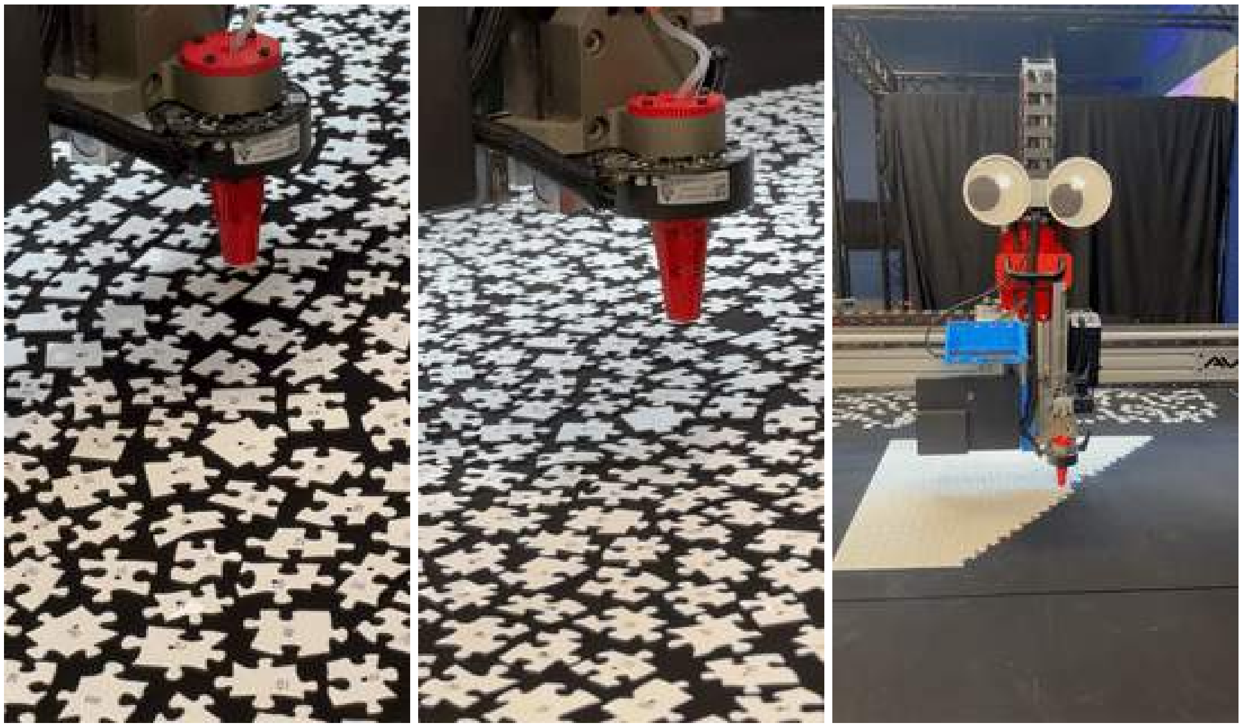


Jigsaw Puzzle Robot, Or:
How We Solved a Puzzle in 15 Months

Mark Rober – Nerd who makes YouTube videos you might have heard of
Ryan Oksenhorn – Co-founder @ Zipline who loves robots more than sleep
Ian Charnas – Chief Engineer at Mark Rober's YouTube channel and karaoke champ

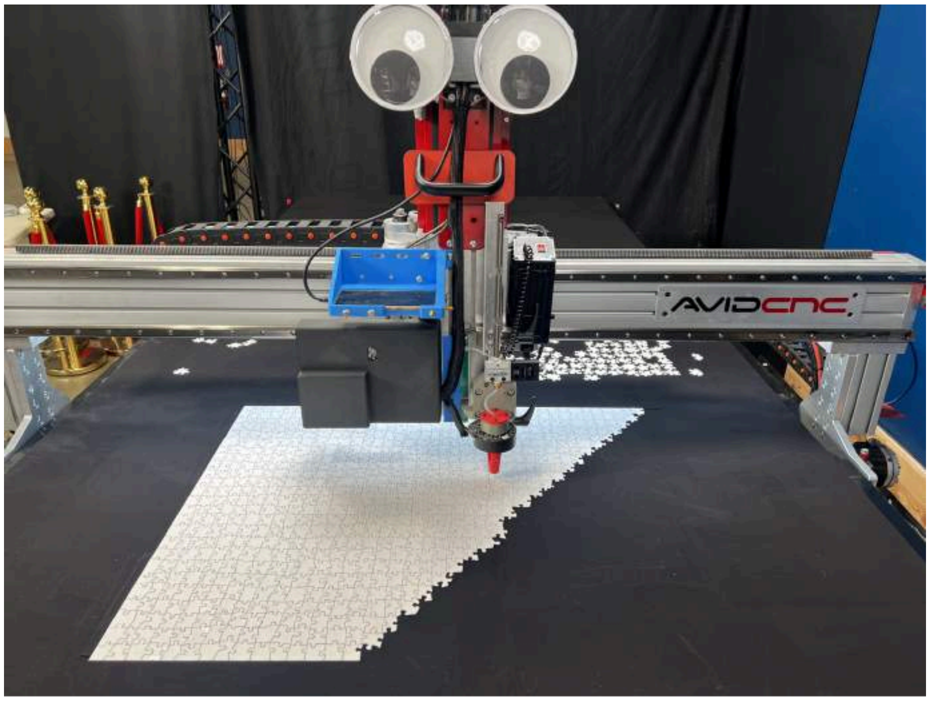
After seeing others valiantly attempt (and struggle) to build a puzzle-solving robot, we couldn't stop asking ourselves: *this can't actually be that hard, right?*

Bored on a long flight, Ryan decided to start writing a puzzle solving algorithm. Fifteen months (and many long nights) later, we have a robot that reliably and autonomously solves pretty hard jigsaw puzzles. We even crushed the Guinness record holder in a head-to-head race. Turns out, it *was* really really hard.

If you haven't seen the video of our results, check that out here.

If you're curious about all the technical details and access to the source code, read on.

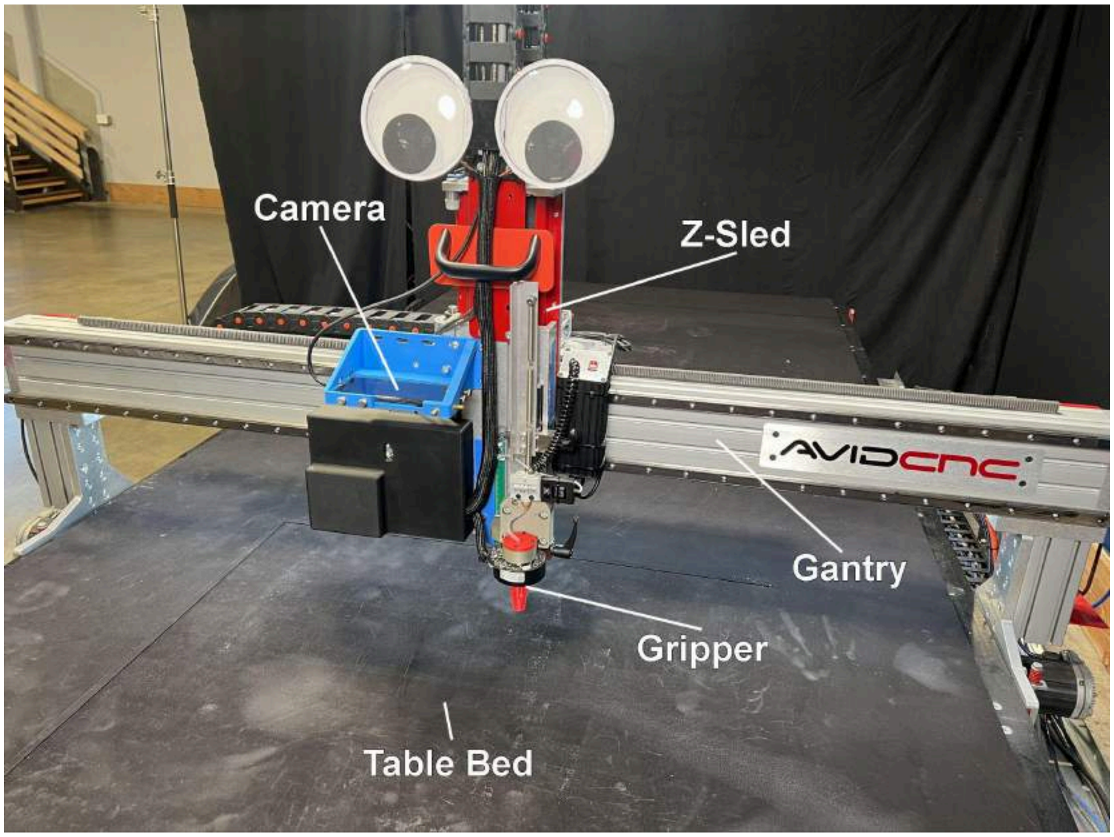
Robot Overview

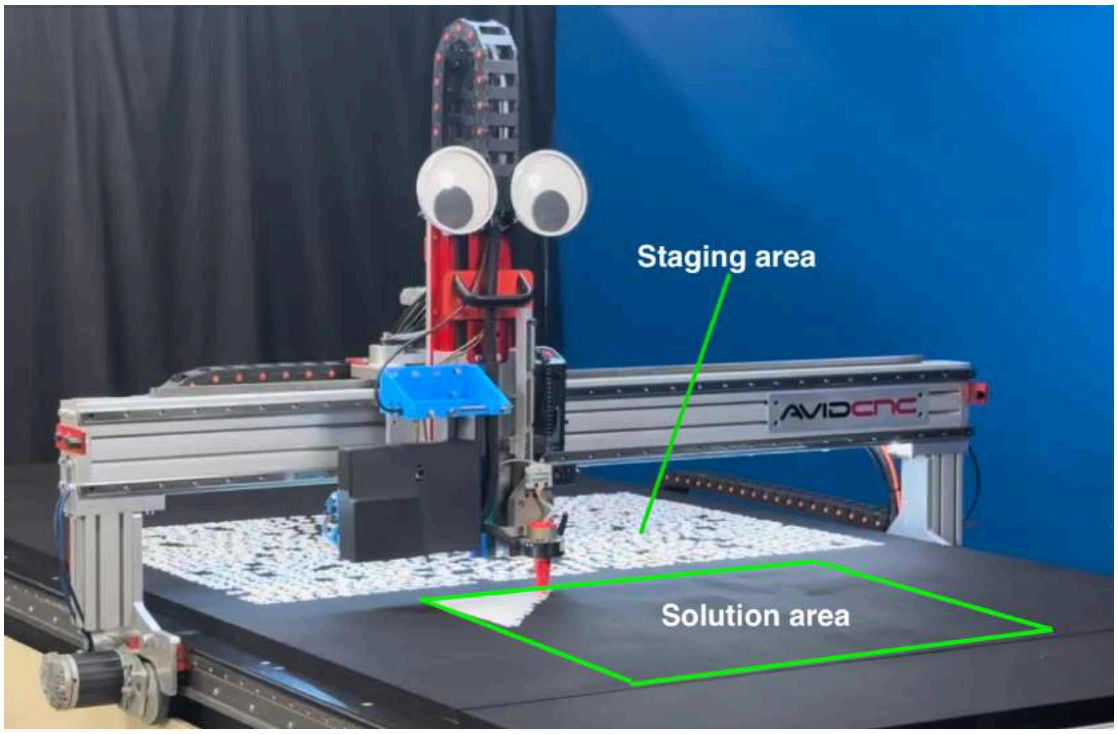


Jigsaw is a single-purpose robot that reliably and autonomously solves and assembles “normal” jigsaw puzzles. Jigsaw’s biggest achievement so far is **finishing a 1000-piece all-white puzzle with no human intervention**. The robot does this in three phases:

1. Taking photographs of all of the pieces
2. Computing a solution to the puzzle
3. Moving all of the pieces into place to form the assembled puzzle

The robot itself is constructed of a repurposed CNC router from AvidCNC, together with a purpose-built vacuum gripper and a Google Pixel smartphone used for the camera, which allows the robot to see the shape and position of the puzzle pieces.


Camera
Z-Sled
Gantry
Gripper
Table Bed

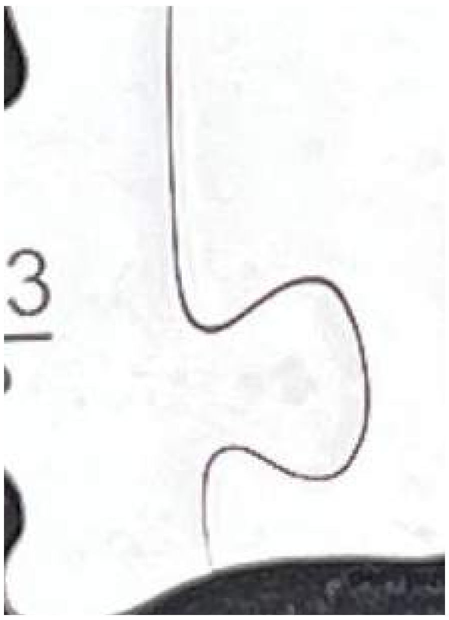

Staging area
Solution area

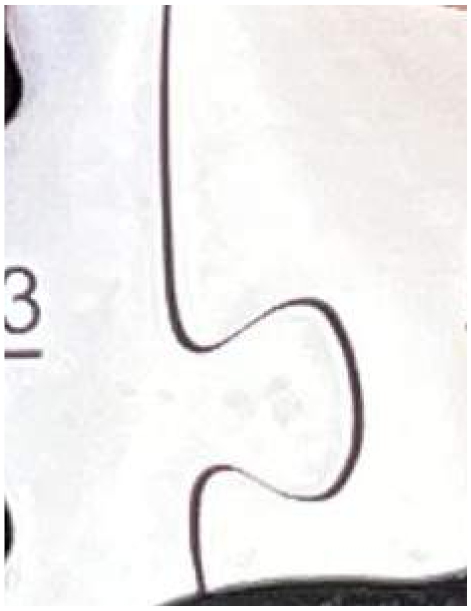
What makes this hard

Imagine solving a jigsaw puzzle. Specifically, imagine two pieces on the table in front of you that you're pretty sure should fit together. Sometimes you need to wiggle and press those pieces to fit together. And sometimes you get it wrong: turns out, those two pieces that look like they really should fit together, don't. That precision in *seeing* and *feeling* is hard – and it's what makes puzzles engaging. We initially underestimated this level of accuracy, which is one of many challenges that turned this project into "a hard but fun project" into "a hard, years-long effort."

