PLAYING TIC-TAC-TOE WITH TEKKOTSU: THE DEVELOPMENT OF THE GRASPER

By

Glenn V. Nickens
B.S. May 2007, University of the District of Columbia
A Thesis submitted to the Faculty of
Norfolk State University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

COMPUTER SCIENCE

Norfolk
September 2011

Approved by:

_____
Thorna Humphries (Advisor)

_____
Mona Rizvi

_____
David Touretzky

This work was supported in part by National Science Foundation awards DUE-0717705 and CNS-0742106 to Dr. David Touretzky, and award CNS-0742198 to Dr. Thorna Humphries.

TABLE OF CONTENTS

Acknowledgements

I would like to acknowledge and extend thanks to the following persons who helped make the completion of this research possible:

Dr. Thorna Humphries, for her continuous encouragement to keep pushing and getting me back on track when life presented its many challenges.

Dr. David Touretzky, for his patience and dedication while advising me throughout this research and for designing the Hand/Eye robot.

Ethan Tira-Thompson, for his help with using and fixing Tekkotsu and Mirage.

Jonathan Coens, for fixing some bugs and for his help making the Grasper smarter while he worked on his thesis, “Taking Tekkotsu Out Of The Plane”.

List of Figures

Abstract:

The Tekkotsu robot programming framework has a collection of interacting modules, known as the "Crew", that make it easy to construct complex behaviors. In this thesis, a new member, the Grasper, was developed. The purpose of the Grasper is to control a robot's arm in order to enable the manipulation of objects. For this research, a simple Hand/Eye robot with a three-link planar arm was used. Manipulating objects from one position to another using a planar arm involves various kinematics calculations, collision detection, and path planning. Kinematics calculations are required to determine the arm configurations that will place the arm in a desired position. Collision detection is performed to keep the arm from accidentally hitting itself or obstacles in its environment. Path planning is required in order to move the arm and an object from point A to B. The path planning algorithm that was used in this research is a randomized algorithm.

Since the hand/eye robot does not have closable fingers, a path planning constraint was developed to ensure that the robot does not loose grip of objects while moving them. Paths are smoothed to remove jerky and meandering characteristics. Every manipulation performed by the Grasper is carefully planned and executed.

A tic-tac-toe player, which requires the manipulation of game pieces, was developed to demonstrate the effectiveness of the Grasper. As a result of this research, Tekkotsu can easily manipulate objects with a three-link planar arm with a code segment that consists of only a few lines of code.

1 Introduction

Tekkotsu is an open source robot programming framework developed mainly in C++. It is an object-oriented and event passing architecture that makes full use of the template and inheritance features of C++. Programmers use high-level primitives, such as “look at this object” or “walk to that location”, to control robots. These primitives abstract from the low-level concepts of robot programming such as joint angles and motor torques, allowing programmers to focus on what they want the robot to do, as opposed to how to do it.

Tekkotsu was initially designed for programming Sony’s AIBO dogs. Once production and support for these dogs ceased it was time for Tekkotsu to find other platforms to support. With a very limited number of inexpensive tabletop robots available on the market, the developers of Tekkotsu decided to construct their own robots. They created the Hand/Eye robot that has a planar arm for manipulating objects.

Tekkotsu includes a collection of interacting software modules known as the “Crew” [1] that provide capabilities that make it easy to construct complex behaviors. The four Crew members are the MapBuilder, the Lookout, the Pilot, and the Grasper. The Grasper, a software module whose job is to control the robot’s arm to manipulate objects, is the newest Crew member, and its construction is the subject of this thesis.

In this thesis, the Hand/Eye robot is used to test the design and implementation of the Grasper. The problems and solutions for advancing Tekkotsu’s ability to manipulate objects with the arm are explored. Inverse kinematics calculations are used to determine if objects can be manipulated. Manipulation paths are planned using a fast, randomized planning algorithm. Collision detection is accomplished using a simple algorithm that

detects when two convex shapes are overlapping. Ensuring that objects stay within the fingers of the arm's hand necessitated the development of the direction of motion constraint. A manipulation engine was designed to give developers the ability to express desired manipulations as simple requests. A tic-tac-toe player was developed to show that the Grasper can effectively manipulate objects with the planar arm, specifically the planar arm of the Hand/Eye Robot.

2 The Hand/Eye Robot

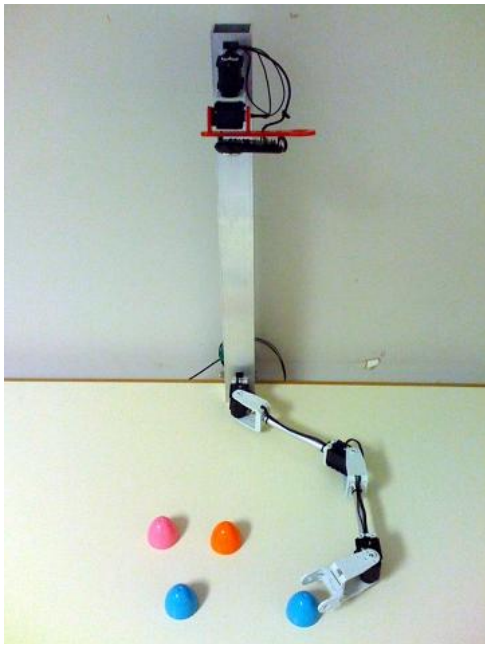

Figure 1 - Hand/Eye robot

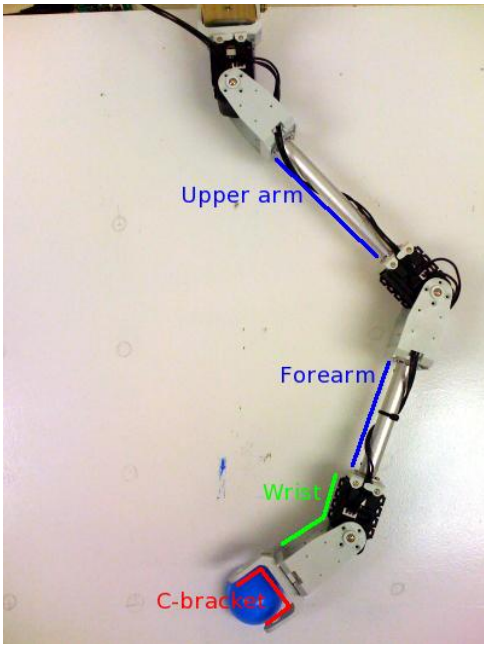

Figure 2 - Manipulation surfaces

Developed in the Tekkotsu Lab at Carnegie Mellon University, the Hand/Eye robot [2] is a simple robot made of a web-cam and a planar arm. The web-cam, which is attached to a pan-tilt joint assembly, is fastened to the top of an aluminum mast and the arm is fastened to the bottom. The mast is about 0.6 meters tall. This configuration allows the robot to see all around itself and far beyond the reach of its arm. The web-cam together

with the vision system built into Tekkotsu give the Hand/Eye robot the ability to detect objects in its environment. Both the camera and the arm are connected via USB to the developer's computer. Figure 1 shows the Hand/Eye robot playing with some plastic egg shells.

The planar arm is a three-link arm, made of three Dynamixel AX-12 servos, two aluminum tubes and a c-bracket. The c-bracket is the arm's end effector or hand. All three joints are rotational. The servos in the arm are arranged so the joints can only move parallel to the surface on which the robot is placed, hence the planar arm. Figure 2 shows the arm's manipulation surfaces, the forearm and upper arm, the wrist, and the interior of the c-bracket. The latter is the primary manipulation surface.

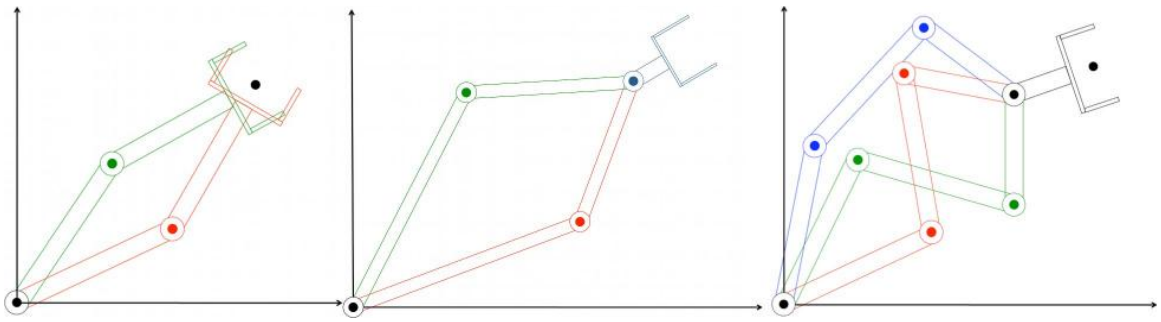

Figure 3 - Illustrations showing a two, three and four link arm respectively

The Hand/Eye robot has a three-link arm because manipulation in a plane requires at least three degrees of freedom to independently control, the x and y coordinates of the end-effector and its orientation. A three-link arm allows the end-effector to be placed in infinitely many orientations, allowing infinitely many ways to manipulate an object. Manipulation with a two–link planar arm would be inadequate because given the coordinates of the end-effector, there are at most two achievable orientations. The first image in figure 3 shows the two achievable orientations of the end-effector of a two-link arm given the desired end-effector coordinates. Figuring out how to place the end-

effector of a three-link arm in a desired configuration, x and y coordinates and an orientation, is a straightforward process that produces at most two solutions. The end-effector on an arm with four or more links can also be placed in infinitely many orientations. However, there may be infinitely many solutions. In addition, the process of finding these solutions is much more complex. An arm with four or more links would be better to be able to reach around obstacles, but it would also be heavier and more expensive, requiring more power and wiring.

Because the arm does not have closable fingers, it can only push objects; it cannot pull them. This simplifies the hardware but complicates the path planning task. A solution to this problem is one of the main accomplishments of this thesis.

