4-28-2000

Computer Vision: Object Recognition

Michael Zalokar '00
Illinois Wesleyan University

Computer Vision: Object Recognition

By Michael Zalokar and Dr. Lon Shapiro*

Illinois Wesleyan University, Bloomington, Illinois

Department of Mathematics and Computer Science

Friday, April 28, 2000

Table of Contents:

Abstract

One of the growing fields in computer science is that of Artificial Intelligence or AI. Many theories have evolved to make a computer intelligent and so far no one has succeeded (Dreyfus 1992). One of the methods used by the Shelley Project in the past has been to use a back propagation neural network that is the backbone of the GNU Neural Network Visualizer (GNNV). GNNV uses a neural network to try to identify known objects, like faces, in the field of view. A different method, that is the focus of this research, is to identify objects in the image. These objects could be squares, circles or even blobs. Neural networks can work through changes in environment without changing the code provided appropriate training. However it is tough to know what the neural network is actually learning. One advantage this research has over neural networks is that as the programmer you know exactly what it knows. Instead, the problems are of the form, "How do I tell it what a circle is?" or "How do I have it determine what is noise that should be ignored?" The goal of this project is to create a program capable of taking in an image from a digital camera and identifying the tic-tac-toe game. This is inspired from past work done for the Shelley Project which included playing tic-tac-toe (without the vision component) and the Shelley Integrated Environment (SIE). Various problems arose during implementation. The majority of these were system and API related. For others like the perspective correction and game detection, stepping away from the computer with pencil and paper in hand was invaluable. Through it all the goal of playing a game of tic-tac-toe against Shelley has finally become a reality.

Introduction:

One measure of how well a piece of software works is to look at how user friendly it is. This is because computers are unable to interact with people the same way that people interact with each other. One growing field in computer science is that of computer vision, which tries to give a computer the dominant human sense of vision. When sighted people meet they do not smell the other person nor taste nor touch them either (at least not at first).

The goal of this project is to be able to have the computer, Shelley, play a completely interactive game of tic-tac-toe with people. This starts with Shelley using her camera to see the board and ends with her using one of her robotic arms to make her move. The target for the vision system to identify is a white wooden board 15 centimeters square and one centimeter high. It is divided into nine sections separated by narrow cuts into the wood filled with black tubing. Each section has a small divot drilled into the center. In these small holes are where the marbles are placed by each player. The marbles are two centimeters in diameter. They are in two colors: red and green. See Figure 1.

Figure 1

This research concentrates on the vision component required for Shelley to play a game. However, some explanation of how the vision system fits into the entire picture is needed. This consists of four sections not including the tic-tac-toe vision system (`ttt_vision`). See Figure 2.

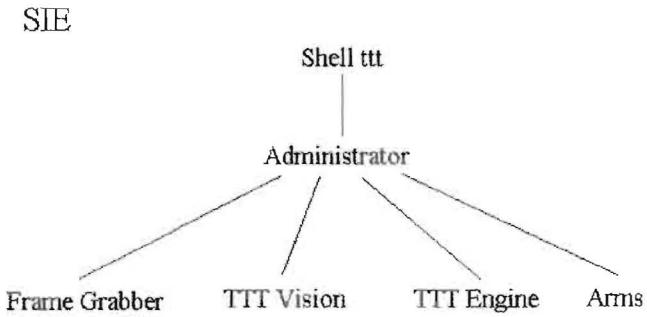

Figure 2

The name for the entire collection of programs for Shelley is the Shelley Integrated Environment (SIE). Andy Ritger started SIE in 1999 for research honors as part of the Shelley Project (Ritger 1999). In his paper, he explains how the various Shelley programs should work with each other. The main element in SIE is the administrator. The administrator is the central program that handles access by the purely

software side of SIE to the hardware devices like Shelley's cameras and arms. The administrator is a full deamon or server program. This will allow for new programs written without any change to the administrator itself. With the interfaces to the camera and arms already written for this project future contributions will be able to reuse these features without rewriting them.

In order for Shelley to play tic-tac-toe there is a main shell program (`shell_ttt`). This program or agent in SIE terminology, is where the device requests are made and all game control is handled. The shell interfaces with each device via a C++ wrapper class. For example, the `Frame_Grabber` wrapper class is used to interface with the `frame_grabber` device. In this way, once a wrapper class has been written for one device, like the Frame Grabber, then any future agents can use this wrapper class to interact with the device. This design feature will help future contributors to the Shelley Project since they will not have to redo work already completed.

The Frame Grabber is the camera interface for Shelley. It was rewritten for this research. Andy Ritger wrote the original, grayscale, version of the Frame Grabber as part of his research honors. This program was rewritten to handle color that was required for this project (Sun, "Video" 1994, "Reference" 1997 and "Programmer's" 1997). Both versions were based on an example program supplied by Sun Microsystems named `rtvc_display` (Sun, "rtvc" 1994).

Another SIE component is the tic-tac-toe game engine itself. Originally written in 1996 by Matt Weaver and Chris Stewart the game engine was made SIE compliant by Lon Shapiro for this research. The game engine remembers past games it has played

thereby learning and playing better the more games that get played. It is at this stage of processing that the rules of tic-tac-toe are evaluated. Should the human opponent make an illegal move or is cheating it is at this point that it is discovered and the error reported.

Another, less complicated, game engine was written by Rob Glinka and Becky Schroeder (Glinka, "game" 2000). The computational organization is based on the version written by Matt and Chris without all of the learning functionality. Moves are chosen randomly unless Shelley realizes that there are two of her marbles in a row. In this case she will make the winning move.

After the game engine determines a move to make, the arms use this information and make Shelley move by placing a marble on the board. Craig Materick originally wrote arm control code for Shelley to play with in 1996 for research honors (Materick 1996). Rob Glinka made the necessary changes to make the code work under Linux instead of DOS as it had originally (Glinka, "arms" 2000).

Shelley[1] is a Sun Microsystems UltraSparc 1 with a 143 MHz speed processor running the Solaris 2.6 operating system and with 128 megabytes of RAM. Shelley's eyes consist of two Sun Microsystems SunVideo cameras, though only one was used for this research. The UNIX device name for the camera is rtvc. One of three Intel Pentium 100s running Red Hat Linux 6.1 handles the arm control. Shelley's arms are Robix Kits using Hitech brand servos. The SV203 servo controllers by Pontech connect to the Pentium via an RS-232 serial port.

To play tic-tac-toe against Shelley there needs to be an administrator running on Shelley. Executing the administrator program located in the `sie/bin/` directory can

[1] The Shelley Project takes its name from Mary Shelley the author of Frankenstein.

start an administrator. On the Pentium the servo server needs to be running too. This is achieved by starting the `servo` (`arms/servo/`) executable in a terminal. In a second terminal window on Shelley launch the `shell_ttt` program from the same directory as the administrator. After the `shell_ttt` program is started the screen will change to different colors for a few seconds. This change is because the `ttt_vision` has just been started and is in the process of initializing. The change in colors happens because Shelley has an eight-bit graphics card. Machines that represent color with a 24 bit graphics representation or greater will not have this problem. When the `frame_grabber` is launched a second window will pop up and disappear shortly afterward. Next, the `ttt_engine` is started. There is no graphical display connected to the game engine. Finally, the connection from Shelley to the `servo` server is established. By default Shelley waits for the player to move first and is ready to play when the first frame is displayed in the graphics window.

Reasoning:

Previous work in the Shelley Project has worked on included the GNU Neural Network Visulizer (GNNV). The neural network model that GNNV uses to identify people in an image is a solution to artificial intelligence modeled after human learning (Shufelt 1994). GNNV's method has some general advantages. It does not need to be reprogrammed to handle a different vision problem. By simply changing the set of images it "learns" with it can theoretically be trained to identify any type of object. However, this leads to the problem that GNNV and similar models run into for programmers; "What is it really learning?"

This problem does not have a simple solution. What this research does is add to the Shelley Project an alternative model for computer vision than has been used in the past. Here Shelley is given the knowledge to identify what the target is. The trick is to make the process general enough to handle environmental changes, but not so general as to become confused easily.

A few secondary goals have made significant impacts the research. The first is that the Shelley Project is an ongoing project. Each year new projects are built from projects done in the past. In order to continue this evolving project the work has to be simple enough for others to understand so they can make changes to the code as they see fit. Therefore it will often be said that an algorithm was chosen for its simplicity.

Another secondary goal is to write the code in an easily upgradeable manner. A lot of work went into avoiding implementing the vision specifically for tic-tac-toe while encapsulating as much as possible. Reasons for this are listed in the Future Work section later on.

Preprocessing:

The vision code works in seven main steps (see Appendix A, `ttt_vision`). These are divided into two groups: preprocessing and processing. The first group of steps consists of obtaining the image that is to be analyzed and finding the important information that needs to be passed onto the actual processing steps.

1. Perspective Correction
2. Edge Detection
3. Corner Identification
4. Detect Game
5. Moravec Filter

1. Perspective Correction

The first preprocessing step is to remove camera perspective from the image. Camera perspective results from parallel lines coming together as they stretch farther away from a viewpoint. Thus a circular objects appears elliptical or in this case a square object looks like a trapezoid. See Figure 3. Without removing this effect from the image makes identifying a square tic-tac-toe board difficult because it does not have the shape of a square at this point.

Figure 3

Farther back objects are effected by this more than closer ones. The farther an object goes backward in the view the smaller it becomes. An object also appears higher up in the image the farther away it moves from the camera. The method used to eliminate the effect of perspective is to insert pixels into the image. The image is considered fixed when an object is the same size regardless of where it appeared in the original image. Since objects farther away should appear the same size in the fixed image, more pixels need to be inserted at the top than toward the bottom of the image.

An assumption is made for the insertions. In the case of the tic-tac-toe board the assumption is that the board is lying on a level surface. If it is not, then the perspective effect correction will not work correctly.

Pixels are inserted in the both the x-axis and y-axis directions (See `pers_corr()` in Appendix A, lines 593 - 685). More pixels are inserted into rows at the top than at the bottom as reasoned in the preceding paragraph. Thus taking a square object, like the tic-tac-toe board, when viewed at an angle and making the sides appear parallel to each other. The inserts in the X direction are inserted starting in the middle of the image. Starting at either side would result in an obscure image. This would also map significant and important portions of the image outside of the view port. There is some loss of data from the outer portions of the image being mapped out of the view port, but those are all toward the edges of the image. Thus one requirement for this method to work is that the target image not only exist in the camera's field of view, but also is inside the perspective correction view port. The differences from Figure 3 to Figure 4 include the obvious enlarging of the tic-tac-toe board. Also, note the change of shape to the red region on the right side of each figure. In the original figure the boundary edge is at an angle and in the corrected image it is nearly vertical.

Figure 4

The Y direction perspective correction inserts are started from the top. Since, this direction does not start in the middle like the X direction a significant portion of the image is mapped outside of the view port. Consequently in this direction a correction takes place to map pixels originally from the center of the image back to the center of the corrected image. The following equation is used for this mapping. Cy is the newly mapped Y coordinate, Cy is the calculated Y coordinate before the centering, h is the image height and p is the Y direction perspective correction.

$$My = (h / p) + Cy \qquad \text{(Eqn.1)}$$

The way both corrections work is by advancing one counter in a given direction along the original image. Each step in the image is one pixel in size. A second counter advances at a rate faster than the first. This counter indicates where the current original pixel should be mapped to in the corrected image. One advantage to this is that the correction counter can be specified to move at different values. Should the camera angle need to change, then a change to the value these counters step by can be changed to match that of the new camera angle.

Take for example an insertion value of 1.3 pixels to 1 pixel in the original. Since, 1.3 is not greater than the next whole pixel there is a direct one-to-one mapping for this pixel. The second and third pixels would be mapped to the pixels at 2.6 and 3.9 respectively. Since, they are not greater than the next whole integer there is a one-to-one correspondence from the original image to the corrected image. However, the fourth pixel is mapped to the pixel at 5.2 in the corrected image, so the original pixel is copied into the fourth and fifth pixels. See Figure 5.

Insertation Example

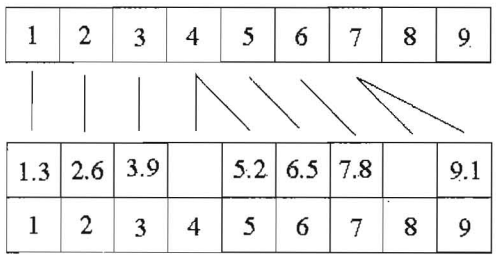

Figure 5

2. Edge Detection

Once the perspective correction is finished, the next step is to identify the edges of objects in the image. To store the edges of the objects a chain code implementation was chosen. The chain code algorithm uses an eight-connectivity model (See Appendix J, Freeman 1961). One reason that chain codes are used is that they only represent the edge of the object and not the entire object. This is reasonable since it is known what the tic-tac-toe board looks like. Therefore only the edges need to be located in the image. Another reason is that the implementation is not terribly difficult to get working. The actual implementation is done using a dynamically allocated doubly linked list. The

relative simplicity is a plus by not creating overly complex code for others to build future Shelley projects from.

The chain code is a list of pixels in order around the detected object. For each element in the list the absolute coordinate of the pixel in the image and the direction moved to get there from the previous pixel are stored. Direction is specified as an integer from zero to seven moving in a counter-clockwise direction. See Figure 6.

3	2	1
4	*	0
5	6	7

Figure 6

The direction is used to calculate the where to start looking for the next pixel. By adding five to the current direction this moves the start point one spot counter clockwise from going back in the direction it came from. The modulus operator takes this value and re-maps it within the interval [0, 7].

$$N = (D + 5) \% 8 \qquad \text{(Eqn. 2)}$$

The algorithm works by starting in the upper left-hand corner and checking each column from left to right (See `ShowChain()` in Appendix J, lines 255 – 326). The image is in color. By taking an average of the red, green and blue components of the pixels the luminance or brightness of the pixels are determined (See `img_pxl_avg()` in Appendix A, lines 517 – 535). This luminance is averaged for the entire image. Next the entire image is tallied for pixels with brightness above this average and those below. Whichever count is greater is declared background and the rest is foreground (See

`background()` in Appendix A, lines 537 – 564). This is done to remove variance that could change with regard to brightness. It also allows for the possibility that future games Shelley will play will not be restricted to having a bright board surrounded by a dark background. A boundary point of an object is found when a pixel is determined in the foreground of the image (See `checkThings()` in Appendix J, lines 50 – 79). Once, a point on an edge has been found it is used as the starting point for the chain code. If one of the neighbor pixels is found to be a boarder pixel then it is added to the end of the chain code and the same procedure is executed using this new pixel in the chain code.

The chain code is halted when there are two sequential elements found to occur twice with the same direction of connection associated with them (See `checkCode()` in Appendix J, lines 19 – 47). This allows for the same pixel to be visited twice where the boundary follows a one-pixel wide extension of the object without ending prematurely. In Figure 7 two pixels are found twice in the chain code. However, by observing the shape of the sample object it makes sense that those two pixels do get counted twice. This is the reason for including the same direction condition for halting the chain code.

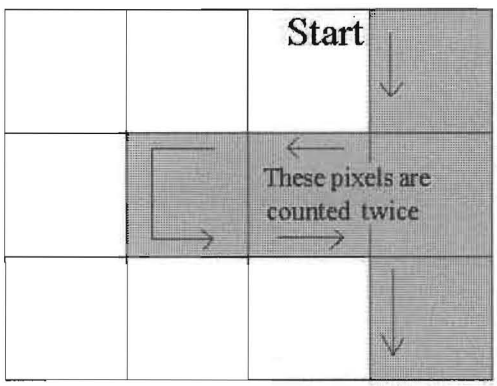

Figure 7

Figure 8 gives a pictorial example of a simple chain code. The pixel labeled start is where the object detection starts. The first step moves from this start pixel, which is not an edge pixel itself, onto the first pixel in the chain code. The direction moves down and right which corresponds to a direction value of seven. This direction value is used to indicate direction preference. Thus the algorithm continues to look down right instead of looking at the pixel to the immediate right. The third move is up. Finally, the chain code loops back onto the first pixel indicating that this chain code is done. Although, it does not technically stop until it checks that the next pixel is a repeat also.

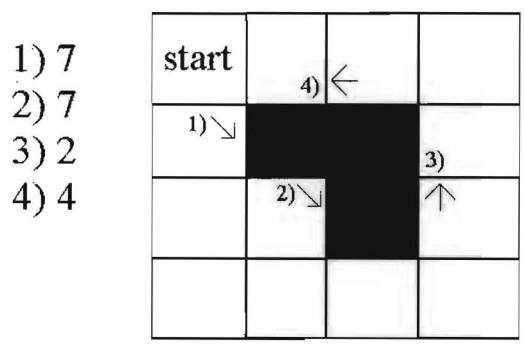

Figure 8

Three optimizations are implemented to speed up processing of the chain codes. The first is that if a chain code with a length of one is found it is ignored. The second is that the constant `MAX_CHAINS` specifies the maximum number of chain codes that are looked for (See Appendix K, line 10). If the view of the target is known to be free of excessive amounts of background noise a smaller value for `MAX_CHAINS` means less computation is required to determine which of the objects is in fact the tic-tac-toe board. Third is the constant `DECENT_SIZED_OBJECT` (See Appendix A, line 81). Before examining each chain code, if the length is less than this value then it is determined that it

is to small to be the target and it gets skipped. This helps eliminate unnecessary computation on small blobs with chain codes longer than one.

3. Corner Detection

When all of the "objects" have had their chain codes found, this information is used to find the corners of the object (See `detect_corners()` in Appendix A, lines 923 – 944). Some geometrical properties of squares are used to accomplish this task. It is assumed that the square target of the tic-tac-toe board has four corners connected together in a circuit. Each edge in this circuit has an even length. Another assumption exploits the property that the two diagonals are of the same length.

The corner detection works in two passes (See `findFirstTwoCorners()` and `findSecondTwoCorners()` in Appendix A, lines 774 – 921). The first pass identifies the first two corners and the second pass finds the last two (See Figure 9). The first set of corners is easy to find. Simply advance through the chain code looking for two points with the greatest distance between them. Unfortunately, the second set of corners is not as simple as doing a second pass looking for the second two farthest points (See Figure 10). The reason this does not work is that using the second longest distance between a different set of coordinates most often includes a neighbor pixel of the first two corners. To get around this problem we go back to the assumption that the tic-tac-toe board has two sets of distinct diagonals. By observing that the remaining two corners are not only the farthest points from the opposite diagonal corner, but they are also the farthest two points from the first two corners. This once again leads back to the problem that the fourth corner will probably be found to be a neighbor of the third corner.

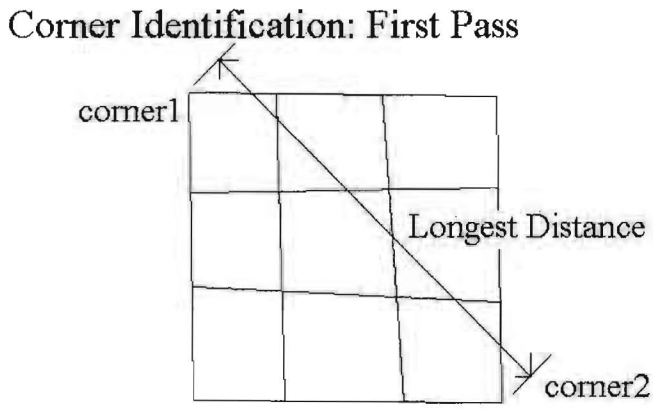

Figure 9

However, a property of the chain code is that all the points are in sequential order. So, the third corner will always be located between the first and second corners in the chain code. The fourth corner will have two candidates. One located before the first corner point in the chain code. The second will occur after the second corner point in the chain code. A simple comparison of the distances of theses two points finds the farther distance of the two from the other three known corners. See Figure 10.

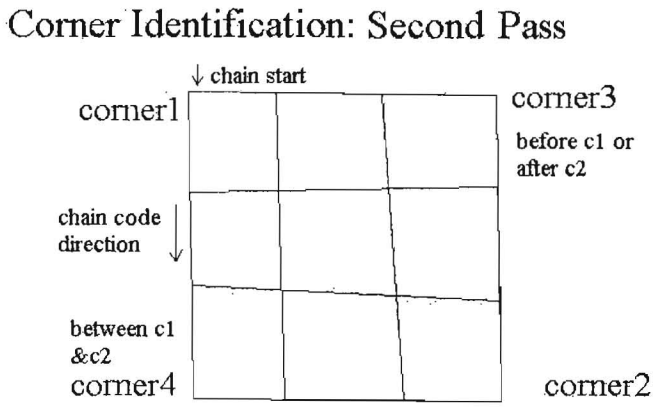

Figure 10

4. Detect Game

With the corners of all of the objects in the image found they can be used to determine which one of the objects is most likely to be the tic-tac-toe board. The fact that the target is square in shape makes this step relatively easy. From geometry a square has

the unique property that it has two sets of parallel sides all four of which are the same length and two diagonals of the same length.

The likeliness of an object being the tic-tac-toe board is found by using the four corners as the vertices of a weighted undirected complete graph on four vertices (See `detect_game()` in Appendix A, lines 973 – 1075). The weight for each edge is the distance between the end points in the image. Each edge is then compared against every other edge that should have the same length as itself. The following formula generates a percent error value.[2]

$$\frac{|\,\text{length1} - \text{length2}\,|}{|\,(\text{length1} + \text{length2}) / 2\,|} \qquad \text{(Eqn. 3)}$$

With the seven error values calculated from this formula, they are averaged together to determine the total percent error for one object. The object with the smallest percent error is considered to be the tic-tac-toe board at this point. However, this does not mean that the always complete using the object with the least percent error. If the percent error of the best possible object is greater than 10% the frame of data is thrown out and the program restarts to the beginning of the process. Throwing out frames with a high percent error at this stage saves time by avoiding further computation of data that would otherwise yield poor results.

