Request: I want you to create a roadmap.md for the following application:

High-level vague: I want to build a web app that interacts with the LEGO Mindstorms EV3 system.

The context for the project:

A standard 1x1 LEGO brick is 0.78cm x 0.78cm by 0.96cm high, and the stud is centered on the surface top of the LEGO 1x1 brick with a 0.48cm diameter and the stud height of 0.17cm, this is the standard 1x1 LEGO brick dimension that you will from now on commit to memory and use are reference.

The build plate is a 16 by 16 stud LEGO plate that is 12.8cm by 12.8cm in width and length with 16 by 16 LEGO studs distributed evenly from end to end of the LEGO plate, the stud having the dimensions of 0.48cm diameter circle, and a height of 0.17cm

A full mechanical system is created using laser-cut pieces for the frame, 3d printed parts, and off-the-shelf LEGO pieces to create a “LEGO 3d printer” that also has the functionality to act as a LEGO 2d mosaic builder as it would. The project was fully inspired by this YouTube video https://youtu.be/ec_BtS97IR8. Our custom-built “LEGO 3d printer” has a 16 by 16 stud Lego build plate that can move using a gear rack and pinion system, which is considered as the x-axis. The y-axis is on a permanent set height rail with another gear rack and pinion system that extends past the build plate horizontally to where a row of 1x1 Lego piece dispensers will be. As well on this y-axis will be the “head” that is attached on the y-axis rail that houses an extendable and retractable arm that moves strictly vertically on another gear rack and pinion system that can be considered the z-axis, that is used to pick up and place 1x1 lego bricks vertically. On one of the ends on all of the axes, x-axis (build plate), y-axis (printer head), and z-axis (printer arm) will be touch sensors that will be used for calibration at the start of every “print”, like a normal 3d printer, the motors will move relatively slowly until the ends of the pieces on all axis contact the touch sensor, where it will then again move even slower until it contacts the touch sensor again, this is to ensure perfect calibration and the same starting point each time just like a normal 3d printer. The 1x1 LEGO piece row of dispensers will just be gravity fed at an angle with each individual dispenser having the same color of 1x1 LEGO, therefore down the row of dispensers will be all different colors of 1x1 LEGO bricks, going down the row, which is across the y-axis, you will be able to access supposedly an unlimited amount of different colored 1x1 LEGO bricks but currently we will stick to 6 different colors, therefore 6 different dispensers adjacent in a row(subject to change, and design is made ready for expansion by just extending the y-axis rail). The printer will use the LEGO motor encoders to know its position within the 3d space at all times.

Application:
This is the proposed function of how the “LEGO 3d printer” will work on the software side, creating a web app built for a computer that can be connected via mainly USB cable, Bluetooth,

WIFI or just export the coordinates of bricks to be printed as a txt file, the user can then run their own ROBOTC code to send instructions to the LEGO Mindstorms EV3.

Web app framework:

Use javascript front end, and then mainly use pyodide backend integrating and modifying the current code I have provided for the separate functions that I want implemented

---

General website rules:

Pop-ups:

All pop-ups described in this web app will take up most of the page, having a close button in the top right of the pop-up, as well if the user clicks on the blurred background page outside of the pop-up since the pop-up doesn't take up the entire webpage, the pop-up will exit, and user will be back to where they were originally.

Entire website description:

Homepage:

-The user enters the web application through their browser of choice which is most likely hosted on GitHub pages as it is free. The user is greeted with 2 main giant buttons at the center of the screen, the left one says mosaics, and the right one says 3d models. Behind these 2 buttons is the background image which will be a collage of all the images taken for the creation of this project, it will just be a singular image file that is edited the way the creators see fit.

-Masthead:
There is a "masthead" bar at the top of the website at all times, clearly distinct from the usable webpage below, that is to say, it is a "white bar" on top containing the things that need to be displayed at all times while on the website. There is centered text in the "masthead" of the web app with the name "Brick It" (tentative to change), on the left of the "masthead" within the same "bar" is the text " University of Waterloo Winter 2025 ME101 Term Design Project", with the Logo of University of Waterloo on the furthest left of the text next to the edge of the website, the text wraps around nicely so as to not be to close to the title of web app "Brick It". If the user presses on any area of this "media" the website will open in another tab tentatively the University of Waterloo, Mechanical engineering landing page, here's the link: https://uwaterloo.ca/mechanical-mechatronics-engineering/. On the far right of the "masthead" are the creators of the project in the following order "Kingsley Fong, Adam Benaissa, Victor Constantin Radu, Joseph Schuurman" The text wraps but ensures that nobody's first and last names are cut off, that is to say, the first name and last name are always together. There will be

2 names max in a row separated by a comma like a 2x2 matrix. Again this text wraps aesthetically so as to not be too close to the title of the project “Brick It”, on the farthest right of the creators of the project is the text taking up the same size as the University of Waterloo logo instead with the words “Group 17” wrapped so “group” and “17” are on different lines. All of the names when pressed will open another tab that will display each creators linkedin page, currently for Kingsley Fong, here’s his LinkedIn website: www.linkedin.com/in/kingsley-fong .

-At any time on any page if the user presses on title of the website “Brick It”, it will take them back to this homepage acting as a “home button”, while saving the progress of what the user imputed before.

Header bar:
There will be a header bar right below the masthead at all times with different buttons that will perform different functions as described below, some buttons will only appear when the is on a specific “page” of the site which will be explained in the following description as well. Order of buttons starting from the Left of the header bar. Buttons will all be the color taking up the space needed regarding the text it is displaying within the header bar:

- Connect: There will be an active button no matter what “page” the user is on, once the user presses this button an aesthetic pop-up will appear in the center of the screen, and the user will not be able to access anything else until they finish with the connect pop-up. The pop-up will have a list of “connection mediums” listed out below. Next to each of the following options/”connection medium” will be an active checkmark showing which current option is selected. Selecting any of the options will load another pop up replacing the connection pop-up which has a progress bar including a percentage, with a message box directly above, showing all the connection tests, data logs and general steps that the application is trying to make while trying to connect to the selected connection medium, and of course the newly selected media method will now have the checkbox indicating that it is selected. And of course, all future connection with the LEGO Mindstorms EV3 will be through the media method that the user currently has selected, unless the user selects another “connection medium. Of course if the selected media method fails to connect properly, the web app will display an error/failure message and re-prompt the user with the same pop-up menu with connection options to select another connection method”):
    - Text file: This is the default connection method the web app will use, and with this connection method selected, the web app will just export the text files with the data/coordinates that are necessary for the LEGO print. The text file will go through the protocol to be downloaded to the user’s computer when the user presses the “Print LEGO” button on the far right of the header bar which will be described later.
    - USB: when pressed by the user the website will attempt to connect to the LEGO Mindstorms EV3 through USB protocol and will verify through some test and checking method and establish a working signal with it. With the progress bar and

    - Bluetooth: when pressed by the user the website will attempt to connect to the LEGO Mindstorms EV3 through the Bluetooth protocol and will verify through some test and checking method and establish a working signal with it. With the progress bar and “log” messages
    - Wifi: when pressed by the user the website will attempt to connect to the LEGO Mindstorms EV3 through the wifi protocol, and verify and establish a working signal with it. With the progress bar and “log” messages
- Color-setup:
  Pressing this button will result in a pop-up menu, which it will prompt the user to input how many dispensers the printer has currently after the user enters this data, it will then display each count of the dispensers down a scrollable table, The left column will have a header of “Dispenser Number”, and down the column will be the integer incrementing number of dispensers that the user just imputed of course starting at 1. The second column will have a header title of “Color”, and each cell will have a drop-down button that reveals a section of options for the user to select what dispensers have what colors within them. To keep it simple, let’s only have the current options of: black, white, red, green, blue, yellow, orange, brown, and grey. Please ensure that the user cannot input the same color in more than 1 dispenser, that is to say, when a user selects a color for a dispenser, for the following drop-down menus they continue to press, the option they already “used” will be “greyed out” and not pressable anymore for the user. At the bottom of the pop-up is a “Save data” button which will save the user imputed data unless the user changes it, and will be vital when outputting the render of the prints and data/instructions for the prints.

