quadp

December 4, 2022

1 Quadp

1.1 GUI frontend for program

1.2 ## Manages realsense cam vision and exporting to .ply file

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

1.3 Instructor Sam Siewert

Planning Pothole potential using pointcloud from a ply file, powered by python, pacing on the pipad Notebook primarily useful to explain individual code blocks, code functions best when ran using *python3 quadp.py* or using the script: *./quadp*

1.4 ### Required Imports

- yaml: Allows program to save configuration data to yaml file
- tkinter: Creates GUI and embedded gui widgets
- numpy: Performs complex calculations
- cv2: Handles piping camera vision to GUI
- atexit: Performs functionality on program exit
- os: Allows program to run os commans
- themes: Custom library for holding themes
- calibration: Calibrates Realsense Camera
- calc: Calculation backend (see calc.ipynb)
- pyrealsense2: Python Realsense library (allows control of Realsense Camera)
- time: Used to time code blocks
- PIL: Python Image Library, handles image conversion and processing
- webview: Opens embedded webbrowser for Documentation

[ ]:
```
import yaml
import tkinter as tk
from tkinter import ttk
import numpy as np
```

```
import cv2 as cv2
import matplotlib.pyplot as plt
import atexit
import os
from os.path import exists
from themes import *
import calibration as cal
from calc import *
import pyrealsense2 as rs
import time
import PIL as pil
from PIL import ImageTk
from IPython.display import clear_output  # Clear the screen
import webview
```

1.5 #### Class variables for gui

[ ]:
```
class interface():
    # Class variables (Initialize all as none until they are required)
    root = None
    working_dir = None
    output_file = None
    scanning = None

    # User config variables
    conf_file = None
    conf = None
    username = None
    debug = None
    units = None
    density = None
    densityUnit = None

    # Global GUI Variables
    theme = None
    screen_width = None
    screen_height = None

    # Global Tkinter Widgets
    video_out = None
    cam_controls = None
    s_scan_button = None
    export_button = None
```

1.6 #### GUI Constructor

- Determines width of screen

- Sets global Tkinter widgets

[ ]:
```
    # Constructor for Interface object
    def __init__(self):
        self.root = tk.Tk()  # Calls tktinker object and sets self.root to be␣
↪equal to it
        self.calcBackend = pholeCalc()  # Initialize the calculation backend
        self.working_dir = os.path.dirname(os.path.realpath(__file__))
        #self.working_dir = os.getcwd()
        self.conf_file = self.working_dir+'/data/conf.yml'

        # Generate unique hash to store export of scan (takes some time)
        self.output_file = self.working_dir + "/data/ply/" + self.calcBackend.
↪hash(
            (''.join(random.choice(string.ascii_letters) for i in range(7)))) +␣
↪".ply"

        # Load configuration from yaml file
        self.loadConfig()

        self.screen_width = self.root.winfo_screenwidth()  # Get width of␣
↪current screen
        self.screen_height = self.root.winfo_screenheight()  # Get height of␣
↪current screen

        # Set program title and geometry of root gui
        self.root.title("Quad-P")
        self.root.geometry("%dx%d" % (self.screen_width, self.screen_height))

        # Configure GUI title and Geometry
        self.root.configure(background=themes[self.theme]['background_colo'])

        # Create Location for video feed in GUI
        self.video_out = tk.Canvas(
            self.root, bg="#000000", height=480, width=640, borderwidth=5,␣
↪relief="sunken")
        self.video_out.grid(column=0, row=1, columnspan=10,
                            pady=35, ipadx=5, ipady=5, sticky=tk.NS)
        # Create Location for text output in GUI
        self.cam_controls = tk.Label(self.root, fg=themes[self.
↪theme]['background_colo'], bg=themes[self.theme]['background_colo'],␣
↪height=round(
            self.screen_height*0.002555), width=round(self.screen_width*0.059))
        self.cam_controls.grid(column=0, row=2, columnspan=10,
                               sticky=tk.NS)

        self.s_scan_button = tk.Button(
```

```
            self.cam_controls, text="Enable Camera", command=lambda: self.
↪startScan())
        self.s_scan_button.grid(column=1, row=0, padx=20)

        self.export_button = tk.Button(
            self.cam_controls, text="Export Scan", command=lambda: self.
↪exportScan())
        self.export_button.grid(column=3, row=0, padx=20)

        # Create Location for text output in GUI
        self.b_data = tk.Label(self.root, fg=themes[self.theme]['main_colo'],␣
↪bg=themes[self.theme]['main_colo'], height=round(
            self.screen_height*0.00555), width=round(self.screen_width*0.059),␣
↪borderwidth=5, relief="solid")
        self.b_data.grid(column=0, row=3, columnspan=10,
                         sticky=tk.SW)
```

