Design and Construction of a Vision-Based Golf Simulator

Swarthmore College E90 Project
Authors: Kyle Knapp and Price Ferchill
Faculty Advisor: Professor Erik Cheever

Swarthmore College
Department of Engineering
May 9th, 2014

Abstract

For this project, an affordable computer-vision based golf simulator was designed and constructed. The stroboscopic photography technique was implemented to capture high-speed images of golf balls being struck. The images were processed using code written in MATLAB to determine the initial launch conditions of the golf balls. The initial conditions were used to simulate the golf ball's flight, and the results of the simulation were displayed graphically on a computer monitor in front of the golfer. Our simulator was capable of capturing clear images of golf balls with initial velocities of up to 170 mph and spin rates of 10,000 rpm, and it was also able to accurately predict the carry distance of a golf ball to within twenty percent error. However, the accuracy of the simulation was highly dependent on the quality of the images the system captured: only 29 percent of the attempted simulations captured images that were high enough quality to produce accurate results. Furthermore, our simulator was not able to accurately predict the ball's curvature to the left or to the right due to the fact that we were limited by our budget to using a one-camera system. Ultimately, this project showed that it is possible to design and build an accurate computer-vision based golf simulator with minimal equipment. With a few additions to our system and a small increase in the budget, it would be very feasible to produce an accurate and easy to use computer-vision based golf simulator that is affordable for amateur golfers.

Table Of Contents

Abstract	2
Table Of Contents	3
Introduction	4
Report Organization	5
Project Purpose and Goals	5
Overview	6
Stroboscopic Photography	7
Proof Of Concept	8
Flash Design and Construction	9
Camera Selection	13
Flash/Camera Automation	14
Hardware Component Packaging	15
Testing Area Setup	16
Image Processing	17
Golf Ball Flight Simulation	26
Results	30
Discussion	33
Future Work	36
Conclusion	38
Acknowledgements	39
References	39
Appendices	40

Introduction

Golf simulators are extremely useful tools for competitive golfers. They can be used to help a golfer collect data on their ball flight, perfect the mechanics of their swing, optimize their equipment, or practice indoors during inclement weather. However, many golf simulators are extremely costly due to their use of expensive equipment, complicated installations, and the large amount of floor space required to safely swing a golf club indoors. These simulators are not realistically affordable for the average consumer.

Most golf simulators use the same basic method for providing a golfer with a virtual representation of their ball flight. First, data is collected on the golf ball directly after the golfer strikes it with their club. After the data is captured, it is analyzed to determine the golf ball's initial launch conditions. Then, the initial conditions are fed into an algorithm that simulates the golf ball's flight. A visual representation of the golf ball's flight is usually projected in front of the golfer to give the illusion that they actually struck a golf ball into the distance. Sometimes, statistics concerning the golf balls flight (distance traveled, maximum height achieved, amount of curvature, etc.) are provided to give the golfer more concrete feedback.

While most golf simulators function in the same general way, many different methods have been used to collect golf ball launch data. Some simulators use banks of infrared sensors to track the golf ball through the first few inches of its flight. While this method is generally relatively low cost (infrared based units like the popular Optishot golf simulator usually cost on the order of hundreds of dollars), it does not tend to produce very accurate results. A far more accurate method uses high-speed cameras to capture images of the golf ball at different points in time over the first few inches of its flight, which are then compared to determine the golf ball's initial launch conditions. However, this increased accuracy comes with a price; camera based systems tend to be far more costly due to the expense of high-speed photography equipment. This method also sometimes requires cameras to be permanently installed, which could prove to be inconvenient if a golfer wants to move the location of their simulator, or if they want to use their simulator outdoors. Some other highly accurate golf simulators use radar arrays to collect golf balls' launch data. These systems tend to be much smaller than the infrared

and camera based simulators, but they are still relatively expensive: different models of the Trackman radar based simulator range from $16,000 to over $45,000.

Report Organization

The remainder of this report is broken into several sections. First, we discuss our project's purpose and goals, and we provide a broad overview of the system we implemented. Then, we discuss the hardware and software design processes in more detail before presenting a summary of the results produced by our simulator. Finally, we analyze the results in the discussion section, and propose ways that our golf simulator could be improved with future work.

Project Purpose and Goals

The purpose of this project is to design and build a golf simulator that combines the best qualities of the different types of simulators discussed above; we wish to create a camera based golf simulator that is accurate, affordable, and portable. To objectively evaluate our success in pursuing this purpose, the following design goals were set for our simulator:

1. The simulator should accurately predict the carry distance and curvature (from right to left or from left to right) of golf shots to within twenty percent error.
2. The simulator should have the ability to automatically capture clear, usable images of golf balls traveling up to 170 mph with 10,000 rpm of spin.
3. The simulator should be built on a budget of $500.
4. The simulator should be easily transported and set-up by a single person.

The values of 170 mph and 10,000 rpm were not chosen arbitrarily. We researched PGA Tour statistics and found that PGA Tour professionals have an average ball speed of 170 mph with their drivers, and an average spin rate of 10,000 rpm with their wedges (the golf clubs that tend to produce the highest ball speeds and spin rates). Since most amateur golfers are much less capable than professionals, they are not capable of producing ball speeds or spin rates this large in magnitude. We assumed that if our simulator was

capable of capturing the initial conditions produced by professionals, it would be more than capable of capturing the initial conditions produced by most amateurs.

Overview

The golf simulator design presented here uses the same basic process as commercially available camera based golf simulators. Our simulator automatically captures data in the form of high-speed images depicting the first few inches of a golf ball's flight. The images are then processed to calculate the ball's initial launch conditions (initial velocity, launch angle, and spin rate), which are in turn used to simulate the golf ball's flight. A visual representation of the simulation results is then displayed in front of the golfer.

Figure 1 below depicts a flow chart that summarizes the major technical elements of our golf simulator, and the sequence in which they are implemented. The entire process contains three major sections: high speed automatic image capture, image processing, and golf ball flight simulation. The design process for each of these sections is discussed in greater detail below.

