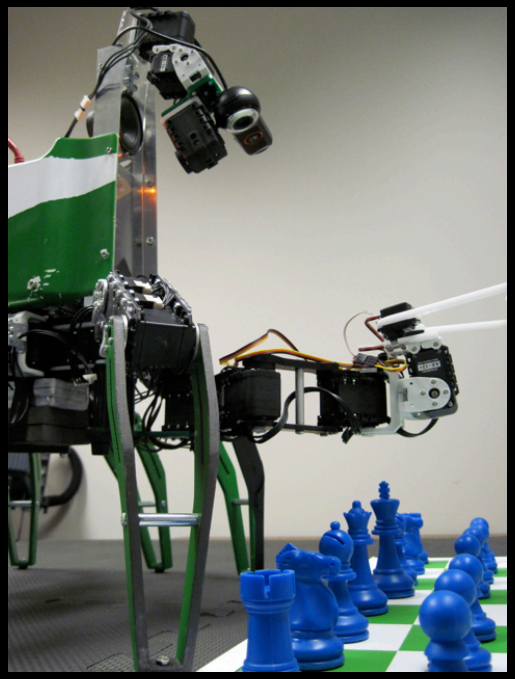
Jonathan Coens

Thesis Committee:
D. Touretzky, chair
I. Nourbakhsh



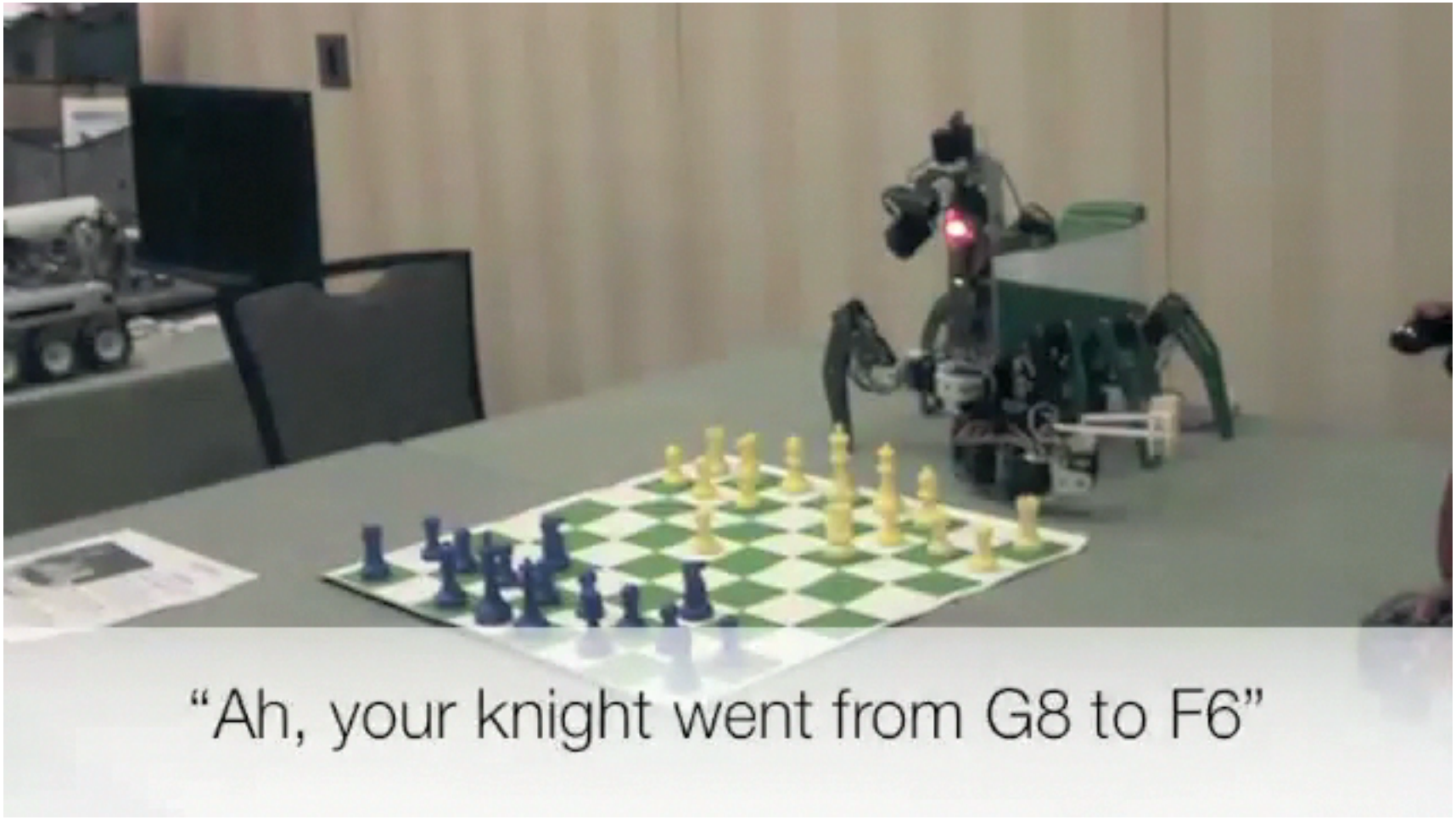
Taking Tekkotsu Out of the Plane

August 2, 2010
MS Thesis Defense
Computer Science Department
Carnegie Mellon University

Introduction

- Extend existing framework to accommodate 3D tasks
  - Perception + manipulation
- Play chess on a real chessboard
  - Competed at AAAI
- Issues
  - Perceiving board
  - Choosing move
  - Manipulating pieces

Chiara robot at AAAI-10

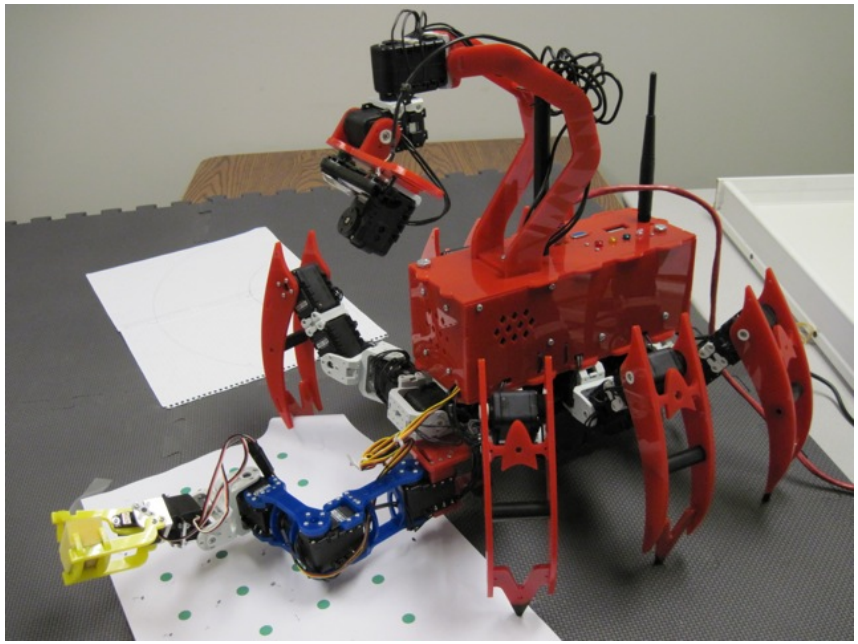


The Chiara

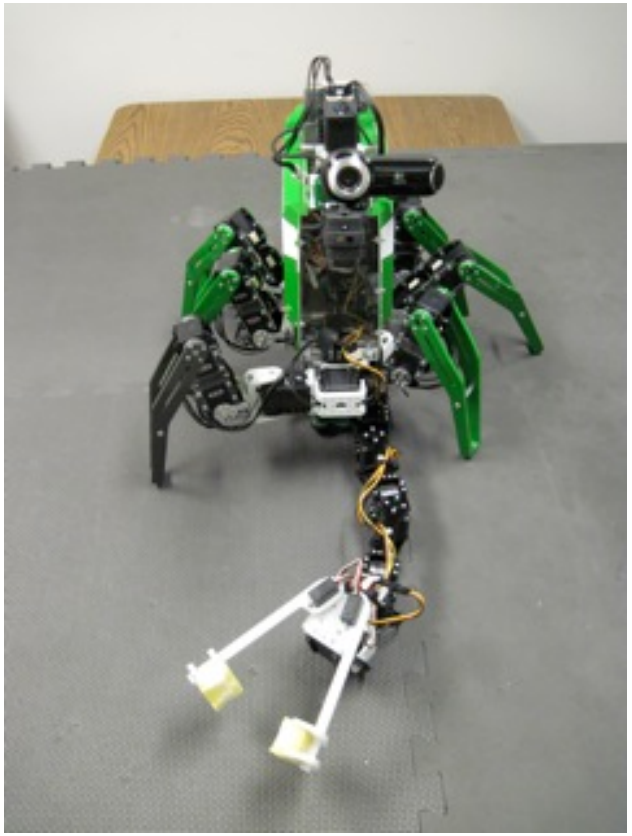
- Hexapod robot from CMU's Tekkotsu Lab
- Onboard computation
  - Gamma series
    - Pico-ITX (1GHz Processor)
  - Delta series
    - Advantech PCM-9361 (1.6GHz Intel Atom)
    - More powerful leg servos (Dynamixel RX-28)
- Ubuntu with Tekkotsu installation

The Chiara

GAMMA SERIES



DELTA SERIES



Tekkotsu

- Robotic software framework
  - Low-level concepts -> high-level primitives
  - The Crew
    - Lookout (Sensor package and readings)
    - MapBuilder (Vision)
    - Pilot (Motion)
    - Grasper (Manipulation)
  - Dual-coding vision system

Why can't it play chess?

- Planar world assumption
  - MapBuilder
    - Cannot handle occlusions
  - Grasper
    - Only 2D manipulation
- Chiara gripper still in development

Autonomous Chess

1. Perceiving the Board
   - Localizing the robot relative to the board
   - Recognizing board changes as opponent's moves
   - Precise piece localization for grasping
2. Choosing a Move (GNU Chess)
3. Executing the Move
   - Body positioning
   - Arm path planning
   - Grasping and releasing pieces

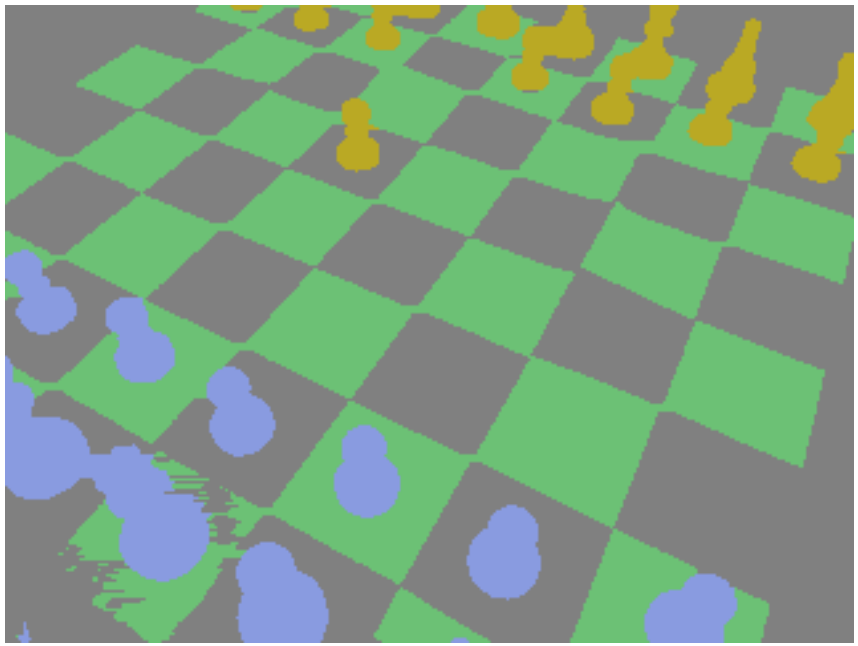
Autonomous Chess

1. Perceiving the Board
2. Choosing a Move
3. Executing the Move

Perceiving the Board

RGB IMAGE

COLOR SEGMENTED IMAGE



Finding the Board

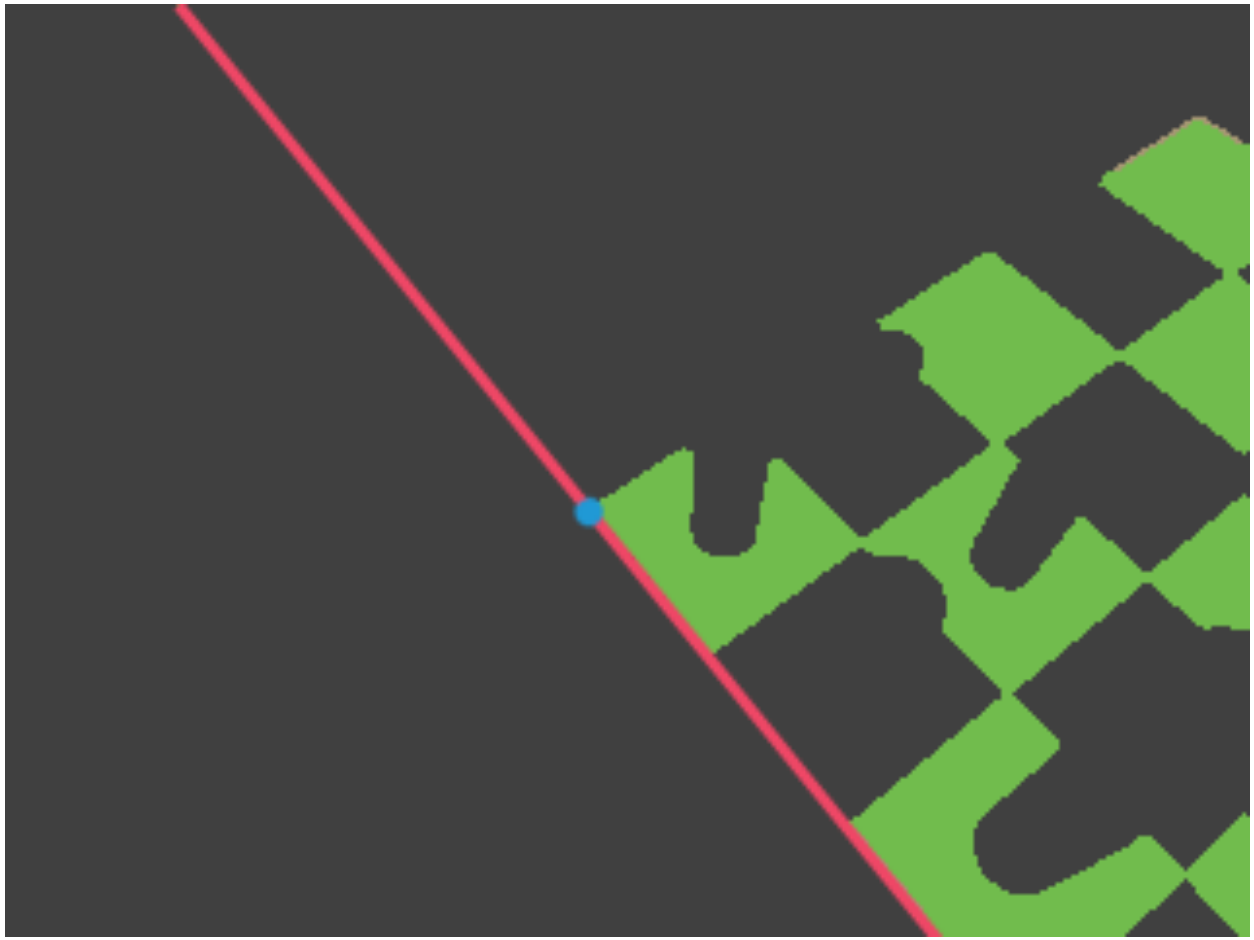
- Need to know where robot is relative to chessboard
- Extract lower-left corner (position) and bottom edge (orientation) of board

---

1. MapBuilder request to find colored things in local space
2. Shift gaze left until green pixels lie in right half plane of image, centered vertically
3. Extract features