Livestream Camera's Vision

[ ]:
```
    def startScan(self):
        pipe = rs.pipeline()                        # Create a pipeline
        cfg = rs.config()                           # Create a default␣
↪configuration
        print("[QUAD_P] Pipeline is created") if gui.debug else None

        print("[QUAD_P] Searching For Realsense Devices..") if gui.debug else␣
↪None
        selected_devices = []                       # Store connected device(s)

        for d in rs.context().devices:
            selected_devices.append(d)
            print(d.get_info(rs.camera_info.name))
        if not selected_devices:
            print("No RealSense device is connected!")
            return

        print(
            "[QUAD_P] (debug) Streaming camera vision to GUI... ") if gui.debug␣
↪else None

        rgb_sensor = depth_sensor = None

        for device in selected_devices:
            print("Required sensors for device:",
                  device.get_info(rs.camera_info.name))
            for s in device.sensors:                                # Show␣
↪available sensors in each device
```

```
                if s.get_info(rs.camera_info.name) == 'RGB Camera':
                    print("[QUAD_P] - RGB sensor found") if gui.debug else None
                    rgb_sensor = s                                  # Set RGB␣
↪sensor
                if s.get_info(rs.camera_info.name) == 'Stereo Module':
                    depth_sensor = s                                # Set Depth␣
↪sensor
                    print("[QUAD_P] - Depth sensor found") if gui.debug else␣
↪None
        # Mapping depth data into RGB color space
        colorizer = rs.colorizer()
        # Configure and start the pipeline
        profile = pipe.start(cfg)

        # Show 1 row with 2 columns for Depth and RGB frames
        fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(24, 8))
        # Title for each frame
        title = ["Depth Image", "RGB Image"]

        # Skip first frames to give syncer and auto-exposure time to adjust
        for _ in range(10):
            frameset = pipe.wait_for_frames()

        # Increase to display more frames
        for _ in range(30):
            # Read frames from the file, packaged as a frameset
            frameset = pipe.wait_for_frames()
            depth_frame = frameset.get_depth_frame()                # Get depth␣
↪frame
            color_frame = frameset.get_color_frame()                # Get RGB␣
↪frame

            # This is what we'll actually display
            colorized_streams = []
            if depth_frame:
                colorized_streams.append(np.asanyarray(
                    colorizer.colorize(depth_frame).get_data()))
            if color_frame:
                colorized_streams.append(np.asanyarray(color_frame.get_data()))

            # Iterate over all (Depth and RGB) colorized frames
            for i, ax in enumerate(axs.flatten()):
                if i >= len(colorized_streams):
                    continue          # When getting less frames than expected
                # Set the current Axes and Figure
                plt.sca(ax)
                # colorized frame to display
```

```
                plt.imshow(colorized_streams[i])
                # Add title for each subplot
                plt.title(title[i])
            # Clear any previous frames from the display
            clear_output(wait=True)
            # Adjusts display size to fit frames
            plt.tight_layout()
            # Make the playback slower so it's noticeable
            plt.pause(1)

        pipe.stop()                                                    # Stop the␣
↪pipeline
        print("[QUAD_P] Done!")
```

Stop Livestreaming Camera Vision

[ ]:
```
    def stopScan(self):
        print("[QUAD_P] (debug) Disabling live feed...") if gui.debug else None
        self.scanning = False
        self.s_scan_button = tk.Button(
            self.cam_controls, text="Enable Camera", command=lambda: self.
↪stopScan())
        self.s_scan_button.grid(column=1, row=0, padx=20)
        if self.export_scan:
            self.exportScan()
```

Export Scan into a .ply file Uses code adapted from pyrealsense github to export what the camera currently sees to a ply file

This exported file will be given a random hash value for a name, unless the user has specified a desired name