3 A Touchstone Problem: Playing Tic-Tac-Toe

A tic-tac-toe player was written to demonstrate the effectiveness of the primitives developed in this research. Tic-tac-toe was chosen because it is a simple game to play if one can locate the board and the game pieces, and if one can move the pieces onto the board. The Hand/Eye can use its pan-tilt web cam and Tekkotsu's vision system to locate the board and the game pieces. Tekkotsu's vision system gives the robot the ability to distinguish objects based on their shape and color. The vision system also reports the location of the objects in the robots environment. The Hand/Eye can use its planar arm to move the game pieces onto the board and to sweep them off the board at the end of the game. Moving game pieces onto the board will become progressively harder as the board fills up because the arm has to reach around pieces that have already been placed on the board, as shown in figure 4.

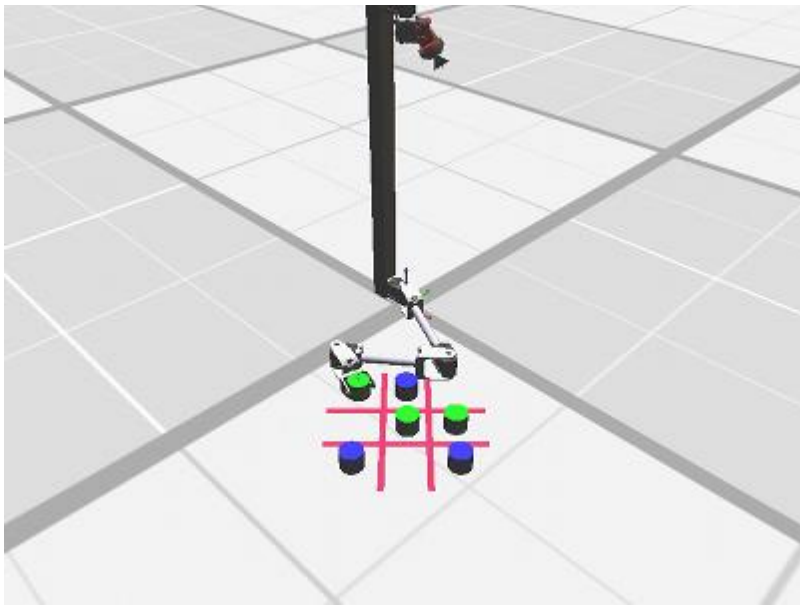

Figure 4 - Simulated robot reaching around one object to place another

Mirage, a simulation environment that allows a robot to operate in a virtual world, was used to test the code developed in this research. The same code that controls a simulated Hand/Eye robot in Mirage can also control a real robot. Using Mirage reduced the time needed for testing and debugging.

4 Technical Problems To Be Solved

Moving a game piece from one location to another requires solutions to several problems:

a) Inverse kinematics (IK): find a configuration of the arm (i.e., a set of joint angles) that puts the fingers of the c-bracket around the object at its start location. Also find a configuration that puts the c-bracket at the destination location.

b) Collision detection: determine if a given arm configuration will cause a collision between some part of the arm and an object or some other part of the arm.

c) Path planning: finding a sequence of arm configurations that move the c-bracket

from a start to a destination location while avoiding collisions with itself or other objects.

d) Constrained path planning: when moving an object, find paths that keep the direction of motion of the c-bracket roughly aligned with the direction in which the fingers are pointing, so that the object cannot slip out.

e) Path smoothing: given a randomly generated path to the destination that avoids obstacles and obeys the direction of motion constraint, find a shorter, smoother path that accomplishes the same result and still obeys all constraints.

f) Manipulation planning: develop a convention for users to express manipulation requests, and an algorithm to translate each request into a sequence of IK and path planning problems to be solved.

The software module that translates user requests into IK and path planning problems, solves those problems, and then executes the solution and reports the result is called the Grasper.

In the following sections each of the above problems and their solutions are described in more detail.

5 Kinematics Calculations

Kinematics calculations describe the relationship between the position and orientation of the end-effector and the joint angles. There are two types of kinematics, forward or direct kinematics and inverse kinematics. Forward kinematics determines the position of the end-effector given a set of joint angles and the distances between the joints. This is easily solved with a series of matrix multiplications. Forward kinematics problems

always have a unique solution.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} c\theta_1 & -s\theta_1 \\ s\theta_1 & c\theta_1 \end{bmatrix} \begin{bmatrix} L_1 \\ 0 \end{bmatrix} + \begin{bmatrix} c(\theta_1+\theta_2) & -s(\theta_1+\theta_2) \\ s(\theta_1+\theta_2) & c(\theta_1+\theta_2) \end{bmatrix} \begin{bmatrix} L_2 \\ 0 \end{bmatrix} + \begin{bmatrix} c(\theta_1+\theta_2+\theta_3) & -s(\theta_1+\theta_2+\theta_3) \\ s(\theta_1+\theta_2+\theta_3) & c(\theta_1+\theta_2+\theta_3) \end{bmatrix} \begin{bmatrix} L_3 \\ 0 \end{bmatrix}$$

The above equation shows the matrix multiplications for a three-link planar arm. θ_1, θ_2, and θ_3 represent the shoulder, elbow and wrist angles respectively. L_1, L_2 and L_3 represent the link lengths between the shoulder and the elbow, the elbow and the wrist, and the wrist and the end-effector respectively.

Inverse kinematics calculations search for a set of joint angles that will produce a desired end-effector configuration. They are harder to perform and may produce multiple solutions, one solution or none. In this research, the shoulder, elbow and wrist are the joints whose angles need to be determined given a desired configuration of the c-bracket. Inverse kinematics calculations for a three-link planar arm involve a four-step process based on a common analytical approach documented in the book 'Robot Modeling and Control' [3].

The steps are as follows:

I. Determine the position of the wrist [x_w, y_w] given a desired c-bracket configuration, a target point [x_t, y_t] and an orientation ϕ_t. This is done using the following equation:

$$\begin{bmatrix} x_w \\ y_w \end{bmatrix} = \begin{bmatrix} x_t \\ y_t \end{bmatrix} - L_3 \begin{bmatrix} \cos(\phi_t) \\ \sin(\phi_t) \end{bmatrix}$$

Where L_3, shown in figure 5, is the distance from the wrist to the target point [x_t, y_t].

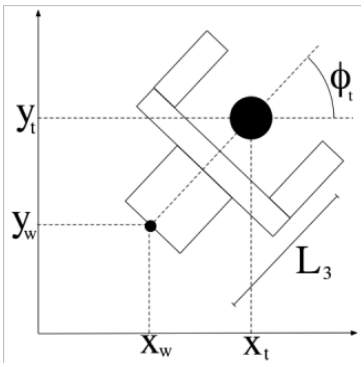

Figure 5 - Wrist position

II. Next, calculate the elbow angle θ_2 using the previously determined wrist position $[x_w, y_w]$ and the arm dimensions L_1 and L_2, shown in figures 6 and 8. This is done with the following equations:

$$D = \frac{x_w^2 + y_w^2 - L_1^2 - L_2^2}{2L_1L_2} = \cos(a)$$

$$q_2 = \tan^{-1}\frac{\pm\sqrt{1 - D^2}}{D}$$

If θ_2 is equivalent to 0 then the elbow is fully extended. When the elbow is not fully extended there are two solutions, $+\theta_2$ and $-\theta_2$, known as the elbow-up and elbow-down solutions. See figure 7.

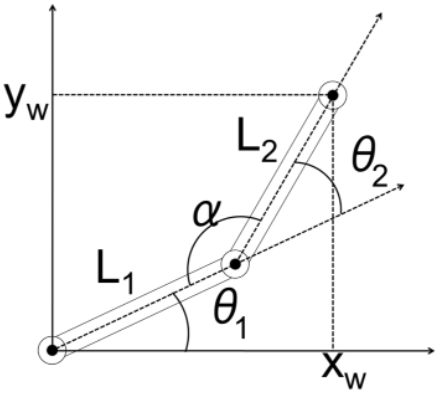

Figure 6 - Elbow angle, θ_2

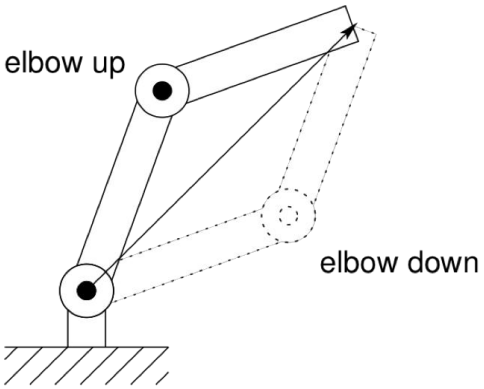

Figure 7 - Elbow up/down configurations

III. Then the shoulder angle, θ_1 is determined using $[x_w, y_w]$, L_1, L_2 and $\pm\theta_2$ with

the following equation:

$$q_1 = \tan^{-1}\left(\frac{y_w}{x_w}\right) - \tan^{-1}\left(\frac{L_2 \sin(q_2)}{L_1 + L_2 \cos(q_2)}\right)$$

IV. Finally, the wrist angle, θ_3 is determined using this equation:

$$q_3 = f_t - (q_1 + q_2)$$

Figure 8 shows the wrist position and all the angles that were calculated using the four steps described above.

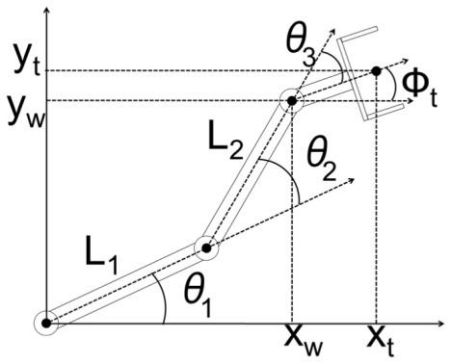

Figure 8 – Entire arm and the variables

The inverse kinematics calculations described above are intended for an abstract three-link planar arm. These calculations produce joint angles between $+\pi$ and $-\pi$. However, the joints on the Hand/Eye's arm cannot turn that far because adjacent links will collide with each other. Each configuration is therefore validated to ensure that each angle is within the corresponding joint's rotation range.

