Sense, Plan, Triple Jump

Daniel J. Brooks, Eric McCann, Jordan Allspaw, Mikhail Medvedev, and Holly A. Yanco
Department of Computer Science, University of Massachusetts Lowell, Lowell, Massachusetts 01854
{dbrooks, emccann, jallspaw, mmedvede, holly} @ cs.uml.edu

***Abstract*—In the field of human-robot interaction, collaborative and/or adversarial game play can be used as a testbed to evaluate theories and hypotheses in areas such as resolving problems with another agent's work and turn-taking etiquette. It is often the case that such interactions are encumbered by constraints made to allow the robot to function. This may affect interactions by impeding a participant's generalization of their interaction with the robot to similar previous interactions they have had with people. We present a checkers playing system that, with minimal constraints, can play checkers with a human, even crowning the human's kings by placing a piece atop the appropriate checker. Our board and pieces were purchased online, and only required the addition of colored stickers on the checkers to contrast them with the board. This paper describes our system design and evaluates its performance and accuracy by playing games with twelve human players.**

I. Introduction

This paper describes a checkers playing robot which aims to be capable of playing the game of checkers against a human opponent. The creation of a robot capable of playing a board game is not a new or novel concept. Some people trivialize the idea, classifying the concept as being more suitable for a class project than a subject for research, a thing you might build using legos during your spare time. This perspective is not unreasonable or surprising; others have successfully created chess and checkers playing robots long before us. However, these systems often struggle to perform the basic manipulation tasks needed to play the game while not even attempting to perform advanced tasks such as stacking pieces. Furthermore, they are subject to many limitations (e.g. strict lighting requirements, fixed board position, and nonstandard boards and pieces) [1], [2], [3] which can create a very rigid and unnatural playing environment.

Simplified systems frequently suffer from problems with perceiving the state of the game. Lewis and Bailey write that their robot chooses its next move after seeing that there are the same number of enemy pieces on the board, and one has moved. This puts the game at risk of illegal moves which could be exploited in the form of cheating, such as transforming pawns into kings [2]. Bem specifically described his system as unable to recognize special situations, "especially cheating from the human side" [3]. Illegal moves violate what game theorists such as Huizinga refer to as the "magic circle," in the confines of which the rules are supposed to be binding. Without boundaries to the game, players' engagement and enjoyment will suffer due to lack of competition [4]. Without adequate perception, the game is much less compelling.

The inability to manipulate normal checkers pieces combined with the presence of intrusive infrastructure surrounding or attached to the game board can add a distinct reminder that the opponent is non-human. For example, one cantilever arm design reached over the board and lifted steel pieces with an electromagnet. The stepper motors controlling the positioning of the arm did not allow for the robot to flip a pawn into a king, nor did the electromagnet have the strength required to lift two pieces simultaneously. Therefore, this robot required the human to flip the game pieces to create kings for both players [2]. To provide a compelling interaction, the robot needs the ability to manipulate the game pieces in every scenario according to the rules of the game, and to do so using using standard game components.

In contrast, the Gambit chess playing robot [5] represents a robust system that shares many of our same design goals such as using an unconstrained game board, unmodified playing pieces, and intelligent game state reasoning. While our system is much more closely related to Gambit than to any of the checkers robots described, the perception and manipulation challenges are quite different. Chess has more types of pieces than checkers which must be distinguished by the system. However, the locations of these pieces are easy to determine using point cloud processing and are easy to manipulate due to their fairly large nature. Checkers are all regular in shape and differ only in color. However, they are also short and therefore hard to "see" in point clouds, which also makes it difficult to distinguish between kings and pawns. Their short height also makes them difficult to manipulate, a problem compounded by the fact that sometimes they must be moved as stacks.

II. System Overview

We have designed a checkers playing system using a Rethink Robotics' Baxter research platform [6] that strives to eliminate the shortcomings of these previously mentioned systems. A redundant and adaptive checker detection system eliminates the need for special lighting conditions or a fixed-position game board. Baxter's two 7 degree of freedom (DOF) arms with custom manipulators eliminate the need for special playing pieces and allow us to achieve previously unobtainable actions such as stacking and moving "kings". Finally, a sophisticated perception system provides a framework for performing human-robot interaction (HRI) studies, setting the stage for future work that will try to differentiate not only between legal and illegal moves, but also whether the human player was attempting to cheat or had simply made a mistake, then handling the different situations appropriately.