[ ]:
```
    def exportScan(self):
        start_time = time.process_time()  # start timer
        print("Searching For Realsense Devices..")
        selected_devices = []                        # Store connected device(s)

        for d in rs.context().devices:
            selected_devices.append(d)
            print(d.get_info(rs.camera_info.name))
        if not selected_devices:
            print("No RealSense device is connected!")
            return

        print(
            "[QUAD_P] (debug) Exporting camera's vison as .ply file...") if gui.
↪debug else None
```

```
        # Declare pointcloud object, for calculating pointclouds and texture␣
↪mappings
        pc = rs.pointcloud()
        # We want the points object to be persistent so we can display the last␣
↪cloud when a frame drops
        points = rs.points()

        # Declare RealSense pipeline, encapsulating the actual device and␣
↪sensors
        pipe = rs.pipeline()
        config = rs.config()
        # Enable depth stream
        config.enable_stream(rs.stream.depth)

        # Start streaming with chosen configuration
        pipe.start(config)

        # We'll use the colorizer to generate texture for our PLY
        # (alternatively, texture can be obtained from color or infrared stream)
        colorizer = rs.colorizer()

        try:
            # Give camera time to adjust to exposure
            for x in range(10):
                pipe.wait_for_frames()

            # Wait for the next set of frames from the camera
            frames = pipe.wait_for_frames()
            colorized = colorizer.process(frames)

            # Create save_to_ply object
            ply = rs.save_to_ply(self.output_file)

            # Set options to the desired values
            # In this example we'll generate a textual PLY with normals (mesh␣
↪is already created by default)
            ply.set_option(rs.save_to_ply.option_ply_binary, False)
            ply.set_option(rs.save_to_ply.option_ply_normals, True)

            print("[QUAD_P] (debug) Saving to ",
                  self.output_file, "...") if gui.debug else None

            # Apply the processing block to the frameset which contains the␣
↪depth frame and the texture
            ply.process(colorized)

            print(f"[QUAD_P] (debug) Export Complete!\n Elapsed time was ",
```

```
                (time.process_time() - start_time) * 1000, "ms.\n") if gui.
↪debug else None
        finally:
            pipe.stop()
```

Calibration function Calibrates the Realsense camera using code provided from the pyrealsense github

Calibration will not do anything if there is no Realsense device connected

[ ]:
```
    # Wrapper for realsense calibration module
    def calibrate(self):
        if gui.checkCam() == None:
            return
        start_time = time.process_time()  # start timer
        self.debugout(9) if self.debug else None
        try:
            cal.main()
            stop_time = time.process_time()
            print(f"[QUAD_P] (debug) Calibration Complete!\n Elapsed time was ",
                  (stop_time - start_time) * 1000, "ms.\n") if gui.debug else␣
↪None
            self.gui_print(text=("\n[QUAD_P] (debug) Calibration Complete!\n␣
↪Elapsed time was ", (
                stop_time - start_time) * 1000, "ms.\n")) if self.debug else␣
↪None
        except:
            self.gui_print(text=(
                "\n[QUAD_P] (exception) Calibration has failed, realsense␣
↪device potentially disconnected."))
            raise Exception(
                "[QUAD_P] (exception) Calibration has failed, realsense device␣
↪potentially disconnected.")
```

1.6.1 GUI Functions

Calculation starting wrapper

1. Checks which file the user desires to perform calcualtions on
2. Checks if the desired file exists
3. Passes in all required variables to calc backend api() function
4. If user has input a density that value is passed to the backend, otherwise -1 is passed

[ ]:
```
    def startCalc(self, desired_file):

        if (desired_file == "input"): # User has selected to perform␣
↪calculations on loaded input file
```

```
            if (exists(self.input_file) and self.input_file != "None"): #␣
↪Verify that input_file exists
                try:

                    print("[QUAD_P] Performing calculations with debugout") if␣
↪self.debug else print(
                        "[QUAD_P] Performing calculations without debugout")
                    self.gui_print(text=("\n[QUAD_P] Performing calculations␣
↪with debugout")) if self.debug else self.gui_print(
                        text=("\n[QUAD_P] Performing calculations without␣
↪debugout"))

                    # Perform calculations with appropriate values via␣
↪calculation api function #
                    self.calcBackend.api(debug=self.debug, username=self.
↪username, dens=self.density, unitType=self.units, infile=self.input_file,␣
↪print_to_gui=self.gui_print) if self.density else self.calcBackend.
↪api(debug=self.debug, username=self.username, dens=-1, unitType=self.units,␣
↪infile=self.input_file, print_to_gui=self.gui_print)
                    # threading.Thread(target=).start()
                except:
                    self.gui_print(
                        text=("\n[QUAD_P] (exception) Calculation raised an␣
↪exception"))
                    raise Exception(
                        "[QUAD_P] (exception) Calculation raised an exception")
            else:
                self.gui_print(text=(
                    "\n[QUAD_P] Input file does not exist, please select one␣
↪via Scan -> open"))
                print(
                    "[QUAD_P] Input file does not exist; please select one via␣
↪Scan -> open")
        else:
            if (exists(self.output_file)): # User has selected to perform␣
↪calculations on exported output file
                try:
                    print("[QUAD_P] Performing calculations with debugout") if␣
↪self.debug else print(
                        "[QUAD_P] Performing calculations without debugout")
                    self.gui_print(text=("\n[QUAD_P] Performing calculations␣
↪with debugout")) if self.debug else self.gui_print(
                        text=("\n[QUAD_P] Performing calculations without␣
↪debugout"))
```

```
                    # Perform calculations with appropriate values via␣
↪calculation api function #
                    self.calcBackend.api(debug=self.debug, username=self.
↪username, dens=self.density, unitType=self.units, infile=self.input_file,␣
↪print_to_gui=self.gui_print) if self.density else self.calcBackend.
↪api(debug=self.debug, username=self.username, dens=-1, unitType=self.units,␣
↪infile=self.input_file, print_to_gui=self.gui_print)

                    # self.calcBackend.api(self.density, self.units, self.
↪output_file, self.gui_print) if self.density else self.calcBackend.api(-1,␣
↪self.units, self.output_file, self.gui_print)
                    # threading.Thread(target=)).start()
                except:
                    self.gui_print(
                        text=("\n[QUAD_P] (exception) Calculation raised an␣
↪exception"))
                    raise Exception(
                        "[QUAD_P] (exception) Calculation raised an exception")
            else:
                self.gui_print(
                    text=("\n[QUAD_P] Output file does not exist, please␣
↪produce one via export"))
                print(
                    "[QUAD_P] Output file does not exist, please produce one␣
↪via export")
```

