Computer Vision Based Ball Catcher

Peter Greczner (pag42@cornell.edu)
Matthew Rosoff (msr53@cornell.edu)
ECE 491 – Independent Study, Professor Bruce Land

Introduction

This project implements a method for catching a ball rolling down an inclined plane. It uses a webcam to capture the image data, computer vision algorithms to correctly identify the ball, and a PID controller to move a motor to receive the ball.

Microcontroller Design

The microcontroller, ATMEGA32, performs a very simple task in our project; take in a reference value sent in from the serial port and run a Single-Input-Single-Output controller with distance feedback from optical shaft encoders attached to the motor. The output of the controller is fed into a PWM module, to control the speed of the motors via an H-bridge circuit.

Motion Control

While the microcontroller's task is simple, it is not trivial. The control loop must work almost flawlessly, otherwise the larger, ball tracking control loop will be unstable, or unreliable. The microcontroller must implement a feedback control loop to operate the motors, and run interrupt-driven serial communication and shaft encoder processing routines.

Electronic Hardware

The hardware involved with the low-level motion control is a

- A microcontroller (ATMEGA32)
- A motor with shaft-encoder feedback
- A high source current power supply
- A motor control (H-Bridge) IC (LMD18200)
- A serial level converter (Max233)

Specifications

The control loop must run at a fixed frequency, so we must queue input, or reference values, as they come in. If the control loop doesn't run at a fixed frequency, the response of the system will change, as we are simulating a continuous system in a discrete, or sampled, fashion.

Baud rate is set to 230400 bps, for the fastest, while still reliable, data transfer rate.

Interrupts from the main control loop from the serial port and from updating the distance variable from the shaft encoder pulses must not significantly alter the main control loop's operating frequency.

Feedback Controller Design and Specs

The specifications of the controller are:

- Zero steady state error
- All closed-loop poles are real, meaning that there are no damped oscillations in the output of the system. All closed loop poles must lie on the negative real axis.
- Robust to slightly uneven response along track length, due to physical setup
- Reliable, i.e. little drift, over time

We decided to use a simple PI controller because it is both fast, robust when used with stable open-loop poles, and produces a zero steady-state error. The system before feedback control is stable, or, in other words, the open loop poles of the system are stable. A proportional controller would meet spec, and performs very well, with a quick response time, T_r. As a matter of fact, our first approach was to simply increase K, the proportional control constant, until the steady state error was close enough to zero and the response was quick enough.

However, several problems arise from this approach. Firstly, there is a time delay, partially from the time it takes to compute the sensor output, and partially from the inertia of the motor and the system. This time delay, when used with a proportional controller with high gain K, will produce complex poles as K increases, and these complex poles will slowly, as K increases even more, wander into the unstable territory (Closed loop poles in the RHP). To solve this problem, we chose a K_p such that the closed loop poles of the system were real, i.e full damping, no damped oscillations, but were marginally real.

We then added integral control to drive the steady state error of the system to zero. We chose a K_I, integration constant, to be much smaller than K_p so that it didn't affect the response by driving the closed loop poles to be complex (damped oscillations in the output), but still large enough to compensate for steady state error. We also implemented an integral wind-up checker, so that the integral would not build up too large and overflow in the microcontroller.

Our final expression for the controller is $K(s)= K_p + K_i / s$. Or in a proper-fraction format: $K(s) = (K_p*s+K_i) / s$. The controller, however, must be brought into the discrete domain, which is very straightforward for the simple PI controller we chose. If we chose a more complicated controller, one would have to use matlab to make a straightforward conversion from the continous s-domain into a discrete time domain.

$K(error_i)= K_p*error +K_i * SUM(error_j, j=0 \text{ to } i-1)$. The sum approximates the integral. The dt time step from the integral is rolled up into the K_I constant.

The experimentally determined constants, K_i = 0.00001 and K_p = 0.06 with a feedback loop frequency of F_{clk} /1024 where F_{clk}= 14.7456 MHz. So, Loop Freq. = 14.4 kHz , which is more than quick enough to update changes in inputs. The shaft encoder will output pulses around 2 kHz max.

Code

All code for the ATMEGA32 was written using C and the WINAVR compiler.

Object Recognition

Capturing:

To capture the video we use a Logitech webcam. The resolution is 320 x 240 pixels RGB color image. Our client program is written in Java, and employs the Java Media Framework (JMF). In this program we connect to the webcam input stream to pipe in the video and create a media player to process capture each frame as it comes in. Once a frame is captured it is sent to be processed and filtered.

Ball Detection:

In order to detect the ball we must go through a multi-stage process. The general idea in the process is to convert the image to grayscale, determine the region of interest (ROI), canny edge detect, and then perform a circular Hough transform.