5. Moravec Filter

With the object that is most likely to be the tic-tac-toe board identified, there is one more preprocessing step needed before the actual processing steps occur. This involves using a Moravec image filter on the image (Moravec, 1977). This filter

[2] In the event that length1 and length2 are both of zero length then the implementation returns FLT_MAX is defined in float.h. This check is necessary to avoid a division by zero error.

highlights edges in an image with a brighter shade of white the more likely it is to be an edge (See `moravec()` in Appendix A, lines 721 – 769). Other similar methods exist including the Laplace operator and the Mexican Hat (Sanka 1999). These other filters do a better job at highlighting edges. However, the Moravec filter offers the simplest implementation. It only uses the surrounding eight neighbor pixels compared to the Mexican Hat that uses 289 pixels for each pixel. Therefore the computation is comparatively light. It will be shown later on that the Moravec filter uses approximately half of the computation time required for a single frame (See Table 1). Since, the objective to play a game of tic-tac-toe in real time against a human player this relatively computationally inexpensive method is the preferred choice.

The output of the following formula is in grayscale. The original formula is:

$$MO(i, j) = .125 \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} | f(k, l) - f(i, j) | \qquad \text{(Eqn. 4)}$$

What this algorithm does is to take a pixel's eight neighbors and determine their average difference to the center pixel and set that value to the pixel in the center. This is performed for every pixel[3] in the image.

However the implementation performs some additional modification. The first modification is to use .5 or (1 / 2) instead of .125 or (1 / 8) for the average. This produces a much brighter pixels for edges in the output image. The second modification is to place a threshold on the filtered image. Thus all filtered values below the specified threshold map to zero and all above this threshold map to 255, where 255 is the maximum brightness of an eight bit pixel representation. The actual threshold is not specified. This allows for the value to be changed to accommodate changes in lighting conditions.

[3] Pixels that are along an edge are skipped to avoid a segmentation fault from stepping out of the bounds of the array. These pixels have their color set to black (0, 0, 0).

Processing:

With the last of the preprocessing steps completed the first of two processing steps occurs. These last two steps overall find detailed information about the location of specific points in the image and the pieces.

6. Detect tic-tac-toe
7. Detect Pieces

6. Detect tic-tac-toe

The first step is to properly order the four corners of the board into the array of the tic-tac-toe board. This is necessary because the order of the corners need to be clamped down with respect to real world coordinates required by the game engine and the arms. The implementation does this by finding the correct permutation group of the corners. In this way the array index position [0][0] will always have the upper right hand corner of the board.

The next step is to find the cross lines of the tic-tac-toe board. Theoretically, the end points of all four cross lines occur at one-third distances between the corners. However, to obtain better accuracy the code uses this value as a starting point to locate where the line actually is. However, the starting point is not exactly at this point. Instead it is moved in toward the center of the board. Currently it is moved $(1 / 24^{th})$ of the distance toward the other side of the board (See `findSidePoints()` in Appendix A, lines 1256 – 1276). The value was chosen because it is smaller than one-third but larger than zero. A value closer to zero works better to avoid interference from a marble (See Figure 11). Taking the halfway point between $1/6^{th}$ and zero is $1/12^{th}$. Since, a spot

closer to zero is desired halving this again is $1/24^{th}$. Thus a "clean" spot of the board is used to locate the dividers.

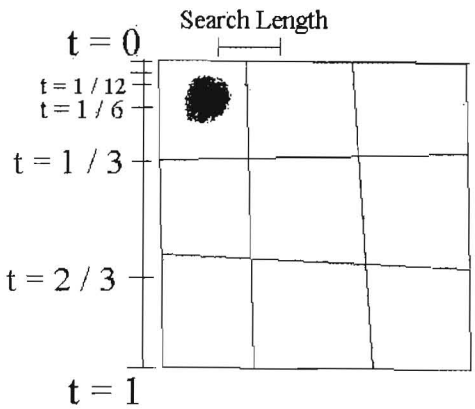

Figure 11

Next, that Moravec filtered image is scanned parallel to the edge of the board for the positive hits of the edges (See `search_moravec()` in Appendix A, lines 1077 - 1154). The scan moves in both directions. The first direction to find the edges highlighted by the Moravec filter is the one that is used. It continues for the theoretical distance of half the size of one of the sections (approximately 2.5 cm) of the tic-tac-toe board looking for the spot that is the most likely location for the end point of the cross line. With the endpoints of the cross lines determined, they next undergo a map that extends them back to the edge of the board (See `find_intersection()` in Appendix A, lines 1157 – 1254). This is to remove the effect of moving the starting point ($1 / 24^{th}$) of the board size inward.

With all of the edge points determined the middle points are found next. By looping along the sides in both the X and Y directions the intersections of all the lines can easily be calculated. The following formula was found by setting

$$Yt = M1*(Xt - Xa) + Ya \qquad \text{(Eqn. 5)}$$

equal to

$$Yt = M2 * (Xt - Xc) + Yc \qquad \text{(Eqn. 6)}$$

to get

$$Xt = \frac{(Xa*M1 - M2*Xc + Yc - Ya)}{M1 - M2} \qquad \text{(Eqn. 7)}$$

where M1 = line 1 slope, M2 = line 2 slope, line 1 point 1 = a, line 1 point 2 = b, line 2 point 1 = c and line 1 point 2 = d.[4] The large X and Y before the letters a, b, c and d refer the X and Y coordinate of that point.

The Y target coordinate is easily found knowing the Xt, or X target point, and by substituting it into the equation

$$Yt = M1 * (Xt - Xa) + Ya \qquad \text{(Eqn. 8)}$$

which represents the line between points a and b in point slope form.

7. Detect Pieces

With all of the important points of the tic-tac-toe game board finally known, work can finally begin on finding the pieces and determining what team they belong to. This starts out by finding points (1 / 3rd) the distance between the corner points for each section of the tic-tac-toe board (See Appendix A, lines 1583 – 1589). By performing this adjustment of the starting points, the search area is skewed far enough away from the Moravec edges of the cross lines to avoid counting them while searching for a potential marble piece.

Next, the marble search area starting points are calculated to half way between the already skewed points from above (See Appendix A, lines 1600 – 1627). Inside this area

[4] In the event that one of the slopes is undefined indicating a vertical line, then it is handled by skipping the Xt calculation, since this value is already known.

the pixels are tallied that have positive edge hits in the Moravec filtered image. By starting the actual search at this halfway point a lot of area is skipped. It is unlikely that a target marble will ever occupy these corner sections of the image and by skipping them speeds up calculation. If the tally for edges of the marble is great enough then it is determined that this square has a marble in it.

Determining the theoretical maximum of edge pixels that need to be counted in order for a marble to be seen is an easy task. Normally lines are considered to have no width only length. However, pixels do have width. So, finding the real world circumference and assuming it will have a width of one like it will in the image means finding its "area" is possible. In the real world the circumference is 2cm * Pi or approximately 6.2 cm. The size of one section of the board is 5cm x 5cm for 25 cm^2. Finding the percentage of the perimeter over the area, (6.2 / 25), yields 25% for the maximum percentage theoretical percentage capable of determining if a marble exists at a given location.

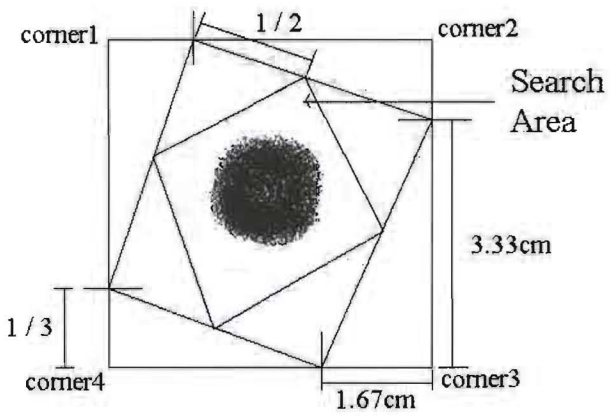

Figure 12

However, there is still a problem with this. There is area lost by moving the search area starting points over 1 / 3rd the distance from the corners. Consequently, 1.67 cm or (5cm * (1 / 3)) is lost from each side. The total area lost for one corner is:

$$(1.67\text{cm} * (5\text{cm} - 1.67\text{cm})) / 2 = 5.56\text{cm}^2 \qquad \text{(Eqn. 9)}$$

The area of the four corners lost is (5.56cm^2) * 4 for 13.88 cm^2. Thus the percentage is (6.2 / 13.88) for a maximum percentage of 45%. Now that the maximum percentage of pixels is known a reasonable value based on this needs to be determined. With a maximum of 45 and a minimum of zero one obvious solution is to average them together. This averaged value is coded into the `EDGE_COUNT` constant (See Appendix A, line 99).

At the same time the tally of these edge hits is going on, the center of these edge hits is also being averaged (See Appendix A, lines 1619 – 1620). If no marble was found then the few points that make up this average are garbage and ignored. However, if a piece was determined to exist in a section of the board, then it is at this location the color of the marble is determined.

The final step in determining the state of the tic-tac-toe board is to determine which team the pieces belong to. Centered on the location determined in the previous step an area of the image is swept determining a color average of the marble in that section of the board. The size of the search area is:

$$((\text{SEARCH\_SIZE} * 2) + 1)^2 \qquad \text{(Eqn. 10)}$$

Now that the color of all of the pieces on the board is known, this information needs to be mapped to binary values indicating each team (See `detect_pieces()` in Appendix A, lines 1694 – 1782). This is accomplished by checking the averaged color value for each pixel and finding the maximum and minimum. The average of these two

values is calculated. Marbles with color averages above this average are on team one; those below are on team two.[5]

Conclusions:

Computer vision programming is a non-trivial exercise. Very subtle things can cause very different output. One example from the early phases of the implementation was that it was not correctly identifying the board every frame. Upon examination of the chain codes that the program said was the board over the correct object the problem showed its ugly face. In those cases a blob was being picked up that contained exactly four pixels. These pixels were aligned in a square thus producing a percent error of zero. Since zero is less than any positive percent no matter how small the error was the tic-tac-toe board was not being found. Hence, the use of the constant `DECENT_SIZED_OBJECT` described earlier.

Other difficulties are that it takes a while for a single frame of data to be processed to find out if it worked or not. By inserting system time calls just before and after each important section of code exact time values can be found. Under normal executing parameters the following averages were obtained.

[5] When only one move has been played it is impossible to know what is the color of the second team. Because of this situation the first player to move is assigned to team one. After the other player moves if it turns out that the team values are switched from the color calculation algorithm they are reassigned to the correct team.

Code	Time (sec)	% of Total
Perspective Correction	0.28	2.8
Chain Code(s)	2.8	28
Corner Detection	2.8	28
Game Detection	0.00049	0.0049
Moravec Filter	5.4	54
tic-tac-toe Detection	0.0031	0.031
Piece Detection	0.012	0.12
Total	9.9	100

Figure 13

From this data it takes approximately ten seconds to processes just one frame. However, this only considers time measured once everything is initialized. Analyzing initialization times has found that there is an average of 8.5 seconds before any frame processing begins. So, far this means that almost 20 seconds has passed and nothing productive has happened yet.

That is not all the time for the first frame though. Network lag time for sending each image over the network is not factored in yet. The average time recorded by the shell for one frame without a move to be detected is 16.9 seconds. That means that on average 41% of the time per frame[6] is spent waiting for network transmissions. At these speeds it takes the first frame on average 26.8 seconds to be processed. The conclusion from the execution speed is that computer vision is not going to be making an impact in our daily lives until computers are capable doing these calculations at nearly instantaneous speeds.

[6] Frame size is 320 by 240 with 3 bytes per pixel. The total number of bytes is 320 x 240 x 3 = 230400.

Future Work:

In the current implementation all of the necessary programs for Shelley to play tic-tac-toe are run locally.[7] Having the functionality for all the programs to communicate via network communication on different machines is not being utilized. In the current SIE network protocol there is no conversion to and from network byte order. Thus Shelley is limited to only communicating to other Sparc Workstations.[8] By implementing this into the protocol other computers, especially those faster than Shelley, could be used to process the computationally expensive programs, i.e. the vision code.

Another possibility for speeding up the process is to compress the data sent over the network. This applies to the images since they are sent over the network not once but fourfold.[9] By compressing the image the amount of data sent over the network is reduced and consequently speeding up the implementation.

All of the code used in the preprocessing steps is highly generalized. It can be used to locate any object in the viewing region. The constant `DIVIDERS` can also be adjusted to identify other game boards. `DIVIDERS` is the value of parallel lines that define the look of the game. For tic-tac-toe the required value is four. Two of the four indicate the cross lines of a traditional tic-tac-toe representation. The remaining value of two indicates the sides of the game board. By setting `DIVIDERS` to a value of nine a chess (or checkers) game board would be looked for. The game of Go would require a

[7] The arm control is an exception.
[8] The arms control communication with the Intel Pentium is currently not implemented as a desired solution for the entire SIE protocol.
[9] The first is from the Frame Grabber to the administrator, second is from the administrator to the shell, third is from the shell to the administrator and forth is from the administrator to the vision program.

value of twenty-one (21). This can be generalized to the formula of (n + 1) where n is the number of spaces in a row of the specific game being played.

All of this research used one camera. By using two cameras the resulting stereoscopic image can be used to extract 3D data about the real world. Such an approach would work for identifying chess pieces, where the most troublesome piece, the knight, would require such 3D world data.

All of the color matching and difference algorithms are based heavily on the luminance of pixels. It is possible to have two pixels with the same luminance, but have their strongest red, green or blue component not even be close to the each other. In these cases most people would easily tell the two apart, but Shelley would have great difficulty. In her current state Shelley could be compared to someone who is colorblind. Future projects could work on ways for correcting her condition, something that her human counterparts can only wish for.