ply Viewer function Uses the open3D viewer to allow the user to see an interactable 3D view of the desired ply file 1. Checks which file the user desires to view 1. Checks if the desired file exists

[ ]:
```
    # View scan in Open3D viewer
    def viewScan(self, desired_file):
        if (desired_file == "input"):
            # if(self.input_file):
            if (exists(self.input_file) and self.input_file != "None"):
                try:
                    pcd = o3d.io.read_point_cloud(
                        self.input_file)  # Read the point cloud
                    # Visualize the point cloud within open3d
                    o3d.visualization.draw_geometries([pcd])
                except:
                    self.gui_print(
                        text=("\n[QUAD_P] (exception) Visualization raised an␣
↪exception"))
                    raise Exception(
```

```
                        "[QUAD_P] (exception) Visualization raised an␣
↪exception")
            else:
                self.gui_print(text=(
                    "\n[QUAD_P] Input file does not exist, please select one␣
↪via Scan -> open"))
                print(
                    "[QUAD_P] Input file does not exist; please select one via␣
↪Scan -> open")
        else:
            if (exists(self.output_file)):
                try:
                    pcd = o3d.io.read_point_cloud(
                        self.output_file)  # Read the point cloud
                    # Visualize the point cloud within open3d
                    o3d.visualization.draw_geometries([pcd])
                except:
                    self.gui_print(
                        text=("\n[QUAD_P] (exception) Visualization raised an␣
↪exception"))
                    raise Exception(
                        "[QUAD_P] (exception) Visualization raised an␣
↪exception")
            else:
                self.gui_print(
                    text=("\n[QUAD_P] Output file does not exist, please␣
↪produce one via export"))
                print("[QUAD_P] Output file or Input file does not exist;␣
↪please produce Output file via export, or select one via Scan -> open")
```

Saves configuration variables Config variables are saved to a yaml file, for persistent storage

[ ]:
```
    # Write the current config dictionary to the yaml file
    def saveConfig(self):
        self.debugout(10) if self.debug else None
        self.conf['debug'] = self.debug
        self.conf['theme'] = self.theme
        self.conf['username'] = self.username
        self.conf['units'] = self.units
        try:
            # Save any modified configs to the yaml file
            with open(self.conf_file, 'w') as f:
                yaml.dump(self.conf, f)
            print(self.conf)
        except:
            self.gui_print(text=(
```

```
                "\n[QUAD_P] (exception) Could not save configuration data to␣
↪file, likely insufficeint directory permissions."))
            raise Exception(
                "[QUAD_P] (exception) Could not save configuration data to␣
↪file, likely insufficeint directory permissions.")
```

Load configuration data Saves configuration data from yaml file into the appropriate member variables

[ ]:
```
    # Load the config dictionary from the yaml file
    def loadConfig(self):
        # Check if userdata file exists in current directory
        file_exists = exists(self.conf_file)
        try:
            # If yaml file DNE create a fresh one and set all values to defaults
            if file_exists == 0:
                dict = {'username': 'guest',
                        'themechoice': 'default', 'debug': 0, 'units': 0}
                with open(self.conf_file, 'w') as f:
                    yaml.dump(dict, f)
        except:
            self.gui_print(text=(
                "\n[QUAD_P] (exception) Could not create fresh config file,␣
↪likely insufficeint directory permissions."))
            raise Exception(
                "[QUAD_P] (exception) Could not create fresh config file,␣
↪likely insufficeint directory permissions.")

        # Load values from yaml file into self.conf
        try:
            with open(self.conf_file) as f:
                self.conf = yaml.safe_load(f)
        except:
            self.gui_print(
                text=("\n[QUAD_P] (exception) Could not open configuration file.
↪"))
            raise Exception(
                "[QUAD_P] (exception) Could not open configuration file.")

        # Try to load values from self.conf into respective vars
        try:
            self.debug = self.conf['debug']
            self.calcBackend.debug = self.debug
            self.theme = self.conf['theme']
            self.username = self.conf['username']
            self.units = self.conf['units']
            self.calcBackend.units = self.units
```

```
        except:
            # os.remove(self.conf_file)
            self.gui_print(text=(
                "\n[QUAD_P] (exception) Could not load data from configuration␣
↪file, potentially corrupted/malformed; remove or correct"))
            raise Exception(
                "[QUAD_P] (exception) Could not load data from configuration␣
↪file, potentially corrupted/malformed; remove or correct")
```