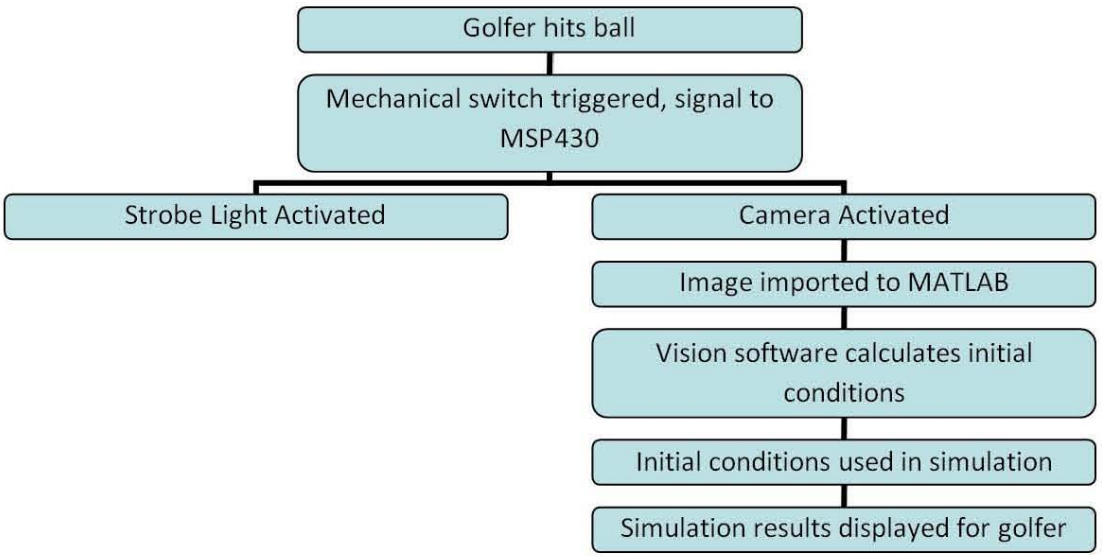


Figure 1: Flow chart depicting the major technical elements of the golf simulator design presented in this report

Stroboscopic Photography

At first glance, capturing high-speed images of a golf ball's flight may seem like a trivial issue; a high-speed camera could be used to easily take the necessary pictures. However, high-speed cameras are extremely expensive. Even the least expensive models can cost thousands of dollars, well over the $500 budget specified in this project's design goals. Because the cost of high-speed cameras was found to be prohibitive, alternative methods of capturing high-speed images were explored. After performing extensive research, we determined that the stroboscopic photography method could be used to capture high-speed images of the first few inches of the golf ball's flight.

In stroboscopic photography, a camera's shutter is left open for the entire duration of an event, and a flash is used to illuminate the subject at specific moments of interest. Each time the subject is illuminated, it is captured through the camera's open shutter. At the end of the event, the photographer is left with a single image depicting the subject once per moment of interest over the course of the event (see Figure 2).

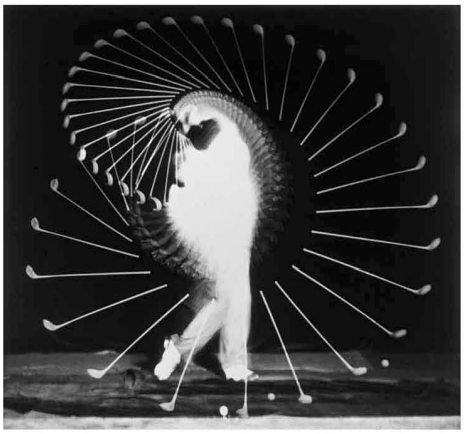


Figure 2: Stroboscopic photograph of a golfer swinging. Image courtesy of http://edgerton-digital-collections.org/galleries/iconic/page/3#hee-nc-38001

Stroboscopic photography is particularly useful in capturing high-speed events because the speed with which images can be captured depends only on the flash. While conventional cameras are limited by the speed with which their mechanical shutters can open and close, stroboscopic methods do not have this problem; it is relatively easy to dictate the length, frequency, and intensity of flashes to capture images extremely quickly.

Minimal equipment is required to implement the stroboscopic photography technique: a digital camera capable of long shutter times, a bright flash, and a timing device are the only necessary hardware. For the purpose of capturing images of golf balls, the camera can be set up parallel to the golfer's target line (just a few inches in front of the ball's initial position), with the flashes mounted next to the camera to provide proper lighting. Based on data collected on PGA Tour professional golfers, the highest spin rates we expect our simulator to have to capture are about 9000 rpm, or 150 Hz.

Proof Of Concept

Before we could begin the design and construction of our camera based golf simulator, we had to prove that the concept of using the stroboscopic photography method to capture high-speed images of golf balls was feasible. To do this, we used a Cannon DSLR digital camera and a stroboscopic tachometer (traditionally used to measure the timing of combustion engines) outfitted with a large flash bulb and an rpm adjustment dial to manually take pictures of golf balls being struck with a seven iron. The tachometer was set to flash at a specific frequency, and the camera trigger was pressed just before the golf club impacted the golf ball. This process was repeated with different flash frequencies and camera shutter speeds until clear images were captured of the golf ball at multiple points in time. After many trials, we found that images of satisfactory quality could be obtained using a flash frequency of 24,000 rpm and a camera shutter speed of 0.25 seconds (Figure 3).

Figure 3: Stroboscopic image of a golf ball being struck by a golf club. Image captured with a Cannon DSLR camera (shutter speed set to 0.25 seconds) and a stroboscopic tachometer (frequency set to 24,000 rpm).

Flash Design and Construction

Once the stroboscopic photography technique proved to be a viable method of capturing high-speed images of golf balls, we had to determine an efficient means of creating precisely timed flashes. While the stroboscopic tachometer was very effective at providing a large amount of light at specific frequencies, it had a few drawbacks that made it ineligible for use in our project. First and foremost, the stroboscopic tachometer had a permanent focal angle that was too narrow for our purposes. It provided sufficient lighting at moments when the golf ball was near the center of the stroboscope's focal beam, but the lighting was too diffuse to create clear images of the ball at other moments in time. Also, the stroboscopic tachometer was an analog device; it would be very difficult to accurately and reliably turn on and off the flash without some kind of digital input.

After exploring other commercially available flash devices such as professional photography flashes and high-frequency LED work lights, we determined that the best and most cost effective way of creating flashes that could be easily controlled would be to simply custom build flash units using arrays of white LED lights and an MSP430 microcontroller.

The MSP430 controlled strobe light was implemented in two segments: C code executed by the MSP430 that controlled the frequency and duration of the LED flashes, and a circuit that flashed the LED lights when it received a voltage from the MSP430.