Finding the Board

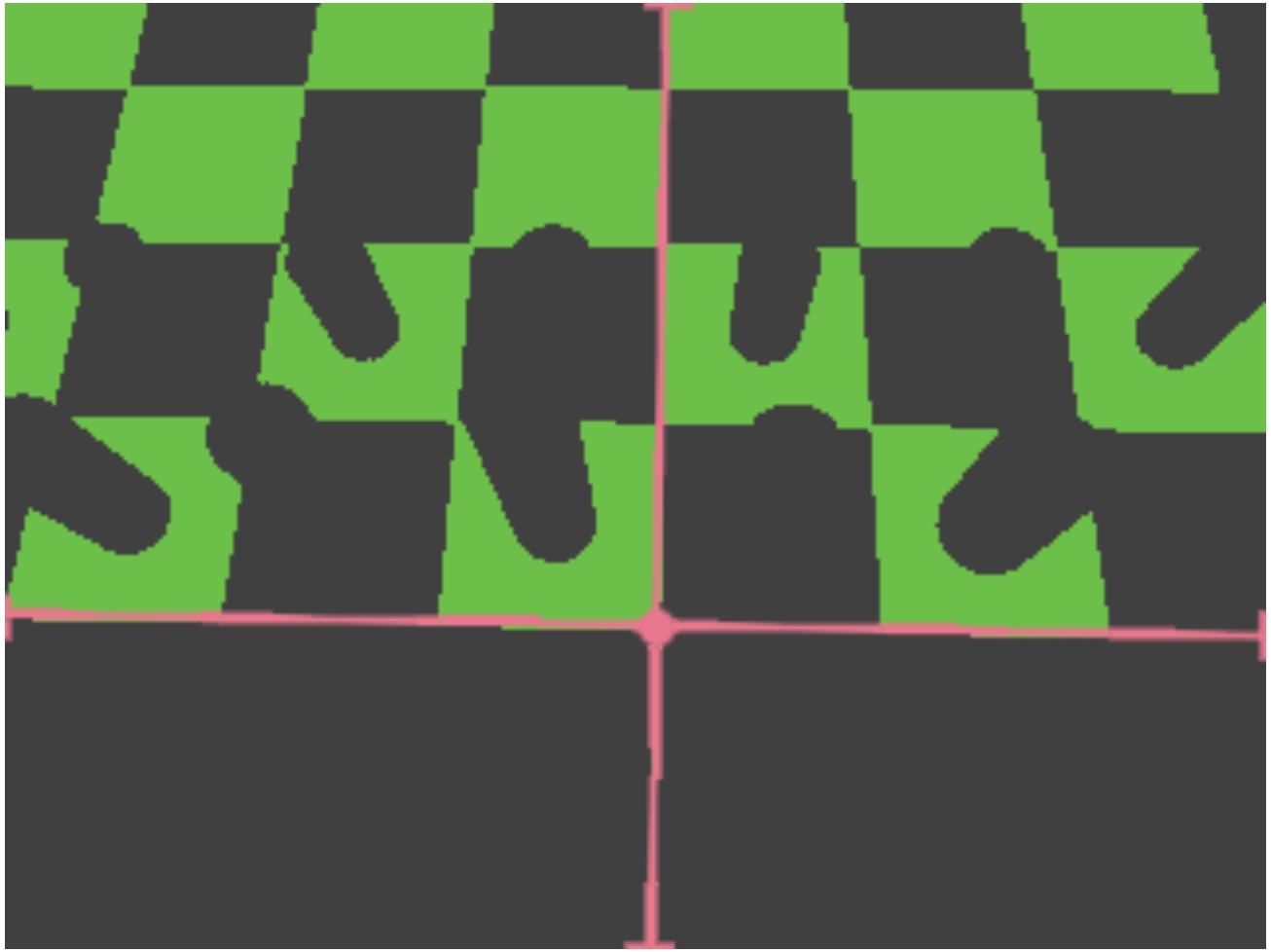
Extracting bottom line for orientation and blue dot for position



Re-localization

- Imperfect locomotion
  - Leg slippage on table surface
  - Inadvertent leg collisions
- Utilize board features immediately in front of the robot:
  - Find a known point and line on the board
  - Use green and white squares

Re-localization

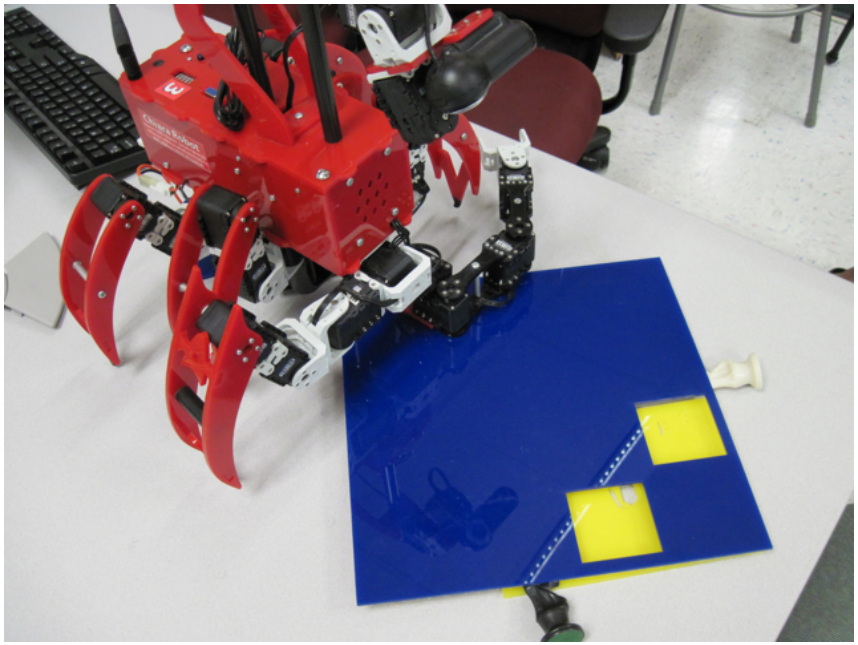


Camera Alignment

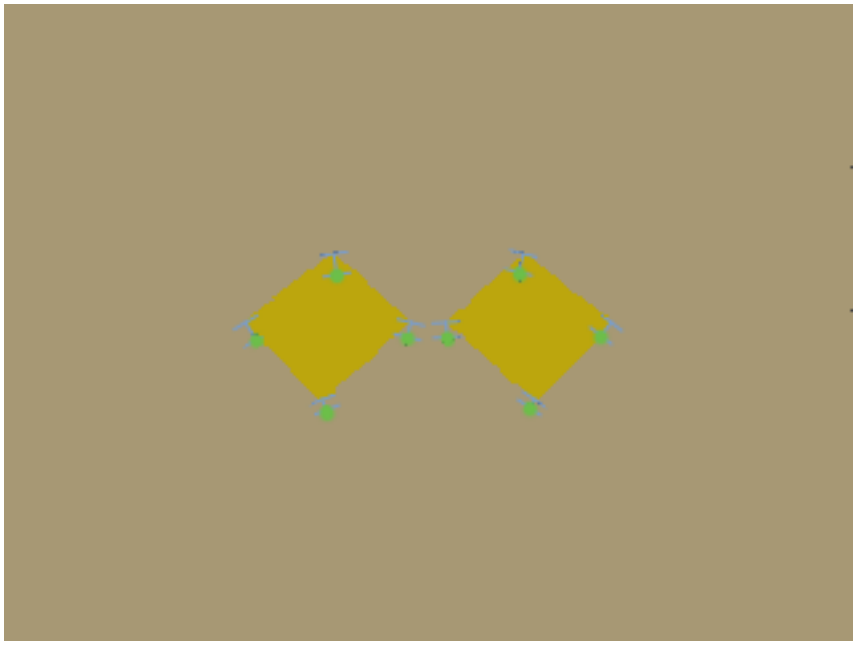
- Poor accuracy in camera projections
  - Physical location vs. ideal location
- Homography correction matrix
  - Correct for rotation, scale, and translation
  - Alignment stage
    - Use specialized rig
    - Build correspondences to derive the matrix
  - Utilization
    - Map perceived camera pixel coordinates to actual coordinates

Camera Alignment

RIG ON ROBOT



PERCEIVED RIG



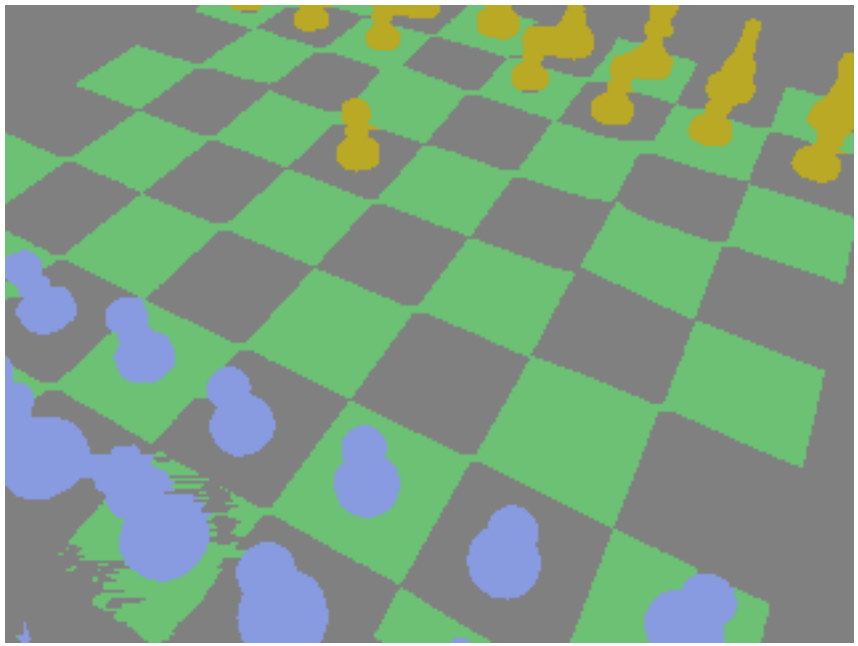
Recognizing opponent's move

- Determine square occupancy
  - Extracting chess pieces
  - Extracting board squares
- Combining multiple pictures
- Deducing move

Determine Square Occupancy

RGB IMAGE

SEGMENTED IMAGE



Determine Square Occupancy

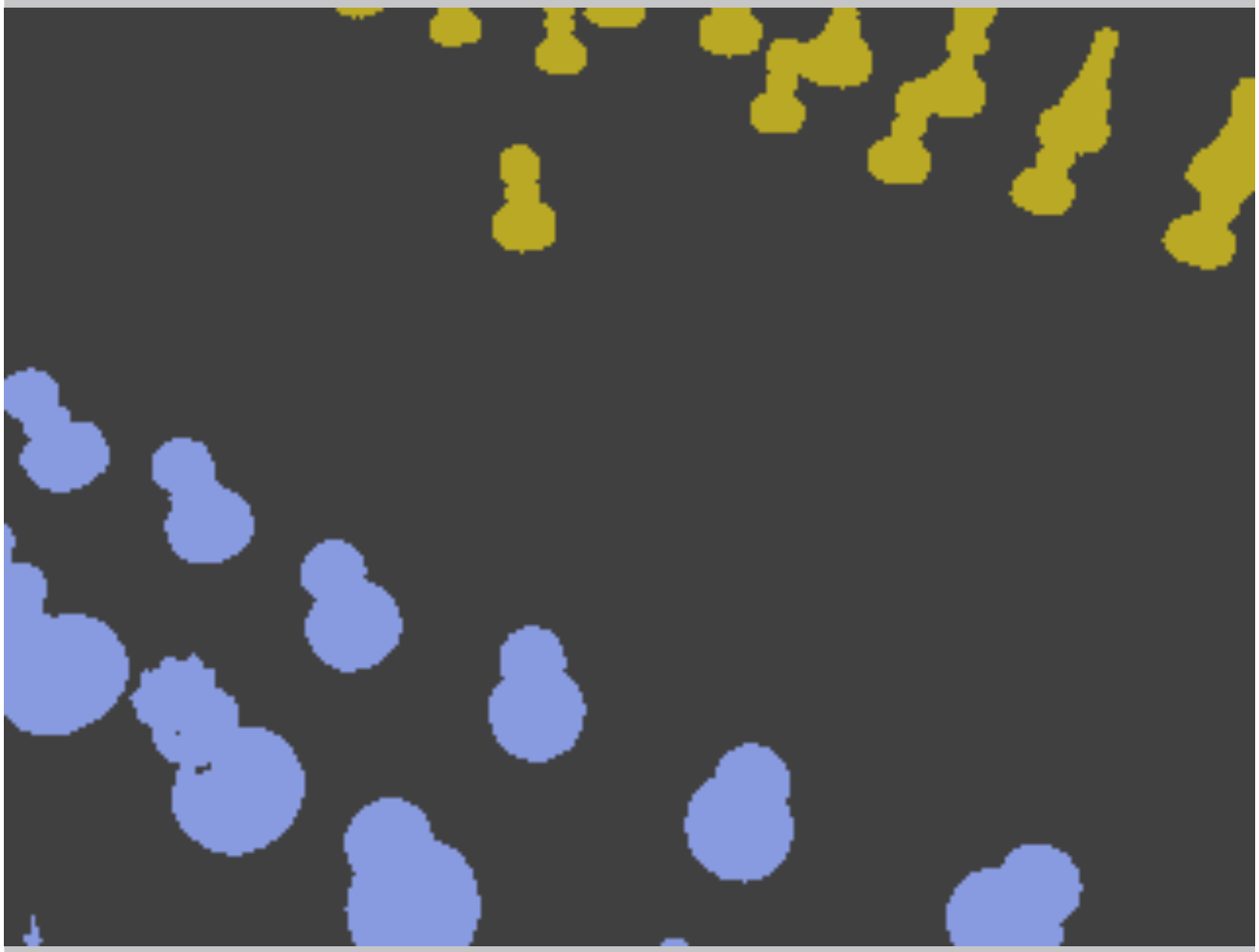
- Assumptions
  - Chessboard fills the majority of the camera frame
  - Board has green + white squares
  - Yellow and blue chess pieces
- Issues
  - Imperfect color segmentation
  - Occlusions
  - Partial board views

Extracting Chess Pieces

- Inspired by previous Tekkotsu work
- Bottom of pieces found with "under-pixels"
  - Pixels of a target color and whose southern neighbor pixel is not the same color
  - Ornamental features create noisy pixel set

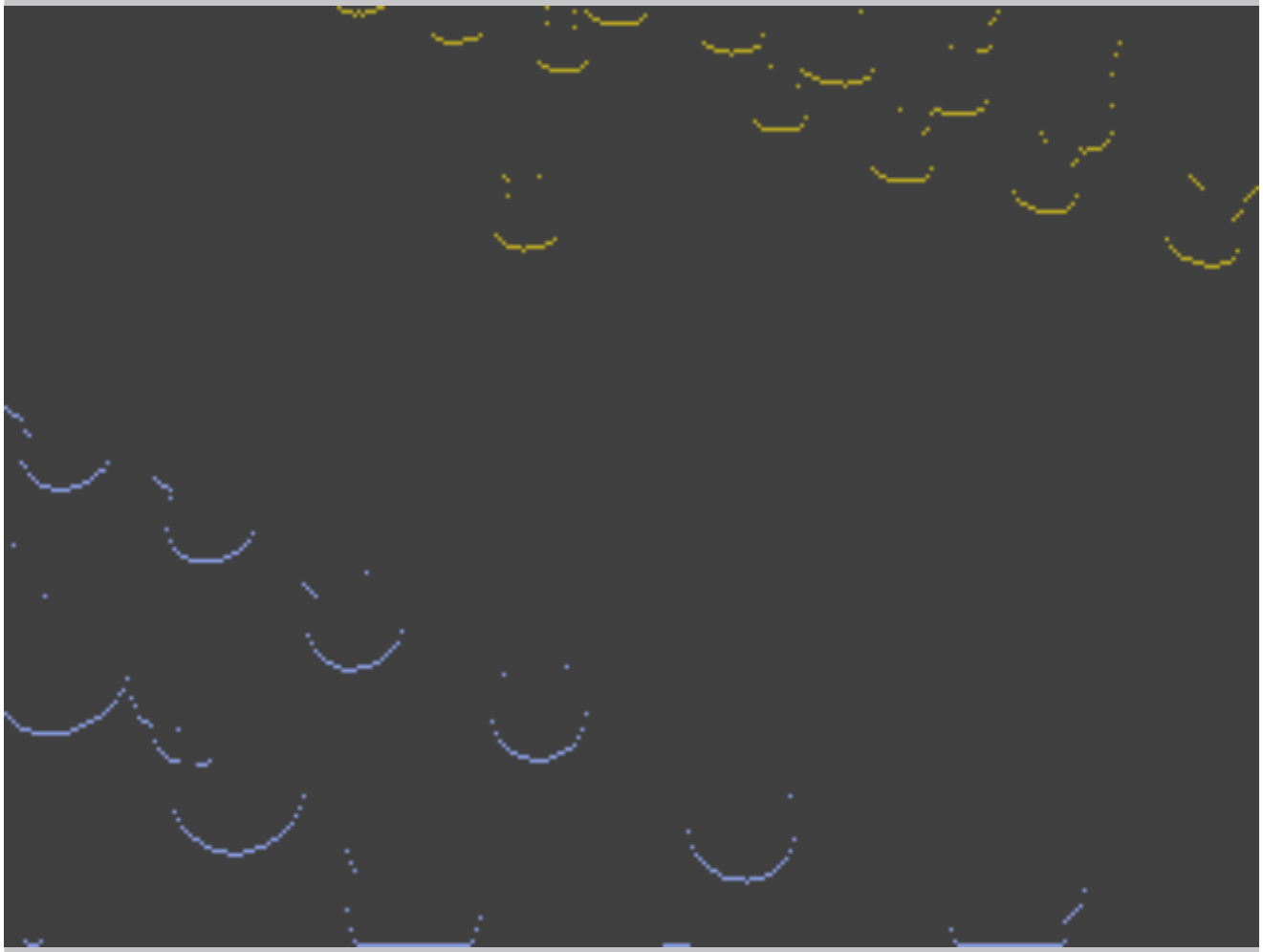
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