Change configuration Opens a GUI window which allows the user to change their configuration data Writes the changed configuration data to the yaml file

[ ]:
```
# Change configuration variables
    def changeConfig(self):
        self.loadConfig()

        # Init tkinter vars
        debug_var = tk.BooleanVar()
        unit_var = tk.IntVar()
        theme_var = tk.IntVar()

        debug_var.set(self.debug)
        for key, val in themeidict.items():
            if val == self.theme:
                print(key)
                theme_var.set(key)

        # Create new window and base it off orginal window
        window = tk.Toplevel(self.root)
        window.configure(background=themes[self.theme]['background_colo'])
        window.geometry("%dx%d" % (self.screen_width*0.5,
                        self.screen_height*0.75))

        def get_name_input():
            self.username = inputname.get("1.0", "end-1c")
            nameinputlabel2.config(text="Username is now: " + self.username)

        def commit_changes():
            self.theme = themeidict[theme_var.get()]
            self.debug = debug_var.get()
            self.units = unit_var.get()
            self.saveConfig()
            window.destroy()

        label = tk.Label(window, text='Configuration', font=(
            "Arial", 15), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=20)
```

```
        label.grid(column=0, row=0, columnspan=10, sticky=tk.NS)

        # Create Horizontal seperator bar
        separator1 = ttk.Separator(window, orient='horizontal')
        separator1.grid(column=0, row=1, columnspan=10, sticky=tk.EW)

        # Username Buttons
        nameinputlabel = tk.Label(window, text='Name', font=(
            "Arial", 10), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=8)
        nameinputlabel.grid(column=0, row=2, padx=20, pady=30)

        inputname = tk.Text(window,  height=2, width=40)
        inputname.grid(column=1, row=2, columnspan=10, padx=20, pady=30)
        inputname.insert(tk.INSERT, self.username)

        nameinputlabel2 = tk.Label(window, text='', font=(
            "Arial", 10), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=60)
        nameinputlabel2.grid(column=2, row=3)

        enterbutton = tk.Button(
            window, text=" ", command=lambda: get_name_input())
        enterbutton.grid(column=5, row=2, padx=20, pady=30)

        # Theme buttons
        theme1 = tk.Radiobutton(window, bg=themes[gui.theme]['main_colo'],␣
↪text="Default",
                                variable=theme_var, value=1)
        theme1.grid(column=0, row=4, padx=20, pady=30)

        theme2 = tk.Radiobutton(window, bg=themes[gui.theme]['main_colo'],␣
↪text="Spicy",
                                variable=theme_var, value=2)
        theme2.grid(column=1, row=4, padx=20, pady=30)

        theme3 = tk.Radiobutton(window, bg=themes[gui.theme]['main_colo'],␣
↪text="Juicy",
                                variable=theme_var, value=3)
        theme3.grid(column=2, row=4, padx=20, pady=30)

        # Unit Selection buttons
        unit1 = tk.Radiobutton(window, bg=themes[gui.theme]['main_colo'],␣
↪text="SI Units",
                               variable=unit_var, value=0)
        unit1.grid(column=0, row=5, padx=20, pady=30)
```

```
        unit2 = tk.Radiobutton(window, bg=themes[gui.theme]['main_colo'],␣
↪text="Imperial Units",
                               variable=unit_var, value=1)
        unit2.grid(column=1, row=5, padx=20, pady=30)

        checkbutton = tk.Checkbutton(
            window, text="DEBUG", variable=debug_var)  # Debug button
        checkbutton.grid(column=0, row=6, padx=20, pady=30)

        commitchanges = tk.Button(
            window, text="Confirm Changes  ", command=lambda: commit_changes())␣
↪ # Commit changes button
        commitchanges.grid(column=0, row=7, padx=20, pady=30)
```

Density input function Opens a GUI window which allows the user to input the density of the desired patching material

The units assigned to this density are specified by the unit type selected in the current configuration