3D Models:

When the user presses the 3D Models button from the “homepage”, the website will go to its dedicated 3D Models page

3D viewports

-3d viewports: all three 3d viewports will zoom, pan, move, etc, and manipulate simultaneously when the user interacts with any of them, therefore, they are all “synced” together. All three 3d viewports will have the same origin point on the same corner of the LEGO-rendered 16x16 stud LEGO plate. For all three build plates, in one direction along the build plate edge starting from the origin will be the x-axis, and along the other edge of the build plate from the origin will be the y-axis, and straight up from the origin will be the z-axis up that will reach up to the height of the real-life dimensions ten 1x1 LEGO pieces high as our maximum build height. You should have extended lines with ticks at intervals that represent the width of a LEGO brick for both the x and y-axis or intervals that represent the height of a LEGO brick for the z-axis along the entire distance following all the axes respectively that only extend to the ends of the build plate and to

the height of ten 1x1 LEGO bricks high as a visual aid for the user. The X-axis line should be red, the Y-axis line should be green and the Z-axis line should be blue. I must restate that this should be applied to all three 3d viewports. The middle viewport “3d grid coordinate viewport” and the right viewport “LEGO rendered viewport” will not be loaded in and rendered unless the user has successfully loaded in an STL object and pressed the LEGOized button mentioned later in the header bar):

- The left 3d viewport will be a normal STL object file render viewport showing the model to-scale with its real-life dimensions on top of the center of the to-scale 16x16 LEGO plate with studs rendered with real-life dimensions for the STL object file model. On the left of this 3d viewport will be a vertical slide bar completely separate of the 3d viewport where you can scale the model up or down by a multiplier, so the scaling starts at the center at 1x and going up can scale it to 5x, and down to 0.1x, this slide bar will directly scale the STL object while still being centered on top of the LEGO 16x16 build plate. This feature allows the user to ensure the size of the 3d LEGO model they want to be rendered. Below the vertical slider bar is a “Scale” button, where the currently selected scale multiplier will render in for the left 3d viewport only when the user presses the “Scale” button, this ensures that the program isn’t constantly trying to render in whatever value the vertical slider is set to, saving computer power.
- The middle 3d viewport will be a render of a 3d rectangular cube grid overlaying the previous 3d viewport with the STL object file model, which has the dimensions of a 1x1 LEGO brick that starts at the origin spanning completely to the other ends both in the X and Y axis, meaning it should overlay and cover exactly 16x16 LEGO brick dimensions perfectly, and it should extend ten 1x1 LEGO pieces high, and for each of the 3d grid rectangles they will be to-size of the real-life dimensions of 1x1 lego pieces. At the center bottom of all of the rectangular cube grids that are to-scale real-life 1x1 LEGO bricks, any of the grids that have an STL object point within it means that a real-life LEGO brick will be “printed” there to recreate the model when “printing”, for these grid rectangular cubes, there will be a grid point with XYZ values in terms of LEGO dimensions representing bricks that must be “printed” there to recreate the model with 1x1 LEGO bricks. That is to say X & Y coordinates will have a domain of 1-16 representing the 1-16 studs on the 16x16 stud LEGO plate we are building on, and the Z coordinate will have a range of 1-10 as we only want to only build to a maximum of 10 LEGO bricks high. Within the 3d viewport, these model bricks will be rendered in a green color. You must also do calculations to determine which 1x1 LEGO bricks will be printed “free-floating” within the real-life space, that is to say, you must determine which points will have LEGO bricks that don’t have an adjacent connection path to another 3d rectangular grid point vertically all the way down reaching either the build plate or another 1x1 LEGO brick, for these points, just like real life, they will be considered “support bricks” and will be rendered as support brick points, and for these support brick points within the 3d viewport, they will be rendered as a red color. If the user has a text file selected currently as their connection media that is described below. The values of all the coordinates for bricks to be printed both model and support must be put onto a Txt file with the columns representing the following and then exported after the user has pressed the print button described below:

X-coordinate, Y-coordinate, Z-coordinate, Brick color(represented by an integer value corresponding to the dispenser number data from the color set-up)

- The right 3d viewport will be a “LEGOized” render of what the actual print will look like, with the points previously in the middle viewport having a rendered LEGO brick filling the rectangular grid position. And of course, for the support brick points, there will also be a support LEGO brick rendered in its 3d rectangular grid position however in a contrasting color, the default color will be white for the 3d model bricks, and the “support bricks” will be a contrasting color of black, these colors can be changed by the user in the paint button. All the LEGO bricks that are at the furthermost top of their column in the 3d space of the 3d viewport will be rendered with LEGO studs on top, just like in real life, as you stack LEGO bricks, you cannot see the studs of the bricks below, so you will only see the studs of the LEGO pieces on top.

Header bar:

This will be the selection of buttons within the header bar continuing from the permanent buttons described before continuing from left to right whenever we are on the 3D Models “page”.

- Pre-loaded models:
  In this pop-up menu, there will be a selection of pre-loaded STL 3d models that are ready to be loaded and “LEGOized” when the user selects the name of a model that they would like to load in for example: “Pikachu”. The program will then exit the pop-up, load in all three 3d viewports with the selected pre-loaded model that is saved within the web app backend. Allowing the user to pan and move the 3d viewports so that they can view the model at any angle they would like, scale the model to their liking, or the user can then either choose another pre-loaded model, upload their own file in the next button, or press the “Print!” button on the far right of header bar described later if they are happy with the model chosen which will then perform the specific actions depending on their connection “media method”.
- Upload:
  If the user clicks this button there will be a pop-menu window that pops up on the center of the screen blurring the background as well, just like a lot of other file upload “windows” on other web apps, which will have an area that allows the user to “drag and drop/copy” strictly STL object files, or click on the entire area which is a button in itself button which will open the user's computer’s file explorer to choose an STL file to open and upload. If the user opens a file that isn’t an STL object file, the app will of course tell the user that the file must be an STL object file in a warning message on the bottom of the pop-up and the user can upload a valid file.
- Paint: This button will stay “greyed out”/Unpressable until the user has uploaded a valid STL object model, and then pressed the “LEGOized” button described later as well, and the user must also have successfully entered all data necessary from the color set-up button described previously. Pressing the paint button will cause a pop-up which first only shows 2 buttons. Left button “Static Colors”. Right button, “Paint Me!
  - Static Colors: if the user presses the static color button, within the pop-up page it will then prompt the user with 2 selections: “Model Color” and “Support Color”, to

the right of both selections will be drop-down menu's with the color data down a list along with the dispenser number they are in that the user should have already provided. Implement the feature where the user cannot choose the same color for both "Model Color" and "Support Color", that is to say, if the user for example chooses "Blue, 1" (color blue in dispenser 1) as the "Model Color", when they select the drop-down menu for "Support Color" the option "Blue, 1" is "Greyed-out" and not selectable for the user, ensuring the user selects another color

    - Paint Me!: if the user presses the Paint Me button the app will then have a sort of a color palette appear in the LEGO rendered 3d viewport on the right 3d viewport, all the colors that the user imputed during set-up appear as "squares" that the user can press on. If currently selected a color, it acts like a "paint brush", now whatever "LEGO brick" they select within the LEGO rendered 3d view port will be rendered as the color they just "painted" it as. This allows the user to "paint" the specific colors right onto a 3d model right away. If the user at any point chooses static colors, all the data that they might have just "painted" will disappear as the colors will become the static color that they chose.

- LEGOize:
  This button will stay "greyed out"/Unpressable until the user has first successfully uploaded an STL object model, and completed the color setup as well. Pressing this button will then have the application calculate and render in the rest of the 2 3d viewports, the "coordinate" viewport, and the LEGO rendered viewport, middle and right 3d viewports respectively. By default the LEGO-rendered viewport will have the 3d model bricks rendered as white, and then the support bricks rendered as black. The user can then change the colors using the paint button if they choose

  Print:
  Once the user presses this button, the web app will perform its commands based on its currently selected connection medium. For example if text file, the app will then just download the text file to the user's computer formatted in the description described within this document

