A Simple Robot Pool Player

by

Wesley H. Huang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1991

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 17, 1991

Certified by .... [illegible]
Christopher G. Atkeson
Associate Professor
Thesis Supervisor

Accepted by ....
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

A Simple Robot Pool Player

by

Wesley H. Huang

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1991, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Electrical Engineering

Abstract

The game of pool can be studied at many levels. One can apply learning algorithms to the problem of sinking a single ball, controlling the position of the cue ball after a shot, or finding an effective game strategy. This thesis presents the design, construction, and testing of a simple robot pool player. It is limited to playing a modified game of pool because of simplifications in its design. In order to determine the usefulness of this robot pool player as a tool for doing learning research on pool, it has been tested for accuracy and resolution.

Thesis Supervisor: Christopher G. Atkeson
Title: Associate Professor

Acknowledgments

Many thanks to Gerrie van Zyl for suggestions and advice in both the design and construction phases of the robot pool player.

Thanks to Andrew Moore for working with me on parts of this project.

And thanks to Chris Atkeson for circus trips, support, and guidance over the course of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The game of pool offers researchers in artificial intelligence and robotics a small wealth of opportunities to explore learning and control. Pool can be studied on many different levels. One can study how a robotic pool player might simply sink a single ball. Adding bounces off the bumpers would add an additional degree of difficulty. Adding spin would open a whole new dimension to sinking a ball. Beyond the level of sinking a single ball, there are also various game-oriented issues that can be addressed, such as deciding which ball to sink and where to leave the cue ball for the opponent.

This thesis describes the design, construction, and testing of a simple robot pool player. This system consists of two parts: the manipulator and the sensors.

A general purpose manipulator for pool would be a very complex undertaking. One might want to position it anywhere on the table or at any orientation with respect to that point. This would require up to 5 degrees of freedom. The design would be further complicated by the fact that sometimes the cue must be removed from the table quickly in order to get out of the way of a shot.

The simple manipulator designed and built in this thesis project has only two degrees of freedom. It is limited to shooting from a fixed position. While this disallows the ability to play a normal game of pool, it greatly simplifies the design and construction process. It is still useful because it is possible to study the basic elements of playing pool — how the sensors can work with the manipulator to make a shot — without using a general purpose manipulator. The details of the design and construction of this manipulator are discussed in chapter 2.

The manipulator alone is not of much use; the robot must somehow sense its world and the effect of its actions upon the world. Vision is the easiest and perhaps the most obvious way of sensing the pool table. An overhead camera looks down on the pool table and can detect and track the locations of both balls. Chapter 3 discusses how the vision hardware is used to provide this sensory data.

There are many variables in the system as a whole that can change between shots. The manipulator will not deliver *exactly* the same shot every time, and vision algorithms may not return exactly the same data every time. Some noise of this sort is, of course, expected. In Chapter 4, I discuss some of the possible sources of this noise and tests done to determine the accuracy of the system.

Finally, some conclusions are presented in chapter 5 along with some suggestions for future work in this direction.

Chapter 2

Design and Construction

Many ideas were generated for possible designs for actually hitting the cue ball. Some of the considerations that led to the choice of the design implemented are discussed in section 2.1. The drawings for the robot appear in appendix A and are discussed in section 2.2.

I built the robot for this project in the Artificial Intelligence laboratory machine shop. It consists of a spring loaded 'cue stick' (not unlike a pinball plunger) mounted so that it may rotate. The cue ball is placed at the center point, and the cue actuator pivots about the center of the ball in order to make shots from different angles.

Because we are limited to making shots from a fixed point and because of the design of the rotating mount, a miniature pool table has been modified by removing the bumper from one of the ends. Thus, this robot will play its 'game' of pool on just one half of the table.

The cue actuator and its mechanical components are not the only elements of this system. Figure 2-1 on page 12 shows how the mechanical components fit it with the electromechanical, electrical, and computer elements of this system. These aspects are discussed in section 2.3.

2.1 Design Considerations

For the initial efforts at this problem, a minature pool table is used; it has approximately the same characteristics as a regular sized pool table and allows the robot to be scaled down in both size and power.

Many ideas for the design of a pool cue manipulator were considered. Some ideas were very similar to how humans handle a cue stick; others were less similar. In the end, I decided to use a spring loaded cue stick. This was the simplest design for a cue actuator. It allows fairly easy construction, variation in the speed of a shot, and the potential for good repeatability.

A rotating cue actuator, as it has been constructed here, does restrict the robot pool player to playing some modified game of pool, as it can only shoot from a fixed point. But again, this design is simpler to construct, and for an initial effort, it is sufficient to start studying some of the aspects of playing pool.

2.2 Structural Components

The robot pool player can be divided into two parts: the cue actuator, which is responsible for actually hitting the cue ball, and the cue positioner, which allows the cue actuator to pivot about the cue ball. The mechanical drawings for the robot pool player appear in appendix A. See the assembly diagrams for an overall view of the robot.

The cue actuator is contained within a rectangular 'frame'. In the front is a linear bearing through which the cue stick passes. The cue stick is not of the same type used by human players, but rather a steel shaft with an ordinary cue tip on the end.

The spring loaded cue stick is drawn back by a mechanism under computer control. A motor turns a ball screw which in turn causes a platform to move back and forth along the length of the frame. There is a catch on this platform which hooks on to the end of the cue stick. When a shot is made, the platform moves to the front of the frame until it latches onto the end of the cue stick. Then the platform is moved back

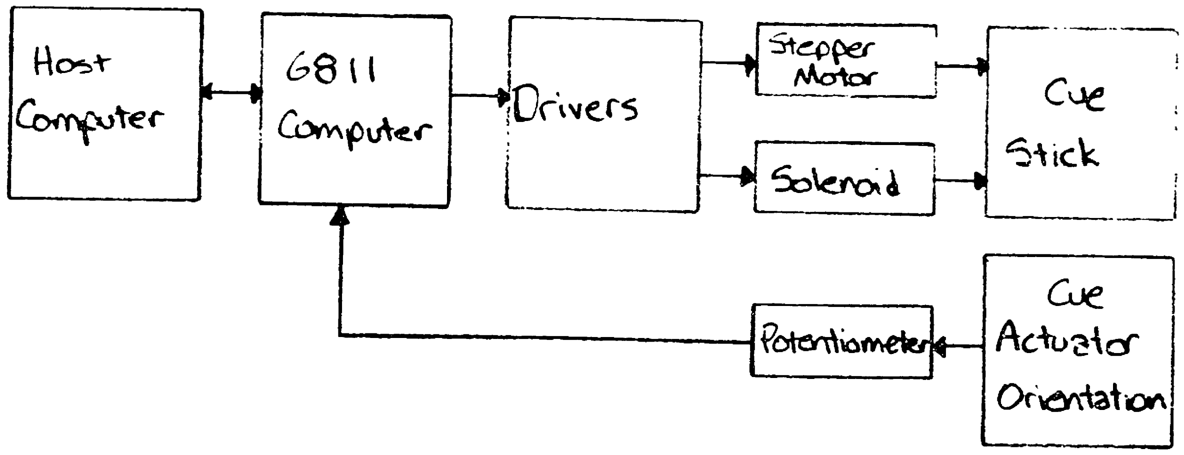


Figure 2-1: Block diagram of the robot pool player

a distance proportional to the desired speed of the shot. A solenoid on the platform releases the cue stick.

The cue positioner consists of two bearings mounted vertically under the pool table. The cue ball is placed right above the bearing so that the cue actuator can rotate about it. The cue actuator is aligned so that the end of the cue just reaches the cue ball.

An arm is attached to the bearing so that it can swing back and forth. The cue actuator is mounted on a bracket fastened to the arm. The bracket was designed so that we could vary the height and angle of the cue actuator, which may be useful should we experiment with the effects of spin.

2.3 Electromechanical components

Although the design and fabrication of all the mechanical components of this robot took considerable effort and time, there are still the electromechanical, electrical, and computer components of the system to consider. Figure 2-1 shows a block diagram of the entire system, from the cue to the host computer.

2.3.1 Stepper motors

The motor used to draw back the cue stick is a stepper motor. It rotates a fixed increment every time the coils of the motor are energized in the proper sequence. It has the advantage of offering reasonably precise positioning using only open loop control.

The driver serves to switch the coils of the stepper motors on and off in the proper sequence for the motor to take a step. For each pulse received on the step input, the driver will sequence the motor coils so that the motor will take a single step. The wiring diagram for the drivers appears in appendix B.

One disadvantage of using stepper motors is that there is no feedback in the case when the motor does not complete a step increment. If the load on the motor is too great, the motor will not be able to take the commanded step before the next step arrives. It is possible for the motor to skip steps in this manner if the motor is not large enough for the load and the commanded velocity. This issue is discussed further in section 4.1.

2.3.2 Solenoid

The solenoid used to release the cue stick is a 24 volt DC solenoid. Its 'driver' consists of an inverter, an optoisolator, and a relay. The schematic appears in appendix B. The solenoid, like the stepper motors, is actuated through parallel output ports of the 6811 processor.

2.3.3 Potentiometer

The angular position of the cue manipulator with respect to the cue ball is measured by a potentiometer attached to the shaft directly under the cue ball. The potentiometer serves as a voltage divider to divide the 5 volts supplied by the 6811 computer down to a voltage proportional to the angular position of the manipulator.

2.4 6811 computer

A 6811 based computer is used between the host computer and the drivers to solve a communication problem. The easiest way to communicate with a Sun, the host computer used in this project, is through a serial line. The drivers require a parallel output port. The use of the 6811 computer also makes it possible to use any host computer that has the capability to communicate on a serial line. The 6811 has built-in analog to digital converters (8 bits) which make it easy to convert the voltage from the potentiometer into a number that can be used to determine the orientation of the cue actuator.

The 6811 computer used here has a FORTH interpreter stored in ROM. It takes some of the computing load off the host computer. The host computer gives positioning commands to the 6811 which computes the appropriate pulse stream to send to the drivers. Section C.1 shows the code that runs on this computer.

Chapter 3

Sensory Input: Vision

Feedback to the robot pool player is provided by a vision system. A camera mounted above the pool table is able to see the entire half-table upon which the robot pool player plays. Data from the vision system is used to determine the position of the ball on the table as well as track the ball during a shot.

The vision system used for this project is the Datacube image processing system. The vision algorithms are discussed in detail in section 3.3, following some preliminary discussion on the Datacube hardware and video camera in sections 3.1 and 3.2 respectively.

3.1 The Datacube

The Datacube is a pipelined image processing system. It consists of a set of specialized boards that live on the VME bus. Each board has a particular function which it performs on the image stream. One board digitizes incoming video, another board serves to store images, and yet another can perform thresholding or histogram operations. These boards are connected to one another by special connectors which plug into the back of a board. In this manner, a custom configuration can be set up for any particular application.

