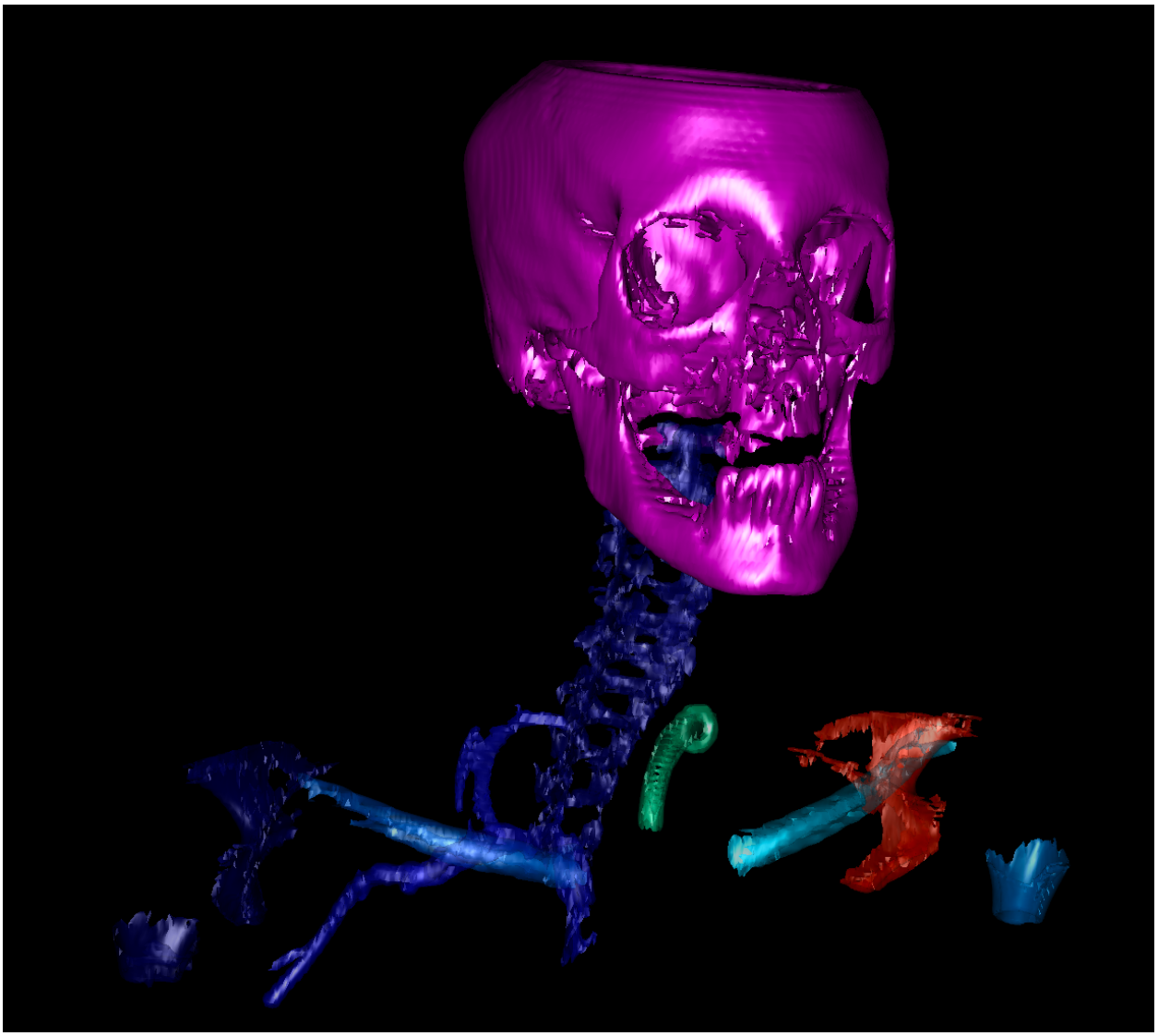

20240129_ConnectedComponents.py

```
import nrrd
from Project.volumeViewer import *
from Project.surface import *
import numpy as np
import matplotlib as mp
import nibabel as nib

# load a CT image to play with
img, imgh = 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/Imp0001-
Decompressed_CT_0_2.nii'
# Load the NIfTI file
# nifti_file = nib.load(file_path)
# Get the data from the file
# img = nifti_file.get_fdata()
```

```
file_path = '/Users/leonslaptop/Desktop/2024 Spring/Research/Pelvis/head-
NIFTI/head-Decompressed_CT_0_1.nii'
voxsz = [imgh['space directions'][0][0], imgh['space directions'][1][1],
imgh['space directions'][2][2]]

d = volumeViewer()
d.setImage(img, voxsz, contrast=1500, level=500, autocontrast=False)
d.update(direction=2, slc=78)
# d.display()

imgzp = np.zeros(np.array(img.shape)+2)
imgzp[1:-1,1:-1,1:-1] = img

s = surface()
createSurfaceFromVolume(s, imgzp, voxsz, isolevel=700)

# undo zero padding
s.verts[:,0] -= voxsz[0]
s.verts[:,1] -= voxsz[1]
s.verts[:,2] -= voxsz[2]

buildGraph(s)
surfaces = connectedComponents(s)
numsurf = np.size(surfaces)
print(f'Found {numsurf} surfaces')

vols = np.zeros(numsurf)
for i in range(numsurf):
    vols[i] = volume(surfaces[i])

maxvol = np.max(vols)
imax = np.argmax(vols)
print(f'Surface {imax} has max volume {maxvol} mm3')

win2 = myVtkWin()

#show largest component in magenta
win2.addSurf(surfaces[imax].verts, surfaces[imax].faces, color=[1,0,1],
opacity=1.0)

cols = mp.colormaps['jet']
for i in range(numsurf):
    if i!=imax and vols[i] > 1000:
        win2.addSurf(surfaces[i].verts, surfaces[i].faces,
                     color=cols(i % 256)[0:3], opacity=0.5)

win2.start()
```

surface.py

```
from Project.volumeViewer import *

class GraphNode:
    def __init__(self, vertex_id):
        self.id = vertex_id
```

```
        self.neighbors = []

# surface class
class surface:
    def __init__(self):
        self.verts = []
        self.faces = []
        self.graph = []

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

    # load CT image
    img, header = nrrd.read('/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

# Function to visualize the surface using VTK
def visualizeSurface(s):
    win = myVtkWin()
    win.addSurf(s.verts, s.faces, color=[1, 0.9, 0.8])
    win.start()

def buildGraph(self):
    # Initialize nodes for all vertices
    for i in range(len(self.verts)):
        self.graph.append(GraphNode(i))

    # Add edges based on faces
    for face in self.faces:
        for i, vertex_id in enumerate(face):
            # Add edge between current vertex and the next vertex in the face
(forming edges of the triangle)
            next_vertex_id = face[(i + 1) % len(face)]
            if next_vertex_id not in self.graph[vertex_id].neighbors:
```

```
                self.graph[vertex_id].neighbors.append(next_vertex_id)
                self.graph[next_vertex_id].neighbors.append(vertex_id)

def bfs(self, start, visited):
    from collections import deque
    queue = deque([start])
    component = []
    while queue:
        vertex_id = queue.popleft()
        if not visited[vertex_id]:
            visited[vertex_id] = True
            component.append(vertex_id)