To quantify this, the system accuracy required to snap two pieces together is between 0.13 - 0.38 mm. That's the thickness of one to three hairs. That means we need to photograph and understand the detailed shape and position of every piece, then move those pieces up to 2 meters over to the area where the finished puzzle is being assembled, then align with neighboring pieces and place them back down, all while accruing an imperceptible amount of error.

We measured this by commanding the robot to try to squash a piece down into a neighboring piece that we held down, then nudging the robot one thousandth of an inch (0.025mm) over, and repeating, until the pieces no longer snapped together.



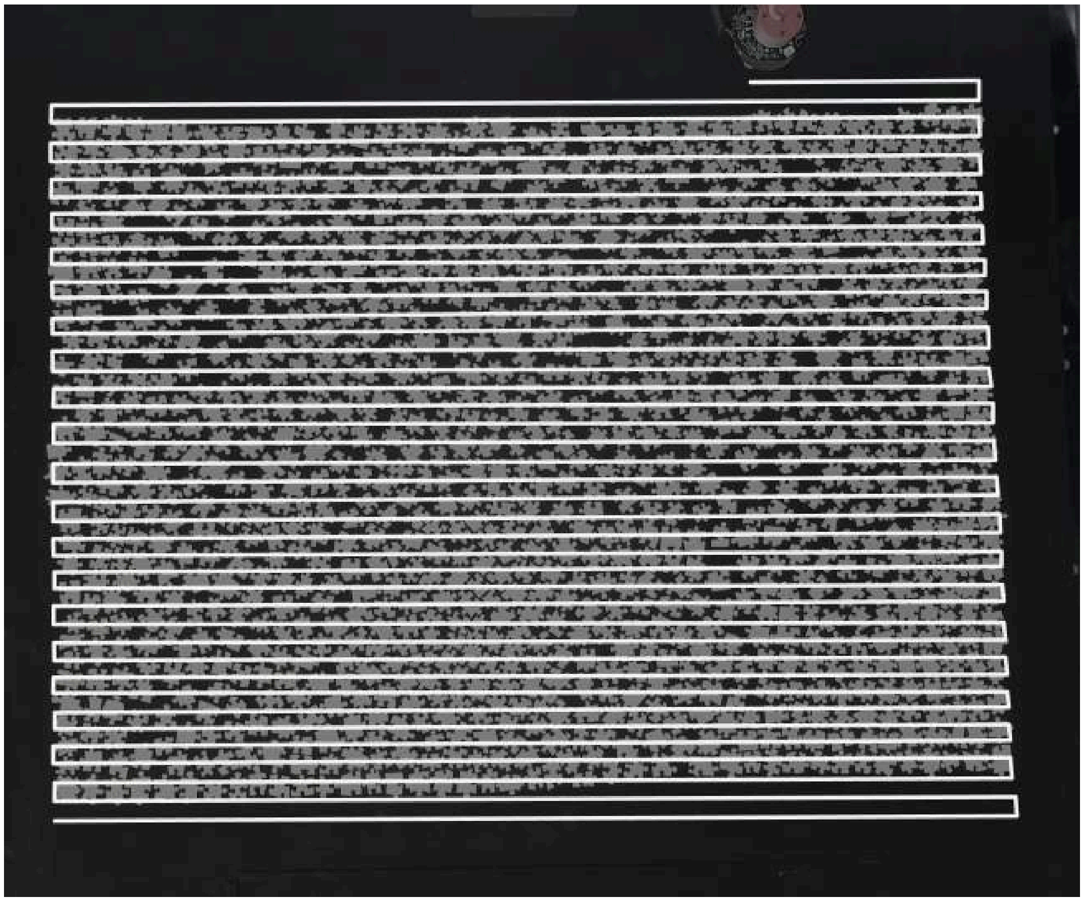


Ok, time to solve the puzzle!

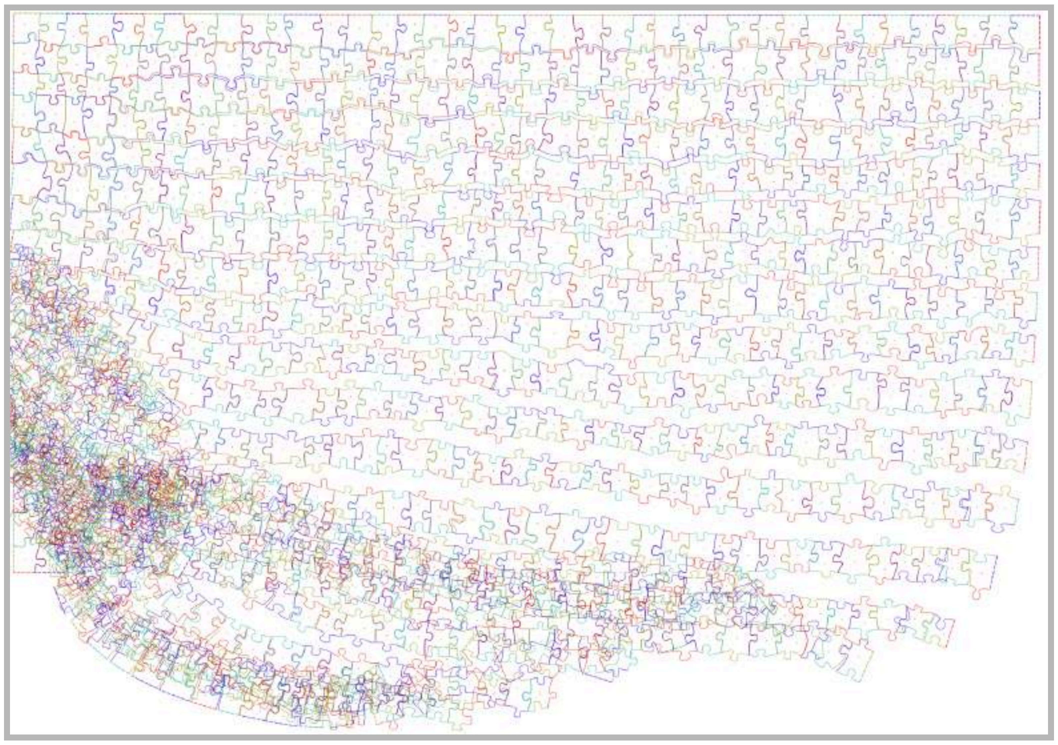
Note: this isn't just one algorithm: solving the puzzle is a chain of a dozen different algorithms so it's easy to get lost in the details. Use the *Outline* on the left to jump around!

Phase 1. Taking photos of all the pieces

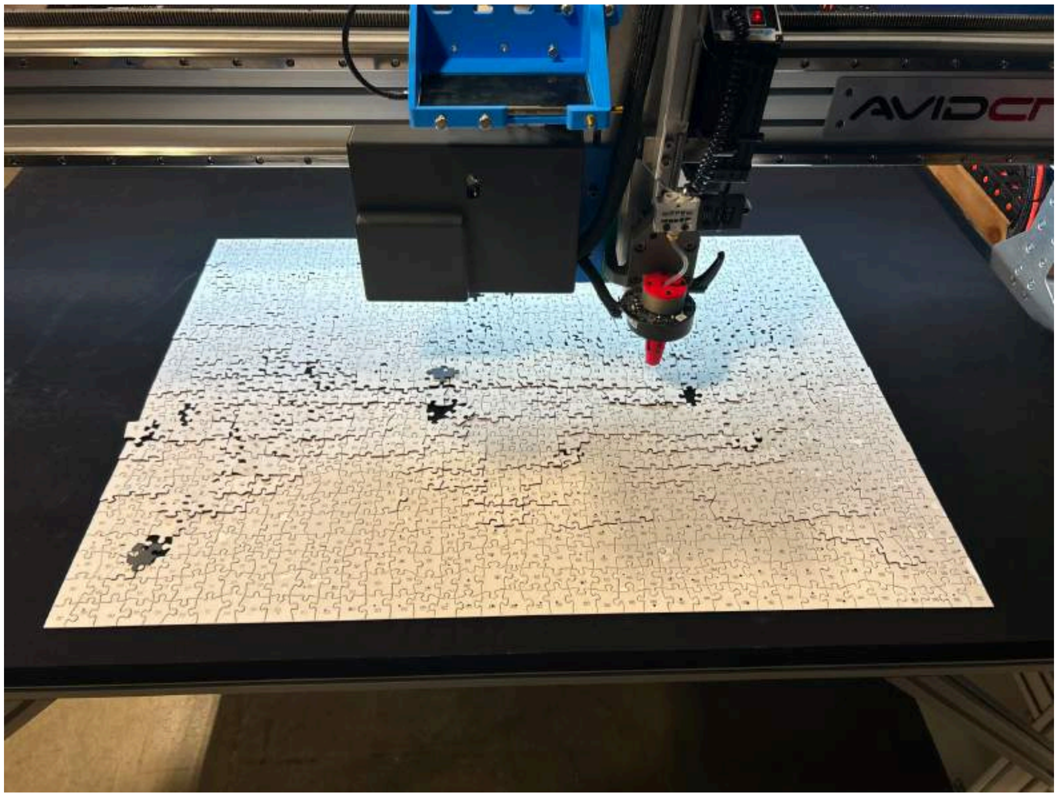
To solve a puzzle, the robot needs to know what the pieces look like and where they are. We require humans to do one favor: lay all the pieces out, face up, in the staging area, with no pieces touching each other. The robot then moves in a tight cornrow pattern over the staging area, taking photos of the pieces as it moves.



Taking pictures sounds like it might be the easy part, but as we fed those initial photos into the puzzle solving algorithm, too much error in those photos compounded to epic fails. We needed to go from where we *see* a piece (in pixel-space), to where a piece actually *is* (in robot-space), to where the piece needs to end up (in our pixel-space solution, as shown below), to the ultimate location in the solved puzzle (back in robot-space). Garbage-in, garbage-out: minimizing errors at the top of that chain (the mapping between what the camera saw to where things physically were) proved to be very important.



Nearly invisible error per piece compounds as we connect pieces. Here, we see the output of how the robot planned on assembling this puzzle, before we fixed many more sources of error.



After weeks of reducing error, we had made progress, but the results were still not perfect. Sidenote: the upside-down pieces are ones that flipped and flung themself like in the "tiddlywinks" game.

So what was causing the outlines of the pieces to appear slightly mis-shaped, and the location of the pieces to be recorded slightly off from the true values? After a thorough investigation we found many sources of error that needed to be accounted for. Some contributed as little as 1 pixel of error and others as much as 10 pixels of error in determining the shape and position of a puzzle piece:

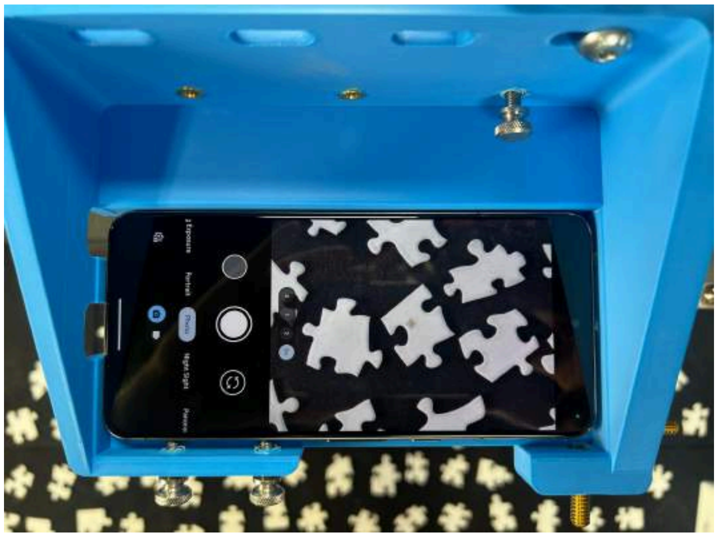
- Robot↔Camera Misalignment, for example:
    - The camera's reference frame was not perfectly aligned with the robot's reference frame, meaning +X motion of the robot

    led to pixel motion in the camera image mostly in the X axis, but with a tiny bit of undesired Y axis motion too
  - The camera was not perfectly parallel with the table, so part of the image was closer to the lens and appeared artificially larger.
- All the pain that comes with using a smartphone as an industrial camera:
  - Lens distortion
  - Aggressive noise reduction leading to artifacts
  - Blooming where bright pixels bleed into neighboring pixels
  - Pixels not being perfectly square
  - Exposure changing when fewer pieces are visible in the frame
  - Focus changing when fewer pieces were visible in the frame

1.1 The Camera

Smartphone cameras and lenses are *just about* good enough to be used for industrial computer vision cameras, if you want to spend weeks calibrating, post-processing, and rigorously controlling your setup. We chose a smartphone over an industrial camera sensor with a telecentric lens because it told a better story. And we felt like making things harder for ourselves.

We used a Google Pixel 8 Pro because it had a nice long lens (113 mm - 5x telephoto), and Android provided simple remote shell automation.



To reduce variance between images, we locked most camera settings (focus, shutter speed, white balance, and ISO) and disabled fancy noise reduction.

We used the Android Debug Bridge (adb) interface to remotely control the Pixel, including triggering the camera shutter and downloading photos from the smartphone. For those interested, look into `adb shell input tap x y` and `adb pull`.

1.2 Lighting

Changes in lighting throughout the day kept biting us: when we'd start the day of testing, too much light led to bloom, where the bright white pixels glowed with fuzzy borders. By the middle of the night, less light in our lab led to grainy photos with strong noise reduction. Inconsistent lighting across a given photo led to computer vision failures. So we blacked out the windows and installed an overhead Aputure Nova P600C light (4700K, 100% brightness) with a softbox, 72 inches above the table bed, which brought the illuminance on the puzzle pieces to 1400 Lux.

1.3 Camera Calibration and Distortion Correction

All camera lenses introduce some distortion into an imaging system. The typical computer vision approach is to take many photographs of a calibration target containing known geometry, then run an optimization process to calculate distortion parameter coefficients, which are finally used to undistort images taken with the imaging system.

High-accuracy calibration target with 10-micron dimensional accuracy

We took 50 photos of this target at a variety of angles using the Google Pixel 8 Pro before it was mounted on the robot, and tried out three camera calibration methods to see which would produce higher accuracy. We ended up using Calib.IO’s software.