6 Collision Detection

Since the Hand/Eye cannot feel, it is incapable of detecting a self-collision or a collision with an object. A self-collision can occur if the arm is turned too far inwards. If this happens the third link could collide with the first link. The second and third links

could collide with the aluminum mast also. A collision with an object, such as a game piece, could occur as the arm moves around. Figures 9 and 10 illustrate these situations. To avoid these collisions, collision detection must be performed.

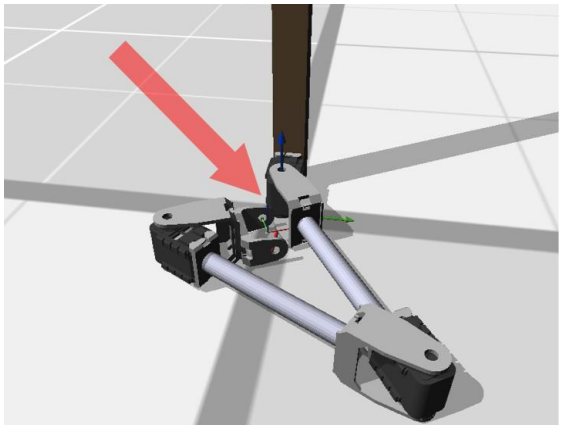

Figure 9 - Third link colliding with first link

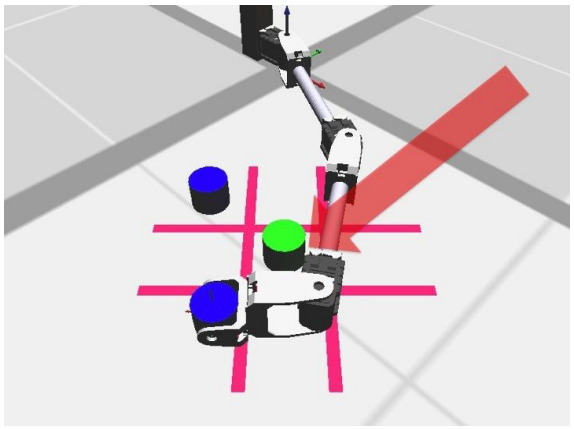

Figure 10 - Second link colliding with a game piece

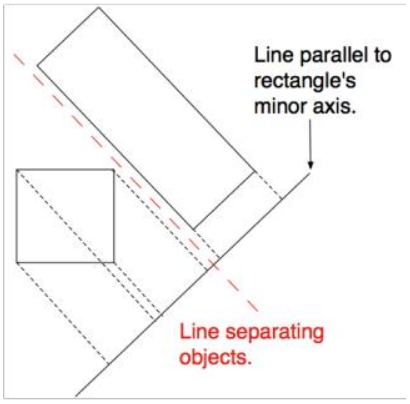

Figure 11 - Separating Axis Theorem

The algorithm used for collision detection in this research is based on the Separating Axis Theorem. This theorem states that given two 2D convex shapes lying in a plane there exists a line onto which their projections will be separate if and only if the shapes are not intersecting [4], see figure 11. For this algorithm, the arm links are modeled as rectangles. The c-bracket is modeled as three rectangles, as part of the third link. All the

objects in the robot's configuration space are also modeled, either as rectangles or spheres, depending on what the object looks like. See figure 12.

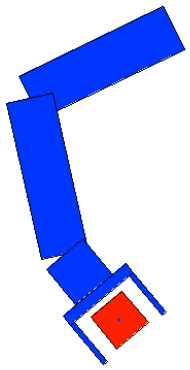

Figure 12 - Arm and object modeled as rectangles

For each configuration, the arm-link rectangles are rotated and translated using the joint values, forward kinematics, and geometry to determine what position each rectangle should be in. The positions of the shapes representing the objects are determined by the position of each object. Once the position of each shape has been determined, each arm-link rectangle is compared to each object shape (either rectangle or sphere). The rectangle representing the first link is also compared to each of the rectangles representing the third link. If at least one of the comparisons determines that the shapes are intersecting, then the configuration will cause a collision.

7 Path Planning Using RRT-Connect

Path planning is accomplished with a randomized planning algorithm called RRT-Connect [5]. RRT-Connect uses Rapidly-exploring Random Trees (RRTs) and a greedy algorithm that tries to connect two RRTs, one beginning from a start configuration and the other from an end configuration.

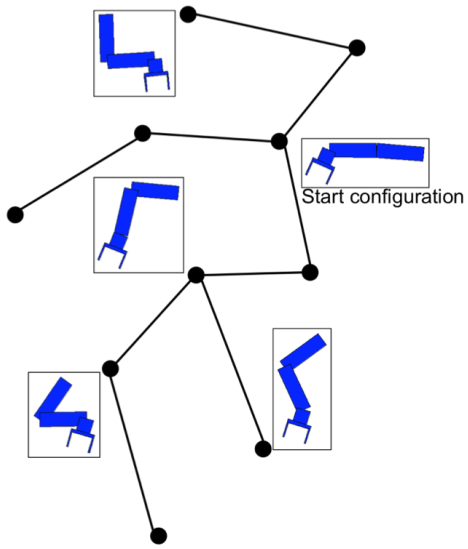

Figure 13 – A single RRT structure

Figure 13 shows a single RRT, which is a tree data structure whose vertices encode configurations of the arm. RRTs grow by iteratively extending themselves towards a randomly generated configuration, q_{rand}. Picking a random number within the joint rotation range for each joint generates a random arm configuration. After generating q_{rand}, a search is performed on the tree to find the vertex that is nearest to the random configuration. Once the nearest vertex, q_{near}, has been found, the tree is extended from q_{near} towards q_{rand}. This is done by creating a new configuration, q_{new}, which is some fixed incremental distance, ε, from q_{near} in the direction of q_{rand}; see figure 14. If q_{rand} is already within ε of q_{near}, it is now considered to be q_{new}. q_{new} is then tested for a collision. q_{new} is added to the tree as a child vertex of q_{near} as long as it is collision free. If q_{new} is not collision free it does not get added to the tree, hence the tree does not get extended during this iteration.

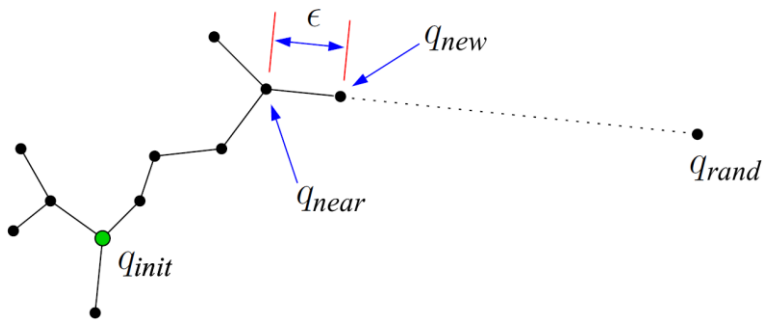

Figure 14 – Extending a tree towards q_{rand} [5]

The RRT-Connect algorithm alternately extends each RRT and biases their growth by also extending the trees towards each other. When extending a tree towards q_{rand}, the RRT-Connect algorithm will continue to extend q_{near} towards q_{rand} until it either reaches q_{rand} or until it cannot be extended any further. After extending one tree, the other tree is extended as far as possible towards the previous tree's q_{new}. This is how the trees are biased to grow towards each other. If a tree is extended all the way to the previous tree's q_{new} then the trees have connected, otherwise they are swapped. Once the trees have connected backtracking is used to extract a path from the trees, starting from the vertices that connected as shown in figure 15.

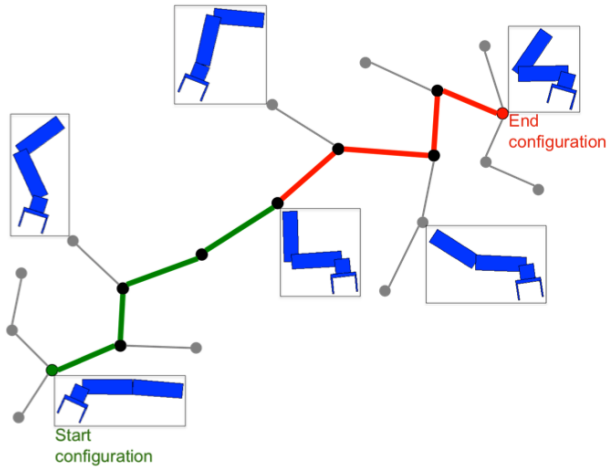

Figure 15 - An extracted path

The Hand/Eye does not have closeable fingers, so it cannot hold on to anything. The only way it can move objects is by pushing them. This means the fingers must be pointing roughly in the same direction that the object is being moved, or the object will pop out of the hand. To ensure that contact is not lost, the c-bracket predicate was developed. The predicate compares the direction of motion between two consecutive configurations to the direction in which the fingers of the preceding configuration are pointing. As long as these directions are within a given range of each other, the object will stay in the c-bracket.

Figure 16 shows a configuration that will cause the c-bracket to move in an appropriate direction. The red dot is where the center of the c-bracket will be should the c-bracket be moved to the red configuration. Not only is the direction of motion from the green to the red configuration appropriate, it is also within the allowed range of motion, the green triangle.