---

Current progress

Currently, I have created an application in Python that almost reaches all the requirements previously laid in this document. The ideal solution would be to turn these python functions and implement them within a web application to the functionality as described before:

```
import numpy as np
import trimesh
```

```
import pyvista as pv
from pyvistaqt import QtInteractor
from PyQt5.QtWidgets import (QApplication, QFileDialog, QMainWindow,
QPushButton, QMessageBox,
                             QSlider, QVBoxLayout, QHBoxLayout, QWidget,
QLabel, QProgressBar, QFrame)
from PyQt5.QtCore import Qt, QThread, pyqtSignal, QTimer
import sys
import os

class STLProcessingThread(QThread):
    """Thread for processing STL files to avoid UI freezing."""
    progress_signal = pyqtSignal(int)
    finished_signal = pyqtSignal(object, object, object, object, object,
object)

    def __init__(self, mesh, voxel_size, stl_file_path):
        super().__init__()
        self.mesh = mesh
        self.voxel_size = voxel_size
        self.stl_file_path = stl_file_path  # Store the STL file path

    def run(self):
        # Update progress - Voxelization starting
        self.progress_signal.emit(5)

        # Voxelize the mesh
        voxel_grid = self.mesh.voxelized(pitch=self.voxel_size)
        voxel_matrix = voxel_grid.matrix.astype(bool)

        # Update progress - Voxelization complete
        self.progress_signal.emit(30)

        if voxel_matrix.size == 0:
            print("Warning: Voxel grid is empty.")
            return

        # Get origin
        origin = self.mesh.bounds[0]
```

```
        # Update progress - Starting LEGO brick generation
        self.progress_signal.emit(40)

        # Generate LEGO bricks
        lego_bricks, support_bricks =
self.generate_lego_bricks(voxel_matrix, origin, self.voxel_size)

        # Update progress - LEGO brick generation complete
        self.progress_signal.emit(70)

        # Export coordinates
        self.export_voxel_coordinates(voxel_matrix, origin,
self.voxel_size)

        # Update progress - Coordinates exported, preparing visualization
        self.progress_signal.emit(80)

        # Signal completion with results
        self.finished_signal.emit(voxel_grid, voxel_matrix, lego_bricks,
support_bricks, origin, self.voxel_size)

    def generate_lego_bricks(self, voxel_matrix, origin, voxel_size):
        """Generate LEGO bricks from voxel matrix."""
        # Create empty lists for bricks and studs
        bricks = []
        studs = []
        support_bricks = []

        # Get dimensions of voxel matrix
        x_dim, y_dim, z_dim = voxel_matrix.shape

        # Create a matrix to track support bricks
        support_matrix = np.zeros_like(voxel_matrix, dtype=bool)

        # Identify support bricks
        for x in range(x_dim):
            for y in range(y_dim):
                filled_indices = np.where(voxel_matrix[x, y, :])[0]
                if len(filled_indices) > 0:
                    lowest_z = filled_indices[0]
```

```
                    for z in range(lowest_z):
                        support_matrix[x, y, z] = True

        # Generate model bricks
        for z in range(z_dim):
            for y in range(y_dim):
                for x in range(x_dim):
                    if voxel_matrix[x, y, z]:
                        # Create a brick at this position
                        brick = self.create_lego_brick(origin, x, y, z,
voxel_size)
                        bricks.append(brick)

                        # Create a stud on top of the brick
                        stud = self.create_lego_stud(origin, x, y, z,
voxel_size)
                        studs.append(stud)

        # Generate support bricks
        for z in range(z_dim):
            for y in range(y_dim):
                for x in range(x_dim):
                    if support_matrix[x, y, z] and not voxel_matrix[x, y,
z]:
                        # Create a support brick at this position
                        support_brick = self.create_lego_brick(origin, x,
y, z, voxel_size)
                        support_bricks.append(support_brick)

        return [bricks, studs], support_bricks

    def create_lego_brick(self, origin, x, y, z, voxel_size):
        """Create a LEGO brick at the specified position."""
        # LEGO brick dimensions (slightly smaller than voxel for
visualization)
        brick_size = voxel_size * 0.95

        # Calculate position
        pos_x = origin[0] + x * voxel_size
        pos_y = origin[1] + y * voxel_size
```

```
        pos_z = origin[2] + z * voxel_size

        # Create brick
        brick = pv.Cube(center=(pos_x + voxel_size/2, pos_y +
voxel_size/2, pos_z + voxel_size/2),
                        x_length=brick_size, y_length=brick_size,
z_length=brick_size)
        return brick

    def create_lego_stud(self, origin, x, y, z, voxel_size):
        """Create a LEGO stud at the specified position."""
        # LEGO stud dimensions
        stud_radius = voxel_size * 0.2
        stud_height = voxel_size * 0.1

        # Calculate position
        pos_x = origin[0] + x * voxel_size + voxel_size/2
        pos_y = origin[1] + y * voxel_size + voxel_size/2
        pos_z = origin[2] + z * voxel_size + voxel_size

        # Create stud
        stud = pv.Cylinder(center=(pos_x, pos_y, pos_z + stud_height/2),
                           direction=(0, 0, 1), radius=stud_radius,
height=stud_height)
        return stud

    def export_voxel_coordinates(self, voxel_matrix, origin, voxel_size):
        """Export voxel coordinates to a text file."""
        # Extract the base name of the STL file and create the custom TXT
file name
        base_name =
os.path.splitext(os.path.basename(self.stl_file_path))[0]
        txt_file_name = f"{base_name}_lego.txt"

        # Get dimensions of voxel matrix
        x_dim, y_dim, z_dim = voxel_matrix.shape

        # Create a matrix to track support bricks
        support_matrix = np.zeros_like(voxel_matrix, dtype=bool)
```

```
        # Identify support bricks
        for x in range(x_dim):
            for y in range(y_dim):
                filled_indices = np.where(voxel_matrix[x, y, :])[0]
                if len(filled_indices) > 0:
                    lowest_z = filled_indices[0]
                    for z in range(lowest_z):
                        support_matrix[x, y, z] = True

        # Create output file
        with open(txt_file_name, "w") as f:
            f.write("# LEGO Coordinates (X, Y, Z, Type)\n")
            f.write("# Type: 1 = Model Brick, 2 = Support Brick\n")

            # Write model brick coordinates
            for z in range(z_dim):
                for y in range(y_dim):
                    for x in range(x_dim):
                        if voxel_matrix[x, y, z]:
                            # Convert to LEGO coordinates (1-based)
                            lego_x = int(round((origin[0] + x *
voxel_size) / voxel_size)) + 1
                            lego_y = int(round((origin[1] + y *
voxel_size) / voxel_size)) + 1
                            lego_z = z + 1

                            # Write to file if within buildable area
                            if 1 <= lego_x <= 16 and 1 <= lego_y <= 16 and
1 <= lego_z <= 10:
                                f.write(f"{lego_x}, {lego_y}, {lego_z},
1\n")

            # Write support brick coordinates
            for z in range(z_dim):
                for y in range(y_dim):
                    for x in range(x_dim):
                        if support_matrix[x, y, z] and not voxel_matrix[x,
y, z]:
                            # Convert to LEGO coordinates (1-based)
```

```
                            lego_x = int(round((origin[0] + x *
voxel_size) / voxel_size)) + 1
                            lego_y = int(round((origin[1] + y *
voxel_size) / voxel_size)) + 1
                            lego_z = z + 1

                            # Write to file if within buildable area
                            if 1 <= lego_x <= 16 and 1 <= lego_y <= 16 and
1 <= lego_z <= 10:
                                f.write(f"{lego_x}, {lego_y}, {lego_z},
2\n")

        # Print confirmation message to terminal
        print(f"LEGO coordinates exported successfully to
'{txt_file_name}'")
        print(f"File saved in: {os.path.abspath(txt_file_name)}")

class LegoSlicerApp(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setWindowTitle("LEGOized Slicer")
        # Start in normal windowed mode with a reasonable size
        self.resize(1200, 800)

        # Create central widget
        central_widget = QWidget()
        self.setCentralWidget(central_widget)

        # Main layout with header and viewports
        main_layout = QVBoxLayout(central_widget)
        main_layout.setContentsMargins(10, 10, 10, 10)

        # Header bar with controls
        header_bar = QWidget()
        header_layout = QHBoxLayout(header_bar)
        header_layout.setContentsMargins(0, 0, 0, 10)
```

```
        # STL file selection button
        self.select_stl_button = QPushButton("Choose STL File", self)
        self.select_stl_button.setMinimumWidth(120)
        self.select_stl_button.clicked.connect(self.select_stl_file)
        header_layout.addWidget(self.select_stl_button)

        # Scale label
        scale_label = QLabel("Scale Factor:")
        header_layout.addWidget(scale_label)

        # Scale slider
        self.scale_slider = QSlider(Qt.Horizontal)
        self.scale_slider.setMinimum(1)  # 0.1x
        self.scale_slider.setMaximum(100)  # 10.0x
        self.scale_slider.setValue(10)  # 1.0x (default)
        self.scale_slider.setTickPosition(QSlider.TicksBelow)
        self.scale_slider.setTickInterval(10)
        self.scale_slider.valueChanged.connect(self.update_scale_label)
        header_layout.addWidget(self.scale_slider, 1)  # Give slider more
space

        # Scale value label
        self.scale_value_label = QLabel("1.0x")
        self.scale_value_label.setMinimumWidth(50)
        header_layout.addWidget(self.scale_value_label)

        # Scale button
        self.scale_button = QPushButton("SCALE", self)
        self.scale_button.setMinimumWidth(80)
        self.scale_button.clicked.connect(self.apply_scale)
        self.scale_button.setEnabled(False)  # Disabled until STL is
loaded
        header_layout.addWidget(self.scale_button)

        # Reset View button (replacing Fullscreen button)
        self.reset_view_button = QPushButton("Reset View", self)
        self.reset_view_button.setMinimumWidth(80)
        self.reset_view_button.clicked.connect(self.reset_all_views)
        header_layout.addWidget(self.reset_view_button)
```

```
        # Add header to main layout
        main_layout.addWidget(header_bar)

        # Progress bar
        self.progress_bar = QProgressBar()
        self.progress_bar.setVisible(False)  # Hide initially
        main_layout.addWidget(self.progress_bar)

        # Viewports container
        viewports_container = QWidget()
        self.viewports_layout = QHBoxLayout(viewports_container)
        self.viewports_layout.setContentsMargins(0, 0, 0, 0)
        main_layout.addWidget(viewports_container, 1)  # Give viewports
more space

        # Create frames for each viewport
        self.viewport_frames = []
        self.viewport_plotters = []

        for i in range(3):
            # Create a frame to hold the viewport
            frame = QFrame()
            frame.setFrameShape(QFrame.StyledPanel)
            frame.setFrameShadow(QFrame.Raised)
            frame.setMinimumSize(300, 300)

            # Create a layout for the frame
            frame_layout = QVBoxLayout(frame)
            frame_layout.setContentsMargins(0, 0, 0, 0)

            # Create a PyVista QtInteractor for this frame
            plotter = QtInteractor(frame)
            # Set background color to light grey
            plotter.set_background('lightgrey')
            frame_layout.addWidget(plotter)

            # Add to layouts and lists
            self.viewports_layout.addWidget(frame)
            self.viewport_frames.append(frame)
            self.viewport_plotters.append(plotter)
```

```
        # Set up initial buildplates
        self.setup_initial_viewports()

        # Initialize variables
        self.stl_file = None
        self.mesh = None
        self.original_mesh = None
        self.current_scale = 1.0

    def setup_initial_viewports(self):
        """Set up initial empty viewports with buildplates."""
        viewport_titles = ["Original STL Model", "Voxelized Model",