More commonly, a standard configuration will be established, and each program will route the image stream accordingly. This is made possible by the multiple outputs and multiplexing circuitry available on most of these boards.

3.2 The Video Camera

Another very important aspect of the vision system is the video camera. There are many aspects of the video camera that must be taken into consideration; the two most important are the aspect ratio of the camera and the shutter speed.

The aspect ratio of the camera is a measure of the distortion that results from having a rectangular frame which has the same horizontal and vertical dimensions in pixels. This means that the pixels are actually rectangular, so if you examine a spherical ball, it will appear elliptical. Generally, video monitors also have rectangular pixels, so the effect is not noticed there, but in order to accurately measure angles and compare velocities, distances in pixels must be scaled appropriately.

The scaling of the measurements from the vision system is not done until the data is analyzed. This is discussed in section 4.2 of chapter 4.

The shutter speed of the camera may also affect the quality of the data. A normal video camera averages the image over 1/30 second. If an object such as a pool ball is moving across the frame during that time, it will become smeared. Instead of being circular, it will be an elongated oval. The cameras used for this project have the capacity of using a high speed shutter (1 ms.) which will freeze the ball much more effectively.

3.3 Vision Algorithms

The most commonly used feature of the Datacube system for this project is its ability to do feature extraction in real time. We are working with light coloured balls on a dark green table. This lends itself to a simple thresholding operation to pick out the brightest pixels in the frame; the feature extraction of the datacube will record the coordinate pairs of these pixels. The host computer can read this list of pixels and simply calculate a centroid to determine the ball's position.

I implemented this method on the datacube using two boards. The first board, the DigiMax, digitizes the incoming video signal. The second board, the FeatureMax, is capable of performing a comparison on each pixel in real time. If the comparison is true, then the x and y coordinates of that pixel are stored in a table (the feature table).

In order to achieve real time tracking of the balls, the FeatureMax looks at one field of the video signal. During the second field of the frame, the host computer reads the data from the feature table and performs centroid operations and/or other computations.

The tracking program used to measure shots appears in section C.2.1. It makes use of library routines which I wrote in order to take care of the lower level datacube functions. These library routines appear in appendix C.2.2. This program tracks a single ball for 5 seconds or until it stops moving. It reports every position at which the ball was seen, as well as the number of pixels in the image that belonged to the ball in that frame.

The tracking program fires up the datacube, performing a few centroid operations just to get warmed up. It takes another centroid for a reference frame. Generally, there is nothing in the frame, so it 'sees' a centroid of $(0,0)$. Then, it keeps performing centroid operations until the frame changes[1]. The program enters its recording mode, and records the centroids of every frame for up to 150 frames (5 seconds worth of video) or until the ball stops moving[2].

The program is fairly robust, alththough it sometimes skips a frame, reporting a centroid of $(0,0)$ when there is certainly a ball within the fame. But overall, it is a quick way to get reasonably accurate data for a shot in real time.

[1]Actually, until the centroid changes.

[2]This is determined by waiting until there are 5 identical frames in a row.

Chapter 4

Testing and Results

Once completed, the robot pool player was tested for repeatability and the resolution of its sensing. This included testing the repeatability of the draw of the cue stick, the variance in direction and velocity of the shots, and a comparison with the angular resolution of the sensing.

First, two basic elements of the system were tested: the cue actuator and the vision system. Since the cue actuator is run under open loop control, it is possible for the drawback mechanism to accumulate some offset over time. The testing method and results for the cue actuator are discussed in section 4.1. The vision system needs calibration because of the aspect ratio of the video camera. In other words, something 10 pixels high and 10 pixels wide is not actually square in the real world. The method and results for testing the vision system are given in section 4.2.

Finally, the shooting characteristics of three different shots (soft, medium, and hard) from two different angles are tested. The results are analyzed for repeatability in both the angle and velocity of the shot. The testing method is presented in section 4.3, the angular trajectory results are discussed in section 4.4, and the velocity results appear in section 4.5.

Draw	(# of steps)	4096	5376	8192	9472
Trial 1	extended	5.072	5.468	6.342	6.379
	rest	3.785	3.786	3.788	3.789
Trial 2	extended	5.065	5.463	6.339	6.739
	rest	3.787	3.787	3.788	3.788
Trial 3	extended	5.065	5.460	6.345	6.739
	rest	3.787	3.789	3.790	3.784

Draw	4096	5376	8192	9472
Average draw	1.281	1.677	2.553	2.948
Pulses per inch drawn	3197.5	3205.7	3208.8	3208.7

Table 4.1: Measurements of cue actuator draw. (All measurements are in inches.)

4.1 Repeatability of the cue actuator

As discussed in section 2.3.1 of chapter 2, it is possible for the motor to skip pulses if the load is too large. This is a result of commanding an instantaneous velocity from the stepper motor. Since the motor is reasonably powerful, and it is geared down with a ball screw, I expected little problems with repeatability for this reason.

The repeatability of the cue actuator was determined by measuring the distance from the inside of the front of the shooter to the front of the traveling platform (which draws back the cue stick). The front to platform distance was measured both at the rest position and at the maximum draw for the shot. Four different draws were tested three times each. The three trials for each amount of draw were performed consecutively. The measurements appear in table 4.1. It turns out that the draw was repeatable to within 0.007 inches, which is more than precise enough to give a repeatable shot, though small errors could accumulate over time. The figures for average pulses per inch also indicate that the shooter is fairly consistent but hint that the motor may be skipping a few pulses.

4.2 Calibration of vision system

The scaling factors relating distance in the real world to distance in the image were determined by placing a six inch and a twelve inch ruler on the pool table. Aligning the rulers horizontally resulted in a scale of 15.69 pixels per inch in the x-direction. Vertical alignment showed a scale of 19.88 pixels per inch in the y-direction. This works out to give us pixels that are 0.064 inches wide and 0.050 inches tall.

Thus, the pixels are not square, but short fat rectangles.

4.3 Testing shots

There are many factors that can change between 'identical' shots. For instance, the draw of the cue manipulator may vary slightly or the angle of the cue may change slightly, even though it is clamped into place. The primary source of error for this system, however, is positioning the cue ball. Currently, we are using an allen wrench taped to the pool table to reposition the ball for each shot. A secondary source of error could possibly be the rotation of the cue stick. Since the cue tip is probably not perfectly symmetric, rotation of the cue stick could result in different trajectories for the cue ball.

Yet another alignment problem could cause variance between shots from different angles. Ideally, the cue manipulator rotates about the center of the cue ball. If instead, it rotates about a different center, shots from different angles would not be consistent because they would be hitting the cue ball to one side.

The tests shots were made from two different positions and with three different draws. The first position was approximately normal to the back bumper. The second position was still aimed at the back bumper, but close to the corner pocket. See figure 4-1 for an illustration of the two trajectories.

For each trajectory, three trials were made: soft (4096 steps), medium (6144 steps), and hard (8192 steps). Each trial consisted of 10 shots. For each shot, the cue stick was drawn back, the cue ball placed in position, and the tracking program[1] started. Once the tracking program had established the state of the pool table, the cue stick was released, and the tracking program waited until there was no longer any change in the pool table.

[1]See section 3.3 for details about the tracking program.

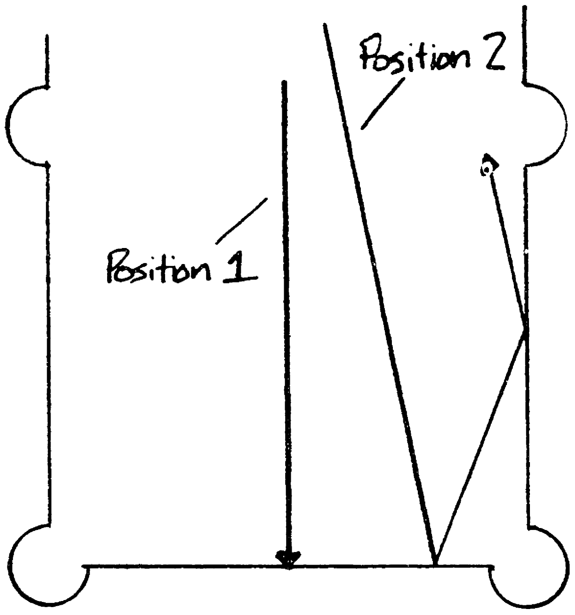


Figure 4-1: Trajectories used for test shots

Some shots were discarded because of errors in the tracking program; from time to time, it missed a frame and decided that there was no ball on the table. The first few points of each shot were also discarded because the ball had not completely entered the frame, so the centroid returned was inaccurate. Points were discarded based on the number of pixels that were detected in the ball.

For the rest of the data analysis in this thesis, the data is restricted to the trajectory during the time from when the ball completely enters the frame until just before it bounces off the back bumper. The latter criteria is easily determined because the pool table is aligned so that the back bumper is horizontal. Thus, we look at the data until the y coordinate stops increasing.[2] Section C.3 contains the LISP program that processed the raw data and transformed it into the proper format for the statistics package used for analysis.

[2]The coordinate axes of the video screen are oriented so that the origin is the top left hand corner of the screen. The x-axis increases to the right of the screen , the y-axis towards the bottom.

4.4 Angular trajectory results

Statistical analysis of the trajectory data was done using RS1. A linear regression was performed on each shot to fit a line to the data points. The angle of the shot was computed from the slope of the line, and statistics were computed on all the angles of a trial. Table 4.2 shows the results.

The last column in table 4.2, the extended standard deviation, is a measure of the distance along the back bumper covered by the angle of the standard deviation. The extended standard deviation is computed from the expression

$$2d \tan \frac{\theta}{2}$$

where d is the distance from the starting point of the cue ball to the back bumper. For the setup used in these trials, $d = 31$ inches. Most combinations of position and shot strength can deliver a consistent shot accurate to an eighth of an inch at this distance.

Note that the extended standard deviation is, in general, two to three times the size of a pixel, so the resolution of the vision system is sufficient to give us useful information about the state of the pool table.

The A/D readings from the potentiometer for the two positions are 140.6 (average of 10 readings) and 150. The average angles from the two positions are 90.18° and 77.39°. So the angular resolution of the potentiometer and A/D converter is

$$\frac{90.18^\circ - 77.39^\circ}{150 - 140.6} = 1.36^\circ$$

If this angle is extended to the back bumper, we find that it covers a distance of 0.747". So, the resolution of the shooting angle measurement is much coarser than the resolution of the vision system.