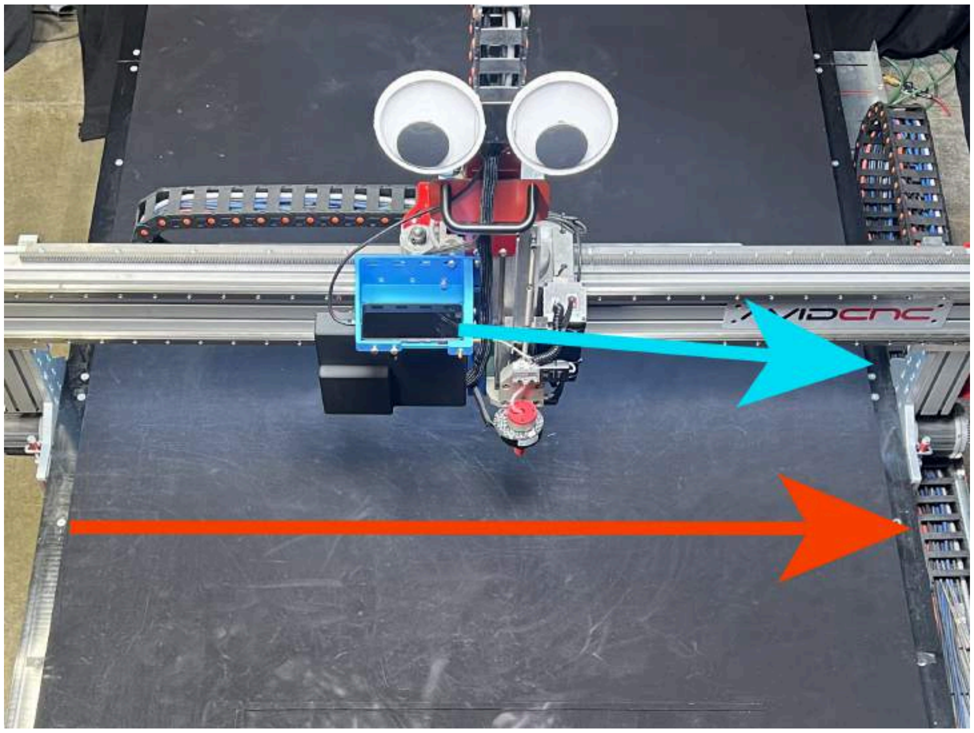
Calibration Method	RMS RPE (lower is better)
OpenCV's calibrateCamera	0.278 px
OpenCV's calibrateCameraRO	0.161 px
Calib.IO's Camera Calibrator	0.091 px

1.4 Perspective Correction

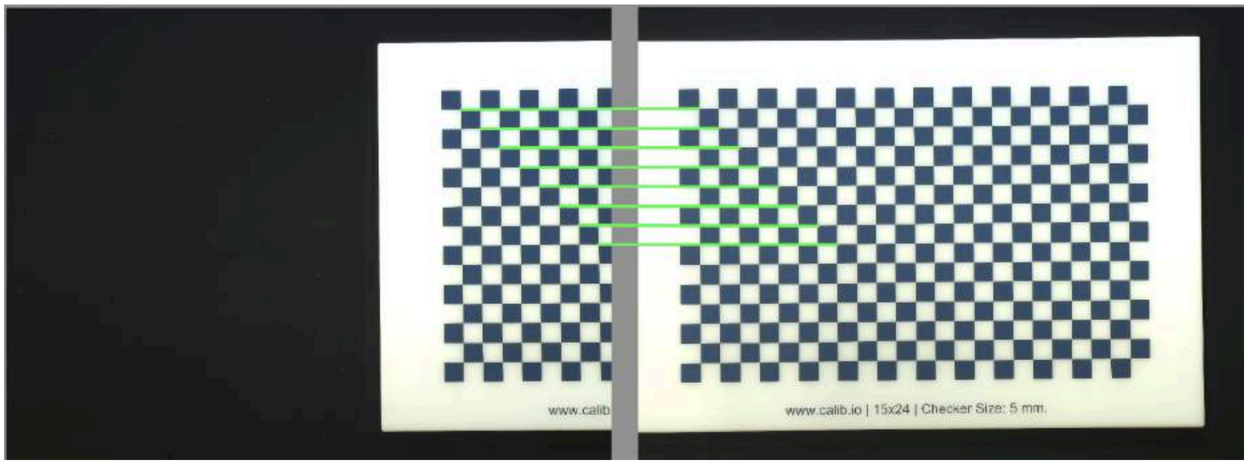
Solving a jigsaw puzzle is inherently a 2D problem, and the photos of the puzzle pieces ideally should show perfect top-down orthographic projection of each puzzle piece. However, photos of the camera calibration target resting on the table, taken by the Google Pixel 8 Pro mounted on the robot, showed a trapezoid instead of a rectangle, meaning the camera wasn't perfectly parallel with the table. We corrected this using OpenCV's getPerspectiveTransform and warpPerspective functions.

1.5 Rotational Correction

Similarly, we noticed that the camera's X axis was not perfectly parallel with the robot's X axis. Of course, this means their corresponding Y axes were also out of parallel by the same amount. We needed to measure and then correct for this error (the angle) between the camera and robot reference frames.



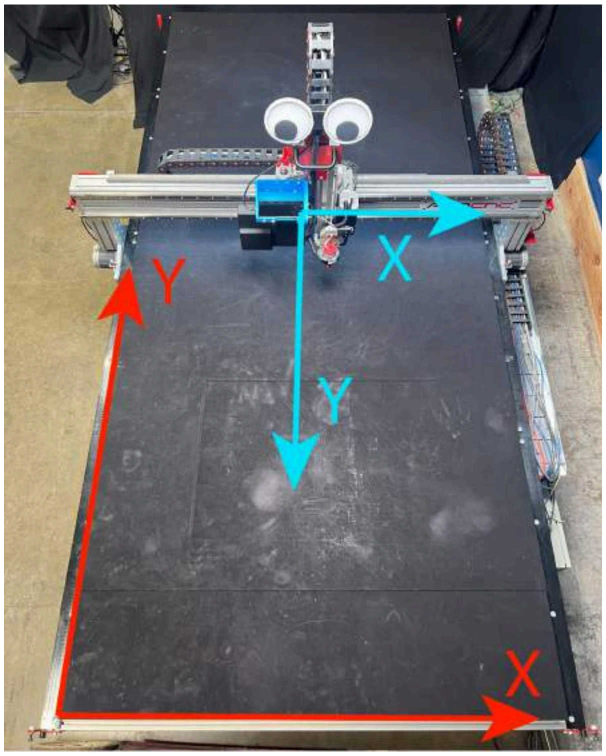
To do this, we took a photo of the camera calibration target, then moved the robot in the X axis only, and took a second photo of the target. We then used OpenCV's findChessboardCornersSB function to find the chessboard corners (saddle points) in both images, and drew imaginary line segments between corresponding features.



The angle between each line segment and the X axis was then measured, and the average angle was determined and recorded. Later, this angle was used to correct for the rotation error and ensure that robotic movements in the X axis resulted purely in X translations of the image.

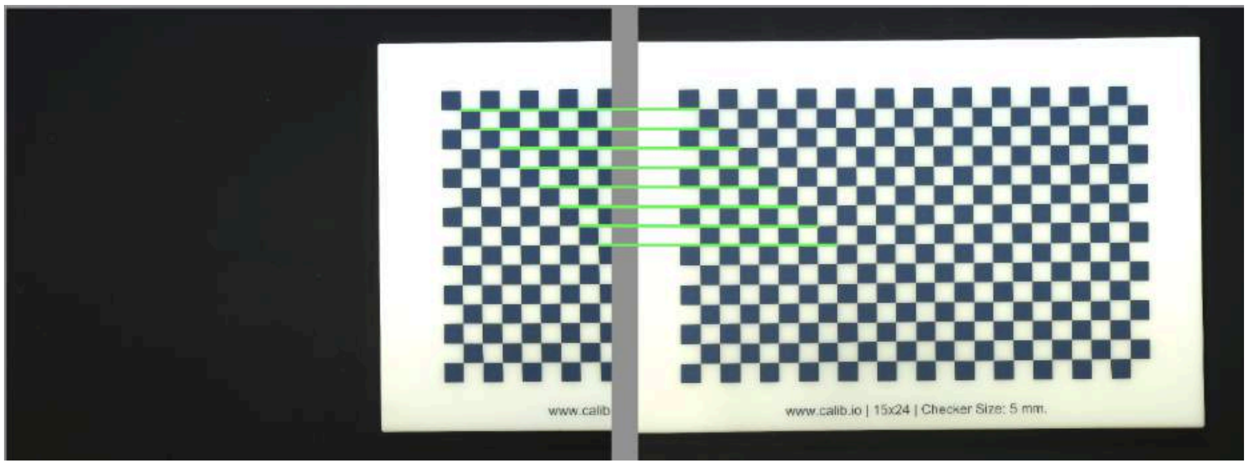
1.6 Converting Between Camera and Robot Coordinate Systems

In order to be able to translate between where we see a piece (in pixel-space), to where a piece actually is (in robot-space), we need to know how to convert pixels (as seen by the camera) to motor counts (the fundamental unit of motion control in the robot).



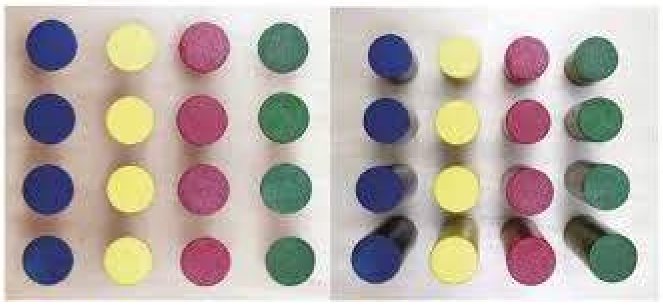
Robot (red) and camera image (blue) coordinate systems

We calculated this by first taking an initial photo of the camera calibration target, then moving the robot by a certain known distance (in motor counts) in both X and Y, and then taking a second photo. It was then straightforward to calculate the relationship between the distance the robot moved in motor counts and the distance the pixels moved in the image.



1.7 Approximating Telecentricity to Minimize Parallax

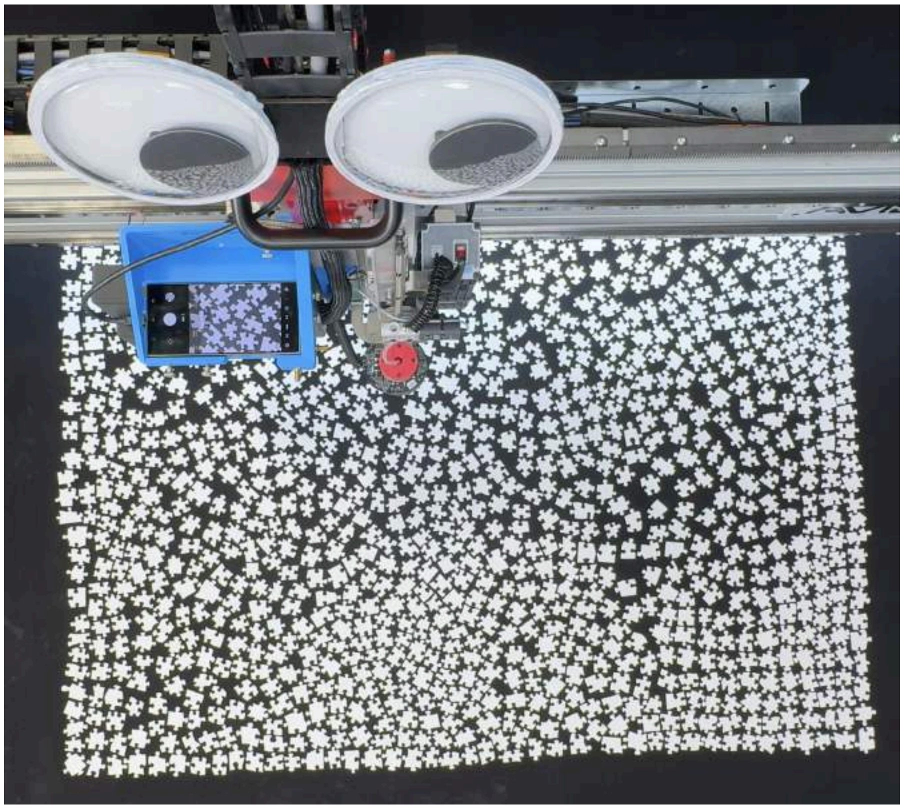
The primary goal of the camera system is to capture images of the puzzle pieces that perfectly show the shape (outline) of each piece. The problem is, conventional lenses suffer from parallax, where objects look different depending on how far they are from the center of the image.



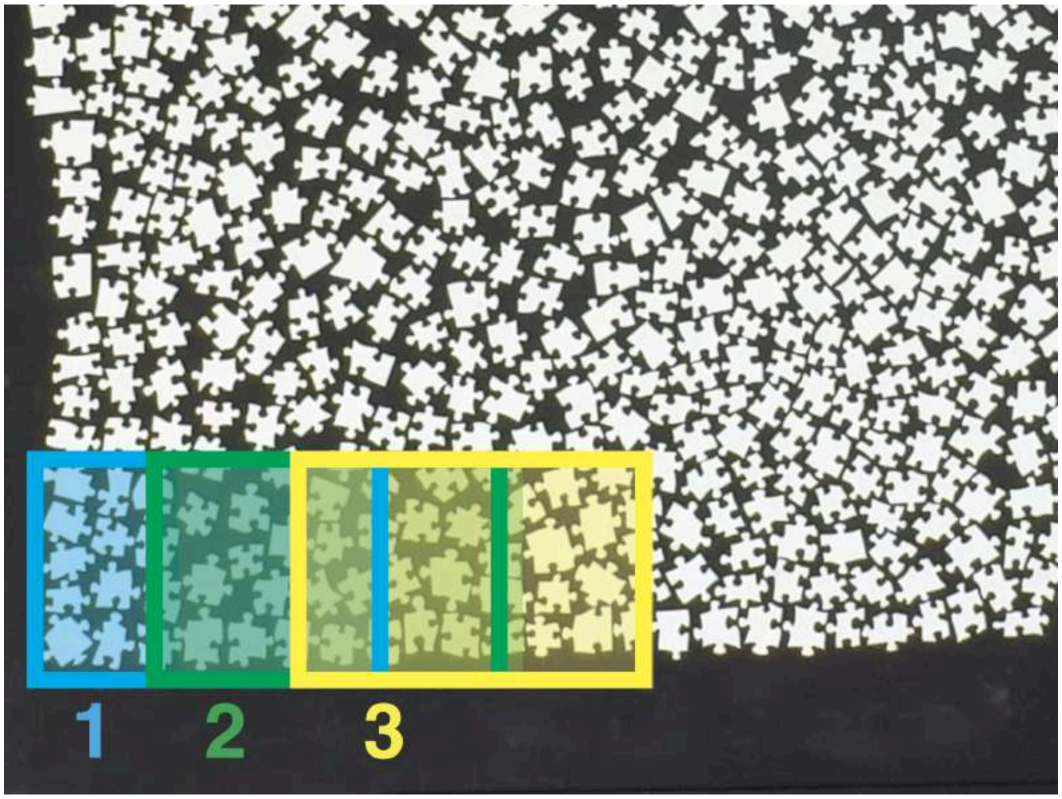
If we were smarter, we would have solved this with a telecentric lens that captures photos like the pegs on the left, but alas our smartphone lens captures images like the pegs on the right. But we can approximate telecentricity by moving the camera further away, using the longest available zoom lens, and even cropping the image tighter to the center of the image. Look at the four pegs in the center of the right image: those ones have much less parallax. The drawback is that we have to take many more photos to be overtop each piece. Photo capture time went from 10 minutes to 90 minutes.

