Aidan Fischer 4/7/2022

CS 532 HW 3 I pledge my honor that I have abided by the Stevens Honor System.

Source Code

main.py – This homework only had one file, so I decided to put some functionality in main.py instead of offloading all function code to other files and keeping only function calls in main. This contains the function that loads the camera projection matrices from the xml files, the main loop over the voxel grid to create the model, and the model output function.

```
# Load a projection matrix from an xml file in the calibration folder.
def load_P_from_xml(file):
    tree = ET.parse(file)
    root = tree.getroot()
    Ptext = root.text
    Psplit = Ptext.split(" ")
    Pvals = []
    for elem in Psplit:
        if elem == "":
            pass
        else:
            Pvals.append(float(elem))
    P = np.matrix(np.array(Pvals).reshape((3, 4)))
    return P


# For every voxel in the provided grid, determine
# whether it is occupied or free, and whether it
# is a boundary voxel. Returns a list of boundary
# voxels.
def process_voxels(V, cameras):
    boundary_voxels = set()
    occupied_indices = np.zeros(V.grid.shape[:3], np.bool8)
    for i in trange(V.grid.shape[0], desc="Processing..."):
        for j in trange(V.grid.shape[1], leave=False):
            for k in range(V.grid.shape[2]):
                free = False
                X, Y, Z = V.index_to_coords(i, j, k)
                for c in cameras:
                    # If voxel is not in the silhouette of a camera, it
                    # is free. No need to check rest of cameras.
                    if not c.get_silhouette_presence(X, Y, Z):
                        free = True
                        break
                # If all cameras have this voxel in their silhouette, it
```

```
                # is occupied.
                if not free:
                    V.set_voxel(X, Y, Z, [255, 255, 255])
                    occupied_indices[i, j, k] = True
                # Check for difference between this voxel and priors,
                # in order to set boundary voxels. Only check:
                #   i - i, j ,k
                #   i, j - 1, k
                #   i, j, k - 1
                # Since these have already been checked for free/occupied.
                # If any of these 3 are of a different state than this voxel
                # then set all of the different ones, as well as this voxel,
                # to a boundary. Explained further in PDF
                r1, g1, b1 = V.grid[i, j, k]

                if free != (not occupied_indices[i - 1, j, k]) and i > 0:
                    r2, g2, b2 = V.grid[i - 1, j, k]
                    boundary_voxels.add((i, j, k, r1, g1, b1))
                    boundary_voxels.add((i - 1, j, k, r2, g2, b2))
                if free != (not occupied_indices[i, j - 1, k]) and j > 0:
                    r2, g2, b2 = V.grid[i, j - 1, k]
                    boundary_voxels.add((i, j, k, r1, g1, b1))
                    boundary_voxels.add((i, j - 1, k, r2, g2, b2))
                if free != (not occupied_indices[i, j, k - 1]) and k > 0:
                    r2, g2, b2 = V.grid[i, j, k - 1]
                    boundary_voxels.add((i, j, k, r1, g1, b1))
                    boundary_voxels.add((i, j, k - 1, r2, g2, b2))
    return boundary_voxels


# Creates the PLY file for the boundary voxel array provided
def create_ply(V, boundary, cameras):
    b = list(boundary)
    # Discard free boundary voxels
    b = list(filter(lambda elem: not (elem[3] == 0 and elem[4] == 0
                    and elem[5] == 0), b))
    b_colored = []
    print("Determining point colors.")
    for voxel in tqdm(b):
        i, j, k = voxel[0], voxel[1], voxel[2]
        X, Y, Z = V.index_to_coords(i, j, k)
        r, g, bl = V.calc_color(X, Y, Z, cameras)
        b_colored.append((i, j, k, r, g, bl))
    print("Creating PLY file.")
    with open("../Output/output.ply", "w+") as plyf:
```

```
        plyf.write("ply\n")
        plyf.write("format ascii 1.0\n")
        plyf.write(f"element vertex {len(b_colored)}\n")
        plyf.write("property float x\n")
        plyf.write("property float y\n")
        plyf.write("property float z\n")
        plyf.write("property uchar red\n")
        plyf.write("property uchar green\n")
        plyf.write("property uchar blue\n")
        plyf.write("element face 0\n")
        plyf.write("end_header\n")
        for voxel in b_colored:
            X, Y, Z = V.index_to_coords(voxel[0], voxel[1], voxel[2])
            r, g, b = voxel[3], voxel[4], voxel[5]
            plyf.write("{0} {1} {2} {3} {4} {5}\n".format(float(X), float(Y),
                                                          float(Z), int(r),
                                                          int(g), int(b)))
        plyf.write("\n")


def main():
    # Define images
    Ps = []
    images = []
    silhouettes = []
    cameras = []
    print("Loading projection matrices.")
    for i in range(8):
        P = load_P_from_xml(f"../calibration/cam0{i}.xml")
        Ps.append(P)
    print("Loading images.")
    for i in range(8):
        image = Image.get_image(f"../cam0{i}_00023_0000008550.png")
        images.append(image)
    print("Loading silhouettes.")
    for i in range(8):
        silhouette = Image.get_image(f"../silh_cam0{i}_00023_0000008550.pbm")
        silhouettes.append(silhouette)
    print("Creating camera objects.")
    for i in range(8):
        camera = Camera(Ps[i], images[i], silhouettes[i])
        cameras.append(camera)
    print("Creating voxel grid.")
    V = VoxelGrid()
    print("Processing voxels.")
```

```
    boundary = process_voxels(V, cameras)
    create_ply(V, boundary, cameras)
    print("Done.")


if __name__ == "__main__":
    main()
```