Position	Trial	Mean angle	Standard deviation	Extended Standard deviation
1	soft	90.56°	0.25°	0.135"
	medium	90.70°	0.19°	0.102"
	hard	89.29°	0.19°	0.100"
2	soft	77.43°	0.27°	0.146"
	medium	77.62°	0.12°	0.065"
	hard	77.13°	0.35°	0.191"

Table 4.2: Angular statistics. All trials consist of 10 shots. See text for explanation of extended standard deviation.

4.5 Velocity results

In addition to being able to shoot the cue ball accurately in a certain direction, it is also desirable to be able to control its velocity. The velocity of the ball for these tests was computed by subtracting two consecutive position vectors. If the coordinates have been scaled so that they are in inches, the units of the velocity are inches per 1/30 second. For N points of position data, there would be $N - 1$ points of velocity data, but since it is possible that the last position data point has occurred after a bounce, the last velocity data point is discarded. The same LISP procedures were used to generate the position data from the raw data except for some modifications which appear in section C.3.2.

Similar statistical analysis was performed on the velocity data. It turns out that a line very closely approximates the curve formed by plotting the velocities versus the distance from the shooting point of the cue ball. The equation of the line was evaluated at 13" for trials in position 1 and 14" for trials in position 2. Statistics were computed on these velocities. The results appear in table 4.3.

The velocities are actually very consistent. Though the standard deviation of the velocities for the trials are all fairly close together, the percent standard deviation is much greater for the soft shots than the other amounts of draw. Note that the maximum standard deviation is 0.0259 inches per 1/30 second, but a pixel is 0.064 inches wide and 0.050 inches tall! The accuracy of producing a desired velocity is greater than the accuracy of the vision system.

Position	Trial	Mean velocity	Standard deviation	Percent standard deviation
1	soft	1.129	0.0259	2.29%
	medium	1.916	0.0202	1.05%
	hard	3.025	0.0249	0.82%
2	soft	1.140	0.0282	2.47%
	medium	1.887	0.0142	0.75%
	hard	2.976	0.0236	0.79%

Table 4.3: Velocity statistics. Mean velocity and standard deviation are in units of inches per 1/30 second.

Chapter 5

Conclusion

In the course of this thesis project, I have designed, built, and tested a robot pool player. Currently, the sensing of the orientation of the cue stick is the limiting factor that prevents the system from accurately shooting the cue ball along an arbitrary trajectory.

There is an easily implemented solution that may solve the problem of orientation sensing. Currently a potentiometer acts as a voltage divider to provide the input to an analog to digital (A/D) converter. Since the whole range of the potentiometer is not being used, an amplifier could be inserted between the potentiometer and the A/D converter. This could increase the resolution of the orientation sensing by a factor of 6, enough to bring the accuracy of the orientation sensing to a level comparable to the accuracy of shooting the cue ball. If even more resolution were needed in the orientation sensing, a video camera could be mounted atop the cue actuator to look down the cue stick, just as people do when they play pool.

This modification would make the error in placing a shot the limiting factor of the system. It may be the case that shooting the cue ball must be made more accurate. It is likely that the repositioning of the cue ball at its shooting point is a main cause of the noise in the system. A better method of repositioning the ball would be required. Additionally, a second linear bearing, installed to reduce the amount of play in the cue stick, could also help.

The velocity produced by the cue actuator turns out to be very consistent; the vision system is the limiting factor for determining the velocity. I believe that the velocity repeatability is more than accurate enough to play pool since variations in the direction of a shot have far more effect on the shot than variations in velocity.

Though the robot pool player is currently limited to shooting from a fixed point, this rotating mount could potentially be attached to a movable base so that the robot pool player might play a semi-normal game of pool. The robot pool player could then choose not only the direction and power of the shot, but also where to hit the cue ball to impart the desired amount of spin. An additional camera, mounted atop the cue actuator, would most likely be needed to guide the precise positioning of the cue stick; the overhead camera would not have enough resolution to do so. The cue ball repositioning problem would then be completely eliminated.

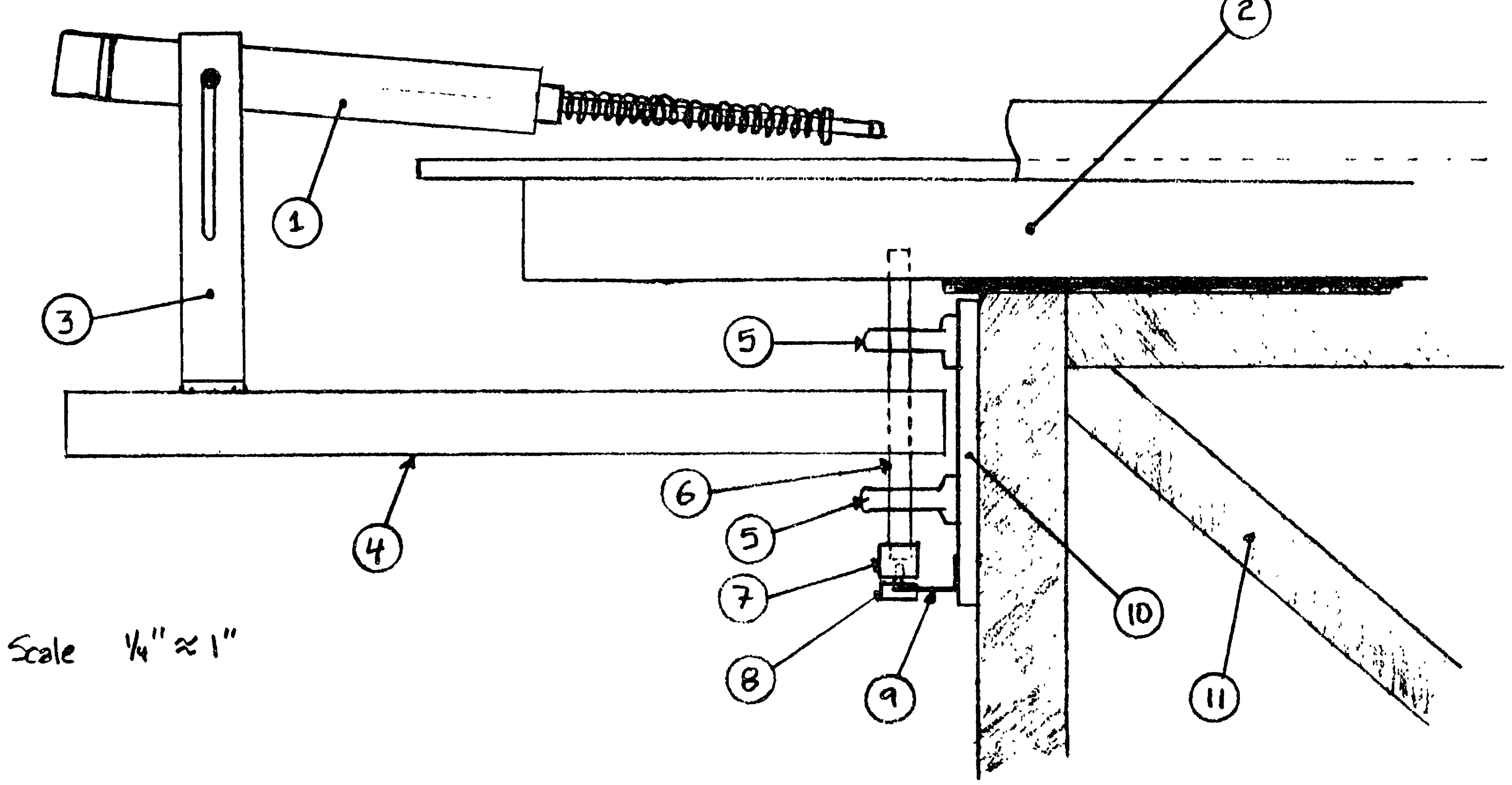
The work in this thesis can be extended and built upon in order to study various aspects of learning through the many levels of playing pool. One could write down the physics of two ball collisions in order to have a system that could sink a single ball. Alternatively, the robot pool player could be told nothing about the physics of pool and be left to learn on its own. Controlling the position of the cue ball after a shot and overall game strategy are two other areas that could be studied. There are many possibilities.

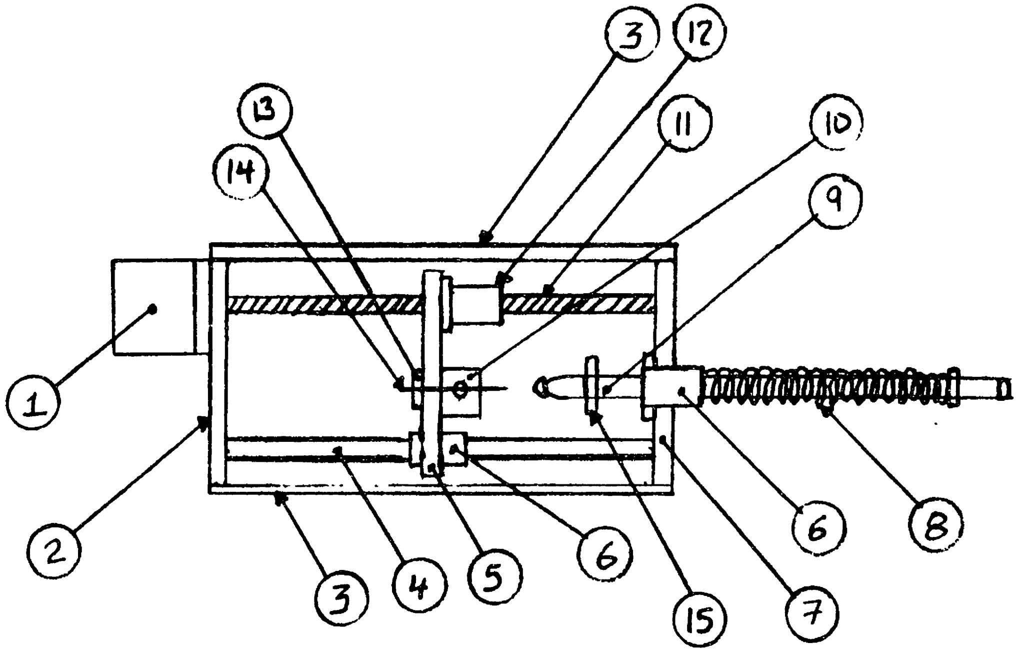
Appendix A

Mechanical drawings of the robot pool player

Assembly Diagram for the Robot Pool Player

1	Cue Actuator	7	Potentiometer Coupling
2	Miniature Pool table	8	Potentiometer
3	Mounting Bracket	9	Potentiometer Mounting
4	Mounting Arm	10	Mounting Plate
5	Bearing	11	Table
6	Pivot Shaft		