A. Platform and Environment

Baxter is a low cost, two armed, stationary robot intended for performing manufacturing tasks. Our system was designed so that a person could play a game of checkers with the robot using a generic checkers board game. Rather then using an overhead camera that would have required a special rig, we outfitted the robot with an Asus Xtion 3D camera system which

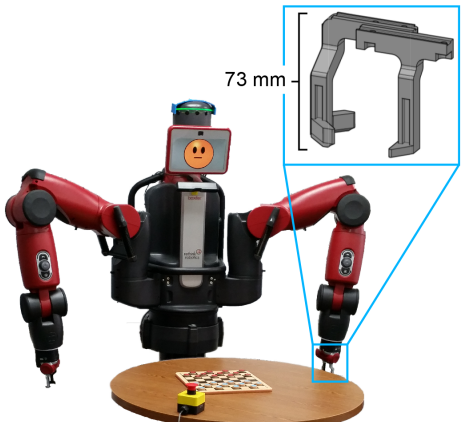

Fig. 1. Baxter Robot with Checkers Board and expanded rendering of our gripper design with the "Claw" on the left and the "Tooth" on the right.

we used as our primary sensor for looking at the board. The only modifications made to the board game were the addition of colored stickers that we placed on the checkers to give them a different color scheme than the board. The game was placed on a table positioned in front of the robot as shown in Fig. 1.

Baxter was designed to be compliant; it can safely operate with people inside its work area without the risk of causing injuries. This was an important feature since anyone playing with the robot would be well withing the robot's reach. The robot made use of both arms to provide a maximum range of motion, each of which had a wide angle color camera, distance sensor, and linear actuator. At the end of the robot's turn, it would draw both arms away from the board into a lowered, non-threatening position. The robot also features an articulated 25.4 cm screen with a built-in camera, usually used to portray the robot as having a face in commercial models. We used this screen to draw several different facial expressions which we paired with actuation such as head nodding and a text-to-speech system to provide audio and visual cues about whose turn the robot thinks it is.

B. Gripper Design

Each of Baxter's arms comes mounted with a linear actuator and stock plastic paddles for grasping objects. Although the paddles could grasp a single checker, we struggled to grasp and hold two stacked checkers as the plastic would bow slightly around the top checker. The stock paddles' size also created problems when working with the confined space between checkers on a regular sized game board. We therefore decided to create custom grippers built out of an acrylic photo-polymer and constructed using 3D printing technology.

Our grippers (Fig. 1) are the result of several iterations of designs and testing, and feature several improvements over the stock paddles while retaining the aesthetics used by Rethink Robotics. The paddles of the stock grippers were designed to be symmetric, with the grasping point in the middle of the arm's major axis. While an excellent design for general use, the positioning was not ideal for visual servoing (further discussed in Section V-B) since grasped checkers were not entirely visible in the camera frame due to its position. We shifted the grasping point to be located as close to directly below the camera as possible without obscuring the distance sensor. Next, we replaced the paddles with a V shaped "claw" to grasp from two points on one side of the checker and on a triangular "tooth" which acted a single point of contact on the other side. This firmly grasped a range of different sized checkers (22-25mm diameter) using the smallest footprint possible to reduce the likelihood of bumping other pieces on the board during manipulation. The claw had a beveled bottom so that as it descended for grasping it would push the checker into position. The flat face of the tooth was wide enough to prevent checkers from becoming pinched between one of the tips of the claw. For grasping stacks of checkers, the sides of the claw were also tall enough to prevent the top checker from falling out, and the tooth was made slightly smaller then the average checker height to prevent grasping problems.

C. Software Architecture

Baxter's software interface was designed using ROS [7]. This allowed us to link our custom control software directly with the robot and also take advantage of other open source robotics software packages published by the ROS community. Our architecture could be broken into three major categories - *Sensing*, *Perception*, and *Manipulation* (see Fig. 2). The modules in the *sensing* category used 3D point cloud analysis and computer vision techniques to find the game board and checkers in the world. This information was then passed on to our *perception* pipeline which tracked objects, assigned meaning based on the objects' relative locations, and interpreted that meaning as a game of checkers. Finally, the results of the perception were sent to our manipulation modules which planed the robot's movements and manipulated game pieces using visual servoing.