The C code (Appendix A) was implemented as a simple while loop. Inside the while loop, the voltage of the P1.0 pin on the MSP430 was set to high, followed by a delay, then set to low, and again followed by a delay. When P1.0 was high, the MSP430 provided an output voltage, and when P1.0 was low there was no output voltage. When this toggling action was combined with the circuit that included the LED lights, it caused the lights to flash.

To make sure that the delays after each toggle of the P1.0 voltage were timed very precisely, the __delay_cycles() command was used. The __delay_cycles() command is a function-like operator that is intrinsic to the MSP430's C/C++ compiler. It is designed to occupy the number of clock cycles indicated within the parentheses with absolutely no side effects – much like a "do nothing" command in assembly language. To make the use of the __delay_cycles() intrinsic more intuitive, the MSP430's clock was reset to run at 1MHz. This made it so that each clock cycle took exactly .000001 seconds, or 1 microsecond.

The circuit (Figure 4) was designed such that the P1.0 output pin of the MSP430 was fed into the gate of an IRFZ14 MOSFET transistor. The transistor's drain was connected in parallel to the cathodes of 18 white LEDs via resistors, and the LEDs' anodes were connected to a power supply (Vcc) of 5 volts. The transistor's source and the MSP430's ground were both connected to a common ground. When the P1.0 output pin on the MSP430 was set to high, the transistor's gate received a high voltage, effectively connecting Vcc to ground through the LEDs, lighting the LEDs in the process.

In this design, there were no resistors included between the LEDs and ground to protect them from overheating. The LEDs' data sheet listed their typical forward voltage as 3.6 volts for a forward current of .03 amps. While the 5 volt power source used in our system was larger than the typical forward voltage of 3.6 volts, it was assumed that the relatively short amount of time that the LEDs would be turned on (the pulse widths that turned on the lights were on the order of ten thousandths of a second, and were very short

relative to the period of the flashing) would keep them from overheating. Experimental tests showed that this assumption was correct.

A 1 MΩ resistor was included between the gate of the transistor and ground. This ensured that the gate of the transistor was grounded when it was disconnected from the MSP430, and kept the lights from flashing unintentionally. Also, a 100 μF capacitor was included to stabilize the voltage across the LED lights. The current from the power supply had to travel through a long length of wire that had a small resistance associated with it, which could decrease the voltage across the LEDs. The charge from the capacitor helped ensure that the voltage was a stable 5 volts.

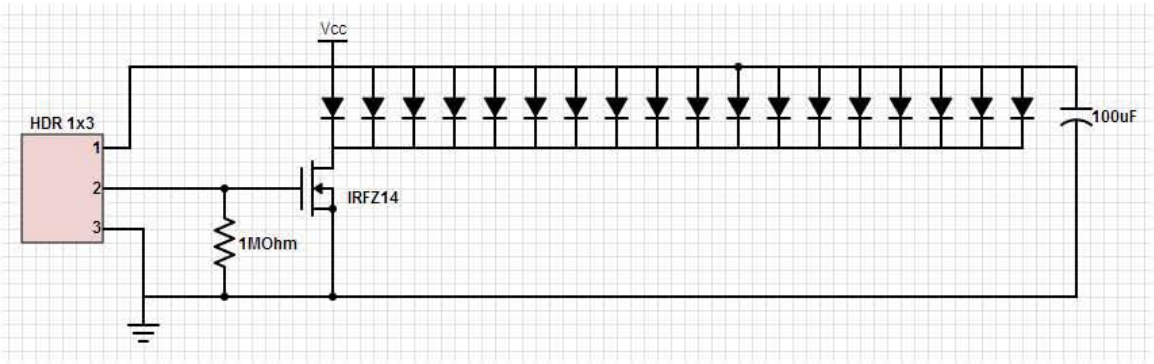


Figure 4: Circuit diagram of the LED flash circuits. On the HDR 1x3 socket, location one was used as the high voltage (5 volts) input, location 2 was the MSP430 signal input, and location 3 was the ground input.

Once the circuits were designed, they were implemented on a breadboard (Figure 5) and tested using the same Cannon DSLR camera used during our initial proof of concept experiments. The lights were programmed to flash at 24,000 Hz with very small pulse duration. After multiple trials, low quality stroboscopic images were obtained using the LED array (Figure 6).

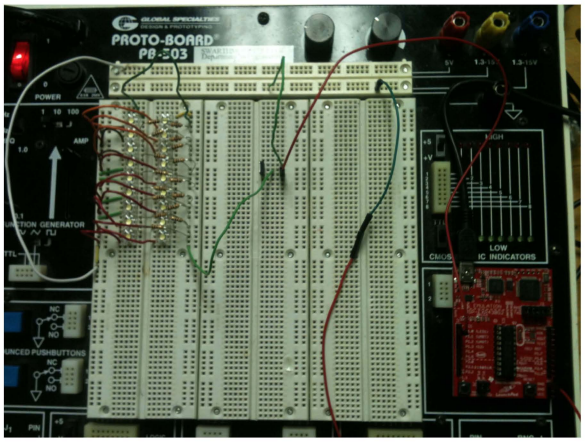


Figure 5: Breadboard implementation of the LED circuit design

Figure 6: Low quality stroboscopic images captured using breadboard implementation of LED circuit design

Most of the poor quality seen in the initial images captured using the LED array was attributed to a poor testing setup; the cardboard, linoleum tiles, and white shoes present in the frame were highly reflective, and caused a significant amount of overexposure that almost entirely washed out some of the golf balls in the image. We were confident that the LEDs would be capable of providing the necessary flashes if the number of lights was increased and our testing setup was optimized.

To make the LED array smaller and more easily manipulated, custom made printed circuit boards were designed in Ultiboard and printed. Then necessary circuit elements were soldered to them (Figure 7). A total of 8 light panels were built (including a total of 144 individual LED lights) to increase the amount of light that was produced, and to create more freedom to focus and distribute the light as necessary to create high quality images.

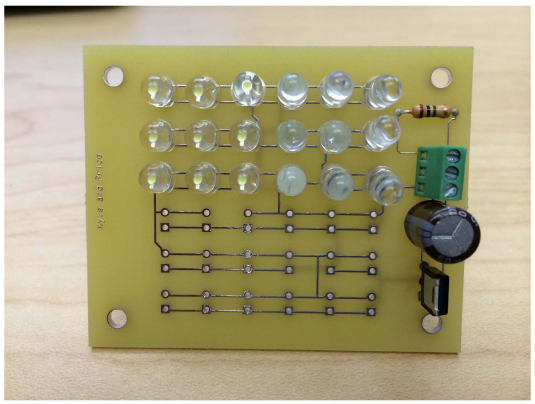


Figure 7: Fully assembled LED flash array.

