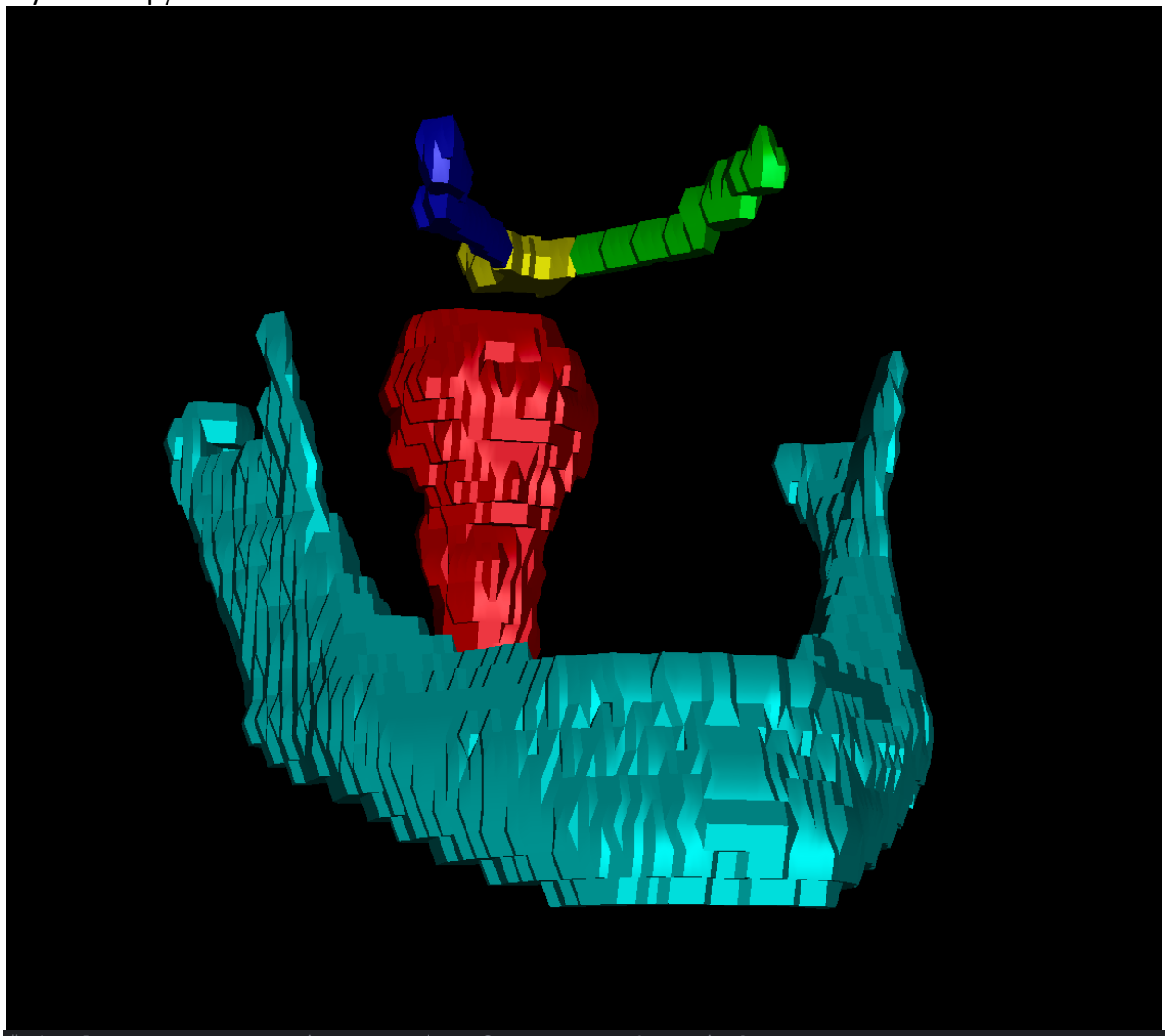
myVTKWin.py:



```
# % Class to create interactive 3D VTK render window
# % EECE 8396: Medical Image Segmentation
# % Spring 2024
# % Author: Prof. Jack Noble; jack.noble@vanderbilt.edu
#

# % Example usage shown in the following demo functions below:
#  demoPointsAndLines()
#  demoSurfaceAppearance()
#  demoSurfaceEdgesAndColors()
#  demoDepthOfField()
#  brainPointPick()
#  bouncingBallsAnimation()
#  brainAnimation()
#  demoSurfaceFromNRRD()

import vtk
import numpy as np
```

```
class vtkObject:
    def __init__(self, pnts=None, poly=None, actor=None):
        self.pnts = pnts
        self.poly = poly
        self.actor = actor

    def updateActor(self, verts):
        for j,p in enumerate(verts):
            self.pnts.InsertPoint(j,p)
        self.poly.Modified()


def ActorDecorator(func):
    def inner(verts,faces=None,color=[1,0,0],opacity=1.0, colortable=None,
coloridx=None):
        pnts = vtk.vtkPoints()
        for j,p in enumerate(verts):
            pnts.InsertPoint(j,p)

        poly = func(pnts,faces)

        #important for smooth rendering
        norm = vtk.vtkPolyDataNormals()
        norm.SetInputData(poly)

        mapper = vtk.vtkPolyDataMapper()
        mapper.SetInputConnection(norm.GetOutputPort())

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        if coloridx is None:
            actor.GetProperty().SetColor(color[0],color[1],color[2])
        else:
            scalars = vtk.vtkDoubleArray()
            for j in range(len(verts)):
                scalars.InsertNextValue(coloridx[j] / (len(colortable)-1))

            lut = vtk.vtkLookupTable()
            lut.SetNumberOfTableValues(len(colortable))
            for j in range(len(colortable)):
                lut.SetTableValue(j,colortable[j,0],colortable[j,1],
colortable[j,2])

            lut.Build()

            poly.GetPointData().SetScalars(scalars)
            norm.SetInputData(poly)
            mapper.SetInputConnection(norm.GetOutputPort())
            prop = actor.GetProperty()
            # prop.SetColor(0,0,0)
            mapper.SetLookupTable(lut)
            mapper.SetScalarRange([0.0, 1.0])

        actor.GetProperty().SetOpacity(opacity)
        actor.GetProperty().SetPointSize(4)
        obj = vtkObject(pnts, poly, actor)
        return obj
```

```
    return inner

@ActorDecorator
def pointActor(pnts, faces=None):
    cells = vtk.vtkCellArray()
    for j in range(pnts.GetNumberOfPoints()):
        vil = vtk.vtkIdList()
        vil.InsertNextId(j)
        cells.InsertNextCell(vil)

    poly = vtk.vtkPolyData()
    poly.SetPoints(pnts)
    poly.SetVerts(cells)

    return poly

@ActorDecorator
def linesActor(pnts,lines):
    cells = vtk.vtkCellArray()
    for j, f in enumerate(lines):
        vil = vtk.vtkIdList()
        vil.InsertNextId(lines[j,0])
        vil.InsertNextId(lines[j,1])
        cells.InsertNextCell(vil)

    poly = vtk.vtkPolyData()
    poly.SetPoints(pnts)
    poly.SetLines(cells)

    return poly

@ActorDecorator
def surfActor(pnts,faces):
    cells = vtk.vtkCellArray()
    for j, f in enumerate(faces):
        vil = vtk.vtkIdList()
        vil.InsertNextId(faces[j,0])
        vil.InsertNextId(faces[j,1])
        vil.InsertNextId(faces[j,2])
        cells.InsertNextCell(vil)

    poly = vtk.vtkPolyData()
    poly.SetPoints(pnts)
    poly.SetPolys(cells)

    poly.BuildCells()
    poly.BuildLinks()

    return poly



class myVtkWin(vtk.vtkRenderer):
    def __init__(self, sizex=512, sizey=512, title="3D Viewer (press q to
quit)"):
        super().__init__()
```

```
        self.renwin = vtk.vtkRenderWindow() #creates a new window
        self.renwin.SetWindowName(title)