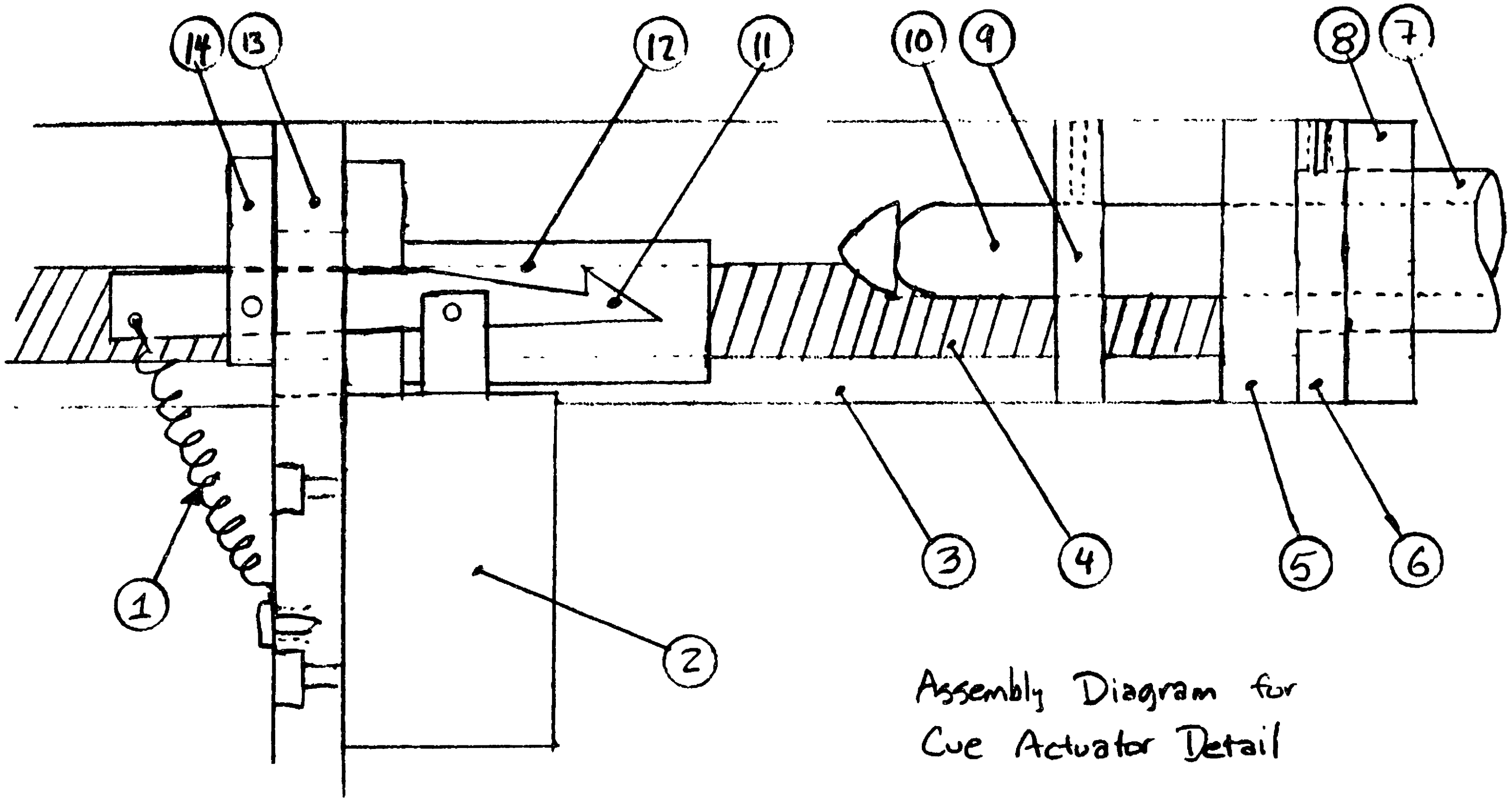


Assembly Diagram for the Cue Actuator

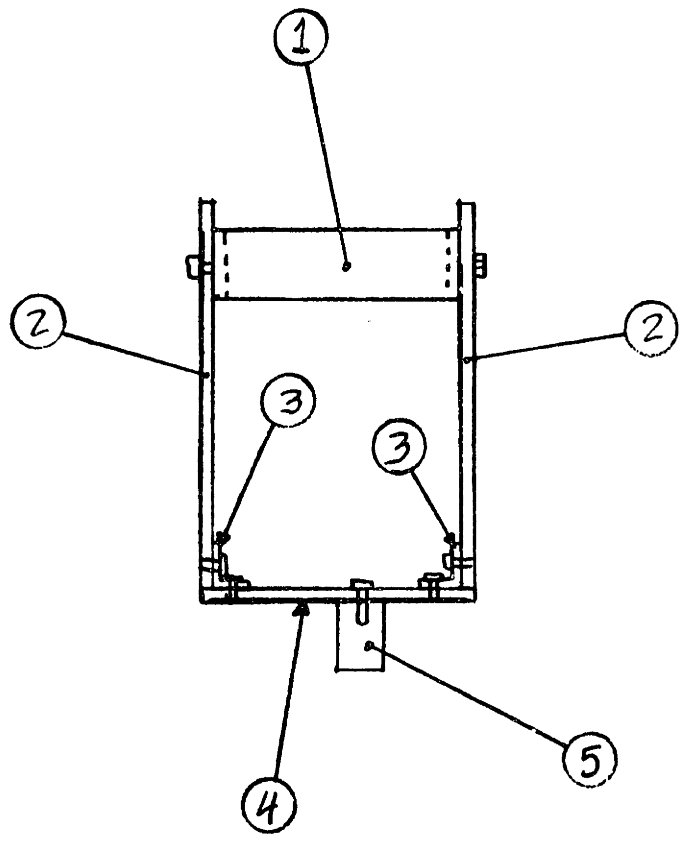
1	Stepper Motor
2	Back
3	Side
4	Guide Shaft
5	Platform
6	Linear Bearing
7	Front
8	Springs
9	Cue Stick
10	Solenoid
11	Ball Screw
12	Ball Nut
13	Pivot Block
14	Cue Latch
15	Cue Stop

Scale 1/4" = 1"

1	Spring	6	Foam Support	11	Cue Latch
2	Solenoid	7	Linear Bearing	12	Ball Screw
3	Side	8	Front	13	Platform
4	Ball Screw	9	Cue Stop	14	Pivot Plate
5	Foam	10	Cue Stick		



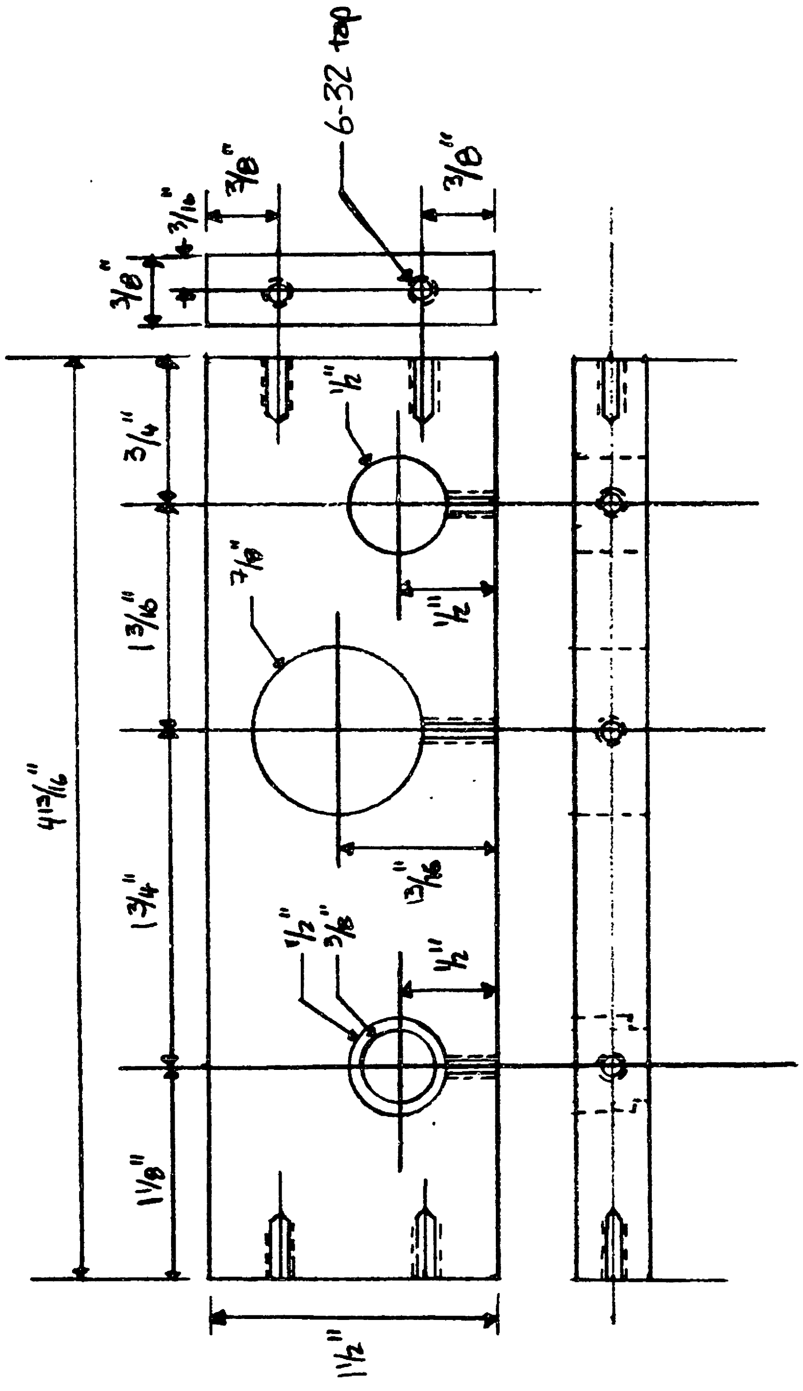
Assembly Diagram for
Cue Actuator Detail