1.8 Putting it all Together

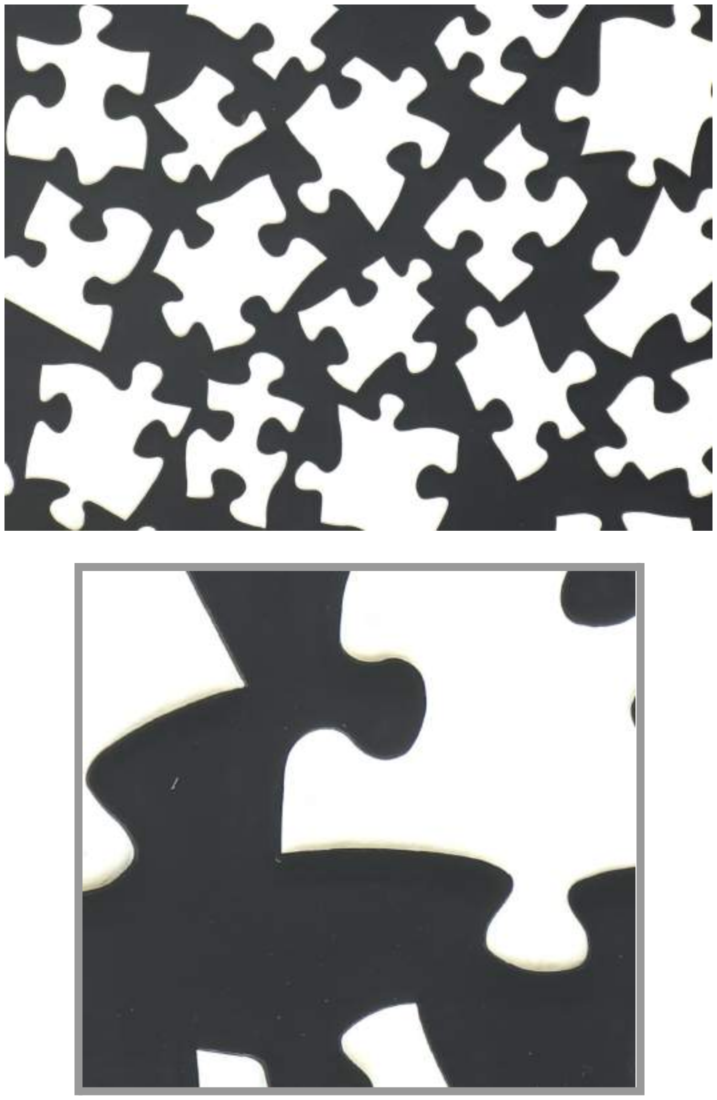
After we've carefully placed all pieces face up in the staging area (and made sure no pieces were accidentally touching), we start the robot.


All 1000 white puzzle pieces laid out on a table (painted with Musou black)

The robot moves in a serpentine path over the staging area, taking pictures every few inches of movement to ensure that every puzzle piece appears towards the center of at least one image.



After each photo is taken, corrections are applied: first camera undistortion, then perspective correction, then rotation correction.



Left: Unedited photo of well-lit pieces on the black backdrop; Right: closeup

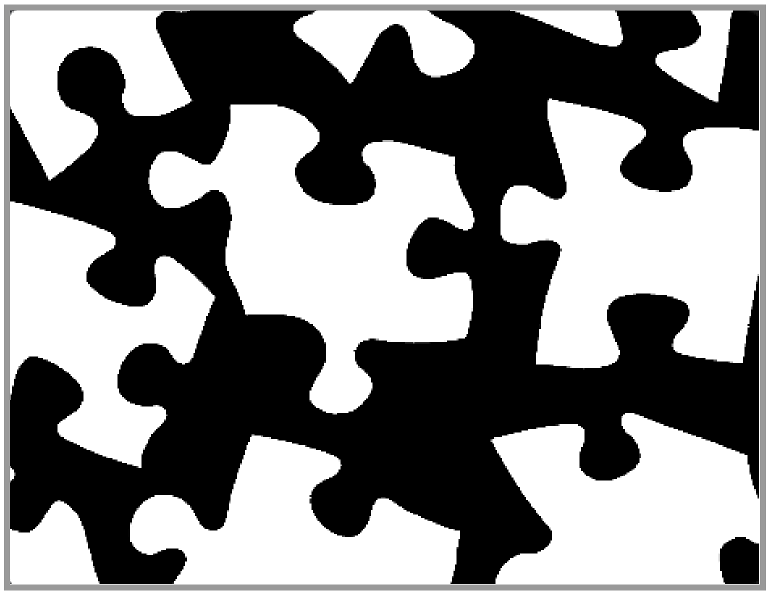
Each photo is saved along with the robot's X and Y position where the photo was taken. These images and metadata are then fed into the solver algorithm described in the next section.

Phase 2. Computing the solution

The overarching goal of Phase 2 is to take the photos of the staging area puzzle and compute a list of robotic moves required to assemble the puzzle. This involves a series of computer vision steps to understand the shape of all puzzle pieces, then find the solution: how pieces fit together.

2.1 Photo Segmentation

We crop photos to remove the excess photo overlap, then convert these photos into binary bitmaps using a pixel brightness threshold that we tuned for our lighting conditions: bright white pieces on a matte black backdrop. We save these off as BMP images.



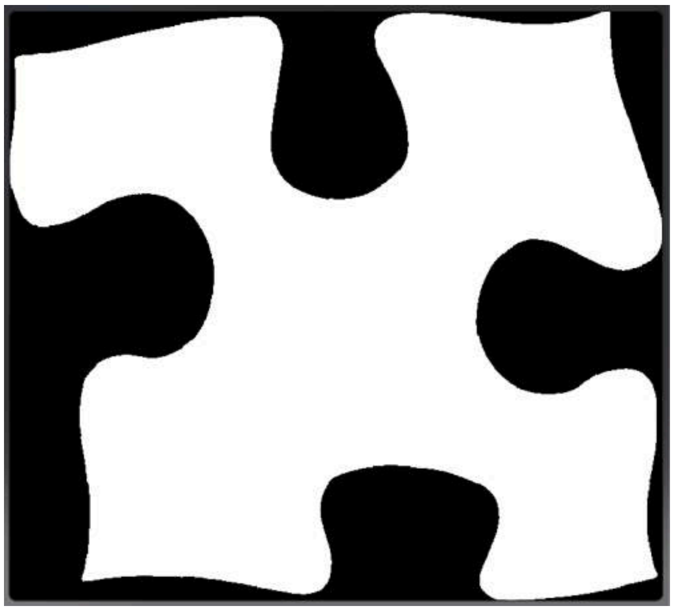
Bitmap after cropping and segmentation

2.2 Extract Pieces

We use an optimized island-finding algorithm on the binary data to find and extract each piece from each BMP. We throw out any islands that touch the border of the image (because those pieces are partially cropped, and we know we'll have a better photo of that piece elsewhere).

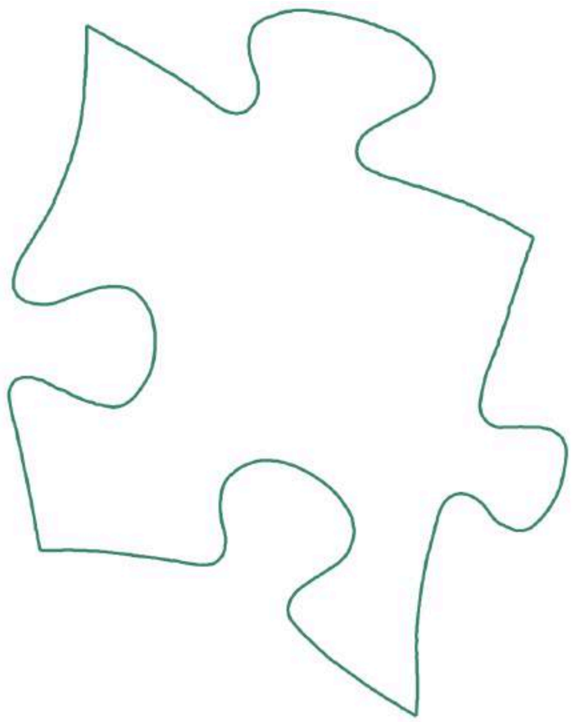
We then clean up the border of the piece to remove small debris like dust and hairs by applying a simple convolution filter across each successfully-extracted piece.

We end up with a bunch of BMPs that look like this:



Note: we will have some duplicate pieces, because we may have fully captured any given piece in a few photos. We'll deal with that later.

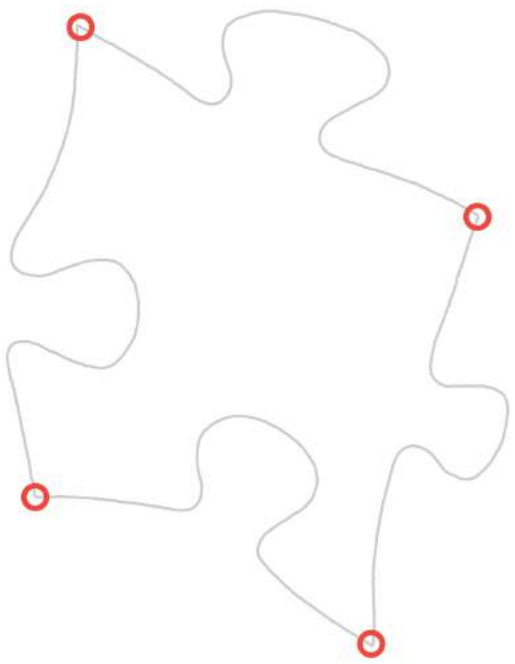
2.3 Create a vectorized polygon of each piece



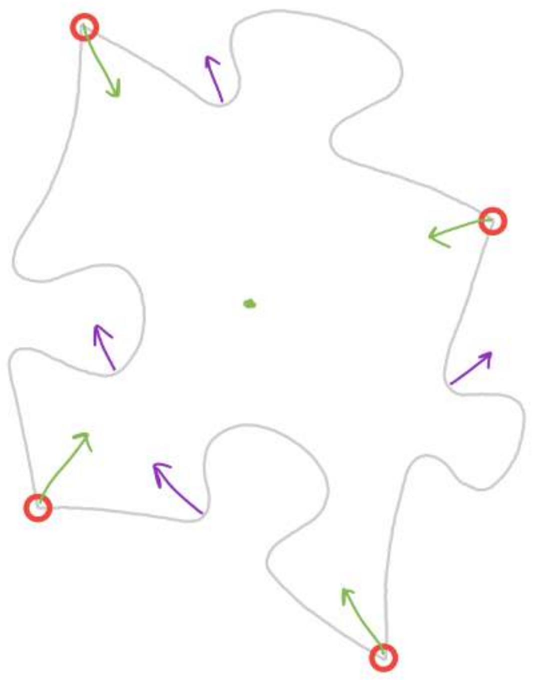
We want to be able to compare how two pieces fit together. This requires us to understand the sides of pieces, and the shape that side has. Bitmap files are just grids of pixels. This step finds the edge of the piece in the binary (black and white) bitmap image, then walks along the edge, creating a dense vector path: a list of coordinates. This polyline represents the vector outline of the piece.

2.4 Find the 4 Corners of Each Piece

Jigsaw is designed to solve a grid-shaped puzzle. This means we can assume each piece has four sides. To find those four sides, we “just” need to find the corners, and the sides will be all the vertices between those corners. Corner detection turned out to be more difficult than you might imagine, but ultimately we were able to reliably detect the four corners of the piece with an algorithm that finds the best four candidates based on a handful of heuristics.

