```
import maya.cmds as mc
from functools import partial
from operator import itemgetter
'''
23_07_15
-adding reparent current shape node
------> GINA this is an extra file for testing mesh controls before (or
without) adding them to PPmeshCntrl

'''
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

class meshFunctionsUI(object):

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

        self.workOnChildNodes = True
        ### keep a list of the movePolyVert nodes
        self.movePolyVertList = []

        self.buildUI()

    def buildUI(self):
        clm1 = 65
        clm2 = 65
        clm3 = 125
        separation = 15

        if self.workOnTransformNodeStatus == True:
            mc.text('- Operate on transform nodes',
parent=self.parentLayout)
        else:
            mc.text('- Operate on shape nodes', parent=self.parentLayout)
```

```
        separator = mc.separator(style ="none", w=300, h=10,
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)))

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

        startEndLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.text('Start')
        self.startTF = mc.textField('startTF', parent=startEndLayout,
width=25, text='10')
        mc.text('Duration')
        self.durationTF = mc.textField('blendDurationTF',
parent=startEndLayout, width=25, text='10')

        self.functionNameRow = mc.rowLayout(numberOfColumns=2,
columnWidth2=[(clm1 + clm2), clm3], parent=self.parentLayout)
        mc.text('                  Function name :',
parent=self.functionNameRow, align='right')

        functionNameTF = mc.textField('functionNameTF',
text='transformComponent', width=clm3, parent=self.functionNameRow)


        creationLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.button(label = 'Create', width=clm3,
command=self.performFunction,\
            parent=creationLayout)
        mc.button(label = 'Delete', width=clm1,
command=self.deleteFunctionEffects,\
            parent=creationLayout)

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

    def performFunction(self, *args):
        ###########>>>> check the function text field
        self.transformComponent()

    def transformComponent(self):
        random = 10
        value = 0.02
        setTwoKeys = True

        ### get ordered node List
        nodeList = self.getNodeList()
```

```
        for node in nodeList:
            # get the start and end based on vis keys on the node
            start, end = self.getStartAndEndKeys(node)
            # print 'start, end == ', start, end

            ## add transform component
            moveVert = mc.polyMoveVertex(node, constructionHistory=1,
random=random)[0]
            ### add it to the list in order to delete it
            self.movePolyVertList.append(moveVert)

            ##set key 1
            key1Time = start - 1
            mc.setKeyframe("{moveVert}.ltx".format(moveVert=moveVert),
time=(key1Time,key1Time))
            mc.setKeyframe("{moveVert}.lty".format(moveVert=moveVert),
time=(key1Time,key1Time))
            mc.setKeyframe("{moveVert}.ltz".format(moveVert=moveVert),
time=(key1Time,key1Time))


            if setTwoKeys == True:
                ###### the following sets 2 keys

                ## set attr for key 2
                key2Time = end + 1
                mc.currentTime(key2Time)

mc.setAttr("{moveVert}.localTranslateX".format(moveVert=moveVert), value)

mc.setAttr("{moveVert}.localTranslateY".format(moveVert=moveVert), value)

mc.setAttr("{moveVert}.localTranslateZ".format(moveVert=moveVert), value)

                ##set key 2

mc.setKeyframe("{moveVert}.ltx".format(moveVert=moveVert),
time=(key2Time,key2Time))

mc.setKeyframe("{moveVert}.lty".format(moveVert=moveVert),
time=(key2Time,key2Time))

mc.setKeyframe("{moveVert}.ltz".format(moveVert=moveVert),
time=(key2Time,key2Time))


            else:
                ###### the following sets 3 keys
                ##set key 3
                key3Time = end + 1

mc.setKeyframe("{moveVert}.ltx".format(moveVert=moveVert),
time=(key3Time,key3Time))

mc.setKeyframe("{moveVert}.lty".format(moveVert=moveVert),
time=(key3Time,key3Time))
```

```
mc.setKeyframe("{moveVert}.ltz".format(moveVert=moveVert),
time=(key3Time,key3Time))

                ## set attr for key 2