"LEGOized Model"]

        for i, plotter in enumerate(self.viewport_plotters):
            # Add buildplate
            buildplate = self.create_lego_buildplate()
            plotter.add_mesh(buildplate, color="gray", opacity=0.5)

            # Add axes
            plotter.show_axes()
            self.add_origin_axes(plotter)

            # Add title
            plotter.add_text(viewport_titles[i], position='upper_left')

            # Reset camera
            plotter.view_isometric()
            plotter.reset_camera()

    def link_viewport_cameras(self):
        """Link the camera views of all viewports."""
        # Get the camera position from the first viewport
        camera_pos = self.viewport_plotters[0].camera_position

        # Apply to other viewports
        for plotter in self.viewport_plotters[1:]:
            plotter.camera_position = camera_pos
```

```
    def update_scale_label(self):
        """Update the scale label when the slider changes."""
        scale_value = self.scale_slider.value() / 10.0
        self.scale_value_label.setText(f"{scale_value:.1f}x")

    def apply_scale(self):
        """Apply the selected scale to the STL model."""
        if not self.original_mesh:
            return

        # Get scale value
        scale_value = self.scale_slider.value() / 10.0
        self.current_scale = scale_value

        # Create a copy of the original mesh
        self.mesh = self.original_mesh.copy()

        # Apply scale
        self.mesh.apply_scale(scale_value)

        # Place on buildplate
        self.place_on_buildplate()

        # Process the model
        self.process_model()

    def select_stl_file(self):
        """Open a file dialog to select an STL file."""
        file_path, _ = QFileDialog.getOpenFileName(self, "Open STL File",
"", "STL Files (*.stl)")
        if file_path:
            self.stl_file = file_path
            # Load the mesh
            self.mesh = trimesh.load_mesh(file_path, process=True)
            # Convert from mm to cm (STL files typically use mm)
            self.mesh.apply_scale(0.1)
            # Store original mesh for scaling operations
            self.original_mesh = self.mesh.copy()
            # Enable scale button
            self.scale_button.setEnabled(True)
```

```
            # Place on buildplate without scaling
            self.place_on_buildplate()
            # Process the model
            self.process_model()

    def place_on_buildplate(self):
        """Centers the STL model and places it on the buildplate without
scaling."""
        if self.mesh:
            # Center the model in X and Y
            centroid = self.mesh.centroid
            self.mesh.apply_translation([-centroid[0], -centroid[1], 0])

            # Move to center of buildplate
            self.mesh.apply_translation([6.4, 6.4, 0])

            # Ensure model sits on buildplate
            min_z = self.mesh.bounds[0][2]
            self.mesh.apply_translation([0, 0, -min_z + 0.16])

    def create_lego_buildplate(self):
        """Creates a 16x16 LEGO build plate with real-world dimensions."""
        buildplate = pv.Plane(center=(6.4, 6.4, 0), i_size=12.8,
j_size=12.8)
        return buildplate

    def process_model(self):
        """Process the STL model in a separate thread with progress
updates."""
        if not self.mesh:
            return

        # Show progress bar
        self.progress_bar.setValue(0)
        self.progress_bar.setVisible(True)

        # Create and start processing thread, passing the STL file path
        self.processing_thread = STLProcessingThread(self.mesh,
voxel_size=0.78, stl_file_path=self.stl_file)
```

```
self.processing_thread.progress_signal.connect(self.update_progress)

self.processing_thread.finished_signal.connect(self.display_processed_mode
l)
        self.processing_thread.start()

    def update_progress(self, value):
        """Update the progress bar."""
        self.progress_bar.setValue(value)

    def display_processed_model(self, voxel_grid, voxel_matrix,
lego_bricks, support_bricks, origin, voxel_size):
        """Display the processed model in the 3D viewports."""
        # Update progress - Starting to render models
        self.progress_bar.setValue(85)

        # Check if model is too big for buildable area
        model_too_big = self.check_model_size(voxel_matrix, origin,
voxel_size)

        # Count bricks
        model_bricks, support_bricks_count, outside_bricks =
self.count_bricks(voxel_matrix, origin, voxel_size)
        total_bricks = model_bricks + support_bricks_count

        # Clear all viewports
        for plotter in self.viewport_plotters:
            plotter.clear()
            # Ensure background color is set to light grey
            plotter.set_background('lightgrey')

        # Update progress - Cleared viewports
        self.progress_bar.setValue(90)

        # Set up each viewport with a buildplate and axes
        for plotter in self.viewport_plotters:
            buildplate = self.create_lego_buildplate()
            plotter.add_mesh(buildplate, color="gray", opacity=0.5)
            plotter.show_axes()
```

```
            self.add_origin_axes(plotter)

        # Original Model Viewport
        plotter = self.viewport_plotters[0]
        pv_mesh = pv.wrap(self.mesh)
        plotter.add_mesh(pv_mesh, color="#39FF14", opacity=0.8)
        plotter.add_text("Original STL Model", position='upper_left')
        plotter.add_text(f"Scale: {self.current_scale:.1f}x",
position='upper_right')

        # Update progress - Original model rendered
        self.progress_bar.setValue(93)

        # Voxelized Model Viewport
        plotter = self.viewport_plotters[1]
        voxel_points = voxel_grid.points
        plotter.add_points(voxel_points, color="red", point_size=10,
render_points_as_spheres=True)
        # Add support voxel points
        support_points = np.array([origin + np.array([x, y, z]) *
voxel_size for x, y, z in zip(*np.where(voxel_matrix)) if not
np.any(voxel_matrix[x, y, :z])])
        if len(support_points) > 0:  # Check if there are any support
points
            plotter.add_points(support_points, color="blue",
point_size=10, render_points_as_spheres=True)
        plotter.add_text("Voxelized Model", position='upper_left')
        self.add_grid_overlay(plotter, voxel_grid)

        # Update progress - Voxelized model rendered
        self.progress_bar.setValue(96)

        # LEGOized Model Viewport
        plotter = self.viewport_plotters[2]
        for brick in lego_bricks[0]:
            plotter.add_mesh(brick, color="#4169E1", opacity=1.0)
        for stud in lego_bricks[1]:
            plotter.add_mesh(stud, color="black", opacity=1.0)
        # Add support bricks
        for support_brick in support_bricks:
```

```
            plotter.add_mesh(support_brick, color="black", opacity=1.0)
        plotter.add_text("LEGOized Model", position='upper_left')

        # Add brick count information at the bottom of the viewport
        brick_count_text = f"Total: {total_bricks} | Model: {model_bricks}
| Support: {support_bricks_count}"
        plotter.add_text(brick_count_text, position=(0.05, 0.02),
font_size=12, color='black', viewport=True)

        # Add warning if model is too big
        if model_too_big:
            plotter.add_text("MODEL SIZE TOO BIG - RENDER ISSUES",
position='lower_left', color='red', font_size=14)

        # Update progress - LEGOized model rendered
        self.progress_bar.setValue(99)

        # Reset camera and update all viewports
        for plotter in self.viewport_plotters:
            plotter.reset_camera()
            plotter.update()

        # Link the camera views
        self.link_viewport_cameras()

        # Update progress - All rendering complete
        self.progress_bar.setValue(100)

        # Hide progress bar after a short delay
        QTimer.singleShot(500, lambda:
self.progress_bar.setVisible(False))

    def add_origin_axes(self, plotter):
        """Add XYZ axis lines from the origin at the corner of the
buildplate."""
        origin = [0, 0, 0]
        x_axis = pv.Line(origin, [12.8, 0, 0])
        y_axis = pv.Line(origin, [0, 12.8, 0])
        z_axis = pv.Line(origin, [0, 0, 12.8])
        plotter.add_mesh(x_axis, color="red", line_width=2)
```

```
        plotter.add_mesh(y_axis, color="green", line_width=2)
        plotter.add_mesh(z_axis, color="blue", line_width=2)

    def add_grid_overlay(self, plotter, voxel_grid):
        """Add a 3D grid overlay to the voxelized model viewport."""
        buildplate_size = 12.8
        x_range = np.arange(0, buildplate_size, 0.78)
        y_range = np.arange(0, buildplate_size, 0.78)
        z_range = np.arange(0, buildplate_size, 0.96)

        for x in x_range:
            for y in y_range:
                plotter.add_mesh(pv.Line((x, 0, 0), (x, 0,
buildplate_size)), color="black", opacity=0.1)
                plotter.add_mesh(pv.Line((x, buildplate_size, 0), (x,
buildplate_size, buildplate_size)), color="black", opacity=0.1)

        for y in y_range:
            for z in z_range:
                plotter.add_mesh(pv.Line((0, y, z), (buildplate_size, y,
z)), color="black", opacity=0.1)

    def check_model_size(self, voxel_matrix, origin, voxel_size):
        """Check if any part of the model is outside the buildable
area."""
        x_dim, y_dim, z_dim = voxel_matrix.shape
        for z in range(z_dim):
            for y in range(y_dim):
                for x in range(x_dim):
                    if voxel_matrix[x, y, z]:
                        lego_x = int(round((origin[0] + x * voxel_size) /
voxel_size)) + 1
                        lego_y = int(round((origin[1] + y * voxel_size) /
voxel_size)) + 1
                        lego_z = z + 1
                        if not (1 <= lego_x <= 16 and 1 <= lego_y <= 16
and 1 <= lego_z <= 10):
                            return True
        return False
```

```
    def count_bricks(self, voxel_matrix, origin, voxel_size):
        """Count model bricks, support bricks, and bricks outside
buildable area."""
        x_dim, y_dim, z_dim = voxel_matrix.shape

        model_bricks = 0
        support_bricks = 0
        outside_bricks = 0

        # Create a matrix to track support bricks
        support_matrix = np.zeros_like(voxel_matrix, dtype=bool)

        # Identify support bricks
        for x in range(x_dim):
            for y in range(y_dim):
                filled_indices = np.where(voxel_matrix[x, y, :])[0]
                if len(filled_indices) > 0:
                    lowest_z = filled_indices[0]
                    for z in range(lowest_z):
                        support_matrix[x, y, z] = True

        # Count bricks
        for z in range(z_dim):
            for y in range(y_dim):
                for x in range(x_dim):
                    if voxel_matrix[x, y, z] or support_matrix[x, y, z]:
                        # Convert to LEGO coordinates
                        lego_x = int(round((origin[0] + x * voxel_size) /
voxel_size)) + 1
                        lego_y = int(round((origin[1] + y * voxel_size) /
voxel_size)) + 1
                        lego_z = z + 1

                        if 1 <= lego_x <= 16 and 1 <= lego_y <= 16 and 1
<= lego_z <= 10:
                            if voxel_matrix[x, y, z]:
                                model_bricks += 1
                            else:
                                support_bricks += 1
                        else:
```

```
                            outside_bricks += 1

        return model_bricks, support_bricks, outside_bricks

    def reset_all_views(self):
        """Reset all viewport cameras to the default isometric view."""
        for plotter in self.viewport_plotters:
            plotter.reset_camera()
            # Set to isometric view
            plotter.view_isometric()
            plotter.update()

    def closeEvent(self, event):
        """Handle the window close event to properly clean up
resources."""
        # Stop any running processing threads
        if hasattr(self, 'processing_thread') and
self.processing_thread.isRunning():
            self.processing_thread.terminate()
            self.processing_thread.wait()

        # Clean up PyVista plotters
        for plotter in self.viewport_plotters:
            plotter.close()
            plotter.deep_clean()

        # Accept the close event
        event.accept()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = LegoSlicerApp()
    window.show()
    sys.exit(app.exec_())
```