III. Sensing

The primary sensor used in our system is an Asus Xtion 3D camera which we used to detect the playing board and checker pieces using a combination of computer vision and point cloud processing techniques. We made no assumptions about the location or positioning of the game board or checkers pieces other than that everything would be within the robot's reach, correctly set up at the beginning of the game, and reasonably oriented (e.g. the board not turned ambiguously sideways). Therefore, the first problem we needed to solve was identifying where the board was located.

A. Board Detection

Knowing the position and orientation of the game board was crucial for assigning meaning to the locations of the checkers on the table and describing their positions in a board relative coordinate space. The board was initially located using OpenCV to process the camera's RGB image. Hough lines were fit to Canny edges found in the image, which were then searched for quadrilaterals. If a quadrilateral was found, the intersections of those lines represented the corners of the board. We then calculated the real world position of the board corners using the 3D point information that corresponded with the image pixel locations for the corners. If no quadrilateral was found or if the intersections did not have feasible values, the board was inferred to be occluded.

The real-world board corner positions were used to create a coordinate transform between the robot and the board such that future calculations could be written relative to the playing board. This allowed us to compute a square, cropped, top down

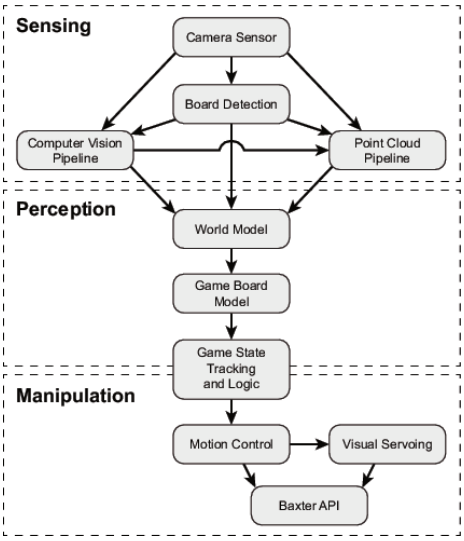

Fig. 2. Software system overview, showing the flow of information.

view perspective image and corresponding point cloud of the playing board which we used to speed up the detection of checkers on the board. Additionally, we were able to use the transformation to calculate the approximate locations for the centers of the squares on the board.

B. Checker Detection

We took advantage of both computer vision and point cloud processing techniques to identify the locations, sizes, and colors of the checkers. The only assumptions we made about the checkers being used were that they were circular, reasonably sized (2cm to 2.5cm diameter), different colors than the board, and not deformable. This meant we needed to identify not only where the checkers were, but also what they looked like.

We leveraged OpenCV to find the locations and colors of checkers on the playing board. Hough Circle detection was applied to the perspective board image provided by the board detection to locate possible positions of checkers. A weighted history consisting only of pixels inside the detected circles was maintained to reduce noise. Watershed was then applied to the weighted history to detect closed regions, which we considered to be the most likely locations of checkers. Frequency counts of the hue of pixels inside the detected checkers were then used to approximate the colors of the playing pieces being used. This process proved very fast and effective; however it did not allow us to distinguish between kings and pawns, could not find checkers located off the board, and would occasionally miss a piece due to light glare.

To resolve these shortcomings, we made use of the 3D point cloud information provided by the camera. This information was processed using the Point Cloud Library (PCL) [8] at a much slower rate then our computer vision detector. A voxel filter was used to reduce the density of the information being processed, and plane segmentation was used to eliminate background noise such as arms resting on the table. We then used a Euclidean clustering algorithm to identify checkers which could then be matched to the colors produced by the computer vision to distinguish between players pieces. Clusters could be identified as being a king if they contained points at a height only achieved by stacking two checkers. We used an alternative clustering method called color based region growing to identify the locations of checkers not on the board. Information from these two detection methods were then combined and presented as a single source of information describing the most recent estimate of each checker's position, color, and king status.

IV. Perception and Logic

After detecting the board and checkers, the information needed to be interpreted as a meaningful representations of the game of checkers. The first step in this process was to construct a *world model* which could track and smooth the locations of the game board and checkers pieces, which we were able to do using the ROS-enabled WIRE [9] software. WIRE tracked objects' positions as well the checkers' color (as a discrete value referred to as red, blue, or unknown) and whether each checker was a pawn, king, undetermined.