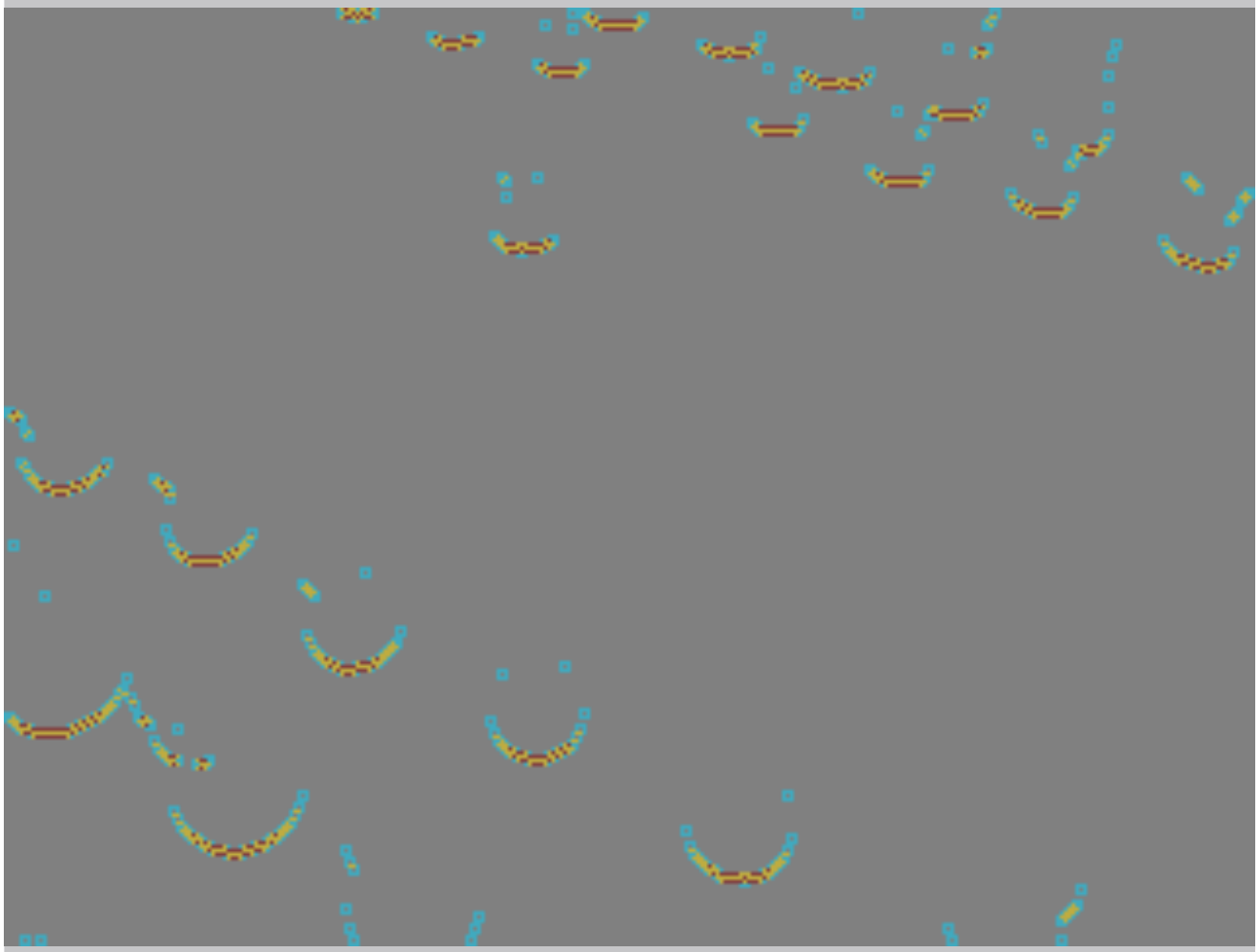
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



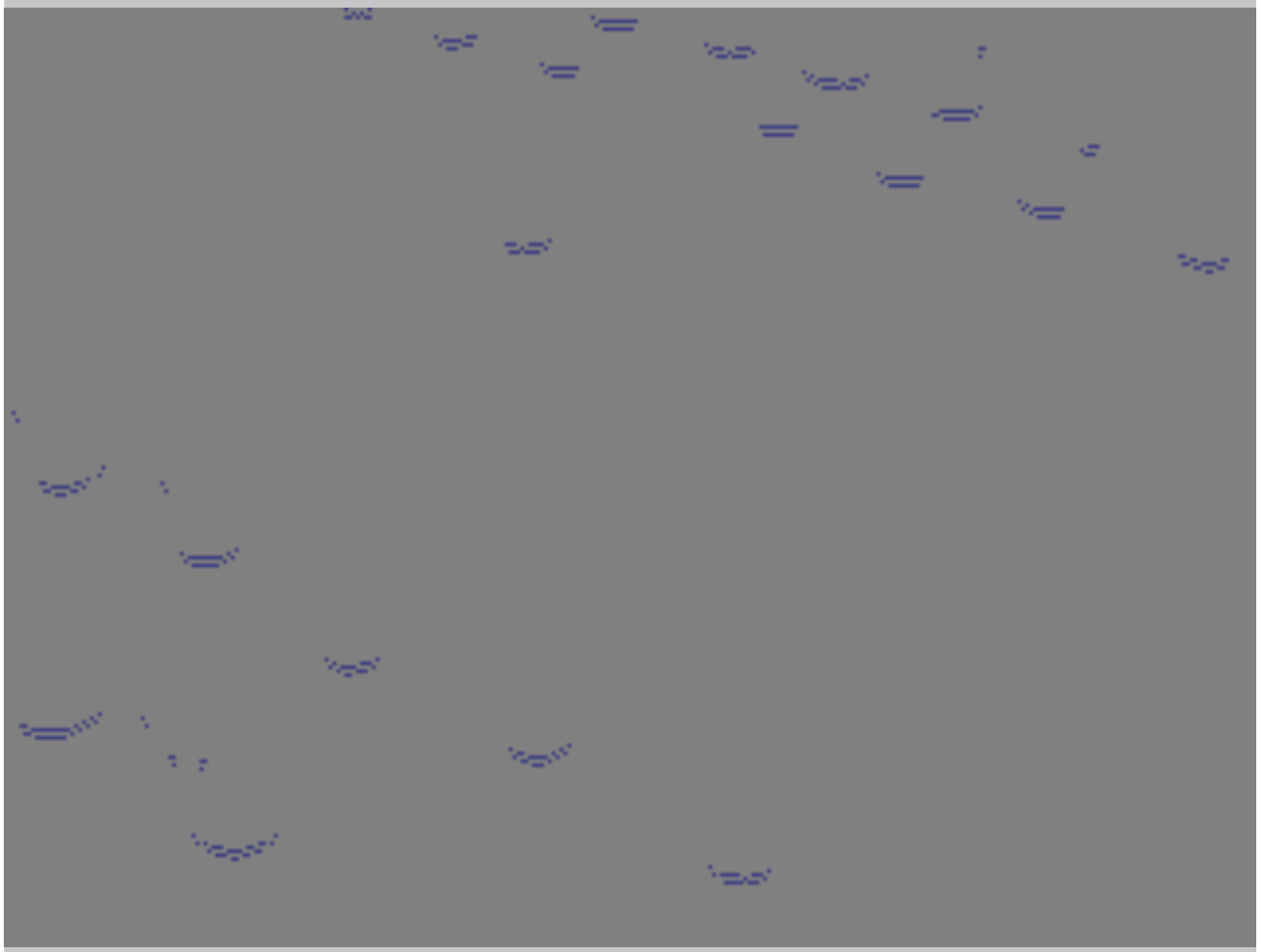
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



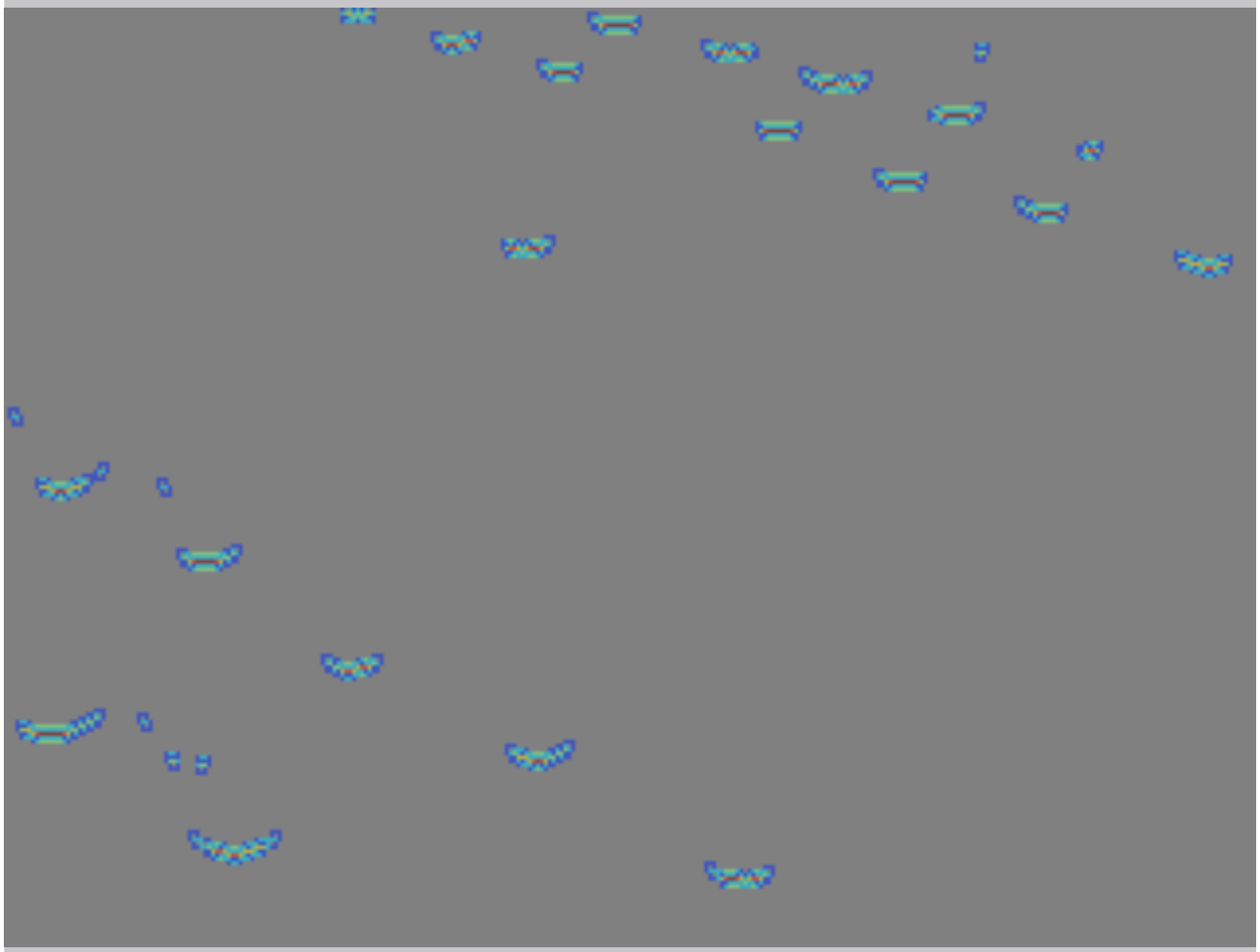
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



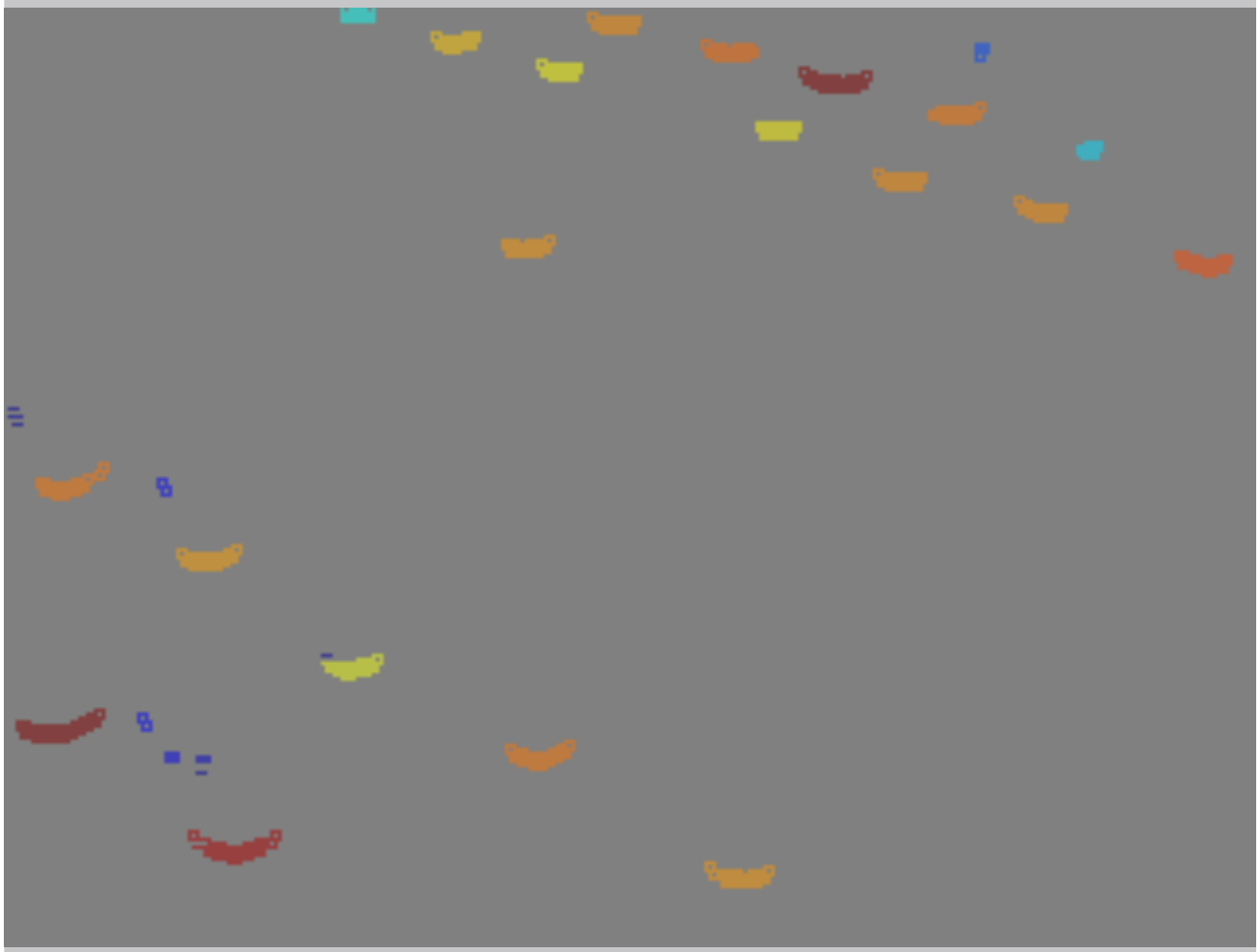
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



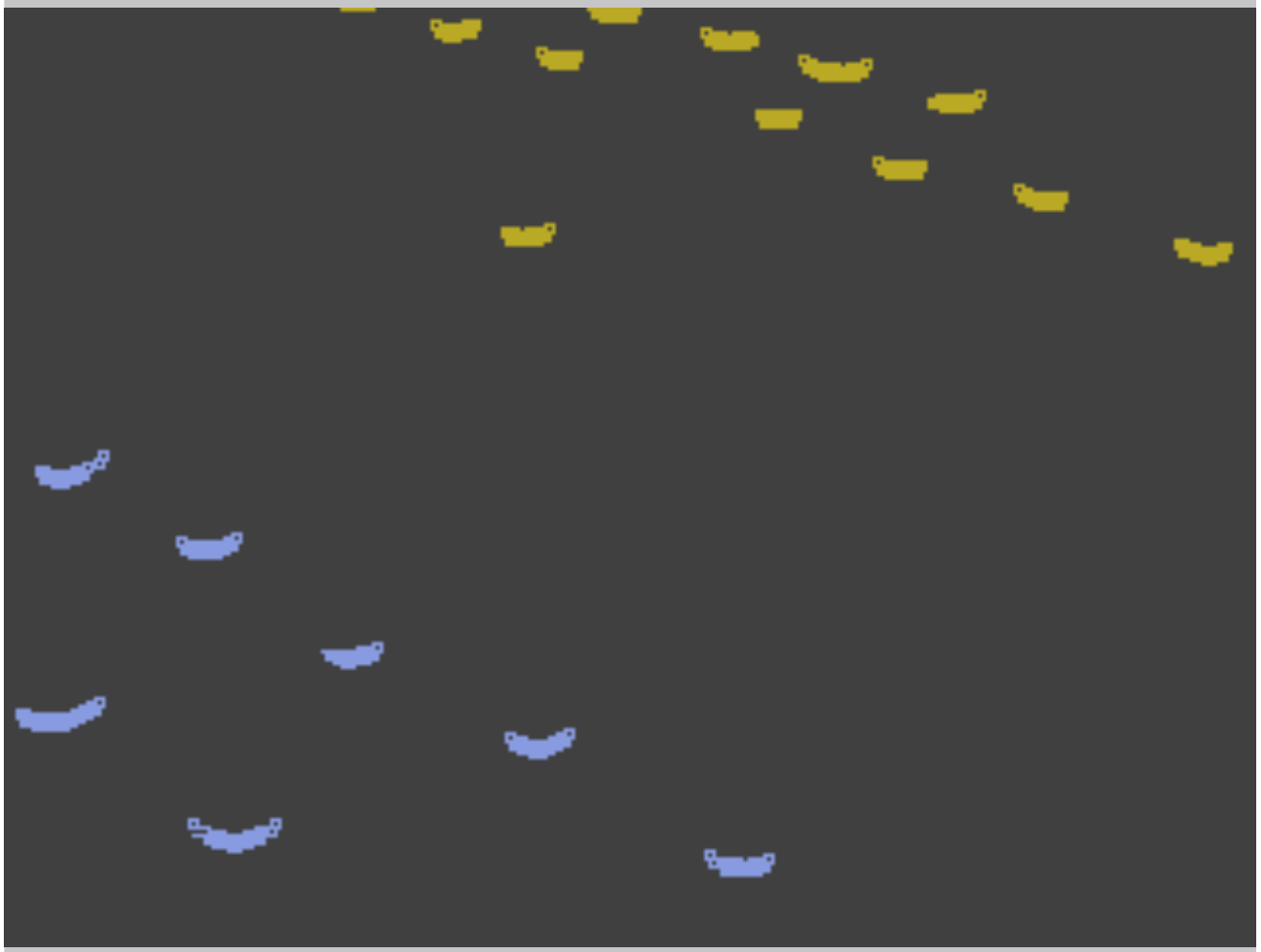
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