Mosaics

“Viewports”:

Taking up the main screen will be 2 media displays that are empty until the user chooses an image to “print” and turn into a mosaic. The left “viewport” will be the normal image displayed in a square crop at all times to ensure that when it is “LEGOized” it will be properly printed onto the 16 by 16 stud square “printed”. The right viewport will be the LEGOized version of the image, where there will be a 16x16 grid evenly overlaid on top of the square image. Each grid will take the strongest/most prominent color in the grid, and then find the closest color that the user has already imputed when they first do “color set-up”. The rendered image will look like a mosaic of the original image with colors that the user has imputed. The user can then press “Print It” to send the mosaic to be printed of course the default as mentioned before is exporting a txt file downloaded to the users computer with the LEGO coordinates, for the “Mosaics” exported coordinates all that is needed is X, Y, and Z coordinates speaking only in terms of the LEGO position. Of course the Z value being 1, as mosaics will only “print” one layer of LEGO as opposed to the “3D models” option. Of Course the X and Y coordinate values will be bounded to 1-16 as the build plate is only 16 by 16 studs. The bottom left of the image will be considered the 1, 1, 1; X, Y, Z coordinates with the X being horizontally across the image, and the Y being vertically up the image. Of course with each grid coordinate must also have a color integer as the last value, the color integer value will correspond to the imputed data the user must’ve imputed when doing the color set-up. The program will not allow “image loading” unless the user has completed the color set-up. For example in most top left corner of the image grid, the most prominent color is a light blue, if the user only has “blue, 6 “(dispenser 6 has blue 1x1 LEGO bricks), then the exported entry for that grid coordinate will be 1, 16, 1, 6; the x,y,z,color integer. Please order all the coordinates for each incrementing value of X first, incrementing the Y values secondarily. For example:

1,1,1,4
1,2,1,4
1,3,1,7
1,4,1,9
…(continuing)
2,1,1,8
2,2,1,4
…

Header bar:

This will be the selection of buttons within the header bar continuing from the permanent buttons described before continuing from left to right whenever we are on the 3D Models “page”.