Image.py – My personal PIL/numpy abstraction utility functions

```
def get_image(path, gray=False):
    image = Image.open(path)
    if gray:
        image = ImageOps.grayscale(image)
    return np.array(image)


def save_image(image_array, name):
    image = Image.fromarray(image_array)
    image.convert("RGB").save(name)
    return


def copy_image(image):
    return np.copy(image)


def to_array(PILimage):
    return np.array(PILimage)
```

Camera.py – Defines a camera class that provides access methods between a voxel coordinate and a pixel in that camera's image through a project function. Mechanics of project explained below.

```
import numpy as np
from sympy import Matrix
# Uses projection matrix to convert meter coordinates
# to an x,y point, and reverse. Also provides accessor
# functions to get information from the camera's image
# and silhouette from voxel coordinates.
class Camera():
    def __init__(self, P, image, silhouette):
        self.P = P
        self.image = image
        self.silhouette = silhouette
        C = Matrix(P)
```

```
        C = C.nullspace()[0]
        C = C/C[3]
        self.center = np.matrix(C)

    # Returns X, Y coordinates of voxel (in meters)
    # in camera's image. Does NOT check image bounds,
    # but does check voxel bounds.
    @jit(types.UniTuple(int64, 2)(float64[:, :], float64, float64, float64),
         nopython=True, cache=True)
    def project(P, x, y, z):
        if(x < -2.5 or x > 2.5 or y < -3
           or y > 3 or z < 0 or z > 2.5):
            print("Error: Voxel coordinates outside range. ")
            return (-1, -1)
        pt = np.array([x, y, z, 1])
        # Some less efficient matrix mult to allow for numba to
        # significantly increase overall efficiency by compiling
        # this function to machine code
        # Speedup is about 3.5x
        homogenous_coords = [np.sum(np.multiply(P[0, :], pt)),
                             np.sum(np.multiply(P[1, :], pt)),
                             np.sum(np.multiply(P[2, :], pt))]
        X = int(homogenous_coords[0] / homogenous_coords[2])
        Y = int(homogenous_coords[1] / homogenous_coords[2])
        return X, Y

    # Returns true if voxel is in this camera's silhouette
    # false if not. x, y, z in meters.
    # Returns None if pixel is outside image bounds
    def get_silhouette_presence(self, x, y, z):
        if(x < -2.5 or x > 2.5 or y < -3
           or y > 3 or z < 0 or z > 2.5):
            print("Error: Voxel coordinates outside range. ")
            return None
        X, Y = Camera.project(self.P, x, y, z)
        if(X >= self.silhouette.shape[1] or X < 0 or
           Y >= self.silhouette.shape[0] or Y < 0):
            return None
        silhouettept = self.silhouette[Y, X]
        if(silhouettept > 0):
            return True
        return False

    # Gets the color of the pixel corresponding to the provided
    # voxel. x, y, z in meters.
```

```
    # Returns None if pixel is outside image bounds
    def get_image_color(self, x, y, z):
        if(x < -2.5 or x > 2.5 or y < -3
           or y > 3 or z < 0 or z > 2.5):
            print("Error: Voxel coordinates outside range. ")
            return None
        X, Y = Camera.project(self.P, x, y, z)
        if(X >= self.image.shape[1] or X < 0 or
           Y >= self.image.shape[0] or Y < 0):
            return None
        return self.image[Y, X]
```