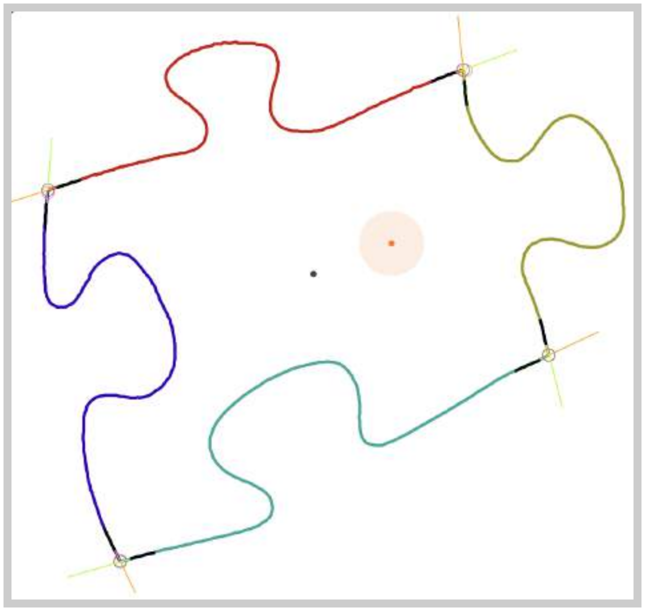


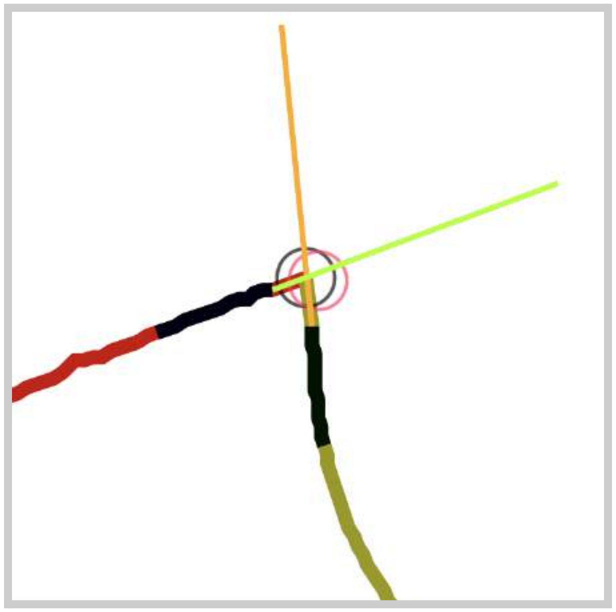
The heuristics compute a score for every point along the vector path, scoring lower based on how sharp the point is and how far from the center of the piece that sharp angle points. For example, the midangles of the four actual corners point generally toward the center, where as the other angular sections of a piece tend to point away:



For wacky shaped pieces, we'll end up with many candidates for our four corners. So we then grade sets of 4 candidate corners based on how cohesive they are together, for example: are they evenly spread radially around the center? We spent more hours than we'd like to admit photographing different puzzles and tuning these heuristics to reliably find the correct four corners.

Because the corners of the pieces can become dented over time, the corner location is further enhanced by finding where the two sides would intersect, creating and saving a "virtual" or "idealized" corner location.





Left: vectorized piece showing circled crosshairs where corners were detected; Right: detail of the top-right corner showing the vertex that was chosen as the best corner candidate (red circle) and the idealized corner (black circle).

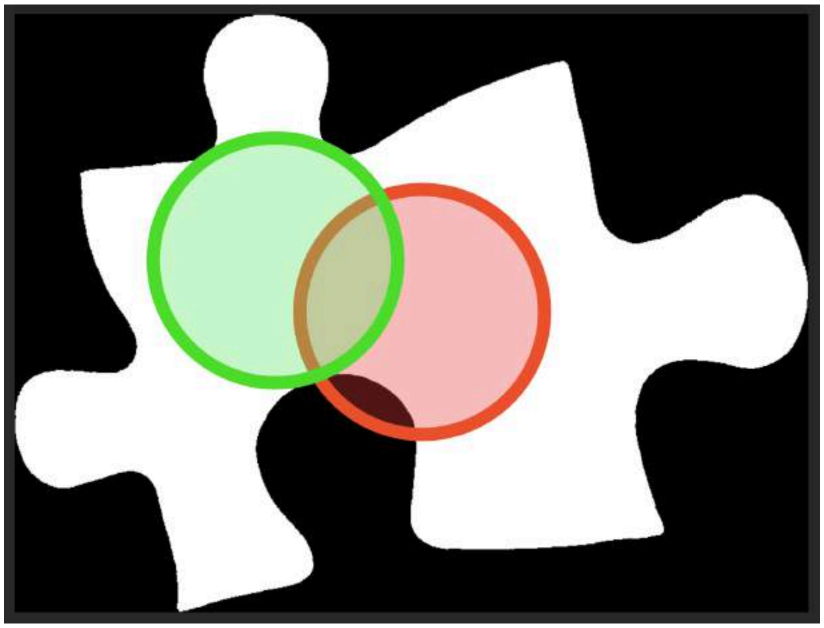
2.5 Extract the Sides of Each Piece

Now that the corners are computed, it is a simple matter to extract the four sides by recording all of the vertices between two consecutive corners.

In this step, we also note which sides are edges along the border of the puzzle by calculating how close to perfectly straight each side is.

2.6 Compute the Grip Point

Some pieces are really tiny and have cutouts near the center of the piece (center of the bounding box). Our robot uses suction to pick up a piece, so it needs a filled area that is 1/4" diameter. Instead of using the geometric center, we compute the *incenter* of the polygon – the most "inland" point that is furthest from any edge where the suction gripper should pick up and rotate around.



The center (red) won't work for gripping, but the incenter (green) is perfect

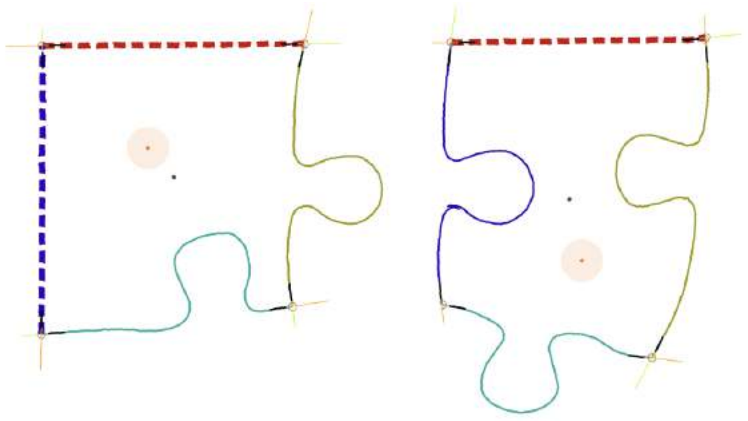
2.7 Deduplicate Pieces

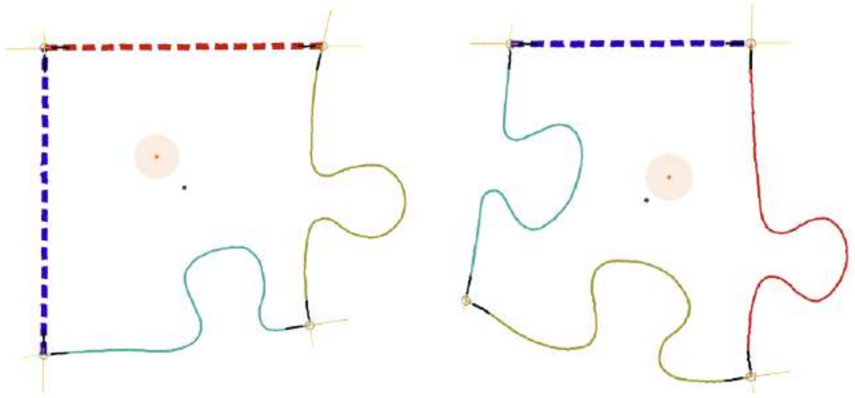
We end up with some pieces photographed two or more times. We detect duplicates then intelligently choose the best to keep and discard the rest. We do this by first computing where each piece exists on the table (aka in robot space), using the position its photo was taken at plus the pixel location of that piece converted into robot space using the ratio we found in step 1.6. Two pieces centered within a few millimeters of each other must be duplicates. The one we keep is the one closest to the center of its original photo, to have the least parallax and distortion.

2.8 Compute Connectivity Graph

It's now time to see which pieces could potentially fit together. In this step we compare how well all 4 sides of every piece "fit" with every other side from every other piece. We compute a score, and keep the best dozen or so candidates. In a 1000 piece puzzle, you'll have 4000 sides, and dozens look nearly identical down to just a few pixels of difference.

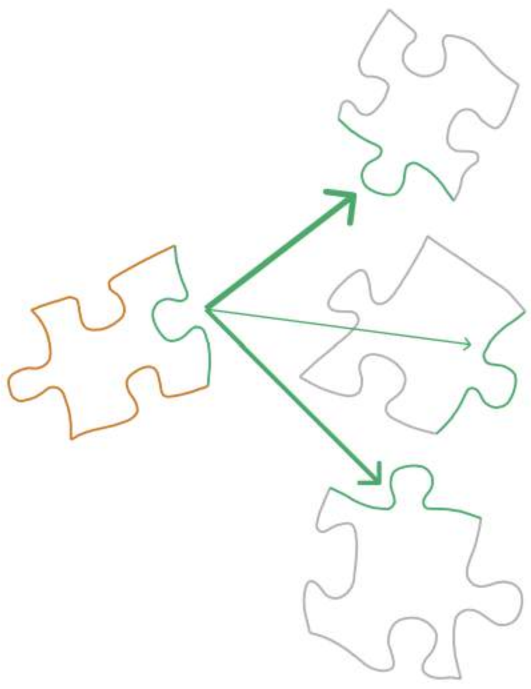
We quickly eliminate many pairings by looking at the length of the 2 sides: if one is much longer than the other, they're not a match. Then we do more expensive math to essentially compute the error between the two curves.


Pieces with similar corner distances that DO fit together


Pieces with similar corner distances that DO NOT fit together

Small improvements to this part of the algorithm have a massive impact on the time that the solver (described in the next section) takes to find a solution. If we find too many possible matches for a given side, the search space blows up and finding a solution quickly balloons from seconds to days to millenia (no exaggeration).

The way to think about this is as a connectivity graph:



For the orange piece, we found 3 possible sides that plug into its green side. The weight of the edge is how perfect the match is. Then each of the four sides on each of *those* other pieces has a set of sides *they* connect to, and those sides have their own matches, and so on. This expands into a massive graph with 4000 nodes (4 sides per piece) in a 1000 piece puzzle. And the number of edges in the graph is at best the one proper match per side, and at worst, up to a few dozen possible matches that look really similar. Early tests of the robot found on average 13 close matches per side, (over 50,000 total edges in the graph), and the solver failed to finish after 24 hours of runtime, unsuccessfully exploring over 100 million paths through the graph. Our final system found 6 close matches per side, but even more important had a near-perfect hit rate for the edge with the highest weight: for all but 26 sides, the correct match was the first guess, and those remaining 26 were the second guess.

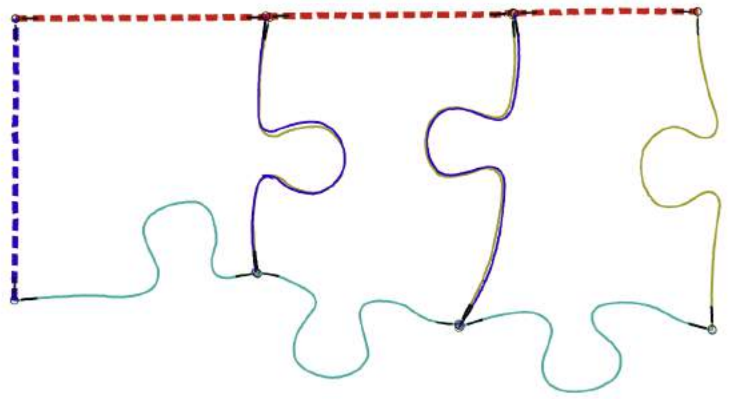
The computational complexity of the graph-search balloons so quickly depending on how good the heuristic is that it's really important that our first guess for most pieces is the correct answer. This is why every step leading up to this point is so crucial: tiny geometric distortions in position and scale can lead us down the wrong path.

2.9 Solve the Puzzle

To solve the puzzle, all we have to do is find the right path through the connectivity graph. But what does the right path mean?

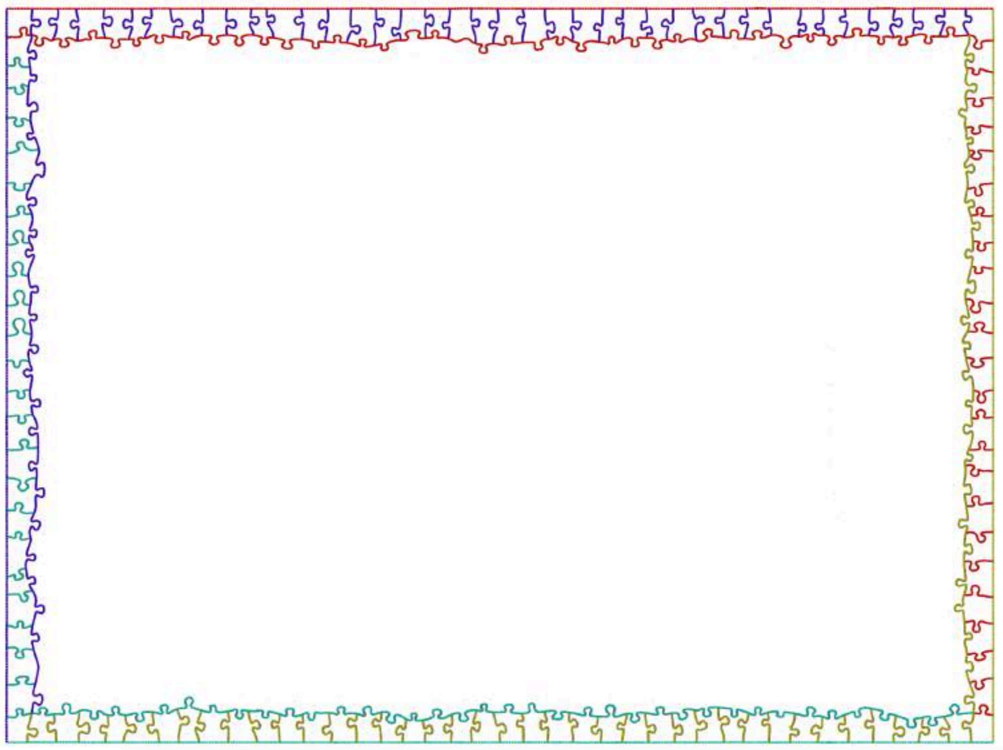
- We traverse the graph going through every single node exactly once
- The traversal makes geometric sense with respect to the shape of a piece
    - We add virtual edges in the graph between sides on the same piece, connected to a node at the "center" of the piece to make sure you have to go through a piece, to bind the four sides of a piece together.
    - These "center" nodes can be traversed more than once.
- We rely on the heuristic of a border to help us out in two ways:
    - First we start with a corner piece and solve clockwise in "spiral" order, walking the whole border then spiraling inward.
    - We also require that pieces placed along the border have an edge side that butts up against the border.