A power distribution board was built by connecting the high voltage, ground, and MSP430 signal inputs on each of the eight flash boards to the same circuit board (Figure 8). This ensured that all of the flashes shared a common high voltage and ground, and that they were perfectly synched with one another. A voltage amplifier was also included on the power distribution board to increase the voltage that the IRFZ14 transistor received from the MSP430 (Figure 8). According to the IRFZ14 data sheet, 4 volts are required to fully activate the transistor. However, the MSP430 only outputs about 3 volts. Including a voltage amplifier ensured that the transistor was turned on fully each time it received a signal from the MSP430, and the LED lights flashed as brightly as possible.

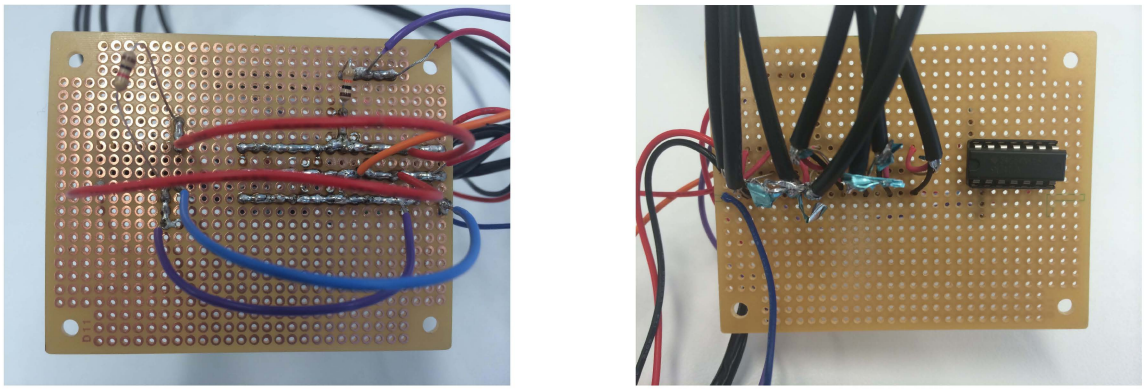


Figure 8: Power distribution circuit top (left) and bottom (right).

Camera Selection

While the Cannon DSLR camera was able to capture accurate images in our proof of concept tests, we were not able to use it in our final design. The Cannon DSLR costs

over $600, which we could not afford on our $500 budget. Also, the Cannon DSLR camera could not be interfaced with the MATLAB image acquisition toolbox that we planned to use to digitally control the camera.

We tried capturing images using a handheld Sony digital camera, but the resolution of the Sony camera was worse than the Cannon DSLR, and it did not allow us to control the exposure time. We estimated the exposure time on the Sony camera was about 0.1 seconds. Using the Sony camera, it was harder to consistently catch the golf ball mid-flight and when we did capture an image, the quality of the image was noticeably worse than the Cannon DSLR.

Next, we tried capturing images using web cameras, which easily interface with the MATLAB Image Acquisition toolbox. The toolbox allowed us to set the exposure rate on the webcam and capture a digital image that could easily be processed further in MATLAB. First, we tested a Logitech C200 Webcam. Surprisingly, we were able to capture images of the golf ball mid-flight using this inexpensive webcam. However, the images were dim and fuzzy due to the camera's low resolution.

After we learned that it was possible to capture an image of the golf ball mid-flight using a webcam and MATLAB, we tested a higher quality webcam: the Logitech C930e HD, which featured an adjustable exposure time, 1080p HD resolution, and a 90 degree field of view for only $130. Using the Logitech C930e HD webcam we were able to capture high quality images of the ball mid-flight.

Flash/Camera Automation

To automate the image acquisition process, a simple mechanical switch was placed between the 5 volt power source and the P2.0 pin on the MSP430 (which was set to be an input) through a 1 kΩ resistor (Figure 9). When the switch was tripped by the golf club (an instant before it made impact with the ball), the 5 volt power source was briefly connected to P2.0, sending a pulse of high voltage to the MSP430 that acted as a signal to flash the lights. This same pulse of high voltage spurred the MSP430 to signal the MATLAB program to take a picture using the webcam connected to the PC via USB.

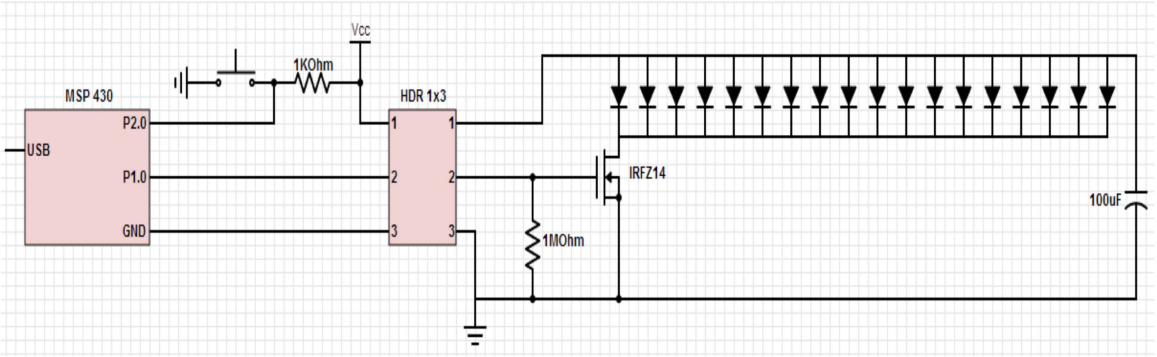


Figure 9: Circuit diagram demonstrating how the mechanical switch was integrated with the MSP430 and LED flash panels. Only one LED flash panel was included here for readability's sake.

Hardware Component Packaging

Once all of the hardware components had been designed and built, they needed to be packaged efficiently. A fixture was built that condensed all of the LED circuit boards into a small volume, and that allowed the lights to be easily manipulated to control the direction and intensity of the flashes. The fixture was made out of a cardboard box lid with holes drilled in it to create a pegboard (Figure 10). Each of the LED boards was "pegged" to the box lid using machine screws, and the wires that connected them to the power distribution board were fed over the top of the box lid. Each time the lights needed to be adjusted or reoriented, the LED boards were simply removed from the fixture and then replaced in different locations. The MSP430, the power distribution circuit board, and the wiring were all housed in the concave section of the fixture (Figure 10).