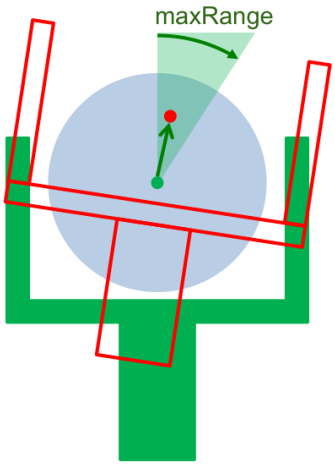

Figure 16 – Appropriate direction of motion

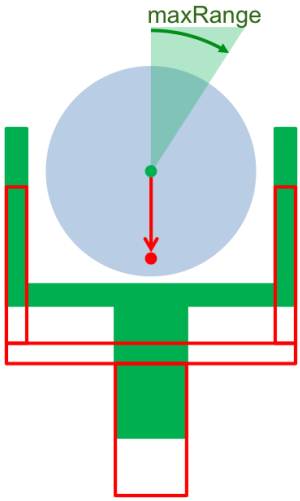

Figure 17 – Inappropriate direction of motion

Figure 17 shows a situation where the subsequent configuration, the red configuration,

will cause the c-bracket to move in an inappropriate direction. If the c-bracket were moved from the green configuration to the red configuration, the c-bracket will lose its grasp of the object.

The c-bracket predicate algorithm proceeds as follows:

i. Determine p_{from}, p_{to} and p_{goal}, the center positions of the c-bracket, given q_{from}, q_{to} and q_{goal} respectively. q_{from} and q_{to} are the two consecutive configurations. q_{goal} is the configuration a tree is being extended to.

ii. Determine the c-bracket orientations o_{from} and o_{to} for q_{from} and q_{to}.

iii. Determine the direction of motion dir, from p_{from} to p_{to}.

$$dir = atan2(p_{to} - p_{from})$$

iv. Determine the angular distance angDist, between o_{from} and dir.

v. If angDist is greater than the maxRange (the maximum allowed range of motion between two configurations) a new q_{to} is generated.

$$q_{to} = CreateNewQ(q_{from}, q_{goal})$$

For the c-bracket on the Hand/Eye, the maxRange is set to 40°.

vi. If q_{to} is not valid or not collision free, return false, otherwise proceed to the next step

vii. Add q_{to} as a child vertex of q_{from}. Return true if q_{to} is closer to q_{goal} than q_{from}.

Figures 18 and 19 illustrate what the CreateNewQ function in step v does. Assume, in figures 18 and 19, that the green, blue-outlined and red-outlined configurations are q_{from}, q_{goal} and q_{to} respectively. Where q_{from} is the configuration that the c-bracket is being extended from and q_{goal} is the configuration that the c-bracket is being extended to. Q_{to}, in figure 18 is the proposed step (some fixed incremental distance ε) in the direction of q_{goal}

from q_{from}. The red circle within the fingers of q_{to} in figure 18 marks the center of the c-bracket. Since moving the c-bracket from q_{from} to q_{to} in figure 18 will cause an inappropriate movement the CreateNewQ algorithm will generate a new q_{to} that is centered in the same position as q_{from}. However, the new q_{to} will be rotated 5° about its center in the direction of q_{goal}, as can be seen in figure 19. The red-outlined configuration in figure 19 is the new q_{to} that is generated.

During the next extend iteration q_{to} now becomes q_{from}. Repeated attempts to move the c-bracket to q_{goal} will cause the c-bracket to appear to rotate about the object.

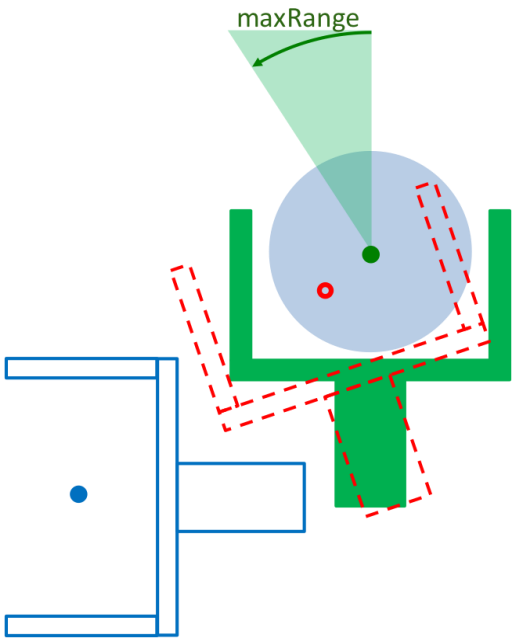

Figure 18 – Proposed q_{to} configuration

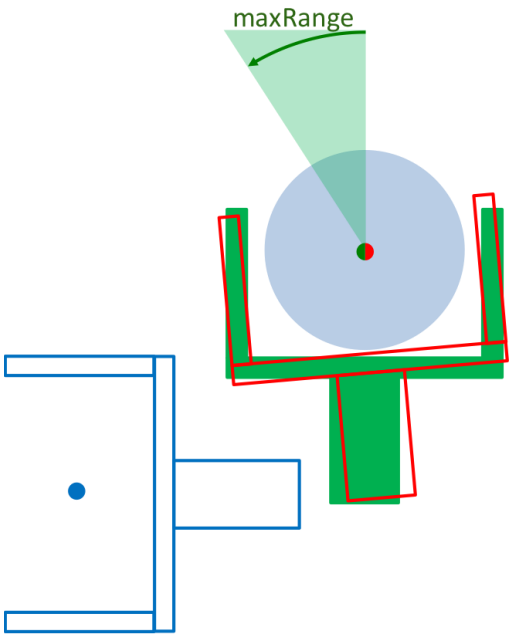

Figure 19 – New q_{to} configuration

Generating a new q_{to} with the CreateNewQ function improved the performance of the path planning process drastically.

9 Path Smoothing

Paths produced by the path planner are often jerky and meandering. This is a result of

the random component of the planning algorithm. A smoothing algorithm is applied to the path to make it a bit more natural and fluid. Smoothing is accomplished by selecting random segments of the path to be substituted. The segments are substituted with a more smooth and often shorter segment after the new segment has been determined to be collision free.

A new segment, S_{new}, is created by extending the first configuration of the randomly selected segment, S_{rand}, directly towards the last configuration of S_{rand}. This is accomplished the same way the RRT-Connect algorithm extends a tree's q_{near} towards q_{rand}. If S_{new} is successfully created, the entire segment S_{rand} is replaced with S_{new}. This process is repeated 2N times, where N is the number of configurations in the original path.

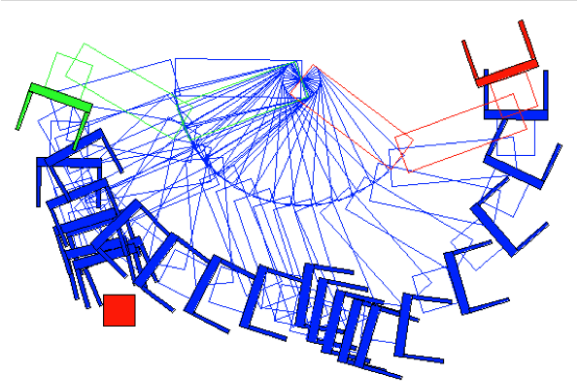

Figure 20 - Path produced by planner prior to smoothing

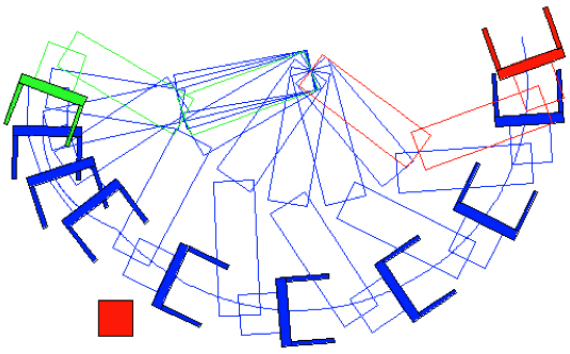

Figure 21 – A smoothed path

Figures 20 and 21 illustrate the effect the smoothing algorithm has on a path. Figure 20 shows a path that was produced by the planner before smoothing. One can see how the c-bracket appears to stick out near the red square, an obstacle. The c-bracket appears to stick out again after passing the obstacle as the arm makes its way from the green to the red configuration. Figure 21 shows a shorter, more fluid path that was created after applying the smoothing algorithm to the path in figure 20.

10 Manipulation Planning and the Grasper

Moving an object with the Hand/Eye's arm is a complex process that involves 6 major steps:

- Finding grasping configurations that put the fingers around the object without collisions
- Planning a path from the current arm configuration to a grasping configuration
- Finding destination configurations with the fingers around the object at the desired destination
- Planning a path to move the object from a grasping configuration to a destination configuration while obeying the direction of motion constraint
- Planning a path to move the arm from the destination configuration to the disengaged configuration
- Executing the manipulation sequence

These are the steps that the Grasper uses to solve manipulation requests. A user submits a manipulation request by supplying the necessary information about the intended manipulation. The user submits information such as what object is to be moved,

what obstacles are present in the robot's environment, where the object should be moved to, and whether the arm is to be moved away from the object once the object has been moved to its destination. Once the request is submitted, it is validated to ensure that all the necessary information is present. For instance, the Grasper verifies that the user has indicated which object is to be moved if a request to move an object is submitted. (Several other types of requests are possible.) The Grasper then follows the above-mentioned steps to try to develop a result.