Appendix A - ttt_vision.c

```
1:  /*ttt_vision.c
2:
3:    This is the main source file for the tic-tac-toe vision system.
4:
5:    Programmer: Michael Zalokar
6:    Date: Spring 2000
7:  */
8:  /*The calculation of the corners is done in the following way.  The first
9:    two corners are found first and named corner1 and corner2.  The second
10:   set of corners are found next, named corner3 and corner4.  When all four
11:   are found the two sets are diagnols in the object.  The corners can be in
12:   any orientation to each other, but they will always have the same two
13:   corners adjacent and the same one as the diagnol.
14:
15:   When the ttt board has been singled out from all of the objects in the
16:   list, the corners are assigned into the board array as follows.  The
17:   assignment to the array occurs inside of the detect_ttt() function.  The
18:   array subscripts are listed in row major form.
19:
20:   corner1
21:   _____ _____ _____ corner4
22:   |[0][0] |[0][1] |[0][2] |[0][3]
23:   |       |       |       |
24:   |       |       |       |
25:   |       |       |       |
26:   -------------------------
27:   |[1][0] |[1][1] |[1][2] |[1][3]
28:   |       |       |       |
29:   |       |       |       |
30:   |       |       |       |
31:   -------------------------
32:   |[2][0] |[2][1] |[2][2] |[2][3]
33:   |       |       |       |
34:   |       |       |       |
35:   |corner3|       |       |
36:   ------------------------- corner2
37:    [3][0]   [3][1]   [3][2]   [3][3]
38:
39:   The pieces array is alligned in the same way.
40:
41:   Since, the corners are calculated from the chaincodes, there will always
42:   the unique property that the 4th corner will be the farthest point from
43:   the 1st and 2nd corners between these two points in the chaincode.  The
44:   third corner is either before the first corner in the chaincode or
45:   after the second corner in the chaincode.
46:
47:   The reason that the corners are determined using distance vs. using slope
48:   is becuase, distance doesn't have a situation where the result is
49:   undefined.
50:
51:   I used the _OpenGL Programming for the X Window System_ "green book", _OpenGL
52:   Programming Guide_ "red book" and the _OpenGL Reference Manual_ "blue book"
53:   to write the OpenGL/glut windowing code.
54:
55:   For the multithreading and socket programming I used the _Pthreads
56:   Programming_ "catapiller book" and _UNIX systems programming for SVR4_
```

```
57:    "lion book" from O'Reilly.
58:  */
59:
60:
61:  #include < stdio.h >
62:  #include < string.h >
63:  #include < math.h >
64:  #include < float.h >
65:  #include < GL/glut.h >
66:  #include < pthread.h >
67:  #include < sched.h >
68:  #include < errno.h >
69:  #include < stdlib.h >
70:  #ifdef __linux
71:  #include < getopt.h >
72:  #endif
73:
74:  #include "ttt_vision.h"
75:  #include "sie_protocol.h"
76:  #include "device_type_id_number.h"
77:  #include "ttt_vision_protocol.h"
78:  #include "shelley_sockets.h"
79:  #include "sie_utils.h"
80:  #include "colors.h"
81:
82:  #define Round(v) ((int)(v+0.5))
83:
84:  /*host:port:file:buffer*/
85:  #define GETOPTARGS "hf:p:b"
86:
87:  /*minimal size to consider an "object" as length of chaincode*/
88:  #define DECENT_SIZED_OBJECT 200
89:
90:  /*Used by the game detection algorithms.  Represent the distance between two
91:    known points as a fraction between 0 and 1.*/
92:  #define ONE_THIRD  (1.0 / 3.0)
93:  #define ONE_HALF   (1.0 / 2.0)
94:  #define ONE_SIXTH  (1.0 / 6.0)
95:  #define ONE_TWELTH (1.0 / 12.0)
96:  #define ONE_24TH   (1.0 / 24.0)
97:  #define ONE_32ND   (1.0 / 32.0)
98:
99:  /*the value used to threash the values of the moravec filter to 0 and 255*/
100:  /*is the sum of the differences between the center point and its neighbors.*/
101:  /*lighted*/
102:  /*#define THREASH_CONSTANT 128*/
103:  /*unlighted*/
104:  #define THREASH_CONSTANT 30
105:  /*the pratical ratio of the area searched for edge pixels of the marbles.*/
106:  #define EDGE_COUNT 0.25
107:  /*Since the Moravec is threashed to 0 and 255 as long as this value is between
108:    the two everything will work.*/
109:  #define EDGES_FOR_PIECE 128
110:
111:  /*the constants that control how many pixels are inserted agains one during
112:    the perspective correction*/
113:  #define X_DIR_PERS_CORR 1.6
114:  #define Y_DIR_PERS_CORR 1.9
```

```
115:
116:  /*closeness factor to consider to marbles on the same team*/
117:  #define DIFFERENCE .5
118:
119:  #define DIVIDERS 4
120:
121:  /*error on the board to be off of perfectly square*/
122:  #define ALLOWABLE_PERCENT_ERROR 7.5
123:
124:  /*structure defined as the object type.  All identified objects are stored in
125:    a struct of this type.  This is a very general data type and is not
126:    specific to a tic-tac-toe game board.*/
127:  typedef struct
128:  {
129:    coord corner1;
130:    coord corner2;
131:    coord corner3;
132:    coord corner4;
133:  }object;
134:
135:  /*This struct contains all the relavent information about the location,
136:    orientation, etc. of the tic-tac-toe game board itself.*/
137:  /*typedef struct
138:    {*/
139:     coord points[DIVIDERS][DIVIDERS];
140:
141:     /*The corners of the board.  corner1 and corner2 are always opposite
142:       (diagnol), just like corner3 and corner4 too.*/
143:    /* object corners;*/
144:
145:     /*The sides.  The numbers represent the to corners that the point is
146:      between.  The first number is the corner that it is closest to.*/
147:    /*   coord side14, side13, side23, side24;
148:     coord side31, side32, side41, side42;*/
149:
150:     /*The points surrounding the middle square.  The number indicates the
151:       closest corner.*/
152:    /*   coord middle1, middle2, middle3, middle4;*/
153:  /*}ttt;
154:   */
155:  /*Evil globals, but not do-able otherwise.*/
156:  static PGMImage *img_cur;       /*current*/
157:  static PGMImage *img_original;  /*original*/
158:  static PGMImage *img_pers_corr; /*perspective correction*/
159:  static PGMImage *img_grayscale;
160:  static PGMImage *img_moravec;   /*moravec*/
161:  /*static*/ int HSIZE;  /*width*/
162:  /*static*/ int VSIZE;  /*height*/
163:  /*static*/ int MVAL;   /*max val*/
164:
165:  /*pointer to a ttt structer containing the board info.*/
166:  /*static ttt *ttt_board;*/
167:
168:  /*pointer to an array of list_info structs  which point to the first, last and
169:    current nodes in each of the chain codes.*/
170:  list_info* chain_codes;
171:
172:  /*pointer to the array of objects found in the image.  Counting starts at
```

```
173:    zero and the array must be less than MAX_CHAINS in size.   Unused values
174:    should be set to zero.*/
175:  static object *all_objects;
176:
177:
178:
179:  /*used to determine if the lines connecting the corners should be
180:    drawn to the screen.  Set to < 0 if no abstract lines are to be drawn.
181:    Otherwise is set to the number of objects found (AKA number of chaincodes).*
182:  int draw_abstract_lines = -1;
183:
184:  /*used to draw the single object most likely to be considered the
185:    tic-tac-toe board.  Should be < 0 if this should not be drawn.
186:    Should be equal to the chain-code number (from 0 to MAX_CHAINS - 1)
187:    if it is to be drawn.*/
188:  int draw_abstract_board = -1;
189:
190:
191:  /*thread stuff*/
192:  pthread_t vision_thread;
193:  pthread_t conn_thread;
194:  pthread_mutex_t calculate_mutex;/* = PTHREAD_MUTEX_INITIALIZER;*/
195:  pthread_mutex_t images_mutex;/* = PTHREAD_MUTEX_INITIALIZER;*/
196:
197:  /*command line variables*/
198:  char *PGMfileName; /*string containing the filename*/
199:  char *hostname;
200:  int port_number;
201:  int is_buffered; /*  Is set to TRUE or FALSE.*/
202:
203:  int socket_num;
204:
205:  /*array of where the pieces are*/
206:  RGB_INT *data;
207:  unsigned char pieces[3][3];
208:
209:  bool redraw_needed = 1;
210:
211:  /*do be able to determine which team is which color when only one move has bee
212:    made, this variable stores that color until the second move is made to
213:    compare colors*/
214:  int team_first;
215:
216:  /*boolean toggle for displaying time data*/
217:  bool display_time = TRUE;
218:  /*boolean toggle for displaying verbose text*/
219:  bool display_verbose = FALSE;
220:
221:  /*************Time Display funcition*************************************/
222:  /************************************************************************/
223:  /*1st: initial time set by gettimeofday()
224:    2nd: final time set by gettimeofday();
225:    3rd: text message of the type of calculation timing*/
226:  void do_time(struct timeval t0, struct timeval t1, char *message)
227:  {
228:     double elapsed = t1.tv_sec - t0.tv_sec;
229:
230:     if(!display_time)
```

```
231:        return;
232:
233:     if(t1.tv_usec  > t0.tv_usec)
234:        elapsed += (t1.tv_usec - t0.tv_usec) / 1000000.0;
235:     else
236:        elapsed += (1000000.0 - abs(t1.tv_usec - t0.tv_usec)) / 1000000.0;
237:
238:     if(!message)
239:        printf("Processing took: %f seconds\n", elapsed);
240:     else
241:        printf("Processing %s took: %f seconds\n", message, elapsed);
242:  }
243:
244:  /**************Drawing funcitions*************************************/
245:  /*********************************************************************/
246:
247:  /*This function draws a single pixel to the screen.  (0, 0) is the lower left
248:   corner.*/
249:  /*1st: Integer of the x pixel to be drawn.
250:    2nd: Integer of the y pixel to be drawn.
251:    3rd: RGB values as stored as integral types with values between 0 - 255.*/
252:  void setCPixel(int ix, int iy, RGB_INT color)
253:  {
254:    float x = (ix*2.0)/HSIZE - 1.0;
255:    float y = (iy*2.0)/VSIZE - 1.0;
256:
257:    float red = (float)color.red/(float)MVAL;
258:    float green = (float)color.green/(float)MVAL;
259:    float blue = (float)color.blue/(float)MVAL;
260:
261:    glColor3f(red, green, blue);
262:
263:    glBegin(GL_POINTS);
264:       glVertex2f (x, y);
265:    glEnd();
266:  }
267:
268:  /*This function draws a line segment to the screen.  (0, 0) is the lower
269:    left corner.*/
270:  /*1st: x coord. of the 1st endpoint.
271:    2nd: y coord, of the 2nd endpoint.
272:    3rd: x coord. of the 2st endpoint.
273:    4th: y coord. of the 2st endpoint.
274:    5th: RGB values as stored as integral types with values between 0 - 255.*/
275:  void setCLines(int ix1, int iy1, int ix2, int iy2, RGB_INT color)
276:  {
277:    float x1 = (ix1*2.0)/HSIZE - 1.0;
278:    float y1 = (iy1*2.0)/VSIZE - 1.0;
279:    float x2 = (ix2*2.0)/HSIZE - 1.0;
280:    float y2 = (iy2*2.0)/VSIZE - 1.0;
281:
282:    float red = (float)color.red/(float)MVAL;
283:    float green = (float)color.green/(float)MVAL;
284:    float blue = (float)color.blue/(float)MVAL;
285:
286:    glColor3f(red, green, blue);
287:
288:    glBegin(GL_LINES);
```

```
289:      glVertex2f (x1, y1);
290:      glVertex2f (x2, y2);
291:    glEnd();
292:  }
293:
294:  /*This function draws a square to the screen.  (0, 0) is the lower left
295:    corner.*/
296:  /*1st: x coord. of the 1st endpoint.
297:    2nd: y coord, of the 2nd endpoint.
298:    3rd: x coord. of the 2st endpoint.
299:    4th: y coord. of the 2st endpoint.
300:    5th: RGB values as stored as integral types with values between 0 - 255.*/
301:  void setCRect(int ix1, int iy1, int ix2, int iy2, RGB_INT color)
302:  {
303:    float x1 = (ix1*2.0)/HSIZE - 1.0;
304:    float y1 = (iy1*2.0)/VSIZE - 1.0;
305:    float x2 = (ix2*2.0)/HSIZE - 1.0;
306:    float y2 = (iy2*2.0)/VSIZE - 1.0;
307:
308:    float red = (float)color.red/(float)MVAL;
309:    float green = (float)color.green/(float)MVAL;
310:    float blue = (float)color.blue/(float)MVAL;
311:
312:    glColor3f(red, green, blue);
313:
314:    glBegin(GL_POLYGON);
315:      glVertex2f (x1, y1);
316:      glVertex2f (x1, y2);
317:      glVertex2f (x2, y2);
318:      glVertex2f (x2, y1);
319:    glEnd();
320:  }
321:
322:  /*draws a string of text to the screen.*/
323:  /*1st: The x coord of the lower left corner where it is to be displayed.
324:    2nd: The y coord of the lower left corner where it is to be displayed.
325:    3rd: String that is to be displayed.
326:    4th: The color to display the text in.*/
327:  void drawString(int ix, int iy, char theString[256], RGB_INT color)
328:  {
329:     float x = (ix*2.0)/HSIZE - 1.0;
330:     float y = (iy*2.0)/VSIZE - 1.0;
331:     int i;
332:
333:     float red = (float)color.red/(float)MVAL;
334:     float green = (float)color.green/(float)MVAL;
335:     float blue = (float)color.blue/(float)MVAL;
336:
337:     glColor3f(red, green, blue);
338:
339:     glRasterPos2f(x, y);
340:     for (i = 0; theString[i] != '\0'; i++) /* draw the chars one at a time */
341:       glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, theString[i]);
342:  }
343:
344:  /*Draws a PGMImage to the window.*/
345:  /*1st: The pgm image to display.*/
346:  void showColor (PGMImage *img)
```

```
347:  {
348:     int i, j; /*loop counting: i = y, j = x*/
349:     /*This is the simplist way I found to use the fast OpenGL direct drawing
350:       functions without a major rewrite of the code.  First pack the data
351:       into this 3D array.  If that little bit of extra speed is needed, then
352:       the PGMImage functionality would need to be written as a 3D array and not
353:       a 2D array of structs.*/
354:     GLubyte checkImage[(*img).height][(*img).width][3];
355:     int test = 0;
356:     for(i = 0; i <  (*img).height; i++)
357:     {
358:        for(j = 0; j <  (*img).width; j++)
359:        {
360:           checkImage[i][j][0] = (GLubyte) (*img).data[i][j].red;
361:           checkImage[i][j][1] = (GLubyte) (*img).data[i][j].green;
362:           checkImage[i][j][2] = (GLubyte) (*img).data[i][j].blue;
363:     if(checkImage[i][j][0] == 0)
364:       test++;
365:        }
366:     }
367:
368:     /*tell OpenGL that the values are packed into byte sized values*/
369:     glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
370:     /*since OpenGL is setup for -1 to 1 size, specify (-1, -1) for the lower
371:       left corner which is (0, 0) to most of the program*/
372:     glRasterPos2f(-1, -1);
373:     /*draw the current image*/
374:     glDrawPixels((*img).width, (*img).height, GL_RGB,
375:            GL_UNSIGNED_BYTE, checkImage);
376:
377:     glFlush();
378:  }
379:
380:  /*called by showAbstract()*/
381:  /*draws the (x, y) values of the coordinates specified at that coordinate*/
382:  /*1st: The x coord.
383:    2nd: The y coord.
384:    3rd: Additional text to display, aside from the (x, y) determined from the
385:    1st and 2nd parameters.*/
386:  void display_labels(coord point, int row, int col, char *more_text)
387:  {
388:     char text[40]; /*for displaying the locations in text form*/
389:     int offset = 8; /*offset text # pixels from location*/
390:     char null_char[1];
391:
392:     if(more_text == NULL)
393:     {
394:        /*give more_text something to point to, to avoid seg. faults*/
395:        null_char[0] = '\0';
396:        more_text = null_char;
397:     }
398:
399:     if(row <  0 || col <  0)
400:        sprintf(text, "%s (%d, %d)", more_text, point.x, point.y);
401:     else
402:        sprintf(text, "%s [%d][%d](%d, %d)", more_text, row, col,
403:          point.x, point.y);
404:     drawString(point.x + offset, point.y + offset, text, blue);
```

```
405:  }
406:
407:  /*display the global abstract data*/
408:  void showAbstract(object* object_list, coord ttt_data[DIVIDERS][DIVIDERS])
409:  {
410:     int i, j;
411:
412:     /*when displayeing the chaincodes, this is the temp to do that.*/
413:     list_info temp;
414:
415:     glPointSize(2); /*make points more visible, if desired*/
416:     /*glLineWidth(4);*/
417:
418:     /*draw the chaincodes*/
419:     /*chain_codes is a global pointer to an array of list_info types*/
420:     for(i = 0; (i <  MAX_CHAINS) && chain_codes && chain_codes[i].cur; i++)
421:     {
422:        memcpy(&temp, &chain_codes[i], sizeof(list_info));
423:        while(RetrieveNextNode(&temp).cur)
424:        {
425:     setCPixel(RetrieveInfo(&temp).location.x,
426:               RetrieveInfo(&temp).location.y, yellow);
427:     Advance(&temp);
428:        }
429:     }
430:
431:     /*first check for non-null pointer, next check for dereferenced pointers
432:       in the array of head pointers and lastly make sure things stay in
433:       bound of the max incase all MAX_CHAINS number of chains are used.*/
434:     /*draw_abstract_lines is the global that holds the number of "objects"
435:       to draw abstract information for*/
436:     for(i = 0; i <  draw_abstract_lines && i <  MAX_CHAINS; i++)
437:     {
438:        setCLines(object_list[i].corner1.x, object_list[i].corner1.y,
439:            object_list[i].corner3.x, object_list[i].corner3.y,yellow);
440:        setCLines(object_list[i].corner4.x, object_list[i].corner4.y,
441:            object_list[i].corner2.x, object_list[i].corner2.y,yellow);
442:        setCLines(object_list[i].corner3.x, object_list[i].corner3.y,
443:            object_list[i].corner2.x, object_list[i].corner2.y,yellow);
444:        setCLines(object_list[i].corner4.x, object_list[i].corner4.y,
445:            object_list[i].corner1.x, object_list[i].corner1.y,yellow);
446:
447:        setCPixel(object_list[i].corner1.x, object_list[i].corner1.y,dk_red);
448:        setCPixel(object_list[i].corner2.x, object_list[i].corner2.y,dk_blue);
449:        setCPixel(object_list[i].corner3.x, object_list[i].corner3.y,white);
450:        setCPixel(object_list[i].corner4.x, object_list[i].corner4.y,dk_green);
451:
452:        /*labels for corner points*/
453:        display_labels(object_list[i].corner1, -1, -1, "c1");
454:        display_labels(object_list[i].corner2, -1, -1, "c2");
455:        display_labels(object_list[i].corner3, -1, -1, "c3");
456:        display_labels(object_list[i].corner4, -1, -1, "c4");
457:     }
458:
459:     /*if there is board to draw, draw it to the screen*/
460:     if((draw_abstract_board  > -1))
461:     {
462:        for(i = 1; i <  DIVIDERS - 1; i++)
```

```
463:        {
464:     setCLines(points[i][0].x, points[i][0].y, points[i][DIVIDERS - 1].x,
465:               points[i][DIVIDERS - 1].y, magenta);
466:
467:     setCLines(points[0][i].x, points[0][i].y, points[DIVIDERS - 1][i].x,
468:               points[DIVIDERS - 1][i].y, magenta);
469:
470:     setCPixel(points[i][0].x, points[i][0].y, red);
471:     setCPixel(points[DIVIDERS - 1][i].x, points[DIVIDERS - 1][i].y, red);
472:        }
473:
474:        /*lables for all ttt board points*/
475:        for(i = 0; i <  DIVIDERS; i++)
476:     for(j = 0; j <  DIVIDERS; j++)
477:     {
478:        display_labels(points[i][j], i, j, NULL);
479:     }
480:     }
481:     glFlush();
482:  }
483:
484:  /*********************Support functions*****************************/
485:  /*******************************************************************/
486:
487:  /*take the source pixel and copy it to the destination pixel.*/
488:  /*1st: dest is the pointer to the PGMImage that is to have the pixel changed.
489:    2nd: the row(y coord) that is to be the destination
490:    3rd: the column(x coord) that is to be the destination
491:    4th: dsrc is the pointer to the PGMImage that is to have its pixel copied.
492:    5th: the row(y coord) that is to be the source
493:    6th: the column(x coord) that is to be the source
494:  */
495:  void pxlcpy(PGMImage *dest, int dest_row, int dest_col,
496:        PGMImage *src, int src_row, int src_col)
497:  {
498:    /*make sure values are within bounds*/
499:    if(dest_col  > 0 && dest_col <  (*dest).width
500:       && dest_row  > 0 && dest_row <  (*dest).height
501:       && src_col  > 0 && src_col <  (*src).width
502:       && src_row  > 0 && src_row <  (*src).height)
503:    {
504:      (*dest).data[dest_row][dest_col].red =
505:        (*src).data[src_row][src_col].red;
506:
507:      (*dest).data[dest_row][dest_col].green =
508:        (*src).data[src_row][src_col].green;
509:
510:      (*dest).data[dest_row][dest_col].blue =
511:        (*src).data[src_row][src_col].blue;
512:    }
513:  }
514:
515:  /*calculate the average of the red, green and blue values of a sinfle pixel.*/
516:  /*1st: the RGB_INT to be averaged*/
517:  /*the average of the red, green and blue is returned*/
518:  int rgb_avg(RGB_INT cur_pxl)
519:  {
520:     /*convert each RGB to the average of the original*/
```

```
521:     return ((cur_pxl.red + cur_pxl.green + cur_pxl.blue) / 3);
522:  }
523:
524:  /*Returns average (with RGB avg) pixel value for the image passed in.*/
525:  /*1st: a pointer to a pgm image*/
526:  /*the return value is the average of all the pixel in the image.  Each pixel's
527:    value is the average of its RGB parts.*/
528:  int img_pxl_avg(PGMImage* img)
529:  {
530:     int i, j; /*loop counting*/
531:     int sum = 0;
532:     int temp;
533:
534:     for(i = 0; i <  (*img).height; i++)/*collumn*/
535:        for(j = 0; j <  (*img).width; j++)/*row*/
536:           sum += rgb_avg((*img).data[i][j]);
537:
538:
539:     temp = (sum / ((*img).height * (*img).width));
540:
541:     return temp;
542:  }
543:
544:  /*determines what in the image is determined to background and foreground.*/
545:  /*1st: value to compare whether more of the pixels are greater than this or
546:    less than this.
547:    2nd: the image to be checked.*/
548:  /*return  &gt0 if number of pixels is greater than img. pxl. avg., return < 0
549:    number of pixesl is less than img. pxl. avg. and return zero of equal*/
550:  int background(int treash_value, PGMImage* img)
551:  {
552:     int i, j; /*loop counting*/
553:     int pxl_less = 0, pxl_more = 0;
554:
555:     for(i = 0; i <  (*img).height; i++)/*collumn*/
556:        for(j = 0; j <  (*img).width; j++)/*row*/
557:        {
558:     if(rgb_avg((*img).data[i][j]) <  treash_value)
559:       pxl_less++;
560:
561:           if(rgb_avg((*img).data[i][j])  > treash_value)
562:       pxl_more++;
563:        }
564:
565:     if(pxl_less  > pxl_more)
566:        return -1;
567:     else if(pxl_less <  pxl_more)
568:        return 1;
569:     else
570:        return 0;
571:  }
572:
573:  /*Used by showColor() and detect_pieces() to intrapolate the location of a
574:    point along a line between two known points.
575:    r = (1 - t)p1 + t*p2
576:    This is calculated twice, once for x direction and once for y direction.*/
577:  /*1st: coordinate of the closer point that t will intropolate to.
578:    2nd: coordinate of the farther point that t will intrapolate to.
```

```
579:    3rd: the fractional value, with point1 considered t = 0 and point2
580:    considered t = 1 that the return value is intropolated.*/
581:  /*returns the coordinate that is intropolated*/
582:  coord find_dividers(coord point1, coord point2, float t)
583:  {
584:     coord temp;
585:
586:     temp.x = (int)(((1.0 - t) * point1.x) + (t * point2.x));
587:     temp.y = (int)(((1.0 - t) * point1.y) + (t * point2.y));
588:
589:     return temp;
590:  }
591:
592:  /*takes two coordinates as x and y pairs and returns the distance betweem them
593:     as a decimal*/
594:  float findDist(int x1, int y1, int x2, int y2)
595:  {
596:     return sqrt(((x1 - x2) * (x1 - x2)) + ((y1 - y2) * (y1 - y2)));
597:  }
598:
599:
600:  /******Perspective correction*****************************************
601:  ********************************************************************/
602:
603:  /*The image that we will be looking at will from the camera will likely
604:    have a perspective, so look at removing the perspective effect.  The x
605:    direction correction starts in the middle and moves toward the sides
606:    stretching the image with more  stretching occuring with higher rows.
607:    The y direction correction inserts more lines near the top than towards the
608:    bottom of the image.  There is a y direction recentering calculation
609:    performed.*/
610:  /*1st: The pointer to the pgm image that the image will be copied into with
611:    the perspective removed.
612:    2nd: The pointer to the pgm image that will have the perspective removed.*/
613:  void pers_corr(PGMImage* new_img, PGMImage* org_img)
614:  {
615:     /******************X direction variables*********/
616:
617:     /*i and j are the left half, k and l are the right half*/
618:     float i, k; /*loop counting x dir*/
619:     int j, l; /*loop counting x dir*/
620:     int old_i, old_k; /*x dir. counting to remove moire iterference*/
621:
622:     float ins_s = X_DIR_PERS_CORR; /*insert constant starting value x dir.*/
623:     float ins_k = ins_s; /*current insert x dir.*/
624:
625:     /*The halfway marks in the width.*/
626:     int mid_width_left;
627:     int mid_width_right;
628:
629:
630:     /******************Y direction variables*********/
631:
632:     /*the recenter in y direction value*/
633:     int recenter;
634:
635:     float m; /*loop counting y dir*/
636:     int n; /*loop counting y dir*/
```

```
637:     int old_n = (*new_img).height, old_n_inc; /*y dir remove moire iterference*
638:
639:     float ins_t = Y_DIR_PERS_CORR; /*insert constant starting value y dir.*/
640:     float ins_j = ins_t; /*current insert value y dir.*/
641:
642:     /*just to be thourough clear the memory and reset maxes*/
643:     memset(new_img, 0, sizeof(PGMImage));
644:     (*new_img).height = (*org_img).height;
645:     (*new_img).width = (*org_img).width;
646:     (*new_img).maxVal = (*org_img).maxVal;
647:
648:     /*setting these before the new image size was stupid.  never do that again.
649:       Doing so resulted in the first frame being garbaged.*/
650:     mid_width_left = ((*new_img).width / 2) - 1;
651:     mid_width_right = ((*new_img).width / 2);
652:
653:     /*since the x dir is calculated from the middle, the middle is always
654:       mapped to the middle.  Since the y dir is calculated from the top the
655:       image is pushed to the bottom.  To correct this add to the new pixel
656:       location this value to recenter the image.*/
657:     recenter =  ((*new_img).height / ins_t);
658:
659:     /*Loop through each row from top to bottom...*/
660:     for(m = n = ((*new_img).height - 1);
661:         (n  >= 0);
662:         m -= ins_j, n--)
663:     {
664:        /*...reset moire interference removal counter x dir...*/
665:        old_i = ((*new_img).width / 2) - 1;
666:        old_k = ((*new_img).width / 2);
667:
668:        /*...so each half is ajusted to remove perspective effect...*/
669:        /*initalize the x and y starting conditions*/
670:        for(i = j = mid_width_left, k = l = mid_width_right;
671:      /*x and y ending conditions*/
672:      ((j  >= 0) && (l <  (*new_img).width));
673:      /*incremental*/
674:      i -= ins_k, j--, k += ins_k, l++)
675:        {
676:     for(;old_i  >= (int)i; old_i--)  /*...in the left half...*/
677:        for(old_n_inc = old_n; old_n_inc  >= m; old_n_inc--)
678:           pxlcpy(new_img, old_n_inc + recenter, old_i, org_img, n, j);
679:
680:     for(;old_k < = (int)k; old_k++)  /*...in the right half.*/
681:        for(old_n_inc = old_n; old_n_inc  >= m; old_n_inc--)
682:           pxlcpy(new_img, old_n_inc + recenter, old_k, org_img, n, l);
683:        }
684:        /*Move the new image x_coord pixel counter to next new image pixel*/
685:        ins_k -= ((ins_s - 1.0) / (*new_img).height);
686:        ins_j -= ((ins_t - 1.0) / (*new_img).width);
687:
688:        old_n = m; /*store for next row (y direction)*/
689:     }
690:
691:
692:  }
693:
694:  /*****convert color to grayscale*************************************
```

```
695: ************************************************************************/
696:
697: /*Turn a color image into a black and white image.*/
698: /*1st: The pointer to the pgm image that the image will be copied into with
699:   the perspective removed.
700:   2nd: The pointer to the pgm image that will have the color removed.*/
701: void color_to_gray(PGMImage* new_img, PGMImage* org_img)
702: {
703:    int row, col; /*loop counting*/
704:    RGB_INT cur_pxl; /*current pixel*/
705:
706:    /*just to be thourough clear the memory and reset maxes*/
707:    memset(new_img, 0, sizeof(PGMImage));
708:    (*new_img).height = (*org_img).height;
709:    (*new_img).width = (*org_img).width;
710:    (*new_img).maxVal = (*org_img).maxVal;
711:
712:    /*Starting with the top row...*/
713:    for(row = (*new_img).height - 1; row  >= 0; row--)
714:       for(col = 0; col <  (*new_img).width - 1; col++)
715:       {
716:    cur_pxl = (*org_img).data[row][col]; /*more readable*/
717:
718:    /*convert each RGB to the average of the original*/
719:    (*new_img).data[row][col].red =  rgb_avg(cur_pxl);
720:    (*new_img).data[row][col].green =  rgb_avg(cur_pxl);
721:    (*new_img).data[row][col].blue =  rgb_avg(cur_pxl);
722:       }
723: }
724:
725: /*******Moravec Edge Highlighting*************************************
726: ************************************************************************/
727:
728: /*use the Moravec algorithm to highlight the edges in an image.  This
729:   algorithm takes the normally grayscale result and threasholds it to
730:   the values of 0 and 255 with the divide at THREASH_CONSTANT*/
731: /*1st: The pointer to the pgm image that the image will be copied into with
732:   the moravec filter applied.
733:   2nd: The pointer to the pgm image that will have the moravec filter*/
734: void moravec(PGMImage* new_img, PGMImage* org_img)
735: {
736:    int row, col; /*loop counting*/
737:    int i, j, k, l; /*Sanka, Hlavac & Boyle; p. 97 f. 4.73*/
738:    int running_sum;
739:    float K = 0.5; /*.125 according to org. formula, but .5 is brighter*/
740:
741:    /*just to be thourough clear the memory and reset maxes*/
742:    memset(new_img, 0, sizeof(PGMImage));
743:    (*new_img).height = (*org_img).height;
744:    (*new_img).width = (*org_img).width;
745:    (*new_img).maxVal = (*org_img).maxVal;
746:
747:    /*starting at the top row*/
748:    /*don't count pixels along image edge to avoid segmentation fault*/
749:    for(row = (*new_img).height - 1 - 1; row  > 0; row--)
750:       for(col = 1; col <  (*new_img).width - 1; col++) /*left col start*/
751:       {
752:    i = row;
```

```
753:     j = col;
754:     running_sum = 0;
755:
756:     /*Sanka, Hlavac & Boyle; p. 97 f. 4.73*/
757:     for(k = i - 1; k < = i + 1; k++) /*row*/
758:       for(l = j - 1; l < = j + 1; l++) /*column*/
759:         running_sum += abs(rgb_avg((*org_img).data[k][l]) -