With WIRE tracking all the physical components of the game in the *world model*, the next step was to build associations between individual checkers objects and the squares marked on the game board. Each checker in the *world model* was classified as being clearly located on a particular square, on the board but ambiguously placed, or not on the board. Information about the color of each checker and whether it was a stack was then used to assign state to individual squares, unless the checker's position was ambiguous in which case it was added to an "unplaced checkers" list. We called this reorganization of semantic information a *game board model*.

A. Game Board Modeling

The processes of generating a *game board model* was based on fulfilling two constraints on which the game state logic (discussed in Section IV-B) relied. The first was that each square on the playing board held at most one checker (or stack of checkers). The second constraint was that this representation was supposed to be very conservative compared to a human's interpretation, always erring on the side of being incomplete rather then incorrect. These constraints resulted in the following four steps:

1) Filtering overlapping objects reported by the *world model*,
2) Discretizing checker locations into unique squares,
3) Assigning state to squares based on its checker type, and
4) Filtering/censoring potentially unreliable results.

The *world model* would occasionally report multiple overlapping objects, especially whenever attributes would change such as when a pawn became a king. In the case of this example, we would keep the king while discarding the pawn. Overlapping objects with different hues were considered ambiguous and were discarded. Otherwise, if all the objects had matching attributes they were combined into a single object.

The next step was to associate each checker object with the square on the game board that it was sitting on, a problem otherwise known as weighted bipartite matching. All the checkers objects were initially all marked as "Unassigned" and the squares were marked as "Free". The process of discretizing checker locations iteratively matched each checker to the center of the nearest square, and each square to the nearest checker, creating a set of bi-directional matches as shown in Algorithm 1. Each iteration, checker-square pairs that marked each other as closest and whose distance apart was less then half the width of a square were removed from

Algorithm 1 Checker to Square Matching Algorithm

Inputs: Lists of all Checkers (C) and Squares (S) as 3-tuples with location (L), a matched object (M), and status (R).

Output: Checker ($c \in C$) and Square ($s \in S$) objects which have been matched to each other ($c_M = s \Leftrightarrow s_M = c$), unoccupied squares ($s_R =$ Free), and ambiguous or off board checkers ($c_R =$Unplaced).

$C = \{\langle L, M, R\rangle, ...\}$
$S = \{\langle L, M, R\rangle, ...\}$
$C' = \{\forall k \in C : k_R = \text{Unassigned}\}$
$S' = \{\forall k \in S : k_R = \text{Free}\}$
while $\exists k \in C'$ **do**
  for each $c \in C'$ **do**
    for each $s \in S'$ **do**
      if dist$(c_L, s_L) <$ dist$(c_L, [c_M]_L)$ **then**
        $c_M \leftarrow s$
      if dist$(s_L, c_L) <$ dist$(s_L, [s_M]_L)$ **then**
        $s_M \leftarrow c$
  for each $s \in S' : \exists s_M$ **do**
    $c \leftarrow s_M$
    if $s = c_M$ **then**
      if dist$(c_L, s_L) <$ width(s)/2 **then**
        $c_R \leftarrow$ Placed
        $s_R \leftarrow$ Occupied
      else
        $c_R \leftarrow$ Unplaced

the unplaced checkers and free squares pools by changing their statuses to "Placed" and "Occupied", respectively. If the threshold was exceeded the checker's status was changed to "Unplaced" while the square remained "Free". The process repeated until all the checkers had been marked as either "Placed" or "Unplaced".

In the third step, every square containing a checker was given a state based on the checker's attributes. A square could have seven possible states. Squares that had been marked as `Free` during the previous step simply kept their designation. "Occupied" squares were reassigned as `Human Pawn`, `Human King`, `Human Unknown`, `Robot Pawn`, `Robot King`, or `Robot Unknown` according to the associated checker object's color and height attributes. Which color pieces belonged to which player was determined at the beginning of each game by checking the colors of the checkers on each side of the board just prior to the first move, with the color closest to the robot being the robots color. The colors were remembered, and used to identify each player's pieces the remainder of the game.