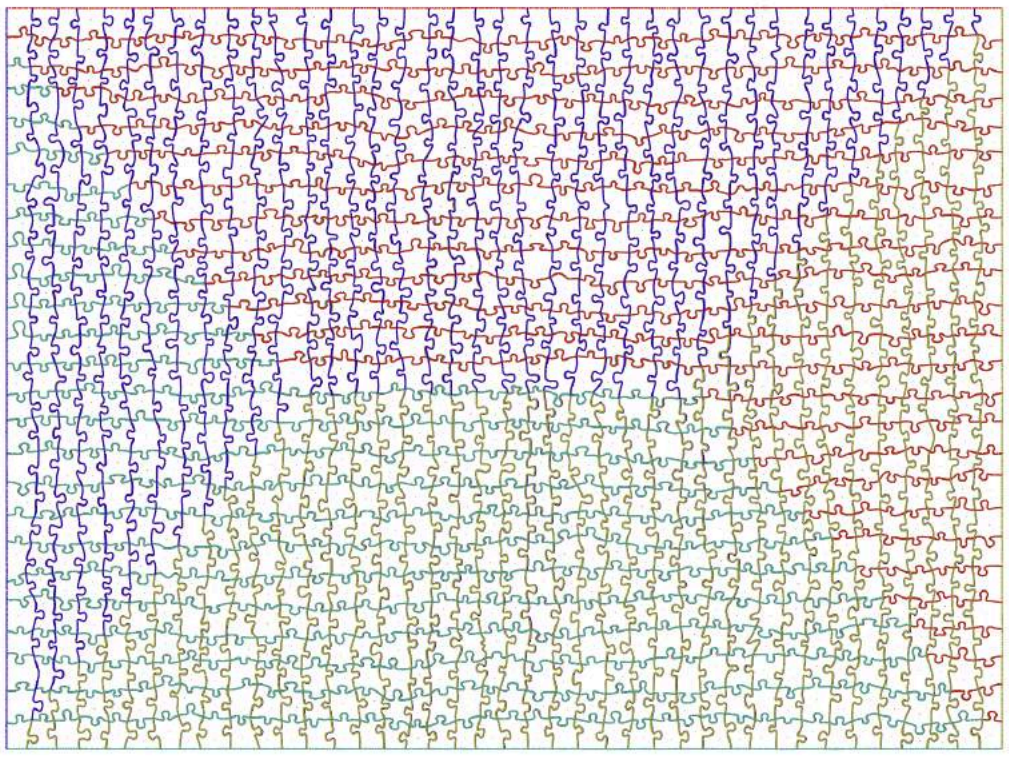
- This allows us to find the correct path along the border quite quickly: there are way fewer border pieces and their orientation has to be constrained.


The first three pieces have been placed correctly.

If it doesn't solve the entire puzzle and can't place the next piece, it un-spirals and starts looking at the second best fit before spiraling forward again.

This process repeats until all 126 border pieces for this 1000 piece all-white puzzle are placed, and keeps spiraling inward.





With all our refinements, a good graph traversal took as few as 1026 steps (1000 would be perfect - correct placement for each piece on the first try). This step now runs in under 1 second.

2.10 Refine and “Relax” the Solution

The previous step aligns pieces in the solution to be pixel-perfect overlaps. It turns out that the actual puzzle needs a little bit of breathing room between pieces. Before we did anything to correct for this, the robot was consistently placing pieces slightly on top of each other, trying to solve the puzzle in an area about 2% too tight.

We measured how wide the puzzle actually needed to be for pieces to snap together to find the precise amount to relax and expand the spacing.

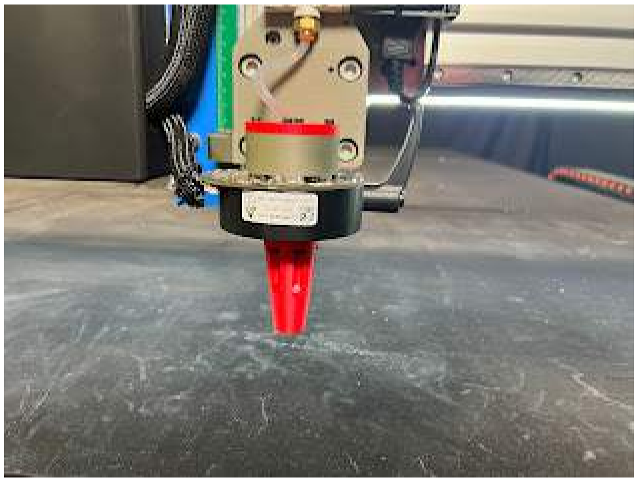
Phase 3. Assembling the Puzzle

You’ve made it this far! Now all we have to do is move the pieces into place. Can’t be that hard, right?

3.1 Where is the Gripper?

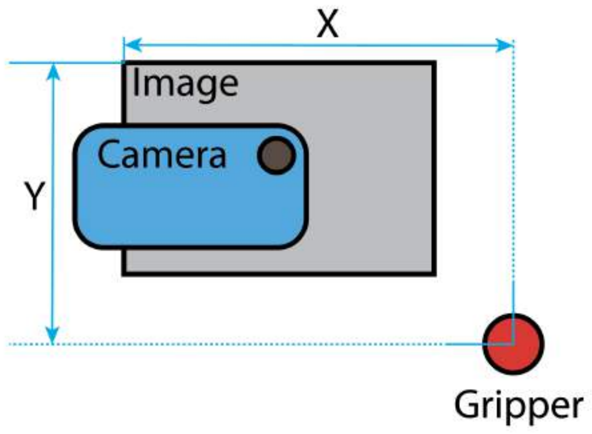
At this point we have the robot-space X,Y position of where we want to pick up each piece. We have that because we know the robot-space position where each photo was taken, and we can convert pixel locations inside those photos into motor counts using the motor-counts-per-pixel relationship developed in step 1.6. It seems like nothing is missing! Except of course, we don't know where the gripper is.

I mean, of course *we* can see the gripper. It's the red thing right here that will manipulate the pieces:

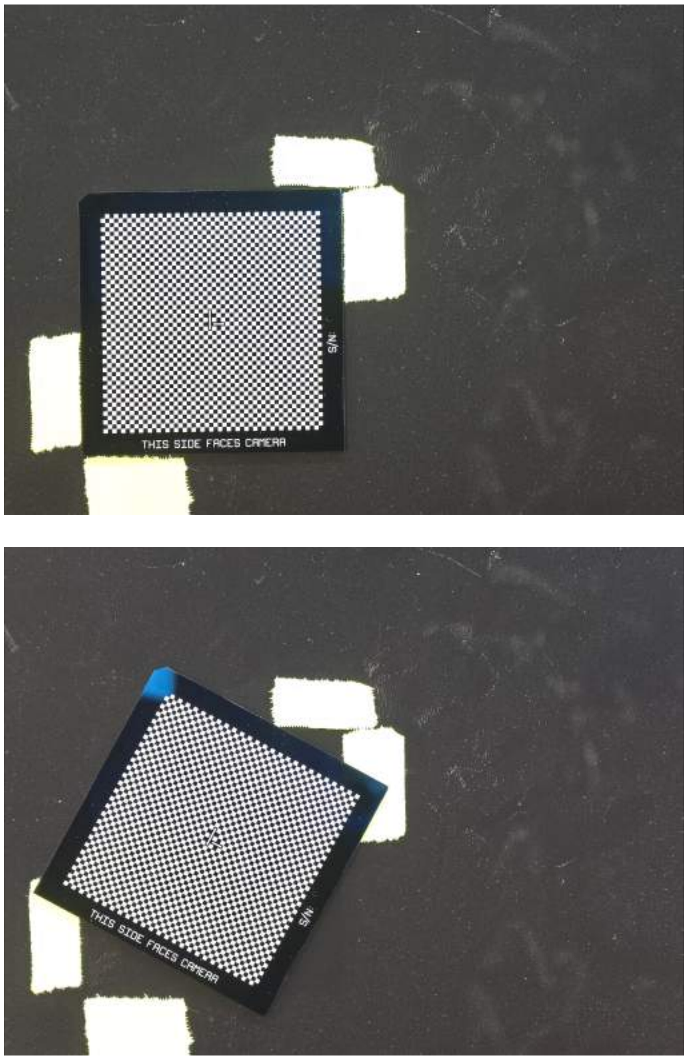


However, we don't yet have a mathematical relationship between the camera's frame of reference and the gripper's frame of reference.

To answer this, we need to measure the X,Y distance between the camera's coordinate system (with its origin at the top left of the photo) and the center of the gripper.



In the world of robotics this is called eye-in-hand or hand-to-eye calibration, and our idea to accomplish this was to take a picture of a calibration target, then command the robot to move the gripper *roughly* over the center of the calibration target (as measured roughly by our eyes), then the gripper would pick up and rotate the calibration target and put it back down. We'd then move the camera back to the initial position and take a second photo. So we have two photos, one where the target has been rotated by the gripper.

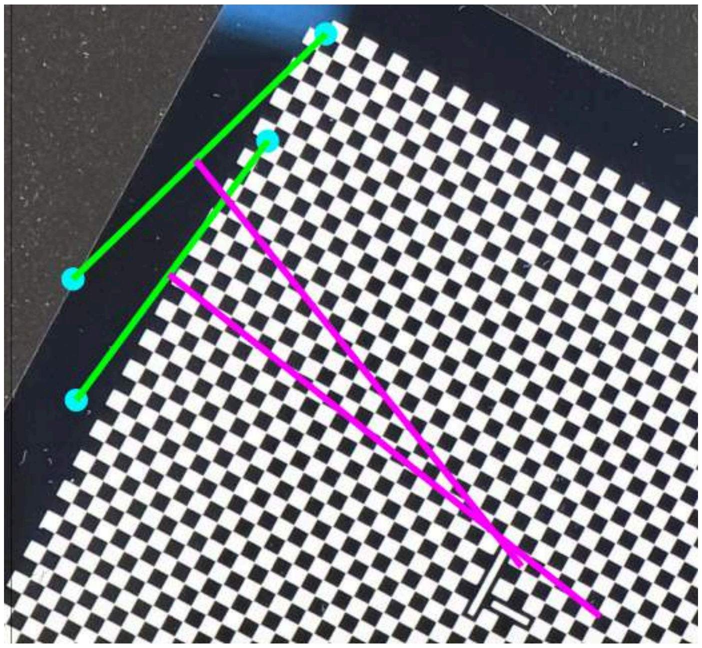


Left: target in starting position; Right: target has been rotated by the gripper

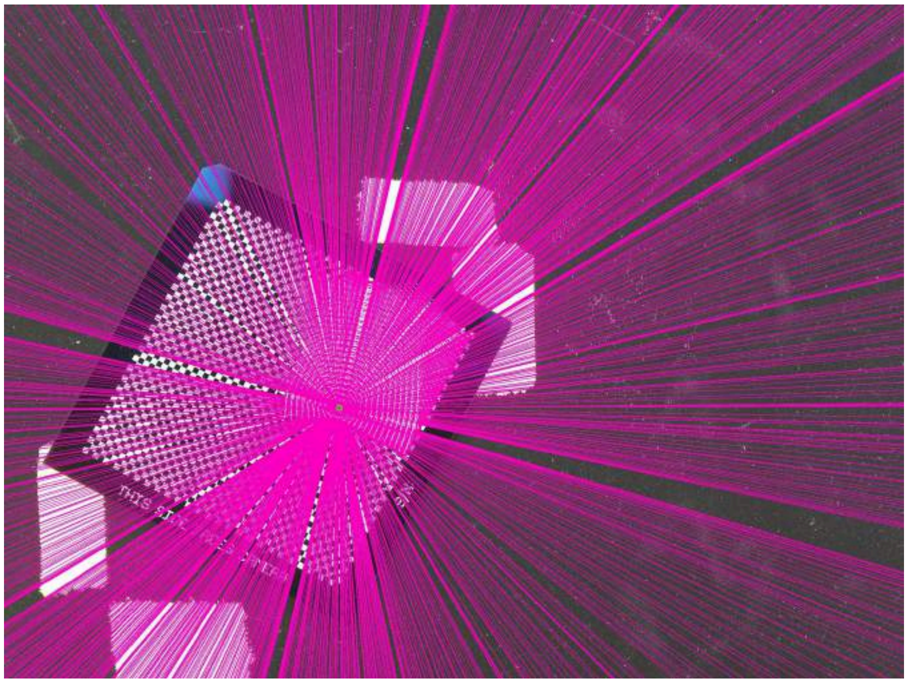
If we can calculate the center of rotation where the target was picked up and rotated, then we have calculated where the gripper is located with respect to the camera's frame of reference.

Calculating the center of rotation turns out to be rather straightforward. Once again we use OpenCV's findChessboardCornersSB to find the corresponding corners in each image, and once again we draw a line segment (shown in green) between those corresponding points.

However this time rather than just measuring the distance, we draw a perpendicular line (shown in pink) in the middle of that line segment. If we do that for 2 sets of corresponding features and find the intersection of the pink lines, we have found the center of rotation.

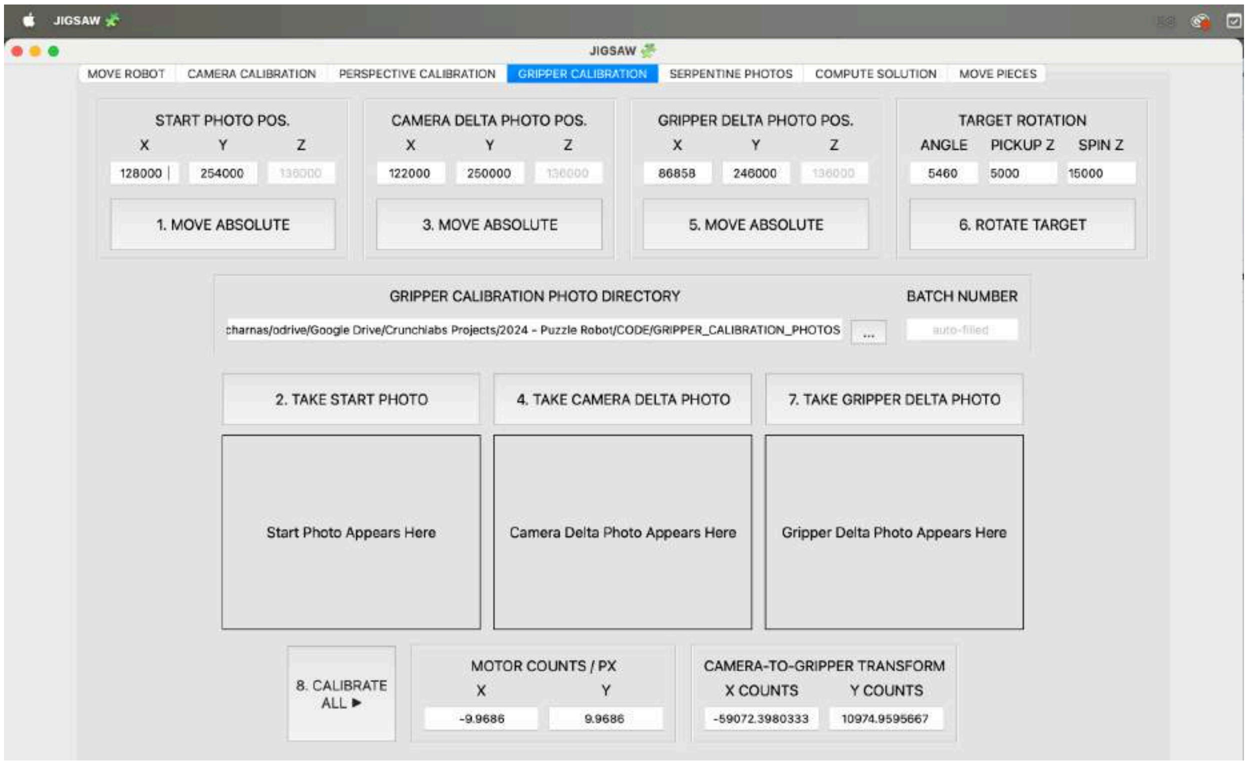


To minimize error, we make these line segments for not just 2 corresponding features, but from all of them.