I. Finding a grasp configuration

Given the position of the object, the Grasper will try to determine some valid configurations that will place the arm's fingers around the object. It does so by sampling numerous c-bracket configurations, starting with

$$[x_t, y_t] = [x_{obj}, y_{obj}] \text{ and } \phi_t = 0^\circ$$

Where $[x_{obj}, y_{obj}]$ are the center coordinates of the object. The Grasper uses inverse kinematics calculations to determine the arm configurations for the desired c-bracket configuration. If valid configurations are found, they are stored and the next c-bracket configuration is sampled. The next c-bracket configuration to be sampled is

$$[x_t, y_t] = [x_{obj}, y_{obj}] \text{ and } \phi_t' = \phi_t + 5^\circ$$

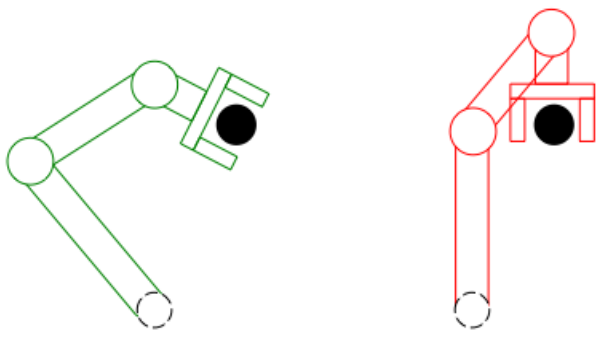

Figure 22 – A valid (green) and invalid (red) sample configuration

This sampling process continues until the c-bracket has completed a full circle around the object. If no valid configurations are found then the object cannot be reached and hence cannot be moved.

Figure 22 shows two arm configurations for two sampled c-bracket configurations. The red configuration is invalid because it would cause the c-bracket to collide with the upper arm.

II. Planning a grasp path

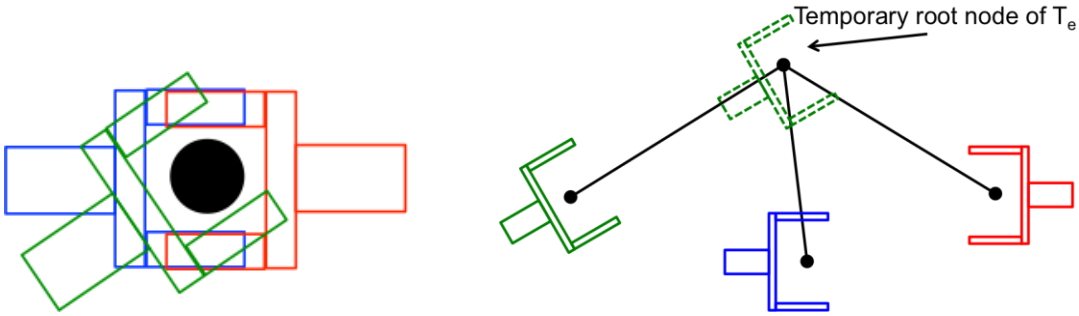

Figure 23 – Initial nodes in T_e

If, in step I, at least one valid grasping configuration is calculated, the Grasper will attempt to plan a path to grasp the object (i.e. place the fingers around the object). The Grasper uses the RRT-Connect algorithm, without the direction of motion constraint, to plan the path. The planner does not use the direction of motion constraint because it will not be moving an object during this part of the manipulation. One tree, T_s, is grown from the arm's current configuration, while the other tree, T_e, is grown from the configurations calculated in Step I. The first few nodes of T_e are comprised of the configurations from step I. Each configuration verified to be collision free is added to T_e as a child of the root of T_e. It doesn't matter which of these configurations is reached; they are assumed to be equally good. The root node is arbitrarily made to be

a copy of the first collision free configuration. Assume the three c-bracket configurations in figure 23 are three valid sampled configurations from step I and the green configuration is the first to be sampled. Then a copy of the green configuration is made the temporary root node of T_e.

If a path is found, the temporary root node of T_e, the copy of the green node, is removed. This is done because the temporary root node and its successful child may be far apart and moving the arm from the child to the root may cause a collision.

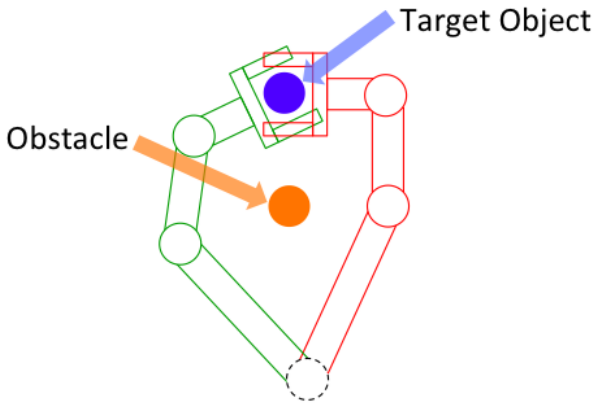

Figure 24 – A temporary root node (green) and a child node (red)

Figure 24 shows a temporary root node, the green configuration, and one of its children, the red configuration. If the arm were moved from the red configuration to the green configuration, the arm would collide with the obstacle in between them.

III. Finding destination configurations

This step is similar to step I. However, instead of using the coordinates of the object to generate configurations to populate T_e, the coordinates of the objects' desired destination are used.

$$[x_t, y_t] = [x_{dest}, y_{dest}] \text{ and } \phi_t = 0^\circ$$

If no valid configurations are found then the desired destination cannot be reached,

hence the object cannot be moved.

IV. Planning a move path

If at least one configuration is generated in step III, the Gasper will proceed to plan a move path. The move paths' end tree, T_e, is setup much like the grasp paths' T_e, except the configurations used are those calculated in step III. In this step however, the start tree, T_s, is grown from the last configuration of the grasp path as opposed to the arm's current configuration. The Grasper uses the direction of motion constraint when planning this path because it needs to ensure the c-bracket does not lose contact with the object. The direction of motion constraint is used in three places during the planning process. First it is used every time a tree is being extended towards a randomly generated configuration. It is also used during the connect process, when a tree is being extended towards the other. Lastly, it is used during path smoothing.

V. Planning a disengage path

Once a path has been calculated to move the object to its destination, the arm is either left as is or is moved away. If the arm is to be left as is then the Grasper is done planning. If the arm is to be moved away, a path is planned to move the arm from the last configuration of the move path to a predetermined rest configuration. The predetermined rest configuration is specified as part of the manipulation request. As with the grasp path-planning process, the disengage path is planned without the direction of motion constraint. The arm will not be moving an object so there is no need to pay attention to the direction in which the fingers are pointing.

VI. Executing the manipulation request

In steps II, IV and V (if the arm is to be disengaged from the object) paths were

planned and stored. The Grasper now executes these paths in the order in which they were planned

- Grasp path – move the arm to grasp the object
- Move path – move the object to its destination
- Disengage path – move the arm away from the object

If any of the first four steps fails, the Grasper will discontinue the planning process and report why it failed.

11 Results

To demonstrate the results of this research a simple randomized tic-tac-toe player was developed. The player randomly picks an empty position on the board to move the next game piece to. The player is setup as a state-machine, where each node has a specific task to perform.

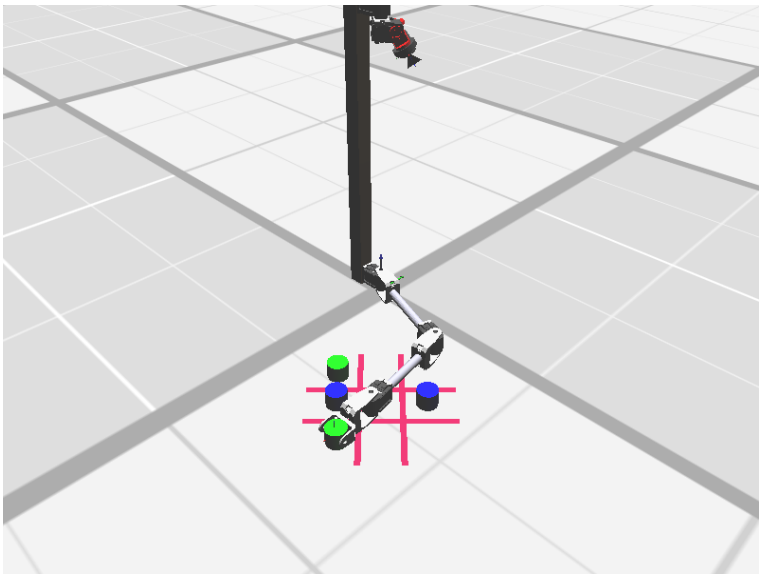

Figure 25 – Hand/Eye playing tic-tac-toe

The first node, ReadBoard, locates the board (pinks lines) and the game pieces (blue

and green objects). The next node, ParseBoard, parses the lines and game pieces found in the ReadBoard node. It first determines the nine spaces that make up the game board; then it determines which positions have not been occupied. If the board is full, the ParseBoard node will report this and proceed to end the game. Otherwise, it randomly picks one of the unoccupied positions as the next position to play.

The next node, SpawnNextPiece, was developed to place the next game piece in the robot's environment. This was necessary because it was very hard to arrange the un-played game pieces in a way that the Hand/Eye robot could reach all of them. If the game pieces were well spread out, the last few un-played pieces would be out of reach of the arm, see figure 26. Placing all of the un-played game pieces within the arm's reach resulted in the pieces being so close that the Grasper would rarely find a path to grasp or move one.

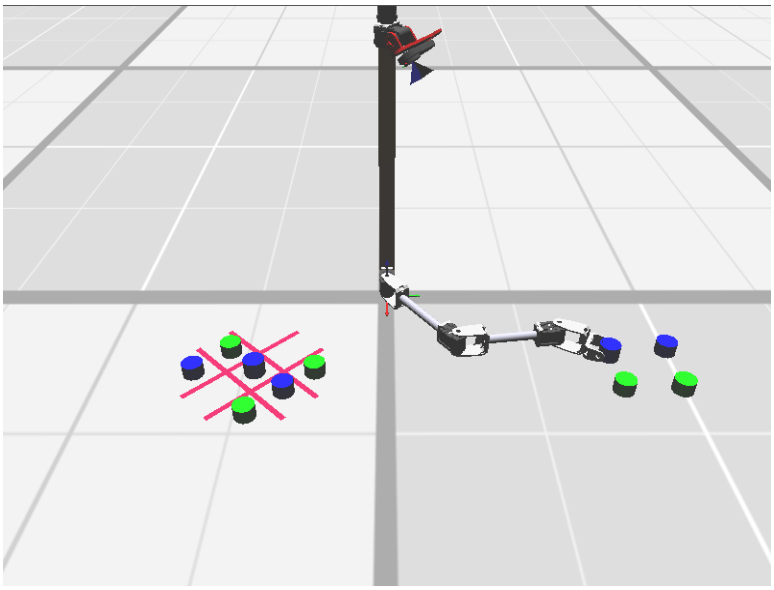

Figure 26 – Game pieces to the far right will be hard to reach

SpawnNextPiece places a green or blue game piece, depending on which color piece was last played, in a pre-determined position away from the board where the arm can

easily grasp it. Once the next game piece has been placed, the node FindPieces makes the Hand/Eye look around for all of the game pieces in its environment.

