Taking Tekkotsu Out Of The Plane

Jonathan A. Coens

CMU - CS - 10 - 139
August 2010

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David S. Touretzky, Chair
Illah Nourbakhsh

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

This research was supported in part by National Science Foundation (NSF) award DUE-0717705 to David S. Touretzky. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily represent the views of the NSF or Carnegie Mellon University.

Keywords: Tekkotsu, Computer vision, Motion planning, Chess

Abstract:
The Tekkotsu robotic software framework's perception and manipulation primitives have largely been confined to planar worlds. In this thesis I bring Tekkotsu out of the plane by solving a three-dimensional manipulation task: autonomously playing chess on a real board. The project used a new version of the Chiara hexapod robot with a gripper customized for tournament-standard chess pieces. I developed techniques for detecting pieces, for using multiple images to deal with occlusions, for inferring opponent moves from noisy information about changes in square occupancy, and for localizing the robot with respect to the board. Accurate localization of pieces and the robot itself were achieved by a camera alignment procedure that produced a homography correction matrix that was then applied to the robot's camera projections. The Chiara's limited reach required moving its body relative to the board. I developed motion strategies for positioning the robot close to the board while keeping the legs from intruding into the playing area. Pieces were modeled as vertical cylinders of varying heights, and manipulation planning algorithms were developed to execute moves, including captures, using an optimal combination of arm trajectories and body motions. The robot successfully competed in the AAAI-2010 Small-Scale Manipulation Challenge. As a result of this work, Tekkotsu's dual-coding vision system can locate objects more accurately on its world map, and its manipulation planner has become more sophisticated. The techniques developed here can be applied to similar manipulation tasks, such as playing other board games.

I. Introduction

The majority of Tekkotsu's perception and manipulation primitives have been restricted to working in a 2D space. Having Tekkotsu solve a real-world 3D manipulation task—playing chess on a real chessboard—provided motivation to upgrade these primitives to handle the third dimension. This paper explores the issues and solutions for advancing Tekkotsu's capabilities to play chess. Chess pieces, modeled as vertical cylinders, were detected by combining multiple camera images and manipulated using a custom gripper designed for standard chess pieces. Inferring the opponent's moves and precise localization of the robot relative to the board required new visual reasoning techniques. Camera alignment, via a homography correction matrix, was used to adjust camera projections to more accurately localize visual features. The choice of the Chiara hexapod robot, with its limited reaching capabilities, led to new strategies for motion planning around the chessboard to be able to reach squares on the board. The Chiara's planar arm necessitated new manipulation planning strategies to pick up and place chess pieces without disturbing the state of the board. Combining all of these advancements allowed Tekkotsu's dual-coding vision system to more accurately locate objects in its world and perform certain 3D manipulation tasks that were previously not possible.

1. Tekkotsu

The Tekkotsu robotic software framework [4] is a robot-independent framework that abstracts away low-level concepts of robot programming by providing users with a set of higher-level primitives. It allows users to develop solutions to problems in terms of

desired outcomes instead of lower-level concepts such as servo positions or pixel values. For example, Tekkotsu allows a user to create a robot-independent behavior to look to the left of the robot, find the largest contiguous blue object, and estimate how far away from the robot the object is. Concepts such as what servo angle cause the camera to look to the left, where the largest blue object appears in the camera frame, and where that object projects to relative to the robot are all abstracted away. The most abstract Tekkotsu components, known as the Crew, are like members of a ship, offering specialized skills that empower the user to easily create complex robot behaviors. The four implemented members of the Crew are the Lookout, MapBuilder, Pilot, and Grasper.

1.1 The Crew [7]

The Lookout directs the robot's visual focus, determining how to point the camera. It also handles issues surrounding image collection, such as motion blur and camera noise. Lookout requests take the form of specifying a point for the camera to point at in order to capture an image or do rangefinder scans.

The MapBuilder determines the state of the world around the robot. It makes requests to the Lookout to collect images of areas of interest. Using these images, the MapBuilder has methods for extracting various geometric shapes, such as points, lines, blobs, and ellipses, and generating a model of the world in terms of these shapes. MapBuilder requests specify the areas of interest as well as which features and shapes a user is looking for. For example, one request could ask for the blue blobs in the

camera image when looking to the robot's left while a different request could ask for all perceived green lines to the left or right.

The Pilot is in charge of the robot's motion and navigation within the world. It knows how the robot locomotes and can plan a path to a destination while avoiding obstacles. The Pilot can invoke MapBuilder or Lookout requests in order to track navigation markers or attempt to detect new obstacles. Pilot requests take the form of specifying the robot's desired position relative to either itself or an object in the world.

The Grasper manipulates objects in the robot's world. It knows the capabilities of the robot's body and manipulators in order to generate actions that will satisfy the user's requests. It can issue MapBuilder requests to get more information about obstacles in the world and the object being manipulated. Grasper requests specify whether to grasp, release, or move some object to some location.

1.2 Dual-Coding Vision

Tekkotsu utilizes a powerful dual-coding vision system [6] to allow for extracting features from images and expressing them in several coordinate systems. The system exposes both iconic and symbolic representations for each known shape. The iconic representations are called "sketches," and the symbolic representations are called "shapes." This allows for different operations on each of the representations, and easy conversions between them. For example, a line can be recognized from a camera image sketch and translated into an algebraic line shape. The algebraic representation can be altered—rotated by 90 degrees for example—and a new sketch can be rendered from the altered shape. These operations take place in one of three coordinate spaces:

Camera space, where the sketches are camera images and the shapes are in pixel coordinates; local space, where sketches have pixels of 1 mm^2 area and shapes' coordinates are egocentric to the robot; or World space, which is similar to local space except that all pixels and coordinates are relative to a fixed point in the world rather than to the robot. The system also allows projections between each of the coordinate spaces through kinematic knowledge of the robot. The MapBuilder is the main entry point for a user to request the system to take an image, parse out shapes from sketches, and return those shapes in the appropriate coordinate space.

The dual-coding vision system uses color segmentation to categorize pixels of similar colors into a small number of color classes. Tekkotsu provides training tools to categorize colors from provided training images. Color segmentation is a powerful tool for removing computer vision issues that arise from solid colored objects having different shades of the same color throughout an image. However, color segmentation is sensitive to changes in the environment between the training images and the real images the robot perceives. For example, if the lighting in the room changes enough, a new set of training images may be necessary to correctly segment the camera image. Notice how each of the yellow and blue pieces is entirely classified as yellow and blue respectively in the color segmented picture in Figure 2. The green of the chessboard is almost entirely properly segmented. Also notice how specular reflections from a light near the lower left of the chessboard, underneath the queen, cause some pixels to not be classified as green.

Figure 1 - Raw RGB image of the chessboard

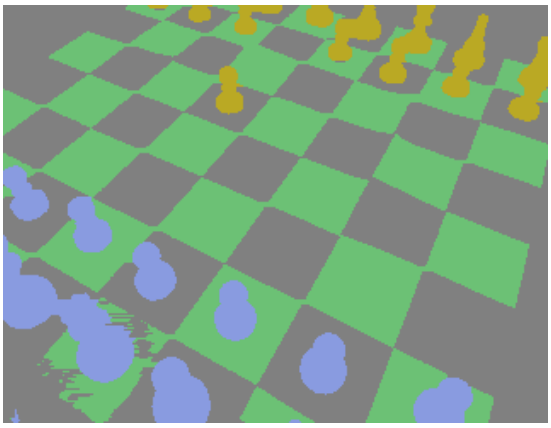

Figure 2 - Color segmented image

All vision algorithms and techniques used must be robust to a certain amount of noise in color segmentation. Different solid colors for the chess pieces and chessboard were chosen specifically for allowing color segmentation to effectively differentiate between pieces of the two players and the board itself.

1.3 Planar World Assumption

The majority of Tekkotsu's Crew and dual-coding system were written with the assumption that the objects in its world were 2D in the same plane that supported the robot. This worked for the majority of the tasks that Tekkotsu users set out to do because the cameras of the robots were high enough and the objects were flat enough that the elongation of tall objects was minimal. For example, users in robot education labs using Tekkotsu completed projects that parse out information from colored tape on the ground, such as tic-tac-toe parsing, and that use plastic Easter-egg shell halves, perceived as movable colored blobs for game pieces. However, the planar world assumption is not satisfied when playing chess on a real chessboard due to the heights of chess pieces. Without the planar world assumption, some members of the Crew did

not act as required. The Lookout was able to function because points the camera but doesn't interpret the images, but the MapBuilder, in conjunction with the dual-coding vision system, could not properly determine the state of the world when objects have had substantial height. The occlusions that the chess pieces cast on objects behind them rendered the shape parsing useless to detecting individual chess pieces. The Pilot was able to continue without the planar world assumption because movement around a chessboard can still be decomposed into a 2D problem, so the Pilot did not need 3D information to succeed. Lastly, the Grasper had no concept of going over an object, only around it. Trying to go around all obstacles would create far too many constraints to easily play chess, so the Grasper's 2D manipulation planning was not applicable. All of these issues are addressed in my work.