760:                            rgb_avg((*org_img).data[i][j]));
761:
762:     /*assign the new pixel value*/
763:     /*since all the data is initalized to 0, we only worry when it
764:             shouldn't be*/
765:     if((int)(K * running_sum)  >= THREASH_CONSTANT)
766:     {
767:        (*new_img).data[row][col].red = 255;
768:        (*new_img).data[row][col].green = 255;
769:        (*new_img).data[row][col].blue = 255;
770:      }
771:       /*this is the original code...*/
772:       /*(*new_img).data[row][col].red = (int)(K * running_sum);
773:     (*new_img).data[row][col].green = (int)(K * running_sum);
774:     (*new_img).data[row][col].blue = (int)(K * running_sum);*/
775:       }
776: }
777:
778: /*******Corners*************************************************
779: ***************************************************************/
780:
781: /*find the first two corners of all of the obejects in the image.  This is
782:   accomplished by looking at the chaincodes for the images and determining
783:   the two points that are the farthest apart.*/
784: /*1st: a pointer to an array of object structs.
785:   2nd: the array of chaincodes that is searched for the farthest two points.*/
786: void findFirstTwoCorners(object *objects_array, list_info* chainCodes)
787: {
788:    int i; /*loop counting*/
789:    list_info temp, search; /*temp copies of the data*/
790:    double max_dist, test_dist; /*temp distance holders*/
791:
792: /*printf("\nFinding first 2 corners.\n");*/
793:
794:    /*as long as there are codes to check, keep checking.*/
795:    for(i = 0; ((i <  MAX_CHAINS) && chainCodes[i].cur); i++)
796:    {
797:       memcpy(&temp, &chainCodes[i], sizeof(list_info));
798:
799:       max_dist = 0.0;  /*reset this for each iteration*/
800:
801:       while(RetrieveNextNode(&temp).cur) /*while there are nodes to check*/
802:       {
803:    /*set the faster moving search pointer to temp,
804:      this increases the effiecency a lot compared to
805:      setting it equal to the first node..*/
806:          memcpy(&search, &temp, sizeof(list_info));
807:
808:    while(RetrieveNextNode(&search).cur)
809:          {
810: /*setCPixel(RetrieveInfo(&temp).location.x,
```

```
811:    RetrieveInfo(&temp).location.y, green);*/
812:
813:              /*determine if found a new maximum distance between two locations*
814:              if((test_dist = findDist(RetrieveInfo(&search).location.x,
815:                                       RetrieveInfo(&search).location.y,
816:                                       RetrieveInfo(&temp).location.x,
817:                                       RetrieveInfo(&temp).location.y)) &gtmax_dist)
818:              {
819:                 max_dist = test_dist;
820:                 objects_array[i].corner1.x = RetrieveInfo(&temp).location.x;
821:                 objects_array[i].corner1.y = RetrieveInfo(&temp).location.y;
822:                 objects_array[i].corner2.x = RetrieveInfo(&search).location.x;
823:                 objects_array[i].corner2.y = RetrieveInfo(&search).location.y;
824:              }
825:              Advance(&search);
826:           }
827:           Advance(&temp);
828:        }
829:     }
830:  }
831:
832:  /*determine the second to corners for the objects.*/
833:  /*1st: the array of objects that contains the first to corners already.
834:    2nd: the array of chaincode data*/
835:  void findSecondTwoCorners(object *objects_array, list_info* chain_code_array)
836:  {
837:     int i; /*loop counting*/
838:     list_info temp;
839:     float temp_dist1, temp_dist2; /*distance between point and each corner*/
840:     coord canidate_coord1, temp_coord;
841:     float canidate_dist1 = 0.0, max_dist;
842:     int corner_count;
843:
844:  /*printf("\nFinding last 2 corners.\n\n");*/
845:
846:     /*for each chain code find the corners*/
847:     for(i = 0; (i <  MAX_CHAINS) && chain_code_array[i].cur; i++)
848:     {
849:        memcpy(&temp, &chain_code_array[i], sizeof(list_info));
850:
851:        /*reset these for the next chain code*/
852:        max_dist = 0.0;
853:        corner_count = 1;
854:
855:        /*while there are nodes in the chain code to check*/
856:        /*if there isn't a next node cur is NULL, which is checked*/
857:        while(RetrieveNextNode(&temp).cur)
858:        {
859:  /*setCPixel(RetrieveInfo(&temp).location.x,
860:    RetrieveInfo(&temp).location.y, color1);*/
861:
862:    /*determine the first canidate coord for corner 3/4*/
863:    if(((RetrieveInfo(&temp).location.x == objects_array[i].corner1.x)
864:        && (RetrieveInfo(&temp).location.y == objects_array[i].corner1.y))
865:       || ((RetrieveInfo(&temp).location.x == objects_array[i].corner2.x)
866:         &&(RetrieveInfo(&temp).location.y == objects_array[i].corner2.y)))
867:    {
868:      /*if this corner found is the first of the two allready known
```

```
869:        corners, then set the first canidate coord data and reset data
870:        to find the next canidate corner point*/
871:       if(corner_count == 1)
872:       {
873:          canidate_coord1.x = temp_coord.x;
874:          canidate_coord1.y = temp_coord.y;
875:          canidate_dist1 = max_dist;
876:
877:          corner_count = 2; /*set for next corner*/
878:          max_dist = 0.0;
879:       }
880:       else if(corner_count == 2)
881:       {
882:          /*the second canidate is always a corner*/
883:          all_objects[i].corner4.x = temp_coord.x;
884:          all_objects[i].corner4.y = temp_coord.y;
885:
886:          max_dist = 0.0; /*set for next corner canidate*/
887:       }
888:    }
889:
890:    /*calculate the distance between the current point being checked and
891:      each corner point*/
892:    temp_dist1 = findDist(all_objects[i].corner1.x,
893:                          all_objects[i].corner1.y,
894:                          RetrieveInfo(&temp).location.x,
895:                          RetrieveInfo(&temp).location.y);
896:    temp_dist2 = findDist(all_objects[i].corner2.x,
897:                          all_objects[i].corner2.y,
898:                          RetrieveInfo(&temp).location.x,
899:                          RetrieveInfo(&temp).location.y);
900:
901:    /*if the current point is the furthest away sofar, store this point
902:      untill it is overridden or becomes a canidate point*/
903:    if((temp_dist1 + temp_dist2)  > max_dist)
904:    {
905:       temp_coord.x = RetrieveInfo(&temp).location.x;
906:       temp_coord.y = RetrieveInfo(&temp).location.y;
907:
908:       max_dist = (temp_dist1 + temp_dist2);
909:    }
910:
911:    Advance(&temp);
912:        }
913:
914:        /*from the three canidate coords find the two real corners.*/
915:        /*the second canidate will always be a corner, must test 1 vs 3, where
916:    three is in the variables temp_coord and max_dist.*/
917:        if(canidate_dist1  > max_dist) /*first canidate*/
918:        {
919:           all_objects[i].corner3.x = canidate_coord1.x;
920:     all_objects[i].corner3.y = canidate_coord1.y;
921:        }
922:        else /*third canidate*/
923:        {
924:           all_objects[i].corner3.x = temp_coord.x;
925:     all_objects[i].corner3.y = temp_coord.y;
926:        }
```

```
927:     }
928:  }
929:
930:  /*returns the number of objects found*/
931:  /*places the corner info in the global corners data*/
932:  int detect_corners(list_info* current_chaincodes, object *objects_array)
933:  {
934:     int i; /*temporarily holds number of chaincodes*/
935:
936:     if(display_verbose)
937:        printf("Looking for corners.\n");
938:
939:     /*clear the array of objects*/
940:     memset(objects_array, 0, sizeof(object) * MAX_CHAINS);
941:
942:     /*find the first two corners.  they will be across a diagnal.*/
943:     findFirstTwoCorners(objects_array, current_chaincodes);
944:     /*find the second two corners.  they will be across a diagnal too.*/
945:     findSecondTwoCorners(objects_array, current_chaincodes);
946:
947:     /*when this stops, i holds the number of chaincodes*/
948:     for(i = 0; (i <  MAX_CHAINS) && chain_codes && chain_codes[i].cur; i++);
949:
950:     return i;
951:  }
952:
953:  /*******Find the game board*****************************************
954:  ******************************************************************/
955:
956:  /*compare two distances between points.*/
957:  /*1st: the distance between two points
958:    2nd: the distance between two points*/
959:  /*returns the difference of the lengths divided by the lengths average.
960:    If there is a division by zero, the two lengths are both zero.  The return
961:  value in this case is FLT_MAX defined in float.h*/
962:  float compare_length(float length1, float length2)
963:  {
964:     float denom_check = fabs((length1 + length2) / 2);
965:
966:     if(denom_check == 0)
967:     {
968:        /*the only way to possibly pull this off is if one point is
969:    is considered more than one corner.  This is most likely
970:    to happen where the chain was only two or three nodes long.*/
971:        /*Just set the error to the maximum value obtained from float.h,
972:    since such a small chain is not what we are looking for.*/
973:
974:        return FLT_MAX;
975:     }
976:     else
977:        return fabs(length1 - length2) / denom_check;
978:  }
979:
980:  /*determine which object is the tic-tac-toe board from those found in the
981:    chaincodes and stored as objects.*/
982:  /*1st: pointer the chaincode array.
983:    2nd: pointer to the array of objects.*/
984:  /*return the object that is most likely the chaincode.  This value is
```

```
985:    the array subscript value for the object array.  If none is found then
986:    returns -1.*/
987:  int detect_game(list_info *current_chaincodes, object *object_array)
988:  {
989:     float length2to4[MAX_CHAINS], length1to3[MAX_CHAINS]; /*side pairs*/
990:     float length1to4[MAX_CHAINS], length2to3[MAX_CHAINS]; /*side pairs*/
991:     float length1to2[MAX_CHAINS], length3to4[MAX_CHAINS]; /*diagnaols*/
992:
993:     float error24to13[MAX_CHAINS], error14to23[MAX_CHAINS];/*opp. sides*/
994:     float error14to13[MAX_CHAINS], error23to24[MAX_CHAINS];/*share corner*/
995:     float error31to32[MAX_CHAINS], error41to42[MAX_CHAINS];/*share corner*/
996:     float error12to34[MAX_CHAINS];/*diagnaols*/
997:
998:     float error_avg;/*average of the errors stored in error##to## variables*/
999:
1000:     int i; /*loop counting*/
1001:
1002:     /*the most likely object (0 to num_of_corners) that is to be considered
1003:       as the board.  The float is the error associated with this object.*/
1004:     int most_likely = -1;
1005:     float ml_error = FLT_MAX; /*just to make sure*/
1006:
1007:     if(display_verbose)
1008:        printf("Finding game board.  ");
1009:
1010:     /*for each chaincode*/
1011:     for(i = 0; (i <  MAX_CHAINS) && current_chaincodes &&
1012:    current_chaincodes[i].cur; i++)
1013:     {
1014:        /*count the number of nodes in the chaincode.  Unless the size is
1015:         considered long enough, skip it and move on.*/
1016:        if(Length(&current_chaincodes[i]) <  DECENT_SIZED_OBJECT)
1017:    continue;
1018:
1019:        /*since points 1 & 2 are at a diagnal, and 3 & 4 are at a diagnol,
1020:           then the dist between  2 and 4 & 1 and 3 should be close
1021:           in value.*/
1022:        length2to4[i] = findDist(object_array[i].corner2.x,
1023:                         object_array[i].corner2.y,
1024:                         object_array[i].corner4.x,
1025:                         object_array[i].corner4.y);
1026:        length1to3[i] = findDist(object_array[i].corner1.x,
1027:                         object_array[i].corner1.y,
1028:                         object_array[i].corner3.x,
1029:                         object_array[i].corner3.y);
1030:
1031:        /*the other side pair*/
1032:        length1to4[i] = findDist(object_array[i].corner1.x,
1033:                         object_array[i].corner1.y,
1034:                         object_array[i].corner4.x,
1035:                         object_array[i].corner4.y);
1036:        length2to3[i] = findDist(object_array[i].corner2.x,
1037:                         object_array[i].corner2.y,
1038:                         object_array[i].corner3.x,
1039:                         object_array[i].corner3.y);
1040:
1041:        /*diagnols... always will be 1 & 2 and 3 & 4*/
1042:        length1to2[i] = findDist(object_array[i].corner1.x,
```

```
1043:                          object_array[i].corner1.y,
1044:                          object_array[i].corner2.x,
1045:                          object_array[i].corner2.y);
1046:        length3to4[i] = findDist(object_array[i].corner3.x,
1047:                          object_array[i].corner3.y,
1048:                          object_array[i].corner4.x,
1049:                          object_array[i].corner4.y);
1050:
1051:        /*calculate percent errors for all edge (and diagnal) combinations*/
1052:        error24to13[i] = compare_length(length2to4[i], length1to3[i]);/*op.side
1053:        error14to23[i] = compare_length(length1to4[i], length2to3[i]);
1054:        error14to13[i] = compare_length(length1to4[i], length1to3[i]);/*1 crn.*
1055:        error23to24[i] = compare_length(length2to3[i], length2to4[i]);
1056:        error31to32[i] = compare_length(length1to3[i], length2to3[i]);
1057:        error41to42[i] = compare_length(length1to4[i], length2to4[i]);
1058:        error12to34[i] = compare_length(length1to2[i], length3to4[i]);/*diag.*/
1059:
1060:        /*average all of the error values together*/
1061:        error_avg = ((error24to13[i] + error14to23[i] + error14to13[i] +
1062:               error23to24[i] + error31to32[i] + error41to42[i] +
1063:               error12to34[i]) / 7);
1064:
1065:        /*determine if the current object is considered the most likely to
1066:           be the ttt board so far.  Average of all the error##to##'s.
1067:           If the current is */
1068:        if(ml_error  > error_avg)
1069:        {
1070:           most_likely = i;
1071:           ml_error = error_avg;
1072:        }
1073:     }
1074:     if(display_verbose)
1075:        printf("Object %d is most likely the board with %f\%% error.\n",
1076:        most_likely, ml_error * 100);
1077:
1078:    if(ml_error* 100 <  ALLOWABLE_PERCENT_ERROR)
1079:      return most_likely; /*return the object number that is the board*/
1080:    else
1081:      return -1;
1082:  }
1083:
1084:  /*searches the moravec image for a cross line in the tic-tac-toe board
1085:   int the neighborhood of t = 1/3 */
1086:  /*1st: the first 'corner' to search from.  (t = 0)
1087:    2nd: the second 'corner' to search from. (t = 1)
1088:    3rd: a pointer to the moravec image used in finding the lines.*/
1089:  /*returns the coord of the location determined to be the end of the cross
1090:    line being searched for.  If one isn't found by looking at the image,
1091:    then the return value is the theoretical location where t = 1/3 */
1092:  /*note: the corners passed in are moved from the actuall corners.  see
1093:    find_side_points()*/
1094:  coord search_moravec(coord point1, coord point2, PGMImage *moravec,
1095:                float t_divider)
1096:  {
1097:     int i, j; /*loop counting*/
1098:     float t;
1099:     coord temp1, temp2;
1100:     float dist = findDist(point1.x, point1.y, point2.x, point2.y);
```

```
1101:     int search_size = 2; /*'radius' that defines the neighborhood of a point*/
1102:
1103:     coord most_likely_corner = {-1, -1};
1104:     int most_likely_sum1 = 0, most_likely_sum2 = 0;
1105:     int sum_moravec_points1, sum_moravec_points2;
1106:
1107:     /*calculate the coordinates where the divider edges should be.*/
1108:     for(t = 0; t <  ONE_TWELTH; t += (1 / dist))
1109:     {
1110:        /*check in both the + and - directions simultaniously.*/
1111:        temp1 = find_dividers(point1, point2, t_divider + t);
1112:        temp2 = find_dividers(point1, point2, t_divider - t);
1113:
1114:        /*clear this before next iteration*/
1115:        sum_moravec_points1 = sum_moravec_points2 = 0;
1116:
1117:        /*search the neighborhood for edge 'hits' in the moravec image*/
1118:        for(i = -(search_size); i < = search_size; i++)
1119:        {
1120:    for(j = -(search_size); j < = search_size; j++)
1121:    {
1122:       if(rgb_avg((*moravec).data[temp1.y + i][temp1.x + j])  > 128)
1123:       {
1124:  /*setCPixel(temp1.x + j, temp1.y + i, red);*/
1125:          sum_moravec_points1++;
1126:       }
1127:       if(rgb_avg((*moravec).data[temp2.y + i][temp2.x + j])  > 128)
1128:       {
1129:  /*setCPixel(temp2.x + j, temp2.y + i, red);*/
1130:          sum_moravec_points2++;
1131:       }
1132:
1133:    }
1134:        }
1135:
1136:        /*if the current point in the search is the best, return it*/
1137:        /*reusing search_size seems good, since it is about the right size*/
1138:        if((search_size < = sum_moravec_points1) &&
1139:    (most_likely_sum1 < = sum_moravec_points1) &&
1140:    !most_likely_sum2)
1141:        {
1142:    most_likely_sum1 = sum_moravec_points1;
1143:    most_likely_corner = temp1;
1144:        }
1145:        else if(((search_size < = sum_moravec_points2) &&
1146:          (most_likely_sum2 < = sum_moravec_points2) &&
1147:          !most_likely_sum1))
1148:        {
1149:    most_likely_sum2 = sum_moravec_points2;
1150:    most_likely_corner = temp2;
1151:        }
1152:     }
1153:
1154:     /*if a sutible point is not found, return the theoretical point*/
1155:     if((most_likely_corner.x <  0) || (most_likely_corner.y <  0))
1156:     {
1157:        most_likely_corner = find_dividers(point1, point2, ONE_THIRD);
1158:     }
```

```
1159:
1160:     return most_likely_corner;
1161:  }
1162:
1163:
1164:  /*return the intersection of two lines*/
1165:  /*1st: coordinate one of first line
1166:    2nd: coordinate two of first line
1167:    3rd: coordinate one of second line
1168:    4th: coordinate two of second line*/
1169:  /*return the coordinate where the lines intersect.  If none-was found
1170:    or an error occured in finding one return (-1, -1).*/
1171:  coord find_intersection(coord line1_point1, coord line1_point2,
1172:                  coord line2_point1, coord line2_point2)
1173:  {
1174:     float line1_slope, line2_slope; /*temp slope holding variables*/
1175:     float denominator, numerator; /*temp fraction holding varaibles*/
1176:     float x_float; /*use to avoid obscure roundoff errors*/
1177:     coord target = {-1, -1}; /*initalize temp target point*/
1178:
1179:     /*find slope for first line*/
1180:     if((line1_point1.x - line1_point2.x) != 0)
1181:     {
1182:        line1_slope = ((float)(line1_point1.y - line1_point2.y)) /
1183:   ((float)(line1_point1.x - line1_point2.x));
1184:     }
1185:     else /*otherwise handle the undefined slope*/
1186:     {
1187:       /*find slope for secon line when first is undefined*/
1188:        if((line2_point1.x - line2_point2.x) != 0)
1189:        {
1190:    line2_slope = ((float)(line2_point1.y - line2_point2.y)) /
1191:      ((float)(line2_point1.x - line2_point2.x));
1192:        }
1193:        else /*this should never happen, but could if someone specifed the same
1194:               line twice*/
1195:    return target; /*target is initalized to (-1, -1)*/
1196:
1197:        /*since the slope is undefined the x coord in known*/
1198:        target.x = line1_point1.x;
1199:        target.y = line2_slope * target.x + line2_point1.y; /*y = mx + b*/
1200:  /*printf("line one has undefined slope\n");*/
1201:        return target;
1202:     }
1203:
1204:     /*find slope for second line*/
1205:     if((line2_point1.x - line2_point2.x) != 0)
1206:     {
1207:        line2_slope = ((float)(line2_point1.y - line2_point2.y)) /
1208:   ((float)(line2_point1.x - line2_point2.x));
1209:     }
1210:     else
1211:     {
1212:        /*since the slope is undefined the x coord in known*/
1213:        target.x = line2_point1.x;
1214:        target.y = line1_slope * target.x + line1_point1.y; /*y = mx + b*/
1215:  /*printf("line two has undefined slope\n");*/
1216:        return target;
```

```
1217:     }
1218:
1219:     /* slope is m and defined by:
1220:           (y1 - y2)
1221:       m = ---------
1222:           (x1 - x2)
1223:
1224:       target calculated by setting Yt = M1*(Xt - Xa) + Ya equal to
1225:       Yt = M2 * (Xt - Xc) + Yc to get
1226:
1227:              (Xa*M1 - M2*Xc + Yc - Ya)
1228:         Xt = -------------------------
1229:                    M1 - M2
1230:
1231:       then to get the y coordinate sub Xt into:
1232:
1233:       Yt = M1 * (Xt - Xa) + Ya
1234:
1235:       Where line1_point1 = a
1236:             line1_point2 = b
1237:             line2_point1 = c
1238:      line12point2 = d
1239:       and are indicated as subscripts of their X or Y coordinate.
1240:             M1 = line1_slope
1241:             M2 = line2_slope
1242:     */
1243:
1244:     /*calculate the x coords nominator and demoninator*/
1245:     numerator = (((float)((float)line1_point1.x * line1_slope)
1246:           - (float)((float)line2_point1.x * line2_slope)
1247:           + (float)line2_point1.y - (float)line1_point1.y));
1248:     denominator = (float)(line1_slope - line2_slope);
1249:
1250:     /*find the x coord, store is as a float to avoid obscure round off errors
1251:       when using it to calculate the y coord.*/
1252:     x_float = numerator / denominator;
1253:
1254:     target.x = (int)x_float;
1255:
1256:     /*find the y coord*/
1257:     target.y = line1_slope * ((float)x_float - (float)line1_point1.x)
1258:       + (float)line1_point1.y;
1259:
1260:     return target;
1261: }
1262:
1263: /*move the location to start searching off of the corners a little so the
1264:   search isn't right on top of the any cross lines that are perpendicular to
1265:   the target line.*/
1266: /*anchor - the corner closest to the side divider point being looked for.
1267: common - the corner that the line between itself and the anchor corner contai
1268:    the target divider point.
1269: anchor_opposite - the corner adjacent to the anchor corner & oppsoite common.
1270: common_opposite - the corner adjacent to the common corner & oppposite anchor
1271: */
1272: /*return the divider point, or (-1, -1) on error*/
1273: coord find_side_points(coord anchor, coord common, coord anchor_opposite,
1274:                 coord common_opposite, PGMImage *moravec,
```

```
1275:                   float t_divider)
1276:  {
1277:       coord temp1, temp2;
1278:
1279:       temp1 = find_dividers(anchor, anchor_opposite, ONE_24TH);
1280:       temp2 = find_dividers(common, common_opposite, ONE_24TH);
1281:
1282:       return search_moravec(temp1, temp2, moravec, t_divider);
1283:  }
1284:
1285:
1286:  /*find the ttt board's important coordinates*/
1287:  /*1st: The object that is determined to be the board.
1288:    2nd: the pointer to the struct that contains all of the information about
1289:    the tic-tac-toe board important coordinates.
1290:    3rd: a pointer to the Moravec pgm image*/
1291:  void detect_ttt_board(object board_object,
1292:                 coord board_details[DIVIDERS][DIVIDERS],
1293:                 PGMImage *moravec)
1294:  {
1295:     int i, j;
1296:
1297:     /*copy the corners into the ttt datatype*/
1298:     if(board_object.corner1.x  > board_object.corner2.x)
1299:     {
1300:        board_details[0][0] = board_object.corner2;
1301:        board_details[DIVIDERS - 1][0] = board_object.corner2;
1302:        board_details[0][DIVIDERS - 1] = board_object.corner1;
1303:        board_details[DIVIDERS - 1][DIVIDERS - 1] = board_object.corner1;
1304:
1305:        if((board_object.corner3.y  > board_object.corner4.y)
1306:     && (board_object.corner3.x  > board_object.corner4.x))
1307:        {
1308:     board_details[DIVIDERS - 1][0] = board_object.corner4;
1309:     board_details[0][DIVIDERS - 1] = board_object.corner3;
1310:        }
1311:        else if((board_object.corner3.y  > board_object.corner4.y)
1312:     && (board_object.corner3.x <  board_object.corner4.x))
1313:        {
1314:     board_details[0][0] = board_object.corner3;
1315:     board_details[DIVIDERS - 1][DIVIDERS - 1] = board_object.corner4;
1316:        }
1317:        else if((board_object.corner3.y <  board_object.corner4.y)
1318:         && (board_object.corner3.x  > board_object.corner4.x))
1319:        {
1320:     board_details[0][0] = board_object.corner4;
1321:     board_details[DIVIDERS - 1][DIVIDERS - 1] = board_object.corner3;
1322:        }
1323:        else
1324:        {
1325:     board_details[DIVIDERS - 1][0] = board_object.corner3;
1326:     board_details[0][DIVIDERS - 1] = board_object.corner4;
1327:        }
1328:
1329:     }
1330:     else
1331:     {
1332:        board_details[0][0] = board_object.corner1;
```

```
1333:         board_details[DIVIDERS - 1][0] = board_object.corner1;
1334:         board_details[0][DIVIDERS - 1] = board_object.corner2;
1335:         board_details[DIVIDERS - 1][DIVIDERS - 1] = board_object.corner2;
1336:
1337:         if((board_object.corner3.y  > board_object.corner4.y)
1338:     && (board_object.corner3.x  > board_object.corner4.x))
1339:         {
1340:     board_details[DIVIDERS - 1][0] = board_object.corner4;
1341:     board_details[0][DIVIDERS - 1] = board_object.corner3;
1342:         }
1343:         else if((board_object.corner3.y  > board_object.corner4.y)
1344:     && (board_object.corner3.x <  board_object.corner4.x))
1345:         {
1346:     board_details[0][0] = board_object.corner3;
1347:     board_details[DIVIDERS - 1][DIVIDERS - 1] = board_object.corner4;
1348:         }
1349:         else if((board_object.corner3.y <  board_object.corner4.y)
1350:          && (board_object.corner3.x  > board_object.corner4.x))
1351:         {
1352:     board_details[0][0] = board_object.corner4;
1353:     board_details[DIVIDERS - 1][DIVIDERS - 1] = board_object.corner3;
1354:         }
1355:         else
1356:         {
1357:     board_details[DIVIDERS - 1][0] = board_object.corner3;
1358:     board_details[0][DIVIDERS - 1] = board_object.corner4;
1359:         }
1360:      }
1361:
1362:      /*loop along the four sides, finding the side points for the cross lines.*
1363:      for(i = 1; i <  DIVIDERS - 1; i++)
1364:      {
1365:         /*"side"*/
1366:         board_details[i][0] =
1367:     find_side_points(board_details[0][0],
1368:                      board_details[DIVIDERS - 1][0],
1369:                      board_details[0][DIVIDERS - 1],
1370:                      board_details[DIVIDERS - 1][DIVIDERS - 1],
1371:                      moravec, i / (DIVIDERS - 1.0));
1372:
1373:         /*side opposite "side"*/
1374:         board_details[i][DIVIDERS - 1] =
1375:     find_side_points(board_details[0][DIVIDERS - 1],
1376:                      board_details[DIVIDERS - 1][DIVIDERS - 1],
1377:                      board_details[0][0],
1378:                      board_details[DIVIDERS - 1][0],
1379:                      moravec, i / (DIVIDERS - 1.0));
1380:
1381:         /*side to the left of "side"*/
1382:         board_details[0][i] =
1383:     find_side_points(board_details[0][0],
1384:                      board_details[0][DIVIDERS - 1],
1385:                      board_details[DIVIDERS - 1][0],
1386:                      board_details[DIVIDERS - 1][DIVIDERS - 1],
1387:                      moravec, i / (DIVIDERS - 1.0));
1388:
1389:         /*side opposite the side to the left of "side"*/
1390:         board_details[DIVIDERS - 1][i] =
```

```
1391:    find_side_points(board_details[DIVIDERS - 1][0],
1392:                     board_details[DIVIDERS - 1][DIVIDERS - 1],
1393:                     board_details[0][0],
1394:                     board_details[0][DIVIDERS - 1],
1395:                     moravec, i / (DIVIDERS - 1.0));
1396:     }
1397:
1398:
1399:     /*correct the length of the lines that is shortend to calculate them
1400:       in the first place above.  simply find the intersection of the *short*
1401:       line with that of the outside line between the corners.*/
1402:
1403:     for(i = 1; i <  DIVIDERS - 1; i++)
1404:     {
1405:        board_details[i][0] = find_intersection(board_details[i][0],
1406:                                                board_details[i][DIVIDERS - 1],
1407:                                                board_details[0][0],
1408:                                                board_details[DIVIDERS - 1][0]);
1409:
1410:        board_details[i][DIVIDERS - 1] =
1411:    find_intersection(board_details[i][DIVIDERS - 1],
1412:                      board_details[i][0],
1413:                      board_details[0][DIVIDERS - 1],
1414:                      board_details[DIVIDERS - 1][DIVIDERS - 1]);
1415:
1416:
1417:        board_details[0][i] = find_intersection(board_details[0][i],
1418:                                                board_details[DIVIDERS - 1][i],
1419:                                                board_details[0][0],
1420:                                                board_details[0][DIVIDERS - 1]);
1421:
1422:
1423:        board_details[DIVIDERS - 1][i] =
1424:    find_intersection(board_details[DIVIDERS - 1][i],
1425:                      board_details[0][i],
1426:                      board_details[DIVIDERS - 1][0],
1427:                      board_details[DIVIDERS - 1][DIVIDERS - 1]);
1428:     }
1429:
1430:     /*now that the sides are found, find the intersections to find the
1431:       middle points*/
1432:     for(i = 1; i <  DIVIDERS - 1; i++)
1433:        for(j = 1; j <  DIVIDERS - 1; j++)
1434:    board_details[i][j] =
1435:       find_intersection(board_details[0][j],
1436:                         board_details[DIVIDERS - 1][j],
1437:                         board_details[i][0],
1438:                         board_details[i][DIVIDERS - 1]);
1439:
1440:     /*print all the coords*/
1441:     /*for(i = 0; i <  DIVIDERS; i++)
1442:        for(j = 0; j <  DIVIDERS; j++)
1443:    printf("points[%d][%d]: (%d, %d)\n", i, j, points[i][j].x,
1444:    points[i][j].y);*/
1445:  }
1446:
1447:  /*not a callback, called by menu() and buffer()*/
1448:  int showGame()
```

```
1449:  {
1450:      int object; /*temp variable*/
1451:      struct timeval t0, t1;
1452:
1453:      printf("\n\n");
1454:
1455:      /*correct image perspective*/
1456:      pthread_mutex_lock(&images_mutex);
1457:
1458:      img_cur = img_original;
1459:      gettimeofday(&t0, NULL);
1460:      pers_corr(img_pers_corr, img_cur);
1461:      gettimeofday(&t1, NULL);
1462:      do_time(t0, t1, "perspective correction");
1463:
1464:      img_cur = img_pers_corr;
1465:
1466:      pthread_mutex_unlock(&images_mutex);
1467:
1468:
1469:
1470:      /*find chain codes*/
1471:      pthread_mutex_lock(&calculate_mutex);
1472:      pthread_mutex_lock(&images_mutex);
1473:
1474:      gettimeofday(&t0, NULL);
1475:      showChain(img_pers_corr, &chain_codes); /*chain codes*/
1476:      gettimeofday(&t1, NULL);
1477:      do_time(t0, t1, "chain codes");
1478:
1479:      pthread_mutex_unlock(&images_mutex);
1480:      pthread_mutex_unlock(&calculate_mutex);
1481:
1482:
1483:
1484:      /*find corners*/
1485:      pthread_mutex_lock(&calculate_mutex);
1486:
1487:      gettimeofday(&t0, NULL);
1488:      detect_corners(chain_codes, all_objects);
1489:      gettimeofday(&t1, NULL);
1490:      do_time(t0, t1, "corner detection");
1491:
1492:      pthread_mutex_unlock(&calculate_mutex);
1493:
1494:
1495:
1496:
1497:      /*detect the ttt game board*/
1498:      pthread_mutex_lock(&calculate_mutex);
1499:
1500:      gettimeofday(&t0, NULL);
1501:      object = detect_game(chain_codes, all_objects);
1502:      gettimeofday(&t1, NULL);
1503:      do_time(t0, t1, "game detection");
1504:
1505:      pthread_mutex_unlock(&calculate_mutex);
1506:
```

```
1507:
1508:     if(object == -1)
1509:     {
1510:        if(display_verbose)
1511:    printf("No board was found\n");
1512:        return object;
1513:     }
1514:
1515:
1516:     /*do moravec first, then perspective to prevent the loss of edge sharpness
1517:       from doing the perspective first*/
1518:     pthread_mutex_lock(&images_mutex);
1519:
1520:     gettimeofday(&t0, NULL);
1521:     moravec(img_grayscale, img_original);
1522:     pers_corr(img_moravec, img_grayscale);
1523:     gettimeofday(&t1, NULL);
1524:     do_time(t0, t1, "Moravec filter");
1525:
1526:     pthread_mutex_unlock(&images_mutex);
1527:
1528:
1529:
1530:
1531:     /*if a board was found, find the specific information*/
1532:     pthread_mutex_lock(&images_mutex);
1533:     pthread_mutex_lock(&calculate_mutex);
1534:
1535:     gettimeofday(&t0, NULL);
1536:     detect_ttt_board(all_objects[object], points, img_moravec);
1537:     gettimeofday(&t1, NULL);
1538:     do_time(t0, t1, "detect tic-tac-toe");
1539:
1540:     pthread_mutex_unlock(&calculate_mutex);
1541:     pthread_mutex_unlock(&images_mutex);
1542:
1543:
1544:     return object;
1545: }
1546:
1547:
1548: /*****Find the pieces*****************************************************
1549: ************************************************************************/
1550:
1551: /*Takes four points that must be in order going around the region.  Also,
1552:   needs a pointer to an image with the moravec filter applied.*/
1553: /*1st - 4th: The coordinates in order (starting point arbitrary, as long
1554:   as they are in order) that the area inside of is searched for pieces.
1555:   5th: a pointer to the Moravec image to be used.
1556:   6th: a pointer to the memory location were the averages of the pixels
1557:   vals for the marble are stored. (only valid when return value is TRUE)
1558: */
1559: /*returns TRUE if a marble exists at the location, FALSE otherwise.*/
1560: bool search_area(coord point1, coord point2, coord point3, coord point4,
1561:          PGMImage *moravec, PGMImage *color, RGB_INT *area_avg)
1562: {
1563:     /*The following variables are used to calculate the existence of a marble
1564:       at a location on the board or not.  Uses the following basic formula
```

```
1565:          new_point = ((t-1) * 1st point) + (t * 2nd point)
1566:        where the start & end coords are opposite sides of the area being
1567:        searched.  If you take the entire area of a single position on a ttt
1568:        board, the area that is searched is the region that is bouned by the
1569:        points halfway between those passed in to this function.*/
1570:
1571:     int search_size = 2, search_area, i, j;
1572:
1573:     /*cant use pixel_sum directly becuase it is only */
1574:     RGB_INT pixel_sum = {0, 0, 0};
1575:     int pixel_sum_red = 0, pixel_sum_green = 0, pixel_sum_blue = 0;
1576:
1577:     float t, s, r; /*hold values incrementing from 0 to 1*/
1578:     coord t_start, t_end, s_start, s_end;
1579:     coord temp_t, temp_s, temp_r;
1580:     float dist_t, dist_s, dist_r;
1581:
1582:     /*pixel count of those over the threashhold and total pixels*/
1583:     int pixel_count = 0, moravec_count = 0;
1584:
1585:     /*running storage for the middle of the marble*/
1586:     coord sum_locations = {0, 0}; /*sum of x and y coords for average location
1587:
1588:     coord move_start1, move_start2, move_start3, move_start4;
1589:
1590:     /*this translation of moving the starting points skews the search area
1591:       off enough to avoid counting "hit" boundry pixels that exist along the
1592:       divders/edges and not the marble itself.*/
1593:     move_start1 = find_dividers(point1, point2, ONE_THIRD);
1594:     move_start2 = find_dividers(point2, point3, ONE_THIRD);
1595:     move_start3 = find_dividers(point3, point4, ONE_THIRD);
1596:     move_start4 = find_dividers(point4, point4, ONE_THIRD);
1597:
1598:     t_start = find_dividers(move_start1, move_start2, ONE_HALF);
1599:     t_end = find_dividers(move_start1, move_start4, ONE_HALF);
1600:
1601:     s_start = find_dividers(move_start3, move_start2, ONE_HALF);
1602:     s_end = find_dividers(move_start3, move_start4, ONE_HALF);
1603:
1604:     dist_t = findDist(t_start.x, t_start.y, t_end.x, t_end.y);
1605:     dist_s = findDist(s_start.x, s_start.y, s_end.x, s_end.y);
1606:
1607:     /*march two points along that parallel each other in the search area*/
1608:     for(t = 0.0, s = 0.0; t < = 1.0; t += (1.0 / dist_t), s += (1.0 / dist_s))
1609:     {
1610:        temp_t = find_dividers(t_start, t_end, t);
1611:
1612:        temp_s = find_dividers(s_start, s_end, s);
1613:
1614:        dist_r = findDist(temp_t.x, temp_t.y, temp_s.x, temp_s.y);
1615:
1616:        /*march a single point along that starts at temp_s and ends
1617:  at temp_t.  This fills in the region for the search.*/
1618:        for(r = 0.0; r < = 1.0; r += (1 / dist_r))
1619:        {
1620:    temp_r = find_dividers(temp_s, temp_t, r);
1621:
1622: /*setCPixel(temp_r.x, temp_r.y, red);*/
```

```
1623:    /*talley the number of edge points found (rgb. avg.  >= 128)*/
1624:    if(rgb_avg((*moravec).data[temp_r.y][temp_r.x])  >= EDGES_FOR_PIECE)
1625:    {
1626:       sum_locations.x += temp_r.x;
1627:       sum_locations.y += temp_r.y;
1628:  /*setCPixel(temp_r.x, temp_r.y, blue);*/
1629:       moravec_count++;
1630:    }
1631:    pixel_count++; /*increment the number of pixels*/
1632:        }
1633:
1634:     }
1635:
1636:     if(moravec_count)
1637:     {
1638:        sum_locations.x /= moravec_count;
1639:        sum_locations.y /= moravec_count;
1640:     }
1641:
1642:     /*if the number of edge pixels is greater than 25% of the total, then
1643:       this is an edge of marble.  This is based on the fact the the percentage
1644:       of pixels highlighted theoritically be 25% of the total count.  By
1645:       dividing the circumfernce of the marble (in cm) by the area of the searc
1646:       Since the radius is 2cm and the length and width are 5cm the math works
1647:       out to 25% of the total (theoretically).  However, since the starting
1648:       points are moved 1/3 off the corners this theo. percentage jumps to
1649:       45% for the max.  For practical implementation EDGE_COUNT is set
1650:       to .25 or 25 percent.*/
1651:     /*printf("ttt fp mc: %d  tc: %d   per: %f\n", moravec_count, pixel_count,
1652:       ((float)moravec_count / (float)pixel_count) * 100.0);*/
1653:
1654:     if(moravec_count   >= EDGE_COUNT * pixel_count)
1655:     {
1656:        for(i = -search_size; i < = search_size; i++) /*y dir*/
1657:    for(j = -search_size; j < = search_size; j++) /*x dir*/
1658:    {
1659:       pixel_sum_red += (*color).data
1660:          [sum_locations.y + i][sum_locations.x + j].red;
1661:       pixel_sum_green += (*color).data
1662:          [sum_locations.y + i][sum_locations.x + j].green;
1663:       pixel_sum_blue += (*color).data
1664:          [sum_locations.y + i][sum_locations.x + j].blue;
1665:
1666:  /*setCPixel(sum_locations.x + j, sum_locations.y + i, yellow);*/
1667:
1668:    }
1669:  /*setCPixel(sum_locations.x, sum_locations.y, green);
1670:  glFlush();*/
1671:
1672:        search_area = (((search_size * 2) + 1) * ((search_size * 2) + 1));
1673:
1674:        pixel_sum.red =
1675:    pixel_sum_red / search_area;
1676:        pixel_sum.green =
1677:    pixel_sum_green / search_area;
1678:        pixel_sum.blue =
1679:    pixel_sum_blue / search_area;
1680:
```

```
1681:         memcpy(area_avg, &pixel_sum, sizeof(RGB_INT));
1682:         /*      printf("pxl avg: %d\n", pixel_sum);*/
1683:         return TRUE;
1684:      }
1685:
1686:      /*otherwise a marble isn't here*/
1687:      return FALSE;
1688:   }
1689:
1690:
1691:   int weight_colors(RGB_INT color)
1692:   {
1693:      int red_to_green = abs(color.red - color.green);
1694:      int red_to_blue = abs(color.red - color.blue);
1695:      int green_to_blue = abs(color.green - color.blue);
1696:
1697:      return red_to_green + red_to_blue + green_to_blue;
1698:   }
1699:
1700:
1701:   /*looks for the tic-tac-toe pieces on the board*/
1702:   /*1st: the pointer to the board information.
1703:     2nd: the pointer to the Moravec image used to find the edges.*/
1704:   void detect_pieces(coord board_data[DIVIDERS][DIVIDERS],
1705:                      PGMImage *moravec, PGMImage *color)
1706:   {
1707:      int i, j; /*loop counting*/
1708:      int teams_pixel_avg, team1_val = 255, team2_val = 0;
1709:      RGB_INT store[3][3]; /*area pixel averages*/
1710:      int team1_assign = 1, team2_assign = 2;
1711:
1712:      if(display_verbose)
1713:         printf("Finding pieces.\n");
1714:
1715:      memset(pieces, 0, 9 * sizeof(unsigned char));
1716:      memset(store, 0, 9 * sizeof(RGB_INT));
1717:
1718:      for(i = 0; i <  DIVIDERS - 1; i++) /*y*/
1719:        for(j = 0; j <  DIVIDERS - 1; j++) /*x*/
1720:        {
1721:          pieces[i][j] = search_area(board_data[i][j],
1722:                                     board_data[i + 1][j],
1723:                                     board_data[i + 1][j + 1],
1724:                                     board_data[i][j + 1],
1725:                                     moravec, color, &store[i][j]);
1726:        }
1727:
1728:      /*first position to check*********************************/
1729:      for(i = 0; i <  DIVIDERS - 1; i++) /*y*/
1730:      {
1731:         for(j = 0; j <  DIVIDERS - 1; j++) /*x*/
1732:         {
1733:     if(!pieces[i][j]) /*if there isn't a piece*/
1734:        continue;
1735:
1736:     pieces[i][j] = weight_colors(store[i][j]);
1737:
1738:     if(pieces[i][j] <  team1_val)
```

```
1739:    {
1740:       team1_val = pieces[i][j];
1741:    }
1742:
1743:    if(pieces[i][j]  > team2_val)
1744:    {
1745:       team2_val = pieces[i][j];
1746:    }
1747:        }
1748:     } /*for*******************************************************/
1749:
1750:     if(team1_val == team2_val) /*only one move made so far*/
1751:        /*at this point we don't know which color each team is.  So, store
1752:  this into the team_first global.*/
1753:        team_first = team1_val;
1754:
1755:     teams_pixel_avg = (team1_val + team2_val) / 2;
1756:
1757:     /*the first team to go is assigned team 1.  If the color determining
1758:       sets it to team 2, then this loop sets what needs to be switched.*/
1759:     if(abs(team_first - team1_val)  > abs((team1_val - team2_val) / 2))
1760:     {
1761:        team1_assign = 2;
1762:        team2_assign = 1;
1763:     }
1764:
1765:     for(i = 0; i <  3; i++) /*y*/
1766:     {
1767:        for(j = 0; j <  3; j++) /*x*/
1768:        {
1769:    if(pieces[i][j])
1770:    {
1771:       /*check if team 1*/
1772:       if(pieces[i][j] < = teams_pixel_avg)
1773:       {
1774:          pieces[i][j] = team1_assign;
1775:       }
1776:       /*check if team 2*/
1777:       else if(pieces[i][j]  >= teams_pixel_avg)
1778:       {
1779:          pieces[i][j] = team2_assign;
1780:       }
1781:       /*to many different colored teams*/
1782:       else
1783:       {
1784:          pieces[i][j] = -1;
1785:       }
1786:    } /*if(pieces[i][j]*/
1787:        }
1788:     }
1789:  }
1790:
1791:  /*display the state of the board in ascii graphics*/
1792:  void display_board()
1793:  {
1794:     int i, j;
1795:
1796:     printf("    ");
```

```
1797:     for(j = 0; j <  DIVIDERS - 1; j++)
1798:        printf(" [%d] ", j);
1799:     printf("\n-------------------\n");
1800:
1801:     for(i = 0; i <  DIVIDERS - 1; i++)
1802:     {
1803:        printf("[%d] ", i);
1804:        for(j = 0; j <  DIVIDERS - 1; j++)
1805:        {
1806:   printf("  %d  ", pieces[i][j]);
1807:        }
1808:        printf("\n");
1809:     }
1810: }
1811:
1812: void showPieces()
1813: {
1814:     struct timeval t0, t1;
1815:     /*if we get here there is a likely canidate found*/
1816:     pthread_mutex_lock(&calculate_mutex);
1817:     pthread_mutex_lock(&images_mutex);
1818:
1819:     gettimeofday(&t0, NULL);
1820:     detect_pieces(points, img_moravec, img_cur);
1821:     gettimeofday(&t1, NULL);
1822:     do_time(t0, t1, "detect peices");
1823:
1824:     if(display_verbose)
1825:        display_board();
1826:
1827:     pthread_mutex_unlock(&images_mutex);
1828:     pthread_mutex_unlock(&calculate_mutex);
1829: }
1830:
1831:
1832: /* ====================================================================
1833:  * Callback functions.
1834:  *
1835:  * color = displayed graphics in window
1836:  * menu = menu event handling
1837:  * keyboard = deyboard event handling
1838:  * ---------------------------------------------------------------- */
1839: void color(void)
1840: {
1841:     glutReshapeWindow(HSIZE, VSIZE);
1842:
1843:     if(pthread_mutex_trylock(&images_mutex) == EBUSY)
1844:     {
1845:        return;
1846:     }
1847:
1848:     /*show the current image*/
1849:     showColor(img_cur);
1850:     pthread_mutex_unlock(&images_mutex);
1851:
1852:     if(pthread_mutex_trylock(&calculate_mutex) == EBUSY)
1853:     {
1854:        return;
```

```
1855:     }
1856:     /*show the current abstract information*/
1857:     showAbstract(all_objects, points);
1858:     pthread_mutex_unlock(&calculate_mutex);
1859:  }
1860:
1861:  void sie(void)
1862:  {
1863:     glutReshapeWindow(HSIZE, VSIZE);
1864:
1865:     if(redraw_needed)
1866:     {
1867:        pthread_mutex_lock(&images_mutex);
1868:        showColor(img_cur);
1869:        redraw_needed = FALSE;
1870:        pthread_mutex_unlock(&images_mutex);
1871:     }
1872:     else
1873:        sched_yield();
1874:  }
1875:
1876:  void* buffer(void *param)
1877:  {
1878:     while(is_buffered)
1879:     {
1880:       /*the mutexes for these calcs are done inside showGame()*/
1881:        draw_abstract_board = showGame();
1882:
1883:        if(draw_abstract_board != -1)
1884:        {
1885:    showPieces();
1886:        }
1887:     }
1888:
1889:     return NULL;
1890:  }
1891:
1892:
1893:
1894:  #define RESTART 0
1895:  #define PERS_CORR 3
1896:  #define COLOR_TO_GRAY 4
1897:  #define MORAVEC 5
1898:  #define EDGES 6
1899:  #define CORNERS 7
1900:  #define BUFFERS 8
1901:  #define GAME 9
1902:  #define PIECES 10
1903:
1904:  /*this is not a callback function, but is used inside menu() for setting
1905:    new states of execution*/
1906:  void reset_state(PGMImage* new_current)
1907:  {
1908:     img_cur = new_current;
1909:     draw_abstract_lines = -1;
1910:     draw_abstract_board = -1;
1911:     free_chaincodes(&chain_codes);
1912:  }
```

```
1913:
1914: void menu(int selection)
1915: {
1916:    if(selection == RESTART)
1917:    {
1918:       reset_state(img_original);
1919:    }
1920:    if(selection == PERS_CORR)
1921:    {
1922:       pers_corr(img_pers_corr, img_cur);
1923:       reset_state(img_pers_corr);
1924:    }
1925:    if(selection == COLOR_TO_GRAY)
1926:    {
1927:       color_to_gray(img_grayscale, img_cur);
1928:       reset_state(img_grayscale);
1929:    }
1930:    if(selection == MORAVEC)
1931:    {
1932:       moravec(img_moravec, img_cur);
1933:       reset_state(img_moravec);
1934:    }
1935:    if(selection == EDGES)
1936:    {
1937:       showChain(img_cur, &chain_codes);
1938:    }
1939:    if(selection == CORNERS)
1940:    {
1941:       showChain(img_cur, &chain_codes);
1942:
1943:       draw_abstract_lines = detect_corners(chain_codes, all_objects);
1944:    }
1945:    if(selection == BUFFERS)
1946:    {
1947:       if(is_buffered == FALSE)
1948:       {
1949:   is_buffered = TRUE;
1950:   pthread_create(&vision_thread, NULL, &buffer, NULL);
1951:   glutIdleFunc(&color);
1952:       }
1953:       else
1954:       {
1955:   is_buffered = FALSE;
1956:   pthread_join(vision_thread, NULL);
1957:   glutIdleFunc(NULL);
1958:       }
1959:    }
1960:    if(selection == GAME)
1961:    {
1962:       draw_abstract_board = showGame();
1963:    }
1964:    if(selection == PIECES)
1965:    {
1966:       draw_abstract_board = showGame();
1967:
1968:       if(draw_abstract_board != -1)
1969:   showPieces();
1970:    }
```

```
1971:
1972:     glutPostRedisplay(); /*redraw the image*/
1973:  }
1974:
1975:  void keyboard(unsigned char key, int x, int y)
1976:  {
1977:     switch (key)
1978:     {
1979:        case 27: /*Esc*/
1980:           exit(0);
1981:           break;
1982:
1983:    /*save current image to file image1.pgm*/
1984:        case 115: /*s*/
1985:    pthread_mutex_lock(&images_mutex);
1986:    save(img_cur);
1987:    pthread_mutex_unlock(&images_mutex);
1988:    break;
1989:
1990:    /*save original image to file image1.pgm*/
1991:        case 111: /*o*/
1992:    pthread_mutex_lock(&images_mutex);
1993:    save(img_original);
1994:    pthread_mutex_unlock(&images_mutex);
1995:    break;
1996:
1997:    /*enable display of calculation times.*/
1998:        case 116: /*t*/
1999:    if(display_time)
2000:       display_time = FALSE;
2001:    else
2002:       display_time = TRUE;
2003:    break;
2004:
2005:    /*enable display of verbose text.*/
2006:        case 118: /*v*/
2007:    if(display_verbose)
2008:       display_verbose = FALSE;
2009:    else
2010:       display_verbose = TRUE;
2011:    break;
2012:     }
2013:  }
2014:
2015:  void mouse(int button, int state, int x, int y)
2016:  {
2017:    char temp[50];
2018:
2019:    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
2020:    {
2021:       sprintf(temp, "(x, y): (%d, %d)  red: %d green: %d blue: %d\n",
2022:        x, VSIZE - y, (*img_cur).data[VSIZE - y][x].red,
2023:        (*img_cur).data[VSIZE - y][x].green,
2024:        (*img_cur).data[VSIZE - y][x].blue);
2025:       setCRect(0, 0, 200, 12, black);
2026:       glColor3f(1.0, 0.0, 0.0);
2027:       drawString(0, 0, temp, red);
2028:       glFlush();
```

```
2029:    }
2030:  }
2031:
2032:
2033:  /* ------------------------------------------------------------------
2034:     Connection functions for work with sie.
2035:     ------------------------------------------------------------------ */
2036:
2037:  /* now connect to the administrator */
2038:  int admin_connect(int argc, const char** argv)
2039:  {
2040:     int sock;
2041:
2042:     sock = device_handshake (argc, argv,
2043:                         TTT_VISION_DEVICE_TYPE_ID_NUMBER, 1);
2044:
2045:     if (sock == -1)
2046:     {
2047:        printf ("failed in handshake; aborting...\n");
2048:        exit (1);
2049:     }
2050:
2051:     return sock;
2052:  }
2053:
2054:  void* sock_com(void* ptr)
2055:  {
2056:     PGMImage *img = (PGMImage*)ptr;
2057:     int sock = socket_num;
2058:     int8 value8, command;
2059:     int32 value32, return_val;
2060:     int x, y;
2061:
2062:     struct timeval t0, t1; /*time 0 and finished time*/
2063:
2064:     /******* main loop  ******/
2065:     command = TTT_VISION_INIT_VALUE;
2066:     while (command != ADMIN_DISCONNECT_DEVICE)
2067:     {
2068:       /* block until we get the next command */
2069:        if ((shelley_sockets_read (sock, (char*) &command, sizeof (int8))) == -
2070:    exit (0);
2071:
2072:        else
2073:        {
2074:   /*printf("ttt command: %d\n", command);*/
2075:    switch (command)
2076:    {
2077:       case ADMIN_QUERY_DEVICE_STATUS:
2078:          value8 = DEVICE_A_OK;
2079:          shelley_sockets_write (sock, &value8, 1);
2080:          break;
2081:
2082:       case TTT_VISION_SET_RESOLUTION:
2083:          shelley_sockets_read (sock, (char*) &value32, sizeof (int32));
2084:          (*img).width = HSIZE = value32;
2085:          shelley_sockets_read (sock, (char*) &value32, sizeof (int32));
2086:          (*img).height = VSIZE = value32;
```

```
2087:          (*img).maxVal = MVAL = 255;
2088:
2089:          /*we need to use a buffer the same size as the data being read
2090:            in since the PGMImage type declares the type as  MAX by MAX
2091:            in size.  Without this buffer the data could not be unpacked
2092:            from the socket stream and into the PGMImage in the correct
2093:            way.*/
2094:          data = (RGB_INT*)malloc(HSIZE * VSIZE * sizeof(RGB_INT));
2095:
2096:          return_val = 1;
2097:          shelley_sockets_write (sock, (char*) &return_val,
2098:                                 sizeof (int32));
2099:          value8 = 99; /*the admin expects something always returned*/
2100:          shelley_sockets_write (sock, (char*) &value8, sizeof (int8));
2101:          break;
2102:
2103:       case TTT_VISION_FIND_TTT:
2104:          /*read the data into a region of memory before placing
2105:           it into the image data in the correct way.*/
2106:          value32 = shelley_sockets_read (sock, (unsigned char*) data,
2107:                                          HSIZE * VSIZE * 3);
2108:
2109:          pthread_mutex_lock(&images_mutex);
2110:
2111:          /*repack the data into the correct storage size.*/
2112:          for (y = 0; y <  VSIZE; y++)
2113:          {
2114:             for (x = 0; x <  HSIZE; x++)
2115:             {
2116:                (*img).data[y][x].red = data[(y * HSIZE) + x].red;
2117:                (*img).data[y][x].green = data[(y * HSIZE) + x].green;
2118:                (*img).data[y][x].blue = data[(y * HSIZE) + x].blue;
2119:             }
2120:          }
2121:                  pthread_mutex_unlock(&images_mutex);
2122:
2123:          img_cur = img;
2124:
2125:          /*do the timing stuff*/
2126:          gettimeofday(&t0, NULL);
2127:          menu(PIECES); /*using the menu function is the lazy way*/
2128:          gettimeofday(&t1, NULL);
2129:          do_time(t0, t1, "total");
2130:
2131:          value32 = 9 * sizeof(unsigned char);
2132:
2133:          shelley_sockets_write (sock, (char*) &value32, sizeof(int32));
2134:          shelley_sockets_write (sock, (char*) pieces, value32);
2135:          break;
2136:
2137:       case ADMIN_DISCONNECT_DEVICE:
2138:          /* we're supposed to quit, now */
2139:          exit (0);
2140:          break;
2141:
2142:       default:
2143:          printf("TTT_VISION: unknown command: %d\n", command);
2144:    }
```

```
2145:         }
2146:      }
2147:      pthread_exit(NULL);
2148:      return NULL; /*compilers complain otherwise*/
2149:   }
2150:
2151:
2152:   /* ===================================================================
2153:    * init() & parse_command_line()
2154:    * initalize none-OpenGL related things.
2155:    * ------------------------------------------------------------------- */
2156:
2157:   void usage(const char* name)
2158:   {
2159:      printf("Usage: %s < [-h < hostname > -p < port_number >] [-f < filename >]
2160:      "  -h      - remote host\n"
2161:      "  -p      - port number\n"
2162:      "  -f      - file name\n"
2163:      "  -b      - buffer\n"
2164:      "\n", name);
2165:   }
2166:
2167:   /*set global flag variables from things specified at commandline.*/
2168:   /*return the filename specified, if none specified exit().*/
2169:   void parse_command_line(int argc, const char** argv)
2170:   {
2171:     /*int c;*/
2172:      hostname = (char*)malloc (1024);
2173:      port_number = SIE_DEFAULT_PORT_NUMBER;/*needs to be default sie port*/
2174:      PGMfileName = NULL;
2175:      is_buffered = FALSE;
2176:
2177:      strcpy(hostname, "localhost");
2178:
2179:      if(argc == 2)
2180:         PGMfileName = (char*)argv[1];
2181:
2182:      /*parse the command line*/
2183:      /*while((c = getopt(argc, argv, GETOPTARGS)) != EOF)
2184:      {
2185:         switch(c)
2186:         {
2187:      case 'h':
2188:         strcpy(hostname, optarg);
2189:         is_buffered = TRUE;
2190:         break;
2191:
2192:      case 'p':
2193:         port_number = atoi(optarg);
2194:         break;
2195:
2196:      case 'f':
2197:         PGMfileName = optarg;
2198:         break;
2199:
2200:      case 'b':
2201:         is_buffered = TRUE;
2202:         break;
```

```
2203:
2204:    default:
2205:       usage(argv[0]);
2206:       exit(1);
2207:        }
2208:        }*/
2209:
2210:  }
2211:
2212:  void init_image(PGMImage *image)
2213:  {
2214:     memset(image, 0, sizeof(PGMImage));
2215:     (*image).width = HSIZE;
2216:     (*image).height = VSIZE;
2217:     (*image).maxVal = MVAL;
2218:  }
2219:
2220:
2221:  void init (int argc, const char **argv)
2222:  {
2223:     int threaded;
2224:
2225:     /*parse the command line into globals*/
2226:     parse_command_line(argc, argv);
2227:
2228:
2229:
2230:  /*
2231:   * Read in image file. - note: sets our global values, too.
2232:   * ------------------------------------------------------------------ */
2233:
2234:     /*allocate memory for original image*/
2235:     img_original = (PGMImage*) malloc(sizeof(PGMImage));
2236:     memset(img_original, 0, sizeof(PGMImage));
2237:     img_pers_corr = (PGMImage*) malloc(sizeof(PGMImage));
2238:     memset(img_pers_corr, 0, sizeof(PGMImage));
2239:     img_grayscale = (PGMImage*) malloc(sizeof(PGMImage));
2240:     memset(img_grayscale, 0, sizeof(PGMImage));
2241:     img_moravec = (PGMImage*) malloc(sizeof(PGMImage));
2242:     memset(img_moravec, 0, sizeof(PGMImage));
2243:
2244:     if(argc == 3)
2245:     {
2246:        socket_num = admin_connect(argc, argv);
2247:        threaded = pthread_create(&conn_thread, NULL, sock_com,
2248:                                  (void*)img_original);
2249:        sleep(1); /*give new thread time to make copy of sock_number*/
2250:        /*set the filename so the text in the window says the hostname*/
2251:        PGMfileName = (char*)malloc(1024); /*this better be enough*/
2252:        strcpy(PGMfileName, hostname);
2253:
2254:        img_cur = img_original; /*VERY IMPORTANT to set these*/
2255:        HSIZE = 100;
2256:        VSIZE = 100;
2257:        MVAL = 255;
2258:        /*
2259:        (*img_original).width = HSIZE;
2260:        (*img_original).height = VSIZE;
```

```
2261:        (*img_original).maxVal = MVAL;
2262:
2263:        (*img_pers_corr).width = HSIZE;
2264:        (*img_pers_corr).height = VSIZE;
2265:        (*img_pers_corr).maxVal = MVAL;
2266:
2267:        (*img_grayscale).width = HSIZE;
2268:        (*img_grayscale).height = VSIZE;
2269:        (*img_grayscale).maxVal = MVAL;
2270:
2271:        (*img_moravec).width = HSIZE;
2272:        (*img_moravec).height = VSIZE;
2273:        (*img_moravec).maxVal = MVAL;
2274:        */
2275:     }
2276:     else if(argc == 2)
2277:     {
2278:        getPGMfile(PGMfileName, img_original);
2279:        HSIZE = (*img_original).width;
2280:        VSIZE = (*img_original).height;
2281:        MVAL = (*img_original).maxVal;
2282:
2283:        img_cur = img_original; /*VERY IMPORTANT to set this*/
2284:
2285:        /*allocate memory for pers. corr. image*/
2286:
2287:        init_image(img_pers_corr);
2288:        init_image(img_grayscale);
2289:        init_image(img_moravec);
2290:     }
2291:     else
2292:     {
2293:        usage(argv[0]);
2294:        exit(1);
2295:     }
2296:
2297:     all_objects = (object*) malloc(sizeof(object) * MAX_CHAINS);
2298:     memset(all_objects, 0, sizeof(object) * MAX_CHAINS);
2299:
2300:     memset(points, 0, sizeof(points));
2301:     memset(pieces, 0, sizeof(pieces));
2302:  }
2303:
2304:
2305:  int main(int argc, const char** argv)
2306:  {
2307:     int WindowID; /*unique window id, there is only one used*/
2308:
2309:  /*
2310:   * Call our init function to define non-OpenGL related things.
2311:   * ---------------------------------------------------------------- */
2312:     init (argc, argv);
2313:
2314:  /*
2315:   * Initialize the glut package.
2316:   * ---------------------------------------------------------------- */
2317:
2318:     glutInit(&argc, (char**)argv);
```

```
2319:     glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
2320:
2321:   /*
2322:    * Define a new window (its size, position and title).
2323:    * ------------------------------------------------------------- */
2324:     /*glutInitWindowSize (HSIZE, VSIZE);*/
2325:     glutInitWindowPosition (10, 10);
2326:     WindowID = glutCreateWindow (PGMfileName);
2327:     glutSetWindow(WindowID);
2328:     glutDisplayFunc(color);
2329:
2330:   /*
2331:    * select clearing color - white
2332:    */
2333:
2334:     glClearColor (1.0, 1.0, 1.0, 0.0);
2335:
2336:
2337:   /*
2338:    * initialize viewport values
2339:    */
2340:     glMatrixMode(GL_PROJECTION);
2341:     glLoadIdentity();
2342:     glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
2343:
2344:     /*add menus*/
2345:     if(argc == 2)
2346:     {
2347:        glutCreateMenu(menu);
2348:        glutAddMenuEntry("restart", RESTART);
2349:        glutAddMenuEntry("perspective correction", PERS_CORR);
2350:        glutAddMenuEntry("color to grayscale", COLOR_TO_GRAY);
2351:        glutAddMenuEntry("Moravec filter", MORAVEC);
2352:        glutAddMenuEntry("detect edges", EDGES);
2353:        glutAddMenuEntry("detect corners", CORNERS);
2354:        glutAddMenuEntry("detect game", GAME);
2355:        glutAddMenuEntry("detect pieces", PIECES);
2356:        glutAddMenuEntry("toggle buffering", BUFFERS);
2357:        glutAttachMenu(GLUT_RIGHT_BUTTON);
2358:     }
2359:     else
2360:        glutIdleFunc(sie);
2361:
2362:     glutMouseFunc(mouse);
2363:     glutKeyboardFunc(keyboard);
2364:
2365:     glutMainLoop();
2366:
2367:   /*
2368:    * When we reach here, we've left the event loop and are ready to
2369:    * exit.
2370:    * ------------------------------------------------------------- */
2371:     return 0;
2372:   }
2373:
```