The last two nodes, MovePiece and SweepPieces, demonstrate how easy it is to program the Hand/Eye robot to manipulate an object. The MovePiece node uses the Grasper to move the next game piece onto the board, after the FindPieces node is complete. The SweepPieces node is invoked at the end of the game to clear the board.

The MovePiece node sets up a Grasper request to move the next game piece onto the board. First, the target location on the board is set.

```
graspreq.targetLocation = targetLocationOnBoard;
```

"targetLocationOnBoard" is the random position that was picked by the ParseBoard node. Next the object to be moved and the obstacles are set.

```
graspreq.object = targetObject;
graspreq.envObstacles = obstacles;
```

The MovePiece node now sets two variables that will let the Grasper know to disengage the arm from the object after it has been placed and what configuration to move the arm to.

```
graspreq.restType = GrasperRequest::settleArm;
graspreq.armRestState = {0,0,0} // {shoulder, elbow, wrist} angles
```

The MovePiece node also sets some RRT parameters:

- numberOfStatesForRRT: the number of states the RRT should allocate to use
- RRTTolerance: maximum distance between two states
- RRTstepsize: max angular distance a joint can move per execution
- RRTItrStepsize: max angular distance a joint can turn during an interpolation

Once these variables are set, the Grasper proceeds to formulate a plan to grasp the

object, move the object onto the board, and then disengage the arm from the object.

The SweepPieces node sets up a request for the Grasper to sweep all of the game pieces off of the board. This node does not set the graspreq.object variable because it is not going to move a particular object. Like the MovePiece node it does set the RRT parameters and the graspreq.envObstacles variable. The SweepPieces node also sets two additional variables that are necessary for a sweep operation.

```
graspreq.sweepStartPos = 90.0 * (M_PI/180); (shoulder angle; elbow angle = 0.0)
gaspreq.sweepDirection = -180.0 * (M_PI/180);
```

The graspreq.sweepStartPos variable tells the Grasper where to start the sweep from and the graspreq.sweepDirection variable tells the Grasper what direction to sweep and how far to sweep. The wrist is automatically turned 45° in the direction of the sweep. This is to keep the objects from slipping away from the arm.

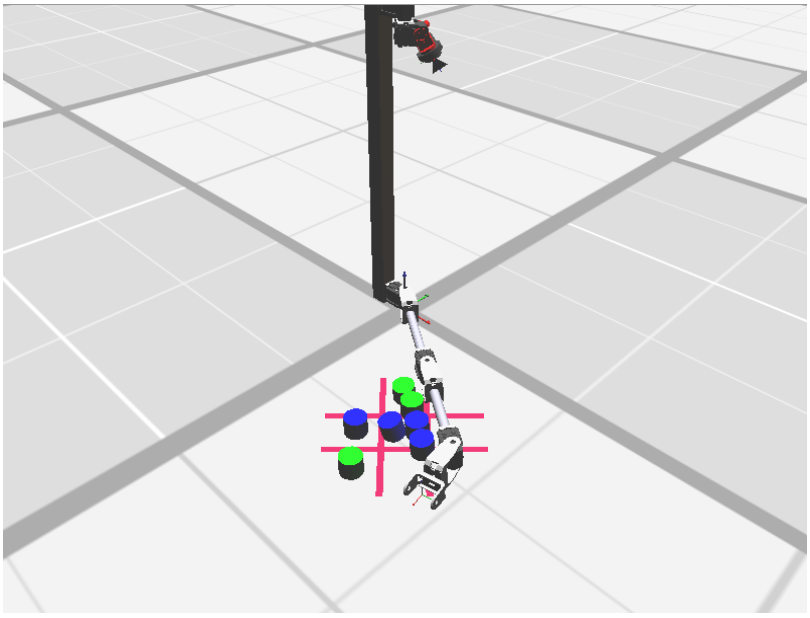

Figure 27 – Hand/Eye sweeping objects from left to right

After these variables have been set the Grasper proceeds to formulate a plan to move the arm to the graspreq.sweepStartPos configuration, while avoiding all obstacles in

graspreq.envObstacles. Then it swings the arm about the shoulder according to the graspreq.sweepDirection variable.

12 Conclusions

Object manipulations can vary in complexity from a very simple task of moving something from a start to a destination in a straight line, to moving an object around obstacles. Manipulations can also be as complex as requiring obstacles to be moved out of the way then moving the object to its destination. The latter can be broken down into a series of simpler manipulations with a little extra planning. However, no matter how simple the desired manipulation there is still a lot of work that has to be done. The object must be acquired then moved and in some cases the manipulator must also be moved away from the object. These three steps require path planning, collision detection and both forward and inverse kinematics. These steps must be planned and executed in a timely manner. The manipulation planner developed in this research does just that.

The manipulation planner gives developers the ability to easily program a robot to manipulate an object with a three-link planar arm. It is very easy to use, requiring only a few lines of code to setup the planner and one line to execute the manipulation. The planner allows developers to focus on the big picture, when developing applications such as a tic-tac-toe player, instead of the low level details of how the robot will move the game pieces around.

Persons studying path planning, collision detection and kinematics can use the planner as a tool to better understand these topics. Students can program a robot to perform simple manipulations and see the robots use the above-mentioned algorithms to perform

the manipulation. Tekkotsu is an open source software platform, which means the code can be read and modified to enhance one's comprehension of what and how the planner performs manipulations.

13 Future Work

Work on this project can be continued in many ways. Currently the Grasper only performs manipulations using the inside of the c-bracket. A more complex Grasper will be able to plan manipulations using the outer sides of the two fingers or any other part of the arm.

In the Graspers current state, it is assumed that all manipulations, once planned, are performed without error. Preferably the Grasper should be able to detect if a desired manipulation was properly performed and if not it should automatically correct the mistake. This could be achieved either by verifying that the target object is at the target location or by tracking the movement of the object during the manipulation, ensuring that is always in the c-bracket. If either of these fails, the Grasper would locate the target object and re-plan the manipulation.

Grasping an object is very difficult if it is too close to obstacles. Sometimes placing an object at a target location is impossible because one or two obstacles are in the way. Future work on the Grasper could enable it to move these obstacles aside, grasp or place the target object, then if need be move the obstacles back in place.

Being able to accurately and efficiently move objects around with a three-link planar arm offers limitless capabilities for Tekkotsu developers. Eventually one might even be able to manipulate moving targets.

References

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

[2] G. V. Nickens, E. J. Tira-Thompson, et al, An Inexpensive Hand-Eye System for Undergraduate Robotics Instruction. SIGSCE-2009. March 7, 2009. Chattanooga, TN

[3] M. Spong, S. Hutchinson, M. Vidyasagar. Robot Modeling and Control. John Wiley and Sons, Inc., 2005.

[4] Wikipedia. Separating axis theorem. http://en.wikipedia.org/wiki/Separating_axis_theorem, accessed Feb 2011

