calc

December 2, 2022

1 Calc

1.0.1 Calculation backend for program

1.1 ### Calculated volume of .ply file passed in from GUI frontend

- Creation Date 9/7/22
- Author: Dalton Bailey
- Course: CSCI 490
- Instructor Sam Siewert

Notebook primarily useful to explain individual code blocks, code functions best when ran using *python3 quadp.py* or using the script: *./quadp*

> **Planning Pothole potential using pointcloud from a ply file, powered by python, pacing on the pipad**

1.2 Required Imports

1.2.1 STD Python library imports

- random
- os
- time
- string

1.2.2 Math imports

- numpy
- from scipy.spatial; ConvexHull

1.2.3 Visualization Imports

- matplotlib
- mpl_toolkits.mplot3d
- matplotlib.pyplot

1.2.4 Other imports

- open3d
- sqlite3
- hashlib

[ ]:
```python
# STD Python library imports
import random
import os
from os.path import exists
import time
import string

# Math imports
import numpy as np
from scipy.spatial import ConvexHull

# Visualization Imports
import matplotlib as plot
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# Other imports
import open3d as o3d
import sqlite3
import hashlib
```

Calculation Class

- Class wider member variables are also declared here

[ ]:
```python
class pholeCalc():

    # Class variables (Initialize all as none until they are required)
    input_file = None
    working_dir = None
    debug = None
    refx = None
    refy = None
    refz = None
    ref_points = None
    reference_plane = None
    untrimmed_point_cloud = None
    trimmed_point_cloud = None
    volume = None
    density = None
    mass = None
    units = None
    unitType = None
    salt = None
    conn = None
    c = None
    gui_print = None
```

Initialization function

1. Sets all member variables to their initial values
2. Creates database if it does not exist

[ ]:
```
def __init__(self):

        self.working_dir = os.path.dirname(os.path.realpath(__file__))
        self.salt = ''.join(random.choice(string.ascii_letters)
                            for i in range(10))
        #sqldb = self.working_dir+"/data/localstorage.db"
        sqldb = "data/localstorage.db"
        try:
            # self.conn = sqlite3.connect(self.working_dir+"/data/localstorage.
↪db")
            self.conn = sqlite3.connect(sqldb)
        except:
            raise Exception(
                "Database connection to " + sqldb + " failed; potentially␣
↪corrupted/malformed, or permission error")
        self.c = self.conn.cursor()
        # Create databse if it does not exist
        try:
            self.c.execute("""CREATE TABLE IF NOT EXISTS phole_VMP_Data (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            hash_id TEXT,
            username TEXT,
            input_file TEXT,
            date TEXT,
            position REAL,
            unit_type TEXT,
            volume REAL,
            density REAL,
            mass REAL
            )""")
        except:
            raise Exception(
                "Database creation has failed; potentially corrupted/malformed,␣
↪or permission error")
```

Database closing wrapper Closes the opened sqlite database

[ ]:
```
def closeDBconn(self):
        self.debugout(13)
        try:
            self.conn.commit()
            self.conn.close()
        except:
```

```
            raise Exception(
                "Database comitting & closing has failed; potentially corrupted/
↪malformed, or permission error")
```

Hash generator function Generates a hash based on passed in string Used to create unique IDs for each calculation result, and unique names for each pothole scan

[ ]:
```
def hash(self, hashingvalue):
        hash = hashlib.sha256()
        hash.update(hashingvalue.encode("utf-8"))
        return hash.hexdigest()
```

api function Function called from gui to perform calculations on passed in ply file Performs all function calls in the proper order for calculation