        self.renwin.AddRenderer(self)
        self.renwin.SetSize(sizex, sizey)
        self.inter = vtk.vtkRenderWindowInteractor() #makes the renderer
interactive
        self.inter.AddObserver('KeyPressEvent',self.keypress_callback,1.0)
        self.lastpickpos = np.zeros(3)
        self.lastpickcell = -1
        self.inter.SetRenderWindow(self.renwin)
        self.inter.Initialize()

self.inter.SetInteractorStyle(vtk.vtkInteractorStyleTrackballCamera())

        self.objlist = []

        self.renwin.Render() # paints the window on the screen once

    def __del__(self):
        del self.renwin, self.inter


    def addPoints(self, verts, color=[1.,0.,0.], opacity=1.):
        obj = pointActor(np.asarray(verts), color=color, opacity=opacity)
        self.objlist.append(obj)
        self.AddActor(obj.actor)

    def addLines(self, verts, lns, color=[1.,0.,0.], opacity=1.):
        obj = linesActor(np.asarray(verts), np.asarray(lns), color=color,
opacity=opacity)
        self.objlist.append(obj)
        self.AddActor(obj.actor)

    def addSurf(self, verts, faces, color=[1.,0.,0.], opacity=1.,
                specular=0.9, specularPower=25.0, diffuse=0.6, ambient=0,
edgeColor=None,
                colortable=None, coloridx=None):
        obj = surfActor(np.asarray(verts), np.asarray(faces), color=color,
opacity=opacity, colortable=colortable, coloridx=coloridx)
        self.objlist.append(obj)
        actor = obj.actor
        if edgeColor is not None:
            actor.GetProperty().EdgeVisibilityOn()
            actor.GetProperty().SetEdgeColor(edgeColor[0], edgeColor[1],
edgeColor[2])
        actor.GetProperty().SetAmbientColor(color[0], color[1], color[2])
        actor.GetProperty().SetDiffuseColor(color[0], color[1], color[2])
        actor.GetProperty().SetSpecularColor(1.0,1.0,1.0)
        actor.GetProperty().SetSpecular(specular)
        actor.GetProperty().SetDiffuse(diffuse)
        actor.GetProperty().SetAmbient(ambient)
        actor.GetProperty().SetSpecularPower(specularPower)

        self.AddActor(actor)
        if len(self.objlist)==1:
            mn = actor.GetCenter()
            self.GetActiveCamera().SetFocalPoint(mn[0],mn[1],mn[2])
```

```
    def keypress_callback(self,obj,ev):
        key = obj.GetKeySym()
        if (key == 'u' or key == 'U'):
            pos = obj.GetEventPosition()

            picker = vtk.vtkCellPicker()
            picker.SetTolerance(0.0005)

            picker.Pick(pos[0],pos[1],0,self)

            self.lastpickpos = picker.GetPickPosition()
            self.lastpickcell = picker.GetCellId()
        return key

    def updateActor(self, id, verts):
        self.objlist[id].updateActor(np.asarray(verts))

    def cameraPosition(self, position=None, viewup=None, fp=None ,
focaldisk=None):
        cam = self.GetActiveCamera()
        if position is not None:
            cam.SetPosition(position[0], position[1], position[2])
        if viewup is not None:
            cam.SetViewUp(viewup[0], viewup[1], viewup[2])
        if fp is not None:
            cam.SetFocalPoint(fp[0], fp[1], fp[2])
        if focaldisk is not None:
            dist  = np.sqrt(np.sum((np.array(cam.GetFocalPoint()) -
np.array(cam.GetPosition()))**2))
            cam.SetFocalDisk(focaldisk*dist)

    def render(self):
        self.ResetCameraClippingRange()
        self.renwin.Render()
        self.inter.ProcessEvents()

    def start(self):
        self.inter.Start()

# function to build cylindrical triangular surface mesh using two endpoints
def cylinder(vert1, vert2, rad=1.0, numcirc=16):
    verts = np.zeros((numcirc*2, 3))
    v = vert2 - vert1
    vec = np.array([1.0,0.,0.])