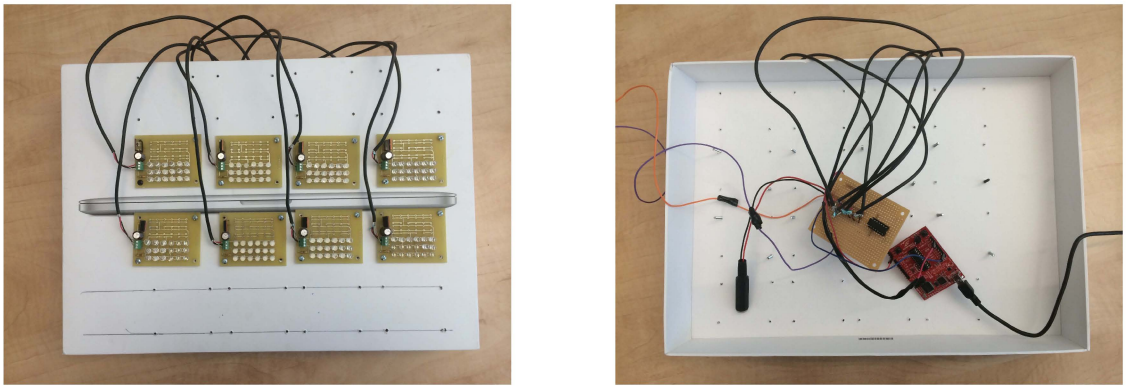


Figure 10: Front and rear views of the LED light fixture.

The mechanical switch and the camera were not attached to the light fixture (Figure 11). The switch was attached to a wooden plank that sat next to the golfer. This

allowed us to easily move the switch to help fine-tune the system's timing, and to repair it in the event that it was broken. The camera, on the other hand, was simply supported by a small cardboard box. Again, this allowed us to easily adjust the camera's location to help fine-tune the picture quality.

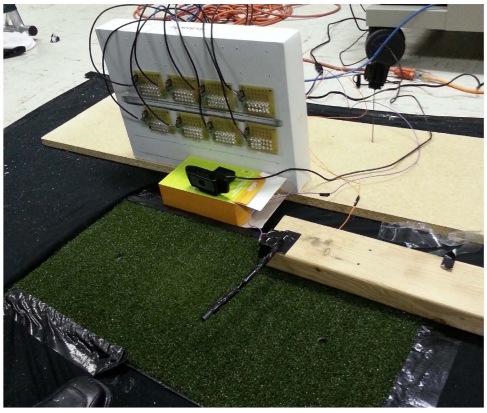


Figure 11: LED light fixture, camera, and switch placed in front of a golfer, ready to capture images.

Testing Area Setup

During the proof of concept experiments and initial LED array design steps, many of the images that were captured suffered from over exposure. The cardboard, linoleum tiles, and other objects in the background of the frame reflected a large amount of the flash's light. Over the course of the multiple flashes that occurred during the camera's long exposure, so much light was reflected that the golf ball images were washed out and appeared very light and blurry.

To reduce the amount of washout in the images, black cotton bed sheets were used to cover the linoleum floor tiles and background of the testing area (Figure 12). The black cotton was very absorbent, and drastically reduced the amount of light reflected from the flashes. To even further reduce the amount of light reflected, the golfer in the images wore black pants and black shoes.

After making these adjustments to the setup, much clearer pictures were obtained that exhibited a very small amount of washout due to overexposure (Figure 13 of the Image Processing section). These images were high enough quality to be used in our image processing software.

Figure 12: View of the testing area after it was covered in light absorbent black cotton sheets.

Image Processing

Once data was captured in the form of digital images, the images were processed to extract the initial launch conditions of the golf ball. A sample image is shown in Figure 13.

Figure 13: Stroboscopic Image of the Initial Flight of the Golf Ball.

In Figure 13, there are two balls in the image. Each ball in the image represents the position of the golf ball when the strobe flashed. The three initial conditions that need to be found via image processing are the ball speed, the spin rate, and the launch angle (Refer to the Golf Ball Flight Simulation section). The first step to obtain these initial conditions is to identify the golf balls in the image. To identify the golf balls, we apply color thresholding techniques to the picture.

Knowledge about how color is represented in Red-Green-Blue (RGB) space is necessary to understand color thresholding. All digital images are composed of tiny individual squares known as pixels. In a color image, each pixel has three values, ranging from 0 to 255, associated to it: a red value, a blue value, and a green value. The specific values associated with each of these color components represent how much of that particular color is present in the pixel. So if the blue component had a value of 0, there would be no blue in the pixel. Likewise if the blue component had a value of 255, blue would contribute significantly to the color of the pixel. It is the combination of these RGB values that form the specific color of the pixel. Figure 14 depicts a visual representation of a normalized RGB space where the values are from 0 to 1 as opposed to 0 to 255.

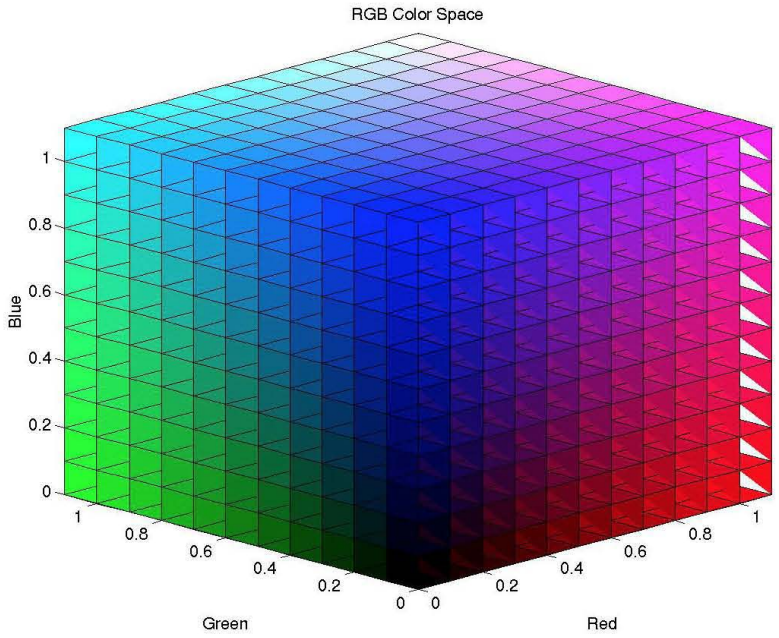


Figure 14: Normalized RGB Space. Image courtesy of: https://www.clear.rice.edu/elec301/Projects02/artSpy/color.html

As seen in Figure 14, a pixel that has a value of 0 for all three components is black. On the other hand, a pixel that has a value of 255 (a value of 1 in Figure 14) for all three components is white. Finally, a color like red has a relative high red component while having relative low green and blue components.