[ ]:
```
def api(self, debug, username, dens, unitType, infile, print_to_gui):

        # Check if userdata file exists in current directory
        self.debug = debug
        self.gui_print = print_to_gui
        # file_exists = exists(infile)
        if (not exists(infile)):
            self.gui_print(text=("\n[QUAD_P]-[calc](exception)" , infile + "␣
↪does not exist"))
            raise Exception(infile + " does not exist")

        # Populate member variables with data received from frontend
        self.density = float(dens)
        self.units = unitType
        self.input_file = infile
        unit_name = None

        if self.units:
            self.densityUnit = "ft3"
            unit_name = "imperial"
        else:
            self.densityUnit = "m3"
            unit_name = "metric"

        # start timer
        start_time = time.process_time()

        # Dump debug information to user
        self.debugout(1)
        self.debugout(14)
        self.debugout(4)
        self.debugout(12)
```

```
        self.debugout(15)
        self.debugout(3)

        # Perform calculations
        self.meshgen()
        self.plotarray() if self.debug else None
        self.refest()
        self.refplot() if self.debug else None
        self.trimcloud()
        self.plottrim() if self.debug else None
        self.volcalc()
        self.masscalc()
        self.debugout(2)

        # Save calculated values to database
        try:
            self.c.execute("INSERT INTO phole_VMP_Data VALUES (NULL, '{hash}',␣
↪'{username}', '{input_file}', DATE('now'), '{pos}', '{db_unit_name}',␣
↪'{vol}', '{dens}', '{mass}')".
                           format(hash=self.hash((str(self.volume)+str(self.
↪density)+str(self.mass)+(self.input_file) + str(self.salt))),␣
↪username=str(username), db_unit_name=str(unit_name), input_file=str(self.
↪input_file), vol=self.volume, dens=self.density, mass=self.mass,␣
↪pos='pos_placeholder'))
        except:
            raise Exception(
                "Database writing has failed; potentially corrupted/malformed,␣
↪or permission error")

        # Calculate total time elapsed during calculation
        end_time = time.process_time()
        print(f"\t[QUAD_P]-[calc] Calculation time: ", (end_time - start_time)␣
↪* 1000, "ms")
        self.gui_print(text=("\n[QUAD_P]-[calc] Calculation time: ", (end_time␣
↪- start_time) * 1000, "ms"))
```

Mesh generation function Generates a 3D numpy array from the passed in ply file Translates the pointcloud data in the ply file into a 3D numpy array for calculation

[ ]:
```
# Generate open3d mesh from pointcloud, and convert it to a 3D numpy array
    # Code adapted from https://stackoverflow.com/questions/36920562/
↪python-plyfile-vs-pymesh
    def meshgen(self):
        self.debugout(4)
        pcd = o3d.io.read_point_cloud(
            self.input_file)  # Read the point cloud
        self.debugout(5)
```

```
        self.meshvis(pcd) if self.debug else None

        # Convert open3d format to numpy array
        self.untrimmed_point_cloud = np.asarray(pcd.points)
        self.debugout(6)
        return
```

Reference plane estimation function Uses the linear best square fit algorithm to establish a reference hyperplane Slope of reference hyperplane should be as close as possible to the slope of the road The more accurate the definition of the reference plane; the more accurate the final volume will be