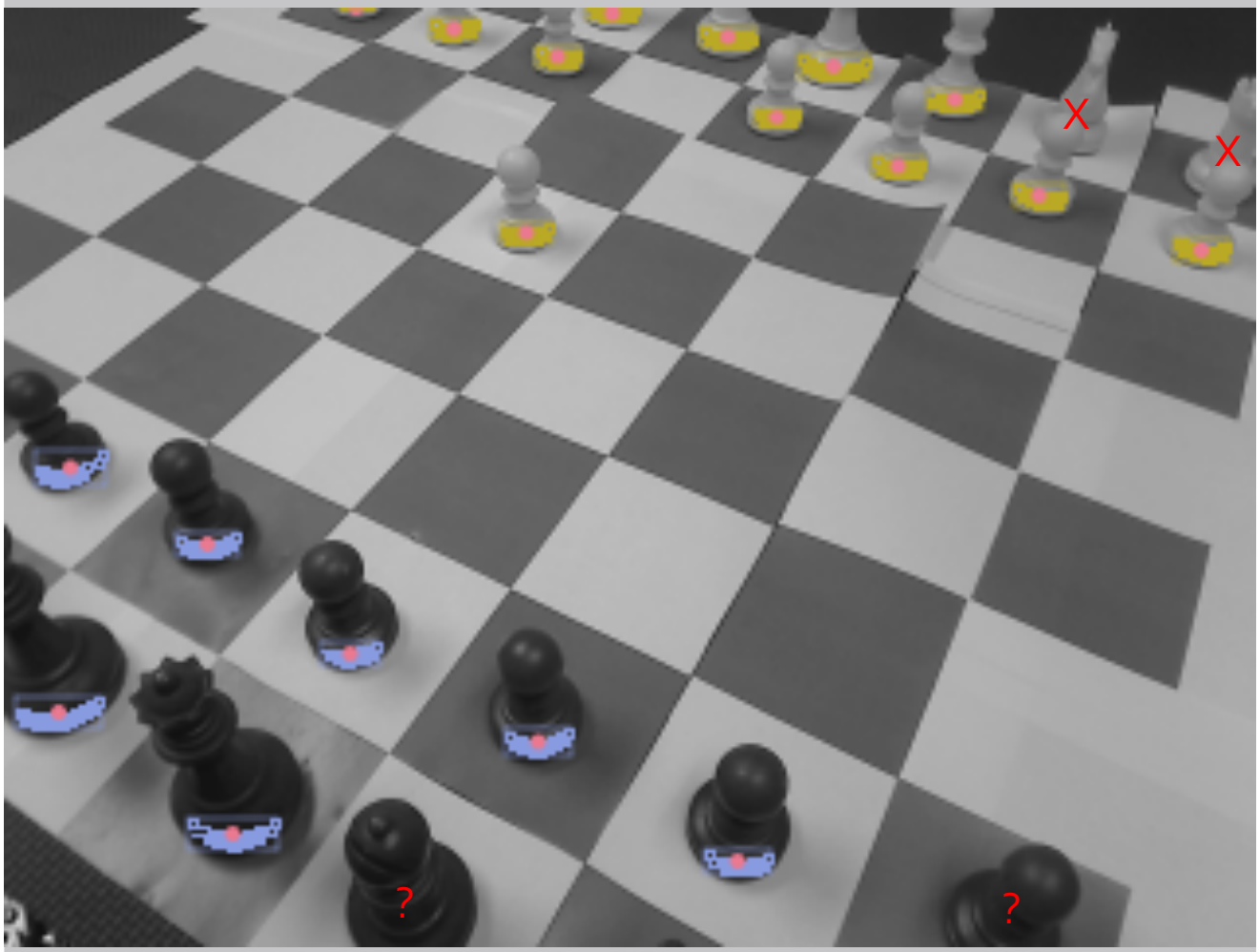
Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay



Extracting Chess Pieces

1. Segment piece colors
2. Calculate under-pixels
3. Bloat "smiles"
4. Select pixels with sufficient number of neighbor pixels
5. Re-bloat components
6. Calculate component areas
7. Piece locations
8. Final overlay

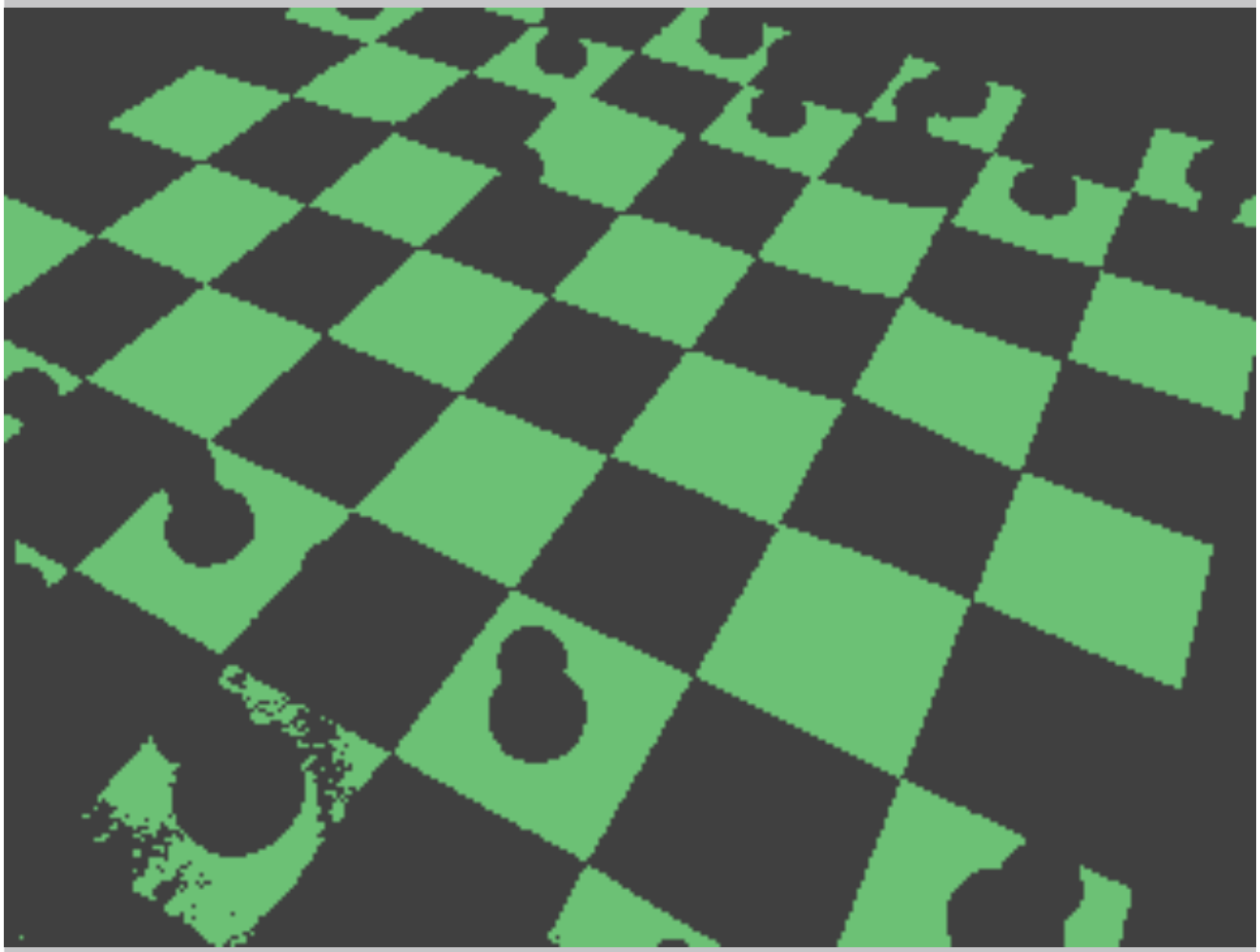


Extracting Board Squares

- Find green edge-pixels
- Apply Hough-transform
- Extract most probable lines
  - Filter and extrapolate probable lines into board lines

