Project report

Paulo Khayat

The goal of the project is to take an image of a voxel world (a 3d grid of cubes) and to try to recreate the positions of the cubes.

In this report I will describe the components of the project I've researched and worked on and explain why I think some features perform worse than expected and suggest alternative solutions I could do in the future.

Image generation

I decided to use a graphics engine to generate images of the cubes for the project. This gives me control over the features of the image and will allow me to render things differently depending on the demands I require.

I chose to use a python wrapper of OpenGL core called ModernGL since I was planning on doing the image recognition stage in python using the same programming language for both would let me incorporate the two parts easily.
Also, I wanted to explore how other OpenGL wrappers differ from the one I learnt in the computer graphics course.

Cube detection – first approach

The general requirements I adhered to are that the textures and dimensions of the cubes are known in advance, however the camera position and orientation aren't.

My first approach consisted of detecting the edges of the cubes on screen in order to detect their faces.
The steps to this detection will include:

1. Line detection
2. Turning edges to a planar graph to find the faces of the cubes
3. Calculating the distance and angle of these faces

1. Line detection

In images of a voxel world, the edges of the cubes tend to create high contrast lines, whether because of how the shading affects the different faces of the cube, or because of a transition to the background or another texture. An edge detector such as a canny filter can be used to detect these edges, which can then be used to find those cubes' positions.

There are several different methods for detecting line segments in an image. I will present the three which I've found and used, and show the behaviors I've observed for them:

Hough Transform / Progressive Probabilistic Hough Transform (PPHT)

This is the only method I knew of before starting this project. The initial step is to run a canny filter on the inputted image and be left with a binary image of high contrast pixels.

Every lit pixel in the binary image votes in Hough space for lines that could pass through it.
The following steps are run recursively:

- Find a line with many votes.
- Among the pixels that voted for this line find a long continuous segment, that follow the trend of the line.
- These pixels are removed from the canny image and their votes are all removed.
- If the segment was longer than a certain threshold, it is added to the result list.

This process is repeated until the original canny image is empty of pixels.
The image library I use has an existing implementation for this detector, based on the paper [Progressive probabilistic hough transform; J. Matas, C. Galambos and J. Kittler].

The library offers the constants for rho and theta sizes, used for determining the granularity of the Hough voting regions, minimum length and maximum gaps in the resulting segments, and a voting threshold.

Line Segment Detector (LSD)

Another line detector implemented in OpenCV is Line Segment Detector, based on the paper [LSD: a Line Segment Detector R. Grompone von Gioi Et. Al.].
This algorithm works very differently from Hough transform. For each pixel, the gradient of the light level is calculated for the grayscale of the input image, and then, nearby pixels which share the same gradient direction are grouped.
A bounding box on the group is then made, rotated in the direction of the gradient they share, which is then turned into a segment after many checks that verify the quality of this segment.

The advantage this method has is that it works without having to tune any hyperparameters.

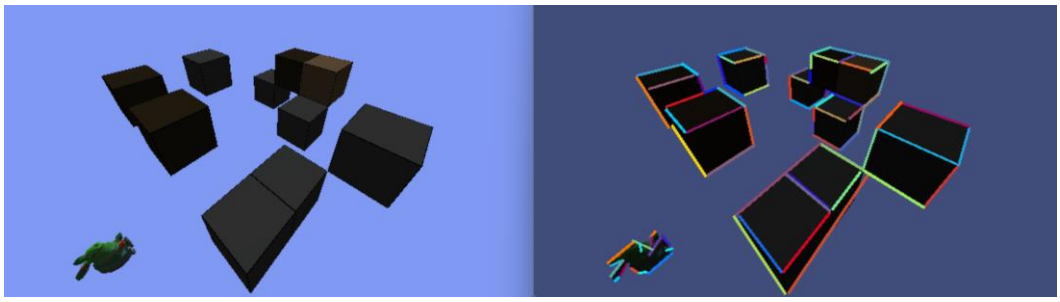
EDLines

While researching how these algorithms I discovered a third algorithm work called EDlines, from the paper [EDLines: A real-time line segment detector with a false detection control], which runs in linear time with the amount of pixels in the image.

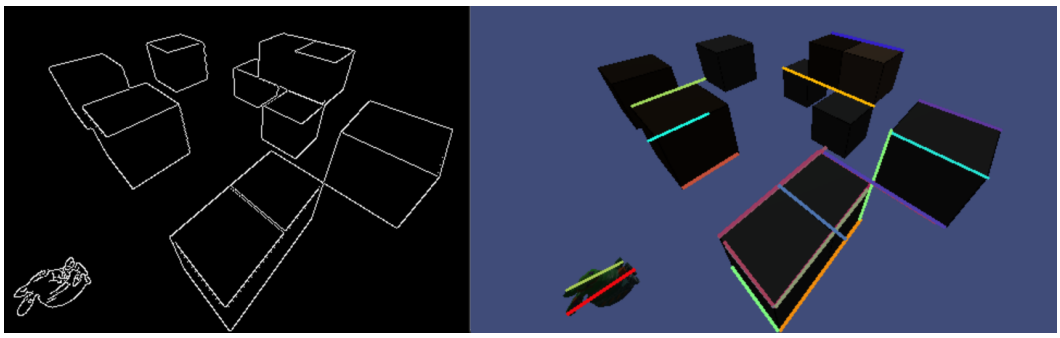
The paper shows that this algorithm can be 10 times faster than LSD and Hough for some images, which intrigued me. However, optimization wasn't my focus at this stage, and it was simpler to use a line detector that is already implemented in OpenCV, especially for debugging the other portions of the project.

Comparing results:

LSD:



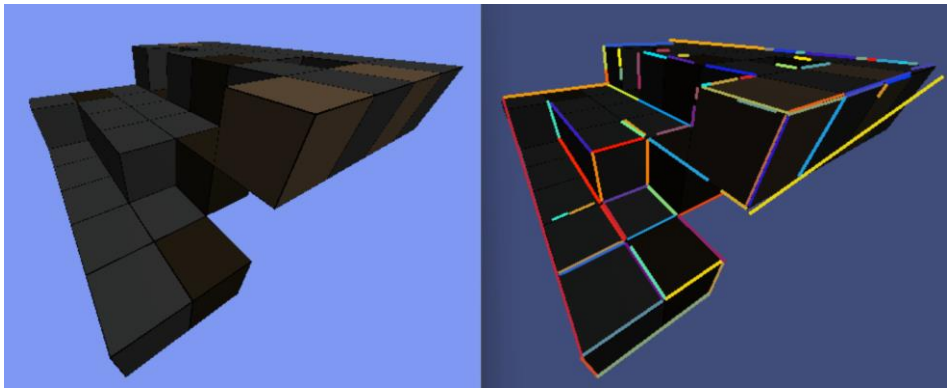
LineHoughProb:



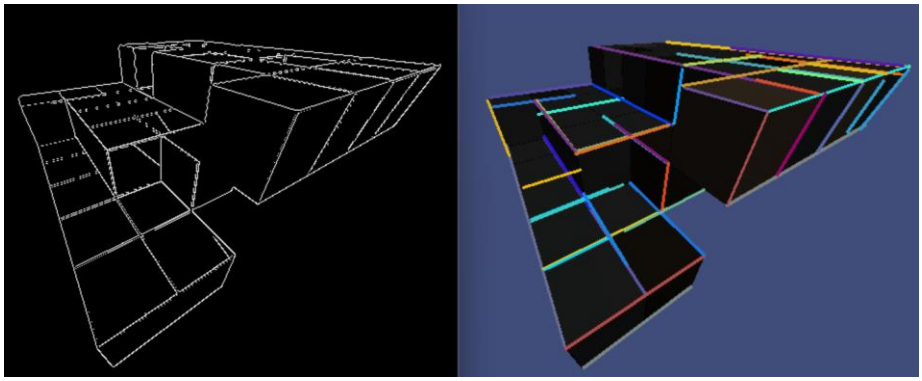
In this example, the LSD algorithm finds most of the relevant edges on the scene and does so accurately.
The Hough transform method consistently fails at detecting edges of faraway cubes. This is because there are no perfect constants for the algorithm that suit every image.
The far away cubes don't get detected because their edges are shorter than the minimum threshold constant. The algorithm also wrongly connects two cubes which aren't touching, because the gap between them is too small.
If I were to change these thresholds, false positives such as diagonal lines would be detected more often, especially for closer cubes.
Also, it is worth noting that some of the missing edges are attributed to the canny filter missing them, which could be solved with better tuning.

<u>7x7 image:</u>

LSD:

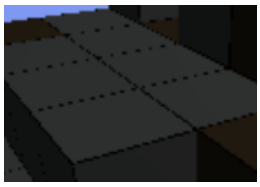


LineHoughP:



In this example the Hough transforms performed better, although some lines are still angled wrong.

The missing lines in the LSD may be a fault of the image generation. Rather than creating a model for the outline of the blocks and rendering them using the GL_LINES feature, I opted for simply making a flat texture with thin outlines, which causes these outlines to not fully render at further distances. This, paired with the fact that I didn't implement anti-aliasing, produced lines that Hough is more compatible with than LSD (seen below).