The first step is to convert the image to grayscale. When we get the initial image is of 0-255 RGB value in an array of size 320 x 240. To convert, we loop through this array and perform an RGB to grayscale conversion with the following code:

```
float red         = (pixels[index] >> 16) & 0xff;
float green       = (pixels[index] >>  8) & 0xff;
float blue        = (pixels[index])       & 0xff;
pixels_gray[index]= (int)(red*.3 + green*.59 + blue*.11);
```

After we have converted the image to grayscale, the next thing we want to do is define the region of interest. In our setup we want the region of interest to be everything that pertains to the black sheet. Due to variations in lighting, the fact that surrounding bodies may have colors of the same as the sheet, and other issues, determining the black sheet is not as easy as a simple black thresholding. In order to calculate the region of interest, we first have to make a few assumptions. One assumption is that what the camera is currently looking at should be dominated by the black sheet, and that at the bottom of the cameras view we will always see the black sheet.

To account for the lighting in any given situation we take five regions in the image. They are centered in the top, bottom, left, right, and center of the image. Each region is 7 x 7 pixels big, and the average values of these pixels are computed. We then take the highest and lowest average values and

throw them out. Out of the three remaining regions, we take the average of their values. This value is what we believe to be representative of the black colored sheet in any lighting.

After we have the hypothesized average pixel value of the black sheet we then compute the Sobel operator in both the X and Y directions to compute the gradients of the gray scale image. Next, using assumption number two that the bottom center of the camera will always be in the black region we implement a recursive function that starts at this point and searches for other similar intensity pixels and marks them as the black sheet or not. The way this works is that it moves to the right checking to see if the pixel value is within a percentage of the average and that the X and Y gradients are less than a possible edge value (because we don't want to continue off the edge of the sheet or into the ball). Once it cannot go right, it searches up, then left, and then down. The recursive search effectively marks the entire black sheet as the black sheet, with very little error. However, to account for some error, and to compute the polygon region of interest, we have to perform the following steps. The algorithm then starts in the bottom left of the image, and moves to the right, until it finds the third marked pixel as a sheet and adds that to its edge list. Then it moves up until the top of the image. Once it reaches the top it performs the same steps, but not starting from the left side and moving to the left until it reaches its third marked pixels to set as an edge for the polygon. After this is all complete, we have defined what we believe to be the black sheet.

The next step in finding the ball is to perform edge detection on the image. Our method of edge detection is a Canny edge detector. This edge detector works by first convolving the image with a Gaussian slight blurring filter. This helps to remove noise from the image that could give false edges, while preserving dominant edge structure. Next, it calculates the X and Y gradients using the Sobel gradient operators. After this, a gradient magnitude is computed at each point. If the magnitude is greater than a threshold in which we are certain is an edge, it is marked as an edge, and if the magnitude is less than a certain threshold, it is marked as not an edge. For the middle regions, we look in the local region of the pixel to see if it is the greatest for its given direction, and if it is locally the highest, we then mark it as an edge. The end result is an array with two values, one indicating an edge, the other not an edge.

We use a circular Hough transform (CHT) to find the ball in our image. The inputs to the CHT is the Canny edge detector array, a radius value, a threshold, and a region of interest.

A CHT works by going to each pixel marked as an edge and it then traces a circle with the specified radius, centered at that edge. We have another accumulator array that has its array index incremented by one, every time a circle intersects that point. So, if you have a perfectly edge detected image, with a circle of radius R, and you input this to the CHT, you will find that at the center of that circle the accumulator value will be a maximum, because the Hough circles of radius R have interested many times over while being drawn. What we do to find these maximums is to split the image up into 16 x 16 grids and mark all pixels with highest value as a regional max. We then take the regional max pixels and check to see if their accumulator value is greater than the threshold. If it is, we record this point, along with its accumulator value, as a possible circle that could be the ball. The point list, along

with the value list is returned at the end of the CHT.

We perform the CHT in a loop where the value of the radius increases from a lower bound to an upper bound, depending on where we believe the ball to be in the image. During each loop we keep track of the highest accumulator value, and its corresponding x and y values and radius. At the end of the loop we have a radius and an X and Y value that is with highest probability a circle in the desired image.

At this point we have determined where in the current image the ball lies and with what radius, assuming a ball is in the region of interest. The next step is the output to the microcontroller a position to move to so that we can catch the ball.

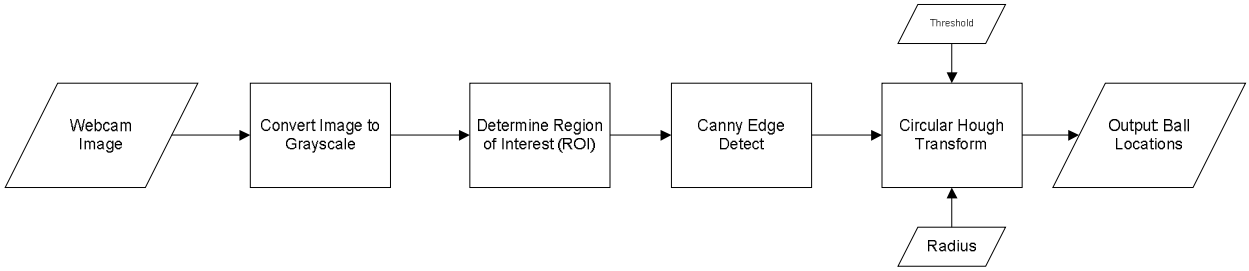


Figure: Shows the general processing steps for a given image, with wanted radius and threshold.