2. The Chiara

The Chiara [5] is a hexapod robot developed at Carnegie Mellon University's Tekkotsu Lab. It has a camera on a pan/tilt mount, six legs for holonomic motion, and a planar three-link arm on the front. Two versions of the Chiara were used in my work. The gamma series Chiara's (Figure 3) hardware consists of the following: 24 Dynamixel AX-12+ servos; a Pico-ITX x86 computer with a 1GHz processor, 1GB of RAM, and an 80GB hard drive; a Logitech QuickCam-Pro 9000 webcam; a Dynamixel AX-S1 IR rangefinder; an 802.11 networking dongle; and a speaker. The delta series Chiara's (Figure 4) hardware is similar to the gamma series' except for the following: a different body design; 12 Dynamixel AX-12+ and 12 Dynamiel RX-28 servos; an Advantech PCM-9361 single board computer with an Intel 1.6GHz Atom processor, 1GB of RAM,

and 250GB hard drive; and a USB-powered speaker. The RX-28 servos, placed in the knee and elevator leg joints, provide higher torque than the AX-12+'s and have metal gears instead of plastic gears. Both robots run Ubuntu Linux with the Tekkotsu robotic framework installed.

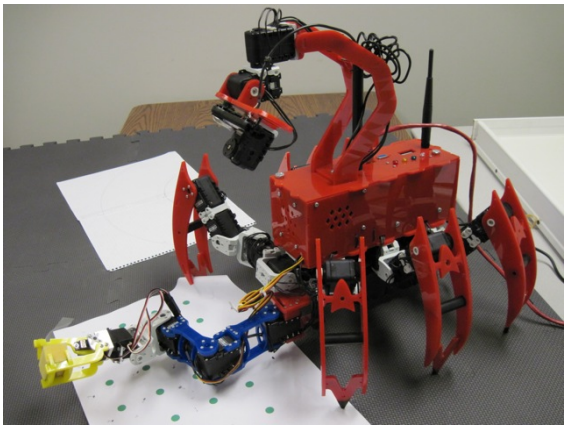

Figure 3 - Gamma series Chiara with first Gripper

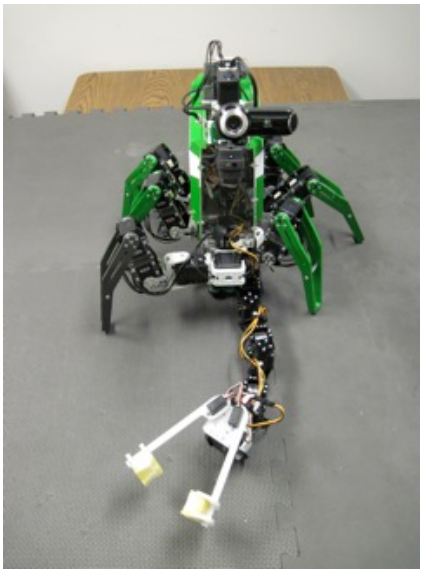

Figure 4 - Delta series Chiara with second gripper

The legged robot allows for greater precision and mobility in locomotion than wheeled robots and an additional degree of freedom vertically for the robot to "stand up." Standing up straightens the legs of the robot to raise its body over 12 centimeters, allowing the arm to reach over objects of significant height and grasp them from above. This changes the plane in which the planar three-link arm operates.

Figure 5 - The robot sitting next to the board

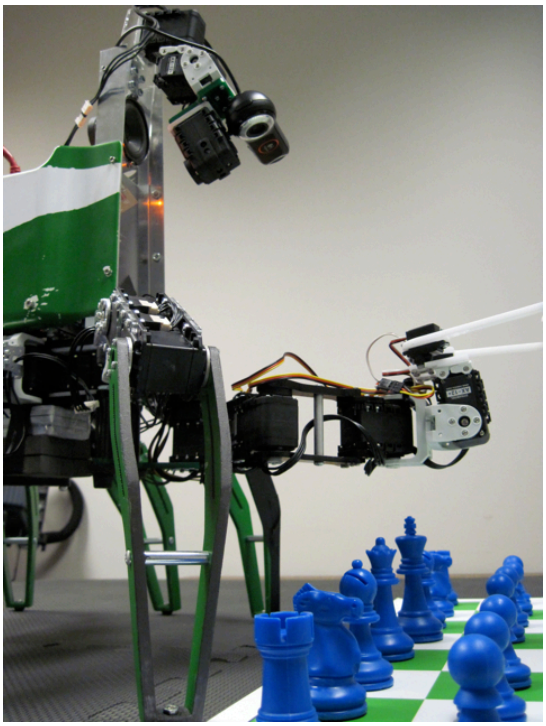

Figure 6 - The robot standing above the board

Combining altering the plane of the arm and planning arm trajectories through the chosen plane opens up possibilities for 3D manipulation. By reaching over the pieces on the chessboard, the Chiara is able to grasp pieces that would otherwise over-constrain the arm.

The delta series Chiara was designed with chess playing in mind. The stresses of frequently standing up to perform manipulations led to upgrading the two most stressed servos per leg to stronger and more reliable servos. Furthermore, since the Chiara did not yet have a standard gripper, a custom gripper was designed for picking up tournament-style chess pieces.

II. Problems

To autonomously play chess on a real chessboard, three separate problems must be solved. First, the robot must visually detect what move the opposing player previously made. This involves detecting changes in square occupancy and appropriately updating internal representations of the game state. Second, the robot must select a legal chess move to perform. Ideally this selection would be intelligent enough to play a competitive game of chess. Lastly, the robot must reliably execute the selected move on the chessboard and leave the board in a state for the opposing player to make a move. This involves planning efficient movement and manipulation trajectories to complete the desired move, including captures, without disturbing the remaining pieces on the board.

1. Determining Opponent's Move

Determining the opponent's move is achieved through visual perception of the chessboard. By combining features extracted from multiple camera images of the board, changes in square occupancy are detected. These occupancy changes are then used to update the internal representations of the game state for later use.

In order to take multiple camera images and accurately move relative to the chessboard, the robot must know where it is relative to the board. With the assumption that the robot is positioned somewhere on its side of the board, a custom localization technique is used to precisely determine the robot's position and orientation in the world frame. From this, the robot can look at different parts of the board. However, given manufacturing inconsistencies in the robot's body and camera, the actual position and orientation of the camera deviate far enough from the expected position that camera

projections become a large source of error. With inaccurate projections, the localization incorrectly calculates the position and orientation of the robot. Therefore, a camera alignment technique was implemented via a homography correction matrix.

Due to chess pieces breaking the dual-coding vision system's heavy reliance on a planar world, new techniques for detecting the features of the chessboard were necessary. Not being able to directly differentiate piece shapes focused attention on detecting square occupancy, which would allow inference of piece identity by tracking occupancy changes. First, the chess pieces themselves needed to be extracted. Since chess pieces can be thought of as tall and slender cylinders, I created methods for extracting where the bottoms of cylinders lay in camera space sketches. This method was susceptible to pieces occluding other pieces, so I explored strategies for taking multiple images from different angles to overcome these occlusions. Second, the board squares themselves needed to be extracted in order to determine the relationships between the detected pieces. No assumptions could be made for how much of the board was in view in each camera image, how many lines should be extracted, and at what orientation the board would lie, so the techniques needed to be robust to all of these.

Once square occupancy was determined, the information gained needed to be compared against the previous state of the board in order to figure out what changes occurred. Noisy information due to occlusion frequently required information from multiple camera images to be combined to confidently construe the opponent's move. To reduce the number of occlusions, a larger, non-standard chessboard was used with

2.75” squares rather than 2.25” squares. As soon as enough criteria were met from all the images, the opponent’s move was deduced and the internal board state was updated accordingly.

2. Selecting the Robot’s Move

The easiest way to achieve competitive chess play was through use of an open source chess engine. Since chess engines are heavily researched algorithms, there were several to choose from. I selected the GNU Chess [2] chess engine to perform all chess logic due to its easy portability.

3. Executing Moves

Executing moves was accomplished by interleaving motion and manipulation actions. Due to Tekkotsu’s original Grasper being restricted to a planar world, the Grasper, as well as all manipulation actions, needed to be extended to handle moving chess pieces in three dimensions. The Chiara’s gripper places constraints on how the robot is capable of performing manipulations. Therefore, I designed multiple iterations of the gripper that utilize a vertical approach to grasp the chess pieces. With this capability in place, the Grasper needed to be extended to use this manipulation strategy while maintaining its ability to operate only in a plane if necessary. With these in mind, I developed primitives to accomplish various manipulation tasks.

In order to get the Chiara within range of the piece it needed to move, I developed motion-planning strategies to get the robot close to the board without entering the playable area. By combining information on all pertinent locations used during a manipulation action, an optimal positioning of the body was calculated to minimize the

number of body movements between manipulations for multi-manipulation chess moves, such as captures. Furthermore, an approach behavior was created to prevent the robot from entering the playable area to perform a manipulation, and to place the legs in a stable position for balanced standing while manipulating.

Some accuracy issues arose from only using dead reckoning for motion and manipulation. The Chiara's locomotion was not precise enough for accurate placement of pieces, so a new localization technique different from the initial localization action was used. This dramatically increased placement accuracy. Once in place, the chess pieces' positions were visually detected. The pieces were not guaranteed to be in the center of each board square, so the visual detection step made the system robust to off-center pieces.

With both motion and manipulation planning strategies in place, a higher-level planner was created to interleave motion and manipulation actions to complete the requested chess move. Captures required much more advanced interleaving since at least three different manipulations needed to happen: moving the captured piece to an unoccupied square, placing the capturing piece into the appropriate square, and picking the captured piece back up for removal from the board. Furthermore, the higher-level planner introduced fallback conditions for whenever a manipulation action was unable to succeed. After executing the series of planned actions, the requested chess move was successfully accomplished.