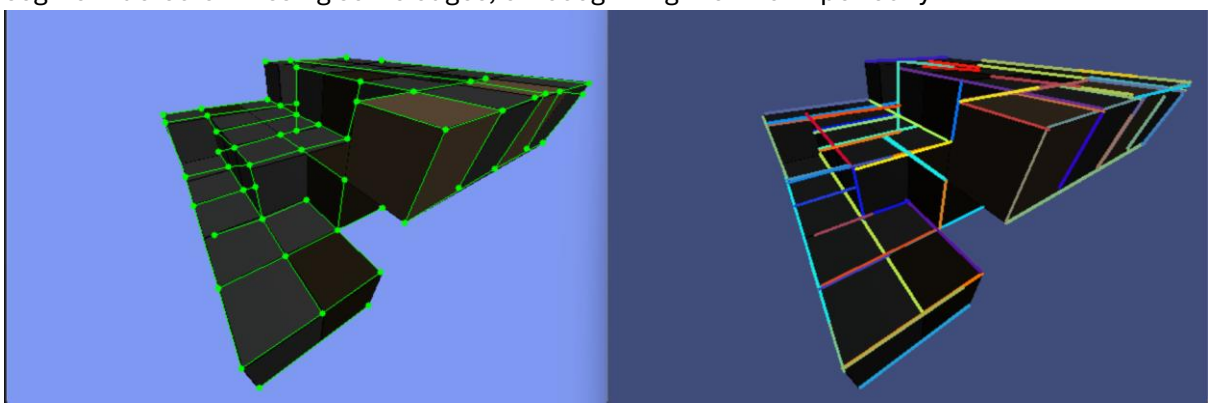


In the end I chose to use LSD, as the results seemed more consistent.

2. Turning segments into a planar graph

In order to find faces from this list of line segments, I combined all the line segments to one planar graph that I could do a search for cycles on.

We need to take into account line segments that are parallel and could be merged, merging vertices that overlap, finding intersections between segments, and accomodating for the line segment detector missing some edges, or recognizing them to imperfectly.



in the left image we can see the result, a planar graph. In the right image we can see the input, many segments found by the PPHT edge detector, with each color being a different edge.

From the list of lines I found in the last part, I create a set of vertices and edges for each line, and combine them, namely by:

- **Merging overlapping vertices:** connect two segments if their vertices overlap. Has an option to limit the neighbors of a vertex, to allow only dead end lines to be combined.
- **Splitting intersecting lines:** finds the intersection between all pairs of segments in the graph, and replaces those segments with new ones with a vertex in between them.
- **Combining Parallel lines:** if two segments are parallel and close enough, create a new parallel segment in between them that combines their lengths.

I've added some features that seemed like they would produce better results, even if they aren't necessary for creating a planar graph. Combining parallel lines eliminates behavior I've seen where some edges are recognized twice and makes each cube edge one segment. There are also other features I added for similar reasons, like allowing segments to extend if it means they can intersect with another segment.

There are problematic errors that happen at this stage which may give us a graph who's edges are further from the "ground truth" segments of the input.
For example, a vertex belonging to a cube close to the camera is merged with one of a cube far away.

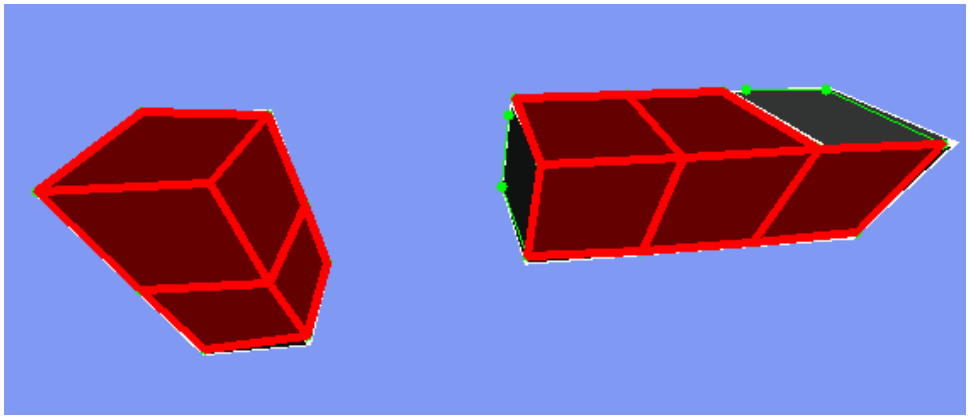
Another problem is that the process of merging vertices or edges involves some guesswork as to where to put the new vertices. There is no right answer in this case, I chose to have the new vertex be the average of the previous two, weighted towards the vertex with more neighbors, as this gave me better results in some tests.

I will expand on this more at the conclusion of this approach.

3. Calculating the distances to the cubes

Now that we have converted the image to a graph of cube edges, we can now attempt to find the distance and angle of these cubes from the camera.

First, we go over the graph and find cycles that resemble faces. This is done by simply searching from each vertex for a closed cycle of four edges, with some optimizations like preventing redundancies and counting two parallel edges as one.



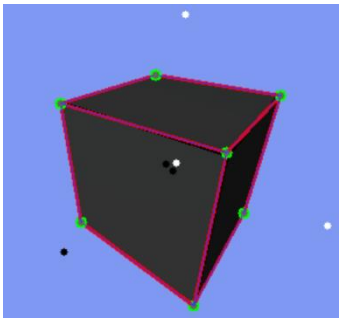
Above we can see the faces that were found, marked in dark red.

The "real world" dimensions of all these faces can be defined as 1x1 meter squares. Intuitively, we can imagine how the distance calculation might be if the square is facing us directly (and we

know the focal length of the camera). This calculation becomes much more complicated however when we introduce rotation for the face.

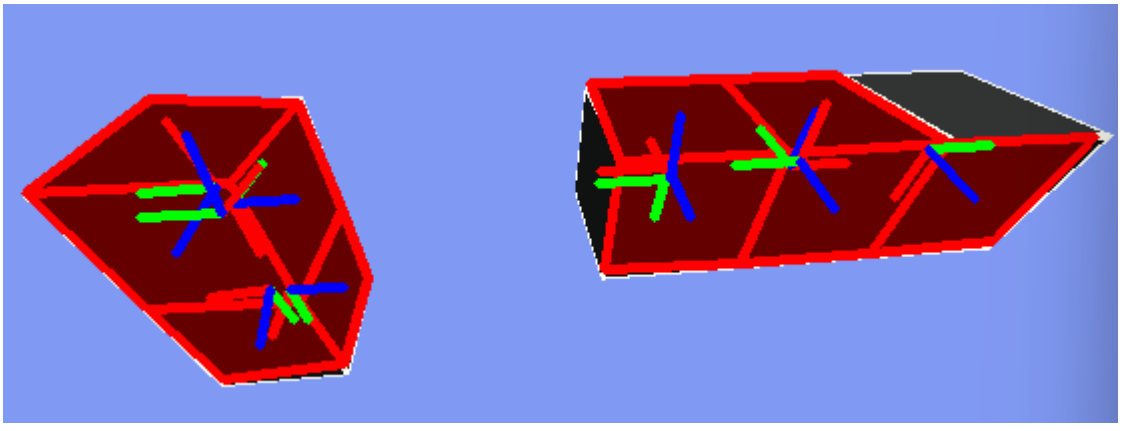
This problem is more generally known as the **Perspective N-point problem** (PNP), in it, a 3D constellation of points is inputted, along with their coordinates in an image, and you are tasked with finding the orientation and offset of the camera from the constellation.

I decided to try this method to get the locations of the faces relative to the camera.
Using the solvePnP function of OpenCV, I found the location of the center of the cube relative to the camera, and visualized it by reprojecting it in the image:

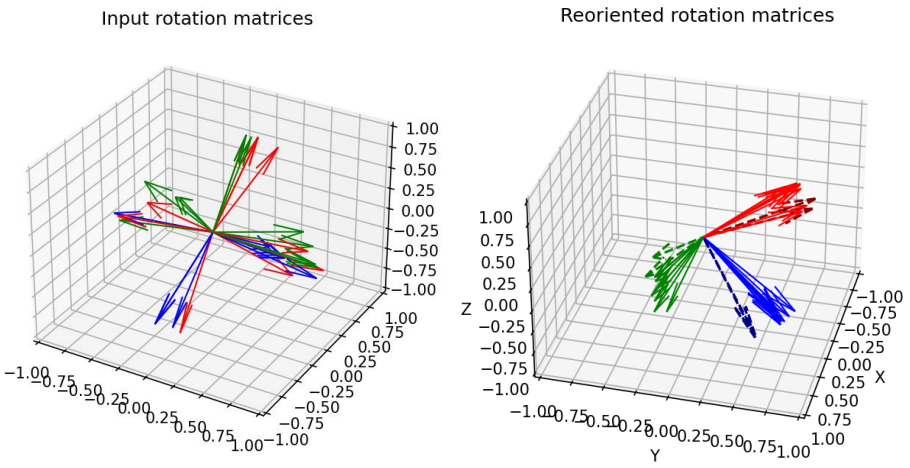


Note there are many possible solutions to this problem, and specifically for these square faces, there are 8 different camera position and rotations which produce the identical image for a square (another way to think of it is the vertices of the squares have 8 different orderings). We might assume that the vertices of the front faces are ordered clockwise, and the solvePnP will find a solution assuming they are counterclockwise in the image.
Because of this, I find the center of both possible cubes the face is touching and choose the one further from the camera.