Although the *world model* was usually very reliable, sensing inaccuracies and the non-instantaneous convergence time of its filters resulted in periods of low reliability that would propagate through the rest of the system if left unchecked. Therefore, the final step in generating *game board models* was to evaluate every model for potentially unreliable results, as shown in Algorithm 2. This was accomplished by tracking the state history of every square on the board in individual first-in first-out (FIFO) queues, which were used to compute each square's "consistency". Consistency was computed as the number of times the queue's mode value occurred divided by the queue's length. A square was declared "stable" if the consistency was greater then an acceptThreshold, or "unstable" if the consistency dropped below a rejectThreshold. *Game board models* were considered reliable if the most recent state value for every "stable" square matched its historical mode. All model values were recorded for evaluating future results, however only reliable models were reported to the game logic. Although it was not required for all the squares to be marked as "stable" for a *game board model* to be considered reliable, unstable squares were changed to look like they were free and their checkers were added to the "unplaced" list just before the model was reported.

B. Game State Logic and Tracking

The game logic was responsible for tracking the progress and state of the game though the use of a finite state automata called the Checkers State Machine (CSM). The CSM defined the structure of each players turn (see Fig. 3), tracked which player's turn it was, and provided contextual information for evaluating perceived game board models.

The Check-Game State (CGS) divided the CSM into two sets of five states which defined each players turn. A persistent reference to a stripped down version of the Raven Checkers game engine was stored inside the CGS and used to track the definitive state of the game. Transitioning between player's turns could only occur when the CGS received a valid game board model that it could use to update the game engine. The rules of checkers allowed us to deterministically calculate the set of all valid changes which could be made to the game at any point in time. Combining this knowledge with the conservative nature of our *game board models* allowed us to confidently accept or reject perceived models that were marked as highly stable. Less stable models could be compared with known legal moves to distinguish between sensing discrepancies and invalid changes to the game.

Each player's turn began with the *initialization state*. During the human's turn this state was used simply to inform the player that it was their turn, while during the robot's turn it calculated the optimal move to make by analyzing the game state. Adversarial game engines often accomplish this using search algorithms such as alpha-beta and its relatives minimax and negamax. These algorithms weigh the benefits of possible next moves in terms of moves the opponent could make in successive turns.

The *initialization state* was followed by the *action state*. During the human player's turn this was characterized by waiting for the person to make their move (which was detected

Algorithm 2 Model Reliability Assessment

procedure ISRELIABLE(gameboardmodel)
  reliable ← True
  for all *square* in *gameboardmodel* **do**
    history ← GETSQUAREHISTORY(*square*)
    expValue ← MODE(*history*)
    consistency ← *history*.count(*expValue*)/*history*.length
    if *consistency* > *acceptThreshold* **then**
      square.isStable ← True
    else if *consistency* < *rejectThreshold* **then**
      square.isStable ← False
    if *square*.isStable & *expValue* ! = *square.curr* **then**
      reliable ← False
  return *reliable*

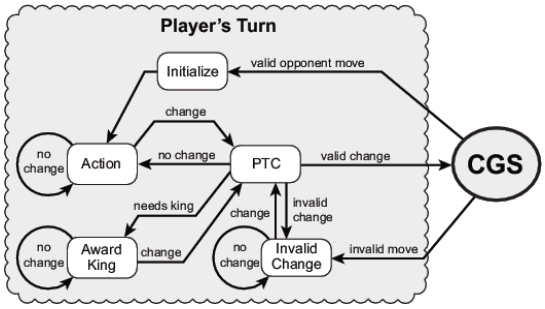

Fig. 3. A single player's side of the CSM.

by changes to the *game board model*). On the robot's turn, the system would determine the necessary steps to perform the strategy it had calculated during initialization and execute them one at a time.

Anytime a change was detected in the perceived *game board model* the currently executing state would jump to the *Pre-Transition Check State* (or PTC) for analysis and determining what should happen next. If the *game board model* represented an invalid configuration but also had a list of unplaced checkers, the PTC would try using context information to identify this as a valid move. This was done under the assumption that the stable components of the model were perfectly accurate, while disparities between "free" squares and any valid configurations might be resolved by disambiguating the locations of unplaced checkers. The PTC would then attempt to update the *game board model* to match a valid configuration by considering the location of each unplaced checker, whose turn it was, the state of the game at the end of the last player's turn, and what legal moves could be made during the current turn. If it was determined that a change was falsely detected due to sensing errors, control was returned to the *action* state. If the board had been changed and the change was identified as the result of a legal move, the *game board model* was submitted to the CGS to transition to the next player's turn. Otherwise, if an illegal move had occurred, control was shifted to the *invalid change* state.