[ ]:
```
# Reference plane calculation using linear best fit algorithm
    # Adapted from https://gist.github.com/RustingSword/e22a11e1d391f2ab1f2c
    def refest(self):
        self.debugout(7)

        # Calculate reference hyperplane
        try:
            (rows, cols) = self.untrimmed_point_cloud.shape

            if (cols != 3):
                raise Exception("Inavlid col num; likely scanner error")

            # Compute a,b,c for function (ax + by + c = z) which defines plane␣
↪that best fits the data
            G = np.ones((rows, cols))
            G[:, 0] = self.untrimmed_point_cloud[:, 0]  # X
            G[:, 1] = self.untrimmed_point_cloud[:, 1]  # Y
            Z = self.untrimmed_point_cloud[:, 2]        # Z
            (a, b, c), resid, rank, s = np.linalg.lstsq(G, Z, rcond=None)

            # Compute the normal vector for the best fitting plane
            normal = (a, b, -1)
            nn = np.linalg.norm(normal)
            normal = normal / nn

            # Compute distance (d) from origin to best fitting plane
            point = np.array([0.0, 0.0, c])
            d = -point.dot(normal)

            # Get x & y max & mins
            maxx = np.max(self.untrimmed_point_cloud[:, 0])
            maxy = np.max(self.untrimmed_point_cloud[:, 1])
            minx = np.min(self.untrimmed_point_cloud[:, 0])
            miny = np.min(self.untrimmed_point_cloud[:, 1])
```

```
            # Compute bounding x & y  points for ref plane
            self.refx, self.refy = np.meshgrid([minx, maxx], [miny, maxy])
            # Compute bounding z points for ref plane
            self.refz = (-normal[0]*self.refx - normal[1]
                         * self.refy - d)*1. / normal[2]

            # Save bounding points of reference plane to 3D numpy array
            self.ref_points = np.dstack(
                (self.refx, self.refy, self.refz)).reshape((4, 3))

            self.debugout(8)

            self.datadump() if self.debug else None
        except:
            raise Exception("Reference plane estimation has failed; " +
                            self.input_file + " may be corrupt or missing ")
```

Trim cloud function Trims the 3D numpy array based on the previouslt established reference hyperplane The trimming process consists of removing all points above the reference hyperplane All points above the reference hyperplane should be the road, thus leaving us with a 3D array which only represents the pothole

[ ]:
```
# All points above reference plane are to be removed
    def trimcloud(self):
        self.debugout(9)

        # Compute normal vector for plane based on 4 edge points
        plane_normal = np.cross(
            self.ref_points[1] - self.ref_points[0], self.ref_points[2] - self.
↪ref_points[0])
        plane_normal = plane_normal / np.linalg.norm(plane_normal)

        # Compute distance from plane_normal to origin of ref_points
        plane_d = -np.dot(plane_normal, self.ref_points[0])

        # Remove all points above plane using calculated normal
        self.trimmed_point_cloud = self.untrimmed_point_cloud[np.dot(self.
↪untrimmed_point_cloud, plane_normal) + plane_d <= 0]
        # self.trimmed_point_cloud = self.untrimmed_point_cloud
        self.debugout(10)
```

1.3 Calculates the volume of the trimmed 3D array

Performs the convex hull method from scipy spatial to calculate the volume of the trimmed array

[ ]:
```
# Volume calculation using convex hull method
    def volcalc(self):
```

```
        self.debugout(11)
        hull = ConvexHull(self.trimmed_point_cloud)

        self.volume = hull.volume
        if(self.units):
            self.volume = self.volume / 0.028317

        print(f"\t[QUAD_P]-[calc] Volume calculation successful!
↪\n----------------------------------------\n\t[QUAD_P]-[calc] Volume is",␣
↪self.volume, " ", self.densityUnit)
        self.gui_print(text=("\n[QUAD_P]-[calc] Volume calculation successful!
↪\n----------------------------------------\n[QUAD_P]-[calc] Volume is ",␣
↪self.volume, " ", self.densityUnit))
```

1.4 Mass calculation function

Calculates the mass of the required patching material based on the density of the patching material If no density is provided then the mass will simply be saved as -1

Formula used mass = density * volume

[ ]:
```
# Calculate mass of pothole
    def masscalc(self):
        if(self.density != -1):
            self.mass = (self.density * self.volume)
        else:
            self.mass = -1

        massUnit = 0
        if(self.units):
            massUnit = "lbs"
        else:
            massUnit = "kg"

        if(self.density != -1):
            print(f"\t[QUAD_P]-[calc] Using input density and calculated volume␣
↪to determine mass\n[QUAD_P]-[calc] Mass of patching material required is␣
↪",self.mass, " ", massUnit)
            self.gui_print(text=("\n[QUAD_P]-[calc] Using input density and␣
↪calculated volume to determine mass\n[QUAD_P]-[calc] Mass of patching␣
↪material required is ",self.mass, " ", massUnit))
```