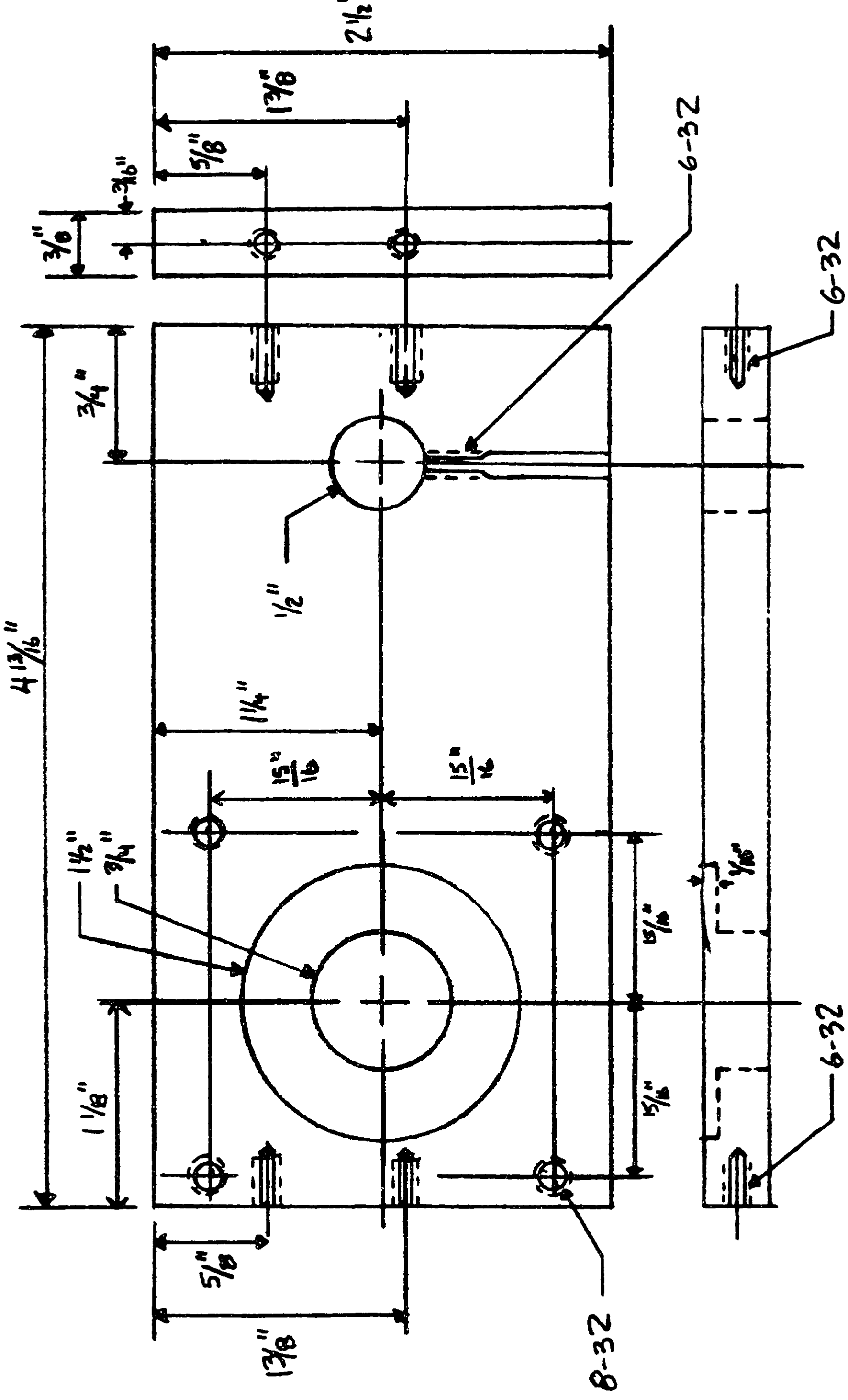
Assembly Diagram for the Mounting Bracket