    if np.abs(np.sum(v*vec)/np.linalg.norm(v))>0.95:
        vec = np.array([0, 1.0,0.])

    v1 = np.cross(v, vec)[np.newaxis,:]
    v1 /= np.linalg.norm(v1)
    v2 = np.cross(v, v1)[np.newaxis,:]
    v2 /= np.linalg.norm(v2)
    theta = np.linspace(0, 2*np.pi, numcirc)[:,np.newaxis]
    verts[0:numcirc,:] = vert1[np.newaxis,:] + rad*(np.cos(theta)*v1 +
np.sin(theta)*v2)
    verts[numcirc::,:] = vert2[np.newaxis,:] + rad * (np.cos(theta) * v1 +
np.sin(theta) * v2)
```

```
    faces = np.zeros((numcirc*2 + 2*(numcirc-2), 3), dtype=int)
    for i in range(numcirc-2):
        faces[i,:] = np.array([0, i+1, i+2])
    for i in range(numcirc-2):
        faces[i+numcirc-2,:] = np.array([0, i+1, i+2]) + numcirc
    for i in range(numcirc):
        faces[i+2*(numcirc-2),:] = np.array([i, (i+1)%numcirc, i+numcirc])
    for i in range(numcirc):
        faces[i+numcirc+2*(numcirc-2),:] = np.array([(i+1)%numcirc,
(i+1)%numcirc+numcirc, i+numcirc, ])

    return verts, faces

# Basic point and line display
def demoPointsAndLines():
    verts = np.array([[0.,0.,0],[1.,1.,1.]])
    win = myVtkWin(title="Two points and Three lines")
    win.addPoints(verts)
    win.cameraPosition(position=[0.,0.,5.],viewup=[0,1,0],fp=[0.5,.5,.5])

    #show three lines
    verts = np.array([[0.,0.,0],[1.,1.,1.],[1.,0.,0.]])
    lns = np.array([[0,1],[1,2],[2,0]])

    win.addLines(verts,lns,color=[0,0,1.])
    win.cameraPosition([0.,0.,5.],[0,1,0],[0.5,.5,.5])
    win.start()

# Different types of surface rendering
def demoSurfaceAppearance():
    verts = np.array([[0.,0.,0],[1.,1.,1.],[1.,0.,0.]])
    win = myVtkWin(title='Ambient, diffuse, and specular rendering')

    # display surface
    sverts,sfaces = cylinder(verts[0,:],verts[1,:],rad=0.1,numcirc=16)
    win.addSurf(sverts,sfaces,color=[.5,.5,.5],opacity=1,specular=.1)

    sverts,sfaces = cylinder(verts[1,:],verts[2,:],rad=0.1,numcirc=32)

win.addSurf(sverts,sfaces,color=[.5,.5,.5],opacity=1,specular=0,diffuse=0,amb
ient=1)

    sverts,sfaces = cylinder(verts[2,:],verts[0,:],rad=0.1,numcirc=32)
    win.addSurf(sverts,sfaces,color=[.5,.5,.5],opacity=1,specular=.9)

    win.cameraPosition([0.,0.,5.],[0,1,0],[0.5,.5,.5])
    win.start()


# Triangle edges can be made visible for wire display
def demoSurfaceEdgesAndColors():
    verts = np.array([[0.,0.,0],[0.,0.,1.]])
    win = myVtkWin(title='Edge visibility/Colormapping')

    # display surface
    sverts,sfaces = cylinder(verts[0,:],verts[1,:],rad=0.1,numcirc=16)
```

```
    colortable = np.concatenate((
        np.concatenate((np.zeros(32),np.linspace(0.0,1.0,32)))[:,np.newaxis],
# red

np.concatenate((np.linspace(0.0,1.0,32),np.linspace(1.0,0.0,32)))[:,np.newaxi
s],  #green

np.concatenate((np.linspace(1.0,0.0,33)[1::],np.zeros(32)))[:,np.newaxis]),ax
is=1)
    mn = np.min(sverts[:,0])
    mx = np.max(sverts[:,0])
    coloridx = np.floor((sverts[:,0] - mn) / (mx - mn) * 63.999).astype(int)

    win.addSurf(sverts,sfaces,ambient=0.9, opacity=1,
edgeColor=[0.,0.,0.],colortable=colortable,coloridx=coloridx)

    win.cameraPosition([5.,0.,.5],[0,0,1],[0,0,.5])
    win.start()

# Can simulate realistic camera optic effects using depth-of-field
def demoDepthOfField():
    verts = np.array([[0.,0.,0],[1.,1.,1.],[1.,0.,0.]])