III. Solutions

1. Determining Opponent's Move

1.1 Initial Localization

The initial localization was solved in a three step process. Upon initialization, the robot first built a map of the world in local space, egocentric coordinates, of all green, yellow, and blue objects to the robot's right, front, and left. This task was a common application of the MapBuilder. Given this map of the world, the average position of all the green, yellow, and blue objects was the most likely candidate for the location of the chessboard. This strategy could fail if there were many other green, yellow, and blue objects in close proximity to the robot, but this was not a concern. From this map, the robot now had a general idea of the direction in which the chessboard lay relative to itself.

The next step attempted to center the camera on the lower left corner of the chessboard. The robot can exploit the assumption that the lower left corner of the chessboard is a green corner because chess regulations dictate that the lower left corner is colored. After centering the camera frame on the center of the approximate chessboard location, the robot performed an iterative technique of looking at large green collections and shifting its gaze until the majority of green things in the camera frame lay in the right half or top half, depending on whether the center of the approximate chessboard was to the left or right of the robot respectively. This should place the lower left corner of the chessboard within the camera frame.

The third step was to extract the location of the lower left corner for positioning and the bottom border of the board for orientation. The corner was extracted by finding the leftmost or bottommost green pixel in the camera frame depending on whether the robot was looking left or right respectively. The bottom line was extracted by fitting lines of increasing orientation through the extracted corner pixel until a suitable line was found. Projecting the point and line from the camera frame to local space coordinates gave a precise measurement of the robot's position and orientation relative to the board. Figure 7 shows a sample extraction of the corner and line in the camera frame. The green pixels are of the chessboard, the blue dot represents the location of the extracted corner pixel, and the pink line represents the extracted bottom line.

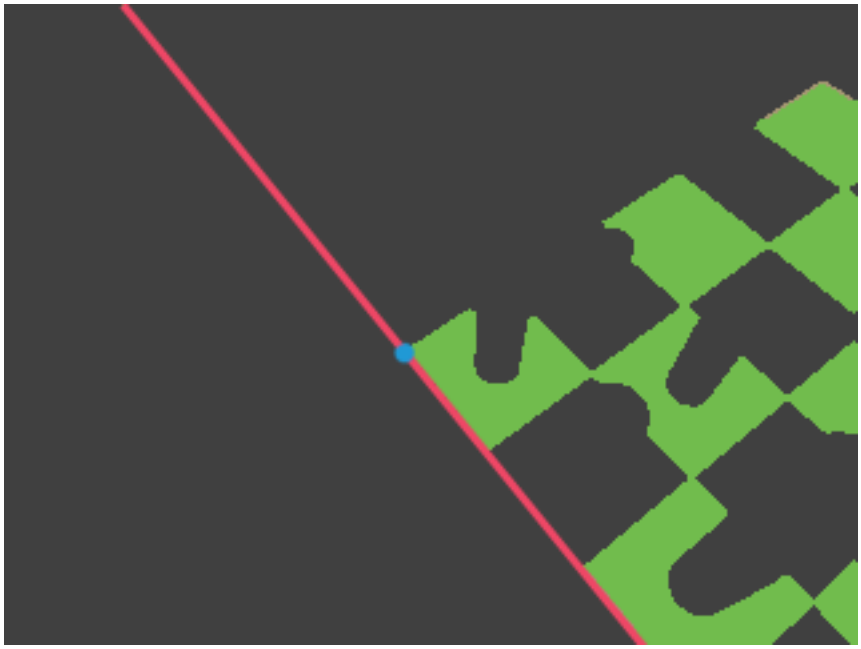

Figure 7 - Parsing the lower left corner (blue dot) and bottom line (pink) for localization when looking left

1.2 Camera Alignment

It became apparent that the positioning of the robot relative to the board was off by a systematic error depending on where the corner pixel lay in the camera frame. This

error was attributed to inaccurate projections from the camera frame into the world. To correct for this, I implemented an alignment technique that applies a correctional homography matrix to all camera projections after calibration. Since camera misalignment results in translations and rotations of the projections without any warping, a homography matrix that maps straight lines from one coordinate space to straight lines in a different coordinate space corrects for these errors [9]. The code for computing the homography matrix was ported from the AprilTags vision system [1]. The implementation gathered correspondences for mapping points in one frame to points in another frame and calculated the matrix that minimized the overall error in transforming each correspondence point to its correct position.

To gather these correspondences from the camera of the robot, a custom alignment rig was created. The rig screwed into the bottom of the Chiara to minimize any shifting and to fix a common point between the robot and the rig. The rig placed two colored diamond shapes directly in front of the robot on the ground plane, one to the right and one to the left.

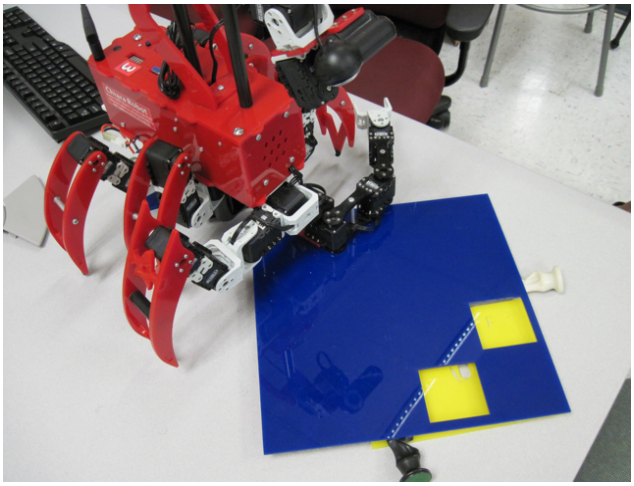

Figure 8 - Alignment rig connected to the Chiara

The robot would look directly in between the two diamonds and extract all four corners from both diamonds. Given the known location of the alignment rig and the diamonds relative to the robot, the expected location of each of the corners of the diamonds in the camera frame was computed. The pairs of expected location corners and extracted corners supplied eight correspondences to be used in calculating the homography. For example, the correspondence between the right corner of the right diamond extracted from the camera frame and its expected location computed from kinematic projections is one factor to weigh into the homography calculation.

Applying the homography to camera projections reduced the projection error to a tolerable level. Viewing the adjustments in the camera frame showed obvious errors in the camera alignment. In Figure 9, the yellow blobs represent the perceived diamonds from the camera rig, the green dots represent the expected location of each corresponding corner, and the blue lines show the error in the correspondence between the expected and extracted corners. Notice how all the green points lie below their extracted corresponding corners.

Figure 9 - A visual representation of the correspondences for camera alignment

Applying the homography to all projections gave more accurate projections for features extracted from camera images. Projecting the locations of chess pieces, points on the chessboard, and lines from local space into the camera frame, all fell within tolerable error levels. The code can easily be adapted to perform alignment of other cameras on other robot platforms running Tekkotsu, given appropriate alignment rigs.

1.3 Detecting Square Occupancy

Detecting square occupancy from a camera image was done in two stages. First, the locations of the chess pieces were extracted through a series of dual-coding sketch operations. This approximately identified the bottom of each chess piece in an image. Second, the board squares were extracted through a custom vision algorithm based on a Hough transform. From these two pieces of information, the state of each board square could be determined.

1.3.1 Detecting chess pieces

The inspiration for detecting chess pieces was taken from previous work in Tekkotsu used to detect the location of Easter-egg halves. When looking for a particular colored Easter-egg half, the most reliable way to extract an exact position is based off the set of pixels in the camera frame that are the target egg's color and whose southern neighbor pixel is not the target's color. These sets of pixels are called the under-pixels. Applying this technique of finding under-pixels to chess pieces also worked well but resulted in a noisy pixel set due to ornamental features in the chess pieces.

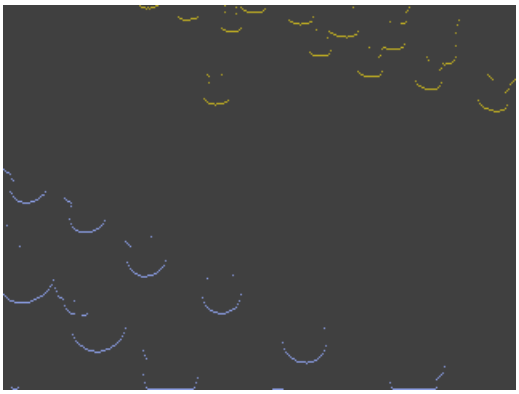

Figure 10 - The under-pixels forming "smiles"

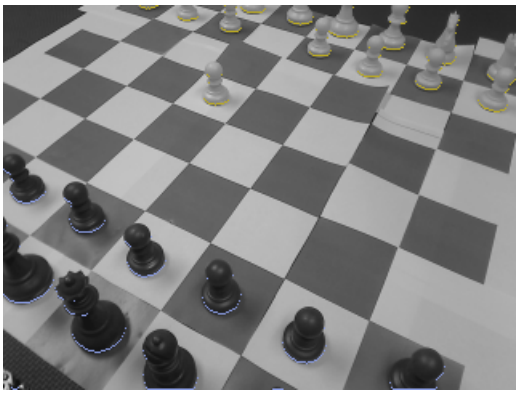

Figure 11 - Under-pixels overlaid on the original image

The pixel set from the pieces formed “smile” shapes around each chess piece in the camera frame. Classifying only the mouths of the smiles as pieces required filtering out the noise from the ornamental features. Through a series of dual-coding visual operations, the pixels were bloated into larger connected components. Filtering out the components with too small an area left only the components derived from smiles. Taking the centers of these remaining components gave usable approximate locations of each piece in the camera frame.