1	Cue Actuator
2	Side
3	Corner
4	Bottom
5	Mounting Arm

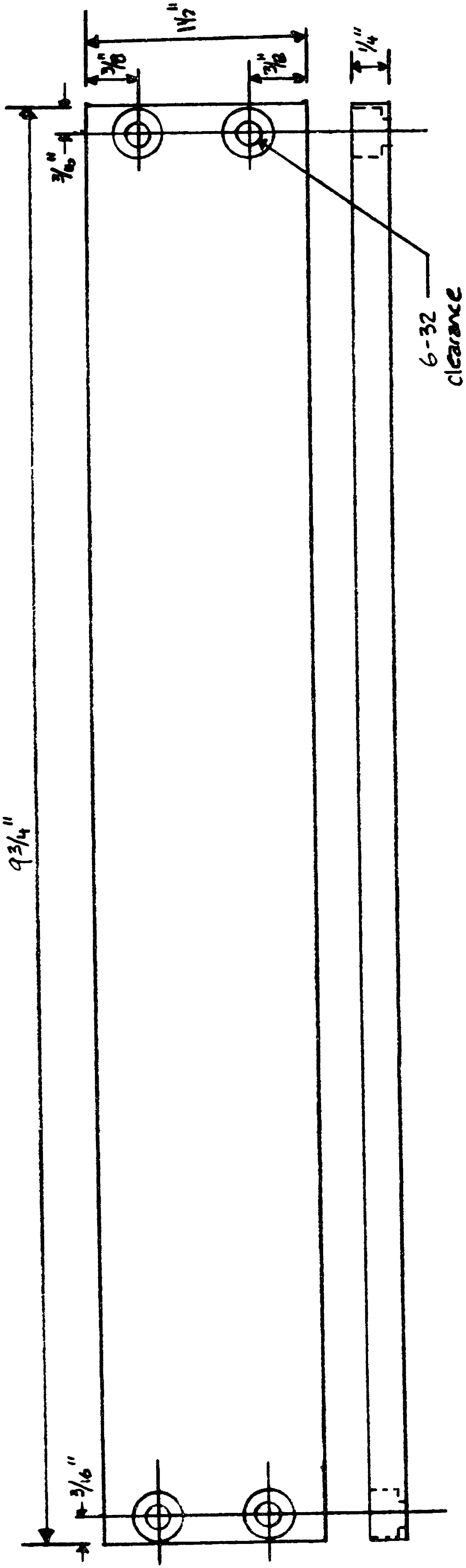
Scale $1/4" = 1"$

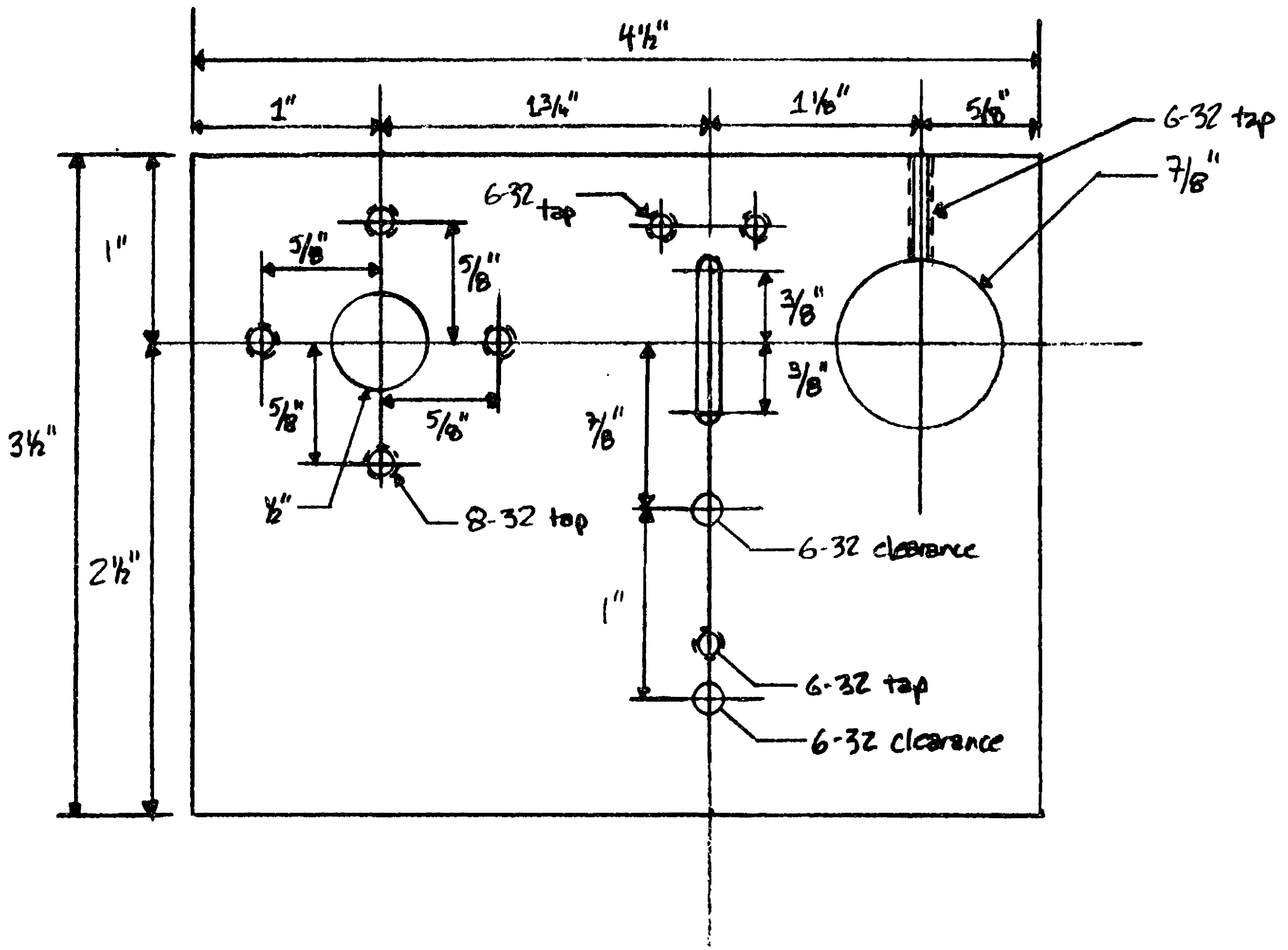


Cue Actuator Front

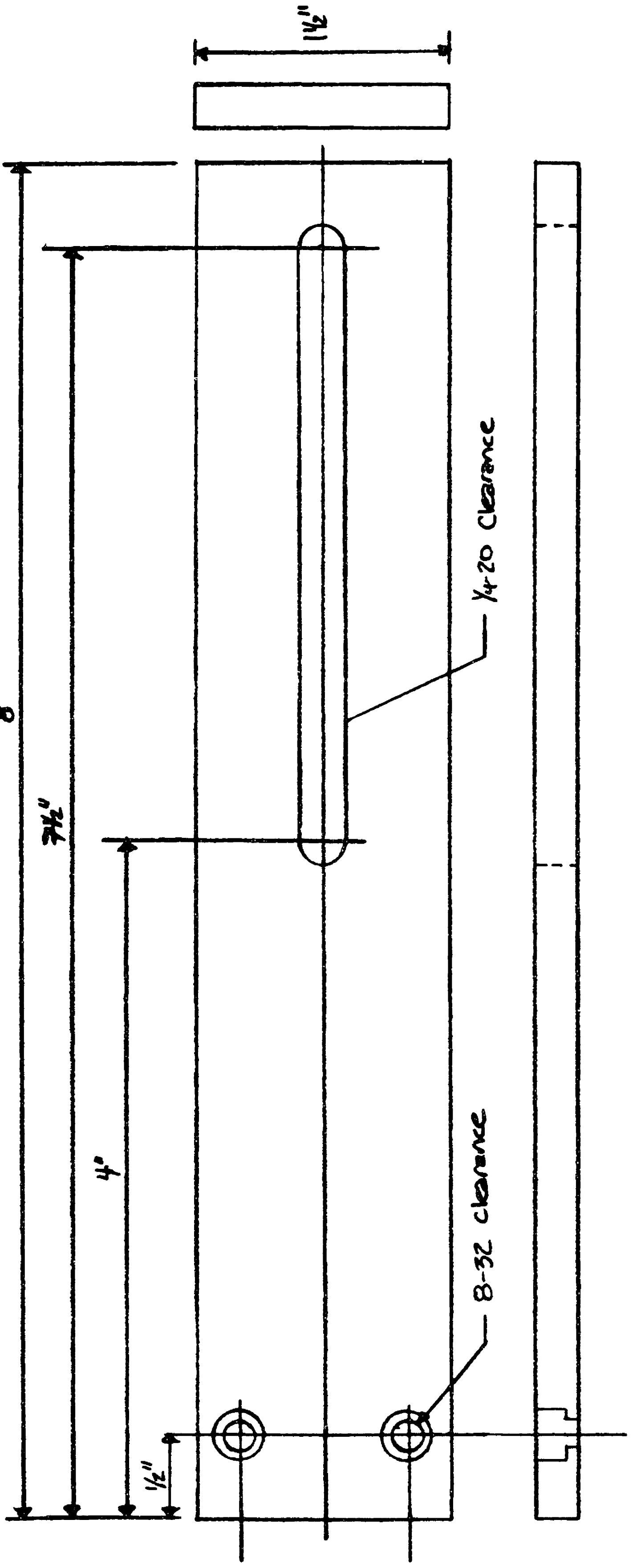


Cue Actuator Back

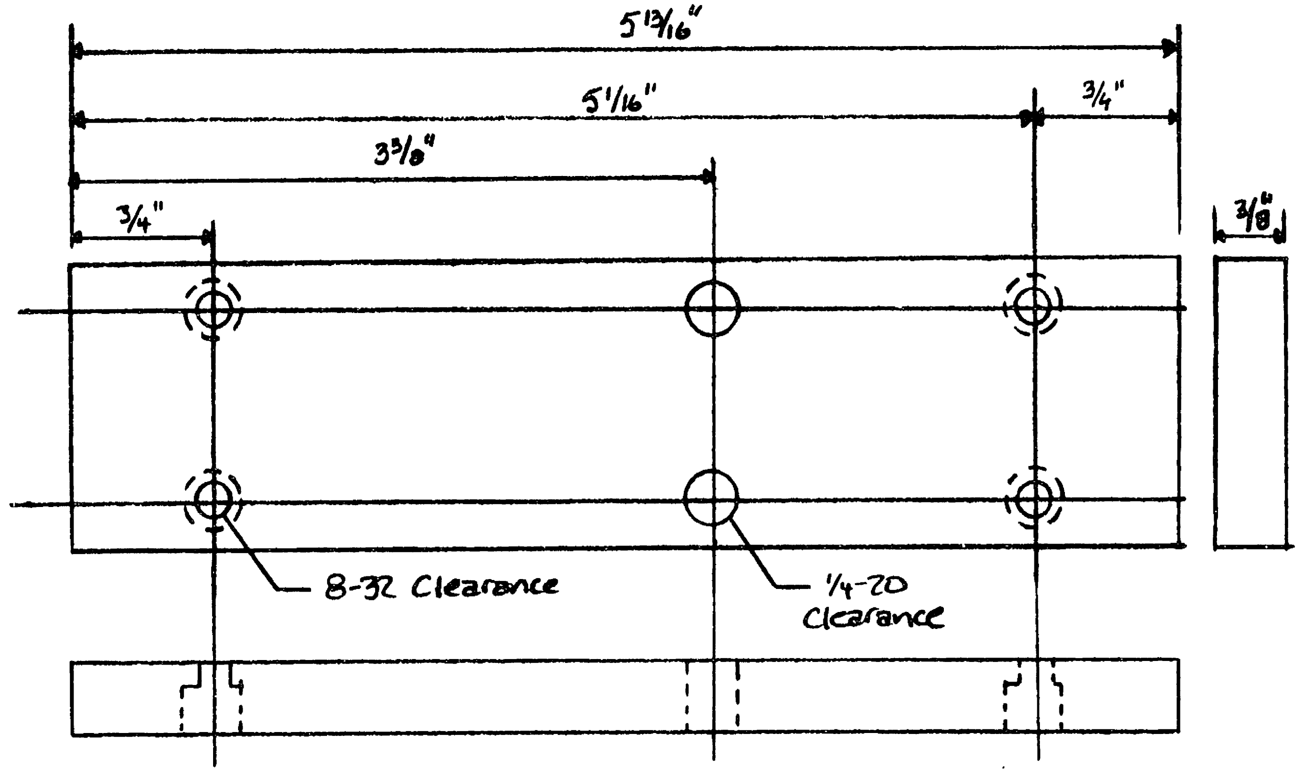

Cue Actuator Side



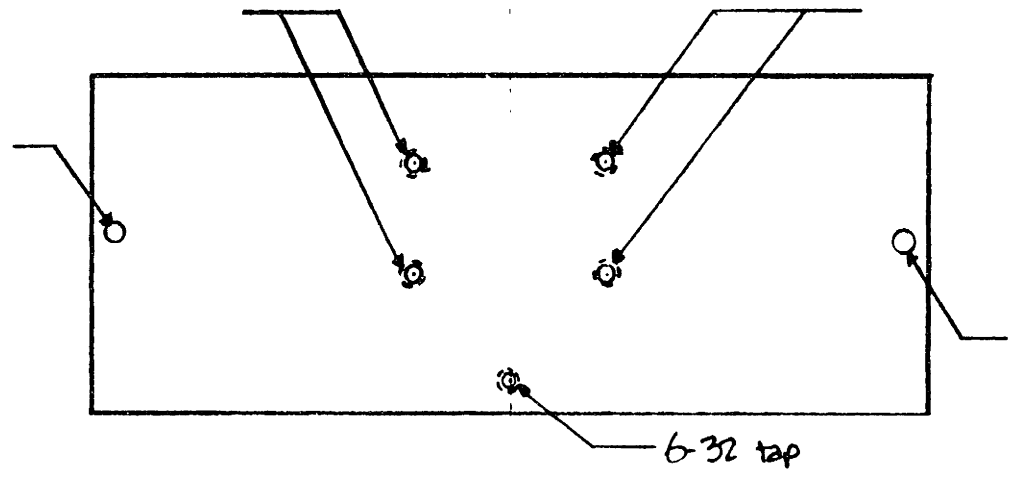
Cue Actuator Platform



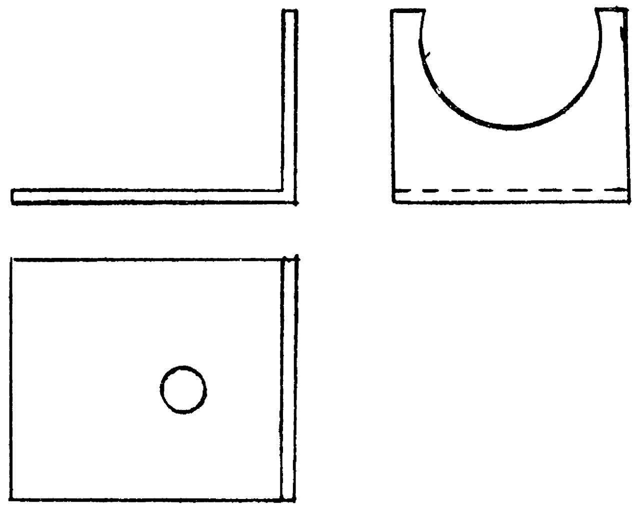
Mounting Bracket
Side



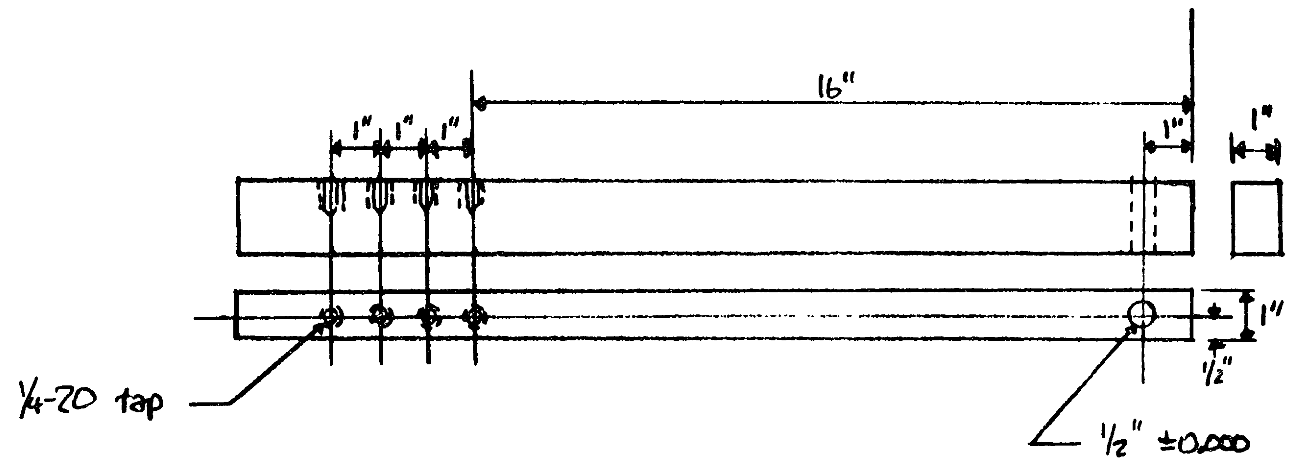
Mounting Bracket
Bottom



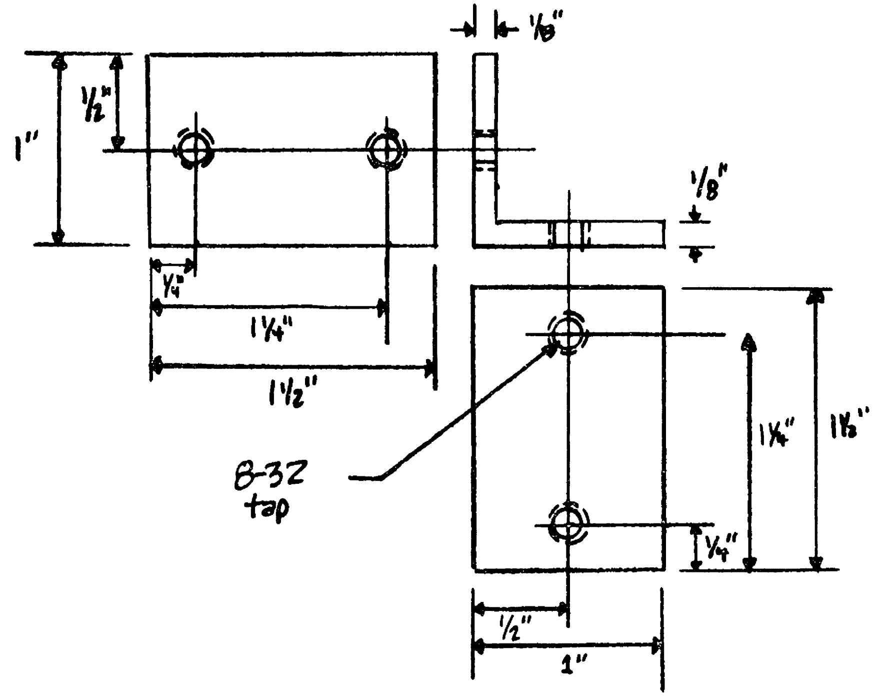
Mounting Plate
Scale 1/4" = 1"