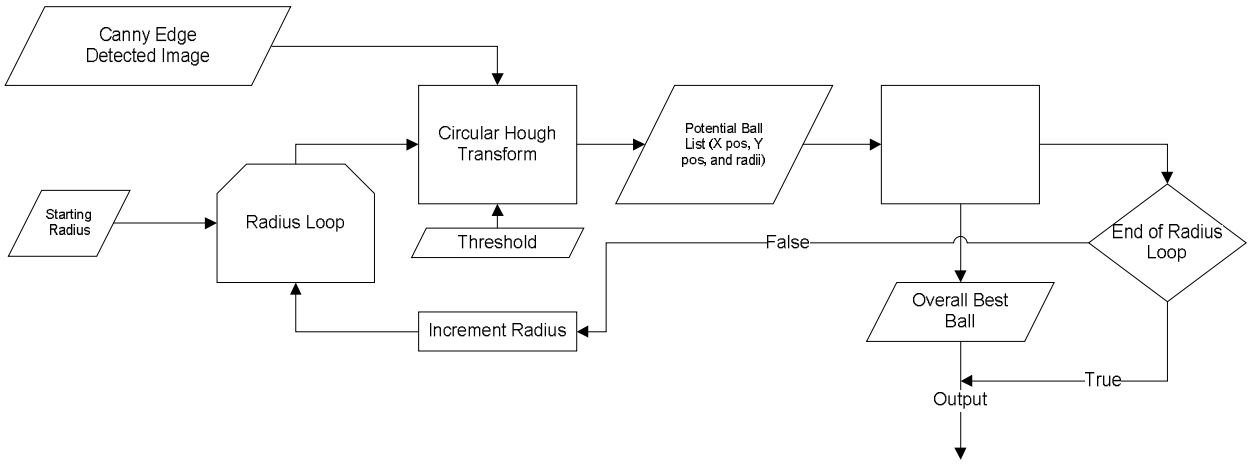


Figure: Shows the basic idea behind how the CHT determines which ball, if any, is the ball we want.

Speed Enhancements:

If we were to perform the ROI calculation, then the Canny edge detection on the entire image, then the CHT on the entire image, the result would be a frame rate of between 2-4 per second. This is not an acceptable value for our real time calculations. Therefore we developed a few techniques to speed up the processing.

First we had to determine what was taking the longest time in our processing. Test benches showed that the Canny edge detector was taking a lot of time to process. There were a few things wrong with how were using the edge detector. First, it was taking as an input an image, and then taking the pixels out of the image into an array. This was unnecessary, because when we determine our ROI, we have created the pixel array. Therefore, we changed the code to just pass in grayscale pixel array. Also, on each iteration, the edge detector was recalculating its kernels. Calculating the kernels only needs to be done on instantiation of the edge detector. Changing this removed unnecessary looping in the code. Also being done, is that it was outputting a binary image as its result, as opposed to a binary array.

Those were enhancements that we made directly to the edge detector, but at the time we were still operating on the entire image at a time and we thought there must be a smarter way. The smarter way we determined was to use the knowledge of previous calculations that produced an X and Y and radius location for the input image. The general idea of our state machine was that if no ball is found, we input the entire 320 x 240 image until a ball is found. When a ball is found, we then check to see what it's Y location is. If it is in the upper third of the image, on our next iteration we will only Canny edge detect on the top half of the image, thus reducing the number of pixels to operate on in half. If it is in the middle third, we will operate on the middle half of the image (1/4 to 3/4 of the height), and if it is in the bottom third, we will input the bottom half of the input image. We also attempt to input a compressed image by a factor of two (half width and half height) after finding a ball in the ALL region. By separating the images into regions based on where the ball was previously, we were able to get rid of unnecessary edge calculations.

The next limiting factor in our processing is our circular Hough transform. What we had been doing is computing the CHT on the entire input image result from the Canny edge detector, but this was slow. To speed things up, we decided to employ another regional technique. If we had a previous value for the ball's center and radius, we would make the input to the CHT a rectangular region that has a minimum X and Y of the previous center X and Y minus three times the previous radius, and a maximum X and Y of the previous center X and Y plus three times the radius. This we would box in the previous ball location with a guess of where the ball should be now. If it turns out our guess is wrong, on our next loop we expand the box to encompass the entire input region.

These techniques were much needed and increased our speed significantly. A table of the speed tests is included below. CED stands for Canny Edge Detector (can be fast or normal version). Fast Hough Regional is where we narrow down based on radius. Overall (bolded) you can see that the speed up was approximately 4 times, but we still lose approximately 20 FPS due to processing.