Since we get the offset and orientation of each of these faces from the camera (or the camera from the faces) we can visualize the coordinate system of each of them after applying these transformations, which we see below:

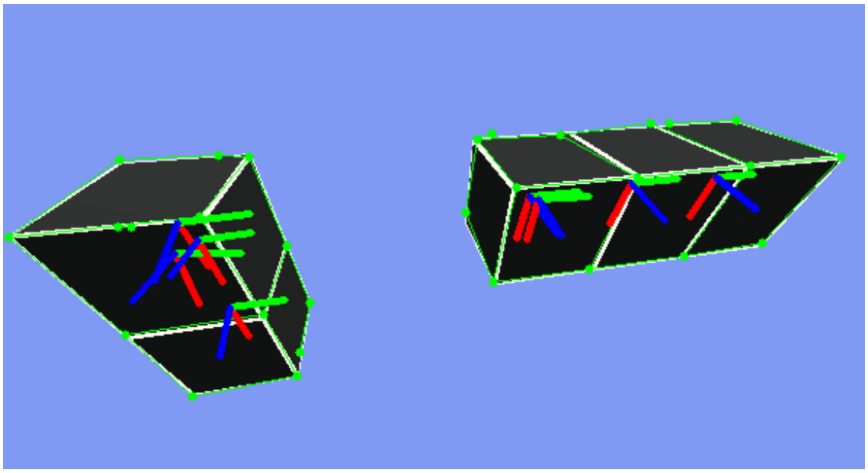


We can see that all of these coordinate systems seem to have parallel axes but the order of the axes is different. This makes sense, since as we said that there is more than one solution to this PNP problem.



rotation matrices of the faces before and after realigning them.

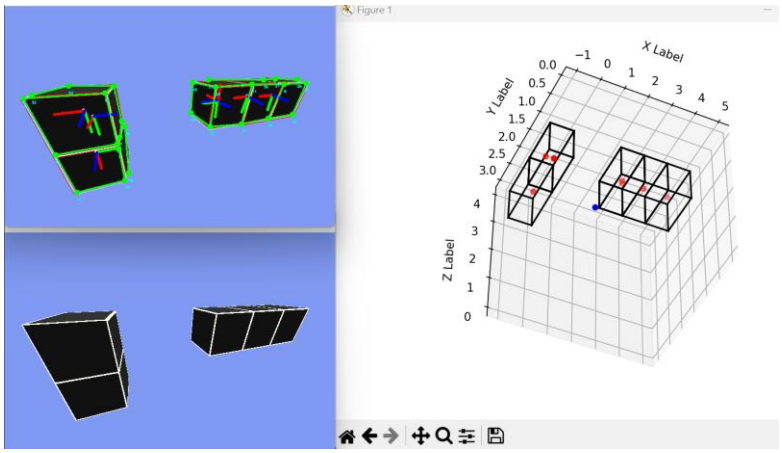
We take all the different transformations and invert or swap the axes around, so they are all facing the same direction. This also helps find outliers.



the cube transformations after corrections and removing outliers.

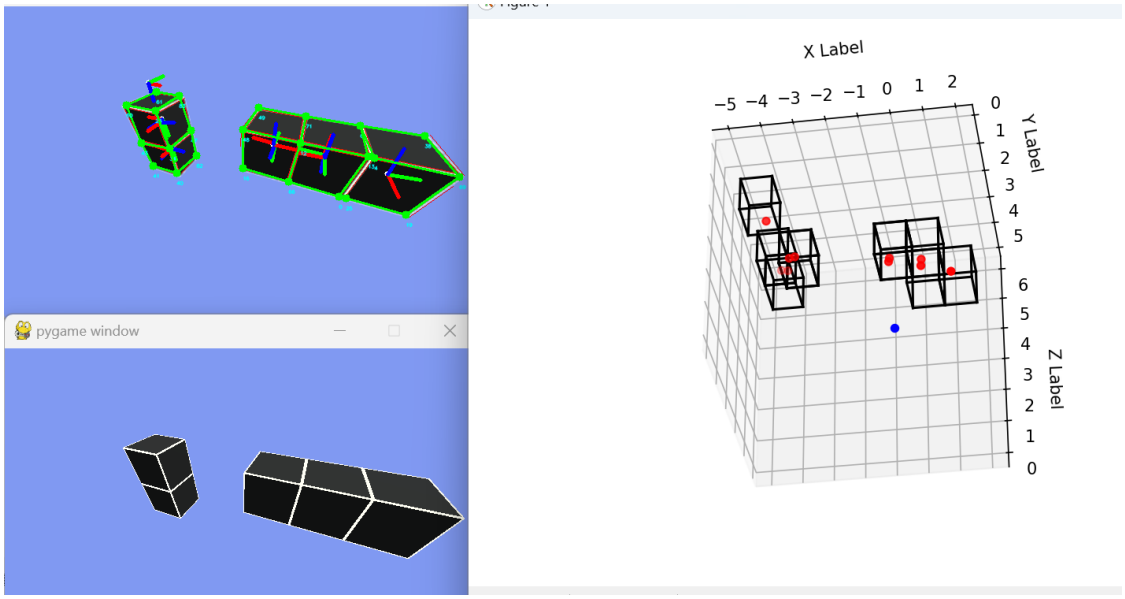
4. Final steps

Finally, we take the cubes and align them by averaging their rotation matrices in relation to the camera and applying the inverse to them.
Below are some results of simple environments:



In this example we can see a perfect reading of the image. The reconstruction on the right shows two vertically stacked cubes, and two spaces to their right, three cubes in a line.

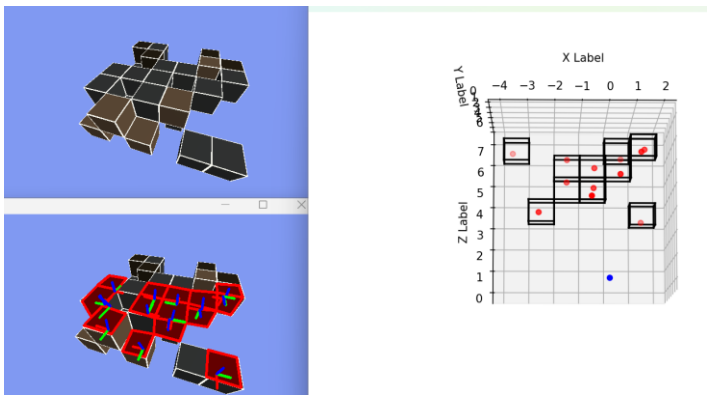
I draw all cubes in the grid that a detected cube center resides in, but before that, I add to all detected cube positions a certain (dx, dy, dz) to minimize the distances of the detected positions to the center of cubes on a grid $(x + 0.5, y + 0.5, z + 0.5)$, This is to overcome the fact that these detected positions are relative to the camera, and if the camera is not grid aligned, All these detected positions might lie on the bound between two cubes.



Most of the time we get imperfect solutions like in the image above. Malformed faces in the graph create cubes with very inaccurate positions (as seen in the top left cube in the top left image), and more importantly the **alignment to the grid isn't accurate**.

The cube positions are relative to the camera, so in order to find the voxels on the grid, I need to realign them to the grid. Thanks to the solvePNP solutions, I know the angles of the faces, so I am able to create a rotation matrix and apply the inverse to all the points.
The estimation that I make here is decent, but in this example a slight error in one of the angles causes many of the points to land in the wrong cubes.

I've attempted other tests that address this by either rotating the points using the ground truth of the camera extracted from the graphics engine when generating the image, or allowing the cubes to not be locked to the grid in the predictions, however the results could still be improved.



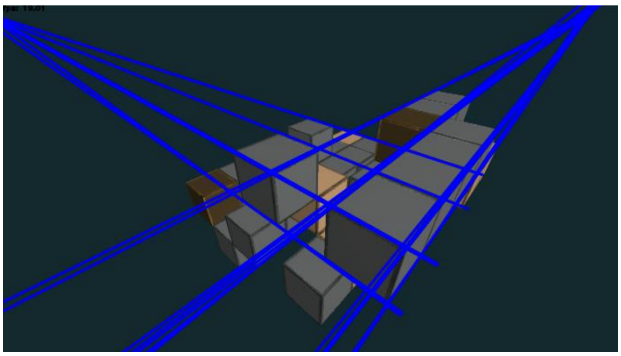
Analysis

There are several issues that cause faults in this process:

- Detecting a cube is highly dependent on finding all four edges of a face: If a face is partially obstructed, these faces should still give us information, but these solutions can't glean anything from them.
- The PNP solver is sensitive to errors: say one edge of the face is wrongly detected as shorter than what is seen on the image, the solver might assume we are seeing the square at a steeper angle.
- The graph creating process shifts around the edges: The intersection and merging of vertices creates a graph that is slightly morphed compared to the original lines inputted, which causes inaccuracies down the road.
- The PNP Solver is too robust: The PNP solver finds the orientation of each face individually, but in reality, all the faces share the same 3 orientations. If we were to input into the solver all the faces as one constellation, we would get a better result, but that requires knowing where each face is, which is what we're trying to solve for.

While the results give us good coverage of some cubes under good conditions, a result more tailored for the problem could show better results.

Cube detection – second approach



An interesting trait seen in the Hough lines of the generated image is that all the lines corresponding to the edges of cubes seem to intersect at 3 points.

This is the **vanishing point** effect, where all coplanar parallel lines in 3D intersect at a point when projected to 2D (in perspective).

