{node}.v'.format(node=node),
query=True)
#             if visKeys != None:
#                 print 'deleting existing keys'
#                 mc.cutKey('{node}'.format(node=node),
attribute='visibility', clear=True)


#             keyMeshVisibility(meshName=node, meshNumber=i,
keySpacing=keySpacing, visTrail=visTrail)
#             i += 1

#     def createBlends(self, *args):
#         visKeyOffTime = False
```

```
#         blendDuration = int(mc.textField(self.blendDurationTF,
query=True, text=True))
#         if self.blendOriginIsWorld == True:
#             blendOrigin = 'world'
#         else:
#             blendOrigin = 'local'

#         if self.workOnChildNodes == True:
#             groupNode = mc.ls(selection=True)
#             tranformNodeList = mc.listRelatives(groupNode,
children=True, type = 'transform')
#             nodeList = tranformNodeList

#         else:
#             nodeList = mc.ls(selection=True)

#         #### create a list to keep track of the blend nodes
#         blendNodeList = []

#         i = 0
#         for index in range(len(nodeList)-1):
#             blendTarget = nodeList[i+1]
#             node = nodeList[i]
#             ### get the shape node
#             # shapeNode = mc.listRelatives(node, children=True, type =
'shape')[0]

#             blendNode = mc.blendShape(blendTarget, node,
origin=blendOrigin, tc=0)[0]

#             #### add the blend node to the list
#             blendNodeList.append(blendNode)

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

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

#                 index += 1

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

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

#                   blendKey1 = visKeyOffTime

#
mc.setKeyframe('{blendNode}.{blendTarget}'.format(blendNode=blendNode,
blendTarget=blendTarget),\
#                       time = blendKey1, value = 1)

#             i += 1

#         print 'blendNodeList is ', blendNodeList

class KeyShaderTransparency(object):

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

        self.buildUI()

    def buildUI(self):
        clm1 = 125
        clm2 = 75
        clm3 = 100
        separation = 15

        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)))

        separator = mc.separator(style ="none", w=300, h=10,
parent=self.parentLayout)

        trailLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.text('Key spacing')
        self.keySpacingTF = mc.textField('keySpacingTF',
parent=trailLayout, width=25, text='10')
        mc.text('     Trail size')
        self.trailSizeTF = mc.textField('trailSizeTF',
parent=trailLayout, width=25, text='10')

        transLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.button(label = 'Create trans keys', width=clm1,
command=self.keyMaterialTrans,\
            parent=transLayout)
```

```
        mc.button(label = 'Delete keys', width=clm2,
command=self.deleteTransKeys,\
            parent=transLayout)

    def deleteTransKeys(self, *args):
                #deletes visibility keys on selected Node and set vis to
1
        if self.workOnChildNodes == True:
            groupNode = mc.ls(selection=True)
            tranformNodeList = mc.listRelatives(groupNode, children=True,
type = 'transform')
            nodeList = tranformNodeList
        else:
            nodeList = mc.ls(selection=True)

        i = 0
        for index in range(len(nodeList)):
            #### in order to get the switch node;
            # get the shape node
            shapeNode = mc.listRelatives(node, children=True, type =
'shape')[0]
            #get the SG on the mesh
            shadingGroup = mc.listConnections(shapeNode,
type='shadingEngine')[0]

            ## get the material from teh shading group
            material = getMaterialFromSG(shadingGroup)[0]
            ### 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


                    #check if there is keys on vis and delete if yes
```

```
            # visKeys = mc.keyframe('{node}.v'.format(node=node),
query=True)
            # if visKeys != None:
            #     print 'deleting existing keys'
            #     mc.cutKey('{node}'.format(node=node),
attribute='visibility', clear=True)