Potentiometer Mounting



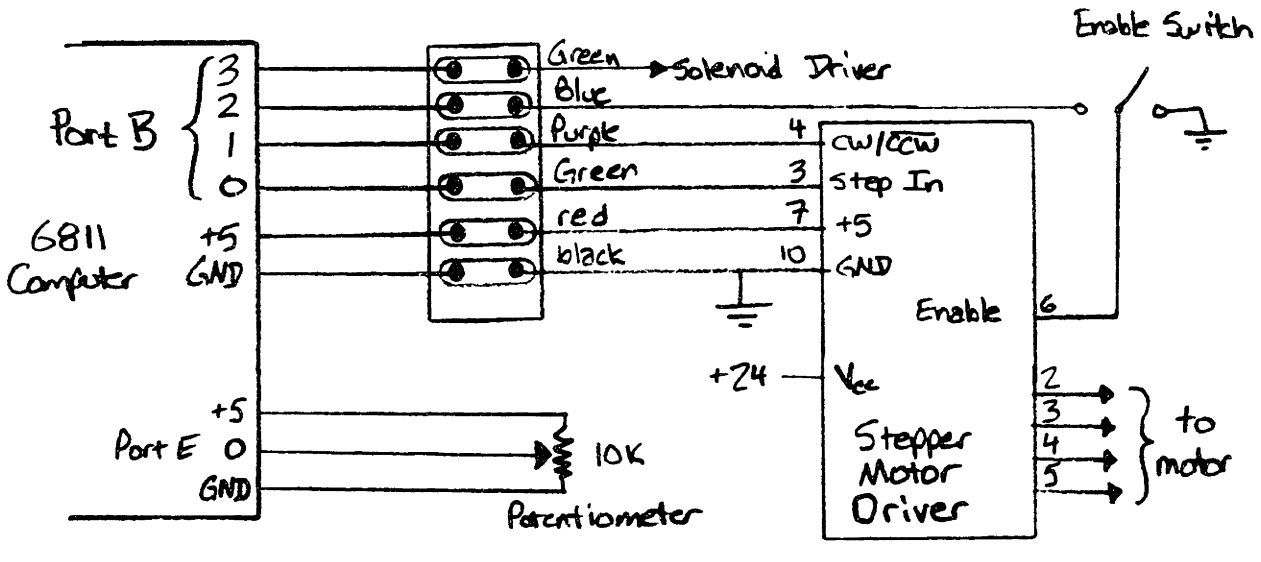
Mounting Arm
Scale ¼" = 1"

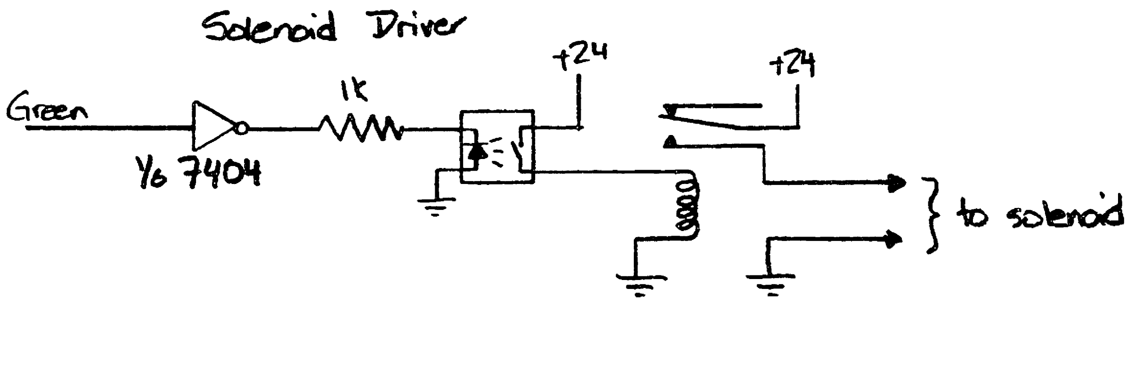


Mounting Bracket
Corner

Appendix B

Wiring of electromechanical elements





Appendix C

Program listings

C.1 FORTH procedures for 6811 board

```
( Move
( user
( memory
HEX
100 TIB !
50 TIB 2+ !
200 DP !
DECIMAL

3 CONSTANT COUNTUP  ( delay between pulses
3000 CONSTANT COUNTUP2  ( delay until releasing solenoid
VARIABLE PORTB 0 PORTB ! ( variable to hold value to be written

( pulse delay routine)
: DELAY COUNTUP 1 DO  LOOP ;
( solenoid delay routine)
: DELAY2 COUNTUP2 1 DO LOOP ;

HEX
  ( write the variable PORTB to Port B
: WRITE-OUT PORTB @ B004 C! ;
  ( turn on the solenoid
: SOLON PORTB @ 8 OR PORTB ! ;
  ( turn off the solenoid
: SOLOFF PORTB @ F7 AND PORTB ! ;
  ( turn on motor 1
: MOTOR1ON PORTB @ 1 OR PORTB ! ;
  ( turn off motor 1
: MOTOR1OFF PORTB @ FE AND PORTB ! ;
  ( Set direction of motor 1 clockwise
: CLOCKW PORTB @ 2 OR PORTB ! ;
  ( Set direction of motor 1 counter clockwise
: CCLOCKW PORTB @ FD AND PORTB ! ;
  ( enable the motor
: ENERGIZE PORTB @ 4 OR PORTB ! WRITE-OUT ;
  ( disable the motor
: POWERDOWN PORTB @ FB AND PORTB ! WRITE-OUT ;
  ( waits for keystroke
: WFT BEGIN ?TERMINAL UNTIL ;
  ( send single pulse
: PULSATE MOTOR1ON WRITE-OUT DELAY MOTOR1OFF WRITE-OUT ;
  ( send a continuous stream of pulses
: PULSECOUNT 1 DO PULSATE LOOP ;
```

```
  ( send a continuous stream of pulses  until user hits a key
: PULSESTREAM BEGIN PULSATE ?TERMINAL UNTIL ;
  ( move platform forwards until keystroke
: FORWARDS CLOCKW PULSESTREAM ;
  ( move platform backwards until keystroke
: DRAWBACK CCLOCKW PULSESTREAM ;
  ( release the pool cue
: LETGO SOLON WRITE-OUT DELAY2 SOLOFF WRITE-OUT ;

( shoots the cue.  called preceeded by one argument, the number of
( steps to draw back the cue.  Maximum shot is approximately 2000
( for example: 1500 SHOOT
( note that these are hex numbers: 1000 SHOOT sends 4192 pulses
: SHOOT ENERGIZE DUP CCLOCKW PULSECOUNT WFT LETGO CLOCKW PULSECOUNT
    POWERDOWN ;

HEX
80 B038 C! ( power up a/d converters
30 B030 C! ( set up a/d for continuous conversion of an0-3
: RAD B031 C@ . ; ( read a/d converter, channel 1
: SCANAD BEGIN RAD CR 100 1 DO LOOP ?TERMINAL UNTIL ;
    ( continuously scan a/d converter until a key is hit
```