General Idea

If we can identify the vanishing points of the image, we can do finer grain line segment detections on the image and filter out the lines that don't pass through a vanishing point.
We can also use the vanishing points to classify the lines that do pass through vanishing points.

Additionally, we can figure out the angle of the camera relative to the grid.

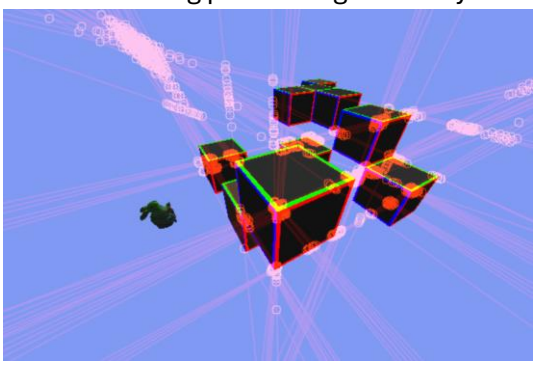
Note: This was my initial idea for how to solve the project, however some practical implementation issues that I will address later led me to focus on the first approach, which I felt could reach tangible results sooner.

Finding the vanishing points

Simple approach: finding intersections

The first approach I attempted is to simply find the intersections of all the Hough lines. This solution proved unfeasible for three reasons:

1. The areas with the most intersections aren't necessarily the areas where the vanishing points are. There are a lot of intersections at the center of the image.
2. The sampling of all lines isn't uniform. Lines of edges in the axis the camera is facing tend to be detected the most, sometimes there are only 2-3 lines touching one of the vanishing points.
3. The closer the vanishing point is to infinity, the further apart the intersections are. All small errors become more impactful at long distances, so bounds of "nearby intersections" would have to accommodate this somehow. There is also the edge case of the vanishing point being at infinity that this solution simply can't solve.



Second approach: fitting the parameters in Hough space

For this approach we will model the vanishing points as functions of the camera parameters, and then match these parameters to fit the lines of the image as good as possible.

First, we can model the vanishing points so that we have fewer parameters to work with for them.

Recall that the perspective projection is a 4x4 matrix applied to homogeneous coordinates:

$$M = \begin{bmatrix} - & R & - & t_x \\ - & U & - & t_y \\ - & F & - & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where R, U, F are the right, up and forward vectors of the camera in the world.

The vanishing point of a line l: (t, y, z); $t \in (-\infty, \infty)$ is the result of projecting the point where t is infinity (or negative infinity depending on where the camera is facing).
The effect of the constants y, z will disappear, so all lines of the pattern (t, y, z) will intersect at the same vanishing point.

Now in order to find the vanishing points as a function of the camera orientation, we will represent R, U, F as a function of $\phi \in [0,\ \pi], \theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

Using spherical coordinates, we define $F = (\sin\theta \sin\phi\,, \cos\phi\,, \cos\theta \sin\phi)$

R will be $R = F \times (0,1,0) = \begin{vmatrix} i & j & k \\ \sin\theta \sin\phi & \cos\phi & \cos\theta \sin\phi \\ 0 & 1 & 0 \end{vmatrix} = (-\cos\theta \sin\phi\,, 0, \sin\theta \sin\phi)$

And after normalizing $\frac{R}{|R|} = \frac{R}{\sqrt{\cos^2\theta \sin^2\phi + \sin^2\theta \sin^2\phi}} = \frac{(-\cos\theta \sin\phi, 0, \sin\theta \sin\phi)}{\sqrt{\sin^2\phi}} = (-\cos\theta\,, 0, \sin\theta)$

$U = R \times F$

$= \begin{vmatrix} i & j & k \\ -\cos\theta & 0 & \sin\theta \\ \sin\theta \sin\phi & \cos\phi & \cos\theta \sin\phi \end{vmatrix} = (-\sin\theta \cos\phi\,, \sin^2\theta \sin\phi + \cos^2\theta \sin\phi\,, -\cos\theta \cos\phi) =$
$(-\sin\theta \cos\phi\,, \sin\phi\,, -\cos\theta \cos\phi)$

In total we get: $M = \begin{bmatrix} - & R & - & t_x \\ - & U & - & t_y \\ - & F & - & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\cos\theta & 0 & \sin\theta & t_x \\ -\sin\theta \cos\phi & \sin\phi & -\cos\theta \cos\phi & t_y \\ \sin\theta \sin\phi & \cos\phi & \cos\theta \sin\phi & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

We will now find the vanishing points by projecting points approaching infinity:

$$p_x = \lim_{x\to\infty} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} - & R & - & t_x \\ - & U & - & t_y \\ - & F & - & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} a(R_x x + R_y y + R_z z + t_x) \\ b(U_x x + U_y y + U_z z + t_y) \\ c(F_x x + F_y y + F_z z + t_z) + d \\ -(F_x x + F_y y + F_z z + t_z) \end{bmatrix}$$

after dividing by W: $p_x = \lim_{x\to\infty} \left(a \frac{R_x x + R_y y + R_z z + t_x}{F_x x + F_y y + F_z z + t_z}, b \frac{U_x x + U_y y + U_z z + t_y}{F_x x + F_y y + F_z z + t_z} \right)$, and by eliminating all constant elements that don't scale with x:

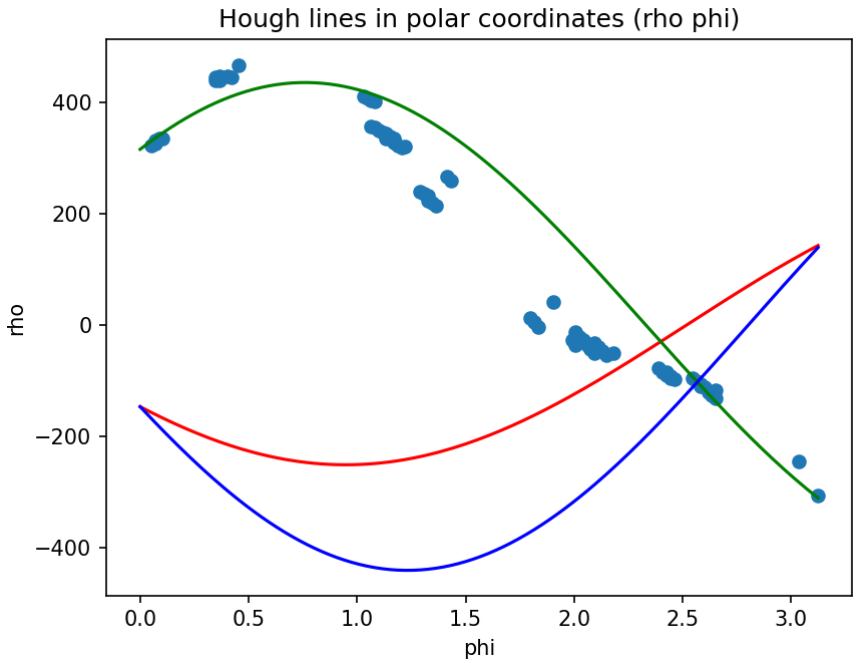
$$px = \left(\frac{R_x}{F_x}, \frac{U_x}{F_x}\right) = \left(a \cdot \frac{\cot\theta}{\sin\phi}, b \cdot \cot\phi\right)$$

$$p_y = \left(\frac{R_y}{F_y}, \frac{U_y}{F_y}\right) = (0, -b \cdot \tan\phi)$$

$$p_z = \left(\frac{R_z}{F_z}, \frac{U_z}{F_z}\right) = \left(-a \cdot \frac{\tan\theta}{\sin\phi}, b \cdot \cot\phi\right)$$

I added additional constants which represent the aspect ratio and FOV constants of the camera. In my attempts however, I assume the FOV is 90 degrees for simplicity.

Now that the screenspace locations of the vanishing points we can fit them to the lines in the image.



the lines detected in an image, and the vanishing point curves for a random angle.
(Note the curve parameters do not match the points)

In the image, we plot all the lines detected in an image using the Hough transform, and draw a curve for every focal point, depicting all the lines that pass through it.

Finding the camera intrinsics from this could involve performing least squares on the curves and minimizing the distance to any of the curves. Since this has non-linear operations however, it would require calculating the partial derivatives for ϕ and θ manually.
Currently, I've implemented simply randomizing camera angles over many iterations and storing the best result.

Next steps

Once the right camera parameters are found, we could classify the edges in the image depending on which focal point they intersect. Given the camera matrix, we can calculate the angle and distance to a line without having to rely on solvePNP for squares.

Conclusion / future directions

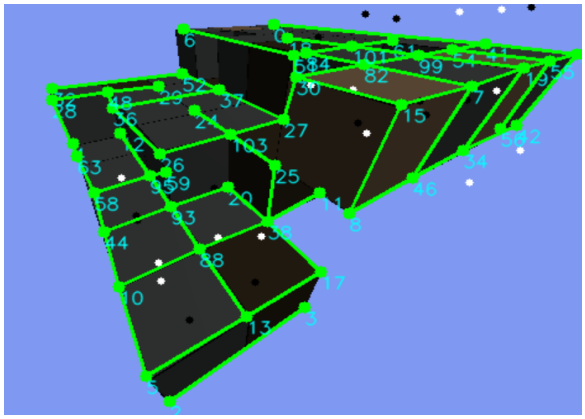
There are many approaches to detecting cubes that I could explore more in the future.
The next steps for the second approach would require finding a proper way to fit the camera parameters, and the math for the distance of faces (or segments) relative to their shape given the projection matrix.