Color thresholding involves choosing a particular color and zeroing out all pixels that are not close to the desired color with respect to the RGB color space. To apply color thresholding to identify the golf ball, it is essential that the color of the golf ball is unique from the rest of the colors in the background of the image. As seen in Figure 13, to obtain this uniqueness in color, we first blacked out the entire background and colored one hemisphere of the golf ball red. This simplified the problem of identifying the golf ball in the image to identifying the color red in the image.

The classic way to color threshold an image is to threshold based on Euclidean distance in the RGB space. In other words, you calculate how far the color of a pixel is from the desired color in RGB space. Then, the pixel is zeroed out if the distance is greater than a specific, predetermined distance. Otherwise, if the distance is less than or equal to the predetermined distance, the pixel value is retained or set to some default value. However, this thresholding technique was not usable in our project due to the fact that the red on the ball varied significantly in intensity. For example, two golf balls can look red (i.e. have a high red component relative to the blue and green components), but one ball may have a significantly higher red value than the other, causing the ball with the higher red value to be a brighter red than the other red. The only way to accommodate for this discrepancy is to make the distance threshold larger. However, this increases the chance that non-red colors are retained.

To overcome the intensity differences of the golf balls, we devised a different color thresholding strategy. First, we thresholded based on the ratio between the value of the red component and the value of each of the other color components. In order for the value of a pixel to be retained, the red to blue and the red to green ratios need to be greater than a specified ratio, 1.5. Ultimately, this ensures that the pixel being retained is primarily red. We then apply a second threshold to the image based on the intensity of the red pixel component. In order for the pixel to be retained, its red component must be

greater than a specified value. The purpose of performing this threshold is to ensure that the darkly shaded red pixels are eliminated from the image. This is appropriate because the strobe light pointed at the golf balls will cause the red of the golf balls to have a high intensity, and therefore, any dimly lit red is probably part of the background and needs to be eliminated. Figure 15 depicts the effects of applying these two color thresholds to Figure 13. Note that the black pixels are the pixels thresholded out and the white pixels are the pixels retained.

Figure 15: Applying Our Color Thresholding Technique to Figure 13.

As seen in Figure 15, our thresholding technique successfully isolated the red hemispheres of the golf balls.

After applying the thresholds to the image, a morphological operator, also known as dilation, was applied to the image. The operator expands the white regions of the image by making all of the pixels around the white pixels white. The purpose of performing this operation is to smooth out the golf ball's hemisphere and fill in any missing pixels inside of the hemisphere. Figure 16 depicts Figure 15 after applying the dilation operation.

Figure 16: Applying Dilation to Figure 15

As seen in Figure 16, the shape and equators of the balls are more pronounced. Since the hemispheres of the golf balls are the only objects visible in the image, we can now extract the initial launch conditions. The most important feature in determining the initial launch conditions is the equator of the ball marked by the edge of the red hemisphere. If the location of the equator can be quantified, then the ball speed, spin rate, and the launch angle can be calculated.

To begin quantifying the location of the equators, the Canny edge detection algorithm is applied to the image in Figure 16 to detect the edges in the image. It accomplishes this goal by looking at the image gradients in the image. An image gradient represents the direction in which a pixel experiences the greatest change in intensity. By searching where this change in intensity is the greatest and how many neighboring pixels share a similar gradient, the pixels that are on edges can be isolated. Figure 17 shows the output of applying the Canny edge detection algorithm on Figure 16.

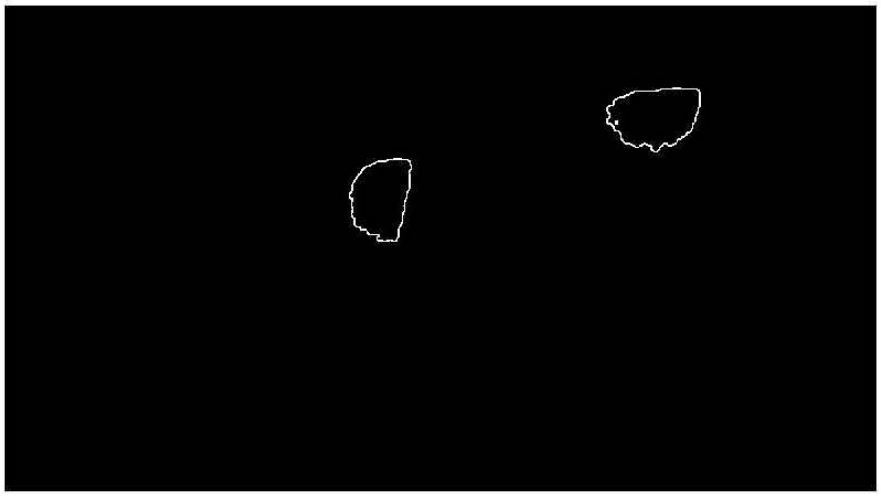


Figure 17: Applying the Canny Edge Detection Algorithm to Figure 16.

As seen in Figure 17, the algorithm successfully isolates the edges of the golf balls. In order to extract the location of the equator, an operation called the Hough transform is applied to the image. The Hough transform determines where the straight lines are in the image. If used properly, the Hough transform will identify the equators of the golf balls since they appear as nearly straight lines. Furthermore, the endpoints of these identified lines can be determined. Thus, we are now able to extract the endpoints of each of the equators of the golf balls. The result of using the Hough transform is shown in Figure 18.