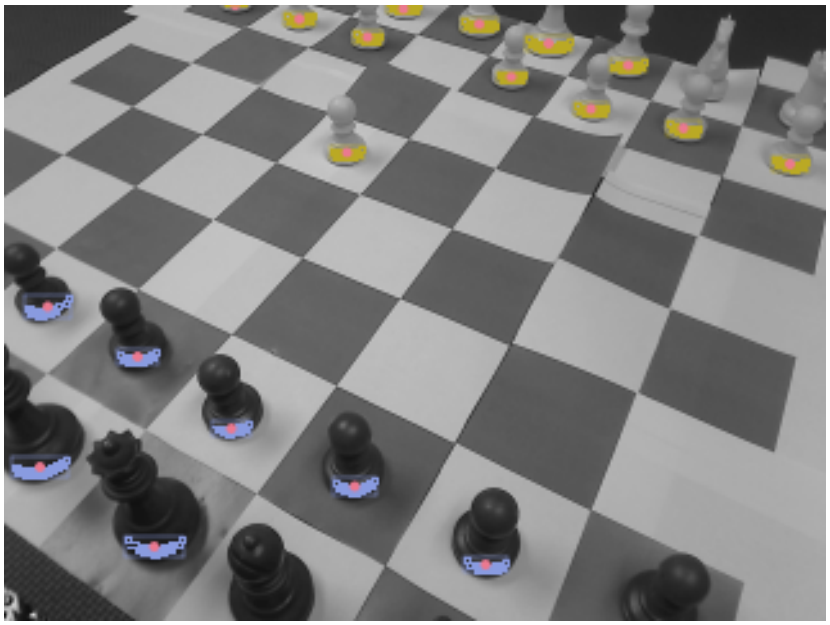

Figure 12 - Extracted approximate chess piece locations for yellow and blue pieces

This technique rarely yielded any false positives in images, but was susceptible to not extracting every piece in the frame due to occlusions from other pieces or only partial piece views. Notice how, in Figure 12, the bishop and pawn along the bottom are not identified. Without reliably seeing the bottom of either piece, the under-pixels do not appear in the camera frame. Also notice how the knight and rook in the upper-right of the image are not identified. The occlusion of those pieces by the pawns in front prevents the appropriate under-pixels from being detected, as can be seen in Figure 10. The lack of a pronounced smile causes the associated connected components to be filtered out. If the same board were viewed from a different angle, though, the missed pieces from this image might be visible and could be extracted successfully. When not enough information was extracted from a picture, more pictures from different vantage points were collected.

To employ multiple images from different viewing angles and locations, I developed a strategy for shifting the robot and combining information from multiple images. If another image was needed, the robot would sway horizontally and shift the camera gaze in the opposite direction. Shifting in this manner maximized the difference in parallax between chess pieces in an attempt to obtain an unobstructed view of occluded pieces. For example, the robot would sway its body to the left, shifting the camera position leftward, and then pan right to look at the right side of the board. If that did not provide enough new information, the robot would then sway its body to the right and look at the left side of the board. If this still failed to provide enough information, the robot would walk, shifting its body location by 100 mm to a new vantage point. This

strategy proved robust at capturing images from different positions and directions to extract enough information to overcome occlusions.

1.3.2 Extracting board squares

Board squares were extracted from the green pixels in the camera frame. To determine which square each chess piece was contained within, each square's pixel area in the camera image needed to be extracted. This resulted in parsing out the lines between squares on the board. Taking all green pixels from the camera frame and using a series of dual-coding visual operations yielded a sketch of the green pixel edges.

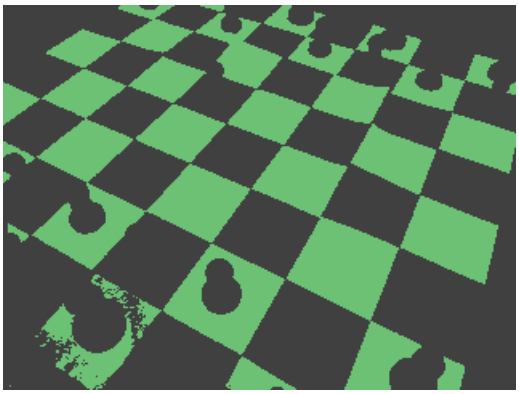

Figure 13 - All green pixels in the camera frame

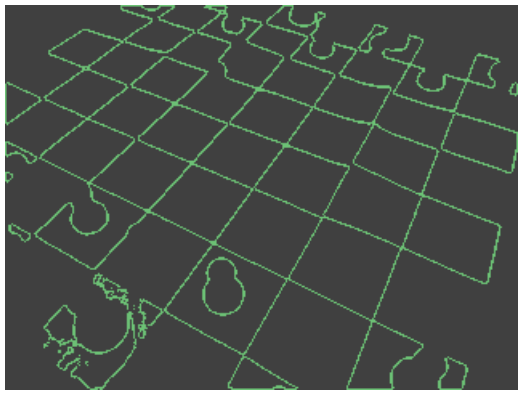

Figure 14 - The green edge lines

These green edge pixels provided a suitable input to a Hough transform to extract the lines. However, a Hough transform by itself was not sufficient to robustly parse out the lines of the board. Since the expected number of lines the transform should perceive depended on how much of the board was in the camera frame, there was no easy rule for what confidence threshold to set on determining which lines from the transform were board lines. Two more issues confounded placing the confidence threshold. Since the camera's aspect ratio was not 1:1, horizontal lines were more probable than vertical lines, causing the transform to pull out more horizontal than vertical lines. Inversely

weighting the probability of each line based on its overall length in the camera frame evened out the confidences of vertical, horizontal, and diagonal lines. Furthermore, each board line did not project a perfectly linear set of pixels in the camera. This pixel aliasing caused the Hough transform to extract multiple probable lines for each board line. To combat this, the 40 most probable lines were extracted out of the transform, and filters were applied to weed out the duplicate probable lines per board line.

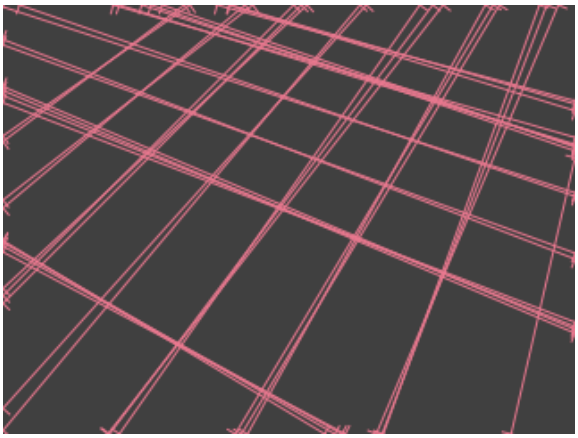

Figure 15 - The 40 most probable lines

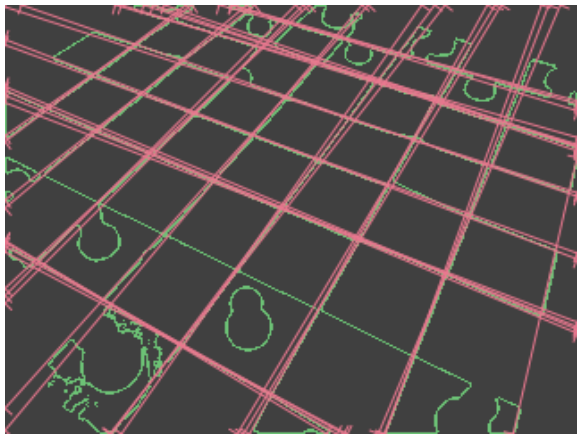

Figure 16 - The lines over the green pixels

As each next most probable line was extracted, its proximity in position and orientation to and intersections with previously extracted lines were tested. Filtering the lines that intersected previously extracted lines of similar orientation or that were too close to previously extracted lines resulted in an adequate base set of board lines. This set of lines was then split into two sets of lines, one per board dimension, and then sorted in terms of proximity. After sorting, one set contained all vertical lines in order from top to bottom and the other contained all horizontal lines in order from left to right.

Taking the 40 most probable lines succeeded in pulling out possible board lines, but it did not guarantee parsing out all possible board lines in the image. Figure 15 clearly

shows this with one of the horizontal lines and three edges of the board not being parsed. This discrepancy forced a post-processing extrapolation step to look for missed lines in between the parsed out lines as well as off the ends of each line set.

To extract missing lines in between parsed lines, the intersections of each line from one dimension's line set with a line from the other dimension's line set formed a set of points in the camera frame. Analyzing patterns in the consecutive differences between these points allowed for outlier detection. If the difference between two points was large enough to be considered an outlier, searching for local maxima in the Hough transform for lines of the same orientation through points a fraction of the way between the two points creating the outlier found the missing line. Depending on whether the outlier difference was on the order of two or three times larger than the average differences, the point to start searching in the Hough transform was either a half or a third of the outlier distance. To extract missing lines off the ends of each line set, a similar approach to finding missed in between lines worked well. Calculating average differences between points along one dimension's line set's intersection with lines in the other dimension gave a basis for where to start searching in the Hough transform off the ends of the list. If a local minimum in the transform was probable enough, it would be added to that line set.