There are several things I could change in order to learn from the project more. Currently, for all the line detectors, I used the existing implementations in OpenCV. I set up the draw calls to allow me to do some of them using modernGL, but I haven't implemented it. Implementing them would be interesting and would teach me more about the intricacies in the paper I haven't gone into in this report, however it wouldn't improve recognizing the cubes at all.

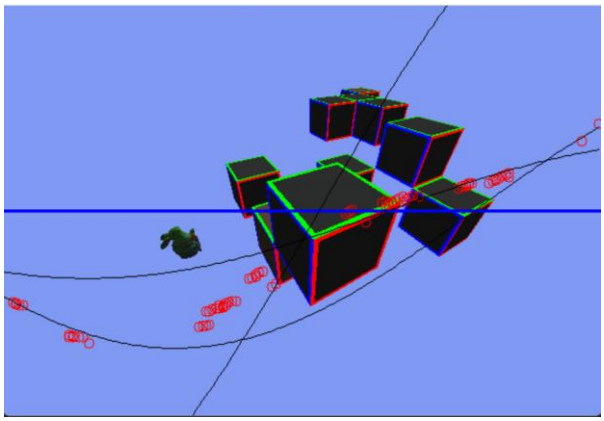
I could also try improving the first approach by tuning the constants, reworking the graph creator to stay faithful to the images lines, or combine ideas from the second approach, like aligning the graph edges to the focal points after the fact.

There are also other possible approaches I could explore, although they might be more complicated than I am anticipating. One such idea is using the color of the faces more. A possible idea would be to use the filled in areas of the screen as a binary mask of where cubes exist and use this to rule out where cubes can be in the result. Alternatively, bring back the cubes that have textures, and work with detecting textures known in advance, rather than lines.

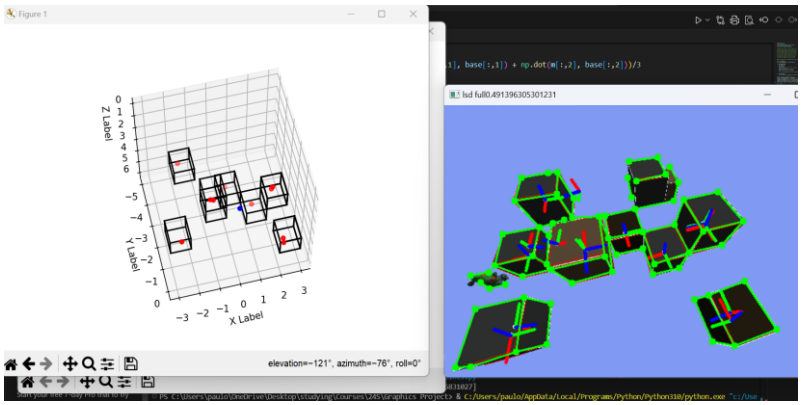
Appendix images:



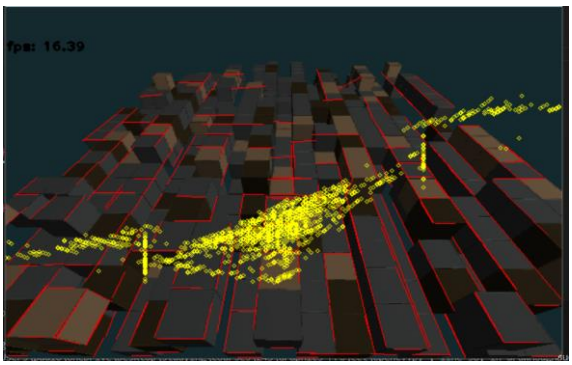
cube center predictions (black & white)



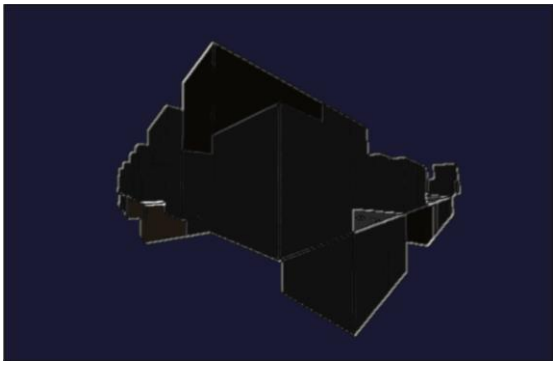
fitted parameters attempt, 200 guesses



successful cube recognition example



Visualizing the lines for a busy image

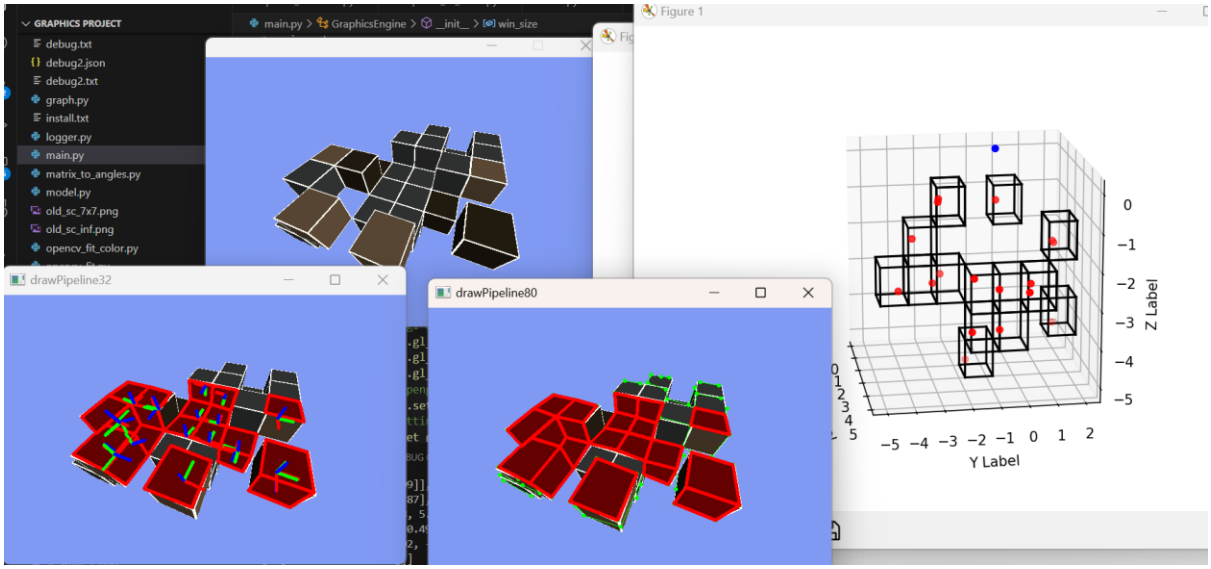


Custom canny shader in modernGL

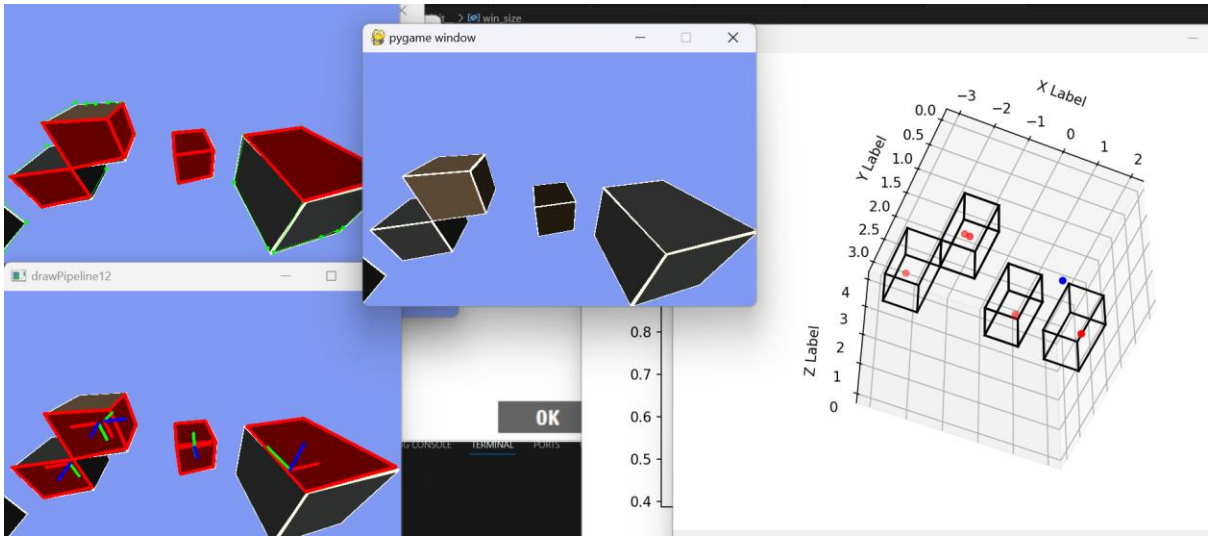
Continuation:

Following the work I showed in the mid report, I decided to rethink the approach I had to the problem, considering the results seemed too unreliable, and I tried different methods to fixing these issues, which one neat solution, I've described in the next segment.

Upon working on that solution, I found a very silly error in the camera intrinsics matrix (dividing by the aspect ratio instead of multiplying by it), after fixing it, the resulting cubes showed of my original approach showed astounding results:



It simply works as envisioned. The faces that are detected do get malformed in the process of converting the lines to a graph, but I overestimated how much it would affect the solvePnP solution. In reality, it was the cause of a bad camera intrinsics matrix, which gave good but not great results.