    win = myVtkWin(title='Simulating real lens depth-of-field')

    # display surface
    sverts,sfaces = cylinder(verts[0,:],verts[1,:],rad=0.1,numcirc=16)
    win.addSurf(sverts,sfaces,color=[.5,.5,.5],opacity=1,specular=.1)

    sverts,sfaces = cylinder(verts[1,:],verts[2,:],rad=0.1,numcirc=32)

win.addSurf(sverts,sfaces,color=[.5,.5,.5],opacity=1,specular=0,diffuse=0,amb
ient=1)

    sverts,sfaces = cylinder(verts[2,:],verts[0,:],rad=0.1,numcirc=32)
    win.addSurf(sverts,sfaces,color=[.5,.5,.5],opacity=1,specular=.9)

    basicPasses = vtk.vtkRenderStepsPass()
    dofp = vtk.vtkDepthOfFieldPass()
    dofp.SetDelegatePass(basicPasses)
    dofp.AutomaticFocalDistanceOff()
    win.SetPass(dofp)

    # small focal disk -> longer depth of field
    win.cameraPosition(fp=[-1,-1,-1],focaldisk=.02, position=[-4, -2.5, -4],
viewup=[0.25, 0.76, -0.6])
    win.start()


# Custom Point/Cell picking implemented with 'u' key
def brainPointPick():
    import json
    f = open('brain.json','rt')
    dct = json.load(f)
    f.close()
    verts = np.array(dct['verts'])
    faces = np.array(dct['faces'])
```

```
    class printPickWin(myVtkWin):
        def keypress_callback(self,obj,ev):
            super().keypress_callback(obj,ev)
            worldPosition = self.lastpickpos
            cell = self.lastpickcell
            print(f'Picked point coordinate: {worldPosition[0]:.2f}
{worldPosition[1]:.2f} {worldPosition[2]:.2f}')
            print(f'Cell Id: {cell:d}')
            cam = self.GetActiveCamera()
            campos = cam.GetPosition()
            camfp = cam.GetFocalPoint()
            camvu = cam.GetViewUp()
            print(f'Camera Position: {campos[0]:.2f} {campos[1]:.2f}
{campos[2]:.2f}')
            print(f'Camera Focal Point: {camfp[0]:.2f} {camfp[1]:.2f}
{camfp[2]:.2f}')
            print(f'Camera View Up: {camvu[0]:.2f} {camvu[1]:.2f}
{camvu[2]:.2f}')

    win = printPickWin(1024,512, title='Point pick using ''u'' key')
    win.addSurf(verts,faces,color=[1.,.8,.8])
    vu = np.array([-.43,-.9,-.12])
    vu = vu / np.linalg.norm(vu)
    fp = np.mean(verts,axis=0)
    win.cameraPosition(position=[500,-40,15],viewup=vu,fp=fp)

    # try point picking with 'u'
    win.start()

# create screenshot test.png and video file test.avi with spinning brain
using ffmpeg
# shows how to (1) move camera, (2) create screenshot, (3) create videos
def brainAnimation():
    import json
    import vtkmodules.vtkRenderingCore
    from subprocess import Popen,PIPE
    from vtk.util.numpy_support import vtk_to_numpy

    f = open('brain.json','rt')
    dct = json.load(f)
    f.close()
    verts = np.array(dct['verts'])
    faces = np.array(dct['faces'])

    win = myVtkWin(1024,512, title='Screenshot and Video using ffmpeg')
    win.addSurf(verts,faces,color=[1.,.8,.8])
    vu = np.array([-.43,-.9,-.12])
    vu = vu / np.linalg.norm(vu)
    fp = np.mean(verts,axis=0)
    win.cameraPosition(position=[500,-40,15],viewup=vu,fp=fp)
    win.render()

    windowToImageFilter =
vtkmodules.vtkRenderingCore.vtkWindowToImageFilter()
    windowToImageFilter.SetInput(win.renwin)
    windowToImageFilter.SetInputBufferTypeToRGBA()
```

```
    windowToImageFilter.ReadFrontBufferOn()
    windowToImageFilter.Update()
    out = windowToImageFilter.GetOutput()

    png = vtk.vtkPNGWriter()
    png.SetInputData(out)
    png.SetFileName("test.png")
    png.Write()


    fps = 15
    N = 100
    cam = win.GetActiveCamera()
    command = ["C:\\Users\\noblejh\\Downloads\\ffmpeg-5.1.2-
essentials_build\\bin\\ffmpeg",
               '-loglevel','error',
               '-y',
               # Input
               '-f','rawvideo',
               '-vcodec','rawvideo',
               '-pix_fmt','bgr24',
               '-s',str(1024) + 'x' + str(512),
               '-r',str(fps),
               # Output
               '-i','-',
               '-an',
               '-vcodec','mpeg4',  #'h264',
               '-r',str(fps),
               '-pix_fmt','bgr24',
               "test.avi"
               ]
    p = Popen(command,stdin=PIPE)
    #timing looks rough in real time rendering but is fine in the final avi
file
    for i in range(N):
        cam.Azimuth(360.0 / N)  # degrees
        win.render()
        windowToImageFilter =
vtkmodules.vtkRenderingCore.vtkWindowToImageFilter()
        windowToImageFilter.SetInput(win.renwin)
        windowToImageFilter.SetInputBufferTypeToRGBA()
        windowToImageFilter.ReadFrontBufferOff()

        windowToImageFilter.Update()
        out = windowToImageFilter.GetOutput()
        sc = out.GetPointData().GetScalars()
        r = vtk_to_numpy(sc)
        r2 = np.flip(np.flip(r.reshape(512,1024,4)[:,:,0:3],axis=2),axis=0)
        r2o = r2.tobytes()
        p.stdin.write(r2o)

    p.stdin.close()
    p.wait()

    win.start()
```

```
# shows how to (1) create surface using marching cubes,
# (2) manipulate surfaces for animations, (3) create custom lighting/shadows
def bouncingBallsAnimation():
    import skimage.measure
    import vtkmodules.vtkRenderingCore
    N = 1000
    rad1 = 1
    rad2 = .5

    # sphere equation on grid
    X,Y,Z = np.meshgrid(np.arange(-25,26), np.arange(-25,26), np.arange(-
25,26), indexing='ij')
    sph = 400 - (X*X +Y*Y + Z*Z)

    # sphere centered at [25,25,25] with radius=20 voxels
    verts, faces, _, _ = skimage.measure.marching_cubes(sph, 0)

    # zero center and normalize radius to 1
    verts = (verts - 25)/ 20

    #create 2 side-by-side spheres
    sph1 = verts*rad1
    sph2 = verts*rad2 + np.array([[2.,0.,0.]])