[ ]:
```
def inputDensity(self):
        # Create new window and base it off orginal window
        window = tk.Toplevel(self.root)
        window.configure(background=themes[self.theme]['background_colo'])
        window.geometry("%dx%d" % (self.screen_width*0.25,
                        self.screen_height*0.15))  # Set size of window
        if self.units:
            densityUnit = "lb/ft3"
        else:
            densityUnit = "g/cm³"

        def get_density_input():
            self.density = densityinput.get("1.0", "end-1c")
            print("[QUAD_P] Density is set to " +
                  self.density + " " + densityUnit)
            self.gui_print(text=("\n[QUAD_P] Density is set to " +
                                 self.density + " " + densityUnit)) if self.
↪debug else None
            window.destroy()

        label = tk.Label(window, text='Input Material Density', font=(
            "Arial", 15), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=20)
        label.grid(column=0, row=0, columnspan=10, sticky=tk.NS)

        # Create Horizontal seperator bar
        separator = ttk.Separator(window, orient='horizontal')
        separator.grid(column=0, row=1, columnspan=20, sticky=tk.EW)
```

```
        # Density input
        densityinputlabel = tk.Label(window, text='Density = ', font=(
            "Arial", 10), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=8)
        densityinputlabel.grid(column=0, row=2, padx=20, pady=30)

        densityinput = tk.Text(window,  height=2, width=40)
        densityinput.grid(column=1, row=2, padx=20, pady=30)
        # densityinput.insert(tk.INSERT, self.density)

        densityunitlabel = tk.Label(window, text=densityUnit, font=(
            "Arial", 10), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=16)
        densityunitlabel.grid(column=2, row=2, padx=20, pady=30)

        enterbutton = tk.Button(
            window, text=" ", command=lambda: get_density_input())
        enterbutton.grid(column=3, row=2, padx=20, pady=30)
```

Rename ply function Opens a gui which allows the user to rename the export ply filename to something more human readable

By default the export ply filename is set to a random hash value, primarily to prevent duplicate filenames

[ ]:
```
def renamePLY(self):
        # Create new window and base it off orginal window
        window = tk.Toplevel(self.root)
        window.configure(background=themes[self.theme]['background_colo'])
        window.geometry("%dx%d" % (self.screen_width*0.25,
                        self.screen_height*0.15))  # Set size of window

        def get_ply_input():
            self.output_file = self.working_dir + \
                "/data/ply/" + plynameinput.get("1.0", "end-1c") + ".ply"
            print("[QUAD_P] Output file name is now " + self.output_file)
            self.gui_print(text=(
                "\n[QUAD_P] Output file name is now " + self.output_file)) if␣
↪self.debug else None
            window.destroy()

        label = tk.Label(window, text='Input .ply Filename', font=(
            "Arial", 15), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=20)
        label.grid(column=0, row=0, columnspan=10, sticky=tk.NS)
```

```
        # Create Horizontal seperator bar
        separator = ttk.Separator(window, orient='horizontal')
        separator.grid(column=0, row=1, columnspan=20, sticky=tk.EW)

        # Density input
        plynameinputlabel = tk.Label(window, text='Name = ', font=(
            "Arial", 10), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=8)
        plynameinputlabel.grid(column=0, row=2, padx=20, pady=30)

        plynameinput = tk.Text(window,  height=2, width=40)
        plynameinput.grid(column=1, row=2, padx=20, pady=30)
        # plynameinput.insert(tk.INSERT, self.output_file)

        enterbutton = tk.Button(
            window, text=" ", command=lambda: get_ply_input())
        enterbutton.grid(column=3, row=2, padx=20, pady=30)
```

View database function Open a GUI which allows the user to view entries in the sqlite database

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

        # Create new window and base it off orginal window
        window = tk.Toplevel(self.root)
        window.configure(background=themes[self.theme]['background_colo'])
        window.geometry("%dx%d" % (self.screen_width*0.4,
                        self.screen_height*0.65))  # Set size of window

        # Label for popup window
        label = tk.Label(window, text='Viewing SQLite Database', font=(
            "Arial", 15), fg=themes[self.theme]['text_colo'], bg=themes[gui.
↪theme]['background_colo'], height=2, width=30)
        label.grid(column=0, row=0, columnspan=10,
                   padx=20, pady=30, sticky=tk.NS)

        separator = ttk.Separator(window, orient='horizontal')
        separator.grid(column=0, row=1, columnspan=20, sticky=tk.EW)

        self.calcBackend.c.execute("SELECT * FROM phole_VMP_Data")

        tree = ttk.Treeview(window)
        tree.grid(column=0, row=2, rowspan=20, columnspan=20,
                  padx=20, pady=30, sticky=tk.NSEW)
        window.columnconfigure(0, weight=1)
        # window.rowconfigure(0, weight=1)