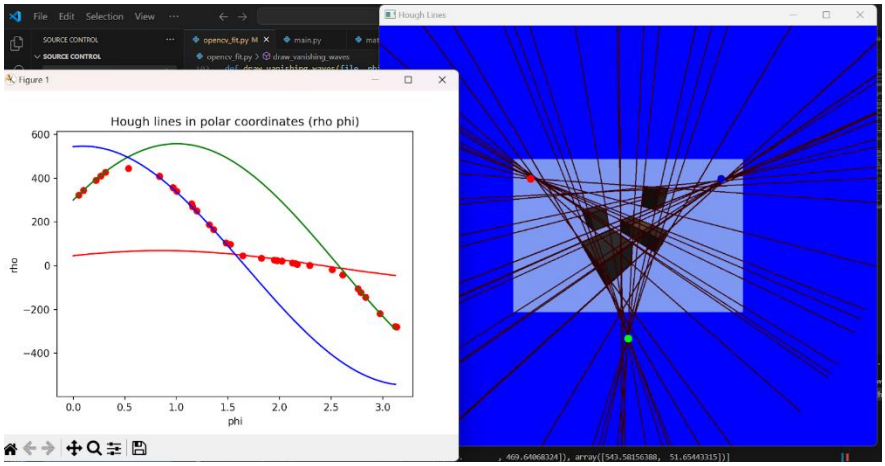
In these images we can see the source image from the simulation I run, aswell as the faces colored in, and the guess of the algorithm as a matplotlib graph. In many examples I run, the results are perfect for the cubes it does recognize.

There still is an expected decrease in accuracy when a face is partially covered by another face, or with far cubes where the thresholds I placed start to create issues, but this is to be expected of this approach.

Now I'll present the culmination of the vanishing point guessing approach, which also gave me good results:

Vanishing point solution

In the mid report I presented the idea of modelling the vanishing points based on the yaw and pitch of the camera, and using the HoughLines to find the correct angles, since all lines parallel to the axes should rest on these curves. I've managed to solve the errors I got stuck with, and we can see a perfect graph, which the code consistently finds:

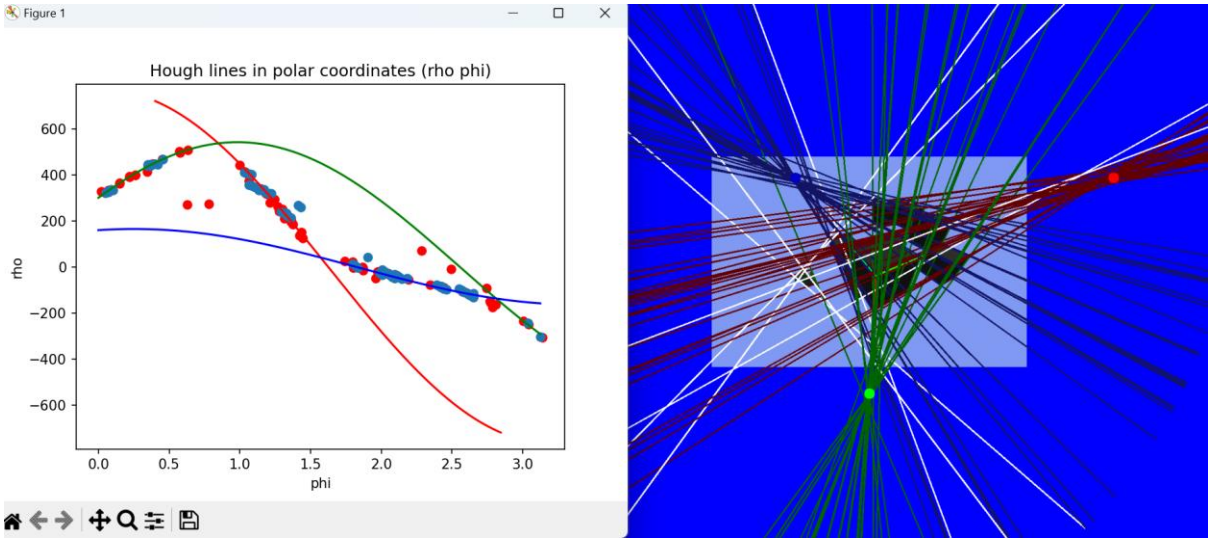


On the left we can see points representing the rho and phi for each of the lines from the edges in the original image (rho and phi is the polar representation of these lines, phi is the slope, rho is the distance from the origin (0,0)).
On the right, we can see these lines visualized as well as the three vanishing points for the three of the axes.

A simple pure random search is executed. The yaw and pitch of the camera are randomized, pitch between $[0, \pi]$ and yaw in $[0, \frac{\pi}{2}]$, since there is a symmetry in the four directions, we can search the yaw in a 90 degree range.

The development from the mid report was sadly no more than sign errors in this visualization, which caused me to assume the answer lied in other solutions.

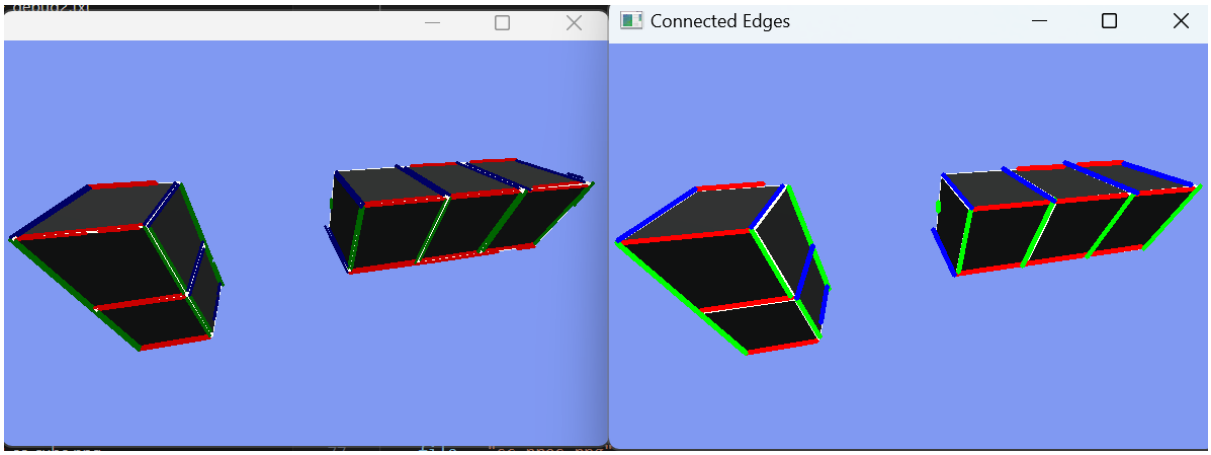
We can now classify these lines by which vanishing point they are closest to:



Notes: The white lines are discarded by being too far from any focal point.
The red and cyan dots in the left graph correspond to LSD and Hough, respectively.

This gives us the bonus benefit of detecting erroneous lines, and thus we can run our edge detectors with lower thresholds which can catch these patterns for cube images which are less obvious, such as without outlines.

What matters to us however is the edges, not the lines. For visualization purposes, we showed the edges as lines extending to infinity, but we now apply the classification to the edges that produced those lines:



The left image is the classification, the right is after merging parallel lines into one.

Finding the cubes – revisited

We now have a lot more data about the cubes than we did previously: We know the angle of the camera in relation to the voxel grid, many edges of cubes and their orientation and length assuming it's a single cube edge, it would be redundant to run a PnP solver, since we already know a lot about the perspective of the faces in relation to the camera.

Mathematical method:

Consider an edge $(x_1, y_1), (x_2, y_2)$ classified as a y-axis facing edge. By intersecting it with edges on other axes we can get most likely a segment that is one meter long.

Given yaw (θ) and pitch (ϕ) that we know, the camera matrix is:

$$M = \begin{bmatrix} - & R & - & t_x \\ - & U & - & t_y \\ - & F & - & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\cos\theta & 0 & \sin\theta & t_x \\ -\sin\theta\cos\phi & \sin\phi & -\cos\theta\cos\phi & t_y \\ \sin\theta\sin\phi & \cos\phi & \cos\theta\sin\phi & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And for the sake of simplicity, the projection matrix is a simple 90-degree FOV, meaning we divide by Z and multiply one axis by the aspect ratio.

Given these parameters, we need to find a real-world edge: $(a, b, c) \rightarrow (a, b+1, c)$ where $a, b, c \in \mathrm{N}$, so that projecting this line results in the $(x_1, y_1), (x_2, y_2)$ from the image.