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

Appendix A: Source Code

CBracketGrasperPredicate.h

```
#ifndef _CBRACKET_GRASPER_PREDICATE_
#define _CBRACKET_GRASPER_PREDICATE_

#include <Planners/RRT/RRTPlanner.h>
#include <Motion/Kinematics.h>
#include <Motion/PlanarThreeLinkArm.h>

class CBracketGrasperPredicate : public RRTFunctorBase {
	KinematicJoint *GripperFrameKJ, *tmpFK, *FKjoints[NumArmJoints];
	PlanarThreeLinkArm functorArm;
	PlanarThreeLinkArm::Solutions functorSol;
	fmat::Column<3> fromPT, toPT, goalPT;
	float fromOri, toOri, dir;

public:
	CBracketGrasperPredicate(): GripperFrameKJ(), tmpFK(), functorArm(), functorSol(), fromPT(),
toPT(), goalPT(), fromOri(), toOri(), dir() {
#ifdef TGT_CALLIOPE
		return;
#endif

#ifdef TGT_HAS_ARMS
		GripperFrameKJ = kine->getKinematicJoint(GripperFrameOffset)->cloneBranch();
		if(GripperFrameKJ != NULL) {
			tmpFK = GripperFrameKJ->getRoot();
			tmpFK->buildChildMap(FKjoints, ArmOffset, NumArmJoints);
		}
#endif
	}

	~CBracketGrasperPredicate() { delete GripperFrameKJ->getRoot(); }

	virtual bool operator()(RRTState* from,
				RRTState* to,
				const RRTState* goal,
				const KinematicJoint* baseFrame,
				const KinematicJoint* effectorFrame,
				RRTStateVector& vec,
				RRTPlanner& planner,
				bool forward,
				std::vector<PlannerObstacle*> obstacles,
				bool postProcess) {
		const float range = 40 * (M_PI/180); // The most the c-bracket can turn without loosing
the object

		fromPT = gripperPosition(from);
		toPT = gripperPosition(to);
		goalPT = gripperPosition(goal);
		fromOri = stateOrien(from);
		toOri = stateOrien(to);
		float aDist, nOri;
		dir = (forward) ? atan2(toPT[1]-fromPT[1], toPT[0]-fromPT[0]) : atan2(fromPT[1]-
toPT[1], fromPT[0]-toPT[0]);
```

```
		aDist = (forward) ? angDist(fromOri, dir) : angDist(toOri, dir);
		if( aDist > range ) {
			nOri = newOri(fromPT, goalPT, fromOri);
			functorSol = functorArm.invKin3LinkRelaxPhi(fromPT[0], fromPT[1], nOri);
			int solNo =	( functorSol.valid ) ? (( functorSol.noSols == 1 ) ? ((
signof(functorSol.angles(0,1)) == signof(from->vec[1]) ) ? 0 : -1) : (( signof(functorSol.angles(0,1)) ==
signof(from->vec[1]) ) ? 0 : 1)) : -1;
			if(solNo == -1) { return false; }
			for(unsigned j = 0; j < NumArmJoints; j++) { to->vec[j] =
functorSol.angles(solNo,j); }
			if( planner.hasCollisions(to) ) { return false; }
		}
		to->parent = from;
		from->addChild(to);
		return ( to->distFrom(goal) < from->distFrom(goal) ) ? true : false;
	}

	int signof(float a) { return (a == 0.0) ? 1 : (a < 0.0 ? -1 : 1); }

	float newOri(fmat::Column<3>& from, fmat::Column<3>& to, float fOri) {
		const float change = 5 * (M_PI/180);
		float diff = fOri - atan2(to[1]-from[1], to[0]-from[0]);
		return AngSignPi( fOri + (fabs(diff) < M_PI ? 1 : -1) * ((diff < 0) ? 1 : -1) * change);
	}

	fmat::Column<3> gripperPosition(const RRTState* FKstate) {
		for(unsigned int j = 0; j < NumArmJoints; j++)
			FKjoints[j]->setQ(FKstate->vec[j]);
		return GripperFrameKJ->getWorldPosition();
	}

	fmat::Column<3> gripperPosition(const RRTStateDef FKstate) {
		for(unsigned int j = 0; j < NumArmJoints; j++)
			FKjoints[j]->setQ(FKstate[j]);
		return GripperFrameKJ->getWorldPosition();
	}

	float stateOrien(const RRTState* FKstate) {
		float ori = 0.0;
		for(unsigned int j = 0; j < NumArmJoints; j++)
			ori += FKstate->vec[j];
		return AngSignPi(ori);
	}

	static float angDist(float a1, float a2) {
		float angle = fmod((float)fabs(a1 - a2),(float)(2*M_PI));
		return ( angle > M_PI ) ? (2*M_PI) - angle : angle;
	}

private:
	CBracketGrasperPredicate& operator=(const CBracketGrasperPredicate &mp);
	CBracketGrasperPredicate(const CBracketGrasperPredicate& mp);
};

#endif
```

RandomTictactoe.h.fsm

```
#include "Behaviors/StateMachine.h"
#include "Wireless/netstream.h"

using namespace std;
using namespace DualCoding;

typedef DualCoding::Shape<DualCoding::LineData> ShLine;
typedef std::vector< ShLine > LineVec;
typedef std::vector< DualCoding::Shape<DualCoding::EllipseData> > EllipseVec;
typedef std::vector< DualCoding::Sketch<bool> > SkBoolVec;

static int nextPosition = -1;

#nodeclass RandomTictactoe : VisualRoutinesStateNode

	#nodeclass WhoGoesFirst : StateNode
		#nodemethod doStart
			std::cout << "\nType yes or no in the 'Send Input' field, then hit Enter...\n" <<
std::endl;
			erouter->addListener(this,EventBase::textmsgEGID);  // and text message events
		#endnodemethod
		#nodemethod doEvent
			switch(event->getGeneratorID()) {
				case EventBase::textmsgEGID: {
					const TextMsgEvent *txtev = dynamic_cast<const
TextMsgEvent*>(event);
					if (txtev->getText() == "yes") {
						postStateCompletion(); }
					else {
						postStateFailure(); }
					break;};

				default:
					std::cout << "Unexpected event: " << event->getDescription()
<< std::endl;
			}
		#endnodemethod
	#endnodeclass

	#nodeclass ReadBoard : MapBuilderNode($,MapBuilderRequest::worldMap) : doStart
		NEW_SHAPE(gazePt, PointData, new PointData(localShS, Point(230,-
230,30,egocentric)));
		mapreq.searchArea = gazePt;
		mapreq.addObjectColor(lineDataType, "pink");
		mapreq.addObjectColor(ellipseDataType, "blue");
		mapreq.addObjectColor(ellipseDataType, "green");
		mapreq.addOccluderColor(ellipseDataType, "blue");
		mapreq.addOccluderColor(ellipseDataType, "green");
		mapreq.groundPlaneAssumption = MapBuilderRequest::custom;
		mapreq.customGroundPlane = PlaneEquation(0,0,1,30);
		mapreq.motionSettleTime = 1000;
		mapreq.rawY = true;
		mapreq.maxDist = 2000;   // millimeters
```

```
	#endnodeclass

	#nodeclass ParseBoard : StateNode
		#nodemethod doStart
			LineVec lines = parseLines();
			if(lines.size() < 4)
				{ postStateFailure(); return; }
			SkBoolVec squares = parseBoundaries(lines[0], lines[1], lines[2], lines[3]);
			if(squares.size() < 9)
				{ postStateFailure(); return; }
			vector<int> positions = parsePieces(squares);
			vector<int> availablePositions = convertBoard(positions);
			if(availablePositions.size() == 0) {
				std::cout << "The board is full!" << std::endl;
				postStateFailure();
				return; }
			srand(time(NULL));
			nextPosition = availablePositions[rand() % availablePositions.size()];
			postStateCompletion();
		#endnodemethod
			LineVec parseLines();
			SkBoolVec parseBoundaries(const ShLine& topLine, const ShLine&
bottomLine, const ShLine& leftLine, const ShLine& rightLine);
			SkBoolVec constructSquares(const ShLine& topLine, const ShLine&
bottomLine, const ShLine& leftLine, const ShLine& rightLine, const ShLine& topBoundary, const
ShLine& bottomBoundary, const ShLine& leftBoundary, const ShLine& rightBoundary);
			vector<int> parsePieces(const SkBoolVec& squares);
			vector<int> convertBoard(vector<int> positions);
	#endnodeclass

	#nodeclass SpawnNextPiece(string color) : StateNode : doStart
		int x_coord[] = { 336, 280, 230, 280, 230, 175, 230, 175, 124};
		int y_coord[] = {-230,-280,-336,-175,-230,-280,-125,-175,-230};
		std::cout << "Move the " << color << " piece to position {" << x_coord[nextPosition] <<
"," << y_coord[nextPosition] << "}" << std::endl;
		static int id = 1;
		char buf[22];
		ionetstream mirage;
		if(!mirage.open("localhost",19785u)) {
			std::cerr << "Connection to mirage refused" << std::endl;
			postStateFailure();
			return;
		}
		plist::Dictionary msg;
		if(color == "green") {
			sprintf(buf, "GreenEggMarker%d", id);
			msg.addValue("ID", buf);
		}
		else {
			sprintf(buf, "BlueEggMarker%d", id);
			msg.addValue("ID", buf);
		}
		id++;
		msg.addValue("Persist",true); // want points to stick around in Mirage
		KinematicJoint nextPiece;
		plist::ArrayOf<plist::Primitive<float> > location(3,0);
```

```
		fmat::Column<3> pos = fmat::pack(200,200,12.5);
		pos.exportTo(location);
		msg.addEntry("Location", location);
		nextPiece.mass = 20;
		nextPiece.model = "CollisionModel";
		nextPiece.material = (color=="green") ? "Green" : "Blue";
		nextPiece.collisionModel = "Cylinder";
		nextPiece.collisionModelScale = fmat::pack(37,37,30);
		nextPiece.centerOfMass = fmat::pack(0,0,-20);
		msg.addEntry("Model", new KinematicJointSaver(nextPiece));
		mirage << "<messages>\n";
		msg.saveStream(mirage,true);
		mirage << "</messages>";
		postStateCompletion();
	#endnodeclass

	#nodeclass FindPieces : MapBuilderNode($,MapBuilderRequest::worldMap) : doStart
		mapreq.addObjectColor(ellipseDataType, "blue");
		mapreq.addObjectColor(ellipseDataType, "green");
		const vector<Point> gazePts = Lookout::groundSearchPoints();
		NEW_SHAPE(gazePoly, PolygonData, new PolygonData(worldShS, gazePts, true));
		mapreq.searchArea = gazePoly;
		mapreq.groundPlaneAssumption = MapBuilderRequest::custom;
		mapreq.customGroundPlane = PlaneEquation(0,0,1,30);
		mapreq.motionSettleTime = 1000;
	#endnodeclass

	#nodeclass MovePiece : GrasperNode($,GrasperRequest::moveTo) : doStart
		int x_coord[] = { 336, 280, 230, 280, 230, 175, 230, 175, 124};
		int y_coord[] = {-230,-280,-336,-175,-230,-280,-125,-175,-230};
		NEW_SHAPE(target, PointData, new PointData(worldShS,
DualCoding::Point(x_coord[nextPosition], y_coord[nextPosition], 30, egocentric)));
		graspreq.targetLocation = target;
		SHAPEROOTVEC_ITERATE(worldShS, s)
		if (s->isType(ellipseDataType)) {
			if (s->getCentroid().coordY() > 0)
				graspreq.object = s;
			else
				graspreq.envObstacles.push_back(s);
		}
		END_ITERATE;
		graspreq.restType = GrasperRequest::settleArm;
		graspreq.RRTItrStepsize = 0.5*M_PI/180.0;
		graspreq.numberOfStatesForRRT = 100000;
		graspreq.RRTstepsize = 0.5*M_PI/180.0;
		graspreq.maxRRTIterations = 100000;
		graspreq.RRTTolerance = 0.01f;
		graspreq.armRestState = 0.0;
	#endnodeclass

	#nodeclass SweepPieces : GrasperNode($,GrasperRequest::sweep) : doStart
		SHAPEROOTVEC_ITERATE(worldShS, s)
			if (s->isType(ellipseDataType))
				graspreq.envObstacles.push_back(s);
		END_ITERATE;
		graspreq.restType = GrasperRequest::settleArm;
```

```
                graspreq.RRTItrStepsize = 0.5*M_PI/180.0;
                graspreq.numberOfStatesForRRT = 100000;
                graspreq.RRTstepsize = 0.5*M_PI/180.0;
                graspreq.maxRRTIterations = 100000;
                graspreq.RRTTolerance = 0.01f;
                graspreq.armRestState = 0.0;
                graspreq.sweepStartPos = 90.0 * (M_PI/180);
                graspreq.sweepDirection = -180.0 * (M_PI/180);
        #endnodeclass

        #nodemethod setup
                #statemachine
                        startnode: SpeechNode("Should blue play first?") =C=> wgf
                        wgf: WhoGoesFirst =C=> player1
                        wgf =F=> player2
                        player1: ReadBoard =MAP=> p1pb
                        player2: ReadBoard =MAP=> p2pb
                        p1pb: ParseBoard =C=> SpawnNextPiece($,"blue") =C=> FindPieces =MAP=>
player1Move
                        player1Move: MovePiece
                        player1Move =GRASP(noError)=> SpeechNode("Next move.") =C=> player2
                        player1Move =GRASP(someError)=> SpeechNode("Sorry! I cannot move the
blue piece.") =C=> sweep
                        p2pb: ParseBoard =C=> SpawnNextPiece($,"green") =C=> FindPieces
=MAP=> player2Move
                        player2Move: MovePiece
                        player2Move =GRASP(noError)=> SpeechNode("Next move.") =C=> player1
                        player2Move =GRASP(someError)=> SpeechNode("Sorry! I cannot move the
green piece.") =C=> sweep
                        p1pb =F=> SpeechNode("Sorry, I could not parse the board.")
                        p2pb =F=> SpeechNode("Sorry, I could not parse the board.")
                        sweep: SweepPieces =GRASP(noError)=> SpeechNode("Please restart me!")
                #endstatemachine
        #endnodemethod
#endnodeclass
```