            for neighbor_id in self.graph[vertex_id].neighbors:
                if not visited[neighbor_id]:
                    queue.append(neighbor_id)
    return component

def connectedComponents(self):
    # Initialize Marked, labels, maxlabel=0
    num_vertices = len(self.verts)
    Marked = [False] * num_vertices
    labels = [-1] * num_vertices
    maxlabel = 0

    nodes = self.graph

    # Function for graph traversal and marking the connected component
    def markComponent(start):
        nonlocal maxlabel
        queue = [start]
        labels[start] = maxlabel
        Marked[start] = True
        while queue:
            current_vertex_id = queue.pop(0)
            current_node = nodes[current_vertex_id]
            for neighbor in current_node.neighbors:
                if not Marked[neighbor]:
                    Marked[neighbor] = True
                    labels[neighbor] = maxlabel
                    queue.append(neighbor)

    # While there are unmarked vertices, perform a graph traversal
    for n in range(num_vertices):
        if not Marked[n]:
            markComponent(n)
            maxlabel += 1

    # Initialize containers
    label_to_vertices = {i: [] for i in range(max(labels) + 1)}
    label_to_faces = {i: [] for i in range(max(labels) + 1)}

    # Map vertices to their labels
    for vertex_index, label in enumerate(labels):
        label_to_vertices[label].append(vertex_index)

    # Iterate over faces to map them to labels
    for face_index, face in enumerate(self.faces):
```

```
        vertex_label = labels[face[0]]  # Assuming all vertices in a face
share the same label
        label_to_faces[vertex_label].append(face_index)

    # Create components
    H = []
    for label, verts_indices in label_to_vertices.items():
        faces_indices = label_to_faces[label]

        # Assuming createComponent can handle lists of indices directly
        component = createComponent(self, verts_indices, faces_indices)
        H.append(component)
    return H

def createComponent(surfaceObj, verts_indices, faces_indices):
    new_component = surface()

    # Directly use numpy array, avoid converting to list unless necessary for
downstream operations
    new_component.verts = surfaceObj.verts[verts_indices]

    # Mapping remains efficient as is
    new_indices_map = {old_idx: new_idx for new_idx, old_idx in
enumerate(verts_indices)}

    # Remap faces to new vertex indices, this part is already quite efficient
    remapped_faces = [[new_indices_map[vertex] for vertex in face] for face
in
                      surfaceObj.faces[faces_indices]]
    new_component.faces = remapped_faces
    return new_component

def volume(self):
    # Ensure verts and faces are not None
    if self.verts is None or self.faces is None:
        raise ValueError("Surface vertices and faces must be defined.")

    # Calculate volume
    volume = 0.0
    for face in self.faces:
        v0, v1, v2 = self.verts[face]
        # The volume contribution of the tetrahedron formed by face and
origin
        tetra_volume = np.dot(v0, np.cross(v1, v2)) / 6.0
        volume += tetra_volume

    # Absolute value to ensure positive volume, in case of inverted normals
    return abs(volume)
```