        tree["columns"] = ("one", "two", "three", "four",
```

```
                            "five", "six", "seven", "eight", "nine", "ten")
        tree.column("one", width=40)
        tree.column("two", width=40)
        tree.column("three", width=40)
        tree.column("four", width=40)
        tree.column("five", width=40)
        tree.column("six", width=40)
        tree.column("seven", width=40)
        tree.column("eight", width=40)
        tree.heading("one", text="id")
        tree.heading("two", text="Hash_id")
        tree.heading("three", text="username")
        tree.heading("four", text="Input_file")
        tree.heading("five", text="Date")
        tree.heading("six", text="Position")
        tree.heading("seven", text="Unit_Type")
        tree.heading("eight", text="Volume")
        tree.heading("nine", text="Density")
        tree.heading("ten", text="Mass")

        for row in self.calcBackend.c.fetchall():
            tree.insert("", tk.END, values=row)
```

Quit wrapper Function which performs some cleanup whenever the program exits

1. Closes the sqlite database
2. Saves the current configuration variables to the yaml file

[ ]:
```
    # Graceful exit function
    def quitWrapper(self):
        if(self.exited == False):
            # gui.debugout(2) if self.debug else None
            try:
                self.exited = True
                self.calcBackend.closeDBconn()
                self.saveConfig()
                #self.root.quit()
            except:
                raise Exception("[QUAD_P] (exception) Graceful exit has failed.
↪")
```

Fullscreen functon Fullscreens the main GUI window

[ ]:
```
    # Allow toggling of fullscreen (Currently just fullscreens with no way to␣
↪reverse)
    def fullScreen(self):
        self.root.attributes(
```

```
            "-fullscreen", not self.root.attributes("-fullscreen"))
```

Import function Allows user to import a pre-existing ply file from the filesystem

[ ]:
```
    # Open .ply file from local filesystem
    def openFromFS(self):
        opend_file = filedialog.askopenfilename(
            title="Select file", filetypes=(("ply files", "*.ply"), ("all␣
↪files", "*.*")))
        if (len(opend_file) == 0):
            return
        else:
            self.input_file = opend_file
            self.gui_print(
                text=("\n[QUAD_P] Input file is now ", self.input_file))
```

View Documentation function Uses an embedded web browser to open up the documentation branch of the official QuadP github repo

[ ]:
```
    def viewDocs(self):
        try:
            webview.create_window(
                'Documentation', 'https://github.com/Yalton/CSCI_Capstone/tree/
↪Documentation')
            webview.start()
        except:
            self.gui_print(text=(
                "\n[QUAD_P] (exception) Viewing Documentation has thrown an␣
↪exception, please check terminal"))
            raise Exception(
                "[QUAD_P] (exception) Viewing Documentation has thrown an␣
↪exception")
```

Opens the contact page of the developers blog Allows users to contact the developer (Dalton Bailey) by using the contact page of his blog

[ ]:
```
    def contact(self):
        try:
            webview.create_window(
                'Contact', 'https://daltonbailey.com/contact/')
            webview.start()
        except:
            self.gui_print(text=(
                "\n[QUAD_P] (exception) Viewing Contact has thrown an␣
↪exception, please check terminal"))
            raise Exception(
                "[QUAD_P] (exception) Viewing Contact has thrown an exception")
```

GUI print function Prints whatever string is passed into the function into the emulated terminal in the main GUI

[ ]:
```
    # gui_print Dumps text to the tkinter console widget
    def gui_print(self, text):
        if(not self.exited):
            try:
                if (isinstance(text, tuple)):
                    text = map(str, text)
                    text = ''.join(text)
                text.replace('}', '')
                self.em_terminal.configure(state="normal")
                self.em_terminal.insert("end", text)
                self.em_terminal.configure(state="disabled")
            except:
                # self.gui_print(
                #     text=("\n[QUAD_P] (exception) Printing to GUI has␣
↪encountered an error"))
                raise Exception(
                    "[QUAD_P] (exception) Printing to GUI has encountered an␣
↪error")
```

Main Function Function which is called whenver the program is booted up

Add commands to top bar and syncs them to the proper commands

[ ]:
```
# Main of program, creates main window that pops up when program opens
if __name__ == "__main__":