Essentially "averaging" many sample points helps reduce calibration error.

This calibration was done with the aid of a python GUI written using PyQT5. Actually, all of the robot's operations were controlled with this app, which helped avoid command-line mistakes late at night when we were very, very tired.



START PHOTO POS.

X	Y	Z
128000	254000	136000

CAMERA DELTA PHOTO POS.

X	Y	Z
122000	250000	136000

GRIPPER DELTA PHOTO POS.

X	Y	Z
86858	246000	136000

TARGET ROTATION

ANGLE	PICKUP Z	SPIN Z
5460	5000	15000

GRIPPER CALIBRATION PHOTO DIRECTORY: charnas/odrive/Google Drive/Crunchlabs Projects/2024 - Puzzle Robot/CODE/GRIPPER_CALIBRATION_PHOTOS

BATCH NUMBER: auto-filled

MOTOR COUNTS / PX

X	Y
-9.9686	9.9686

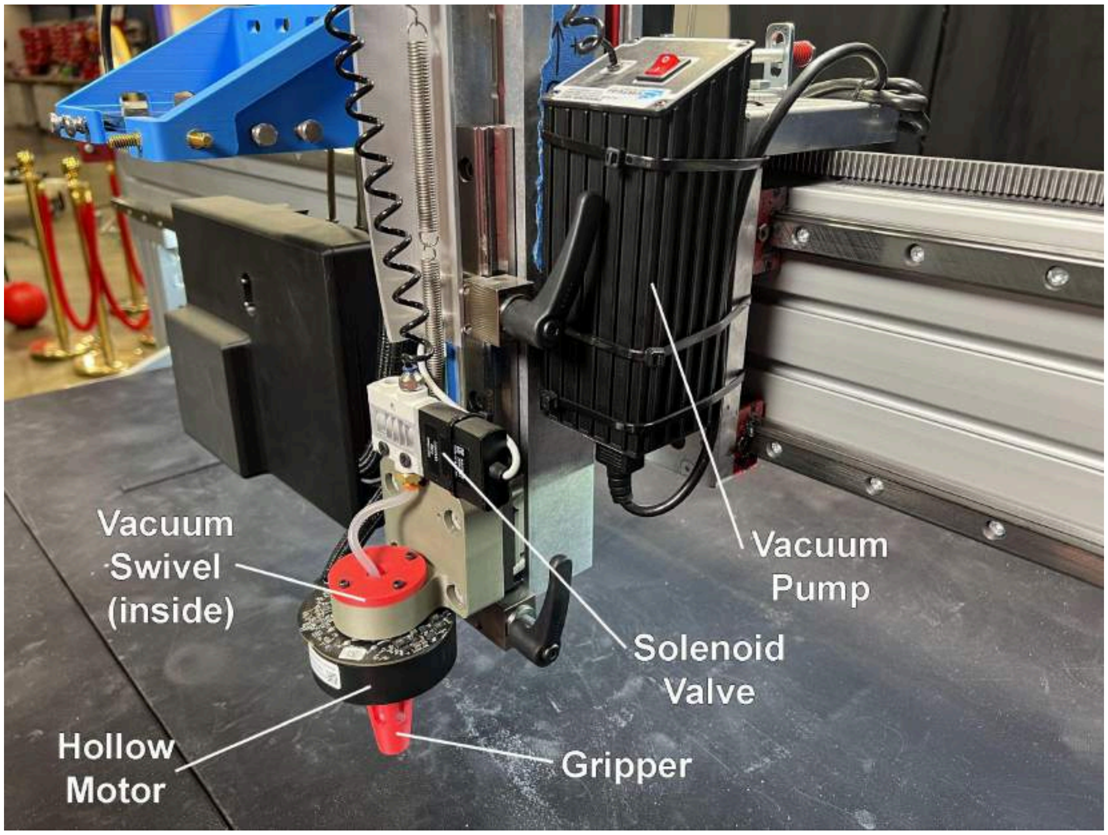
CAMERA-TO-GRIPPER TRANSFORM

X COUNTS	Y COUNTS
-59072.3980333	10974.9595667

Gripper calibration tab of Python QT5 GUI robot control application

3.2 Rotating and Placing Pieces

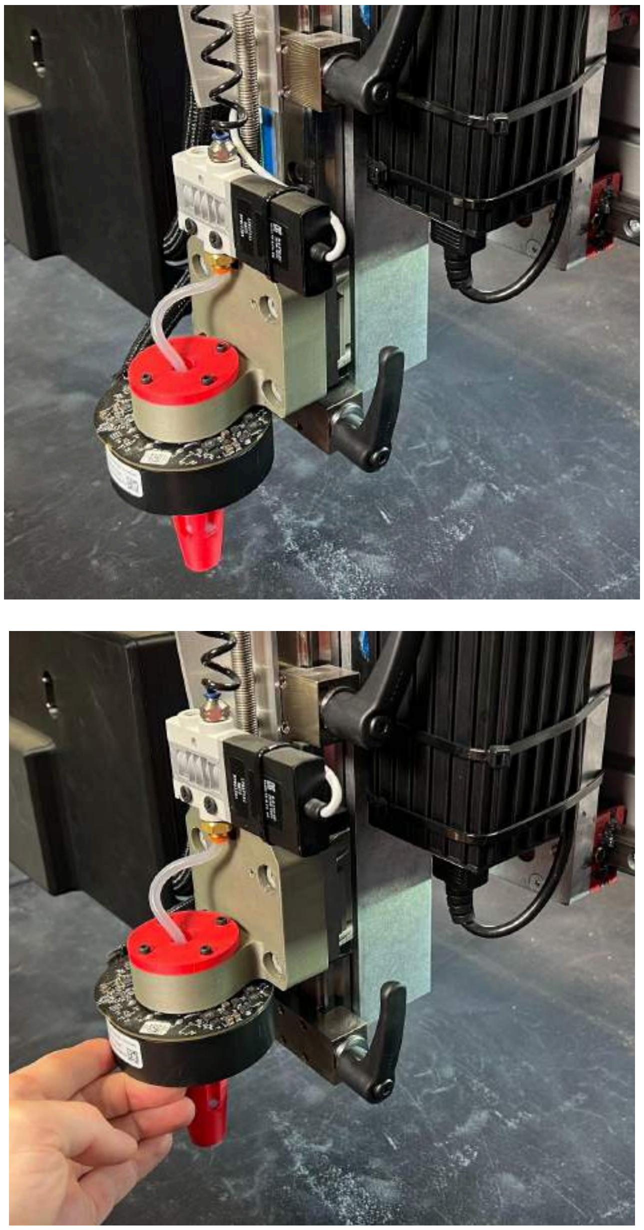
You may be wondering how the rotation works. That's because we haven't yet looked at the mechanism that picks up and rotates the pieces. The gripper is composed of a vacuum pump, a solenoid to connect and disconnect the vacuum pump, a vacuum line, a vacuum swivel (to prevent the line from kinking), a hollow motor that can rotate the gripper, and a vacuum suction cup to pick up the pieces. The red 3D-printed part is there to hold the suction cup and to provide a flat surface to push the pieces down and help them snap into place.



Vacuum-Swivel: Qosina 20022
Hollow Motor: Overview OVU20012
Gripper: Custom 3D printed housing
Solenoid Valve: Festo R-R-FTO-KC-2018-1055
Vacuum Pump: Virtual Industries Tweezer-Vac

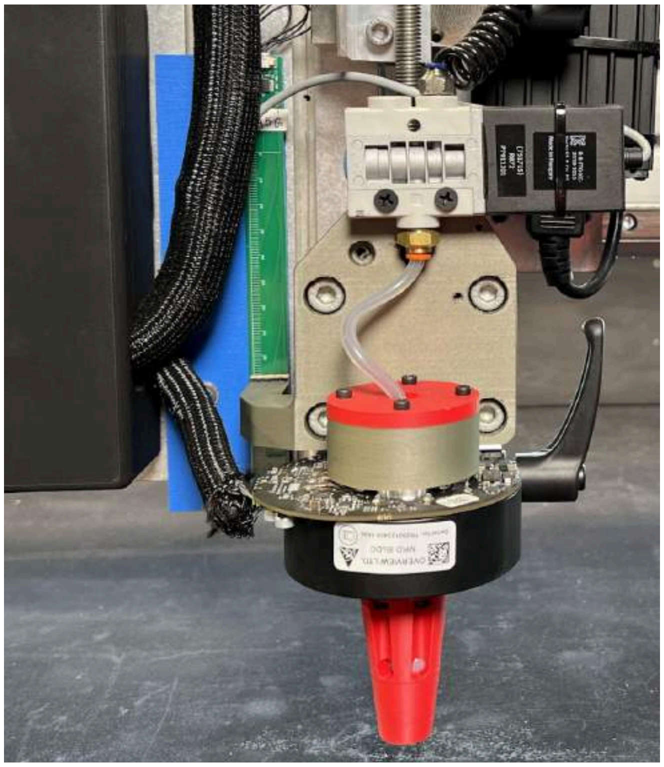
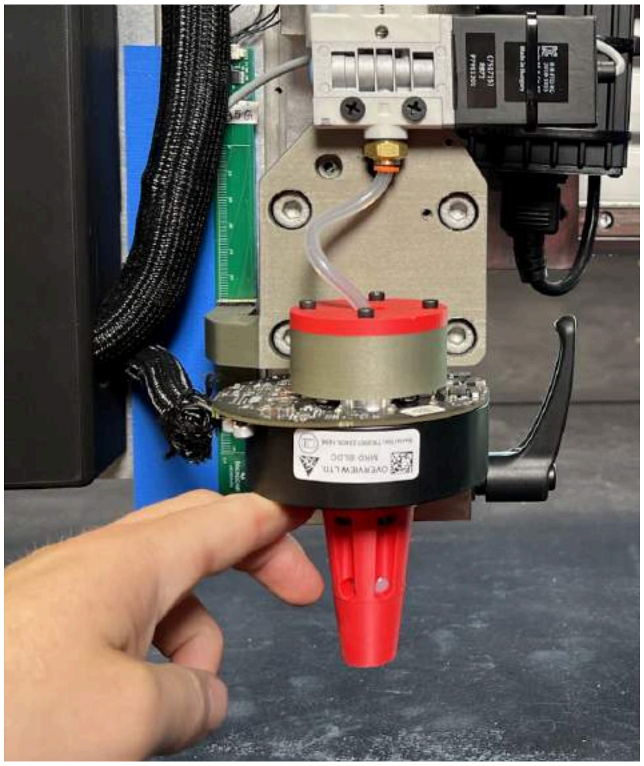
Suction Gripper: McMaster-Carr 3718A63

All of this is riding on a spring-suspension on a linear rail attached to the gantry's Z-Sled. This allows the gripper to slide up and down and ensures it never pushes with too much force onto the table.



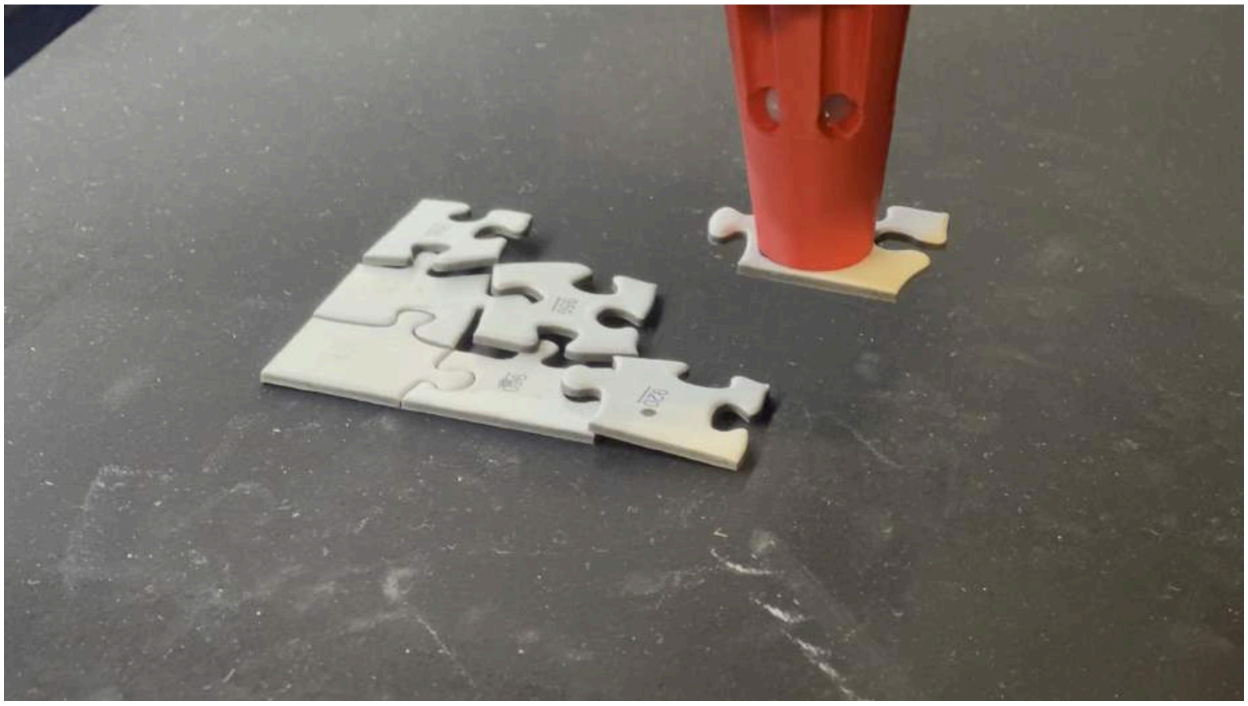
Left: gripper mechanism resting at bottom of linear rail;
Right: gripper lifted upward on linear rail

Not only that, but there is a linear encoder attached to the Z-Sled which allows us to measure how far the gripper has slid along the linear rail. This high accuracy encoder tells the robot whether the piece is fully placed on the table (meaning the gripper is sitting lower on the linear rail) or whether it's stuck on top of another piece (meaning the gripper is sitting higher on the linear rail).