After this post-processing step, the two line sets would contain the most probable board lines. In Figures 17 and 18, the green pixels are the green edge pixels from the camera image and the pink lines are the final extracted board lines. Notice how the previously missed horizontal line was extracted as well as the topmost and leftmost

board edges. The bottommost line did not meet the threshold for being accepted during extrapolation and the rightmost line was already extracted. This was an acceptable failure because enough information from the rest of the board was extracted to deduce the opponent's move.

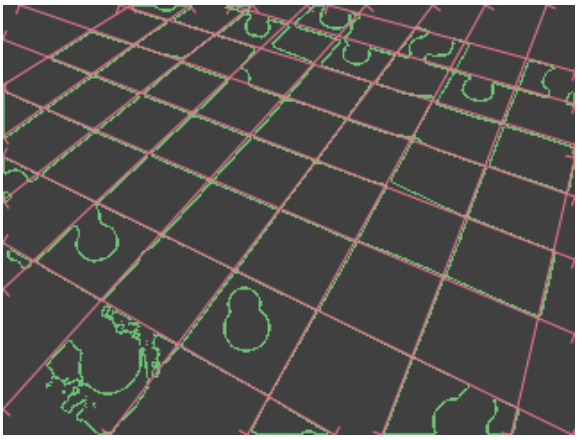

Figure 17 - Extracted board lines from green pixels

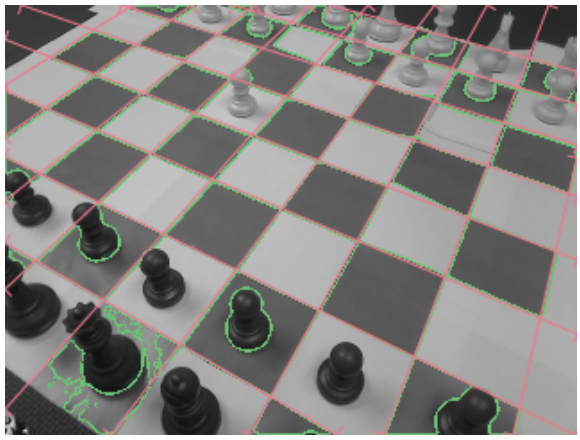

Figure 18 - The lines over the original image

With the chess pieces and board lines parsed out of the camera image, detecting square occupancy was a matter of determining which squares the chess piece centers lay within. For each consecutive pair of lines in both line dimensions, if a piece center lay in between both pairs of lines, then that corresponding square must be occupied by a piece of that color. This results in a parsed color representation of the board.

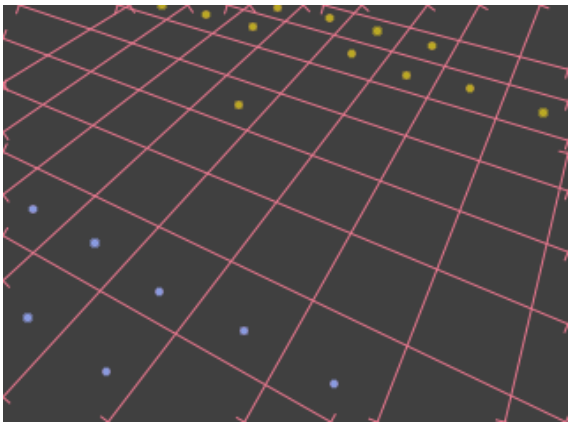

Figure 19 - The parsed lines and piece centers

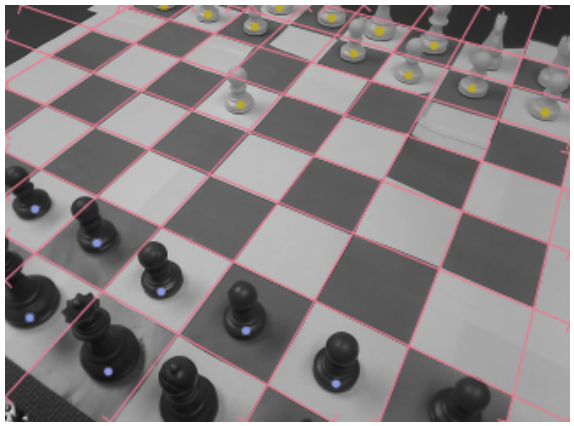

Figure 20 - Parsed features over the image

1.4 Partial Board Views

As seen in Figure 12 of the example board state parse, some squares of the chessboard were not in the camera frame. If the opposing player's turn changed the state of one of the missed board squares, then that change would not be perceived. Therefore, multiple pictures of different sections of the board needed to be compiled into a single perceived board state. As each new picture was parsed, its information was localized against the previously perceived information to find an alignment that minimized the total number of errors between the perceptions. If the alignment was unambiguous, the two states would be merged into a single board state. For example, if the left six columns of the chessboard had already been perceived and a new image with seven perceived columns came in, the alignment between the two boards with the fewest number of errors between them was chosen, resulting in a combined state containing seven or all eight columns. From this, the previously unknown columns would be updated with the information from the new image.

After each merge of perceived information, the perceived state of the board was compared against the previously known state of the board. Using the same board alignment technique, the alignment with the minimum number of errors was chosen. If the minimum alignment was unambiguous, with no two alignments resulting in the same number of errors, then that state was accepted as a successful board perception. Otherwise, more images of the board would be taken to collect more information until alignment was no longer ambiguous.

1.5 Deducing the Move

Given an unambiguous alignment between the perceived state of the board and the previously known board, analyzing the differences between these two states exposed what move the opposing player made. Three types of differences were possible: vacated squares that had a piece in the previous board state but no piece in the perceived board state, occupied squares that had a piece in the perceived board state but not in the previous board state, and taken squares that had a piece of one color in the perceived board state but a piece of the other color in the previous board state. If no vacated squares existed, no squares were occupied or taken, or more than one square was occupied or taken, then the perceived board was rejected. Each of these conditions implied an action that could not have been a legal chess move, so the perceived parsing of the board was assumed to have failed. Castling, where two squares are vacated, was the only exception to these rules and was handled as a special case.

The requirement that there be no more than one vacated square would have led to rejecting too many parses because occlusions and partial or missing piece views in the camera frame created many vacated spaces where pieces should have been. Many images of the board would be necessary to guarantee a perfect board parse, so to reduce the number of pictures taken, this requirement was relaxed and some extra processing was done on the vacated squares.

Given only one occupied or taken square, moving each vacated square's piece to the occupied or taken square was tested as a legal chess move. If only one of the vacated square's pieces moving to the occupied or taken square was a valid chess

move, then that move was confidently accepted and the internal board state was updated accordingly. For example, if the opponent's opening move was moving a pawn forward, the perceived board state would see the pawn's new location as occupied, the pawn's old location as vacated, and a handful of pieces on the opponent's back row of the board as vacated due to occlusions. However, only the pawn's old location to the new location was a valid chess move since no other chess piece could have legally moved to that square. Thus, the internal board state would be updated with the pawn's new location. If more than one vacated squares' piece could have legally moved into the occupied or taken square, the move was considered ambiguous and more images of the board were necessary. Combining these rules for inferring the opponent's move with the ability to take multiple images of the chess board to gain more information, these vision techniques accurately and robustly perceived the state of the chessboard after an opponent's move.

2. Selecting the Robot's Move

To perform all chess logic, I used the GNU Chess [2] chess engine. The engine was FOSS (Free Open Source Software), which allowed for easy integration into Tekkotsu. Some alterations were necessary to interact with the chess engine given the perception's internal representation of the board state. The engine allowed for testing move legality given the current board state, calculating a competitive next move along with specifying chess states such as capturing and being in check, undoing a previous move, and updating game states after perceiving a move. These features were used as a black box to perform all chess game logic on the robot.

3. Executing the Move

3.1 Grasper Primitives

The Grasper was originally written for use on planar robot arms. This was sufficient when the Chiara's planar three-link arm was positioned on the ground plane, but was insufficient for handling all the constraints for moving chess pieces. Therefore, the Grasper was rewritten to handle both 2D planar tasks as well as certain 3D manipulation techniques to play chess. The four primary action requests for the Grasper are picking up an object (grasp), placing an object (release), moving an object (moveTo), and withdrawing after manipulation (rest). Each of these actions was decomposed into planning and various execution stages with all required information coming through Grasper requests.

Depending on what kind of gripper the robot had, different grasping techniques were needed. A planar three-link arm with a fixed end-effector required a different technique than the Chiara's 3D manipulator. Moving an object with a planar three-link arm was decomposed into planning three arm trajectories: one to go from the arm's current state to having the object within its grasp, one to move the object from its current position to the desired location, and one to back the arm away from the object and return to a rest state. Using an RRT path-planner [3], these arm trajectories avoided projected obstacles in the environment, avoided colliding with the robot's body, and maintained any contact constraints imposed by the chosen gripper. This strategy was extended to 3D manipulation by adjusting the height of the arm's working plane and adjusting the obstacles in the environment before executing any of the planned paths. Given a set of

3D obstacles in the environment, a target object to move, and a target location to place the object, I extended the Grasper to adjust its standing height between arm motions to avoid collisions with the environment.