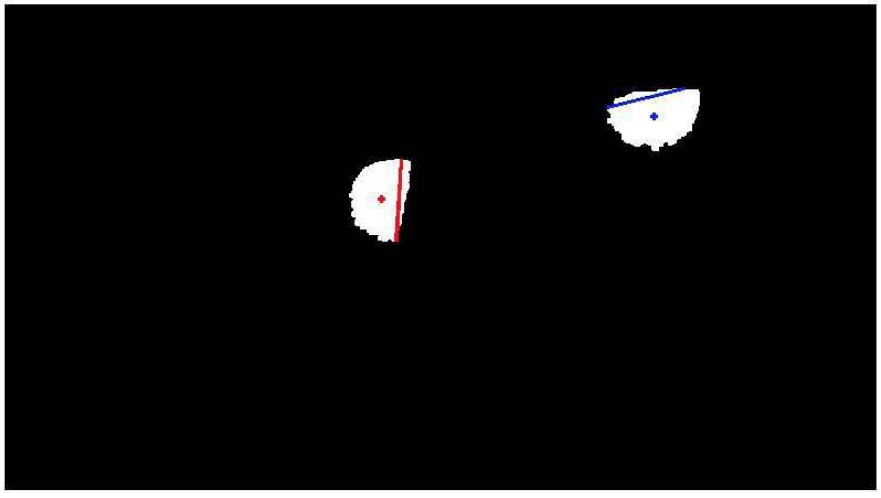


Figure 18: Identified Equators and Centroids of the Hemispheres of the Golf Balls.

In Figure 18, the red and blue lines mark the equators of their respective balls. The red and blue dots mark the centroids of the respective hemispheres of the balls. The process in which these centroids are identified is described later in this section of the report.

With the known location of the equators, we can begin calculating the initial conditions of the golf ball's flight. In order to calculate the speed of the golf ball, we must determine how far the ball has traveled between flashes of the strobe light. We know for a fact that all golf balls have a diameter of exactly 1.680 inches. We are also able to determine the diameter of the golf ball in terms of pixels because we know the endpoints of the lines representing the equators of the golf balls. By averaging the length of these equatorial lines, we can obtain an accurate measurement of the diameter of the golf ball in pixels. Ultimately, this allows us to convert from pixels to inches.

Next, we need to determine the amount a ball has traveled between two flashes of the strobe. Using the midpoints of the equators of two adjacent golf balls, we calculate how many pixels the ball has traveled between flashes. This allows us to calculate how many inches the ball has traveled using our conversion from pixels to inches. In Figure 19, the green line represents the line we measure to determine how far the bar travels between flashes of the strobe.

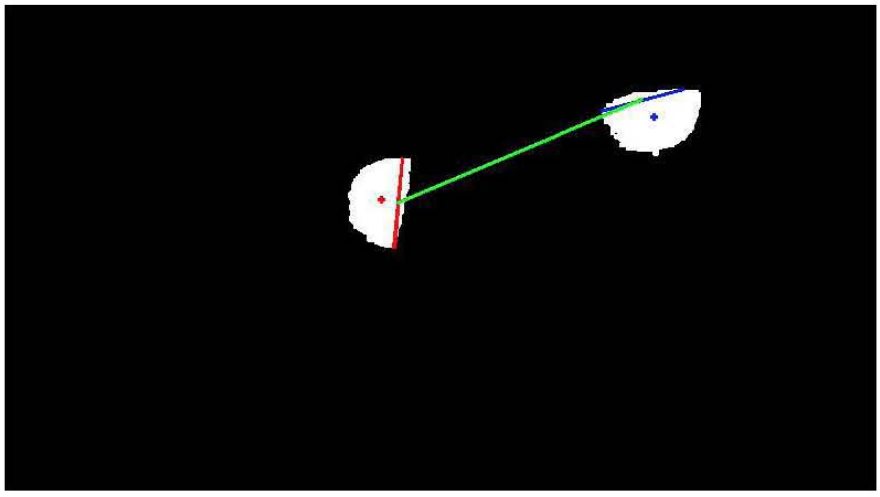


Figure 19: Calculation of Ball Speed.

Since we know the time elapsed between flashes, we can determine the speed of the golf ball by dividing the distance traveled by the time elapsed between flashes.

To calculate the spin rate of the golf ball, we must determine how much the golf ball rotates between flashes of the strobe, and then divide the change of angle by the rate of the LED flash. The main obstacle in determining how much the ball rotates is that we cannot solely rely on the equator of the golf ball. If we ignore where the hemisphere is relative to the equator, the rotational position of the golf ball becomes ambiguous and the amount of rotation can be off by 180 degrees. To account for where the hemispheres are located, blob detection techniques are applied to Figure 16 that identify and calculate the centroids of the hemispheres in the image as seen in Figure 18. To determine the angle of the equator, we find the normal to the line segment of the equator, and then find the angle the normal made with the horizontal of the image. When choosing which normal to use (there are normals for both sides of the line segment), the normal that was closest to the vector created between the centroid of the hemisphere and the midpoint of the equator line segment is used. We then calculate how many degrees the normal changes between balls to calculate how much the ball rotated. Figure 20 and Figure 21 depict the correct and incorrect measurement of the change of angle respectively.

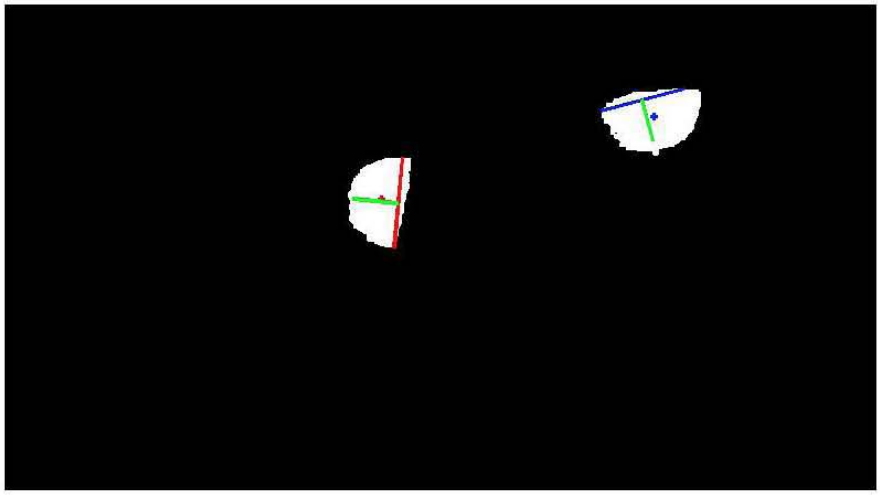


Figure 20: Correct Measurement of Change of Angle.