    # create 'floor' to bounce the spheres on
    vertsfloor = np.array([[-2,-5,0],[6,-5,0],[-2,5,0],[6,5,0]])
    trisfloor = np.array([[0,1,2],[2,1,3]],dtype=int)

    win = myVtkWin(512,512,title='bouncing balls')

    shadows = vtk.vtkShadowMapPass()
    seq = vtk.vtkSequencePass()

    passes = vtk.vtkRenderPassCollection()
    passes.AddItem(shadows.GetShadowMapBakerPass())
    passes.AddItem(shadows)
    seq.SetPasses(passes)

    cameraP = vtk.vtkCameraPass()
    cameraP.SetDelegatePass(seq)

    # Tell the renderer to use our render pass pipeline
    win.SetPass(cameraP)

    win.addSurf(sph1, faces, color=[1,0,0], specular=0.9)
    win.addSurf(sph2, faces, color=[0,1,0], specular=0.9)
    win.addSurf(vertsfloor,trisfloor,color=[1,1,1],ambient=0.2)
    win.cameraPosition(position=[1.5,-15,4],viewup=[0,0,1],fp=[1.5,0,1])

# create static light
    light = vtk.vtkLight()
    light.SetFocalPoint(2.5,0,0)
    light.SetPosition(-15,0,20)
    win.AddLight(light)
    cam = win.GetActiveCamera()

    theta = np.linspace(0,np.pi,50)
```

```
    for i in range(N):
        sph1[:,2] = verts[:,2]*rad1 + rad1 + np.sin(theta[i % 50])
        sph2[:,2] = verts[:,2]*rad2 + rad2 + np.sin(theta[(i+25) % 50])
        win.updateActor(0, sph1)
        win.updateActor(1, sph2)
        cam.Azimuth(360.0 / N)
        win.render()


    win.start()

# surface class
class surface:
    def __init__(self):
        self.verts = None
        self.faces = None

def demoSurfaceFromNRRD():
    import nrrd
    import nibabel as nib
    from skimage import measure

    # load CT image
    img, header = nrrd.read('/Users/leonslaptop/Desktop/2024 Spring/ECE
3892/data/0522c0001/img.nrrd')

    # Specify the path to your NIfTI file
    file_path = '/Users/leonslaptop/Desktop/2024 Spring/Research/Pelvis/head-
NIFTI/head-Decompressed_CT_0_1.nii'
    # Load the NIfTI file
    nifti_file = nib.load(file_path)
    # Get the data from the file
    img = nifti_file.get_fdata()

    #isosurface it at isolevel =700 to separate bone from soft-tissue/air
    #When isosurfacing a binary segmentation mask, often an isolevel=0.5 is
used
    s = surface()
    s.verts, s.faces,_,_ = measure.marching_cubes(img, level=-300)

    # display result in myVtkWin
    win = myVtkWin()
    win.addSurf(s.verts, s.faces, color=[1,.9,.8])
    win.start()

    # create surface accounting for anisotropic voxel size
    voxsz = [header['space directions'][0][0], header['space
directions'][1][1],
             header['space directions'][2][2]] # mm/voxel
    s.verts,s.faces,_,_ = measure.marching_cubes(img,level=700,
spacing=voxsz)

    win = myVtkWin()
    win.addSurf(s.verts,s.faces,color=[1,.9,.8])
    win.start()
```

```
def createSurfaceFromVolume(self, img, voxsz, isolevel):
    from skimage import measure
    # Use marching cubes to generate vertices and faces and assign generated
vertices and faces to class variables
    self.verts, self.faces, _, _ = measure.marching_cubes(img,
level=isolevel, spacing=voxsz)


def projectOneTaskOne():
    # Initialize visualization window
    win = myVtkWin(title="Project One Task One ")

    # Define file paths and isolevels
    structures = [
        ("data/0522c0001/structures/brainstem.nrrd", 0, [1.0, 0.0, 0.0]),  #
Red
        ("data/0522c0001/structures/OpticNerve_L.nrrd", 0, [0.0, 1.0, 0.0]),
# Green
        ("data/0522c0001/structures/OpticNerve_R.nrrd", 0, [0.0, 0.0, 1.0]),
# Blue
        ("data/0522c0001/structures/chiasm.nrrd", 0, [1.0, 1.0, 0.0]),  #
Yellow
        ("data/0522c0001/structures/mandible.nrrd", 0, [0.0, 1.0, 1.0])  #
Cyan
    ]

    # Process and display each structure
    for filePath, isolevel, color in structures:
        s = loadAndProcessStructure(filePath, isolevel)
        win.addSurf(s.verts, s.faces, color=color, opacity=1.0)

    # Finalize and start the visualization
    win.cameraPosition(position=[0, -800, 0], viewup=[0, 0, 1])
    win.start()

def loadAndProcessStructure(filePath, isolevel):
    import nrrd
    # Load NRRD file
    img, header = nrrd.read(filePath)
    voxsz = [header['space directions'][0][0], header['space
directions'][1][1],
             header['space directions'][2][2]] # mm/voxel

    # Create surface
    s = surface()
    createSurfaceFromVolume(s, img, voxsz, isolevel)
    return s



if __name__ == "__main__":

    # demoPointsAndLines()
    # demoSurfaceAppearance()
    # demoSurfaceEdgesAndColors()
    # demoDepthOfField()
    # brainPointPick()
```

```
    # brainAnimation()
    # bouncingBallsAnimation()
    # demoSurfaceFromNRRD()
    projectOneTaskOne()
```

volumeViewer.py:

```