Appendix B - ttt_vision.h

```
1:  /*ttt_vision.h*/
2:
3:  #ifndef RESEARCH_H
4:  #define RESEARCH_H
5:
6:  #include "bool.h"
7:  #include "pgm.h"
8:  #include "common.h"
9:  #include "linked_list.h"
10:  #include "chaincode.h"
11:
12:  /******Prototypes*****************************************************/
13:
14:  void setCPixel(int ix, int iy, RGB_INT color);
15:
16:  void setCLines(int ix1, int iy1, int ix2, int iy2, RGB_INT color);
17:
18:  void setCRect(int ix1, int iy1, int ix2, int iy2, RGB_INT color);
19:
20:  void pxlcpy(PGMImage *dest, int dest_row, int dest_col,
21:         PGMImage *src, int src_row, int src_col);
22:
23:  int rgb_avg(RGB_INT cur_pxl);
24:
25:  int img_pxl_avg(PGMImage* img);
26:
27:  int pxlcmp (RGB_INT pxl1, RGB_INT pxl2, int range);
28:
29:  void drawString(int ix, int iy, char theString[256], RGB_INT color);
30:
31:  int background(int treash_value, PGMImage* img);
32:
33:  coord find_dividers(coord point1, coord point2, float t);
34:
35:  #endif
36:
```