Note: I will assume $t_x, t_y, t_z \in [0,1)$, to avoid redundancy. The cube is captured in relation to the camera, so if the cameras offset contains some whole number, we can subtract that position from the cube.
Also, these three numbers are shared by all cubes and edges.

$$M \cdot \begin{bmatrix} a \\ b \\ c \\ 1 \end{bmatrix} \rightarrow (after\ dividing\ by\ Z) =$$

$$\left[\frac{-\mathrm{a}\cos\theta + c\sin\theta + t_x}{a\ \sin\theta\sin\phi + b\cos\phi + c\cos\theta\sin\phi + t_z}, \frac{-a\sin\theta\cos\phi + b\sin\phi - c\cos\theta\cos\phi + t_y}{a\ \sin\theta\sin\phi + b\cos\phi + c\cos\theta\sin\phi + t_z}\right] =$$

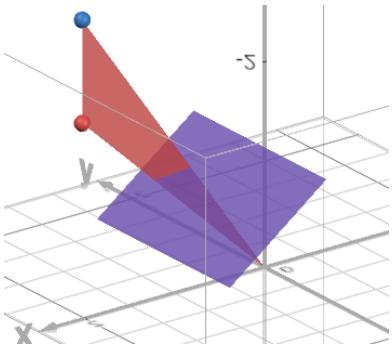
$$M \cdot \begin{bmatrix} a \\ b+1 \\ c \\ 1 \end{bmatrix} \rightarrow (after\ dividing\ by\ Z) =$$

$$\left[\frac{-\mathrm{a}\cos\theta + c\sin\theta + t_x}{a\ \sin\theta\sin\phi + b\cos\phi + c\cos\theta\sin\phi + t_z + \cos\phi}, \frac{-a\sin\theta\cos\phi + b\sin\phi - c\cos\theta\cos\phi + t_y + \sin\phi}{a\ \sin\theta\sin\phi + b\cos\phi + c\cos\theta\sin\phi + t_z + \cos\phi}\right]$$

This approach doesn't simplify nicely like with the vanishing points, and this equation isn't nice enough to work with. The math above has too many variables, which might factor out eventually, but it would be a very arduous task. It does seem, however, that we have enough information to cover this number of degrees of freedom. We now know the camera angle, the angle to a given edge, it's orientation in the world and its length, which the next approach will utilize.

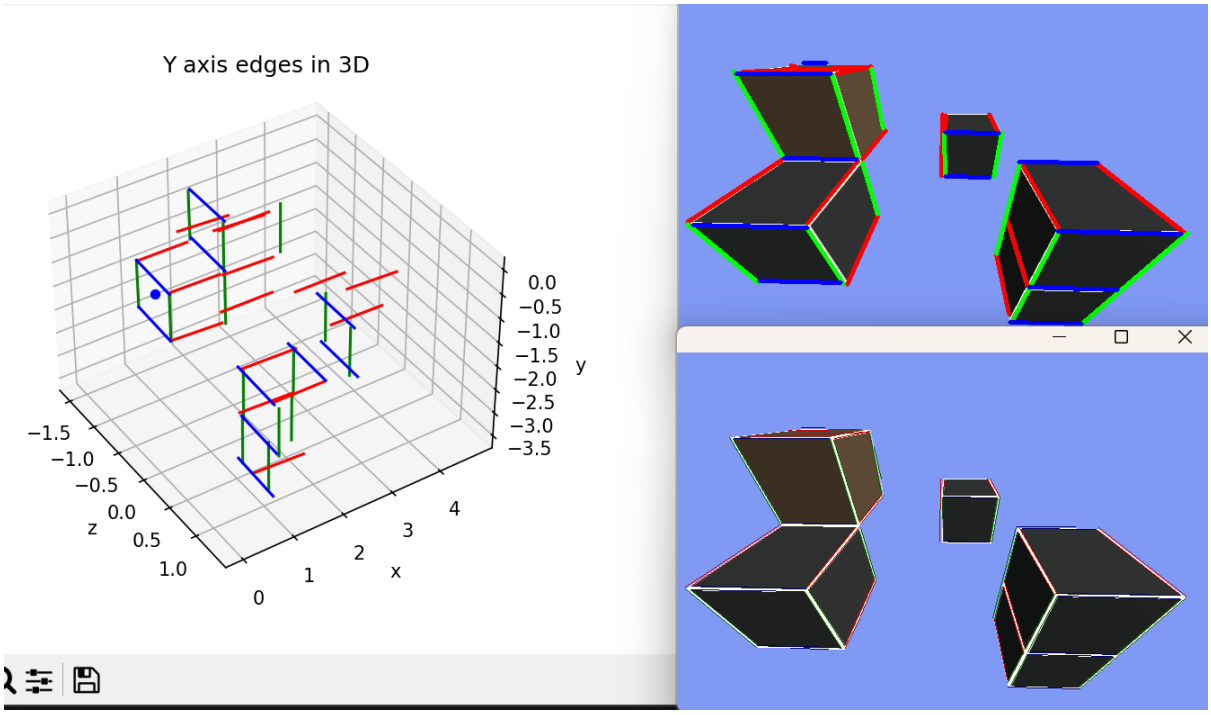
Triangulation method:

The next idea is to calculate the distance to an edge by forming a triangle with it's two ends and the camera, orienting the points in the world using the camera angles, and calculating the distance to the points (the sides of the triangle), by calculating the angles of this triangle.



In the image above we can see the real-world edge and the image plane, the pixels that define this edge are formed by the intersection of this triangle and the image plane. By creating vectors to the pixels on the image plane, we can get vectors of the two triangle sides which touch the camera. The third side, is in this case, the vector (0,1,0). We use the 3D angle between vectors formula: $\theta = \cos^{-1}\frac{u \cdot v}{|v||u|}$, and the triangle sine formula: $\frac{a}{\sin\alpha} = \frac{b}{sin\beta} = \frac{c}{sin\gamma}$ to get the two lengths, which we can now place in the world.

After finding the edges in the world, we can snap them to the nearest/ several nearest options on the grid and cast votes on the cubes they touch. Cubes with several votes will be picked.



I've implemented this and can see that the edges resemble fairly closely the edges of the original image, the key part was aligning the image plane to the world correctly before doing all the calculations.

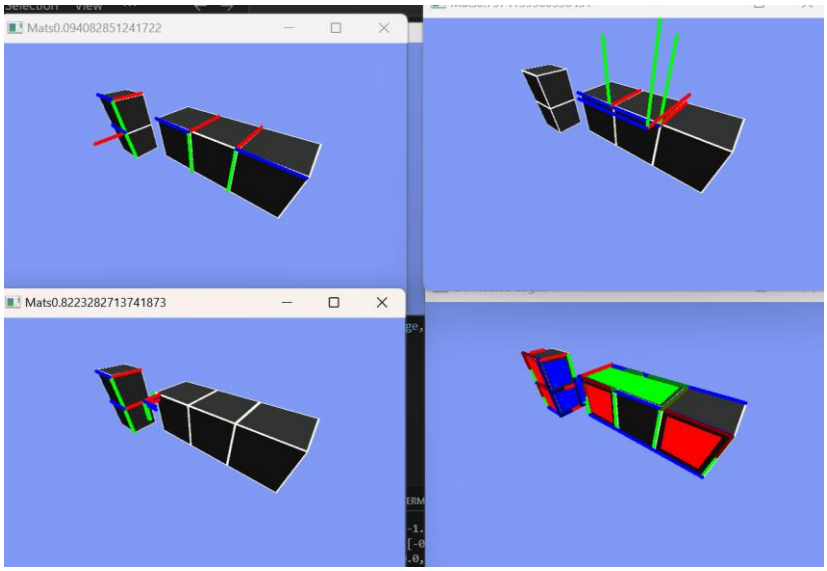
For some reason, although the calculations should be the same, there is incorrect behavior for the X and Z edges in some cases, due to some mistake I made in the projection matrix, so while the edges on the Y axis are always right, for many images the other edges are very wrong. Since I've managed to prove the concept of this portion, I decided to move on to solutions that gave results sooner.

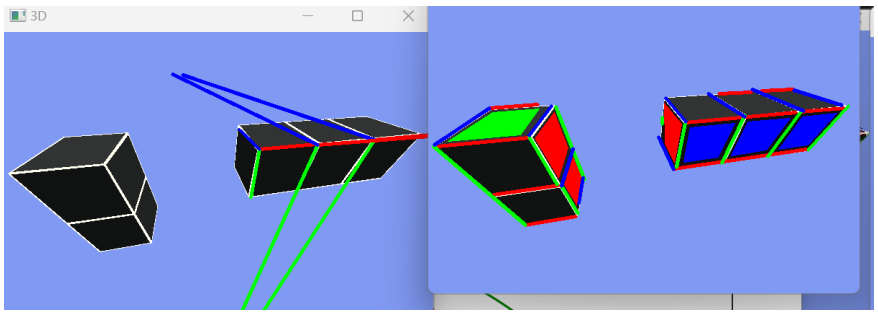
SolvePnP method:

The second approach is to use the solvePNP method which I used in my previous approach. I could use these classified edges to create faces and localize them as I've done before, but this algorithm guesses the orientation of the face In relation to the camera, which we already know, using the camera angles.

I've thought of a way to use the generic solvePNP algorithm, and force it to use our known camera angle:

Normally, we pass the algorithm matching pairs of *(object points, image_points)*. We can trick the algorithm into knowing the angle of the object by passing $[N, 0,0]; [0, \pm N, 0]; [0,0, N], N \to \infty$ as object points, and the vanishing points v_x, v_y, v_z as the corresponding image points.





This solution requires a nuance which wasn't a requirement before, we need to pass the pixels corners of each face in a specific order. Now that the vanishing points force a certain orientation, if we send in the face in a counter clockwise order rather than clockwise, we would form a shape that could never match the given pixels and the SolvePNP algorithm will throw extreme results. I've solved this by just trying all the combinations and picking one which has results that seem to fit a reasonable distance.