VoxelGrid.py – Defines the voxel grid that stores the final model, as well as intermediary functions that allow modifying and accessing the grid through physical space locations. Provides a function that determines a voxel color from provided cameras.

```
# Side length of voxels in meters.
# 0.002 Used in final
# 0.025 Used in testing for speed
VOXEL_DIMENSION = 0.002


def vec_len(vec):
    tot = 0
    for p in vec:
        tot += p * p
    return math.sqrt(tot)


# Creates a voxel grid accessible using meter-based coordinates
# Size of grid is 5m by 6m by 2.5m. x and y access based on
# center (x:-2.5 to 2.5, y:-3 to 3), z based on 0 (z:0 to 2.5)
class VoxelGrid():
    def __init__(self):
        self.grid = np.zeros((math.ceil(5/VOXEL_DIMENSION),
                              math.ceil(6/VOXEL_DIMENSION),
                              math.ceil(2.5/VOXEL_DIMENSION),
                              3), np.uint8)

    # Converts voxel locations in the form (X, Y, Z), where X, Y, and Z are in
    # meters, to array values and returns the voxel value at that location.
    def get_voxel(self, voxel_x, voxel_y, voxel_z):
        if(voxel_x < -2.5 or voxel_x > 2.5 or voxel_y < -3
           or voxel_y > 3 or voxel_z < 0 or voxel_z > 2.5):
            print("Error: Voxel coordinates outside range. ")
```

```
            return None
        grid_x = math.floor((voxel_x+2.5)/VOXEL_DIMENSION)
        grid_y = math.floor((voxel_y+3)/VOXEL_DIMENSION)
        grid_z = math.floor((voxel_z)/VOXEL_DIMENSION)
        # If grid points are exactly at far edge, get actual last voxel
        # instead of nonexistent one.
        if voxel_x == 2.5:
            grid_x -= 1
        if voxel_y == 3:
            grid_y -= 1
        if voxel_z == 2.5:
            grid_z -= 1
        return self.grid[grid_x, grid_y, grid_z]

    # Converts voxel locations in the form (X, Y, Z), where X, Y, and Z are in
    # meters, to array values and sets the value of the voxel at that location
    # new_value is of the form [r,g,b]
    def set_voxel(self, voxel_x, voxel_y, voxel_z, new_value):
        if(voxel_x < -2.5 or voxel_x > 2.5 or voxel_y < -3
           or voxel_y > 3 or voxel_z < 0 or voxel_z > 2.5):
            print("Error: Voxel coordinates outside range. ")
            return None
        grid_x = math.floor((voxel_x+2.5)/VOXEL_DIMENSION)
        grid_y = math.floor((voxel_y+3)/VOXEL_DIMENSION)
        grid_z = math.floor((voxel_z)/VOXEL_DIMENSION)
        # If grid points are exactly at far edge, get actual last voxel
        # instead of nonexistent one.
        if voxel_x == 2.5:
            grid_x -= 1
        if voxel_y == 3:
            grid_y -= 1
        if voxel_z == 2.5:
            grid_z -= 1
        self.grid[grid_x, grid_y, grid_z] = new_value

    # Converts grid indices to voxel coordinates in meters.
    # Since this object expects to be accessd by meter-based
    # coordinates, this is necessary if looping through the grid.
    # Loses some precision, but that is inherent in a grid of voxels.
    # Returns coords of center of voxel.
    def index_to_coords(self, i, j, k):
        X = i * VOXEL_DIMENSION - 2.5 + VOXEL_DIMENSION / 2
        Y = j * VOXEL_DIMENSION - 3 + VOXEL_DIMENSION / 2
        Z = k * VOXEL_DIMENSION + VOXEL_DIMENSION / 2
        # Necessary if voxel size doesn't evenly divide the grid size.