- Pre-loaded pictures:
  In this pop-up menu, there will be a selection of pre-loaded picture files with a small render of the image next to the name so the user can easily select which image they that are ready to be loaded and “LEGOized” when the user selects the name of a model that they would like to load in for example: “CN Tower”. The program will then exit the pop-up, and then load in the image display, and LEGO render.

  load in all three 3d viewports with the selected pre-loaded model that is saved within the web app backend. Allowing the user to pan and move the 3d viewports so that they can view the model at any angle they would like, scale the model to their liking, or the user can then either choose another pre-loaded model, upload their own file in the next button, or press the “Print!” button on the far right of header bar described later if they are happy with the model chosen which will then perform the specific actions depending on their connection “media method”.
- Upload:
  If the user clicks this button there will be a pop-menu window that pops up on the center of the screen blurring the background as well, just like a lot of other file upload “windows” on other web apps, which will have an area that allows the user to “drag and drop/copy” strictly STL object files, or click on the entire area which is a button in itself button which will open the user's computer’s file explorer to choose an STL file to open and upload. If the user opens a file that isn’t an STL object file, the app will of course tell the user that the file must be an STL object file in a warning message on the bottom of the pop-up and the user can upload a valid file.
- Paint: This button will stay “greyed out”/Unpressable until the user has uploaded a valid STL object model, and then pressed the “LEGOized” button described later as well, and the user must also have successfully entered all data necessary from the color set-up button described previously. Pressing the paint button will cause a pop-up which first only shows 2 buttons. Left button “Static Colors”. Right button, “Paint Me!
  - Static Colors: if the user presses the static color button, within the pop-up page it will then prompt the user with 2 selections: “Model Color” and “Support Color”, to the right of both selections will be drop-down menu’s with the color data down a list along with the dispenser number they are in that the user should have already provided. Implement the feature where the user cannot choose the same color for both “Model Color” and “Support Color”, that is to say, if the user for example chooses “Blue, 1” (color blue in dispenser 1) as the “Model Color”, when they select the drop-down menu for “Support Color” the option “Blue, 1” is “Greyed-out” and not selectable for the user, ensuring the user selects another color

    - Paint Me!: if the user presses the Paint Me button the app will then have a sort of a color palette appear in the LEGO rendered 3d viewport on the right 3d viewport, all the colors that the user imputed during set-up appear as “squares” that the user can press on. If currently selected a color, it acts like a “paint brush”, now whatever “LEGO brick” they select within the LEGO rendered 3d view port will be rendered as the color they just “painted” it as. This allows the user to “paint” the specific colors right onto a 3d model right away. If the user at any point chooses static colors, all the data that they might have just “painted” will disappear as the colors will become the static color that they chose.
- LEGOize:
  This button will stay “greyed out”/Unpressable until the user has first successfully uploaded an STL object model, and completed the color setup as well. Pressing this button will then have the application calculate and render in the rest of the 2 3d viewports, the “coordinate” viewport, and the LEGO rendered viewport, middle and right 3d viewports respectively. By default the LEGO-rendered viewport will have the 3d model bricks rendered as white, and then the support bricks rendered as black. The user can then change the colors using the paint button if they choose

  Print:
  Once the user presses this button, the web app will perform its commands based on its currently selected connection medium. For example if text file, the app will then just download the text file to the user's computer formatted in the description described within this document

---

CURRENT MOSAIC CODE:

This is the current code that I made without this web app in mind, it just needs modification to meet my specifications for the web app.

```
import cv2
import numpy as np
import blend_modes
from openai import OpenAI
import urllib.request
from time import sleep

# whether to use AI or a custom image
useAI = False

# Prompt describing the desired image
object = "desired image"
```

```
# Filename of the custom image
filename = "customImage.png"

# Variables for the pixelation algorithm
usepixelation = True
imgSize = 32

# whether to use the lego overlay on images
useOverlay = True
legoOverlay = cv2.imread("lego_overlay.png", -1).astype(float)

# Define the BGR values of Lego colors
colors_bgr = {
    'white': (255,255,255),
    'dark bluish gray': (104, 110, 108),
    'light bluish gray': (169, 165, 160),
    'tan': (158,205,228),
    'reddish brown': (18,42,88),
    'bright pink': (200,173,228),
    'dark purple': (145,54,63),
    'blue': (191,85,0),
    'dark azure': (201,139,7),
    'green': (65,120,35),
    'lime': (11,233,187),
    'yellow': (55,205,242),
    'orange': (24,138,254),
    'red': (9,26,201),
    'black': (0,0,0)
}

# Define the amount of usable Lego bricks in the machine
color_amount = 23
colors_stock = {
    'dark bluish gray': color_amount,
    'light bluish gray': color_amount,
    'tan': color_amount,
    'reddish brown': color_amount,
    'bright pink': color_amount,
    'dark purple': color_amount,
```

```
    'blue': color_amount,
    'dark azure': color_amount,
    'green': color_amount,
    'lime': color_amount,
    'yellow': color_amount,
    'orange': color_amount,
    'red': color_amount,
    'black': color_amount
}

# Function definitions from imageFunctions.py
def confirm_image(imgSize, object, AIfilename):
    original_image = cv2.imread(AIfilename)
    cv2.imwrite("result.png", original_image)
    exit_main_loop = False
    while not exit_main_loop:
        # Read and show the input image
        original_image = cv2.imread("result.png")
        cv2.imshow("Original Image", cv2.resize(original_image, (512,
512), interpolation=cv2.INTER_NEAREST))
        cv2.setWindowProperty("Original Image", cv2.WND_PROP_TOPMOST, 1)
        cv2.moveWindow("Original Image", 136,200)
        print('Image generated.\nPress: "C" to Confirm, "R" to Regenerate,
"T" to Trim.')
        while True:
            key = cv2.waitKey(1)
            if key == ord('c'):
                cv2.destroyAllWindows()
                original_image = cv2.imread(AIfilename)
                cv2.imwrite("result.png", original_image)
                exit_main_loop = True
                break
            elif key == ord('r'):
                cv2.destroyAllWindows()
                print(f'\nGenerating image from prompt: "{object}"\n')
                AIGenerator(object, AIfilename)
                original_image = cv2.imread(AIfilename)
                cv2.imwrite("result.png", original_image)
                break
            elif key == ord('t'):
```

```
                cv2.destroyAllWindows()
                ImgCropper(AIfilename, imgSize)
                exit_main_loop = True
                break

def pixelate_image(original_image, imgSize):
    height, width = original_image.shape[:2]
    rows, cols = imgSize, imgSize
    # Calculate the size of each square
    row_size = height // rows
    col_size = width // cols
    # Create a copy of the original image to apply pixelation
    pixelated_image = np.zeros((rows, cols, 3), dtype=np.uint8)

    for i in range(0, height, row_size):
        for j in range(0, width, col_size):
            center_pixel_color = original_image[i + row_size // 2, j +
col_size // 2]
            pixelated_image[i//row_size, j//col_size] = center_pixel_color

    return pixelated_image

def find_closest_color(pixel, color_dict):
    pixel_array = np.array(pixel)
    color_values = np.array(list(color_dict.values()))
    distances = np.linalg.norm(color_values - pixel_array, axis=1)
    closest_color_index = np.argmin(distances)
    closest_color = list(color_dict.keys())[closest_color_index]

    return color_dict[closest_color], closest_color

def simplify_image(image, colors_dict):
    recipe = {
        'red': 0,
        'orange': 0,
        'yellow': 0,
        'lime': 0,
        'green': 0,
        'dark azure': 0,
        'blue': 0,
```

```
        'dark purple': 0,
        'bright pink': 0,
        'reddish brown': 0,
        'tan': 0,
        'light bluish gray': 0,
        'dark bluish gray': 0,
        'black': 0,
        'white': 0
    }
    height, width, _ = image.shape
    mapped_image = np.zeros((height, width, 3), dtype=np.uint8)

    for i in range(height):
        for j in range(width):
            pixel = image[i, j]
            closest_color, closest_color_name = find_closest_color(pixel,
colors_dict)
            mapped_image[i, j] = closest_color
            recipe[closest_color_name] += 1

    del recipe['white']
    for i in list(recipe.keys()):
        if recipe[i] == 0:
            del recipe[i]

    return mapped_image, recipe

# Function definitions from imageGenerator.py
client = OpenAI(api_key='sk-YOUR KEY')  # Get your own by signing up for
OpenAI's website

def AIGenerator(object, AIfilename):
    text = f"I NEED to test how the tool works with extremely simple
prompts. DO NOT add any detail, just use it AS-IS: {object}. The most
simple 2D cartoon depiction of {object}, but still resembling real. Very
little detail colored clipart."

    # calling the custom function "generate"
    # saving the output in the file "result.jpg"
    url = generate(text)
```

```
    urllib.request.urlretrieve(url, AIfilename)

# function for text-to-image generation
# using create endpoint of DALL-E API
# function takes in a string argument
def generate(text):
    res = client.images.generate(
        model="dall-e-3",
        prompt=text,
        n=1,
        size="1024x1024",
    )
    # returning the URL of one image as we are generating only one image
    return res.data[0].url

# Function definitions from imageCropper.py
# This function requires a filename of the image and imgSize.
# imgSize is a number the final cropped square image must be dividable by,
in both the width and the height.

def mouse_crop(event, x_crop, y_crop, flags, param):
    global x_start_crop, y_start_crop, x_end_crop, y_end_crop, cropping,
initialCrop
    if event == cv2.EVENT_LBUTTONDOWN:
        x_start_crop, y_start_crop, x_end_crop, y_end_crop = x_crop,
y_crop, x_crop, y_crop
        cropping = True
    elif event == cv2.EVENT_MOUSEMOVE:
        if cropping == True:
            sideLength = max(abs(x_crop-x_start_crop),
abs(y_crop-y_start_crop))
            try:
                x_end_crop, y_end_crop =
int(x_start_crop+((x_crop-x_start_crop)/abs(x_crop-x_start_crop)*sideLengt
h)),
int(y_start_crop+((y_crop-y_start_crop)/abs(y_crop-y_start_crop)*sideLengt
h))
            except:
                x_end_crop, y_end_crop = x_start_crop, y_start_crop
    elif event == cv2.EVENT_LBUTTONUP:
```

```
        cropping = False
        initialCrop = True

def mouse_crop_adjust(event, x_crop, y_crop, flags, param):
    global x_start_crop, y_start_crop, x_end_crop, y_end_crop, cropping,
moveStartCrop, moveEndCrop, moveCrop, previous_x_crop, previous_y_crop,
croppedimg
    if event == cv2.EVENT_LBUTTONDOWN:
        cropping = True
        if abs(x_crop-x_start_crop) < 10 and abs(y_crop-y_start_crop) <
10:
            moveStartCrop = True
            moveEndCrop = False
            moveCrop = False
        elif abs(x_crop-x_end_crop) < 5 and abs(y_crop-y_end_crop) < 5:
            moveStartCrop = False
            moveEndCrop = True
            moveCrop = False
        elif x_start_crop < x_crop < x_end_crop and y_start_crop < y_crop
< y_end_crop:
            moveStartCrop = False
            moveEndCrop = False
            moveCrop = True
        else:
            moveStartCrop = False
            moveEndCrop = False
            moveCrop = False
    elif event == cv2.EVENT_MOUSEMOVE:
        if cropping == True:
            if moveStartCrop:
                sideLength = max(abs(x_crop-x_end_crop),
abs(y_crop-y-end_crop))
                try:
                    x_start_crop, y_start_crop =
int(x_end_crop+((x_crop-x_end_crop)/abs(x_crop-x_end_crop)*sideLength)),
int(y_end_crop+((y_crop-y_end_crop)/abs(y_crop-y-end_crop)*sideLength))
                except:
                    pass
            elif moveEndCrop:
```

```
                sideLength = max(abs(x_crop-x_start_crop),
abs(y_crop-y_start_crop))
                try:
                    x_end_crop, y_end_crop =
int(x_start_crop+((x_crop-x_start_crop)/abs(x_crop-x_start_crop)*sideLengt
h)),
int(y_start_crop+((y_crop-y_start_crop)/abs(y_crop-y_start_crop)*sideLengt
h))
                except:
                    pass
            elif moveCrop:
                x_start_crop = x_start_crop+x_crop-previous_x_crop
                y_start_crop = y_start_crop+y_crop-previous_y_crop
                x_end_crop = x_end_crop+x_crop-previous_x_crop
                y_end_crop = y_end_crop+y_crop-previous_y_crop
    elif event == cv2.EVENT_LBUTTONUP:
        cropping = False
        if x_end_crop < x_start_crop:
            x_start_crop = x_start_crop+x_end_crop
            x_end_crop = x_start_crop-x_end_crop
            x_start_crop = x_start_crop-x_end_crop
        if y_end_crop < y_start_crop:
            y_start_crop = y_start_crop+y_end_crop
            y_end_crop = y_start_crop-y_end_crop
            y_start_crop = y_start_crop-y_end_crop
        refPoint = [(x_start_crop, y_start_crop), (x_end_crop,
y_end_crop)]
        if len(refPoint) == 2:  # when two points were found
            croppedimg = image[refPoint[0][1]:refPoint[1][1],
refPoint[0][0]:refPoint[1][0]]
    previous_x_crop = x_crop
    previous_y_crop = y_crop

def ImgCropper(filename, imgSize):
    global image, cropping, initialCrop, x_start_crop, x_end_crop,
y_start_crop, y_end_crop, croppedimg
    cropping = False
    initialCrop = False
    x_start_crop, y_start_crop, x_end_crop, y_end_crop = 0, 0, 0, 0
    cv2.namedWindow("image")
```

```
    cv2.setWindowProperty("image", cv2.WND_PROP_TOPMOST, 1)
    cv2.moveWindow("image", 20,20)
    image = cv2.imread(filename)
    height, width, _ = image.shape
    print('Select the desired region of the image and press "C" to
confirm.')
    if max(width,height) < 850:
        if width > height:
            image = cv2.resize(image, (850,int(height*(850/width))),
interpolation=cv2.INTER_NEAREST)
        else:
            image = cv2.resize(image, (int(width*(850/height)), 850),
interpolation=cv2.INTER_NEAREST)
    else:
        if width > height:
            image = cv2.resize(image, (850,int(height*(850/width))))
        else:
            image = cv2.resize(image, (int(width*(850/height)), 850))
    cv2.setMouseCallback("image", mouse_crop)

    while True:
        imagegui = image.copy()
        if not cropping:
            cv2.imshow("image", image)
        elif cropping:
            cv2.rectangle(imagegui, (x_start_crop, y_start_crop),
(x_end_crop, y_end_crop), (255, 0, 0), 2)
            cv2.imshow("image", imagegui)
        if initialCrop:
            break
        if cv2.waitKey(1) == 27:
            exit()

    if x_end_crop < x_start_crop:
        x_start_crop = x_start_crop+x_end_crop
        x_end_crop = x_start_crop-x_end_crop
        x_start_crop = x_start_crop-x_end_crop
    if y_end_crop < y_start_crop:
        y_start_crop = y_start_crop+y_end_crop
        y_end_crop = y_start_crop-y_end_crop
```

```
        y_start_crop = y_start_crop-y_end_crop
    refPoint = [(x_start_crop, y_start_crop), (x_end_crop, y_end_crop)]
    if len(refPoint) == 2:  # when two points were found
        croppedimg = image[refPoint[0][1]:refPoint[1][1],
refPoint[0][0]:refPoint[1][0]]

    cv2.namedWindow("Cropped image")
    cv2.setWindowProperty("Cropped image", cv2.WND_PROP_TOPMOST, 1)
    cv2.moveWindow("Cropped image", 960,20)
    cv2.setMouseCallback("image", mouse_crop_adjust)
    while True:
        imagegui = image.copy()
        cv2.rectangle(imagegui, (x_start_crop, y_start_crop), (x_end_crop,
y_end_crop), (255, 0, 0), 2)
        cv2.circle(imagegui, (x_start_crop,y_start_crop), 5, (0,255,0),
-1)
        cv2.circle(imagegui, (x_end_crop,y_end_crop), 5, (0,255,0), -1)
        cv2.imshow("image", imagegui)
        try:
            cv2.imshow("Cropped image", croppedimg)
        except:
            pass
        key = cv2.waitKey(1)
        if key == 27:
            exit()
        if key == ord('c'):
            break
    cv2.destroyAllWindows()
    height, width, _ = croppedimg.shape
    if height >= imgSize:
        croppedimg = cv2.resize(croppedimg, (width-(width%imgSize),
height-(height%imgSize)), interpolation=cv2.INTER_NEAREST)
        cv2.imwrite("result.png", croppedimg)
    else:
        cv2.imwrite("result.png", np.zeros((imgSize, imgSize, 3),
dtype=np.uint8))

# Function definitions from ev3Functions.py
def moveXmotor(Xmotor, location, safeDistance, brakee=True,
keepDistance=False):
```

```
    Xmotor.start_move_to(location + safeDistance, speed=100, brake=brakee)
    while Xmotor.busy: pass
    if not keepDistance:
        Xmotor.start_move_to(location, speed=5, brake=brakee)
        while Xmotor.busy: pass

def moveYmotor(Ymotor, location, Ydistance, useYdistance=False,
brakee=True):
    if useYdistance:
        Ymotor.start_move_to(location + Ydistance, speed=100,
brake=brakee)
        while Ymotor.busy: pass
        Ymotor.start_move_to(location, speed=5, brake=brakee)
        while Ymotor.busy: pass
    else:
        Ymotor.start_move_to(location, speed=100, brake=brakee)
        while Ymotor.busy: pass

def pickPixel(Zmotor, Zbottom, Ztop, Zdistance, brakee=True):
    retry = True
    while retry:
        retry = False
        Zmotor.start_move_to(Zdistance, speed=75, brake=brakee)
        while Zmotor.busy: pass
        Zmotor.start_move_to(Zbottom, speed=25, brake=brakee)
        while Zmotor.busy:
            if cv2.waitKey(1) == ord('s'):
                retry = True
                Zmotor.start_move_to(Ztop, speed=100, brake=brakee)
                while cv2.waitKey(1) != ord('c'): pass
        sleep(0.5)
        Zmotor.start_move_to(Ztop, speed=100, brake=brakee)
        while Zmotor.busy: pass

def placePixel(Zmotor, Xmotor, Ymotor, Zbottom, Ztop, Zdistance,
Xposition, Yposition, safeDistance, brakee=True):
    if Yposition < 0: Ymotor.start_move_to(Yposition - safeDistance,
speed=20, brake=brakee)
    while Ymotor.busy: pass
    Zmotor.start_move_to(Zdistance, speed=50, brake=brakee)
```

```
    while Zmotor.busy: pass
    Xmotor.start_move_to(Xposition, speed=5, brake=brakee)
    if Yposition < 0: Ymotor.start_move_to(Yposition, speed=10,
brake=brakee)
    while Xmotor.busy: pass
    while Ymotor.busy: pass
    Zmotor.start_move_to(Zbottom, speed=25, brake=brakee)
    while Zmotor.busy: pass
    sleep(0.5)
    Zmotor.start_move_to(Ztop, speed=100, brake=brakee)
    while Zmotor.busy: pass

def resetZAxis(Zmotor):
    Zmotor.start_move(speed=10)
    sleep(3)
    Zmotor.position = 0
    Zmotor.start_move_to(position=-90, speed=25, brake=True)
    while Zmotor.busy: pass
    Zmotor.position = 0

def resetXAxis(Xmotor, XTouch, Xstart, Xdistance):
    if not XTouch.touched:
        Xmotor.start_move(speed=75)
        while not XTouch.touched: pass
        Xmotor.stop()
    Xmotor.start_move_by(-175, speed=25, brake=True)
    while Xmotor.busy: pass
    Xmotor.start_move(speed=5)
    while not XTouch.touched: pass
    Xmotor.stop(brake=True)
    sleep(0.25)
    Xmotor.position = 0
    moveXmotor(Xmotor, Xstart, Xdistance)
    sleep(0.25)
    Xmotor.position = 0

def resetYAxis(Ymotor, YTouch, Ystart, Ydistance):
    if not YTouch.touched:
        Ymotor.start_move(speed=100)
        while not YTouch.touched: pass
```

```
        Ymotor.stop(brake=True)
    Ymotor.start_move_by(-250, speed=100, brake=True)
    while Ymotor.busy: pass
    Ymotor.start_move(speed=10)
    while not YTouch.touched: pass
    Ymotor.stop(brake=True)
    sleep(0.25)
    Ymotor.position = 0
    moveYmotor(Ymotor, Ystart, -Ydistance, useYdistance=True)
    sleep(0.25)
    Ymotor.position = 0

# Main code starts here
# Create an image for the machine. Saved as "result.jpg"
if useAI:
    AIfilename = "AIresult.png"
    print(f'\nGenerating image from prompt: "{object}"\n')
    imageGenerator.AIGenerator(object, AIfilename)
    imageFunctions.confirm_image(imgSize, object, AIfilename)
else:
    print(f'\nGetting image from path: "{filename}"\n')
    imageCropper.ImgCropper(filename, imgSize)

# Read the input image
original_image = cv2.imread("result.png")

# Pixelate the image
if usepixelation:
    pixelated_image = imageFunctions.pixelate_image(original_image,
imgSize)
else:
    pixelated_image = cv2.resize(original_image, (32, 32),
interpolation=cv2.INTER_AREA)
cv2.imwrite("pixelart.png", cv2.resize(pixelated_image, (256, 256),
interpolation=cv2.INTER_NEAREST))

# Turn it into Lego colors
simplified_image, recipe = imageFunctions.simplify_image(pixelated_image,
colors_bgr)
```

```
cv2.imwrite("pixelart Lego.png", cv2.resize(simplified_image, (256, 256),
interpolation=cv2.INTER_NEAREST))

# Printing the required bricks
print("Image created.\n\nRequired bricks:")
print("----------------------------")
for i in recipe: print(f"{i:<18}| {recipe[i]:>3} pcs.")
print("----------------------------")

# Display the original and pixelated images
cv2.imshow("Original Image", cv2.resize(original_image, (512, 512),
interpolation=cv2.INTER_NEAREST))
cv2.setWindowProperty("Original Image", cv2.WND_PROP_TOPMOST, 1)
cv2.moveWindow("Original Image", 136,200)
cv2.imshow("Pixelated Image", cv2.resize(pixelated_image, (512, 512),
interpolation=cv2.INTER_NEAREST))
cv2.setWindowProperty("Pixelated Image", cv2.WND_PROP_TOPMOST, 1)
cv2.moveWindow("Pixelated Image", 698,200)
if useOverlay: cv2.imshow("Final Image",
blend_modes.multiply(np.dstack((cv2.resize(simplified_image, (512, 512),
interpolation=cv2.INTER_NEAREST), np.ones((512, 512, 1), dtype=float))),
legoOverlay, 1).astype(np.uint8))
else: cv2.imshow("Final Image", cv2.resize(simplified_image, (512, 512),
interpolation=cv2.INTER_NEAREST))
cv2.setWindowProperty("Final Image", cv2.WND_PROP_TOPMOST, 1)
cv2.moveWindow("Final Image", 1260,200)

# Asking for confirmation
print('Press "C" to Confirm.\n')
while True:
    if cv2.waitKey(1) == ord('c'): break
cv2.destroyAllWindows()

# Creating empty white image
current_image = np.full((int(imgSize), int(imgSize), 3), 255,
dtype='uint8')

# Display the targeted and current image
if useOverlay: cv2.imshow("Current Image",
blend_modes.multiply(np.dstack((cv2.resize(current_image, (512, 512),
```

```
interpolation=cv2.INTER_NEAREST), np.ones((512, 512, 1), dtype=float))),
legoOverlay, 1).astype(np.uint8))
else: cv2.imshow("Current Image", cv2.resize(current_image, (512, 512),
interpolation=cv2.INTER_NEAREST))
cv2.setWindowProperty("Current Image", cv2.WND_PROP_TOPMOST, 1)
cv2.moveWindow("Current Image", 417,200)
if useOverlay: cv2.imshow("Targeted Image",
blend_modes.multiply(np.dstack((cv2.resize(simplified_image, (512, 512),
interpolation=cv2.INTER_NEAREST), np.ones((512, 512, 1), dtype=float))),
legoOverlay, 1).astype(np.uint8))
else: cv2.imshow("Targeted Image", cv2.resize(simplified_image, (512,
512), interpolation=cv2.INTER_NEAREST))
cv2.setWindowProperty("Targeted Image", cv2.WND_PROP_TOPMOST, 1)
cv2.moveWindow("Targeted Image", 979,200)

# Starting main loop
print('Starting...\nPress "A" to Abort code early.\n')
coords = list((int(0),int(0)))
while True:

    # Get current pixel color
    current_color = list(simplified_image[imgSize-coords[1]-1, coords[0]])

    # Get current color index
    current_color_index =
list(colors_bgr.values()).index((current_color[0], current_color[1],
current_color[2]))

    # Check if current pixel color isn't white
    if current_color != list((255,255,255)):

        # Checking whether the desired color is present
        current_color_name =
list(colors_bgr.keys())[list(colors_bgr.values()).index((current_color[0],
current_color[1], current_color[2]))]
        if colors_stock[current_color_name] > 0:
            colors_stock[current_color_name] -= 1

        # Ask for refill if needed
        else:
```

```
            print("Please refill all colors, because", current_color_name,
'is almost empty.\nPress "F" to finish fill.')
            while True:
                if cv2.waitKey(1) == ord('f'): break
            for x in colors_stock:
                colors_stock[x] = color_amount
            print("Resetting Z-Axis...")
            print("Resetting X-Axis...")

        # Update current image
        current_image[imgSize-coords[1]-1, coords[0]] = current_color
        if useOverlay: cv2.imshow("Current Image",
blend_modes.multiply(np.dstack((cv2.resize(current_image, (512, 512),
interpolation=cv2.INTER_NEAREST), np.ones((512, 512, 1), dtype=float))),
legoOverlay, 1).astype(np.uint8))
        else: cv2.imshow("Current Image", cv2.resize(current_image, (512,
512), interpolation=cv2.INTER_NEAREST))

    # Going to the next pixel
    if coords[0] < 31: coords[0] += 1
    else: coords[0] = 0; coords[1] += 1

    # Checking whether the code should be ended
    if cv2.waitKey(1) == ord('a'): break
    if coords[1] == 32: break
cv2.destroyAllWindows()
```