A better solution would have been to re-project the face and see if it matches the pixels we provided.

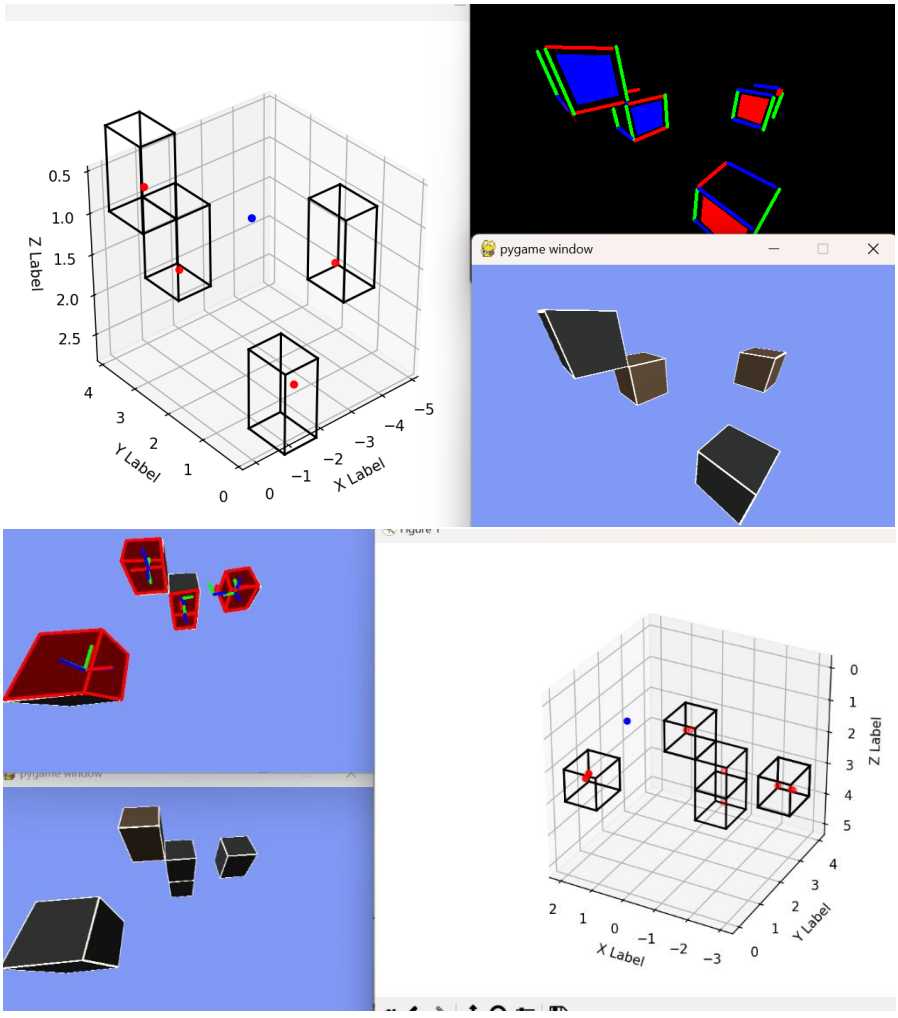
Vanishing point solution vs pure face to SolvePNP solution

I do believe this vanishing point solution could produce better results than the results I've shown in the "Continuation" section, because of the extra information baked in to the solution, the fact that the detected edges are better filtered to exclude noise, and I don't convert the image to a planar graph, which I've implemented to be lossy.

Taking the edges from this solution and using them for the faces solution also produced good results, however in the images I am taking, there is not much noise, and the LSD edge detector is pretty good at not detecting false positives, so the results are still similar.

The results of the pure solution could still be improved by improving the planar graph code, although I deemed that a task that wouldn't be as interesting as exploring other solutions.

The edges in 3D solution performed really well, and if you rotate it so you're looking exactly where the camera is (the blue dot) and changing the field of view, you can see they form the image, even when errors occur and the result is noisy, which is neat.



Examples from combining the vanishing point edge detection, combining edges of different axes to create faces, and the old cube detection via faces.

Conclusion

In this project I've set out to find methods to localize cubes via a 2D image of them, with limited knowledge on the camera position and orientation.

I've successfully managed to implement several approaches which were more tailor made to the problem than existing image to 3D detectors publicly available in OpenCV.

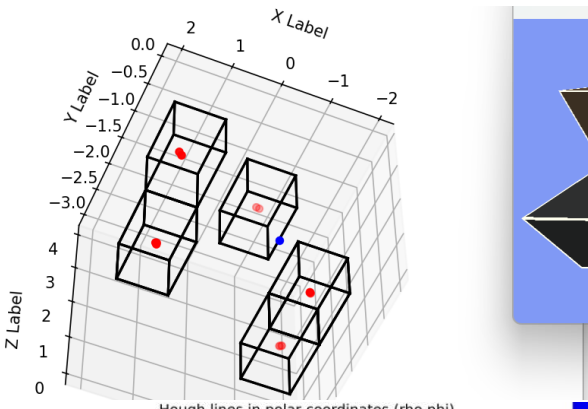
For these approaches, I've had to recreate transformation and projection matrices, create and maintain planar graphs, orient objects in 3D and reorient them to fit new reference frames, work with Hough transforms and got to research several different edge detectors such as PPHT and LSD, as well as researching vanishing points and formulating their positions on the screen.

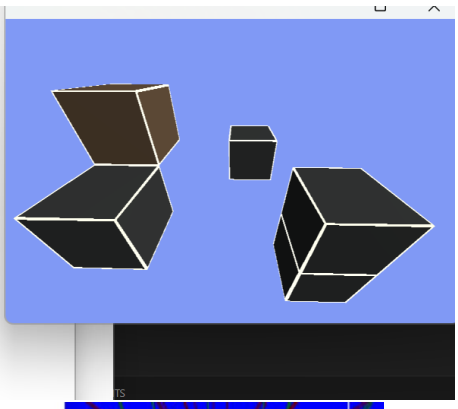
On the graphics engine part of this task, I've been tasked with implementing a sufficient engine in python, and worked with antialiasing, shader code, creating render passes and more.

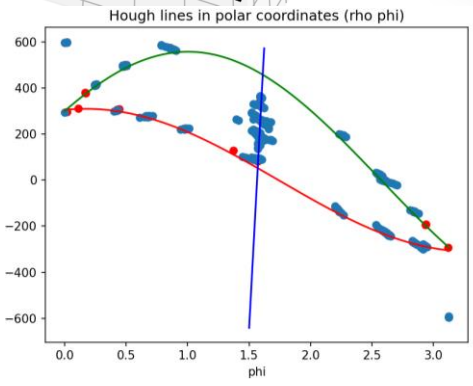
I'm pleased with the results of the camera angle detection, which worked better than I had anticipated, as well as the original solvePNP method of finding the cubes faces, which produced neat results as seen below. I do regret that I could not get some of the non solvePnP solutions to work, as they rely on zero external functions, and can work with cube images even if a face is partially obstructed, something the current detector doesn't handle great.

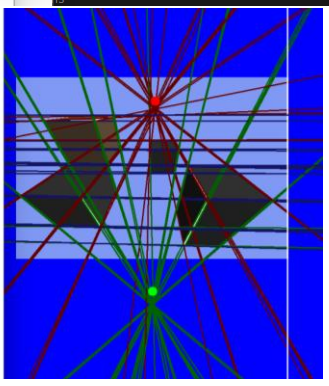
The methods I focused on also required that each cube would be rendered with an outline, something that is unlikely to see in art and games these cubes might appear in. I assume with more consistent results or more testing, it could be possible to try different face shapes, such as 1x2 meters, and keep the result that matches the grid the best.

Overall, it has been a very interesting and fun project which has taught me a lot about many topics in computer graphics, and to the question "wouldn't this be easier with neural networks?" I can say very solemnly 'yes'.



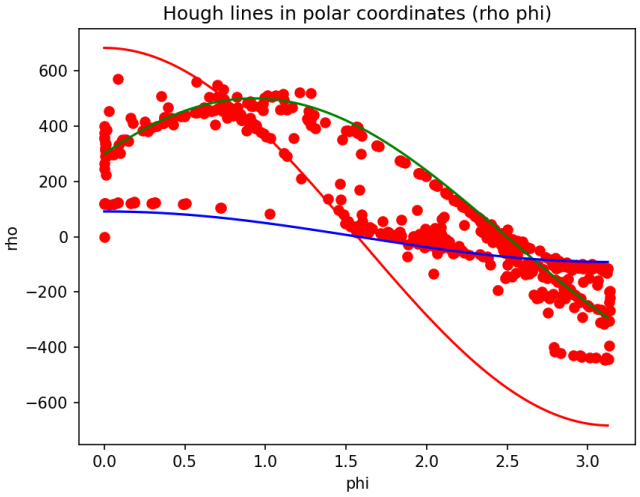






Fun experiments

Lastly, I tried the camera angle getting method on a screenshot from the game minecraft:



Note: the FOV is set to 90 to match the assumption I made when doing the math.

The increase in the number of edges makes the simple learning algorithm I used perform less well, but we can still see the vanishing points are close to what they should be, and only edges are detected that match the grid.

The face detection understandably took very long and produced a lot of noise, this is because the way I wrote my project, each pixel on the texture could be seen as little faces. For that reason my solution would have to go through some optimizations and improvements in order to handle images more packed in detail, where noise is more prevalent.