            ##set node vis to 1
            mc.setAttr('{switchNode}.v'.format(node=node), 1)



    def keyMaterialTrans(self, *args):

        if self.workOnChildNodes == True:
            groupNode = mc.ls(selection=True)
            tranformNodeList = mc.listRelatives(groupNode, children=True,
type = 'transform')
            nodeList = tranformNodeList

        else:
            nodeList = mc.ls(selection=True)


        i=0
        for node in nodeList:
            # get the shape node
            shapeNode = mc.listRelatives(node, children=True, type =
'shape')[0]
            #get the SG on the mesh
            shadingGroup = mc.listConnections(shapeNode,
type='shadingEngine')[0]
            ## get the material from teh shading group
            material = getMaterialFromSG(shadingGroup)[0]

            ### 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:
                print 'creating a single swicth node '
                #create a switch node and connect it to the shader
                switchNode = self.createSwitchNode()


            # ## 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))

            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))

            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))

            ### add the shape to the switchNode
            if not
mc.isConnected('{shapeNode}.instObjGroups[0]'.format(shapeNode=shapeNode)
, '{switchNode}.input[{i}].inShape'.format(switchNode=switchNode, i=i)):

mc.connectAttr('{shapeNode}.instObjGroups[0]'.format(shapeNode=shapeNode)
, '{switchNode}.input[{i}].inShape'.format(switchNode=switchNode, i=i))

            ###         #if its a surface surfaceShader    #### NB. this
doesnt work
            # switchNode = self.createSwitchNode()
            # if not
mc.isConnected('{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparency'.format(material=material)):
            #     print
'{switchNode}.output'.format(switchNode=switchNode),
'{material}.outTransparency'.format(material=material)

            self.createTransAttrs(switchNode=switchNode, index=i)

            i+=1
```

```
            # print 'shadingGroup is ', shadingGroup
            # print 'shapeNode is ', shapeNode
            # print 'material is ',material

    def createTransAttrs(self, switchNode, index):
        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)

        #### 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))

        # singleShadingSwitch1.input[0].inSingle
        onKey, offKey = self.getTransFrameNumbers(meshNumber)

        ### 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, meshNumber):
        frameNumber = int(meshNumber)
        keySpacing = int(mc.textField(self.keySpacingTF, query=True,
text=True))
        trailLength = int(mc.textField(self.trailSizeTF, query=True,
text=True))

        # #set first keyframe to visible
        onKey = frameNumber * keySpacing

        #set secondKey to off
        offKey = onKey + keySpacing + trailLength

        print keySpacing, onKey, offKey
        return onKey, offKey
```

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

    def createSwitchNode(self):
        switchNode = mc.createNode('singleShadingSwitch')
        return switchNode

# def launchUI():
#     clm1 = 50
#     clm2 = 50
#     clm3 = 100
#     separation = 15

#     if mc.window('postProductionTools', query=True, exists=True):
#         mc.deleteUI('postProductionTools')
#     myWindow = mc.window('postProductionTools', title='Post Production
Tools', w=(clm1+clm2+clm3), h=100)
#     mc.showWindow(myWindow)

#     mainColumn = mc.columnLayout()

#     # ########## add UI for blend shapes tween shape nodes
#     # separator = mc.separator(style ="in", w=300, h=separation,
parent=mainColumn)
#     # mc.text('-----  Create Blends between shape nodes ---------------
--------'\
#     #         , parent=mainColumn, align='right' )

#     # CreateBlendSahpes(parentLayout=mainColumn)

#     ########## add UI for blend shapes tween transform? nodes
#     separator = mc.separator(style ="in", w=300, h=separation,
parent=mainColumn)
#     mc.text('>>>>  Create vis keys and blends on mesh nodes -----------
-----'\
#             , parent=mainColumn, align='right' )
#     separator = mc.separator(style ="none", w=300, h=5,
parent=mainColumn)

#     CreateBlendSahpesTweenMeshNodes(parentLayout=mainColumn)

#         ########## add UI for key Transparency tween transform? nodes
#     separator = mc.separator(style ="in", w=300, h=separation,
parent=mainColumn)
#     mc.text('>>>>  Key mesh transparency ----------------'\
#             , parent=mainColumn, align='right' )
#     separator = mc.separator(style ="none", w=300, h=5,
parent=mainColumn)
#     KeyMeshTransparency(parentLayout=mainColumn)

#     mc.showWindow(myWindow)

# launchUI()
```