```

```
        X = 2.5 if X > 2.5 else X
        Y = 3 if Y > 3 else Y
        Z = 2.5 if Z > 2.5 else Z
        return X, Y, Z

    # Determine voxel color from cameras. Uses crude estimation of
    # ray marching.
    def calc_color(self, voxel_x, voxel_y, voxel_z, cameras):
        if(voxel_x < -2.5 or voxel_x > 2.5 or voxel_y < -3
           or voxel_y > 3 or voxel_z < 0 or voxel_z > 2.5):
            print("Error: Voxel coordinates outside range. ")
            return None

        color_cams = []

        for cam in cameras:
            # Obtain a vector from the provided coords to the camera center
            vec_pt_to_cam = (cam.center - [[voxel_x], [voxel_y],
                             [voxel_z], [1]])[:3]
            vec_pt_to_cam = vec_pt_to_cam.astype(np.float32)
            # Set indicator color (so we dont discard a camera for intersecting
            # source voxel)
            self.set_voxel(voxel_x, voxel_y, voxel_z, [128, 128, 128])
            # Set length of vector to 2/3 * voxel dimension.
            vec_pt_to_cam = vec_pt_to_cam / (vec_len(vec_pt_to_cam)) * \
                2 / 3 * VOXEL_DIMENSION
            # Crude approximation of ray-march using resized
            # vec_pt_to_cam as step. Misses some corners, but
            # I hope effect of this is decreased as model
            # resolution increases.
            pt = np.array([[voxel_x], [voxel_y],
                           [voxel_z]]).astype(np.float32)
            seen = True
            while(pt[0, 0] >= -2.5 and pt[0, 0] <= 2.5
                  and pt[1, 0] >= -3 and pt[1, 0] <= 3
                  and pt[2, 0] >= 0 and pt[2, 0] <= 2.5):
                here = self.get_voxel(pt[0, 0], pt[1, 0], pt[2, 0])
                # If voxel is free or the indicator, increment ray-
                # march and continue.
                if np.array_equal(here, [128, 128, 128]) or \
                   np.array_equal(here, [0, 0, 0]):
                    pt += vec_pt_to_cam
                    continue
                # If voxel is any other color, that means it is occupied.
                # Either [255, 255, 255], internal or no assigned color,
```

```
                # or an assigned color (there is prevention of setting
                # an occupied and color-assigned voxel to [128, 128, 128]
                # or [0, 0, 0])
                seen = False
                break
            if seen:
                color_cams.append(cam)

        r = 0
        g = 0
        b = 0
        leng = len(color_cams)
        # Safety net. If no cameras see this voxel,
        # return black.
        if leng == 0:
            return [0, 0, 1]
        # Obtain final color from average of colors from cameras that
        # see the voxel.
        for cam in color_cams:
            color = cam.get_image_color(voxel_x, voxel_y, voxel_z)
            r += int(color[0]) * color[0]
            g += int(color[1]) * color[1]
            b += int(color[2]) * color[2]
        color = [int(math.sqrt(r/leng)),
                 int(math.sqrt(g/leng)),
                 int(math.sqrt(b/leng))]
        # Prevent indicator color
        color = [129, 128, 128] if color == [128, 128, 128] else color
        color = [0, 0, 1] if color == [0, 0, 0] else color
        return color
```