In addition to the PTC finding no change, a legal change, or an illegal change, a fourth scenario was possible. When a player's pawn became a king, the opposing player was obligated to "King" the piece by stacking a second checker on the pawn, thus "crowning" it. This was a special scenario since normally each player's turn could not be completed until the board was in a valid configuration. Furthermore, the next player was normally prohibited from making changes to the board until the current player's turn was finished. However, in this case the current player would have finished their move and waited for their opponent to make the final modification that would result in the valid board configuration needed to end their turn. This special scenario was handled by the *Award King State*. During the human player's turn, the *Award King State* would permit the robot to king the pawn. To do this, the robot would first grasp one of the human player's pawns that had been removed from the game, and place it on top of the piece to be crowned. During the robot's turn, the system solicited the human to crown its pawn using audio and visual prompts. The results of the crowning process would then be detected as a change to the *game board model*, and control would transfer back to the PTC.

The board could be detected as having an illegal change made to it for a number of reasons, and the *illegal change state* was implemented to handle these situations. The first possibility is that there is actually nothing wrong with the board, but rather that despite our best efforts the most recent *game board model* did not accurately reflect the real world. In most cases, this would resolve itself after an updated *game board model* arrived and control was transitioned back to the PTC for evaluation. Alternatively, it may be the case that our model did match the real world, and it was the physical board which needed to be fixed before the game could continue. During the human player's turn, this would have happened either because the person accidentally made a mistake, or intentionally tried to cheat. During the robot player's turn, this could have happened if the robot fumbled a manipulation or accidentally bumped something, or could even have been the result of the person trying to cheat while the robot was busy with its turn. Our current implementation does not attempt to make these distinctions. Instead, it assumes that illegal changes are always mistakes, which are left to the robot's operator to correct.

V. Manipulation

The robot would begin its turn by using Raven Checker's alpha-beta search to calculate its next move, which it expressed as a goal *game board model*. The disparity between the current *game board model* and the goal was then used to derive a series of ordered steps for the robot to follow called a plan. For example, a "jump" consisted of two steps: first moving one of the robot's pieces from its starting location to its destination on the opposite side of the opponent's checker, followed by a step to remove the opponent's checker from the board.

A. Arm Movement

Once the plan had been generated, the *Action State* would be used to execute each step of the plan. Each step represented an action to perform such as movePiece, removePiece, or stackPiece. These actions were defined as specialized variations of a "pick and place" task. For each step, the arm would first be moved coarsely (i.e. using only encoder values for feedback) to an alignment location above a target object. It would then be carefully lowered down to grasp the object using visual servoing before being raised up again. This would be followed by a second coarse motion re-positioning the arm above a target object or location. Finally, the arm would be carefully re-lowered and the object released at the destination.

Because objects might be shifted at any point in time, determining the physical locations the arm needed to move to during coarse alignment was delayed until just before the step was executed. Alignment positions used a static orientation which kept the grippers pointed normal to the board at a fixed height, while the horizontal coordinates used depended on whether or not a checker object was occupying the destination space. Empty squares would return their center coordinates while occupied squares would return the center of the checker object, allowing the alignment to react robustly to checkers that were not placed precisely in the center of squares. The actions removePiece and stackPiece both involved checkers not located on the game board, with the exact off board location being unimportant. In those cases, the locations and colors of the off board checkers were analyzed to either find an empty place to put a checker down or a correct color checker to pick up. In the case of multiple-jump scenarios, additional positions

were added to emulate the way a human player makes multiple jumps in a single move (usually by moving their piece in a pattern over their opponents pieces to demonstrate the path they took, often tapping the board between captured pieces). We elected to have the robot simply trace the path while briefly pausing over squares as opposed to "tapping" to speed up the turn and decrease the risk of bumping other pieces on the board. Once the physical destination or destinations were known, the coarse motions were planned using pre-solved solutions provided by an inverse kinematics solver.

B. Visual Servoing

As with any robotic system, small amounts of error accumulation were present throughout, spanning everything from the camera sensor to arm encoders. Using 3cm wide squares and 2.5cm diameter checkers, the robot could only be allowed a centimeter of position estimation error to successful perform the pick and place actions involved in the game. In order to achieve the level of accuracy necessary to accomplish this we performed the delicate action of lowering the gripper over an object using visual servoing.