Applications	**FPS_min**	**FPS_max**	**FPS_avg**
Draw Image, Draw CED_fast, step down 2	28	32	31
Draw image, no-CED	32	34	33
Draw Nothing	32	34	33
Draw Image, draw CED_normal	12	14	13
Draw Image, no draw CED_normal	12	14	13
Draw Image, CED_fast, 1/3 image	25	27	26

Draw Image, Draw CED_Fast, 1/2 image edge detect	16	19	18.25
Draw Image, draw ced_fast, 1/2 image, scale down by 2	26	28	27.25
draw image, hough, ced_fast, 1/2 image, no scale, 9 rad	7	9	8
draw image, hough, ced_fast, 1/2 image, no scale, 5 rad	10	11	10.5
draw image/CED, regional CED_fast, compute ROI, no scale	9	11	10
draw image/CED, regional CED_fast, compute ROI, no scale, fast_hough	11	13	11.75
Draw Image, Draw CED, Draw Regions, Draw Ball, fast_hough regional	**9**	**15**	**12**
Original	**2**	**4**	**3**

Catching the Ball

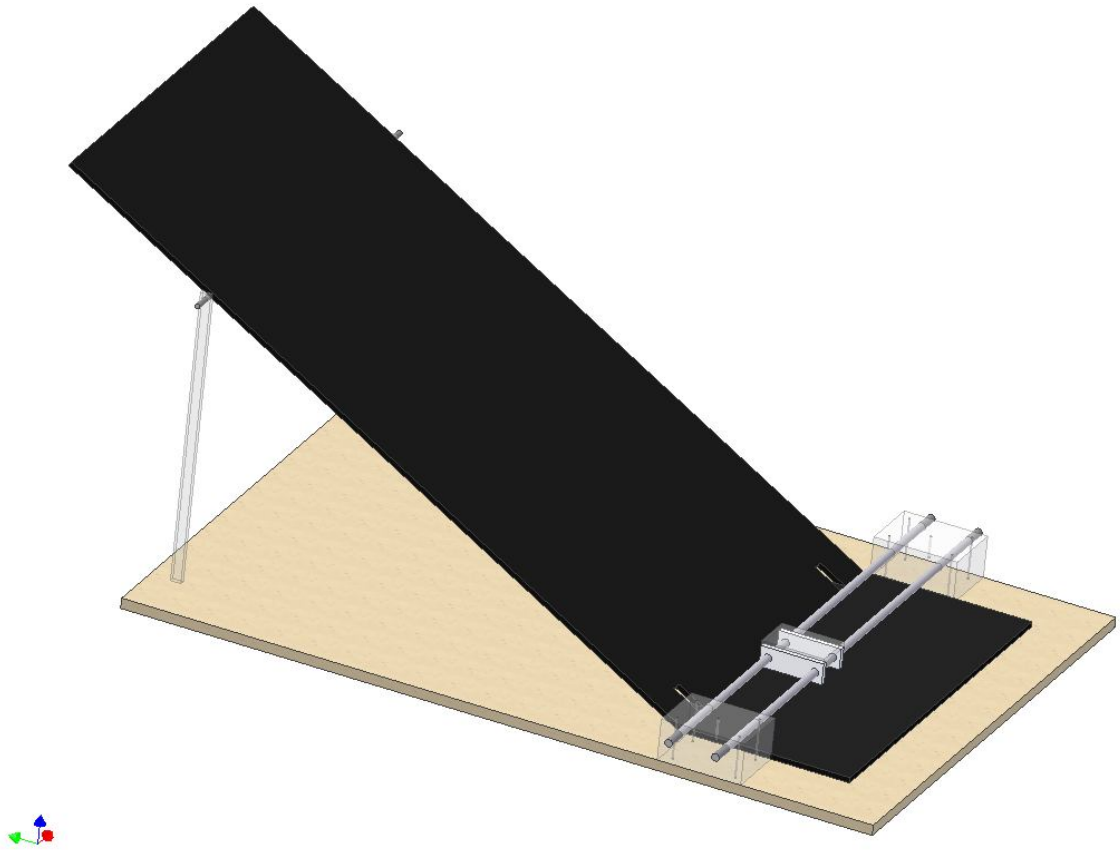
To catch the ball we had to position the plate that the camera sits on, a 3.5" wide platform so that the ball would land within its edges. On screen we are able to determine an X position of the ball as well as a radius. Since the image is 320 pixels wide, an X value greater than 160 means the ball is right of the camera, and anything less means the ball is left of the camera. Now, if the ball has a smaller radius, that means the ball is farther away from the camera, than if the radius is large, and the ball is close to the camera. Knowing these two values can help us guess where the ball is relative to the camera. If the ball is P pixels away from 160 and has a radius of 10 (far away) versus (50) close we would want to move proportionally future for the radius of 10, than the radius of 50, because the ball is probably situated further out. To imagine this, think about when you look down a long stretch of highway. The further you see the closer both sides of the roads come to converging to one point. So, if at the very end of your vision, where the road is perceptively more narrow than in front of you, you have see something that looks like it is one inch to you right, is it actually one inch? No, since you are far this is only an illusion, and it is much more than one inch to your right. Now, if you are looking at some object one foot in front of you, and it looks like it is one inch to your right, odds are it is very close to one inch to your right.

GUI Design and Use

The GUI was written in Java using the Java Comm library (for serial communication), the Java Media Framework, and standard Java. There are three main buttons to be concerned with. The first is the capture video button. This button initializes the connection with the camera and will start processing the data. The start serial button will initialize the serial communication between the PC and the microcontroller and move the camera to the center of the board. Lastly, the Track Ball button is a toggle button. The initial press will make the camera now follow the ball, and a subsequent press will turn following off. When initializing the camera, some error messages will print out to the console, but these can be ignored. It is a result of the GUI trying to paint an image that hasn't been received yet. Otherwise, the application is pretty robust. Details on what you are viewing are included below in the images section.