# % Class to create interactive 3D image/mask viewer
# % EECE 8396: Medical Image Segmentation
# % Spring 2024
# % Author: Prof. Jack Noble; jack.noble@vanderbilt.edu
#

# % Example usage:
# %   >> d = volumeViewer()
# %   >> d.setImage(mr, voxsz)
# %   >> d.display()
# %       % Displays the 3D image in image np 3D array mr with voxel size
voxsz
# %
# %   >> d.setImage(mr,voxsz, contrast=1000, level=0)
# %   >> d.display()
# %       % Displays the 3D image in image struct mr and adjusts the
# %       intensity contrast and window level
# %
# %   >> d.update(direction=0,slc=20)
# %   >> d.display()
# %       % For a currently displayed 3D image, changes the sagittal view (0)
to
# %       slice 20
# %
# %   >> d.setImage(mr, voxsz)
# %   >> d.addMask(segmsk,color=[0,1,1],opacity=0.5, label =
'Mysegmentation')
# %   >> d.display()
# %       % Displays the 3D image in  mr then overlays aqua
# %       colored contours of a segmentation mask in the segmsk struct on the
# %       3D views and displays a 3D isosurface of the mask with 0.5 opacity
# %       in the 3D viewer window


import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backend_bases import MouseButton
from skimage import measure
from myVTKWin import *

class imagevolume:
    def __init__(self,data=0,voxsz=[1,1,1]):
        self.data = data
        self.voxsz = voxsz

class contourclass:
    def __init__(self, data):
        self.data = data

class object:
```

```
    def __init__(self, type, data, color=[1.0,0.0,0.0], opacity = 1.0):
        self.type = type
        self.data = data
        self.color = color
        self.opacity = opacity
        self.ms = None

class mask:
    def __init__(self, data, voxsz=[1.0,1.0,1.0], color=[1.0,0.0,0.0],
label=''):
        self.data = np.asarray(data)
        self.voxsz = np.asarray(voxsz)
        self.color = np.asarray(color)
        self.label = label
        self.cntrs = np.zeros([3,max(np.shape(data))],dtype=contourclass)

    def updateContours(self, win, opacity=1.0):
        dim = np.shape(self.data)
        X,Y = np.meshgrid(np.linspace(0,dim[0]-1,dim[0]),
np.linspace(0,dim[1]-1,dim[1]))
        for i in range(np.shape(self.data)[2]):
            if np.min(self.data[:,:,i])<0.5 and np.max(self.data[:,:,i]) >
0.5:
                cntr =
plt.contour(X,Y,np.transpose(self.data[:,:,i]),levels=[0.5])
                self.cntrs[0][i] = contourclass(cntr.allsegs[0])
        X, Y = np.meshgrid(np.linspace(0, dim[0] - 1, dim[0]), np.linspace(0,
dim[2] - 1, dim[2]))
        for i in range(np.shape(self.data)[1]):
            if np.min(self.data[:, i, :]) < 0.5 and np.max(self.data[:,
i, :]) > 0.5:
                cntr = plt.contour(X, Y, np.transpose(np.squeeze(self.data[:,
i, :])), levels=[0.5])
                self.cntrs[1][i] = contourclass(cntr.allsegs[0])
        X, Y = np.meshgrid(np.linspace(0, dim[1] - 1, dim[1]), np.linspace(0,
dim[2] - 1, dim[2]))
        for i in range(np.shape(self.data)[0]):
            if np.min(self.data[i, :, :]) < 0.5 and
np.max(self.data[i, :, :]) > 0.5:
                cntr = plt.contour(X, Y,
np.transpose(np.squeeze(self.data[i, :, :])), levels=[0.5])
                self.cntrs[2][i] = contourclass(cntr.allsegs[0])

        verts,faces,_,_ = measure.marching_cubes(self.data,0.5,
spacing=self.voxsz)
        win.addSurf(verts,faces,color=self.color, opacity=opacity)


class volumeViewer(myVtkWin):
    def __init__(self, title='Volume Viewer (Press Esc to quit)'):
        super().__init__(title="3D display")
        self.img = None
        self.objs = []
        self.slc = [0,0,0]
        self.contrast = 1
        self.level = 0
        plt.ion()
```

```
        self.fig = plt.figure()
        self.fig.suptitle(title, fontsize=16)
        self.ax = np.zeros([2,2], dtype=plt.Axes)
        self.ax[0,0] = self.fig.add_subplot(2,2,1)
        self.ax[0,1] = self.fig.add_subplot(2,2,2)
        self.ax[1,0] = self.fig.add_subplot(2,2,3)
        plt.axes(self.ax[0,0])

        self.quit = False
        self.focus = -1;
        binding_id2 = plt.connect('button_press_event',self.onMouseClick)
        binding_id3 = plt.connect('key_press_event',self.onKeyPress)

    def onMouseClick(self,event):
        if event.dblclick:
            if event.button is MouseButton.LEFT:
                for i in range(0,3):
                    if event.inaxes == self.ax[i//2, i%2]:
                        self.focus = i
                if self.focus == 2:
                    pnt = [event.xdata, event.ydata, self.slc[2]]
                elif self.focus == 1:
                    pnt = [event.xdata, self.slc[1], event.ydata]
                elif self.focus == 0:
                    pnt = [self.slc[0], event.xdata, event.ydata]
                else:
                    return
                self.centerOnPoint(pnt)
            if event.button is MouseButton.RIGHT:
                if event.inaxes == self.ax[1,0]:
                    self.ax[1,0].set_xlim(left=0, right=np.shape(self.img)[0]
- 1)
                    self.ax[1,0].set_ylim(bottom=np.shape(self.img)[1] -
1,top=0)
                elif event.inaxes == self.ax[0,1]:
                    self.ax[0,1].set_xlim(left=0, right=np.shape(self.img)[0]
- 1)
                    self.ax[0,1].set_ylim(top=np.shape(self.img)[2] -
1,bottom=0)
                elif event.inaxes == self.ax[0,0]:
                    self.ax[0,0].set_xlim(left=0, right=np.shape(self.img)[1]
- 1)
                    self.ax[0,0].set_ylim(top=np.shape(self.img)[2] -
1,bottom=0)


    def centerOnPoint(self,pnt):
        pnt = np.copy(pnt)
        for i in range(3):
            if pnt[i]<0:
                pnt[i]=0
            elif pnt[i]>np.shape(self.img)[i]-1:
                pnt[i] = np.shape(self.img)[i]-1
            self.slc[i] = round(pnt[i])
            self.update(i)
        for i in range(0,3):
            xlim = self.ax[i//2,i%2].get_xlim()
```

```
            ylim = self.ax[i//2,i%2].get_ylim()
            xrng = xlim[1] - xlim[0]
            yrng = ylim[1] - ylim[0]
            if i==0:
                x = pnt[1]
                y = pnt[2]
            elif i==1:
                x = pnt[0]
                y = pnt[2]
            else:
                x = pnt[0]
                y = pnt[1]
            self.ax[i//2,i%2].set_xlim(x-xrng/2,x+xrng/2)
            self.ax[i//2,i%2].set_ylim(y-yrng/2,y+yrng/2)
            plt.axes(self.ax[i//2,i%2])
            plt.plot([x, x], [y + 0.02 * yrng, y - 0.02 * yrng], 'r')
            plt.plot([x + 0.02 * xrng, x - 0.02 * xrng], [y, y], 'r')

    def keypress_callback(self,obj,ev): # overloads myVTKWin key press
callback function
        key = super().keypress_callback(obj,ev)
        if (key == 'u' or key == 'U'):
            pnt = self.lastpickpos / self.voxsz
            self.centerOnPoint(pnt)

    def onKeyPress(self,event):# for matplotlib window
        if event.key == 'escape' or event.key=='q' or event.key=='Q':
            self.quit = True

        # Paging through slices
        if event.key in ['up', 'a']:
            self.slc[self.focus] = min(self.slc[self.focus] + 1,
np.shape(self.img)[self.focus] - 1)
        elif event.key in ['down', 'z']:
            self.slc[self.focus] = max(self.slc[self.focus] - 1, 0)

        # Adjusting window level
        if event.key == 'd':
            self.level += 0.1 * self.contrast
        elif event.key == 'x':
            self.level -= 0.1 * self.contrast

        # Adjusting contrast
        if event.key == 'c':
            self.contrast *= 1.1
        elif event.key == 'v':
            self.contrast *= 0.9

        # Trigger an update to refresh the display with new slice, level, or
contrast
        self.update()

    def
setImage(self,img,voxsz,contrast=1000,level=0,interpolation='bilinear',autoco
ntrast=True):
        self.img = np.asarray(img)
        self.voxsz = np.asarray(voxsz)
```

```
        self.slc = np.array(np.shape(img.data), dtype=int) //2
        self.contrast=contrast
        self.level = level
        self.interpolation = interpolation
        if autocontrast:
            self.autoContrast()
        self.update()

    def autoContrast(self):
        mn = np.amin(self.img)
        mx = np.amax(self.img)
        bns = np.linspace(mn,mx,256)
        h,_ = np.histogram(self.img.ravel(), bins=bns)
        h = h.astype(np.float32)/np.sum(h)
        mini = 0
        tot = h[0]
        while tot<0.1:
            mini+=1
            tot += h[mini]

        maxi = mini
        while tot<0.99:
            maxi +=1
            tot += h[maxi]

        self.contrast = (maxi - mini) * (mx - mn) / 256.0
        self.level = 0.5 * (maxi + mini) * (mx - mn) / 256.0 + mn

    def addMask(self,msk,color = [1.0,0.0,0.0], opacity=1.0, label=''):
        mskobj = mask(msk, self.voxsz, color,label)
        mskobj.updateContours(self, opacity=opacity)
        obj = object(1,mskobj,color=color,opacity=opacity)
        self.objs.append(obj)

    def update(self,direction = -1,slc = -1,level = float("nan"),contrast =
float("nan"),resize = -1):
        if (not np.isnan(level)):
            self.level = level
        if (not np.isnan(contrast)):
            self.contrast = contrast
        if (resize != -1):
            self.resize = resize
        if direction == -1:
            for i in range(4):
                self.update(direction = i)
        else:
            if slc >= 0:
                self.slc[direction] = slc
            if direction <3:
                plt.figure(self.fig)
                plt.axes(self.ax[direction // 2, direction % 2])
                xlim = self.ax[direction//2, direction%2].get_xlim()
                ylim = self.ax[direction//2, direction%2].get_ylim()