To grasp an object using the 3D manipulation technique for the Chiara, the Grasper first planned two arm trajectories, moving the gripper above the object and moving the gripper from that position to a resting state. If both trajectories were possible, the Grasper then commanded the robot to stand up to a level above all environment obstacles, if possible, and execute the first arm trajectory to place the gripper over the target object. Once the gripper was in position, it then ran the appropriate action for grasping the object; pitching the gripper down, lowering the arm's working plane to enclose the object, closing the gripper, standing back up to the previous height, and pitching the gripper back up. Lastly, the second arm trajectory was executed to place the arm in a resting state, and the default standing level was restored. This successfully grasped the object and left it contained in the gripper. The other primary Grasper actions were slightly different versions of the grasping technique described. Releasing an object only differed in moving the arm above the target location from the Grasper request and opening the gripper rather than closing. Moving an object tied grasping and releasing actions together without resting and incorporated the third arm trajectory for moving the gripper from above the pick-up location to above the target location. Resting took the robot from its current state to the rest state.

For each of these tasks, all workspace trajectories were planned before any motion was initiated. If any step of the planning failed, no motion would occur and appropriate

error conditions would be returned to the user. This prevented executing only a portion of a task and allowed the programmer to attempt alternative strategies. For example, if an application requested to move an object to a new location that was out of the arm's reach, the Grasper's planning stage would recognize this and would return an error code indicating that the target location was out of range. Catching errors in planning before execution also prevented expensive path planning operations from occurring while the robot was in a stressed position, since spending excess time in strenuous positions decreased the hardware's lifespan.

The Grasper request provides many parameters to allow users to control the planning and execution of manipulation tasks. The request contained fields that a user could populate for which shape to pick up, where to drop off the shape, and whether to settle both the arm and body on rest or just the arm. Other fields included all appropriate parameters for adjusting RRT planning, whether to perform 3D or 2D manipulation, what angles the gripper should be at when manipulating, and what objects should be treated as obstacles in the environment. Given default values for these fields, a user only needed to set the manipulation action to take, the shape to be moved, and the target location. At that point, the Grasper would execute that action, if possible.

3.2 The Gripper

The chess pieces were tournament-style chess pieces. They were of Staunton design with the king between 85mm and 105mm tall, the king's diameter between 40% and 50% of its height, and all other pieces with similar proportions [8]. To grasp these chess pieces, a new gripper for the Chiara was required. With a strategy for vertical

approach of the pieces, the gripper needed to have a wrist capable of pitching up and down. It also had to be wide enough to surround the piece on approach without disturbing it and be able to close on the piece to enforce compliant envelopment. The first design was made from three Futaba high precision servos that created a wrist with pitch and roll and a gripper in a palm and thumb formation. In order to grasp pieces of different forms, foam was attached to the effector end of the palm while the thumb consisted of a concave arc. The closing motion would swing the arc of the thumb into the foam of the palm, gripping the piece between them.

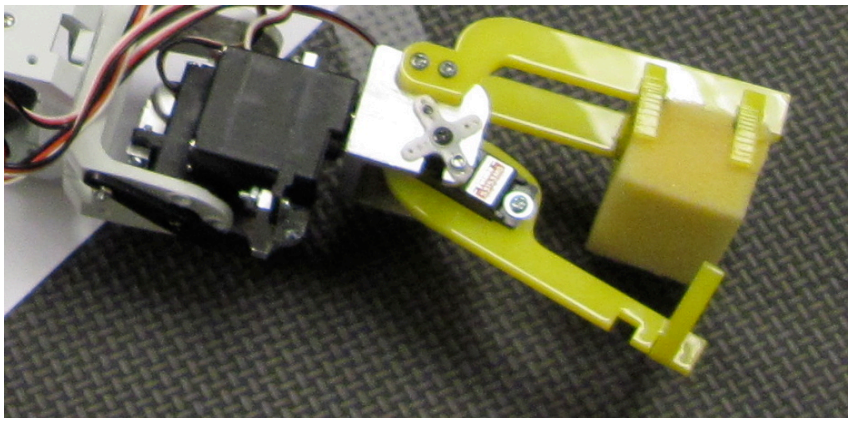

Figure 21 - First gripper with a palm + thumb design

This gripper proved valuable when prototyping the Grasper functionality. However, it had two noticeable flaws. First, the amount of space allowed between the foam of the palm and the arc of the thumb was too small for the level of accuracy required when placing the arm above a chess piece. This led to small errors in perceived piece location causing the gripper to knock over the chess piece when pitching down the wrist, or not obtaining a strong enough grip on pieces due to the thumb's closing trajectory. Second, the amount of room between the end of the gripper and the end of the gripper servo was too small to obtain a solid grip on larger chess pieces, such as kings and queens, even

if the gripper was perfectly accurate when pitching down. Unfortunately, the length of the gripper could not be increased without being too long to remain above the table when pitching down, even with the robot in its tallest stance. These lessons imparted valuable information for the second gripper design.

To increase the amount of space between the grasping surfaces, a two-fingered gripper was chosen for the second gripper. A two-fingered gripper allowed for the gripping surfaces to cover a larger area when closing around a piece to make up for inaccuracies in arm placement. To increase the amount of room between the end of the gripper and the gripper servos, the wrist-roll degree of freedom was removed and the gripper servos' rotational axes were placed above the working plane of the arm. The fingers were based on the first design's thumb, but fitted with foam.

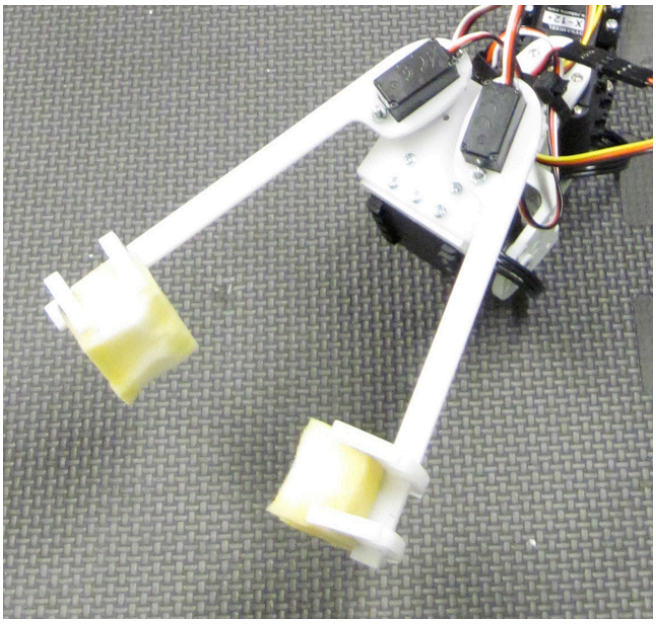

Figure 22 - New two-fingered gripper

This gripper robustly compensated for moderate inaccuracies when picking up pieces and was capable of obtaining strong holds on every piece type by grasping the

pieces close to their bases. It consisted of one AX-12+ for wrist pitch and two Futaba servos for finger open/close. The foam on the two fingers was trimmed to minimize the required width of the fingers when approaching a piece.

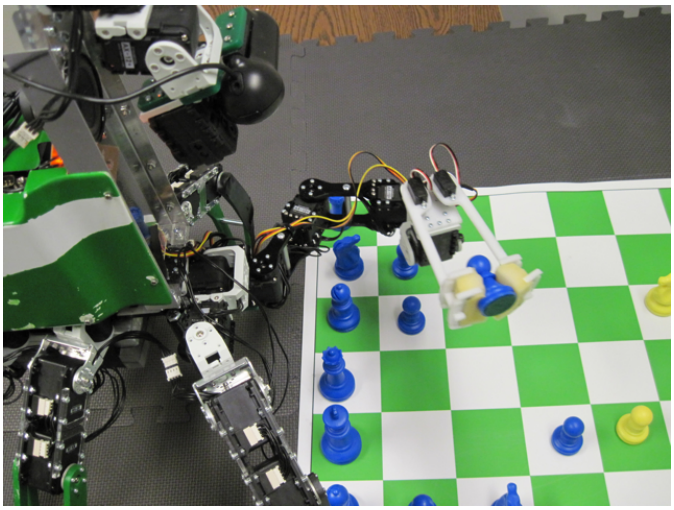

Figure 23 - Grasping a piece over the board

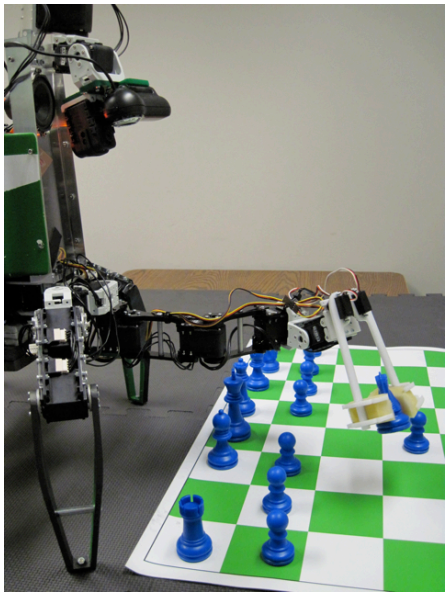

Figure 24 – Placing a piece on the board

To minimize the chances of knocking over pieces on the board, calls made to the Grasper requested the gripper to be placed at a 45-degree angle to the chessboard. If a target location was surrounded by pieces, coming in at a 45-degree angle utilized the most open space to pick up or drop off a piece. If no arm configuration could reach the target location with a 45-degree angle, then angles deviating from 45-degrees were considered. In the worst case, 0 or 90 degree angles were accepted, but only after all other attempted angles failed.