Physical Construction

The physical layout is as shown below. Drawings and modeling was done in Autodesk Inventor.



Materials Used:

- 2'x3' MDF (Fiberboard)
- Clear Polycarbonate plastic
- Delron blocks
- 3/8" Aluminum rods
- Acrylic Matte-finished

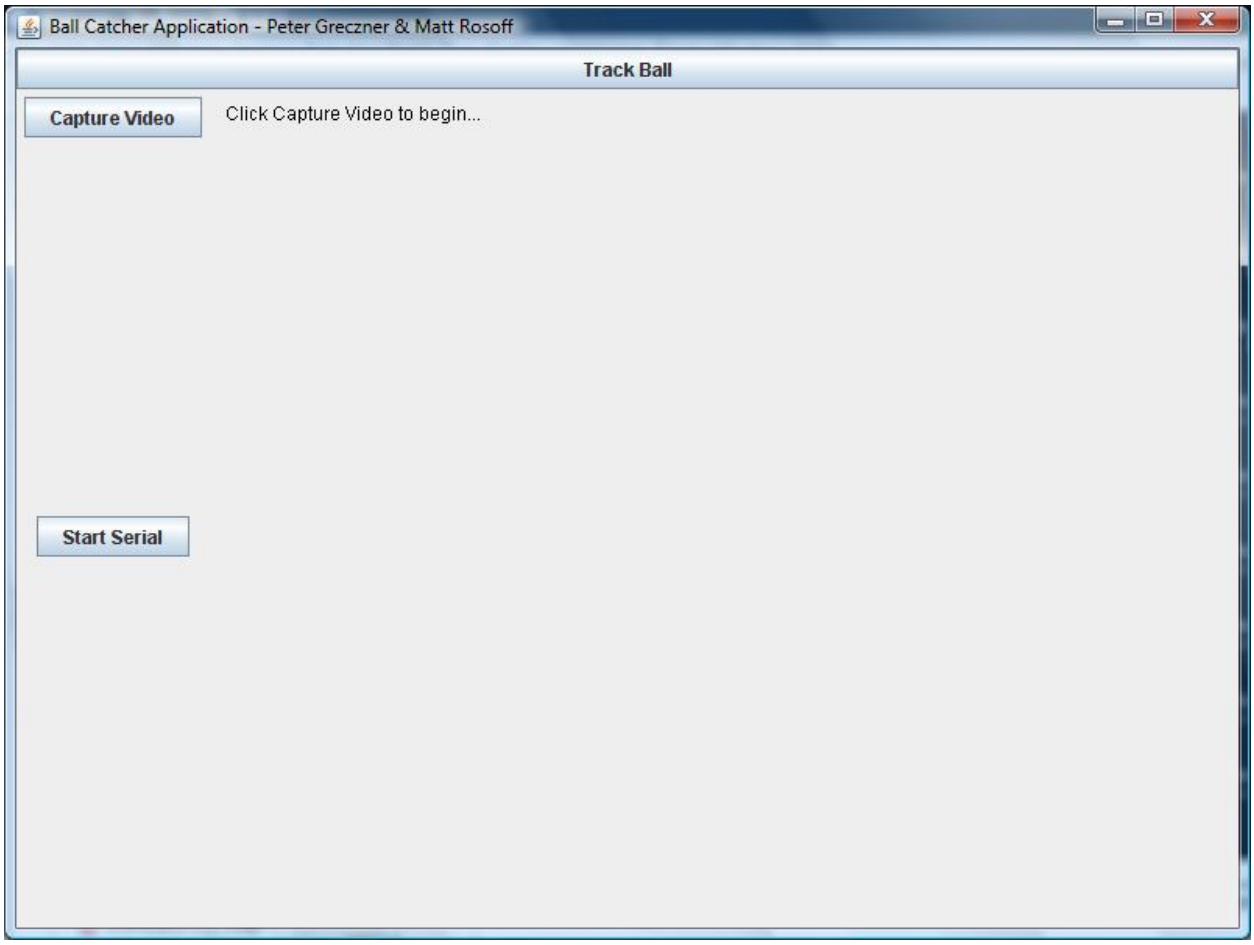
The original plan, shown above, included a pully-driven cart operating on aluminum guide rails. We revised these plans, as shown below, to use a sliding, ball-bearing, track. The ball is placed at the top of the ramp, and rolls down the ramp where it is caught, or tracked, by the cart. Attached are more detailed plans, including dimensions, of the final project.