Model Images

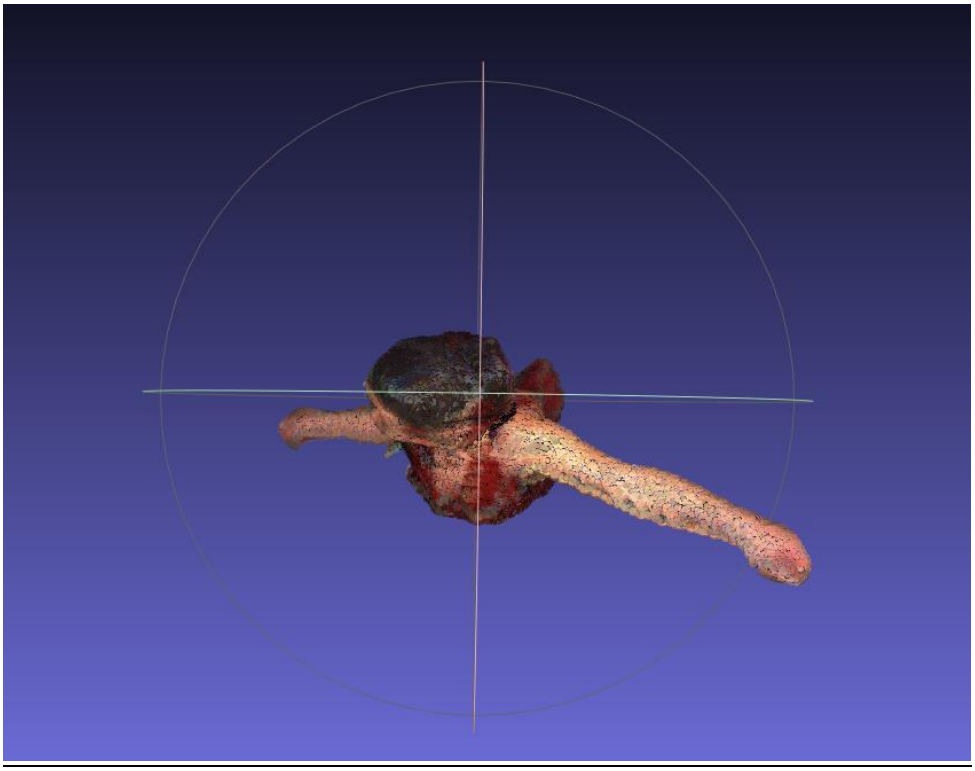
Front

Back

Left

Right

Top



Explanations

1) Loads in the camera projection matrices, the images, and the silhouettes from the provided files, then creates 1 camera object per image/silhouette/projection matrix triple.
2) Creates a voxel grid, with each voxel's side length equal to the programmed constant VOXEL_DIMENSION. I choose 0.002 as the side length. This took over 30 hours to run and produced over 500000 surface vertices.
3) Perform loop over voxel grid
    a. Loop through the voxel grid. For every voxel, get whether that voxel is in the silhouette of each camera. If any camera does not have this voxel in it's silhouette, stop and continue to next voxel without setting the voxel as filled, as it is empty
    b. If a voxel is not free, set it's color to white and set the voxel to occupied in a list called occupied_vertices that is the same size as the voxel grid
        i. Having an indication of whether the voxel is occupied makes checking this more efficient in the next step
    c. For each voxel immediately previous to this one in each axis (i.e. grid[i-1,j,k], grid[i, j-1, k], grid[i, j, k – 1], check if this voxel and that voxel are different (one is free one isn't)
        i. If this is the case, add BOTH voxels to a set of boundary voxels.

        ii. This is sufficient to get a full set of boundary voxels because this method compares all voxels to all their neighbors. The check here explicitly checks the neighbors in the negative direction, and implicitly checks the positive direction, as grid[i+1. j, k] checks grid[i,j,k]'s positive x neighbor, etc.
4) Pass the voxel grid, the boundary voxel list, and the camera list to the model output function
    a. Discard boundary voxels that are free, keeping only surface voxels
    b. For every surface voxel, determine a color for this voxel (Explained in 5)
    c. Open a new output.ply file, write the header information
    d. For every colored voxel, output to the ply file a line containing "x y z r g b", with the proper data replacements
5) The voxel coloring algorithm
    a. A voxel is colored using the average color between each camera that can "see" the voxel.
    b. Which cameras that are able to see the voxel are determined through a crude version of ray marching. I get a vector from the voxel to the camera, and resize it to 2/3 of the voxel side length, then jump forwards from that point to the camera by repeatedly adding that vector to the point.
        i. First, I change the current voxel's color from occupied to an indicator color (128, 128, 128) so the marching algorithm doesn't think this voxel is blocked because of itself
        ii. For every step along the ray, check if the voxel we're in is occupied (not (0,0,0) and not (128,128,128)), and if it is discard that camera. If we reach out of bounds of our voxel grid, assume the voxel is not blocked and add this camera to a list of cameras that can see the voxel
        iii. All such cameras contribute the color of the pixel at the voxel projected to their image, then these colors are averaged and the average is the new voxel color. The "free" color (0,0,0) and "indicator" color (128,128,128) are prevented by changing one element by 1 if they are set to this color.
        iv. Any voxel that is part of the model but blocked for all cameras is set to (1, 0, 0)
6) The "jit" compilation of Camera.project
    a. Since project is called for every voxel, it ended up being the main bottleneck for my implementation
    b. I found a python module called numba that can perform just-in-time compilation of python code to machine code
    c. However, 2 issues cropped up.
        i. Could not interpret Camera object as a type, solved by removing the self parameter and having the projection matrix P be passed as a parameter instead.

    ii. Numpy.matmul not supported as machine code, solved by replacing with a less-efficient method of simply running the matrix multiplication myself
  d. Despite needing to make parts of the project function less efficient by themselves, the jit-compilation sped up overall program execution by 3-4 times, a very significant speedup.
7) Slight coloring errors
  a. I believe the blue streaks on the model are due to an off-by-one error that occurs somewhere in my code, where projection gets a pixel one off to the side of the right pixel. Overall, I am happy with the results of my coloring.