Motion control was transferred to servoing when the arm arrived into an alignment position several centimeters directly above our best estimate of the target object's location. As the gripper descended, the arm's horizontal position was adjusted to keep the colored blob representing the checker centered in the arm's camera. The arm would continue lowering until the arm was at gripping height (determined during prior calibration) or resistance was encountered at which point the arm would either grasp or release the checker. Successful capture of the checker could be confirmed by whether or not the checker blob moved in the image as the arm was raised. Occasionally, the grasping would fail (usually due to the arm being blocked from completely lowering). This resulted in the arm resetting to the initial alignment position and reattempting to grab the checker.

Although the grippers had been repositioned to be closer to the center of the camera view (see Section II-B), the location of the checker in the camera images suffered from a perspective effect which would cause the checker to appear to move away from the center of the image as the arm descended that resulted in an alignment error proportional to the distance from the camera to the checker. This was corrected for using the IR sensor to estimate the distance to the board and adjust the expected position of the blob accordingly.

VI. System Evaluation

To test our system we performed a study in which 12 people played checkers against the robot. Five of the participants were able to complete the game against the system. The remaining 7 either asked for a draw sometime after an hour of playing or were cut off after 90 minutes in a stalemate. An experimenter was present to oversee the games, intervening only to make corrections to the board that would otherwise have prevented the game from progressing.

During the 12 games played, the robot took a combined 537 turns in which it successfully completed 89.1% (637/715) of its actions. The robot successfully perceived the board after 95.7% (1022/1068) of all turns taken by both players. The robot operator need to fix the board a total of 124 times during game play, 46 times for perception errors and 78 times for motion errors (during 68 unique turns). Of the 78 motion errors, 39 were unsuccessfully grasped pawns, 13 were unsuccessfully grasped kings, 6 were unsuccessful regular placements, and 20 were unsuccessful crowning placements. There were only three instances of a checker being placed in the wrong location. The robot's turns lasted 95s on average, which includes time for manipulation and perceiving the board at the beginning and end of each turn.

VII. Conclusion and Future Work

We have presented a system that has shown itself to be capable of playing a game of checkers against a human opponent using a commercially available game. The game needed only very minimal modifications for use by the robot (stickers on the game pieces) and required no augmentations to the environment such as special positioning of the board or storage of game pieces. To our knowledge, it is the first system capable of detecting, creating, and moving stacks of checkers as kings. The robot was also able to track the progress of the game and detect the occurrence of illegal moves.

We believe our system is a suitable testbed for further investigating methods of evaluating and handling illegal board configurations. Our next goal is to implement strategies for having the robot be capable of fixing the board after detecting it has blundered its move. Methods of conveying to the human player what aspect of the board needs to be changed such as gesturing or displaying highlighted images on a screen are another area of our work. Finally, we are interested in distinguishing between mistakes and cheating, which we believe can be differentiated by analyzing the potential costs and benefits of each disparity for each player.

Acknowledgment

This research has been supported in part by the National Science Foundation (IIS-1111125 and IIS-0905228) and the Army Research Office (W911NF-13-1-0299).

References

[1] B. Marsh, C. Brown, T. LeBlanc, M. Scott, T. Becker, C. Quiroz, P. Das, and J. Karlsson, "The rochester checkers player: multimodel parallel programming for animate vision," *Computer*, vol. 25, no. 2, 1992.

[2] D. Lewis and D. Bailey, "A checkers playing robot," *Institute of Information Sciences and Technology Massey University*, 2004.

[3] T. Bem, "Robot playing checkers," *Praca magisterska, WEiTI*, 2009.

[4] J. Huizinga, *Homo ludens*. Taylor & Francis, 1949, vol. 3.

[5] C. Matuszek, B. Mayton, R. Aimi, M. P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J. R. Smith, and D. Fox, "Gambit: An autonomous chess-playing robotic system," in *ICRA'11*, 2011.

[6] Rethink Robotics, "Baxter Research Robot," Webpage, May 9 2014, http://www.rethinkrobotics.com/products/baxter/. Accessed May 2014.

[7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[8] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation*, 2011.

[9] J. Elfring, S. Van Den Dries, M. Van De Molengraft, and M. Steinbuch, "Semantic world modeling using probabilistic multiple hypothesis anchoring," *Robotics and Autonomous Systems*, vol. 61, no. 2, 2013.