Background (Black Matte-Finished Acrylic):

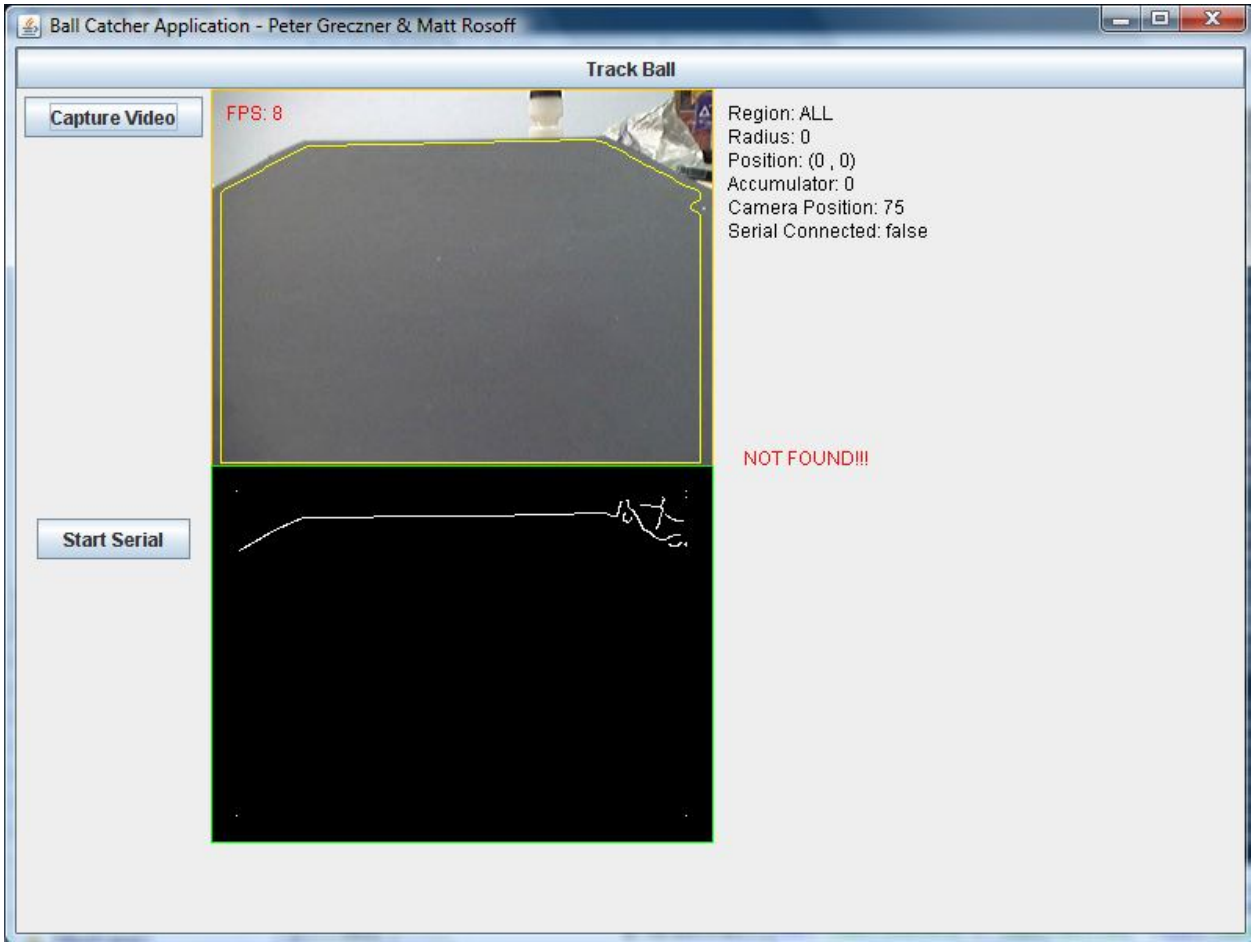
The background, also the ramp, is a sheet of black acrylic. The color and finish were chosen to make the problem of object detection simpler. However, the costly piece of plastic turned out to be far less helpful that previously imagined. We chose to use a black ramp so that the white ball would stand out significantly, and we could implement a simple thresholding algorithm to detect the bright ball. Unfortunately, shadows and uneven lighting made this approach unfeasible, so our choice of a black background made the problem only slightly easier. The matte finish was intended to attenuate

reflections. However, what we found was the matte finish simply blurred reflections. So, rather than seeing reflections, there would be bright spots, or dark spots, on the background. Again, the matte finish was an improvement, but not enough of one to significantly simply the tracking algorithm. A painted piece of wood would have worked equally as well, and would have saved us a significant amount of money.