                key2Time = key1Time + ((key3Time - key1Time) * 0.5)
                mc.currentTime(key2Time)

mc.setAttr("{moveVert}.localTranslateX".format(moveVert=moveVert), value)

mc.setAttr("{moveVert}.localTranslateY".format(moveVert=moveVert), value)

mc.setAttr("{moveVert}.localTranslateZ".format(moveVert=moveVert), value)

                ##set key 2

mc.setKeyframe("{moveVert}.ltx".format(moveVert=moveVert),
time=(key2Time,key2Time))

mc.setKeyframe("{moveVert}.lty".format(moveVert=moveVert),
time=(key2Time,key2Time))

mc.setKeyframe("{moveVert}.ltz".format(moveVert=moveVert),
time=(key2Time,key2Time))


    def getNodeList(self):
        if self.workOnChildNodes == True:
            groupNode = mc.ls(selection=True)
            tranformNodeList = mc.listRelatives(groupNode, children=True,
type = 'transform')
            shapeNodeList = mc.listRelatives(groupNode, children=True,
type = 'shape')
            # print 'tranformNodeList, shapeNodeList', tranformNodeList,
shapeNodeList

            if self.workOnTransformNodeStatus == True:
                unorderedNodeList = tranformNodeList
            else:
                unorderedNodeList = shapeNodeList
        else:
            unorderedNodeList = mc.ls(selection=True)

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

        nodeList = orderList(newList)
        return nodeList
```

```
    def getStartAndEndKeys(self, node):
        ### these are based on vis keys and teh start and end as entered
        visKeyOffTime = False
        #### ADD radio button to set clipToVisOffTime ####
        clipToVisOffTime = True
        start = int(mc.textField(self.startTF, query=True, text=True))
        duration = int(mc.textField(self.durationTF, query=True,
text=True))

        visKeys = mc.keyframe('{node}.v'.format(node=node), query=True)

        ### get VIS on and off keys
        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]
            index += 1


        if visKeyOffTime != False:

            start = visKeyOnTime + start

            if clipToVisOffTime == True:
                # print 'if neccessary will clip to VisOffTime',
clipToVisOffTime

                if int(start + duration) > int(visKeyOffTime):
                    # print 'yep its greater'
                    end = visKeyOffTime
                else:
                    end = start + duration
            else:
                end = start + duration

        return start, end

    def deleteFunctionEffects(self, *args):
        if self.movePolyVertList != []:
            ## go to frame -1
            mc.currentTime(-1)

            for moveVertNode in self.movePolyVertList:
                mc.delete(moveVertNode)

class shuffleMeshFunctionsUI(object):

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

        self.buildUI()

    def buildUI(self):
```

```
        clm1 = 65
        clm2 = 65
        clm3 = 125
        separation = 15

        separator = mc.separator(style ="in", w=300, h=separation,
parent=self.parentLayout)

        if self.workOnTransformNodeStatus == True:
            mc.text('- helpers for woring with transform nodes',
parent=self.parentLayout)
        else:
            mc.text('- helpers for woring with shape nodes',
parent=self.parentLayout)

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

        # creationLayout = mc.rowLayout(numberOfColumns=4,
parent=self.parentLayout)
        mc.button(label = 'Duplicate shape', width=clm3,
command=self.duplicateShape,\
            parent=self.parentLayout)
        mc.button(label = 'Shuffle names', width=clm3,
command=self.shuffleNodeNames,\
            parent=self.parentLayout)

    def getNodeListFromTransform(self, node):
        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:
            # print node
            if checkString1 in node or checkString2 in node:
                print '>>> removing ', node
            else:
                newList.append(node)

        nodeList = orderList(newList)
        return nodeList

    def getNodeNameAndNumber(self, node):
        nodeNameList = node.split('_')

        nodeName = nodeNameList[0]
        nodeNumberStr = nodeNameList[-1]

        if nodeNumberStr == '':
            nodeNumber = 0
        else:
            nodeNumber = int(nodeNumberStr)
```

```