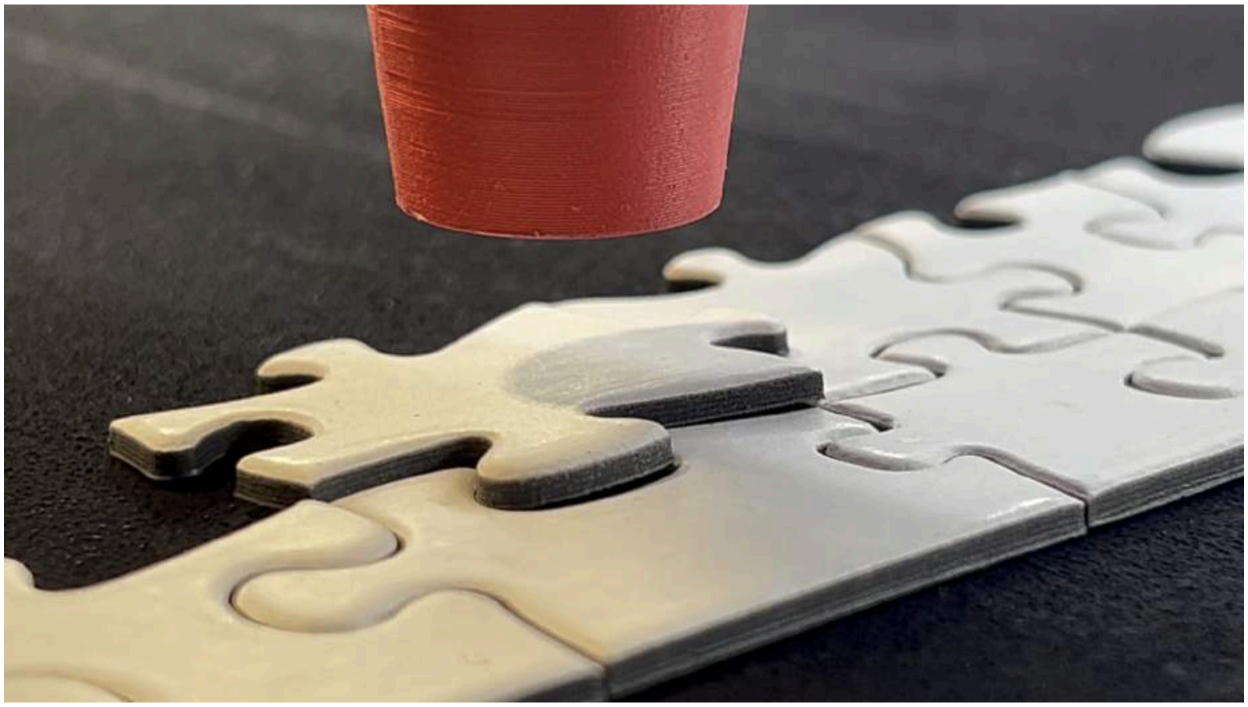


Left: gripper mechanism resting lower on the linear encoder (green PCB)
Right: gripper lifted upward on linear encoder (green PCB)
Linear encoder is a Microchip LX34211 inductive position sensor.

3.3 First Attempts and the Wiggle Routine

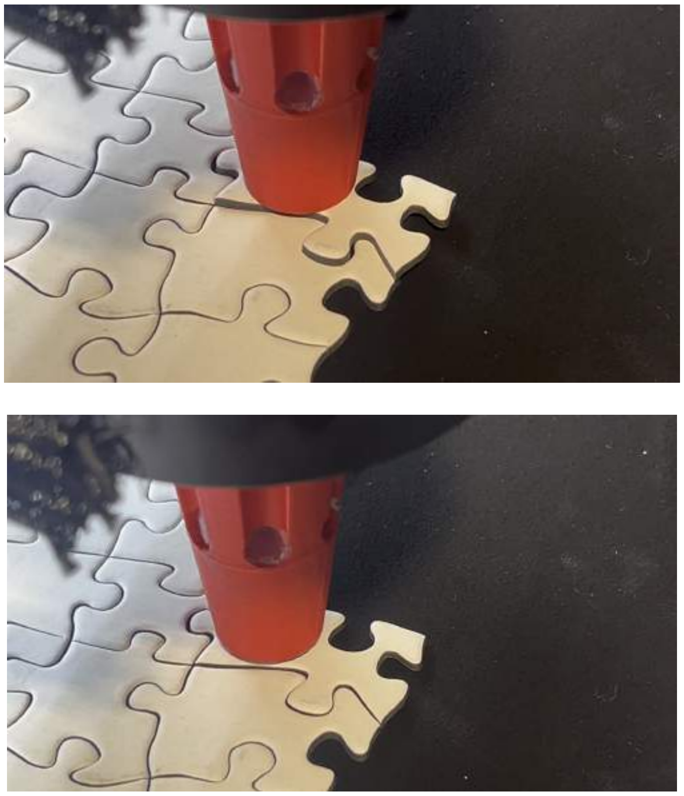
When we first tried to assemble the puzzle, almost none of the pieces fit together perfectly. This was before we had corrected the errors in the computer vision code as described earlier.



However even after we improved the computer vision code, some small errors remained. Many pieces would fit together perfectly, and then you would see one that was ever so slightly out of place, and that could ruin the alignment for the rest of the puzzle if left unresolved.

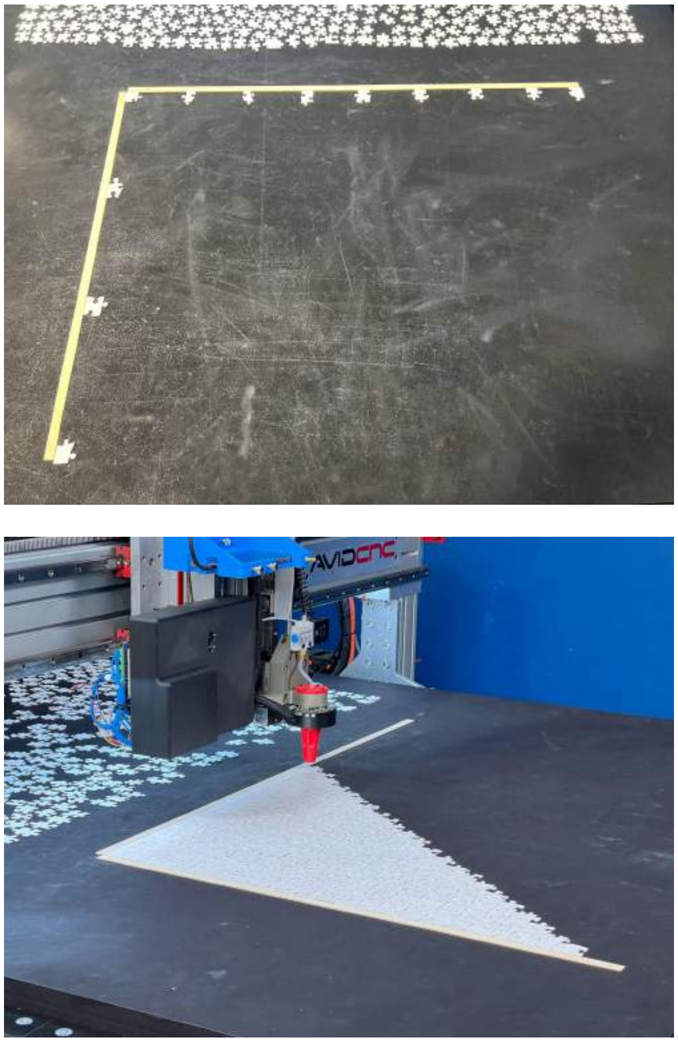


To solve this, we took inspiration from humans. If you try to place a puzzle piece with your hands, you’ll find that often you need to wiggle the piece around to get it to snap into place. So we programmed the robot to do the same thing.



This piece was initially not placed perfectly. The linear encoder detected it was not fully set, and lightly wiggled the piece in both X and Y until it snapped into place.

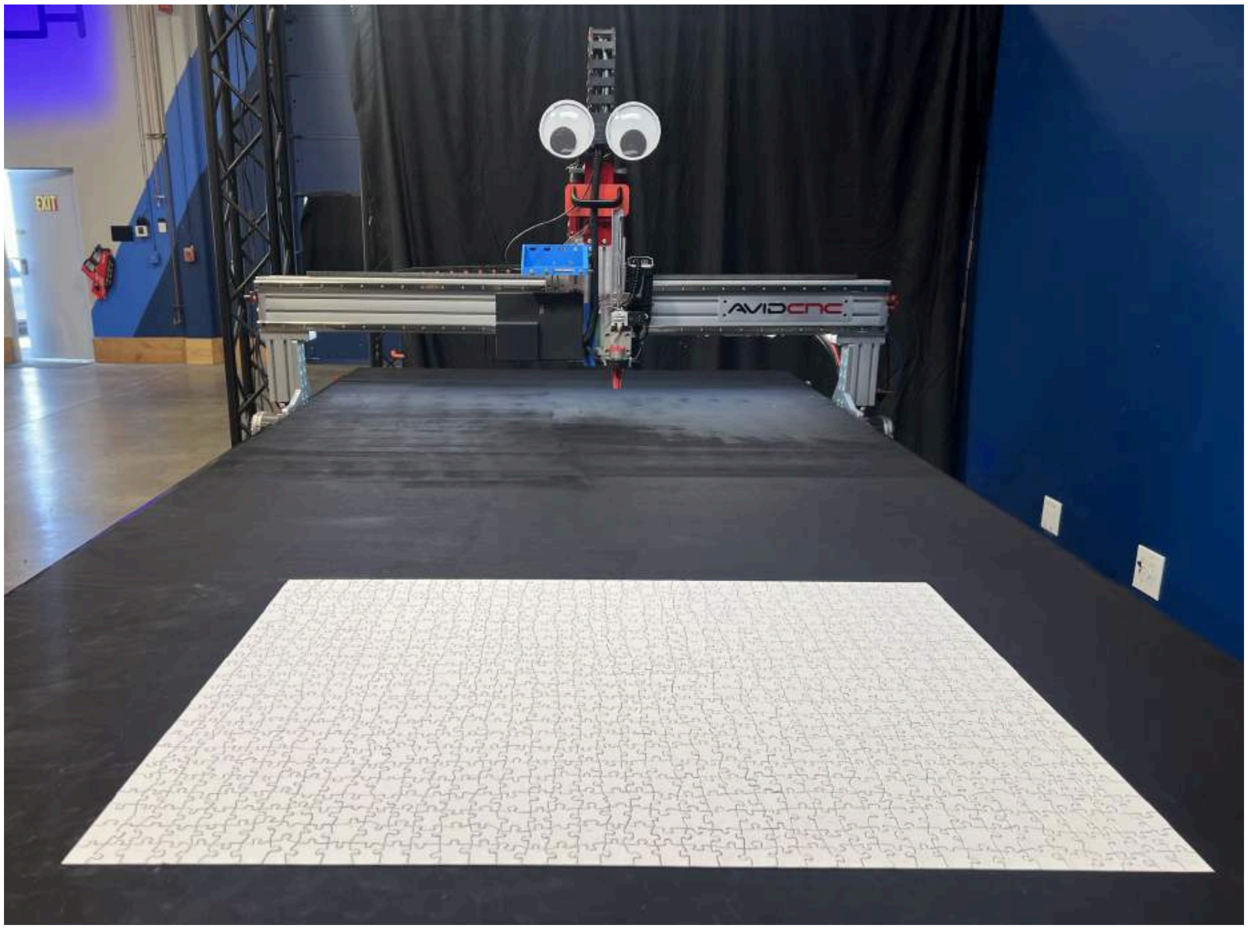
The only problem was that wiggling the piece around would tend to shift the entire puzzle around, messing up the next piece's placement, requiring even more wiggling. We added strips of wood (later painted black) to the table to use as consistent registration stops. The wiggle routine was then modified to push pieces towards these stops once they were placed properly, and this ensured the puzzle pieces would end up in consistent positions, nestled up against the registration strips.



Left: wooden registration strips taped to the table; Right: puzzle robot using the registration strips to place pieces in consistent locations

3.4 Final Attempt

Finally, after more than a year of effort and headaches, excitement and setbacks, close-but-not-perfect runs, the robot finally assembled the 1000 piece all-white puzzle without any human touches to the pieces. Joy, frustration, exhaustion. An end-to-end solve of this 1000 piece puzzle runs as fast as 5 hours.



We predicted the world's foremost jigsaw puzzle solving champion would take 25-40 hours to assemble this 1000 piece all-white puzzle, but Jigsaw solved and assembled the puzzle in under 6 hours! Unless you count the 15 months of development time…

Source Code

Puzzle Solving Codebase in Python and C, by Ryan Oksenhorn

https://github.com/roksenhorn/puzzle-bot

Robot GUI including Calibration Routines in Python, by Ian Charnas

https://github.com/markroberyoutube/puzzle_bot

Special Thanks

Clockwise from lower left: Tammy McLeod, Mark Rober, Ben Varvil, Alexander Kernbaum, Ryan Oksenhorn, Umberto Scarfogliero, Ian Charnas

We couldn't have made this happen without significant contributions from these fine folks:

- Tammy McLeod, for representing humanity in a race against the robot
- Alexander Kernbaum, Umberto Scarfogliero, and Ben Varvil at Robotic Systems

- Ahren Johnson and Cory K. at AvidCNC
- Erik Morrell, Aimee Frank, and Brendan Flosenzier at Teknic
- Jakob Wilm and Eythor Runar Eiriksson at Calib.io