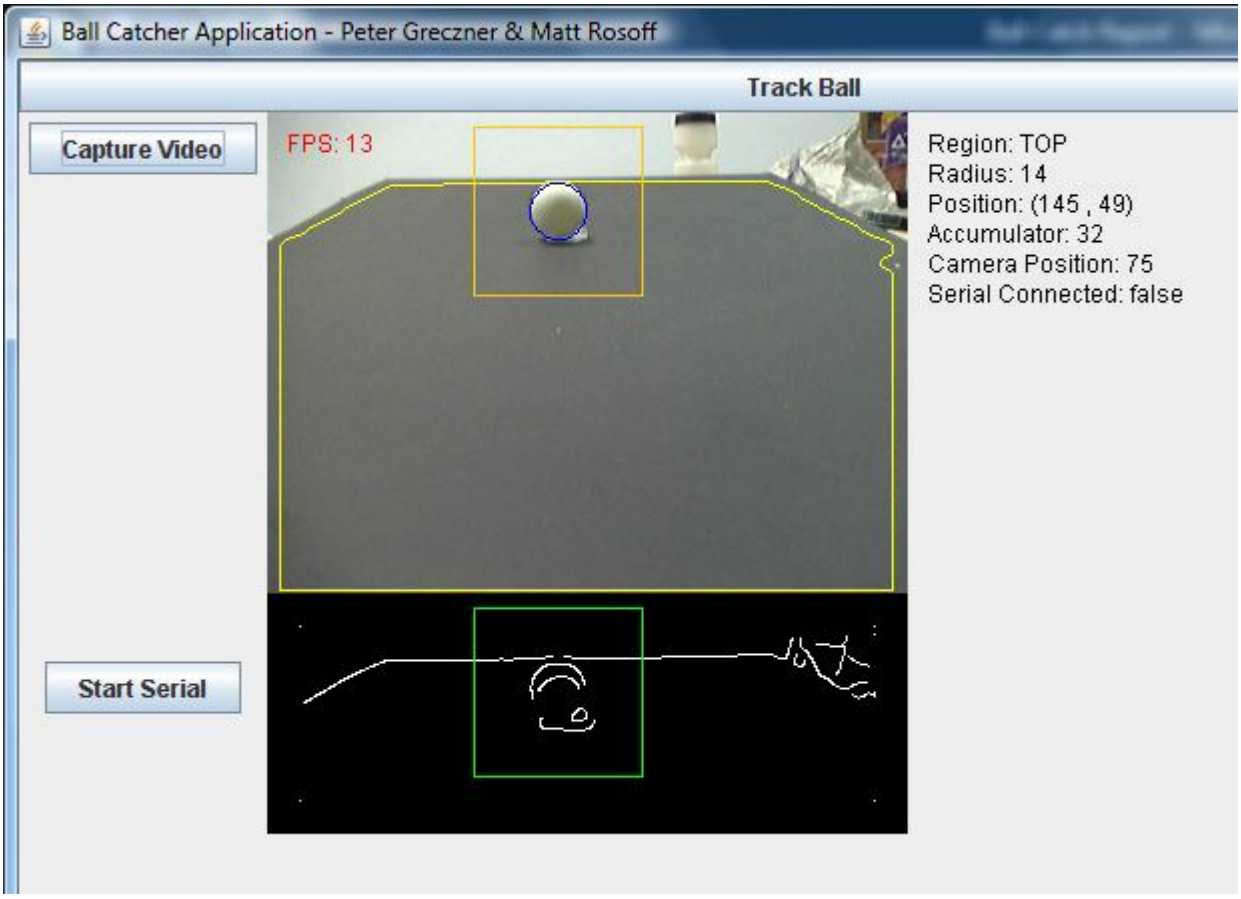
Images



This is an image of the GUI that we wrote.



This is the main image of our GUI. It shows the status of a few important facts. Right now it is processing on the entire image (Region = ALL). The serial communication is not connected, and the ball has not been found. The Camera Position is the position at which we will set the camera when we establish serial communication. In our setup, 75 means center. Also notice the yellow polygon that outlines what our algorithm believes to be the black sheet.



This image shows what the processing looks like when we have found a ball. There are a few interesting things to notice in this new image. First see how the region specified is now TOP. This shows that we have found the ball in the TOP region and for now, that is the only region worth processing on. Also, the green and yellow squares indicate what region we are performing our CHT on. The blue circle, is indicating where the CHT has determined the ball is and with what radius. This information is also printed on the right hand side of the window. One important thing to also notice, is that the FPS has now risen to 13 per second. This is due to our optimization of the region for the Canny edge detection and for the CHT.