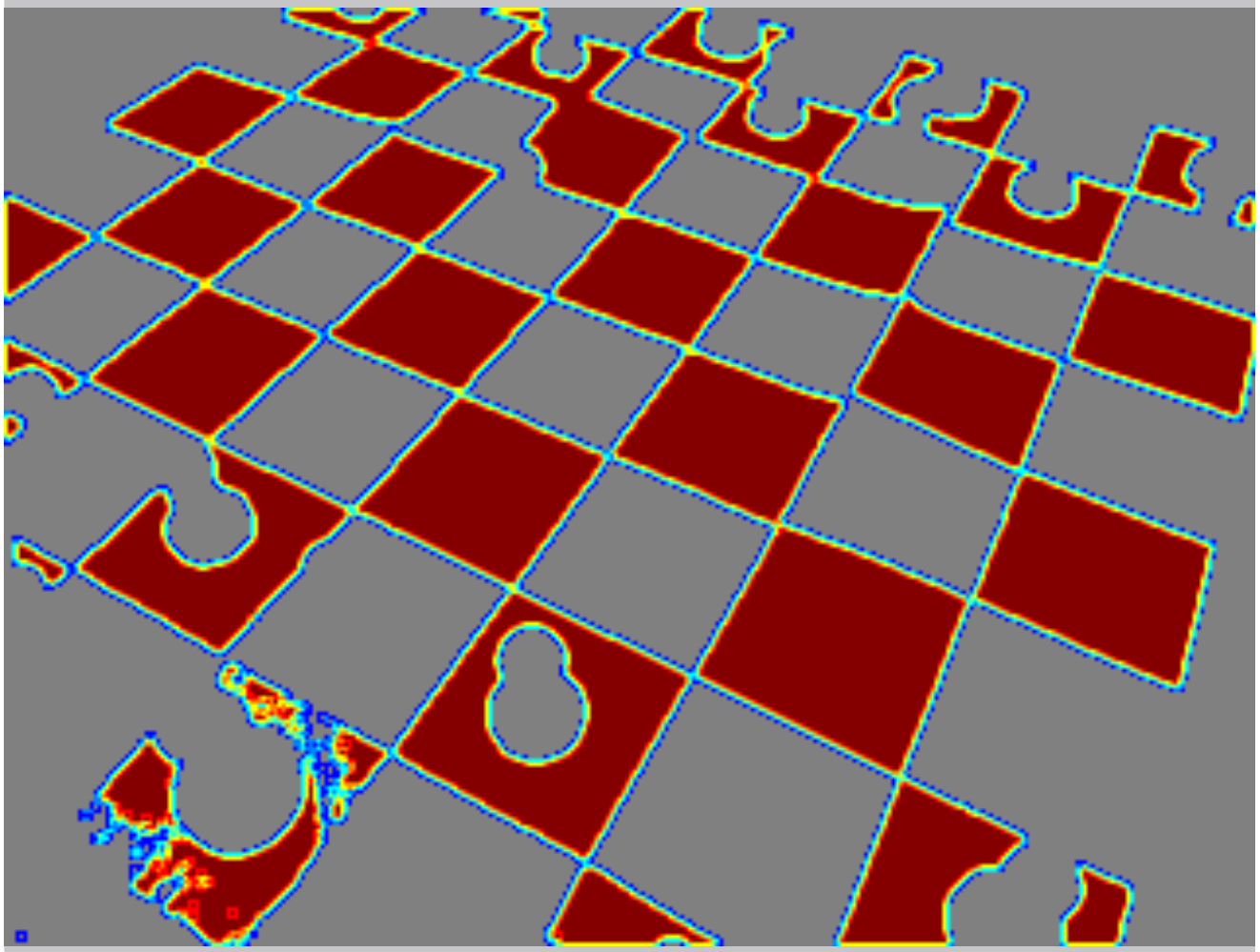
Find green edge-pixels

1. Segment all green pixels
2. Compute neighbor sums
3. Filter, leaving only appropriate pixels



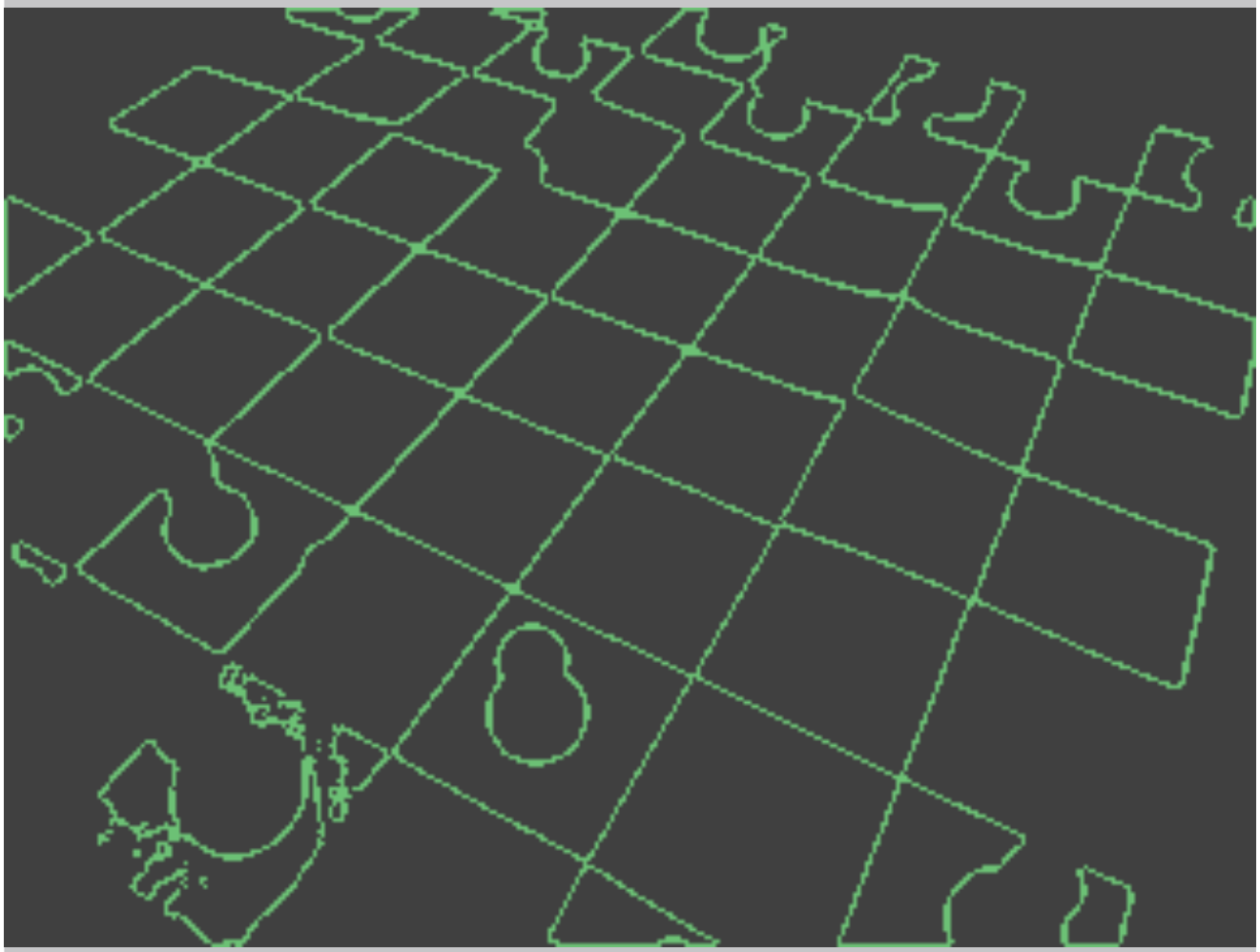
Find green edge-pixels

1. Segment all green pixels
2. Compute neighbor sums
3. Filter, leaving only appropriate pixels



Find green edge-pixels

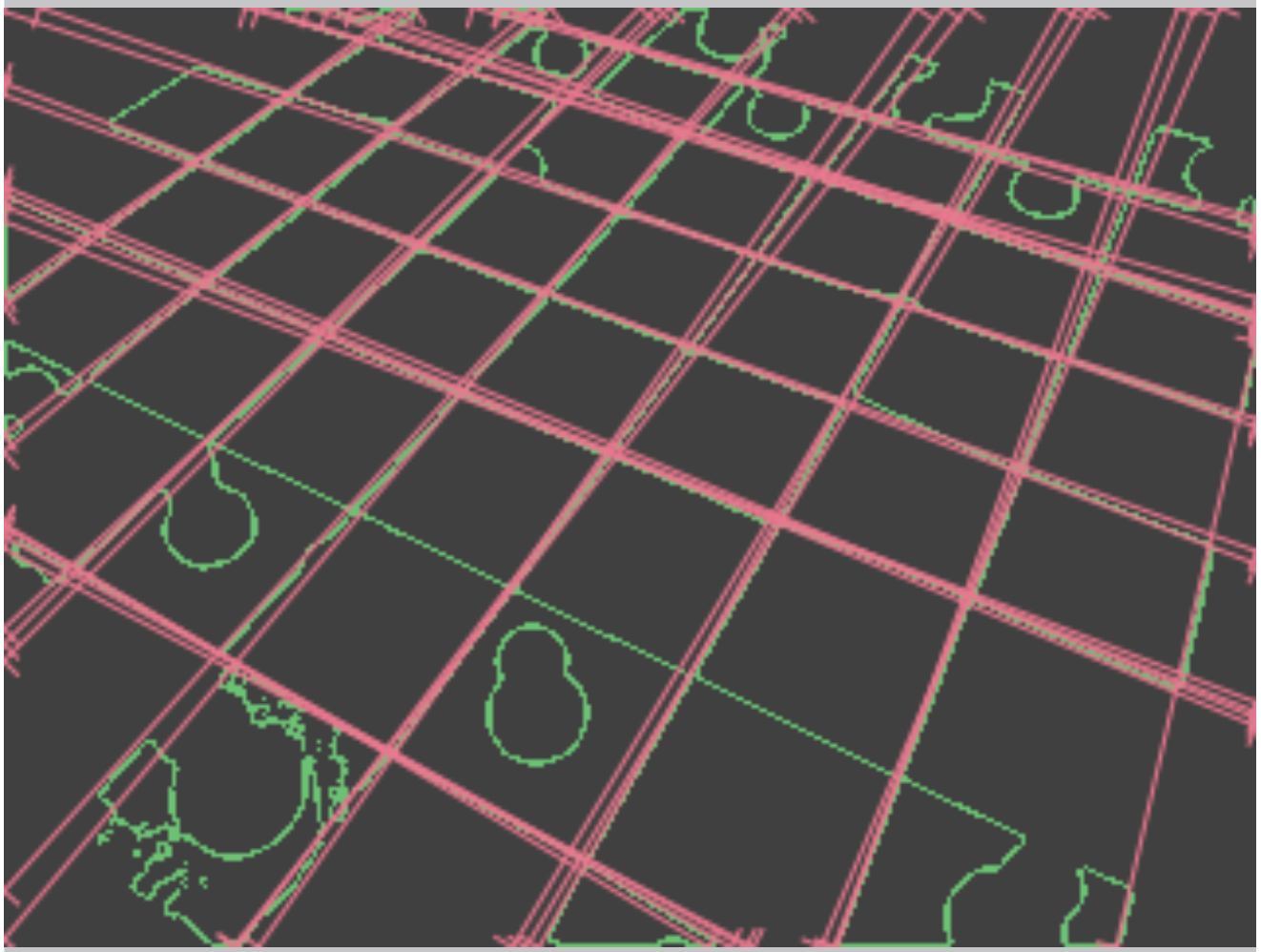
1. Segment all green pixels
2. Compute neighbor sums
3. Filter, leaving only appropriate pixels



Extracting Board Squares

Apply Hough-transform

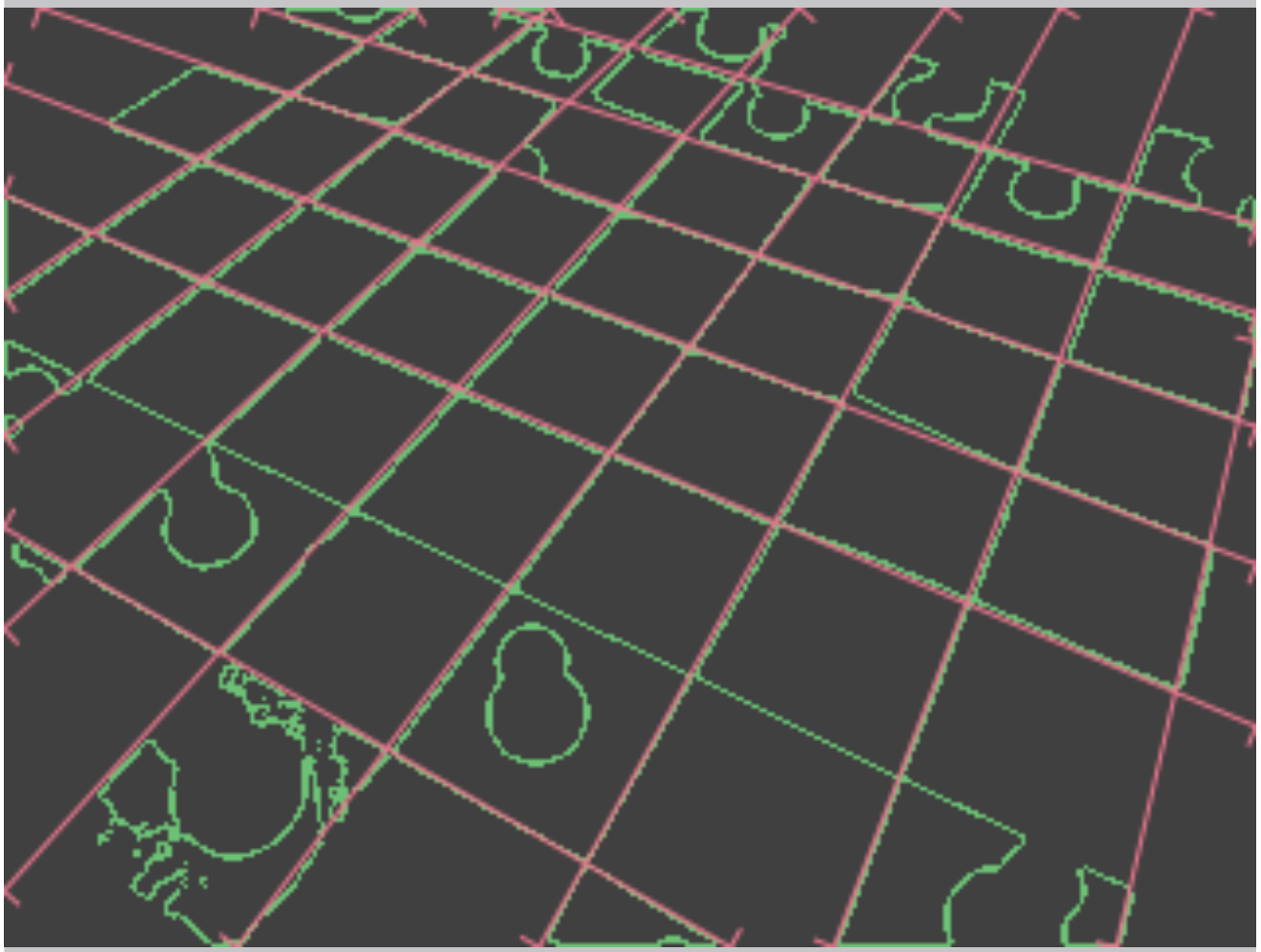
- Multiple extracted lines per board line
- Need to filter duplicate extracted lines per board line
- Need to extrapolate missed lines



Extracting Board Squares

Filter lines

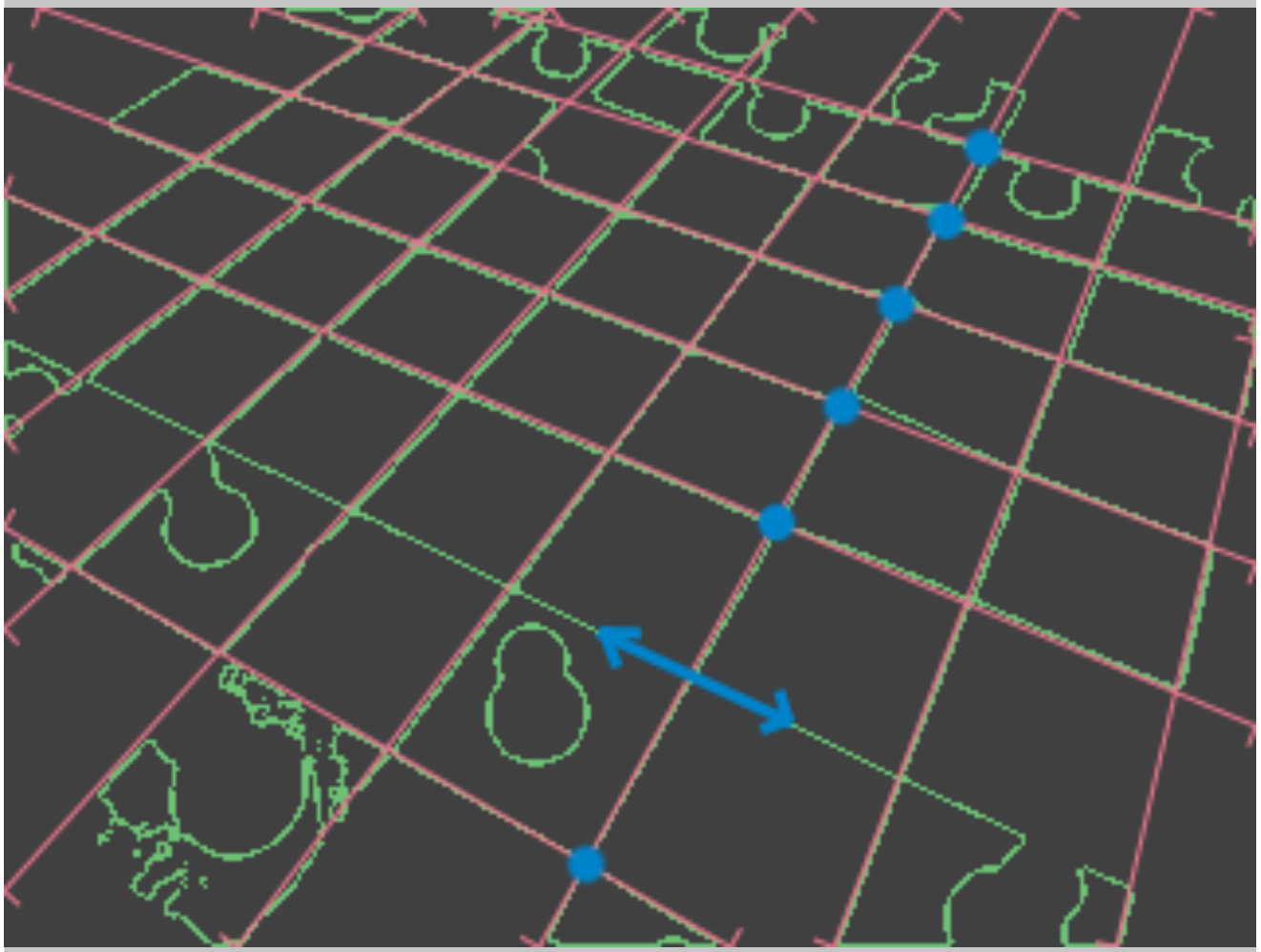
- Remove a line if
  - it intersected a more probable line of the same orientation within the camera frame
  - its position and orientation was too close to a more probable line



Extracting Board Squares

Extrapolate missed in-between lines

- Look at patterns in the positions of line intersections
- Find outlier differences
- Look in Hough-transform for missed line

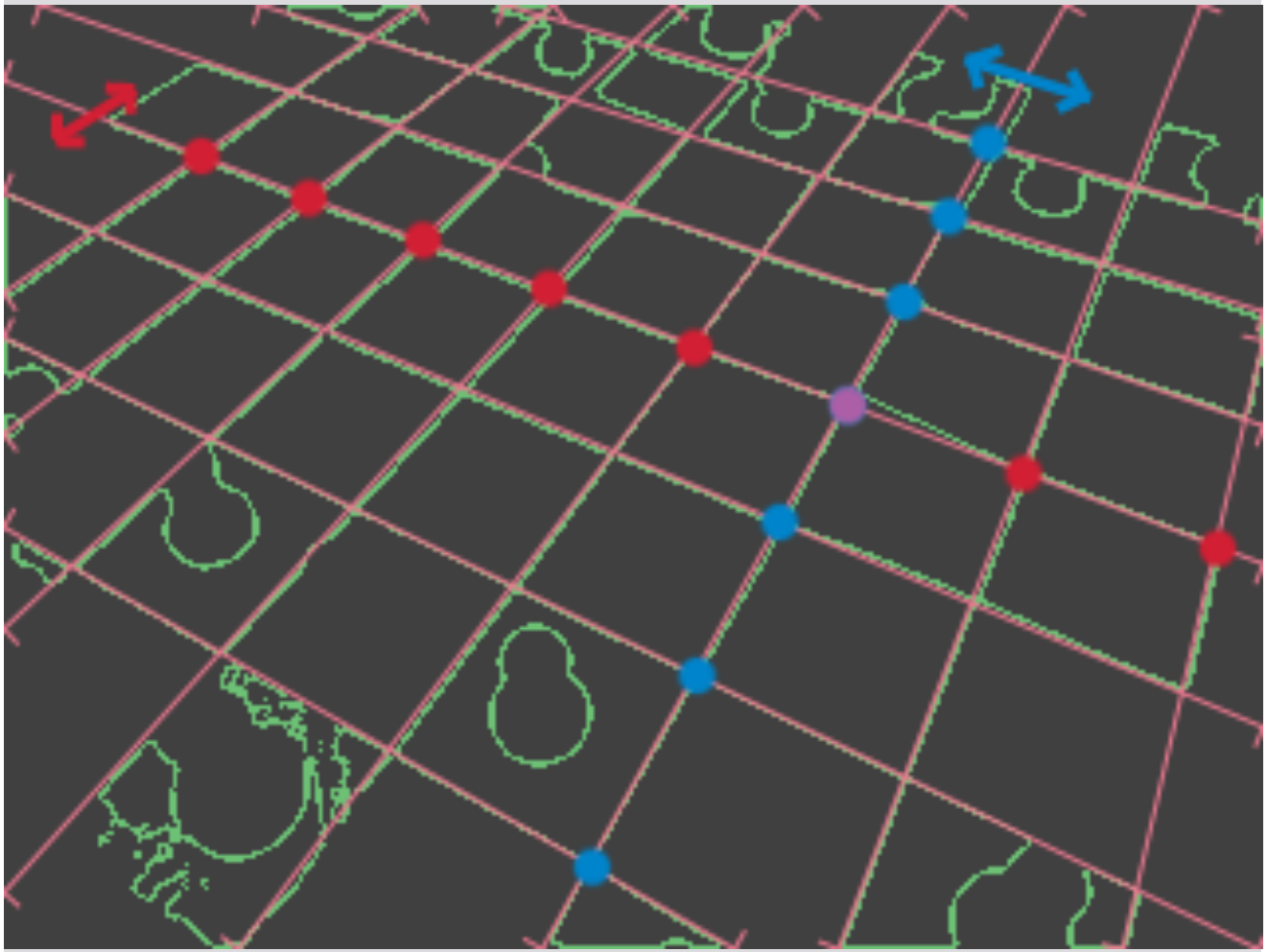


Extracting Board Squares

Extrapolating lines off ends

- Look at the changes in intersection differences off the ends

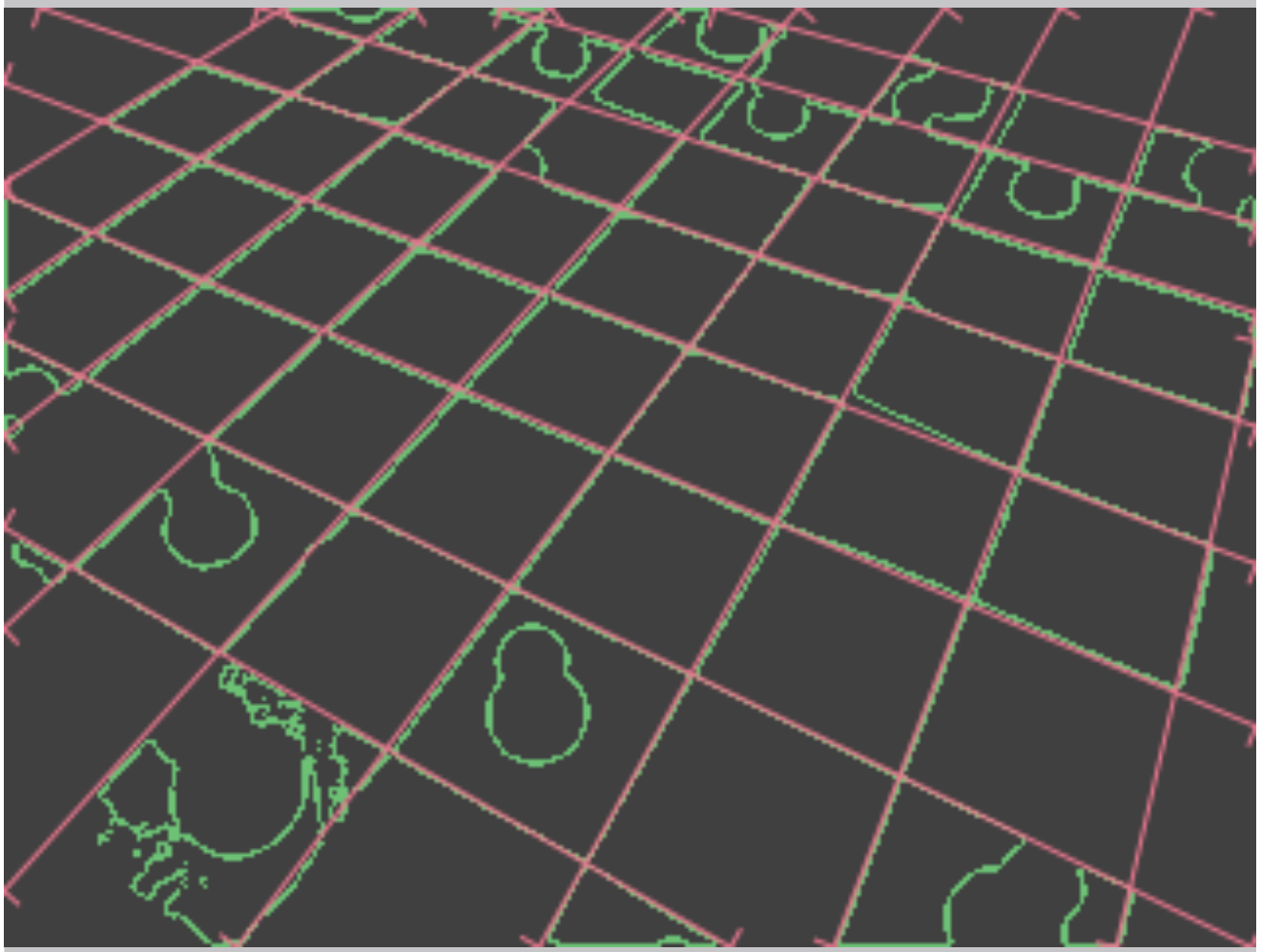
- Look in Hough-transform with the trend continuing for any missed lines