        return nodeName, nodeNumber

    def renameNodes(self, listToBeRenamed):
        #order the list
        # orderedListToBeRenamed = orderList(listToBeRenamed)
        # reverse the list
        listToBeRenamed.reverse()
        print 'listToBeRenamed == ', listToBeRenamed

        # newNameList = []

        for node in listToBeRenamed:
            nodeName = self.getNodeNameAndNumber(node)[0]
            nodeNumber = self.getNodeNameAndNumber(node)[1]
            # nodeNumberStr = str(nodeNumber)
            newNumber = nodeNumber + 1
            newNumberStr = str(newNumber)

            newNodename =
'{nodeName}_{newNumberStr}'.format(nodeName=nodeName,
newNumberStr=newNumberStr)
            #### rename the node
            mc.rename(node, newNodename)

        #     newNameList.append(newlyNamedNode)

        # return newNameList


    def shuffleNodeNames(self, *args):
        selectedNode = mc.ls(selection=True)[0]
        selectedNodeNumber = self.getNodeNameAndNumber(selectedNode)[1]

        nodeList = self.getNodeListFromTransform(selectedNode)

        ########## make new list of nodes to be renames
        nodesToBeRenamed = []

        for node in nodeList:
            nodeNumber = self.getNodeNameAndNumber(node)[1]

            ########## make new list of nodes to be renamed
            if nodeNumber > selectedNodeNumber:
                nodesToBeRenamed.append(node)

        print 'nodesToBeRenamed ==  ', nodesToBeRenamed
        self.renameNodes(nodesToBeRenamed)

        ### getting an error try clearing selection ####### doesnt help
        # mc.select(clear=True)

    def duplicateShape(self, *args):
        ### the following means that all later meshes will be renamed
        #### NB. decided to try this actuvated from a separate button
        shuffle=False

        node = mc.ls(selection=True)[0]
```

```
        nodeTansform = mc.listRelatives(node, parent=True,
type='transform')[0]

        duplicateTransform = mc.duplicate(node)[0]
        # print duplicateTransform
        #
mc.select('{duplicateTransform}|{node}'.format(duplicateTransform=duplica
teTransform, node=node))
        newShapeNode =
'{duplicateTransform}|{node}'.format(duplicateTransform=duplicateTransfor
m, node=node)

        renamedNode = mc.rename(newShapeNode,
'{node}00'.format(node=node))

        # print renamedNode
        mc.parent(renamedNode, nodeTansform, shape=True, relative=True)
        mc.delete(duplicateTransform)

        if shuffle == True:
            self.shuffleNodeNames(node)

# the following done for cloured whippet bind

def renameShapeNodes():
    transformNode = 'colouredWhippet'
    nodeList = getNodeListFromTransform(transformNode)
    print nodeList

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:
        # print node
        if checkString1 in node or checkString2 in node:
            print '>>> removing ', node
        else:
            newList.append(node)

    nodeList = orderList(newList)
    return nodeList

class BindVisibleShapeNode(object):

    def __init__(self, transformNode):
        self.startFrame = 1
        self.endFrame = 1000
        self.transformNode = transformNode

        self.doIt()
```

```
    def doIt(self):
        startFrame = self.startFrame
        endFrame = self.endFrame

        for frame in range(startFrame, endFrame):
            #set the currentTime
            mc.currentTime(frame)
            ## get the visible shape node
            visibleShapes = self.getVisibleShape()

            if visibleShapes == []:
                print '>>>  no shapes visible on this frame'
            else:
                for shape in visibleShapes:
                    # bind the mesh
                    mc.skinCluster('GINGER:bone_knee_BR', shape)
                    print 'frame, shape ==    ', frame, shape


    def getVisibleShape(self):
        shapeNodeList = getNodeListFromTransform(self.transformNode)
        # print shapeNodeList

        visibleShapeList = []
        for shape in shapeNodeList:
            #check if it is visible
            visValue =
mc.getAttr('{shape}.visibility'.format(shape=shape))
            if visValue == True:
                visibleShapeList.append(shape)

        return visibleShapeList


# BindVisibleShapeNode('colouredWhippet')
```