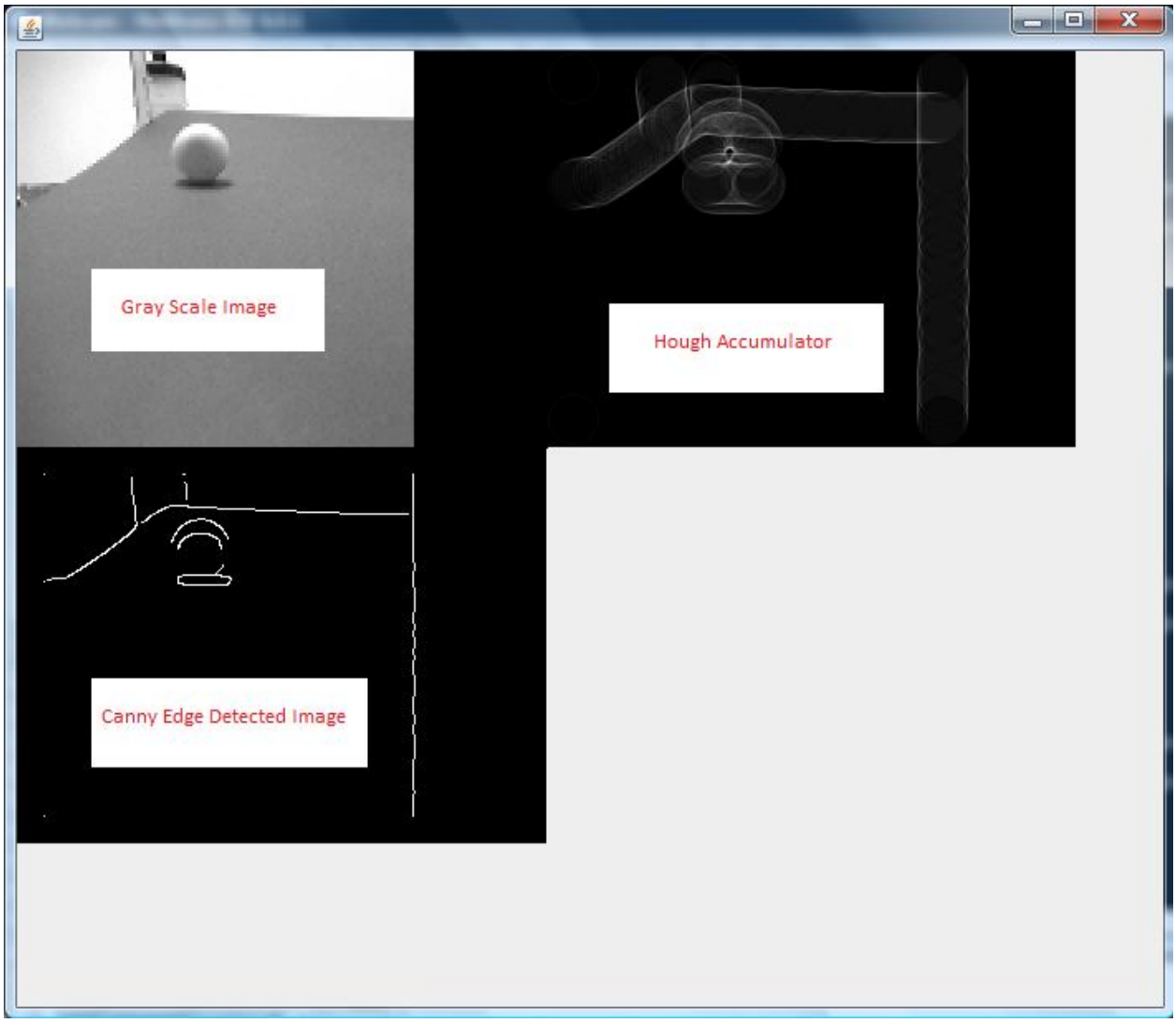
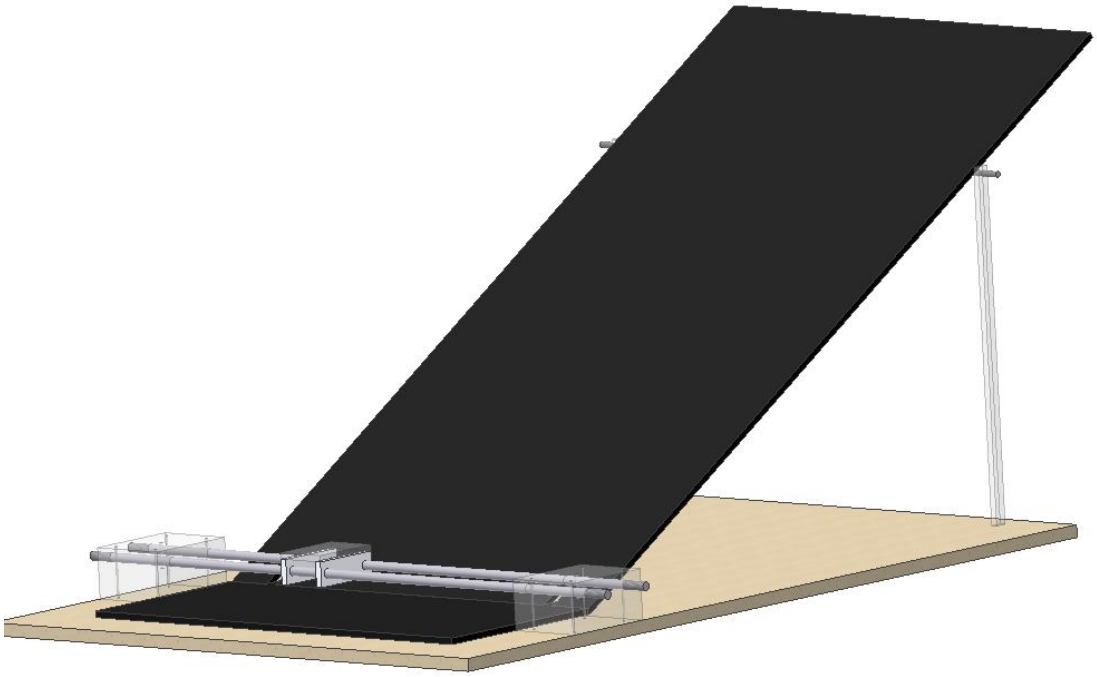


Image showing the gray scale input, the edge image, and the Hough Accumulator. Notice on the Hough accumulator the center region where the concentration of pixels is much higher than in other regions. This indicates that we are most likely looking at a circle of the inputted radius.

Depicted is a rear image of the mechanical setup.



Depicted is an isometric view of the mechanical setup, seen from the front.