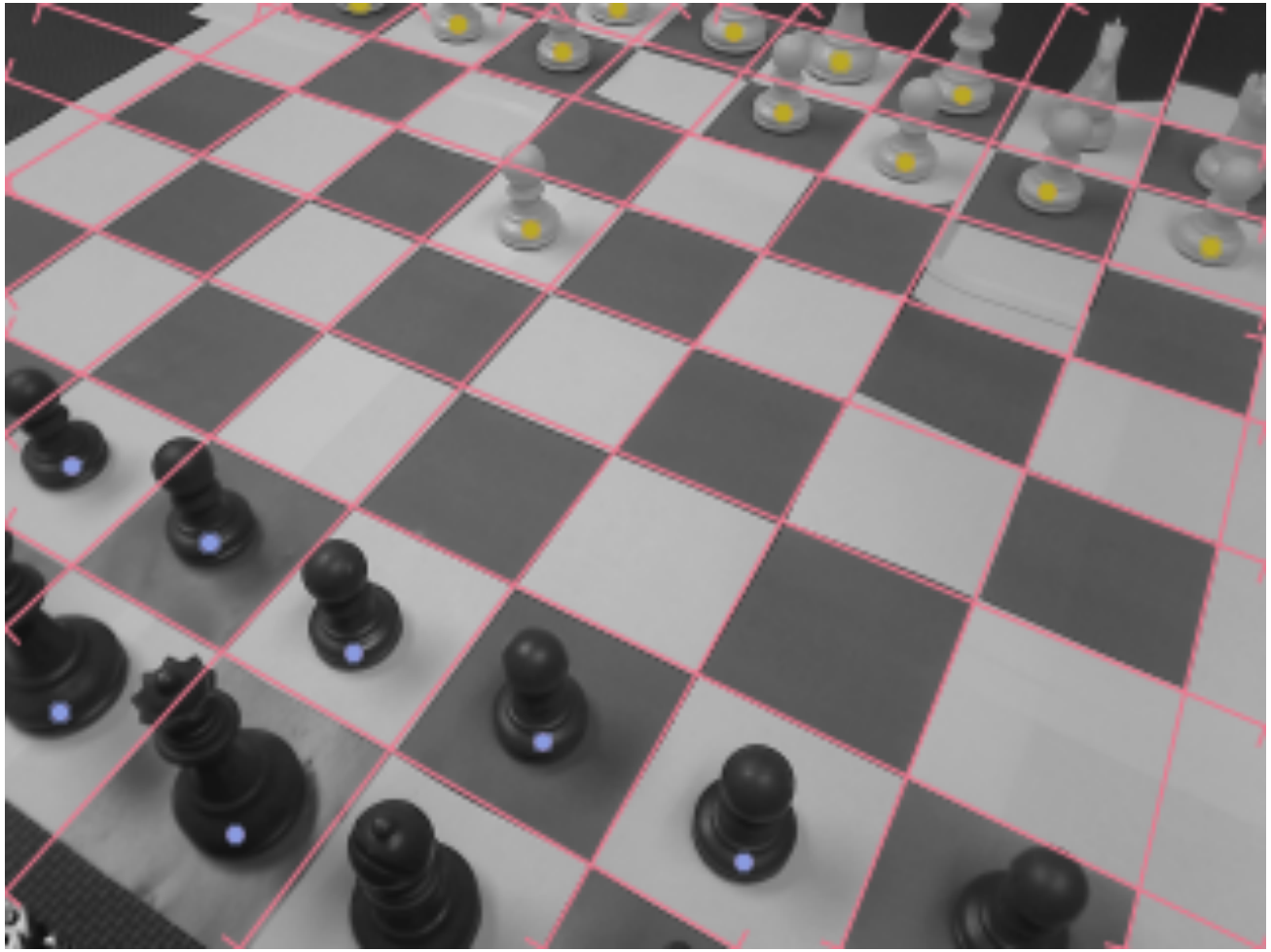
Extracting Board Squares

Accepted lines

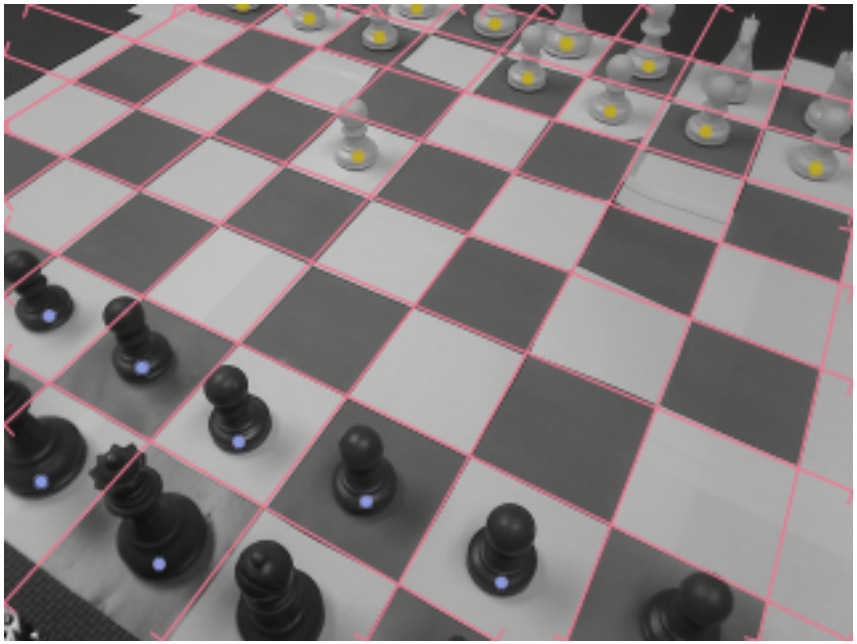
- Bottom line was attempted, but failed to be probably enough
- Right-most line wasn't necessary
  - Only 9 lines per dimension

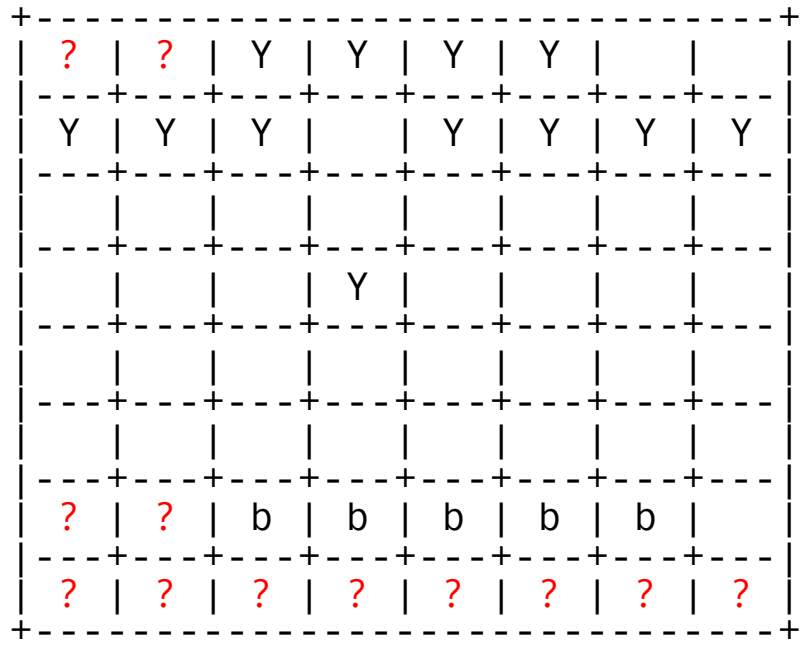


Determine Square Occupancy



Determine Square Occupancy





Gathering More Information

- Adjust vantage point
  - Shift position to change parallax between pieces
  - Shift gaze to view different parts of board
- Custom strategy for what vantage point to consider next

Combining Multiple Pictures

```
+-------------------------------+
| Y | Y | Y | Y | Y | Y | ? | ? |
|---+---+---+---+---+---+---+---|
| Y | Y | Y |   | Y | Y | ? | ? |
|---+---+---+---+---+---+---+---|
|   |   |   |   |   |   | ? | ? |
|---+---+---+---+---+---+---+---|
|   |   |   | Y |   |   | ? | ? |
|---+---+---+---+---+---+---+---|
|   |   |   |   |   |   | ? | ? |
|---+---+---+---+---+---+---+---|
|   |   |   |   |   |   | ? | ? |
|---+---+---+---+---+---+---+---|
| b | b | b | b | b | b | ? | ? |
|---+---+---+---+---+---+---+---|
| b | b | b |   | ? | ? | ? | ? |
+-------------------------------+
```

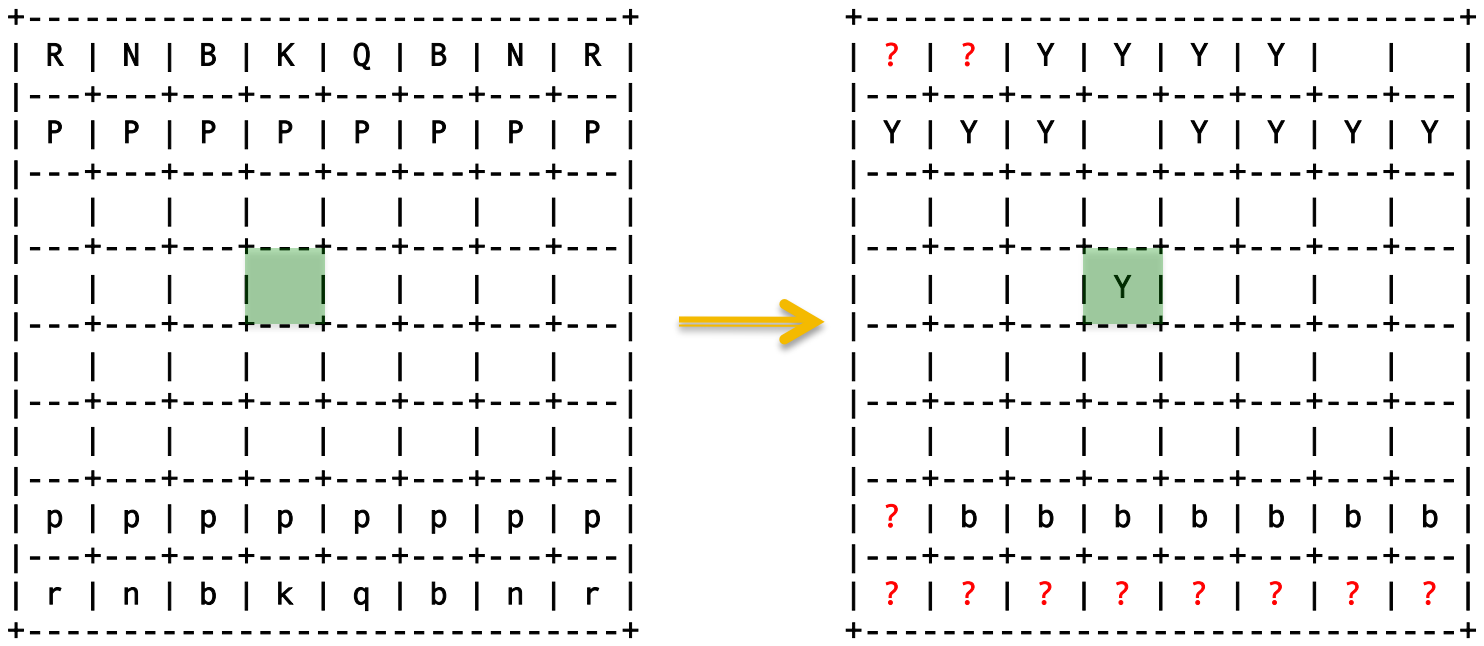
\+

```
+-------------------------------+
| ? | ? | ? | Y | Y | Y | Y | Y |
|---+---+---+---+---+---+---+---|
| ? | ? | ? |   | Y | Y | Y | Y |
|---+---+---+---+---+---+---+---|
| ? | ? | ? |   |   |   |   |   |
|---+---+---+---+---+---+---+---|
| ? | ? | ? | Y |   |   |   |   |
|---+---+---+---+---+---+---+---|
| ? | ? | ? |   |   |   |   |   |
|---+---+---+---+---+---+---+---|
| ? | ? | ? |   |   |   |   |   |
|---+---+---+---+---+---+---+---|
| ? | ? | ? | b | b | b | b | b |
|---+---+---+---+---+---+---+---|
| ? | ? | ? | ? | ? | ? | ? | ? |
+-------------------------------+
```

=

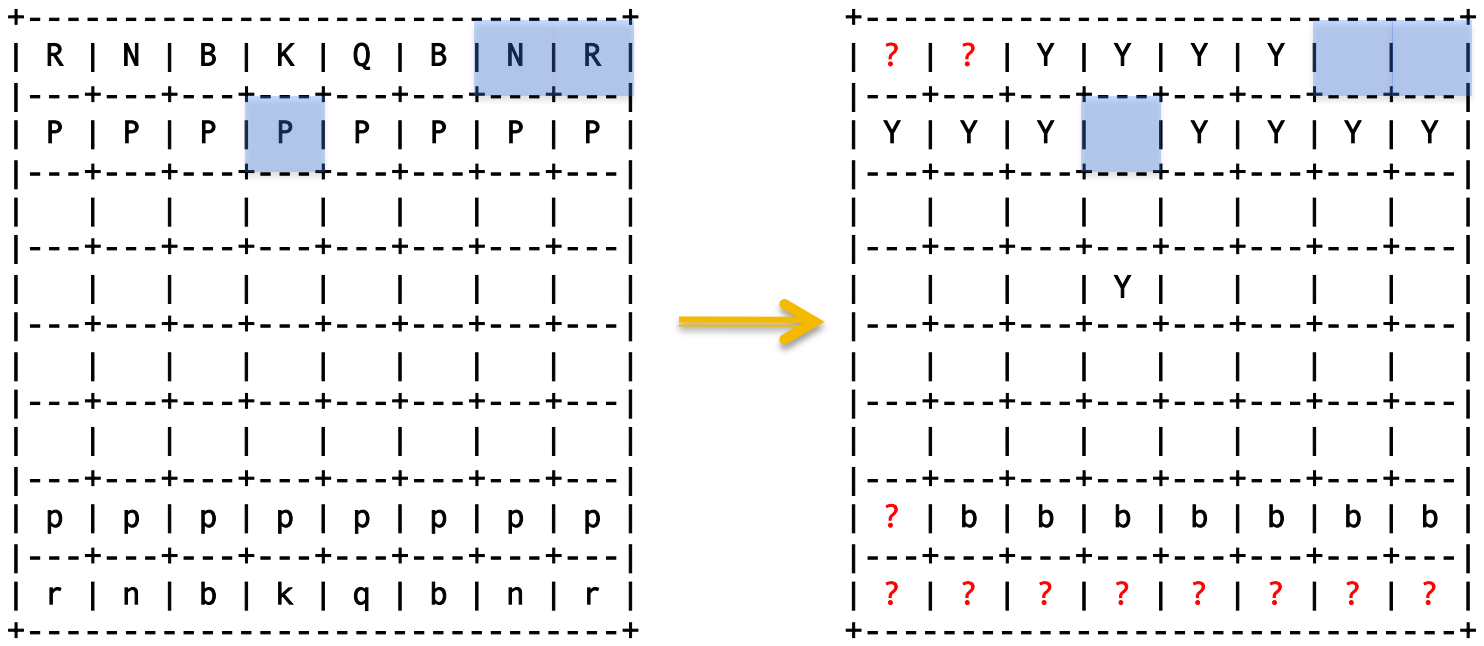
```
+-------------------------------+
| Y | Y | Y | Y | Y | Y | Y | Y |
|---+---+---+---+---+---+---+---|
| Y | Y | Y |   | Y | Y | Y | Y |
|---+---+---+---+---+---+---+---|
|   |   |   |   |   |   |   |   |
|---+---+---+---+---+---+---+---|
|   |   |   | Y |   |   |   |   |
|---+---+---+---+---+---+---+---|
|   |   |   |   |   |   |   |   |
|---+---+---+---+---+---+---+---|
|   |   |   |   |   |   |   |   |
|---+---+---+---+---+---+---+---|
| b | b | b | b | b | b | b | b |
|---+---+---+---+---+---+---+---|
| b | b | b |   | ? | ? | ? | ? |
+-------------------------------+
```