3.3 Moving Around the Board

In order to overcome the Chiara's limited arm reach, the robot needed to move around the board to get the arm into positions where it could successfully grasp or release pieces. To minimize the total amount of time the robot took to complete moves,

the planner attempted to find optimal positions for the robot to manipulate as many objects as possible before needing to reposition. The technique for finding these locations used sketches in the dual-coding vision system's world space. The annulus formed around a target location from all locations larger than the arm's minimum reaching distance and smaller than the arm's maximum reaching distance specified the set of locations that the robot could be in and still reach the target location. Taking the intersection of all annuli from each target location and removing all positions too close to the chessboard yielded collections of areas in world space from which the robot could reach all target locations. Figure 25 shows the piece location as a green dot and its annulus of reach-ability. The robot could be at any location within that annulus and some configuration could place the arm over the piece. Some of these locations placed the robot in collision with the chessboard. Figure 26 shows the set of these locations as a blue square. Figure 27 introduces the destination location (red dot) and its annulus of reach-ability (red ring). Taking the intersection of the two annuli and removing anything in the blue square yielded all viable locations to reach both the piece and its destination in tan, as seen in Figure 28.

Figure 25 - Annulus of piece reach-ability

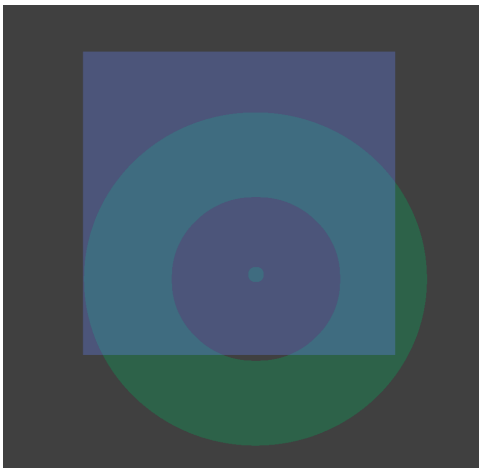

Figure 26 - Invalid locations over annulus of reach-ability

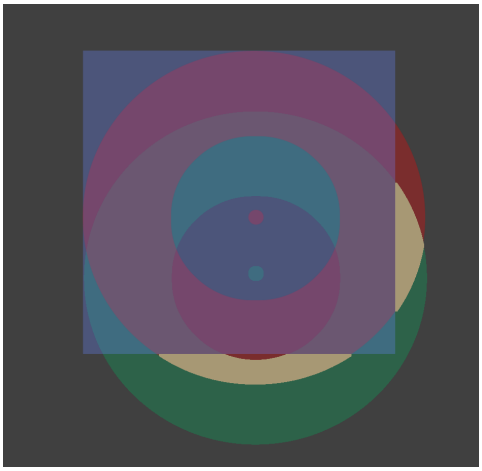

Figure 27 - Intersection of piece and destination

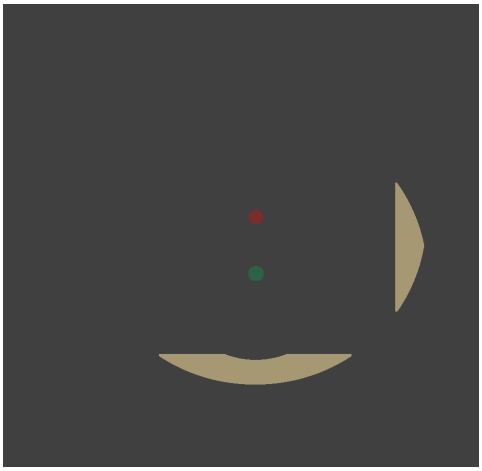

Figure 28 - The remaining viable positions (tan)

However, some chess moves required moving pieces to positions that were well out of range of their original positions. If the intersection of the annuli resulted in no viable positions, then the motion planner split the multiple manipulation actions into single actions. This split will be discussed in the section on interleaving the Pilot and the Grasper.

Once the robot's desired location was planned, a Pilot request was made to move the robot to that location. Given imperfections in the robot's locomotion and outside factors such as slipping on the table, the robot did not end up precisely where

requested. To compensate for these errors, the robot re-localized after every motion through a different localization technique. Since all calculated robot locations were close to and facing an edge of the board, the robot could look directly in front of itself and parse information from its view of the board. The robot based its location from parsing out the lines of the board using the same techniques as when extracting board squares.

From the collections of lines, the robot took the intersection of the lowest horizontal line and the vertical line closest to the middle of the camera frame to compute a reference point on the board. Care was taken to ensure that the lowest horizontal line was the board edge and not a line in the middle of the board. Given this intersection point in the camera frame, the board squares immediately to the left and right of this point were sampled to see which was green and which was not. This narrowed down which vertical board line the robot was looking at when projecting that point from camera space to world space.

Given the approximate location of the robot, a reference point in world space, and whether the square to the left of that point was green, the reference point could be at one of only four places on the board. Assuming that the robot's error for movement was not off by more than the width of a board square, taking the board location candidate closest to the robot's intended position determined the robot's actual location. Furthermore, the robot's orientation could be determined from the extracted horizontal line. If localization determined that the robot was too far off from its intended location, it would make another Pilot request to adjust for this error and get the robot to the correct spot. In Figure 29, the green areas are the extracted green image pixels, the pink lines

are the extracted horizontal and vertical lines to use, and the large pink dot at the lines' intersection is the extracted reference point. The algorithm would determine that the left square is green, and localize off of the projection of the pink dot into world space for the nearest border point whose left square is green.

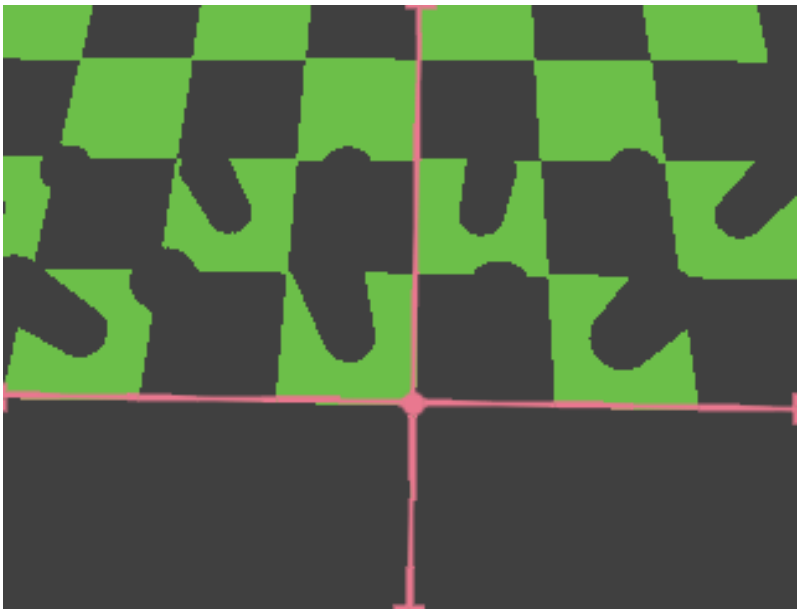

Figure 29 - Localization from extracted board lines (pink)

3.4 Approaching the Board

Unfortunately, the Chiara's six-legged walk did not take world obstacles into account. With a target location close to the board, the walk engine might place one of the front legs into the board area while walking. As this would disrupt the board state, an approach behavior was implemented. To get the Chiara into a position close to the board without affecting the board state, custom locomotion was done by shifting the walk engine's center location, rocking the body forward. The robot then would lift its legs off the ground to rest on its belly, shift its legs forward, place its legs back down to stand up, and then rock the body forward while keeping the leg placements stationary. This

successfully placed the robot at locations close to the chessboard without disturbing any pieces. This behavior also ensured that the legs would be in a stable configuration for standing up during manipulations. Without a stable configuration, the robot could lose its balance and fall onto the board.

3.5 Detecting Off-center Pieces

Since chess pieces were not guaranteed to be in the center of each board square, each piece needed to be visually located before attempting to manipulate it. To get an accurate measurement, the robot would look directly at the square the target piece was in. This placed the piece near the center of the camera frame. Detecting the piece's approximate location was done with the same smile detection heuristic used when determining the opponent's move. The centers of each perceived chess piece were then projected into local coordinates. The piece whose perceived location was closest to the estimated location of the target piece was taken as the appropriate piece to manipulate. To get the precise location of the center of this piece, the bottom edge of the piece was determined from the under-pixel closest to the bottom of the camera frame. The extracted point was projected into local space, and an offset of magnitude equal to the radius of the chess piece was added in the direction away from the camera. This approximated the center of the piece, and its location on the world map was updated. The visual detection of off-center pieces proved to robustly aided manipulation actions in picking up pieces, correcting previous perceptual errors.

3.6 Manipulation Planner: Interleaving the Pilot with the Grasper

With techniques for motion and manipulation planning in place, a higher-level planner was needed to decompose chess moves from the chess engine into a series of Pilot and Grasper requests. If the chosen move was only moving a piece from one square to another, then the manipulation was fairly straightforward. If the piece being moved and its target position were both reachable from a single location, the robot would first be instructed to move to that location. Once in position, the relative locations of the piece and its destination position would be calculated and the Grasper would perform a moveTo operation. After successful manipulation, the robot was then instructed to return to the home position on its side of the board. If the piece and its target position were too far away to be reachable from one location, the planner decomposed the action into more steps. First the robot would need to get in position to pick up the piece, and then perform the grasp action. With the piece held in the gripper, the robot would then move into a position where it could reach the target location. Once in position, the Grasper would perform a release action, placing the piece in its desired location. Lastly, the robot would return to its home position.