                plt.cla()
                if direction == 2:
                    self.ax[direction//2,
direction%2].imshow(np.transpose(self.img[:, :, self.slc[direction]]),
```

```
'gray', interpolation=self.interpolation,

vmin=self.level - self.contrast/2,

vmax=self.level + self.contrast/2)
                    plt.xlabel('x')
                    plt.ylabel('y')
                    self.ax[direction//2,
direction%2].set_aspect(self.voxsz[1]/self.voxsz[0])
                    if xlim[1] != 1:
                        self.ax[direction//2,
direction%2].set_xlim(left=xlim[0], right=xlim[1])
                        self.ax[direction//2,
direction%2].set_ylim(bottom=ylim[0], top=ylim[1])
                    else:
                        self.ax[direction//2, direction%2].set_xlim(left=0,

right=np.shape(self.img)[0] - 1)
                        self.ax[direction//2,
direction%2].set_ylim(bottom=np.shape(self.img)[1] - 1,

top=0)
                    vstr = 'Axial view'
                    z = 'z'
                    for i in range(len(self.objs)):
                        if self.objs[i].type==1 and
self.objs[i].data.cntrs[0][self.slc[2]]:
                            for j in
range(len(self.objs[i].data.cntrs[0][self.slc[2]].data)):

plt.plot(self.objs[i].data.cntrs[0][self.slc[2]].data[j][:, 0],

self.objs[i].data.cntrs[0][self.slc[2]].data[j][:, 1],
color=self.objs[i].color)
                elif direction == 1:
                    self.ax[direction//2,
direction%2].imshow(np.transpose(np.squeeze(self.img[:,
self.slc[direction], :])),

'gray', interpolation=self.interpolation,

vmin=self.level - self.contrast/2, vmax=self.level + self.contrast/2)
                    plt.xlabel('x')
                    plt.ylabel('z')
                    self.ax[direction//2,
direction%2].set_aspect(self.voxsz[2] / self.voxsz[0])
                    if xlim[1] != 1:
                        self.ax[direction//2,
direction%2].set_xlim(left=xlim[0], right=xlim[1])
                        self.ax[direction//2,
direction%2].set_ylim(bottom=ylim[0], top=ylim[1])
                    else:
                        self.ax[direction//2, direction%2].set_xlim(left=0,

right=np.shape(self.img)[0] - 1)
                        self.ax[direction//2, direction%2].set_ylim(bottom=0,
```

```
top=np.shape(self.img)[2] - 1)
                    vstr = 'Coronal view'
                    z = 'y'
                    for i in range(len(self.objs)):
                        if self.objs[i].type==1 and
self.objs[i].data.cntrs[1][self.slc[1]]:
                            for j in
range(len(self.objs[i].data.cntrs[1][self.slc[1]].data)):

plt.plot(self.objs[i].data.cntrs[1][self.slc[1]].data[j][:, 0],

self.objs[i].data.cntrs[1][self.slc[1]].data[j][:, 1],
color=self.objs[i].color)
                elif direction == 0:
                    self.ax[direction//2,
direction%2].imshow(np.transpose(np.squeeze(self.img[self.slc[direction], :,
:])),

'gray', interpolation=self.interpolation,

vmin=self.level - self.contrast/2, vmax=self.level + self.contrast/2)
                    plt.xlabel('y')
                    plt.ylabel('z')
                    self.ax[direction//2,
direction%2].set_aspect(self.voxsz[2] / self.voxsz[1])
                    if xlim[1] != 1:
                        self.ax[direction//2,
direction%2].set_xlim(left=xlim[0], right=xlim[1])
                        self.ax[direction//2,
direction%2].set_ylim(bottom=ylim[0], top=ylim[1])
                    else:
                        self.ax[direction//2, direction%2].set_xlim(left=0,
right=np.shape(self.img)[1]-1)
                        self.ax[direction//2, direction%2].set_ylim(bottom=0,
top=np.shape(self.img)[2]-1)
                    vstr = f'Sagittal view Contrast = {self.contrast:.1f}
Level = {self.level:.1f}'
                    z = 'x'
                    for i in range(0,np.size(self.objs)):
                        if self.objs[i].type==1 and
self.objs[i].data.cntrs[2][self.slc[0]]:
                            for j in
range(len(self.objs[i].data.cntrs[2][self.slc[0]].data)):

plt.plot(self.objs[i].data.cntrs[2][self.slc[0]].data[j][:, 0],

self.objs[i].data.cntrs[2][self.slc[0]].data[j][:, 1],
color=self.objs[i].color)


                plt.title(f'{vstr}: Slice {z} = {self.slc[direction]}')
            elif direction == 3:
                self.render()

    def repaint(self):
        self.fig.canvas.draw_idle()
```

```
        self.fig.canvas.start_event_loop(0.3)

    def display(self,blocking=True):
        self.update()
        if blocking:
            while (self.quit == False):
                self.repaint()
        if self.quit:
            del self

    def __del__(self):
        plt.close(self.fig)
        super().__del__()
```