Deducing Move



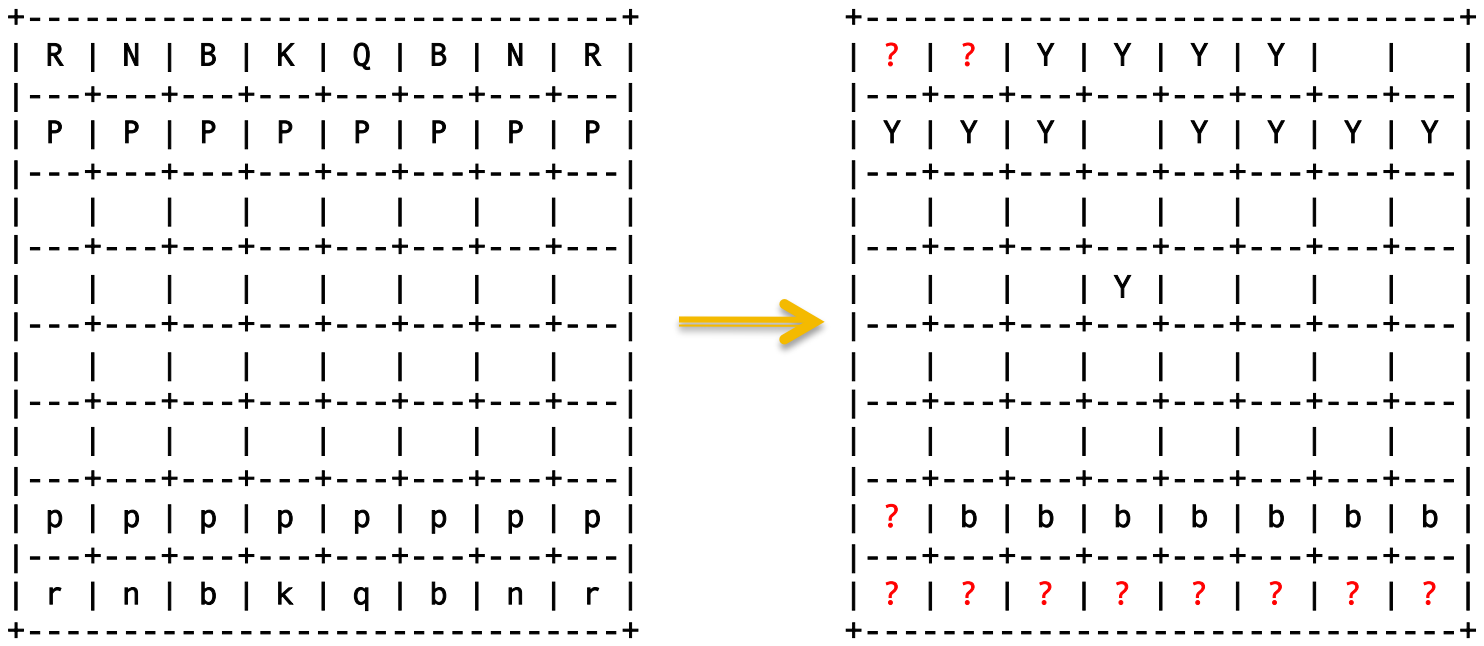
- Check occupancy differences between boards
  - Occupied squares
  - Vacated squares
  - Captured squares

Deducing Move



- Check occupancy differences between boards
  - Occupied squares
  - Vacated squares
  - Captured squares

Deducing Move



- Check occupancy differences between boards
  - Occupied squares
  - Vacated squares
  - Captured squares

Deducing Move

INVALID STATES

- No vacated squares
- No occupied or taken squares
- More than one occupied or taken square

X

VALID STATES

- Single occupied or taken square

OK

Castling is a special case

Deducing Move

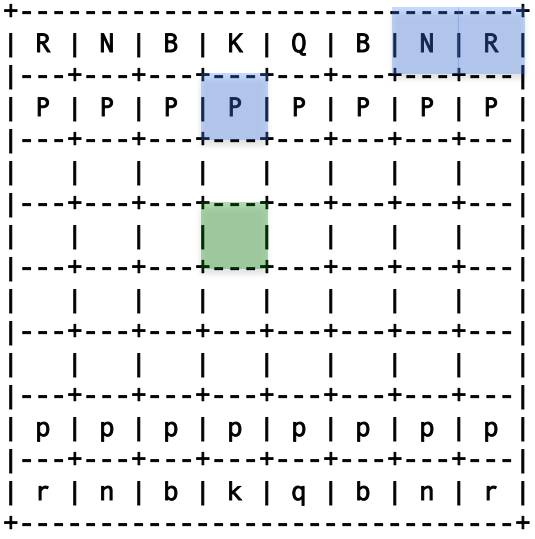
What about multiple vacated squares?

- Caused by occlusions
- Test each move from a vacated or unknown square to the occupied square
  - Accept if a single legal move was possible
  - Else, get more information from board
- Cuts down on total required images

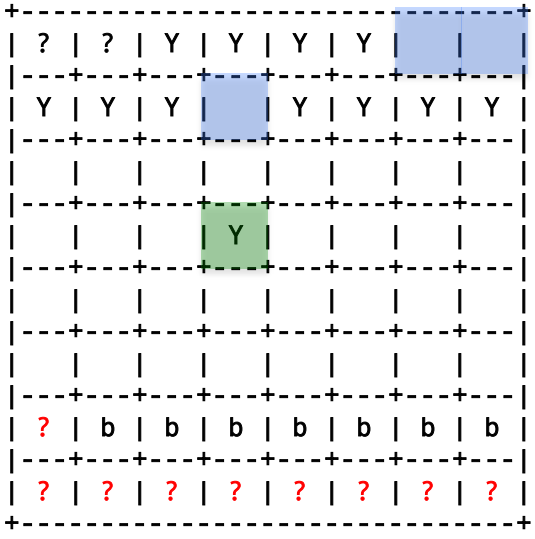
Deducing Move

- Update board information appropriately

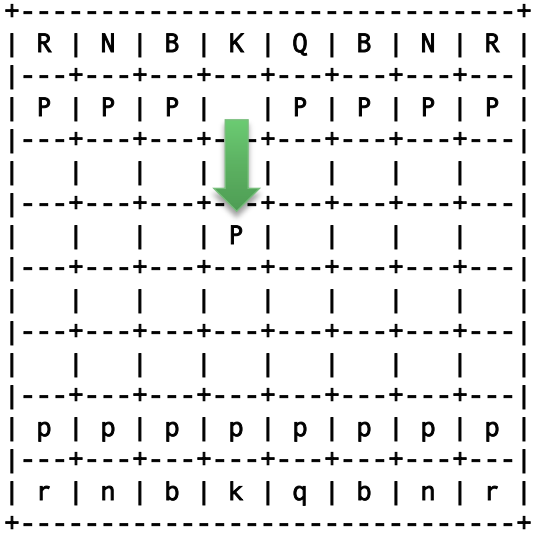
Previous board state



Partial perception



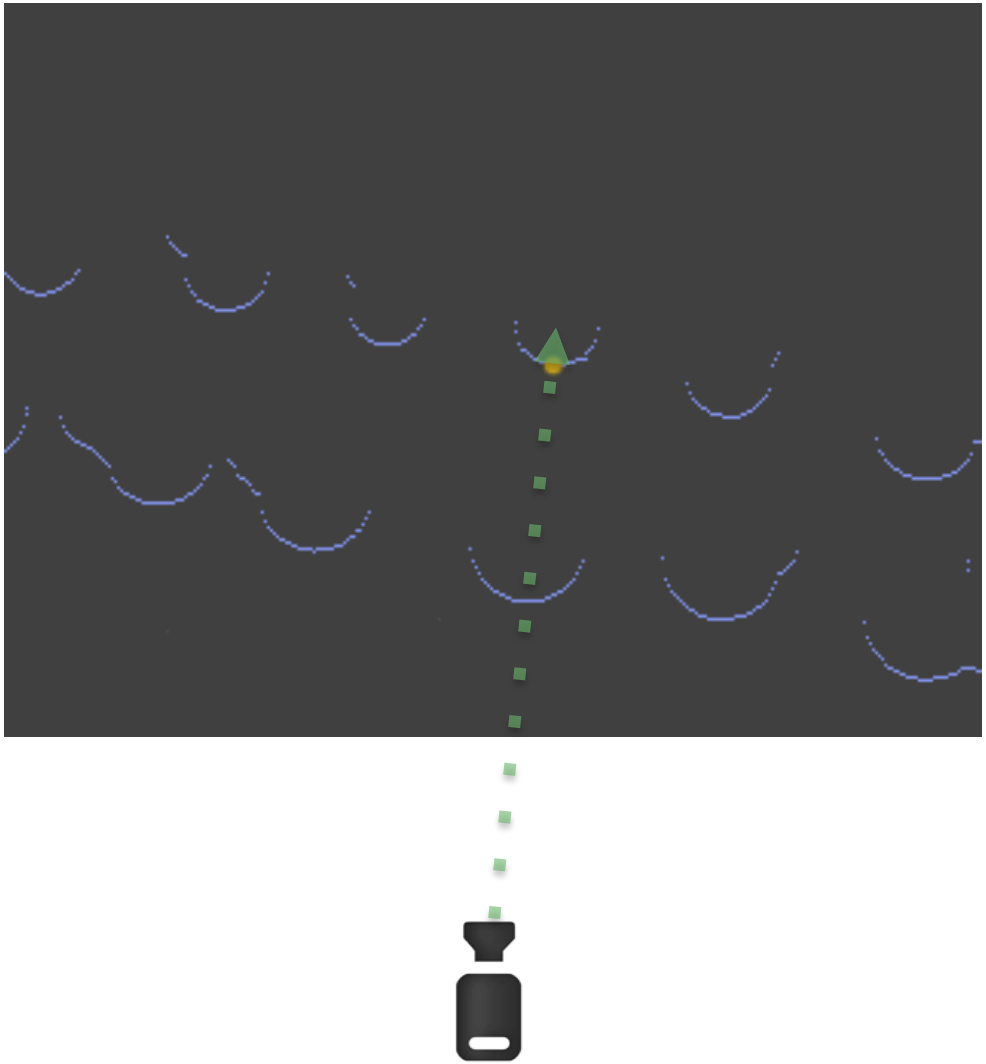
Inferred current state



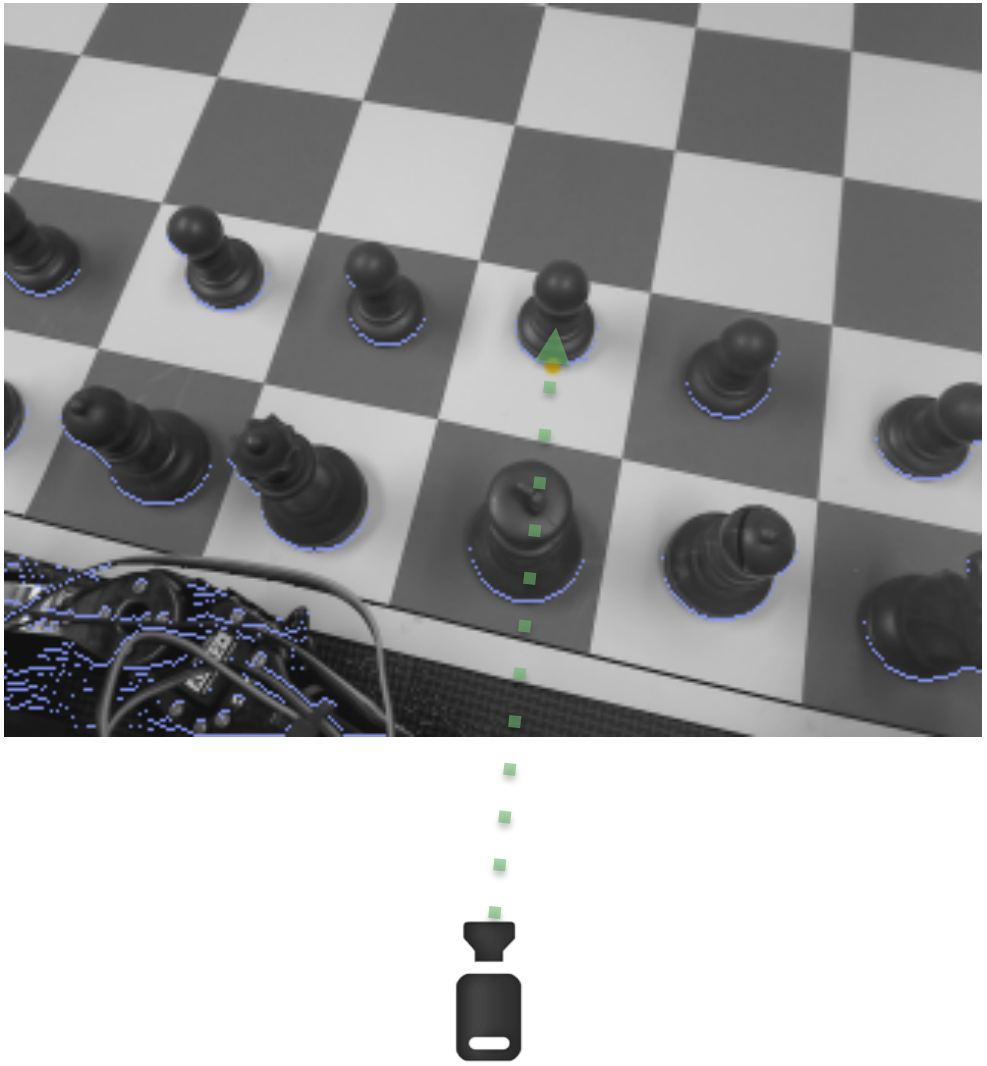
Visual localization of chess pieces

- Detect errors in pieces' actual locations vs. anticipated locations
- Extract pieces same way as determining the opponent's move
- Project bottom of piece to local space
- Add offset away from the camera by the radius of the piece

Projecting Piece Centers



Projecting Piece Centers



Autonomous Chess

1. Perceiving the Board
2. Choosing a Move
3. Executing the Move

Choosing a Chess Move

- GNU Chess
  - FOSS Chess Engine
  - Used for
    - Computing a move
    - Testing move legality
    - Undoing previous move
    - Updating internal board representation

Autonomous Chess

1. Perceiving the Board
2. Choosing a Chess Move
3. Executing the Move

Executing the Move

- Motion planning
- Grasping
  - Gripper design
  - Extending Grasper primitives
- Interleaving grasping and moving

Motion Planning

- Minimize need for body re-positioning
  - Expensive to get into a new position
  - Should have as many manipulation target locations within arm's reach as possible
- Use world space sketches to find satisfactory solutions