RandomTictactoe.cc

```
#include "RandomTictactoe.h"

LineVec RandomTictactoe::ParseBoard::parseLines() {
	LineVec boardLines;
	// 1. sort by length
	LineVec lines=select_type<LineData>(camShS);
	lines = stable_sort(lines,not2(LineData::LengthLessThan()));
	if ( lines.size() < 4 ) {
	cout << "Found " << lines.size() << " lines in the image; needed 4." << endl;
	return boardLines;
	}
	// 2. Find the top and bottom horizontal lines
	Shape<LineData> topLine, bottomLine;
	for(LineVec::const_iterator ln1=lines.begin(); ln1!=lines.end(); ++ln1) {
		if ( LineData::IsHorizontal()(*ln1) ) {
			for(LineVec::const_iterator ln2=ln1+1; ln2!=lines.end(); ++ln2) {
				if ( LineData::ParallelTest()(*ln1,*ln2) ) {
					topLine = IsAbove()(*ln1,*ln2) ? *ln1 : *ln2;
					bottomLine = IsAbove()(*ln1,*ln2) ? *ln2 : *ln1;
					break;
				}
			}
		}
		if ( bottomLine.isValid() )
			break;
	}
	if ( ! bottomLine.isValid() ) {
		cout << "Couldn't find top or bottom line" << endl;
		return boardLines;
	}
	topLine->V("topLine");
	bottomLine->V("bottomLine");
	// 3. Find the left and right sort-of-vertical lines
	Shape<LineData> leftLine, rightLine;
	for(LineVec::const_iterator ln1=lines.begin(); ln1!=lines.end(); ++ln1) {
		if ( !LineData::ParallelTest()(topLine,*ln1) ) {
			for(LineVec::const_iterator ln2=ln1+1; ln2!=lines.end(); ++ln2) {
				if ( ! LineData::ParallelTest()(topLine,*ln2) ) {
					leftLine = IsLeftOf()(*ln1,*ln2) ? *ln1 : *ln2;
					rightLine = IsLeftOf()(*ln1,*ln2) ? *ln2 : *ln1;
					break;
				}
			}
		}
		if ( rightLine.isValid() )
			break;
	}
	if ( ! rightLine.isValid() ) {
		cout << "Couldn't find left or right line" << endl;
		return boardLines;
	}
	leftLine->V("leftLine");
	rightLine->V("rightLine");
```

```
	// return lines in specified order
	boardLines.push_back(topLine);
	boardLines.push_back(bottomLine);
	boardLines.push_back(leftLine);
	boardLines.push_back(rightLine);
	return boardLines;
}

SkBoolVec RandomTictactoe::ParseBoard::parseBoundaries(const ShLine& topLine, const ShLine&
bottomLine, const ShLine& leftLine, const ShLine& rightLine) {
	// Construct board boundary lines
	Point tl = topLine->leftPt();
	Point tr = topLine->rightPt();
	Point bl = bottomLine->leftPt();
	Point br = bottomLine->rightPt();
	Point lt = leftLine->topPt();
	Point lb = leftLine->bottomPt();
	Point rt = rightLine->topPt();
	Point rb = rightLine->bottomPt();
	NEW_SHAPE(leftBoundary, LineData,
			new LineData(camShS, leftMost(tl,bl), leftLine->getOrientation()));
	NEW_SHAPE(rightBoundary, LineData,
			new LineData(camShS, rightMost(tr,br), rightLine->getOrientation()));
	NEW_SHAPE(topBoundary, LineData,
			new LineData(camShS, topMost(lt,rt), topLine->getOrientation()));
	NEW_SHAPE(bottomBoundary, LineData,
			new LineData(camShS, bottomMost(lb,rb), bottomLine->getOrientation()));
	return
constructSquares(topLine,bottomLine,leftLine,rightLine,topBoundary,bottomBoundary,leftBoundary,right
Boundary);
}

SkBoolVec RandomTictactoe::ParseBoard::constructSquares(const ShLine& topLine, const ShLine&
bottomLine, const ShLine& leftLine, const ShLine& rightLine, const ShLine& topBoundary, const
ShLine& bottomBoundary, const ShLine& leftBoundary, const ShLine& rightBoundary) {
	//include the border itself for better robustness
	NEW_SKETCH(board, bool, !(
							visops::leftHalfPlane(leftBoundary) |
							visops::rightHalfPlane(rightBoundary) |
							visops::topHalfPlane(topBoundary) |
							visops::bottomHalfPlane(bottomBoundary)
							));

	// Construct regions for board rows and columns
	NEW_SKETCH(topRow, bool, visops::topHalfPlane(topLine) & board);
	NEW_SKETCH(bottomRow, bool, visops::bottomHalfPlane(bottomLine) & board);
	NEW_SKETCH(midRow, bool, ! (topRow | bottomRow) & board);

	NEW_SKETCH(leftCol, bool, visops::leftHalfPlane(leftLine) & board);
	NEW_SKETCH(rightCol, bool, visops::rightHalfPlane(rightLine) & board);
	NEW_SKETCH(midCol, bool, ! (leftCol | rightCol) & board);

	// Construct regions for the 9 board squares by intersecting rows and columns
	SkBoolVec squares(9);
	for (int i=0; i<3; i++) {
		squares[i].bind(visops::copy(topRow));
```

```
		squares[i+3].bind(visops::copy(midRow));
		squares[i+6].bind(visops::copy(bottomRow));
	}
	for (int i=0; i<3; i++) {
		squares[i*3] &= leftCol;
		squares[i*3+1] &= midCol;
		squares[i*3+2] &= rightCol;
	}
	return squares;
}

vector<int> RandomTictactoe::ParseBoard::parsePieces(const SkBoolVec& squares) {
	// Find the game piece bottoms
	NEW_SHAPEVEC(ellipses, EllipseData, select_type<EllipseData>(camShS));
	NEW_SHAPEVEC(x_pieces, EllipseData, subset(ellipses, IsColor("blue")));
	NEW_SHAPEVEC(o_pieces, EllipseData, subset(ellipses, IsColor("green")));
	NEW_SKETCH(x_render, bool, visops::zeros(camSkS));
	NEW_SKETCH(o_render, bool, visops::zeros(camSkS));
	DO_SHAPEVEC(x_pieces, EllipseData, piece, {
		x_render |= piece->getRendering();});
	DO_SHAPEVEC(o_pieces, EllipseData, piece, {
		o_render |= piece->getRendering();});
	NEW_SKETCH(x_bottoms, bool, x_render & ! x_render[*camSkS.idxS]);
	NEW_SKETCH(o_bottoms, bool, o_render & ! o_render[*camSkS.idxS]);
	int minBottom = 2; //!< minimum area to consider for a piece bottom (noise filter)
	x_bottoms = visops::areacc(x_bottoms)>minBottom;
	o_bottoms = visops::areacc(o_bottoms)>minBottom;

	// Intersect piece bottoms with board regions to determine occupancy of each square
	int xIdx = ProjectInterface::getColorIndex("blue");
	int oIdx = ProjectInterface::getColorIndex("green");
	vector<int> squareValues(9,0);
	for (int i=0; i<9; i++)
		if ( ! ((squares[i] & x_bottoms)->empty()) )
			squareValues[i] = xIdx;
			else if ( ! ((squares[i] & o_bottoms)->empty()) )
				squareValues[i] = oIdx;
				return squareValues;
}

vector<int> RandomTictactoe::ParseBoard::convertBoard(vector<int> positions) {
	vector<int> availablePositions;
	for (unsigned int a = 0; a < positions.size(); a++) {
		if ((positions[a] == 0)) {
			//std::cout << "Position " << a+1 << " is unoccupied." << std::endl;
			availablePositions.push_back(a);
		}
	}
	return availablePositions;
}
```