C.2 Vision routines

C.2.1 track.c

```
/*****************************************************************
        track.c  by Wes Huang  spring 1991

   This program will track a single ball in real time, reporting
   the centroid and the number of pixels for each frame.

   Data is output to stdout and can be saved to a file using
   unix output redirection.  Messages to the user appear
   on stderr, so they will not be recorded in the file.

*****************************************************************/

#include <stdio.h>
#include <dc.h>

/* Threshold for determining whether a pixel is a part of the ball
   Use 35 for high speed shutter, 175 for normal shutter */
#define BALL_THRESH 35

/* how many frames for no change to stop recording */
#define NCHANGE 5

/* max number of frames to record (at 30 frames per second) */
#define MAXFRAMES 150

main()
{

  /* pointers to the datacube boards */
  DG_DESC *dg0,*dg1;
  FM_DESC *fm0;

  /* storage for centroids and number of points */
  int Cx[MAXFRAMES],Cy[MAXFRAMES],n[MAXFRAMES];

  int frames,k,l;

  dg0 = dgOpen(DG_BASE, DG_VECTOR, 0);
  dg1 = dgOpen(DG1_BASE, DG1_VECTOR, 0);
  fm0 = fmOpen(FM_BASE,0);
  /* initializes boards, ensures that camera input is routed properly
     through the pipeline, and sets the threshold for the featuremax
     board */
  dgMaster(dg0);
  setupDG(dg0,dg1,DC_INTERNAL,BILLIARDS_1);
  setupFM(fm0,BALL_THRESH);
  mvRefresh();

  for (k=0;k<MAXFRAMES;k++)
    Cx[k]=Cy[k]=n[k]=0;

  initial_point(fm0,dg0,&Cx[0],&Cy[0],&n[0]);

  /* message displayed to user */
  fprintf(stderr,"Waiting for start.\n");

  frames = wait_and_record(fm0,dg0,Cx,Cy,n);

  for (k=0;k<frames;k++)
    printf("%d %d %d\n",Cx[k],Cy[k],n[k]);

  dgClose(dg0);
  dgClose(dg1);
  fmClose(fm0);
}

/*******************************************

  initial_point does a few centroids to
  warm up, waits for the frame to stabilize
  and settle down, then measures the
  reference frame

*******************************************/
initial_point(fm,dg,x,y,n)
FM_DESC *fm;
DG_DESC *dg;
int *x,*y,*n;
{
  int x1,y1,n1,k;

  for (k=0;k<10;k++) {  /* do 10 frames just to get warmed up */
    extract(fm,dg);
    n1 = centroid(fm,&x1,&y1);
  }
  /* get the reference frame */
  extract(fm,dg);
  *n = centroid(fm,x,y);

  /* if the reference frame wasn't the same as the previous frame,
     things haven't settled down yet.  Keep doing centroids until
     we get two consecutive centroids that are the same. */
  while ((n1 != 0) && (*n != 0) && (x1 != *x) && (y1 != *y)) {
    x1 = *x; y1 = *y;n1 = *n;
    extract(fm,dg);
    *n = centroid(fm,x,y);
  }
}



/************************************

  wait_and_record does centroids on the incoming video signal until it
  finds a frame that has changed from the reference frame.  Then every
  frame is recorded until there are NCHANGE identical frames or until
  MAXFRAMES have been recorded.

************************************/
wait_and_record(fm,dg,x,y,n)
FM_DESC *fm;
DG_DESC *dg;
int x[MAXFRAMES],y[MAXFRAMES],n[MAXFRAMES];
{
  int same,k;

  extract(fm,dg);
  n[1] = centroid(fm,&x[1],&y[1]);

  /* wait until change */
  while ((x[1] == x[0]) && (y[1] == y[0])) {
    extract(fm,dg);
    n[1] = centroid(fm,&x[1],&y[1]);
  }

  /* There has been a change, so start recording */
  k=2;        /* number of next frame to be recorded */
  same = 0;  /* how many identical frames have been recorded? */
  do {
    extract(fm,dg);
    n[k] = centroid(fm,&x[k],&y[k]);
    if ((x[k] == x[k-1]) && (y[k] == y[k-1]))
      same +=1;
    else
      same = 0;
    k++;}
  while ((same < NCHANGE) && (k < MAXFRAMES));

  return(k);
}
```

C.2.2 Selected routines from the dc library

C.2.2.1 setupDG()

```
/*
    setupDG(dg0,dg1,display,view_camera)

   initializes the digimax boards, sets the input for the D/A section,
   and sets the input for the A/D section.

   the argument 'display' may be set to:
    DC_INTERNAL
    DC_P6 (6)
    DC_P7 (7)
    DC_P8 (8)

   the argument 'view_camera' is simply passed to select_camera.
   it may be set to:
     DC_QUERY
    or to a camera name/number

*/
setupDG(dg0,dg1,display,view_camera)
DG_DESC *dg0,*dg1;
int display,view_camera;
{
  dgInit(dg0,DG_UNSGD);
  dgInit(dg1,DG_UNSGD);
  switch (display) {
    case DC_INTERNAL: dgDtoASrc(dg0,DG_INTDISP);
                      dgDtoASrc(dg1,DG_INTDISP);break;
    case DC_P6: dgDtoASrc(dg0,DG_P6DISP);
                dgDtoASrc(dg1,DG_P6DISP);break;
    case DC_P7: dgDtoASrc(dg0,DG_P7DISP);
                dgDtoASrc(dg1,DG_P7DISP);break;
    case DC_P8: dgDtoASrc(dg0,DG_P8DISP);
                dgDtoASrc(dg1,DG_P8DISP);break;
  }
  select_camera(dg0,dg1,view_camera);
}
```

C.2.2.2 setupFM()

```
/*
    setupFM(fm,comp_val)

   sets up the featuremax board to do feature extraction

   the argument 'comp_val' may be set to the desired comparator value,
   or DC_QUERY, in which case the procedure will ask for a comparator
   value.

   if an illegal comparator value is provided, the procedure will do
   nothing and return -1.

   successful completion of the procedure will return the comparator
   value
*/
setupFM(fm,comp_val)
FM_DESC *fm;
int comp_val;
{
  int c;

  if (comp_val == DC_QUERY) {
    printf("Enter value for feature comparator:\n");
    scanf("%d",&c);
  }
  else
    c = comp_val;

  if ((c<0) || (c>255)) {
    printf("illegal featuremax comparator value.\n");
    return(-1);
  }

  fmInit(fm);
  fmIntlMd(fm,FM_INTL);
  fmBlankMd(fm,FM_BL1);
  fmDataMd(fm, DQ_2SCMP);
  fmComp(fm,c);
  fmHOffs(fm,14);
  fmFeatMd(fm,FM_GTNOTAG);
  fmMapMd(fm,FM_XYMAP);
  return(c);
}
```

C.2.2.3 extract()

```
/*
    extract(fm,dg)

  operates the featuremax in feature extraction mode for one field as
   determined by the digimax board.

  setupFM() must be run before the first call of this procedure

  this procedure will wait for the completion of the extraction
*/
extract(fm,dg)
FM_DESC *fm;
DG_DESC *dg;
{
/*  dgWaitEven(dg);*/
  fmCtrlMd( fm, FM_FTCNTR ); /* Clear counter. */
  fmTrigMd( fm, FM_FREE ); /* start triggering unconditionally */
  fmTrigMd( fm, FM_SAFE ); /* stop triggering */
  fmCtrlMd( fm, FM_FTEXT );  /* set feature extraction */
  fmTrigMd( fm, FM_FREE );
  dgWaitFld(dg, 1);      /* process one field */
  fmTrigMd( fm, FM_SAFE );
  mvRefresh();
/*  mvRefresh();*/
}
```

C.2.2.4 centroid()

```
/*
   centroid(fm,Cx,Cy)

  when called after an extract operation, this procedure will find the
   centroid of the pixels in the feature memory

  it will only examine up to MAXFEAT pixels

  returns the number of features counted (which may be greater than
   MAXFEAT)
*/
centroid(fm,Cx,Cy)
FM_DESC *fm;
int *Cx,*Cy;
{
  int feats,f,k;
  unsigned short *ptr;
  long int Sx,Sy;

  feats = (int) fmGetFCount(fm);

  if (feats > MAXFEAT)
    f = MAXFEAT;
  else
    f = feats;

  if (f>0) {
    ptr = (unsigned short *)(fm->MemBase + 2);
    Sx = Sy = 0;
    for (k=0;k<f;k++) {
      Sx += (long int) (*ptr++);
      Sy += (long int) (*ptr++);
    }

    *Cx = Sx/f;
    *Cy = Sy/f;
  }
  else
    *Cx = *Cy = 0;

  return(feats);
}
```

C.3 Data processing procedures

These routines were used to process the raw data produced by track.c to select the data points before the first bounce of the ball and to put the data in a format that can be read by RS1, the package used for statistical computations.

C.3.1 bb.lisp

```
(in-package 'user)

;; bb = selects data points Before the Bounce
;;      (where y coordinate stops increasing)

(setq xscale 15.69)
(setq yscale 19.88)

(defun process-datafiles ()
  (format t "Enter the base file name to be processed.~%")
  (setq fbasename (read-line))
  (format t "Enter output file name.~%")
  (setq foutname (read-line))
  (format t "~%Processing files...~%")
  (with-open-file (ofile foutname :direction :output)
    (do ((i 1 (+ i 1)))
((= i 11))
      (progn
(setq fname (concatenate 'string fbasename
 (format nil "~2,'0D" i)))
(format t "~a =" fname)
(process-file fname ofile)))))

(defun process-file (infile outfile)
  (with-open-file (ifile infile :direction :input)
    (setq frstpt (read-first-point ifile))
    (setq datapts (cons frstpt
                        (read-rest-points ifile (cadr frstpt))))
    (format t "~d points.~%" (length datapts))
    (write-out outfile datapts 0 #'car)
    (write-out outfile datapts 0 #'cadr)))


(defun read-first-point (ifile)
  (let ((x (read ifile))
(y (read ifile))
(n (read ifile)))
```

```
    (if (> n 235)
(list (/ x xscale) (/ y yscale))
(read-first-point ifile))))

(defun read-rest-points (ifile prevy)
  (let ((x (read ifile nil 'all-done))
(y (read ifile nil 'all-done))
(n (read ifile nil 'all-done)))
    (if (or (eq x 'all-done)
    (< y prevy))
nil
(cons (list (/ x xscale) (/ y yscale)) (read-rest-points ifile y)))))

(defun write-out (ofile data written function)
  (cond ((and (eq written 10)
      (not (eq data nil)))
 (format ofile " &~%")
 (write-out ofile data 0 function))
((eq data nil)
 (format ofile "~%"))
(t
 (format ofile "~d " (apply function (list (car data))))
 (write-out ofile (cdr data) (+ written 1) function))))
```

C.3.2 bbv.lisp

```
(in-package 'user)

(setq xcenter 12.859)
(setq ycenter -8.96)

(defun radius (pt)
  (let ((x (- (car pt) xcenter))
(y (- (cadr pt) ycenter)))
    (sqrt (+ (* x x) (* y y)))))

(defun process-file (infile outfile)
  (with-open-file (ifile infile :direction :input)
    (setq frstpt (read-first-point ifile))
    (setq datapts (cons frstpt
                        (read-rest-points ifile (cadr frstpt))))
    (format t "~d points.~%" (length datapts))
    (write-out outfile datapts 0 #'radius)
    (wov outfile datapts 0)))

(defun write-out (ofile data written function)
  (cond
    ((eq (length data) 2)
     (format ofile "~%"))
    ((and (eq written 10)
  (not (eq data nil)))
     (format ofile " &~%")
     (write-out ofile data 0 function))
    (t
     (format ofile "~f " (apply function (list (car data))))
     (write-out ofile (cdr data) (+ written 1) function))))

(defun wov (ofile data written)
  (cond ((eq (length data) 2)
 (format ofile "~%"))
((and (eq written 10)
      (not (eq data nil)))
 (format ofile " &~%")
 (wov ofile data 0 ))
(t
 (format ofile "~f " (velocity (car data) (cadr data)))
 (wov ofile (cdr data) (+ written 1) ))))

(defun velocity (pt1 pt2)
 (let* ((dx (- (car pt2) (car pt1)))
(dy (- (cadr pt2) (cadr pt1)))
(v (sqrt (+ (* dx dx) (* dy dy)))))
   v))
```

Bibliography

[1] W. Cheung, K. Khodabandehloo, and I.J. Rennell. Development of expert robot systems for skilled operation. In *Proceedings of the 20th International Symposium on Industrial Robots in Tokyo, Japan*, 1989.

[2] C.B. Daish. *The Physics of Ball Games.* Hodder and Stoughton, 1972.