Performing a capture required more manipulation planning than moving a single piece. To perform a capture, the planner created actions for moving the captured piece out of its square to a nearby unoccupied square, moving the capturing piece onto the destination square, and then reacquiring the captured piece to remove it from the board. Assuming both pieces were in reach, these three manipulations required keeping track of three different board positions. If both pieces were not simultaneously in reach, then the capturing piece would be picked up from its current location and dropped off in a

separate empty square within reach of the captured piece. From here, the same process of moving the captured piece off of its square, moving the capturing piece onto the square, and removing the captured piece took place. These techniques effectively covered all cases of captures during game play.

3.6.1 Adjusting on failure

Once the robot was in position to perform a manipulation, the planning of arm trajectories would occasionally fail. If the target piece was too close to one of the forward legs, the Grasper would not find a position for the arm that would prevent the gripper and the leg from colliding. If the robot did not get close enough to its computed position, a target piece might be out of reach. If any of these cases occurred, the failure condition was reported to the higher-level planner, and fallback plans were executed

If the Grasper failed to perform the moveTo operation of picking up and dropping off a piece from a single position, then the operation would be decomposed into two Grasper actions. The first operation was getting in position and grasping the piece, the second was getting into position and releasing the piece. With only one location to worry about, the manipulation planner could reliably find a position from which the robot was able to reach the target. If the Grasper failed to perform a grasp or release action, then the target location was most likely out of reach. If the location was too far away, the planner would generate a new target position slightly closer to the target location in order to compensate for robot misplacement. Similarly, if the location was too close, the planner would generate a new target position slightly farther from the target location. However, if the target location was within range and the Grasper was unable to succeed

due to an over-constrained environment, the planner would slightly perturb its current target location. This did not guarantee a problem fix, but a target location that was within range and computed from the motion planner should have succeeded. Slightly adjusting the position gave the Grasper another chance to perform its action with slightly different data.

IV. Results

1. AAAI Competition

The robot successfully competed in the AAAI-2010 Small-Scale Manipulation Challenge. It was able to make legal, competitive chess moves through ten moves on each side with minimal human assistance. Unfortunately, due to time constraints, certain shortcuts were needed. Since the Chiara can only reach over four rows of the chessboard, the motion planning generated positions for the robot to walk around the sides of the board in order to reach the other half. Due to inaccuracies in the robot's walking performance and time constraints for debugging, walking around the board needed to be curtailed. This meant that no move that required manipulation on the half of the board furthest from the robot's playing side could be executed. Curtailing movements was accomplished by overriding the chess engine when it decided to make such a move. When that happened, the chess engine was commanded to undo the move it had decided upon, and a pawn on the robot's playing side was chosen to move up a single square. The chess engine's internal representation was updated accordingly and the action was executed. For the scope of the Manipulation Challenge, this style of

move was not considered a blunder, as further moves in the game would lead the chess engine to decide on other intelligent moves.

Some failures did occur, though. After a leg servo overheated and shut off, it became apparent that the system was not tolerant to hardware failures. This failure resulted in poor walking accuracy and unstable leg positioning for standing. The program needed to be restarted, and then updated with the current board state, to overcome this issue.

The robot was able to reliably pick up every chess piece except for knights. The knight pieces are unique in that they are not concentrically symmetric due to the horse's head. On occasion, the robot would attempt to pick up the knight along the sides of the head where the grasping surface is thin and flat. The foam of the gripper would enforce a friction grip for some time, but the piece would eventually slip out of the grasp. If the robot took a different approach angle to the knights, it would perform the manipulations with ease.

Despite these shortcuts and isolated failures, the robot was able to complete more than 90% of its attempted manipulations and chess moves. It was a competitive player in the challenge despite having certain handicaps in comparison to its robotic opponents, some of which had overhead cameras, which eliminated the occlusion problem, and some of where were large, fixed arms with no mobility. The vision system never misperceived an opponent's move and was able to overcome all occlusion issues. The chess engine made competitive opening moves and chose multiple opportune captures. The manipulation system rarely missed picking up a piece when all hardware

was working properly, and it was able to reliably place pieces within an inch of their desired location.

2. Grasper Use on Chiara As Well As HandEye in Simulation

The rewritten Grasper successfully executed both 3D and 2D manipulations. It lent itself to easily specifying 3D manipulations of chess pieces. Furthermore, the same code running on a planar three-link arm system called the HandEye was able to perform 2D manipulations of playing tic-tac-toe in simulation [10]. This shows that the Grasper was written at an appropriate abstraction level to handle either dimensional case.

V. Conclusions

The Grasper extensions learned through this project could easily be applied to other games, such as checkers, go, or backgammon. Playing chess introduced occlusions and had a crowded board that the other games do not have, forcing the Grasper to solve harder problems. With the work of this project, implementing a system to play any of these other games would be much easier.

The new servos on the delta series Chiara cannot be justified until determining their life expectancy. For four times the cost of the cheaper AX-12+ servos, the RX-28 servos need to prove that their lifespan and reliability merit the investment. That being said, it is clear that the motions provided by the RX-28 servos are smoother and appear stronger than motions provided by the AX-12+ servos. Continual use of the delta series Chiara should validate the RX-28 servo's cost.

Writing a new line extractor was necessary in order to reliably extract the kinds of lines seen on the chessboard. The other line extractors in Tekkotsu did not reliably extract lines of single pixel-width. Furthermore, many assumptions were exploited from the previous knowledge that the perceived lines would form a uniform grid. Even though the new line extractor was developed for a particular feature set, it could be used in work toward a general method of extracting regular structures.

Since three-dimensional manipulation was completed with a vertical approach to pieces, arm path planning was reduced to a 2D problem. This was necessary because Tekkotsu's RRT implementation only worked in a 2D world. For true three-dimensional manipulation, a full 3D RRT implementation could be used to reach underneath archways and around obstacles with different footprints at different heights.

The higher-level planner for interleaving Pilot and Grasper requests highlights the need for a general higher-level planner in Tekkotsu. I wrote my own planner specifically for playing chess, but a more general planner could have handled playing chess and been extensible to other tasks without requiring a user to write his or her own. The work in this project brings Tekkotsu out of the plane, but not into a full-fledged 3D capable framework.

VI. Future Work

There are many opportunities for further work with this project. Currently, the Grasper can only handle one technique for 3D manipulation using a single gripper. A more sophisticated manipulation algorithm could determine the appropriate grasping

technique to complete a task on the fly when given the gripper, hardware capabilities of the robot, and object being manipulated.

To play a full game of chess, the system needs to be able to walk around the sides of the board to reach the other half. Given a sufficient amount of time, this could be incorporated into the existing codebase.

Using color segmentation to identify where chess pieces are in camera images works well, but being able to identify particular pieces in the image would allow for much easier perception of the opposing player's move and for detecting errors. A more sophisticated pattern recognition algorithm could be incorporated and allow for easier perception. In order to perceive promotions correctly, some form of piece detection would be necessary. As the system stands now, it has no way of knowing whether an opposing pawn would promote to a queen, knight, or bishop. Visually identifying the piece would be necessary to determine which promotion occurred.

Being able to play chess opens up possibilities for having the robot play other board games. Checkers, Connect Four, and backgammon could each be played using techniques explored in this project.

VII. Acknowledgments

I would like to thank Professor Dave Touretzky for advising me throughout this project and spearheading the new gripper and robot design; Ethan Tira-Thompson for his help with using and fixing Tekkotsu; Glenn Nickens for his work on the 2D version of the Grasper; and the design team of Wayne Chung, Nathanial Paffett-Lugassy, and

Federico Rios for creating the delta series Chiara. This work was supported in part by National Science Foundation award DUE-0717705 to David S. Touretzky.

References

[1] E. Olson, “AprilTags,” http://april.eecs.umich.edu/, accessed May 2010

[2] “GNU Chess,” GNU Operating System, http://www.gnu.org/software/chess/, accessed April 2010

[3] J. Kuffner and S. LaValle, *RRT-Connect: An Efficient Approach to Single-Query Path Planning*. 2000

[4] “Tekkotsu,” Tekkotsu Lab, http://www.tekkotsu.org, accessed May 2010

[5] D. S. Touretzky, “Chiara Robot,” http://chiara-robot.org/, accessed May 2010

[6] D. S. Touretzky, N. S. Halelamien, E. J. Tira-Thompson, J. J. Wales, and K. Usui, *Dual-coding representations for robot vision programming in Tekkotsu*, Auton Robot (2007) 22:425-435

[7] D. S. Touretzky, E. J. Tira-Thompson, “The Tekkotsu Crew: Teaching Robot Programming at a Higher Level,” AAAI-2010. July 13, 2010. Atlanta, GA

[8] Wikipedia, “Chess Piece,” http://en.wikipedia.org/wiki/Chess_piece, accessed July 2010

[9] Wikipedia, “Homography,” http://en.wikipedia.org/wiki/Homography, accessed May 2010

[10] W. Winston and C. Freeman, “Tic-Tac-Toe,” http://www.andrew.cmu.edu/user/wwan1/15-494%20website/index.html, accessed July 2010