Appendix C - ttt_vision_protocol.h

```
1:
2:
3:  #ifndef __TTT_VISION_PROTOCOL_H__
4:  #define __TTT_VISION_PROTOCOL_H__
5:
6:  #define TTT_VISION_INIT_VALUE 0
7:
8:  #define TTT_VISION_SET_RESOLUTION 1
9:
10:  #define TTT_VISION_FIND_TTT 2
11:
12:  #endif
13:
```

Appendix D - common.h

```
1:  #ifndef COMMON_H
2:  #define COMMON_H
3:
4:  /*cartesian coordinate type*/
5:  typedef struct
6:  {
7:     int x;
8:     int y;
9:  }coord;
10:
11:  #endif
```

Appendix E - colors.h

```
1:  /*colors.h*/
2:
3:  const RGB_INT white      = {255, 255, 255};
4:  const RGB_INT yellow     = {255, 255,   0};
5:  const RGB_INT magenta    = {255,   0, 255};
6:  const RGB_INT cyan       = {0,   255, 255};
7:  const RGB_INT red        = {255,   0,   0};
8:  const RGB_INT green      = {0,   255,   0};
9:  const RGB_INT blue       = {0,     0, 255};
10:  const RGB_INT black      = {0,     0,   0};
11:
12:  const RGB_INT gray       = {128, 128, 128};
13:  const RGB_INT lt_yellow  = {255, 255, 128};
14:  const RGB_INT lt_magenta = {255, 128, 255};
15:  const RGB_INT lt_cyan    = {128, 255, 255};
16:
17:  const RGB_INT lt_red     = {255, 128, 128};
18:  const RGB_INT lt_green   = {128, 255, 128};
19:  const RGB_INT lt_blue    = {128, 128, 255};
20:
21:  const RGB_INT dk_yellow  = {128, 128,   0};
22:  const RGB_INT dk_magenta = {128,   0, 128};
23:  const RGB_INT dk_cyan    = {  0, 128, 128};
24:
25:  const RGB_INT dk_red     = {128,   0,   0};
26:  const RGB_INT dk_green   = {  0, 128,   0};
27:  const RGB_INT dk_blue    = {  0,   0, 128};
```

Appendix F - pgm.c

```
1:  #include < stdio.h >
2:  #include < stdlib.h >
3:  #include "pgm.h"
4:
5:  /**********************File I/O functions*******************************/
6:  /***********************************************************************/
7:
8:  /*Gets an ascii pgm image file, store as a color pgm.*/
9:  void getPGMfile (char filename[], PGMImage *img)
10:  {
11:    FILE *in_file;
12:    char ch;
13:    int row, col, type;
14:    int ch_int;
15:
16:    in_file = fopen(filename, "r");
17:    if (in_file == NULL)
18:    {
19:      fprintf(stderr, "Error: Unable to open file %s\n\n", filename);
20:      exit(8);
21:    }
22:
23:    printf("\nReading image file: %s\n", filename);
24:
25:    /*determine pgm image type (only type three can be used)*/
26:    ch = getc(in_file);
27:    if(ch != 'P')
28:    {
29:       printf("ERROR(1): Not valid pgm/ppm file type\n");
30:       exit(1);
31:    }
32:    ch = getc(in_file);
33:    /*convert the one digit integer currently represented as a character to
34:      an integer(48 == '0')*/
35:    type = ch - 48;
36:    if((type != 2) && (type != 3) && (type != 5) && (type != 6))
37:    {
38:       printf("ERROR(2): Not valid pgm/ppm file type\n");
39:       exit(1);
40:    }
41:
42:    while(getc(in_file) != '\n');              /* skip to end of line*/
43:
44:    while (getc(in_file) == '#')               /* skip comment lines */
45:    {
46:      while (getc(in_file) != '\n');           /* skip to end of comment line */
47:    }
48:
49:    /*there seems to be a difference between color and b/w.  This line is needed
50:      by b/w but doesn't effect color reading...*/
51:    fseek(in_file, -1, SEEK_CUR);              /* backup one character*/
52:
53:    fscanf(in_file,"%d", &((*img).width));
54:    fscanf(in_file,"%d", &((*img).height));
55:    fscanf(in_file,"%d", &((*img).maxVal));
56:
```

```
57:     printf("\n width  = %d",(*img).width);
58:     printf("\n height = %d",(*img).height);
59:     printf("\n maxVal = %d",(*img).maxVal);
60:     printf("\n");
61:
62:     if (((*img).width  > MAX) || ((*img).height  > MAX))
63:     {
64:       printf("\n\n***ERROR - image too big for current image structure***\n\n");
65:       exit(1);
66:     }
67:
68:     if(type == 2) /*uncompressed ascii file (B/W)*/
69:     {
70:        for (row=(*img).height-1; row >=0; row--)
71:          for (col=0; col< (*img).width; col++)
72:          {
73:             fscanf(in_file,"%d", &ch_int);
74:             (*img).data[row][col].red = ch_int;
75:             (*img).data[row][col].green = ch_int;
76:             (*img).data[row][col].blue = ch_int;
77:          }
78:     }
79:     else if(type == 3) /*uncompressed ascii file (color)*/
80:     {
81:        for (row=(*img).height-1; row >=0; row--)
82:          for (col=0; col< (*img).width; col++)
83:          {
84:
85:        fscanf(in_file,"%d", &ch_int);
86:        ((*img).data[row][col].red) = (unsigned char)ch_int;
87:
88:        fscanf(in_file,"%d", &ch_int);
89:        ((*img).data[row][col].green) = (unsigned char)ch_int;
90:
91:        fscanf(in_file,"%d", &ch_int);
92:        ((*img).data[row][col].blue) = (unsigned char)ch_int;
93:          }
94:     }
95:     else if(type == 5) /*compressed file (B/W)*/
96:   /*note: this type remains untested at this time...*/
97:     {
98:        while(getc(in_file) != '\n'); /*skip to end of line*/
99:
100:         for (row=(*img).height-1; row >=0; row--)
101:          for (col=0; col< (*img).width; col++)
102:          {
103:       ch = getc(in_file);
104:             (*img).data[row][col].red = ch;
105:             (*img).data[row][col].green = ch;
106:             (*img).data[row][col].blue = ch;
107:          }
108:     }
109:
110:     else if(type == 6) /*compressed file (color)*/
111:     {
112:        while(getc(in_file) != '\n'); /*skip to end of line*/
113:
114:        for (row=(*img).height-1; row >=0; row--)
```

```
115:          for (col=0; col< (*img).width; col++)
116:          {
117:             (*img).data[row][col].red = getc(in_file);
118:             (*img).data[row][col].green = getc(in_file);
119:             (*img).data[row][col].blue = getc(in_file);
120:          }
121:    }
122:
123:    fclose(in_file);
124:    printf("\nDone reading file.\n");
125:  }
126:
127:
128:  void save(PGMImage *img)
129:  {
130:     int i, j, nr, nc, k;
131:     int red, green, blue;
132:     FILE *iop;
133:
134:     nr = img- &gtheight;
135:     nc = img- &gtwidth;
136:
137:     iop = fopen("image1.pgm", "w");
138:     fprintf(iop, "P6\n");
139:     fprintf(iop, "%d %d\n", nc, nr);
140:     fprintf(iop, "255\n");
141:
142:     k = 1;
143:     for(i = nr - 1; i  >= 0; i--)
144:     {
145:        for(j = 0; j <  nc; j++)
146:        {
147:           red = img- &gtdata[i][j].red;
148:           green = img- &gtdata[i][j].green;
149:           blue = img- &gtdata[i][j].blue;
150:           if(red <  0)
151:           {
152:              printf("IMG_WRITE: Found value %d at row %d col %d\n", red, i, j);
153:              printf("           Setting red to zero\n");
154:              red = 0;
155:           }
156:           if(green <  0)
157:           {
158:              printf("IMG_WRITE: Found value %d at row %d col %d\n", green,i, j)
159:              printf("           Setting green to zero\n");
160:              green = 0;
161:           }
162:           if(blue <  0)
163:           {
164:              printf("IMG_WRITE: Found value %d at row %d col %d\n", blue, i, j)
165:              printf("           Setting green to zero\n");
166:              blue = 0;
167:           }
168:           if(red  > 255)
169:           {
170:              printf("IMG_WRITE: Found value %d at row %d col %d\n", red, i, j);
171:              printf("           Setting red to 255\n");
172:              red = 255;
```

```
173:            }
174:            if(green  > 255)
175:            {
176:               printf("IMG_WRITE: Found value %d at row %d col %d\n", green,i, j)
177:               printf("           Setting green to 255\n");
178:               green = 255;
179:            }
180:            if(blue  > 255)
181:            {
182:               printf("IMG_WRITE: Found value %d at row %d col %d\n", blue, i, j)
183:               printf("           Setting blue to 255\n");
184:               blue = 255;
185:            }
186:
187:      putc(red, iop);
188:      putc(green, iop);
189:      putc(blue, iop);
190:         }
191:      }
192:      fprintf(iop, "\n");
193:      fclose(iop);
194:   }
```

Appendix G - pgm.h

```
1:  #ifndef PGM_H
2:  #define PGM_H
3:
4:  /*max size of an image*/
5:  #define MAX 800
6:
7:  /*
8:  #define LOW_VALUE 0
9:  #define HIGH_VALUE 255
10:  */
11:
12:
13:  /*RGB color struct with integral types*/
14:  typedef struct {unsigned char red;
15:                  unsigned char green;
16:                  unsigned char blue;
17:                 }RGB_INT;
18:
19:  struct PGMstructure
20:  {
21:    int maxVal;
22:    int width;
23:    int height;
24:    RGB_INT data[MAX][MAX];
25:  };
26:
27:  typedef struct PGMstructure PGMImage;
28:
29:
30:  /***prototypes*****************************************************/
31:  /*****************************************************************/
32:
33:  void getPGMfile (char filename[], PGMImage *img);
34:  void save(PGMImage *img);
35:
36:  #endif
```

Appendix H - linked_list.c

```
1:  #include < string.h >
2:  #include "bool.h"
3:  #include "linked_list.h"
4:
5:  void InitList(list_info* list_pointers)
6:  {
7:    (*list_pointers).head = NULL;
8:    (*list_pointers).cur  = NULL;
9:    (*list_pointers).tail = NULL;
10: }
11:
12: void FreeList(list_info* list_pointers)
13: {
14:   ToFirst(list_pointers);
15:   while (!ListIsEmpty(list_pointers))
16:     Delete(list_pointers);
17: }
18:
19: bool ListIsEmpty(list_info* list_pointers)
20: {
21:   return ((*list_pointers).head == NULL ? TRUE : FALSE);
22: }
23:
24: int Length(list_info* list_pointers)
25: {
26:   list_info temp;
27:   int count = 0;
28:
29:   memcpy(&temp, list_pointers, sizeof(list_info));
30:   while(RetrieveNextNode(&temp).cur)
31:   {
32:      count++;
33:      Advance(&temp);
34:   }
35:
36:   return count;
37: }
38:
39: bool ListIsFull()
40: {
41:   return FALSE;
42:   /*
43:   struct linked_list* temp =
44:     (struct linked_list*)malloc(sizeof(struct linked_list));
45:
46:     if(temp)*/ /*if there is memory, return false*/
47:   /*{
48:     free(temp);
49:     return FALSE;
50:   }
51:   return TRUE;*/
52: }
53:
54: bool CurIsEmpty(list_info* list_pointers)
55: {
56:   return ((*list_pointers).cur == NULL ? TRUE : FALSE);
```

```
57:  }
58:
59:  struct linked_list* ToFirst(list_info* list_pointers)
60:  {
61:    (*list_pointers).cur = (*list_pointers).head;
62:
63:    return (*list_pointers).head;
64:  }
65:
66:  struct linked_list* ToEnd(list_info* list_pointers)
67:  {
68:    (*list_pointers).cur = (*list_pointers).tail;
69:
70:    return (*list_pointers).tail;
71:  }
72:
73:  bool AtFirst(list_info* list_pointers)
74:  {
75:    return ((*list_pointers).head == (*list_pointers).cur ? TRUE : FALSE);
76:  }
77:
78:  bool AtEnd(list_info* list_pointers)
79:  {
80:    return ((*list_pointers).tail == (*list_pointers).cur ? TRUE : FALSE);
81:  }
82:
83:  void Advance(list_info* list_pointers)
84:  {
85:    if (CurIsEmpty(list_pointers))
86:      NodeReferenceError();
87:    else
88:      (*list_pointers).cur = (*list_pointers).cur- &gtnext;
89:  }
90:
91:  void Retreat(list_info* list_pointers)
92:  {
93:    if(CurIsEmpty(list_pointers))
94:      NodeReferenceError();
95:    else
96:      (*list_pointers).cur = (*list_pointers).cur- &gtprev;
97:  }
98:
99:  void InsertAfter(el_t e, list_info* list_pointers)
100:  {
101:    node_pointer target = MakeNode(e);
102:
103:    if (ListIsEmpty(list_pointers))
104:    {
105:       (*list_pointers).head = target;
106:       (*list_pointers).cur  = target;
107:       (*list_pointers).tail = target;
108:    }
109:    else if (CurIsEmpty(list_pointers))
110:       NodeReferenceError();
111:    else if (AtEnd(list_pointers))
112:    {
113:      target- &gtprev = (*list_pointers).cur;
114:      (*list_pointers).cur- &gtnext = target;
```

```
115:      (*list_pointers).tail = target;  /*set new tail*/
116:    }
117:    else
118:    {
119:      target- &gtnext = (*list_pointers).cur- &gtnext;
120:      target- &gtprev = (*list_pointers).cur;
121:      (*list_pointers).cur- &gtnext- &gtprev = target;
122:      (*list_pointers).cur- &gtnext = target;
123:    }
124:  }
125:
126:  void Insert(el_t e, list_info* list_pointers)
127:  {
128:    node_pointer target = MakeNode(e);
129:
130:    if (ListIsEmpty(list_pointers))         /* create one-element list */
131:    {
132:      (*list_pointers).head = target;
133:      (*list_pointers).cur  = target;
134:      (*list_pointers).tail = target;
135:    }
136:    else if (CurIsEmpty(list_pointers))     /* empty cur, nonempty list */
137:       NodeReferenceError();
138:    else if(AtFirst(list_pointers))         /* new head node */
139:    {
140:      target- &gtnext               = (*list_pointers).cur;
141:      (*list_pointers).cur- &gtprev = target;
142:      (*list_pointers).head      = target;
143:    }
144:    else                                /* insert between two nodes */
145:    {
146:      target- &gtnext    = (*list_pointers).cur;
147:      target- &gtprev    = (*list_pointers).cur- &gtprev;
148:      (*list_pointers).cur- &gtprev- &gtnext = target;
149:      (*list_pointers).cur- &gtprev       = target;
150:    }
151:  }
152:
153:  void Delete(list_info* list_pointers)
154:  {
155:    node_pointer tmp;
156:
157:    if (CurIsEmpty(list_pointers))          /* empty list*/
158:       NodeReferenceError();
159:    else if (AtFirst(list_pointers) && AtEnd(list_pointers)) /*one node exists *
160:    {
161:        DestroyNode((*list_pointers).cur);
162:        /*set the pointers to null*/
163:        memset(list_pointers, 0, sizeof(list_info));
164:    }
165:    else if (AtFirst(list_pointers))        /* delete head node */
166:    {
167:      tmp = (*list_pointers).cur;
168:      (*list_pointers).cur = (*list_pointers).cur- &gtnext;
169:      (*list_pointers).cur- &gtprev = NULL;
170:      (*list_pointers).head = (*list_pointers).cur;
171:      DestroyNode(tmp);
172:    }
```

```
173:    else if (AtEnd(list_pointers))                    /* delete end node */
174:    {
175:      tmp = (*list_pointers).cur;
176:      (*list_pointers).cur = (*list_pointers).cur- &gtprev;
177:      (*list_pointers).cur- &gtnext = NULL;
178:      (*list_pointers).tail = (*list_pointers).cur;
179:      DestroyNode(tmp);
180:    }
181:    else                                          /* delete middle node */
182:    {
183:      tmp = (*list_pointers).cur;
184:      (*list_pointers).cur- &gtprev- &gtnext = (*list_pointers).cur- &gtnext;
185:      (*list_pointers).cur- &gtnext- &gtprev = (*list_pointers).cur- &gtprev;
186:      (*list_pointers).cur = (*list_pointers).cur- &gtnext;
187:      DestroyNode(tmp);
188:    }
189:  }
190:
191:  void StoreInfo(el_t e, list_info* list_pointers)
192:  {
193:     if(CurIsEmpty(list_pointers))
194:        NodeReferenceError();
195:     else
196:       /*There had better be enough room...*/
197:        memcpy(&(((*list_pointers).cur)- &gtdata), &e, sizeof(void*));
198:  }
199:
200:  el_t RetrieveInfo(list_info* list_pointers)
201:  {
202:    if(CurIsEmpty(list_pointers))
203:       NodeReferenceError();
204:
205:    return (((*list_pointers).cur)- &gtdata);
206:  }
207:
208:  el_t RetrievePrevInfo(list_info* list_pointers)
209:  {
210:    if (CurIsEmpty(list_pointers))
211:       NodeReferenceError();
212:    if(AtFirst(list_pointers))
213:       NodeReferenceError();
214:
215:    return (((*list_pointers).cur)- &gtprev- &gtdata);
216:  }
217:
218:  list_info RetrievePrevNode(list_info* list_pointers)
219:  {
220:    list_info temp;
221:    memcpy(&temp, list_pointers, sizeof(list_info));
222:
223:    if (CurIsEmpty(list_pointers))
224:       NodeReferenceError();
225:
226:    temp.cur = ((*list_pointers).cur)- &gtprev;
227:
228:    return temp;
229:  }
230:
```

```
231:
232:  el_t RetrieveNextInfo(list_info* list_pointers)
233:  {
234:    if (CurIsEmpty(list_pointers))
235:       NodeReferenceError();
236:    if(AtEnd(list_pointers))
237:      NodeReferenceError();
238:
239:    return (((*list_pointers).cur)- &gtnext- &gtdata);
240:  }
241:
242:  list_info RetrieveNextNode(list_info* list_pointers)
243:  {
244:    list_info temp;
245:    memcpy(&temp, list_pointers, sizeof(list_info));
246:
247:    if (CurIsEmpty(list_pointers))
248:       NodeReferenceError();
249:
250:    temp.cur = ((*list_pointers).cur)- &gtnext;
251:
252:    return temp;
253:  }
254:
255:  node_pointer MakeNode(el_t e)
256:  {
257:    node_pointer p;
258:
259:    if(ListIsFull())
260:    {
261:      printf("List is Full.  Exiting.\n");
262:      exit(1);
263:    }
264:
265:
266:
267:  #ifdef __cplusplus
268:    p = new struct linked_list;
269:  #else
270:    p = (struct linked_list*) malloc(sizeof(struct linked_list));
271:  #endif
272:
273:    if (!p)
274:    {
275:      printf("Error making node\n");
276:      exit(1);
277:    }
278:    else
279:    {
280:      memcpy(&(p- &gtdata), &e, sizeof(el_t));
281:      p- &gtnext = NULL;
282:      p- &gtprev = NULL;
283:    }
284:
285:    return p;
286:  }
287:
288:  void DestroyNode(node_pointer p)
```

```
289:  {
290:  #ifdef __cplusplus
291:    delete p;
292:  #else
293:    free(p);
294:  #endif
295:  }
296:
297:  void NodeReferenceError()
298:  {
299:    int *temp = NULL;
300:    printf("ERROR: No current node or an improper node\n");
301:  printf("%d\n", *temp);
302:    exit(1);
303:  }
304:
305:
306:
```

Appendix I - linked_list.h

```
1:  #ifndef LINKED_LIST_H
2:  #define LINKED_LIST_H
3:
4:  #include < stdlib.h >
5:  #include < stdio.h >
6:
7:  #include "bool.h"
8:  #include "common.h"
9:
10:  /*#include "research.h"*/
11:
12:  typedef struct
13:  {
14:    coord location; /*absolute pixel location for starting point*/
15:    int code;
16:  }chain;
17:
18:  typedef chain el_t;
19:
20:  typedef struct linked_list* node_pointer;
21:  typedef struct linked_list
22:  {
23:    node_pointer prev;
24:    el_t  data;
25:    node_pointer next;
26:  }list;
27:
28:  typedef struct
29:  {
30:    list* head;
31:    list* tail;
32:    list* cur;
33:  }list_info;
34:
35:  /*********prototypes*****************************************************/
36:
37:
38:  void InitList(list_info* list_pointers);
39:
40:  void FreeList(list_info* list_pointers);
41:
42:  bool ListIsEmpty(list_info* list_pointers);
43:
44:  int Length(list_info* list_pointers);
45:
46:  bool ListIsFull();
47:
48:  bool CurIsEmpty(list_info* list_pointers);
49:
50:  struct linked_list* ToFirst(list_info* list_pointers);
51:
52:  struct linked_list* ToEnd(list_info* list_pointers);
53:
54:  bool AtFirst(list_info* list_pointers);
55:
56:  bool AtEnd(list_info* list_pointers);
```

```
57:
58:  void Advance(list_info* list_pointers);
59:
60:  void Retreat(list_info* list_pointers);
61:
62:  void InsertAfter(el_t e, list_info* list_pointers);
63:
64:  void Insert(el_t e, list_info* list_pointers);
65:
66:  void Delete(list_info* list_pointers);
67:
68:  void StoreInfo(el_t e, list_info* list_pointers);
69:
70:  el_t RetrieveInfo(list_info* list_pointers);
71:
72:  el_t RetrievePrevInfo(list_info* list_pointers);
73:
74:  list_info RetrievePrevNode(list_info* list_pointers);
75:
76:  list_info RetrieveNextNode(list_info* list_pointers);
77:
78:  el_t RetrieveNextInfo(list_info* list_pointers);
79:
80:  node_pointer MakeNode(el_t e);
81:
82:  void DestroyNode(node_pointer p);
83:
84:  void NodeReferenceError();
85:
86:  #endif
```