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('/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

# Function to visualize the surface using VTK
def visualizeSurface(s):
    win = myVtkWin()
    win.addSurf(s.verts, s.faces, color=[1, 0.9, 0.8])
    win.start()

def connectedComponents(self):
    from scipy.sparse.csgraph import connected_components
    from scipy.sparse import csr_matrix

    # Create adjacency matrix for faces
    edges = np.vstack([self.faces[:, [0, 1]], self.faces[:, [1, 2]],
self.faces[:, [2, 0]]])
    edges = np.sort(edges, axis=1)  # Sort the vertex pairs
    edge_hash = edges[:, 0] * max(self.faces.flatten()) + edges[:, 1]  #
Unique identifier for edges

    # Create sparse matrix with shape (n_vertices, n_vertices)
    graph = csr_matrix((np.ones(len(edge_hash)), (edges[:, 0], edges[:, 1])),
                       shape=(len(self.verts), len(self.verts)))
    graph = graph + graph.T  # Make sure the graph is symmetric

    # Find connected components
    n_components, labels = connected_components(csgraph=graph,
directed=False, return_labels=True)

    # Separate components
    components = []
    for i in range(n_components):
        print(f"Processing component {i + 1}/{n_components}")
        component_verts_indices = np.where(labels == i)[0]
        component_faces = []

        # Filter faces where all three vertices belong to the current
component
        for face in self.faces:
            if all(vertex in component_verts_indices for vertex in face):
                component_faces.append(face)

        if component_faces:
            # Map old vertex indices to new ones in the component
            new_indices_map = {old_idx: new_idx for new_idx, old_idx in
                               enumerate(component_verts_indices)}
            component_faces = np.array(
                [[new_indices_map[vertex] for vertex in face] for face in
component_faces])

            new_component = surface()
            new_component.verts = self.verts[component_verts_indices]
            new_component.faces = component_faces
            components.append(new_component)

    return components
```

```
def visualizeComponents(components, win):
    # Generate a broad range of colors by cycling through RGB values
    def generate_color(i):
        r = (i % 256) / 255.0
        g = ((i // 256) % 256) / 255.0
        b = ((i // (256 * 256)) % 256) / 255.0
        return [r, g, b]

    for i, comp in enumerate(components):
        color = generate_color(i)
        win.addSurf(comp.verts, comp.faces, color=color, opacity=1.0)

if __name__ == "__main__":

    # demoPointsAndLines()
    # demoSurfaceAppearance()
    # demoSurfaceEdgesAndColors()
    # demoDepthOfField()
    # brainPointPick()
    # brainAnimation()
    # bouncingBallsAnimation()
    # demoSurfaceFromNRRD()
    # projectOneTaskOne()
    # Load CT data, generate surface, and extract connected components
    filePath = ('/Users/leonslaptop/Desktop/2024 Spring/ECE
3892/data/0522c0001/img.nrrd')
    isolevel = 700
    print("Loading and processing structure...")
    s = loadAndProcessStructure(filePath, isolevel)
    print(f"Surface loaded with {len(s.verts)} vertices and {len(s.faces)}
faces.")

    print("Extracting connected components...")
    components = connectedComponents(s)
    print(f"Found {len(components)} components.")

    # Initialize visualization window
    win = myVtkWin(title="Connected Components Visualization")
    print("Visualizing components...")
    visualizeComponents(components, win)

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