    # create the root window
    gui = gui()
    print(f"___                    _ ____ \n / _ \ _   _  __ _  __| |  _ \ \n| |␣
↪| | | | |/ _` |/ _` | |_) | \n| |_| | |_| | (_| | (_| |  __/ \n␣
↪\__\_\\__,_|\__,_|\__,_|_|")
    gui.gui_print(
        "___                    _ ____ \n / _ \ _   _  __ _  __| |  _ \ \n| | |␣
↪| | |/ _` |/ _` | |_) | \n| |_| | |_| | (_| | (_| |  __/ \n␣
↪\__\_\\__,_|\__,_|\__,_|_|")
    print(f"\n----------------------------------------")
    gui.gui_print("\n----------------------------------------")
    print("[QUAD_P] Welcome: ", gui.username)
    gui.gui_print(text=("\n[QUAD_P] Welcome: ", gui.username))
    print("[QUAD_P] Working Directory is: ", gui.working_dir)
    gui.gui_print(text=("\n[QUAD_P] Working Directory is: ", gui.working_dir))
    print("[QUAD_P] Output file is: ", gui.output_file)
    gui.gui_print(text=("\n[QUAD_P] Output file is: ", gui.output_file))
    print("[QUAD_P] Theme is: ", gui.theme)
    gui.gui_print(text=("\n[QUAD_P] Theme is: ", gui.theme))
```

```
    print("[QUAD_P] Calculation unit type is: Imperial Units") if gui.units ==␣
↪1 else print(
        "[QUAD_P] Calculation unit type is: SI Units")
    gui.debugout(1) if gui.debug else None
    selected_devices = gui.checkCam()
    print(f"\n----------------------------------------")
    gui.gui_print("\n----------------------------------------")
    menubar = tk.Menu(gui.root, background=themes[gui.theme]
                      ['main_colo'],
                      fg=themes[gui.theme]
                      ['text_colo'], borderwidth=5, relief="solid")

    # Declare file and edit for showing in menubar
    scan = tk.Menu(menubar, tearoff=False, fg=themes[gui.theme]
                   ['text_colo'], background=themes[gui.theme]['main_colo'])
    view = tk.Menu(menubar, tearoff=False, fg=themes[gui.theme]
                   ['text_colo'], background=themes[gui.theme]['main_colo'])
    edit = tk.Menu(menubar, tearoff=False, fg=themes[gui.theme]
                   ['text_colo'], background=themes[gui.theme]['main_colo'])
    help = tk.Menu(menubar, tearoff=False, fg=themes[gui.theme]
                   ['text_colo'], background=themes[gui.theme]['main_colo'])

    calc_submenu = tk.Menu(
        menubar, tearoff=False, fg=themes[gui.theme]['text_colo'],␣
↪background=themes[gui.theme]['main_colo'])
    calc_submenu.add_command(
        label="Input File", command=lambda: gui.startCalc("input"))
    calc_submenu.add_command(
        label="Output File", command=lambda: gui.startCalc("output"))

    # Add commands in in scan menu
    # scan.add_command(label="New", command=lambda: gui.exportScan())
    scan.add_command(label="New", command=lambda: threading.Thread(target=(gui.
↪exportScan())).start())
    scan.add_command(label="Rename", command=lambda: gui.renamePLY())
    scan.add_command(label="Open", command=lambda: gui.openFromFS())
    scan.add_cascade(label="Calc", menu=calc_submenu)
    scan.add_separator()
    scan.add_command(label="Calibrate", command=lambda: gui.calibrate())

    # Add submenu
    view_submenu = tk.Menu(
        menubar, tearoff=False, fg=themes[gui.theme]['text_colo'],␣
↪background=themes[gui.theme]['main_colo'])
    view_submenu.add_command(
        label="Input File", command=lambda: gui.viewScan("input"))
    view_submenu.add_command(
```

```
        label="Output File", command=lambda: gui.viewScan("output"))

    # Add commands in view menu
    view.add_command(label="Database", command=lambda: gui.viewDB())
    view.add_cascade(label="ply file", menu=view_submenu)
    # view.add_command(label="ply file", command=lambda: gui.viewScan())
    view.add_separator()
    view.add_command(label="Toggle Fullscreen",
                     command=lambda: gui.fullScreen())

    # Add commands in edit menu
    edit.add_command(label="Config", command=lambda: gui.changeConfig())
    edit.add_separator()
    edit.add_command(label="M Density", command=lambda: gui.inputDensity())

    # Add commands in help menu
    # help.add_command(label="About")
    help.add_command(label="Docs", command=lambda: gui.viewDocs())
    help.add_command(label="Contact", command=lambda: gui.contact())
    help.add_separator()
    help.add_command(label="Exit", command=lambda: exit())

    # Display the file and edit declared in previous step
    menubar.add_cascade(label="Scan", menu=scan)
    menubar.add_cascade(label="View", menu=view)
    menubar.add_cascade(label="Edit", menu=edit)
    menubar.add_cascade(label="Help", menu=help)

    # Displaying of menubar in the app
    gui.root.config(menu=menubar)

    # # Set keybindings (Broken)
    # gui.root.bind("<F11>", gui.fullScreen())

    # Loop the main
    gui.root.mainloop()
```