Appendix J - chaincode.c

```
1:  #include "linked_list.h"
2:  #include "chaincode.h"
3:  #include "ttt_vision.h"
4:  #include "pgm.h"
5:  #include < string.h >
6:
7:  /*******Evil globals*********************************/
8:
9:  int cur_threash_val; /*for speed optimization of chain code recursion*/
10:  int back_ground;
11:
12:  const RGB_INT chaincode_color = {255, 255, 255};
13:
14:  extern bool display_verbose;
15:
16:  /******CHAIN CODE*************************************************
17:  ****************************************************************/
18:
19:  /*returns TRUE if the chain code loops around itself, otherwise false*/
20:  int checkCode(list_info* list_pointers, int x_current, int y_current,
21:            int x_check, int y_check, int dir)
22:  {
23:     list_info temp;
24:
25:     ToFirst(list_pointers);
26:
27:     while(!AtEnd(list_pointers))
28:     {
29:        /*needed to check two nodes ahead in following if*/
30:        temp = RetrieveNextNode(list_pointers);
31:
32:        /*check two consecutive points*/
33:        if((RetrieveInfo(list_pointers).location.x == x_current) &&
34:           (RetrieveInfo(list_pointers).location.y == y_current) &&
35:           (RetrieveInfo(list_pointers).code == dir) &&
36:           (RetrieveNextInfo(list_pointers).location.x == x_check) &&
37:           (RetrieveNextInfo(list_pointers).location.y == y_check) &&
38:      (RetrieveNextNode(&temp).cur)) /*don't count the new node already*/
39:        {
40:           return TRUE; /*The chain code end looped around*/
41:        }
42:        else
43:     /*currentNode = RetrieveNextNode(list_pointers);*/
44:          Advance(list_pointers);
45:     }
46:     return FALSE;
47:  }
48:
49:  /*
50:  Preconditions:
51:  1st parameter: integer value containing the current x coordinate.
52:  2nd parameter: integer value containing the current y coordinate.
53:  3rd parameter: pointer to the pgm image being used.
54:  Postconditions:  return true if pixel is deemed boarder of object, false
55:     otherwise.*/
56:  int checkThings(int x_check, int y_check, PGMImage *img)
```

```
57:  {
58:
59:     /*check for being in bounds*/
60:     if((x_check <  0) || (y_check <  0) || (x_check  >= (*img).width) ||
61:        (y_check  >= (*img).height))
62:        return FALSE;
63:
64:     /*tests if the next pixel is greater than the threashing value.  This
65:        value (cur_threash_val) should be set to the return value of
66:        img_pxl_avg().  Also, checks if the pixel is considered in the background
67:        or forground agains the value in back_ground.  If so, return true to
68:        indicate create new node.*/
69:     if((rgb_avg((*img).data[y_check][x_check])  > cur_threash_val)
70:        && (back_ground <  0))
71:        return TRUE;
72:
73:     /*same thing as above, just for things less than, rathur than greater than*/
74:     if((rgb_avg((*img).data[y_check][x_check]) <  cur_threash_val)
75:       && (back_ground  > 0))
76:        return TRUE;
77:
78:     return FALSE;
79:  }
80:
81:  chain neighborhood_point(int x_coord, int y_coord, int dir)
82:  {
83:     chain temp = {{x_coord, y_coord}, dir};
84:
85:     if(dir == EAST)
86:        temp.location.x++;
87:     else if(dir == NORTHEAST)
88:     {
89:        temp.location.x++;
90:        temp.location.y++;
91:     }
92:     else if(dir == NORTH)
93:        temp.location.y++;
94:     else if(dir == NORTHWEST)
95:     {
96:        temp.location.x--;
97:        temp.location.y++;
98:     }
99:     else if(dir == WEST)
100:        temp.location.x--;
101:     else if(dir == SOUTHWEST)
102:     {
103:        temp.location.x--;
104:        temp.location.y--;
105:     }
106:     else if(dir == SOUTH)
107:        temp.location.y--;
108:     else if(dir == SOUTHEAST)
109:     {
110:        temp.location.x++;
111:        temp.location.y--;
112:     }
113:
114:     return temp;
```

```
115: }
116:
117: /*determines the chain code for a starting pixel*/
118: /*this chain code uses 8 connectivity*/
119: /*Note: There are many different coordinate systems at work in this function.*
120: /*The (*img).data[][] are column major and have are therefore [y][x] or [j][i]
121:   This is also asuming (0,0) is bottom left, where the other code assumes
122:   top left is (0, 0).  If it looks strange, look harder!!!*/
123: /*Preconditions:
124: 1st parameter: pointer to the pgm image being used.
125: 2nd parameter: integer value containing the current x coordinate.
126: 3rd parameter: integer value containing the current y coordinate.
127: 4th parameter: integer value containing the current direction code.
128: 5th parameter: pointer to the last node of the chain code.
129: 6th parameter: pointer to the first node of the chain code.*/
130: /*This function assumes the (0, 0) point is in the lower left*/
131:
132: int chaincode(PGMImage *img, int x, int y, int dir, list_info* list_pointers)
133: {
134:    int i; /*loop counting*/
135:    int next /*next direction to check*/;
136:    chain next_point_to_check; /*next_point_to_check What else?*/
137:
138:    next = (dir + 5) % 8; /*better initail choice?*/
139:
140: /*printf("next: %d  x: %d  y: %d\n", next, x, y);*/
141:    for(i = 0; i <  8; i++)
142:    {
143:       /*determine the next point to check*/
144:       next_point_to_check = neighborhood_point(x, y, next);
145:
146:       if(checkThings(next_point_to_check.location.x,
147:                next_point_to_check.location.y, img))
148:       {
149: /*setCPixel(next_point_to_check.location.x, next_point_to_check.location.y,
150:   chaincode_color);*/
151:
152:    /*create the new node to go last in the chain*/
153:          InsertAfter(next_point_to_check, list_pointers);
154:          /*Advance(list_pointers);*/
155:
156:    /*if the next chaincode function in the recursion or the current state
157:          is the final state of the the chain code; recursively returns DONE
158:          back through all the levels to stop the chain code.*/
159:    if(checkCode(list_pointers, x, y, next_point_to_check.location.x,
160:               next_point_to_check.location.y, dir))
161:    {
162:       return NONE;
163:    }
164:
165:    if(NONE == chaincode(img, next_point_to_check.location.x,
166:                         next_point_to_check.location.y, next,
167:                         list_pointers))
168:       return NONE;
169:       }
170:
171:       /*set next for next iteration*/
172:       next = (next + 1) % 8;
```

```
173:     }
174:     return FALSE;
175:  }
176:
177:  /*returns true if the point is already in one of the chains, false otherwise*/
178:  int alreadyFound(int initThreashFlag, int i, int j, list_info* found)
179:  {
180:     int k; /*loop counting*/
181:     list_info temp; /*temporary node pointer*/
182:
183:     /*while there are codes to check...*/
184:     for(k = 0; (k <  MAX_NUMBER_OF_CHAINS) && (found[k].head); k++)
185:     {
186:        memcpy(&temp, &found[k], sizeof(list_info));
187:        while(!AtEnd(&temp)) /*...check them.*/
188:        {
189:     /*if point has already been found*/
190:     if((RetrieveInfo(&temp).location.x == i) &&
191:        (RetrieveInfo(&temp).location.y == j))
192:     {
193:        return TRUE;
194:        break;
195:     }
196:     Advance(&temp);
197:        }
198:        /*Don't forget the last point!!!*/
199:        if((RetrieveInfo(&temp).location.x == i) &&
200:     (RetrieveInfo(&temp).location.y == j))
201:        {
202:    return TRUE;
203:    break;
204:        }
205:     }
206:
207:     return FALSE;
208:  }
209:
210:  /*saves a chain code to file.  Was usefull during debuging now is just
211:    a cool thing to leave in*/
212:  /*1st: pointer to beginning of a chaincode
213:  2nd: the index of the chain code for destiqushing the 'chain' files*/
214:  void saveChainCode(list_info save_chain, int count)
215:  {
216:     char filename[12];
217:     FILE* outfile;
218:     sprintf(filename, "chain%d", count);
219:     outfile = fopen(filename, "w");   /*output file for chaincode*/
220:  /*printf("Writing chain code to file %s.\n", filename);*/
221:
222:     ToFirst(&save_chain);
223:     while(!AtEnd(&save_chain))
224:     {
225:
226:        fprintf(outfile, "%d  %d  %d\n", RetrieveInfo(&save_chain).location.x,
227:          RetrieveInfo(&save_chain).location.y,
228:          RetrieveInfo(&save_chain).code);
229:        Advance(&save_chain);
230:     }
```

```
231:     fclose(outfile);
232:  }
233:
234:
235:
236:  /*frees the memory that the chain code(s) are using*/
237:  /*aogccp = address of global chain code pointer*/
238:  /*upon success the pointer whose address is passed in is set to NULL.*/
239:  void free_chaincodes(list_info** aogccp)
240:  {
241:     int i; /*loop counting*/
242:
243:     /*for each chaincode, free all of the nodes.*/
244:     for(i = 0; aogccp && (*aogccp) && (*aogccp)[i].cur && i <  MAX_CHAINS; i++)
245:     {
246:       FreeList(&(*aogccp)[i]);
247:     }
248:
249:     free(*aogccp); /*free the array of pointers to chaincodes*/
250:
251:     *aogccp = NULL; /*set global to NULL indicating there isn't a chaincode*/
252:  }
253:
254:
255:  list_info* showChain(PGMImage *original, list_info** original_chains)
256:  {
257:     int i, j;  /*loop counting*/
258:     int count = 0; /*array index holder*/
259:     int initThreashFlag = img_pxl_avg(original);
260:     list_info *the_lists;
261:
262:     if(display_verbose)
263:        printf("Finding chain code(s).  ");
264:
265:     /*If the_lists point to an array of list_infos (chaincodes), then free
266:       the memory before allocating more.  If the_lists is NULL, then there
267:       aren't allocated chaincodes and nothing will change.*/
268:     free_chaincodes(original_chains);
269:
270:     /*allocate an array of pointers to chainCodes, then initalize their
271:       values to NULL*/
272:     the_lists = (list_info*)
273:       malloc(MAX_NUMBER_OF_CHAINS * sizeof(list_info));
274:     memset(the_lists, 0, MAX_NUMBER_OF_CHAINS * sizeof(list_info));
275:
276:     cur_threash_val = initThreashFlag; /*set global variables*/
277:     back_ground = background(initThreashFlag, original);
278:
279:     /*search image until a pixel is found with threasholded value of the
280:     object.  i & j will then contain the starting coordinate.*/
281:     for(i = 0; i <  (*original).width; i++) /*x direction*/
282:     {
283:        for(j = (*original).height - 1; j  >= 0; j--) /*y direction*/
284:        {
285:           /*skip to the next iteration if pixel isn't "good" enough*/
286:           if(!checkThings(i, j, original))
287:              continue;
288:
```

```
289:            /*skip to the next iteration, which will be at the top of the next
290:              collumn.  This is true when the pixel has already been found in
291:       one of the chain codes.*/
292:            else if(alreadyFound(initThreashFlag, i, j, the_lists))
293:         break;
294:
295:            else
296:            {
297:  /*printf("chaincode: %d\n", count);*/
298:               /*printf("The starting coordinate is (x, y): (%d, %d)\n", i, j);*/
299:
300:               /*Initalize the lists*/
301:         InitList(&the_lists[count]);
302:
303:               chaincode(original, i, j, SOUTHEAST, &the_lists[count]);
304:
305:         if(the_lists[count].cur != NULL)/*avoid writing zero length chain*/
306:         {
307:            /*send the chaincode data to file, usefull for debuggin.*/
308:            saveChainCode(the_lists[count], count);
309:                    count++; /*advance the beginning counter*/
310:         }
311:
312:         /*force end of loops, leaving code when finished looping
313:           to still execute, since the number of chains is filled to max.*/
314:         if(count  >= MAX_NUMBER_OF_CHAINS)
315:            i = (*original).width;
316:
317:               break; /*check the next collumn*/
318:            }
319:         }
320:      }
321:      if(display_verbose)
322:         printf("%d were found.\n", count);
323:
324:      *original_chains = the_lists;
325:      return the_lists;
326:  }
```

Appendix K - chaincode.h

```
1:  #ifndef CHAINCODE_H
2:  #define CHAINCODE_H
3:
4:  #include "linked_list.h"
5:  /*#include "research.h"*/
6:  #include "bool.h"
7:  #include "pgm.h"
8:
9:  /*number of chain codes stored, */
10:  #define MAX_NUMBER_OF_CHAINS 10
11:  #define MAX_CHAINS MAX_NUMBER_OF_CHAINS
12:
13:
14:  /*These constant values are the chain code directions*/
15:  #define NONE      -1
16:  #define EAST      0
17:  #define NORTHEAST 1
18:  #define NORTH     2
19:  #define NORTHWEST 3
20:  #define WEST      4
21:  #define SOUTHWEST 5
22:  #define SOUTH     6
23:  #define SOUTHEAST 7
24:
25:  /*old typedefs*/
26:  /*typedef struct chainCode* chain_t;
27:  typedef struct chainCode
28:  {
29:    chain_t prev;
30:    int code;
31:    coord location;*/ /*absolute pixel location for starting point*/
32:  /*  chain_t next;
33:      }chainCode;*/ /*This struct can be refered to by: struct chainCode or chain
34:
35:
36:
37:
38:  /**********prototypes**************************************************/
39:
40:  int checkCode(list_info* list_pointers, int x_current, int y_current,
41:            int x_check, int y_check, int dir);
42:
43:  int checkThings(int x_check, int y_check, PGMImage *img);
44:
45:  chain neighborhood_point(int x_coord, int y_coord, int dir);
46:
47:  int chaincode(PGMImage *img, int x, int y, int dir, list_info *list_pointers);
48:
49:  int alreadyFound(int initThreashFlag, int i, int j, list_info* found);
50:
51:  void saveChainCode(list_info saveChain, int count);
52:
53:  void free_chaincodes(list_info** aogccp);
54:
55:  list_info* showChain(PGMImage *original, list_info **the_lists);
56:
```

```
57:  #endif
```

Appendix L - Makefile

```
1:  # $Id: Makefile,v 1.2 2000/04/08 03:45:13 mzalokar Exp $
2:  #
3:  # Makefile for the sample SIE device.  Just define the 4 variables below and
4:  # include ../Makefile.devicess.common.  Of course, you just write your own
5:  # more complicated makefile, but this is supposed to make it easier for those
6:  # of you who aren't yet Make gurus (shame on you).
7:
8:  DEVICE_NAME = ttt_vision                    # the name of your device
9:  DEVICE_SRC  = ttt_vision.c chaincode.c linked_list.c pgm.c
10:  DEVICE_FLAGS       = -ansi -fstrict-aliasing $(THREAD_FLAGS)
11:  DEVICE_LIBS        = -L../../lib -lsie_utils $(XLIBS) $(GLUTLIBS) $(THREAD_LIBS
12:               -lposix4
13:
14:  # include the device-common stuff
15:
16:  include ../Makefile.devices.common
17:
```

Appendix M - Ttt_Vision.C

```
1:  #include "Ttt_Vision.H"
2:
3:
4:  Ttt_Vision::Ttt_Vision (int sock_number, int32 device_id, int frame_height,
5:                          int frame_width)
6:  {
7:    sock = sock_number;
8:    id = device_id;
9:
10:    height = frame_height;
11:    width = frame_width;
12:
13:    // what we're doing
14:    int8 val = AGENT_SEND_TO_DEVICE;
15:    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
16:
17:    // to which device
18:    int32 val32 = id;
19:    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
20:
21:    // how long the message is
22:    val32 = sizeof (int8) + sizeof(int32) * 2;
23:    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
24:
25:    // the message (it's about time)
26:    val = TTT_VISION_SET_RESOLUTION;
27:    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
28:
29:    // the next thing going out is the resolution
30:    shelley_sockets_write (sock, (char*) &width, sizeof (int32));
31:    shelley_sockets_write (sock, (char*) &height, sizeof (int32));
32:
33:    shelley_sockets_read (sock, (char*) &val, sizeof (int8));
34:  }
35:
36:
37:
38:  void Ttt_Vision::find_ttt(unsigned char* data, unsigned char game[3][3])
39:  {
40:    unsigned char* i;
41:
42:    /*********************************/
43:    // what we're doing
44:    int8 val = AGENT_SEND_TO_DEVICE;
45:    shelley_sockets_write (sock, (char*) &val, sizeof (int8));
46:
47:    // to which device
48:    int32 val32 = id;
49:    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
50:
51:    // how long the message is
52:    val32 = sizeof (int8) + (height * width * 3);
53:    shelley_sockets_write (sock, (char*) &val32, sizeof (int32));
54:
55:    // the message (it's about time)
56:    val = TTT_VISION_FIND_TTT;
```

```
57:     shelley_sockets_write (sock, (char*) &val, sizeof (int8));
58:     shelley_sockets_write (sock, (char*)data, val32 - 1);
59:
60:     // the next thing coming back is the responce
61:     shelley_sockets_read (sock, (char*)game, sizeof (unsigned char) * 9);
62:   }
63:
```

Appendix N - Ttt_Vision.H

```
1:  #ifndef __TTT_VISION_WAPPER__
2:  #define __TTT_VISION_WAPPER__
3:
4:  #include "sie_protocol.h"
5:  #include "device_type_id_number.h"
6:  #include "ttt_vision_protocol.h"
7:  #include "shelley_sockets.h"
8:
9:  class Ttt_Vision
10:  {
11:  private:
12:    int sock;
13:    int32 id;
14:    int width, height;
15:
16:  public:
17:    Ttt_Vision (int sock_number, int32 device_id, int frame_height,
18:           int frame_width);
19:    ~Ttt_Vision () {};
20:
21:    void find_ttt(unsigned char* data, unsigned char game[3][3]);
22:  };
23:
24:  #endif
25:
```

Appendix O - Makefile

```
1:
2:
3:  CC = g++
4:
5:  CFLAGS = -g -Wall
6:  COMMON_INCLUDE = ../common_include
7:
8:  all: Frame_Grabber.o Ttt_Vision.o Ttt_Engine.o Tt_Arms.o
9:
10:  Frame_Grabber.o: Frame_Grabber.C $(COMMON_INCLUDE)/Frame_Grabber.H
11:     $(CC) $(CFLAGS) -I$(COMMON_INCLUDE) Frame_Grabber.C -c
12:
13:  Ttt_Vision.o: Ttt_Vision.C $(COMMON_INCLUDE)/Ttt_Vision.H
14:     $(CC) $(CFLAGS) -I$(COMMON_INCLUDE) Ttt_Vision.C -c
15:
16:  Ttt_Engine.o: Ttt_Engine.C $(COMMON_INCLUDE)/Ttt_Engine.H
17:     $(CC) $(CFLAGS) -I$(COMMON_INCLUDE) Ttt_Engine.C -c
18:
19:  Tt_Arms.o: Tt_Arms.C $(COMMON_INCLUDE)/Tt_Arms.H
20:     $(CC) $(CFLAGS) -I$(COMMON_INCLUDE) Tt_Arms.C -c
21:  clean:
22:     rm -rf *.o *~ core
```

References:

Buttlar, Dick, Farrell, Jacqueline Proulx, Farrell and Nichols, Bradford. Pthreads Programming. Beijing: O'Reilly. 1996.

Curry, David A. Unix System Programming for SVR4. Cambridge: O'Reilly. 1996.

Davis, Tom, Neider, Jackie and Woo, Mason. OpenGL Programming Guide. 2nd ed. Reading: Addison-Wesley. 1997.

Dreyfus, Hubert L. What Computers Still Can't Do. London: MIT. 1992.

Freeman, H. On the Encoding of Arbitrary Geometric Configuration. IRE Transactions on Electronic Computers. 1961

Frazier, Chris and Kempf, Renate Editors. OpenGL Reference Manual. 2nd ed. Reading: Addision-Wesley. 1997.

Glinka, Rob and Schroeder, Becky. `ttt_engine`. C++ Source Code. Illinois Wesleyan U. 2000.

Materick, Craig. Mapping Robotics Movement to a Three-Dimensional Coordinate System. Manuscript submitted for research honors, Illinois Wesleyan University. 1997.

Moravec, H. P. Towards Automatic Visual Obstacle Avoidance. In proceedings of the 5th International Joint Conference on Artificial Intelligence. Carnegie-Mellon U. 1977.

Kilgard, Mark J. OpenGL Programming for the X Window System. Reading: Addison-Wesley. 1996.

Ritger, Andy. Designing an Integrated Environment for Artificial Intelligence. Manuscript submitted for research honors, Illinois Wesleyan U. 1999.

Shufelt, Jeff. Back Propogation Neural Network. C Source Code. Carnegie Mellon U. 1994.

Sonka, Milan, Hlavac, Vaclav and Boyle, Roger. Image Processing, Analysis and Machine Vision. 2nd ed. Pacific Grove: International Thomson Publishing Inc., 1999.

Stewart, Chris and Weaver, Matt. `ttt_engine`. C++ Source code. Illinois Wesleyan U. 1997.

Sun Microsystems. `rtvc display`. C Source Code. 1994.

Sun Microsystems. Sun Video User's Guide. Mountain View, CA. 1994.

Sun Microsystems. XIL Reference Manual. Mountain View, Ca. 1997.

SunSoft. XIL Programmer's Guide. Mountain View, Ca. 1997.

Acknowledgments:

This research project would not have happened if it were not for the Shelley Project. I would like to say thank you to Rob Glinka and Dr. Shapiro for helping this project get to where it is today. Even though he has graduated Andy Ritger made invaluable improvements to SIE for this project. Special thanks to Chris Stewart, Matt Weaver, Craig Materick, Andy Ritger and Darrin Thomason who's Shelley Project work three years ago made such an indelible impression on me.

*Dr. Lon Shapiro is the faculty advisor for this research project.