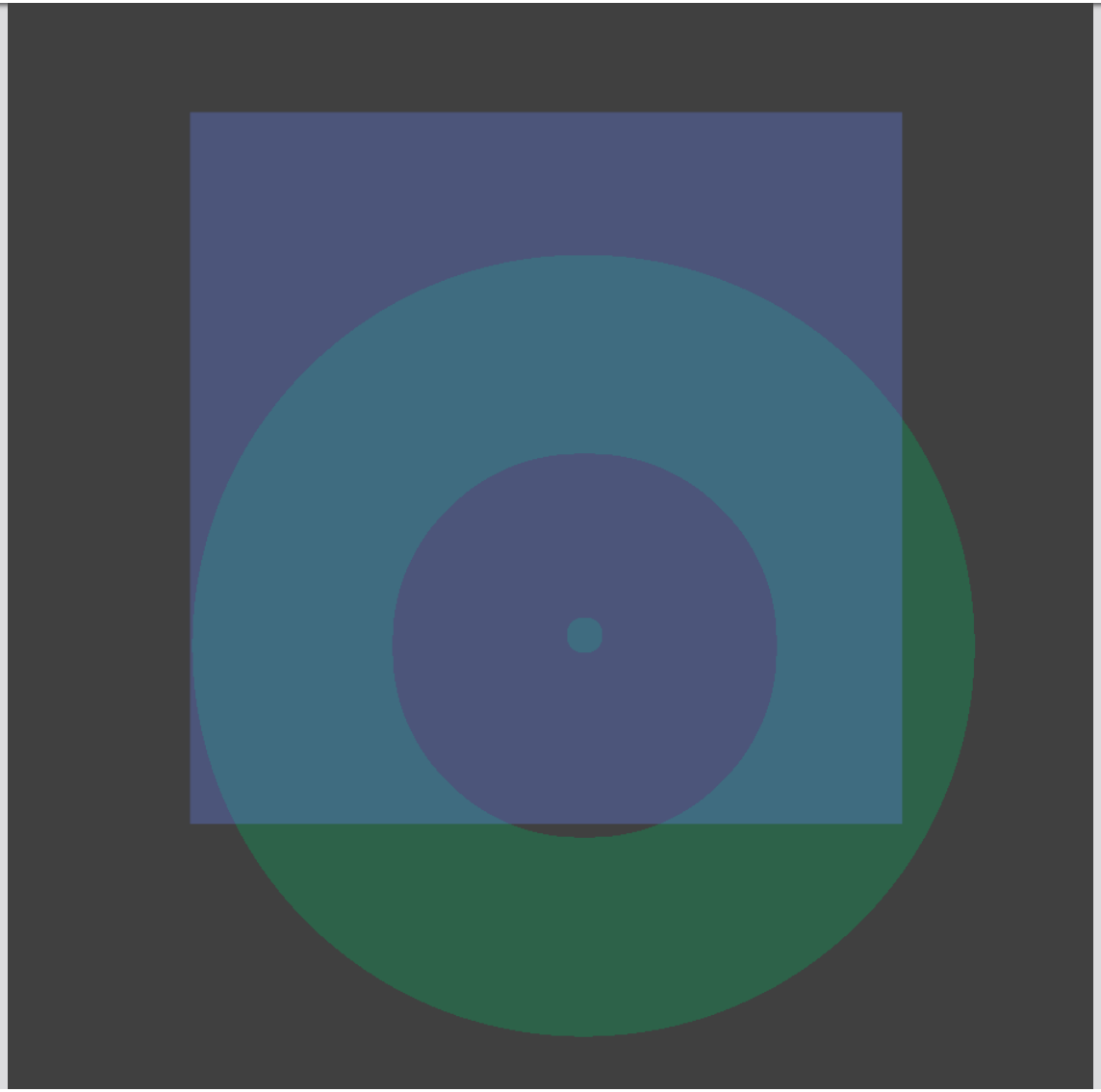
Motion Planning

- Green dot is piece location
- Green ring is set of reachable positions

Motion Planning

- Green dot is piece location
- Green ring is set of reachable positions

- Blue square is set of locations on top of or too close to chessboard

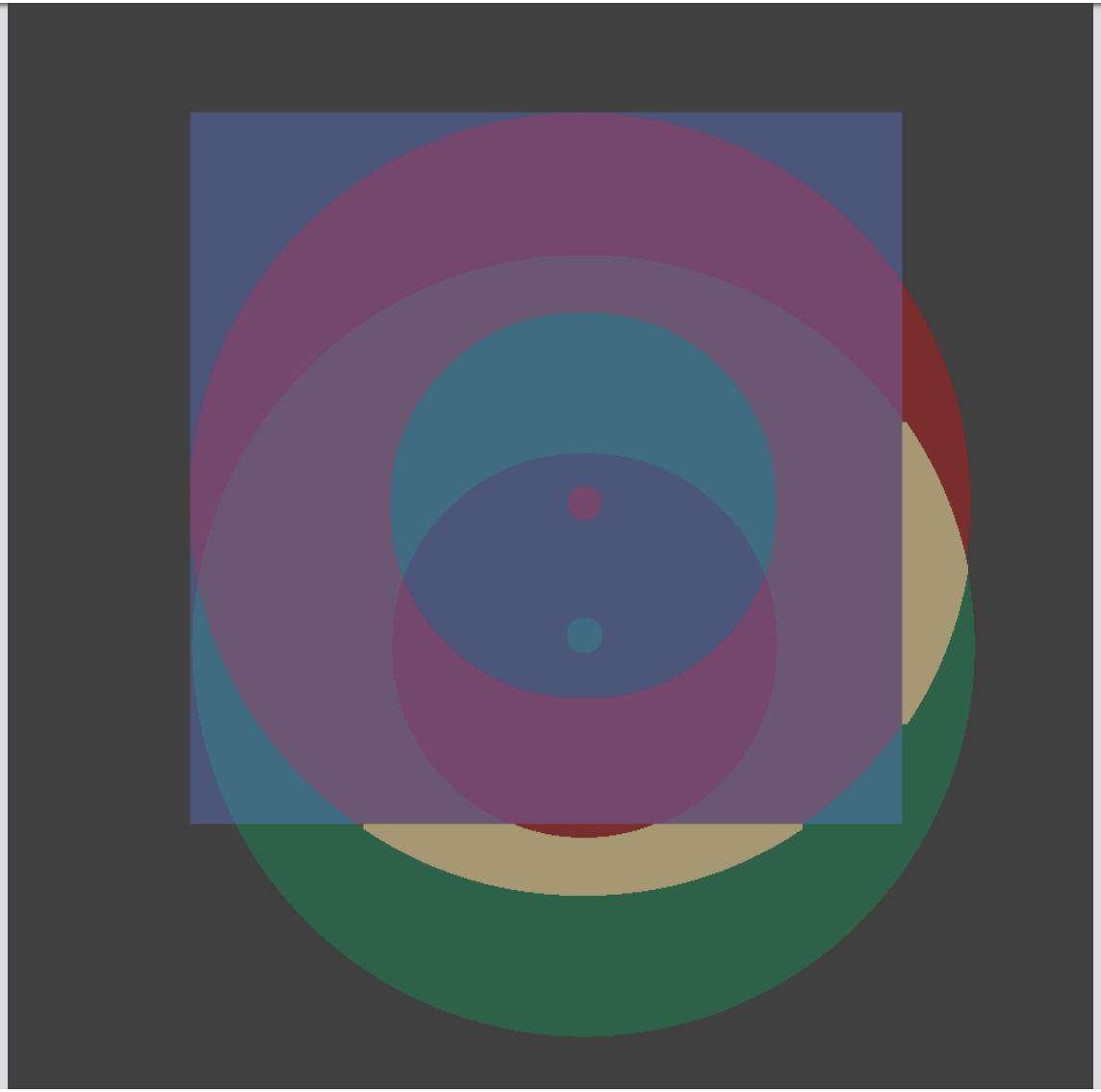


Motion Planning

- Green dot is piece location
- Green ring is set of reachable positions

- Blue square is set of locations on top of or too close to chessboard

- Red dot is piece destination
- Red ring is set of reachable positions



Motion Planning

- Green dot is piece location
- Red dot is destination location

- Tan areas are viable locations for the robot

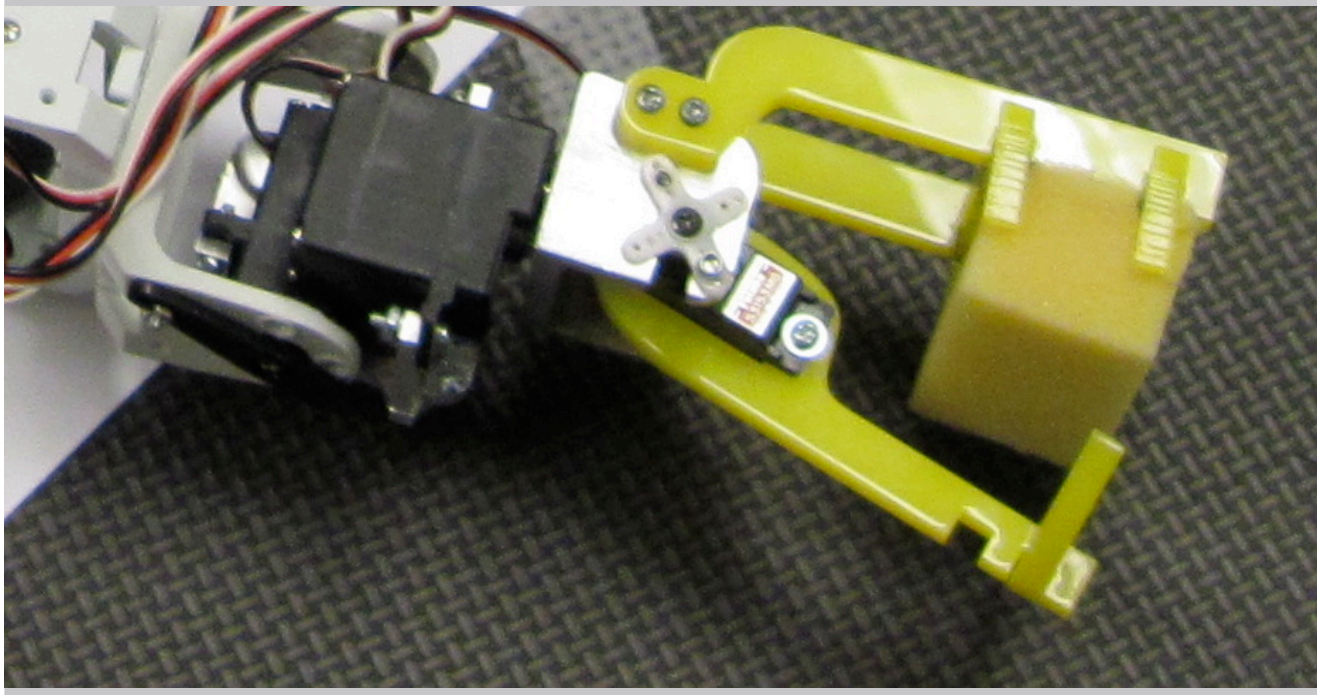
Grasping

- Manipulation strategy: Overhead approach
  - Position arm above piece
  - Pitch gripper down
  - Lower body so fingers surround piece
  - Close gripper : compliant envelopment
  - Raise body
  - Pitch gripper up

Gripper Design #1

- Palm + Thumb design
- Wrist pitch and roll
- 3 Futaba analog servos
- Foam helped form friction grip

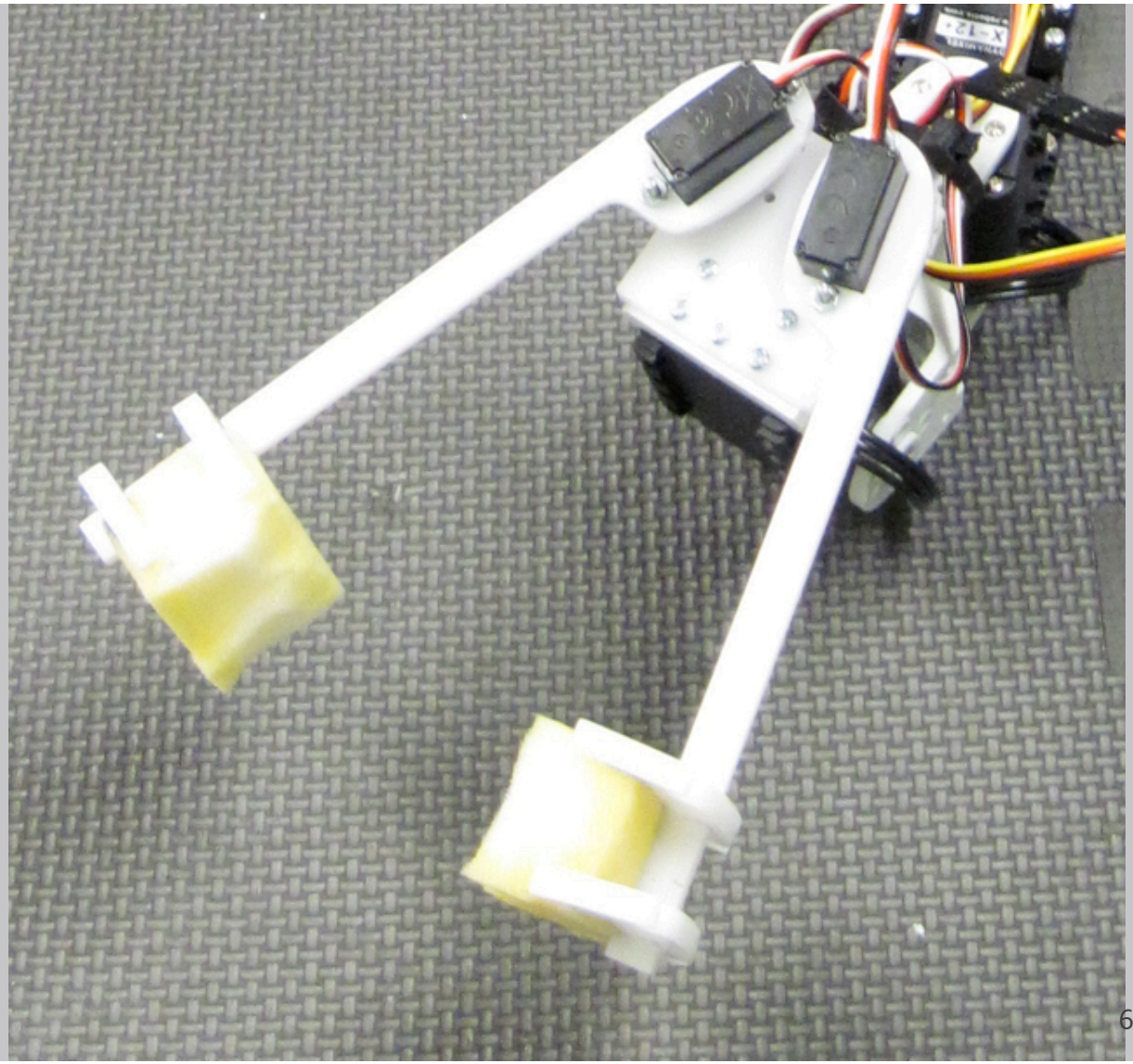
- Required too much precision
- Unable to grasp tall, heavy pieces



Gripper Design #2

- Two-fingered design
- Only wrist pitch
- 2 Futaba servos and 1 AX-12+

- Utilized more free space to overcome precision errors
- Capable of grasping all pieces



Grasper Primitives

- *grasp*
  - Pick up an object
- *release*
  - Place an object
- *rest*
  - Go to a stable configuration
- *moveTo*
  - Move an object from one location to another

Grasper Request

- Grasper request fields
  - Primitive to use
  - Object to manipulate
  - Target location
  - RRT planning parameters
  - Perform 2D or 3D manipulation
  - Acceptable gripper angles
  - Environment obstacles

Implementation

- RRT path planning for arm
  - Avoid obstacles and self-collisions
- Three types of arm trajectories
  - Place gripper within range of object
  - Move object while grasped
    - Subject to constraints of gripper
  - Disengage from object and return to rest state

Adapting for Chess

- Path planning in plane above board where robot's body is the only obstacle
  - Non-trivial leg collisions
  - Reach-ability
- Pitch gripper down to reach into plane of the board

Details of a *grasp* Operation

- Performing a grasp operation
  - Need height and radius of object to be grasped
  - Plan all arm movements before any motion
- Actions:
  - Stand up
  - Execute arm trajectory placing gripper over object
  - Run gripper's manipulation strategy
  - Execute arm trajectory resting arm
  - Place robot in resting position

Other Grasper Primitives

- *release* operation
  - Same as grasp, but open gripper
- *moveTo* operation
  - Combine *grasp* + *release*
  - Introduce third arm trajectory to move gripper
- *rest* operation
  - Rest arm and body

Higher-level Chess Planner

- Turn chess moves into interleaved Pilot and Grasper requests
  - Moving pieces
  - Moving the body
- Plan complex move sequence for captures
- Handle fallback cases when grasping fails

Interleaving grasping and moving

- If moving only one piece…
  - Pilot: Move to position where both piece and destination are reachable
  - Grasper: Perform *moveTo* on piece to location
  - Pilot: Return home
- If piece and destination are not simultaneously reachable
  - Pilot: Move so piece is reachable
  - Grasper: Perform *grasp* on piece
  - Pilot: Move so destination is reachable
  - Grasper: Perform *release* at destination
  - Pilot: Return home

Performing Captures

- If capturing
  - Need to move captured piece out of the way
    - Repositioning body is expensive…
  - Move captured piece to nearby empty square
  - Move capturing piece to appropriate square
  - Reacquire captured piece for removal

Capture example

Planning Failure Recovery

- *moveTo* failed to plan
  - Decompose into 1 *grasp* and 1 *release*

- *grasp* or *release* failed to plan
  - Too far away – Move body closer to target
  - Too close – Move body further from target
  - In range – Slight body position adjustment
    - No collision-free solution (obstructed by body or leg)

Failure Recovery Example

- In the following video
  - Attempts to move bishop in single move, but fails as the piece is too close to its leg
  - Planner recovers from error by changing strategy:
    - Back off from piece
    - Pick up piece
    - Navigate to have destination in range of the arm
    - Drop piece off

Failure Recovery Example

AAAI-2010

- Small-Scale Manipulation Challenge
  - 4 Teams (CMU, Intel, Georgia Tech, University of Alabama)
- Shortcuts
  - Larger chessboard (2.75" squares vs. 2.25")
  - No walking around sides of board
    - Only play on half the board
    - Override chess engine

AAAI-2010 Performance

- Successes
  - Vision never misattributed an opponent's move
  - Completed more than 90% of attempted manipulations
  - Competitive despite handicaps vs. other competitors
- Failures
  - Hardware failures (gripper servo)
  - Dropping knights

Conclusions

- Overcame occlusions and a cluttered board to play a board game
- Uncertain whether new servos are worth the extra cost
- Line extractor is good for single-pixel width lines and extracting grids
- Higher-level planner needed in Tekkotsu

- This work brings Tekkotsu out of the plane, but not into a fully-fledged 3D capable framework.
  - Planning in 2D plane **above** the board
  - Not reasoning about 3D volumes

Future Work

- Walk around board to play on full chessboard
- Identifying individual chess pieces
- Extend Grasper to accommodate additional manipulation strategies
- Play other board games
  - Checkers
  - Connect 4
  - Backgammon

Acknowledgements

- Dave Touretzky
- Ethan Tira-Thompson
- Glenn Nickens
- Design Team (Wayne Chung, Nathaniel Paffett-Lugassy, Federico Rios)

- NSF DUE-0717705