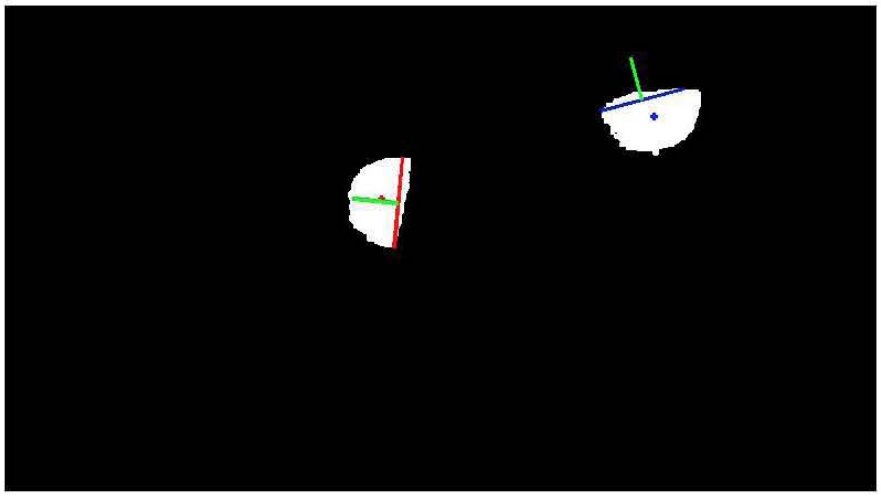


Figure 21: Incorrect Measurement of Change of Angle.

In Figures 20 and 21, the green lines represents the normals used in the measurement. As seen in Figure 20, the normals used are the ones pointing toward the centroid of their respective hemispheres and thus produces the correct change of angle (a little over 90 degrees). On the other hand, in Figure 21, the normals used are inconsistent: the first ball's normal points toward the hemisphere and the second ball's points away from the hemisphere. As a result, the change of angle measured will incorrectly be a little over 270 degrees.

The last initial condition to calculate is the launch angle. To calculate launch angle, we determine the vector between two equator midpoints of adjacent golf ball images. Then we calculate the angle between that vector and the horizontal in the image. This measurement is depicted in Figure 22.

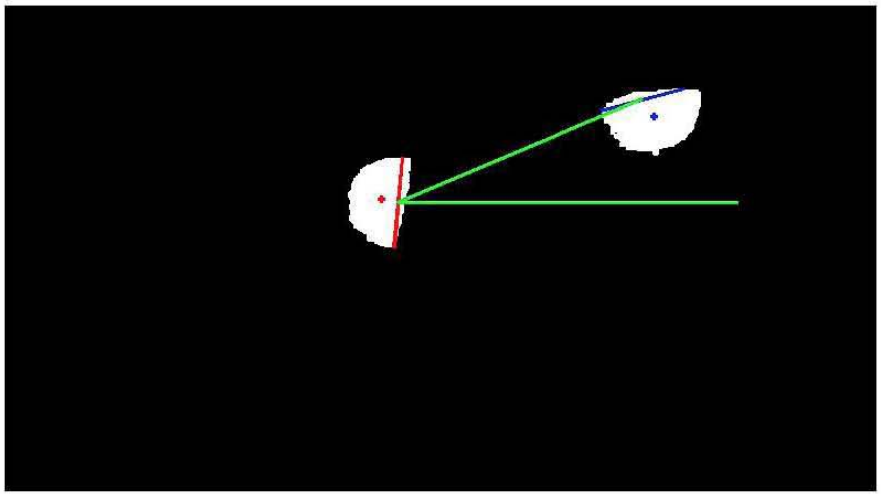


Figure 22: Calculation of Launch Angle.

In Figure 22, the green lines represent the two vectors used in calculating the launch angle.

By determining all three of these initial conditions, we are then able to simulate the flight of the golf ball. The MATLAB code used to implement the image processing part of this system is located in Appendix B of the report.

Golf Ball Flight Simulation

Once the images have been processed, the golf ball's calculated initial launch conditions are fed into a simulation code. This code is implemented in two parts. The first part models the forces acting on the golf ball while it is in flight. The forces acting on the golf ball are depicted in Figure 23.

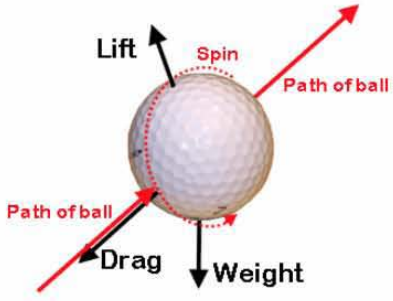


Figure 23: Forces Acting on a Golf Ball in Flight. Image courtesy of: http://www.tutelman.com/golf/design/swing3.php

Based on Figure 23, there are three forces acting on the golf ball midflight: the force due to drag, the force due to lift, and the force due to gravity. By summing these forces and then dividing by the mass of the golf ball, we determine the acceleration of the golf ball.

$$\vec{a} = \frac{F_{drag} + F_{lift} + mg}{m}$$

We can then isolate the x and y components of the acceleration vector. Note that x represents the direction in which the ball travels forward, and y represents the direction in which the ball travels up in the air. The x and y components of acceleration are determined to be:

$$\ddot{x} = -\frac{\rho A}{2m}\left(\dot{x}^2 + \dot{y}^2\right)\left(C_D \cos(\alpha) + C_L \sin(\alpha)\right)$$
$$\ddot{y} = \frac{\rho A}{2m}\left(\dot{x}^2 + \dot{y}^2\right)\left(C_D \cos(\alpha) - C_L \sin(\alpha)\right) - g$$

Where ρ is the density of air, A is the cross-sectional area of the golf ball, C_D is the coefficient of drag, C_L is the coefficient of lift, α is the launch angle, and g is the acceleration due to gravity. Note that our model does not account for how much the ball curves to the right or left of the intended target due to sidespin because the system design implemented here is not capable of capturing images in more than one plane. Thus, we do not include acceleration in the z direction, which accounts for the ball traveling to the left or right. It is also important to note that in the equations for acceleration, the spin rate of the golf ball is not directly used in the equation. However, the coefficients of lift and drag are affected by the spin rate along with the ball speed. In research performed by Bearman and Harvey, coefficients of drag and lift for a golf ball in flight with respect to its speed and spin rate are calculated. We use their results to look up the coefficients of lift and drag for specified ball speeds and spin rates.

Using the formulas for acceleration we are able to simulate the flight of the golf ball using the Euler method, which can be described by:

$$q(t + dt) = q(t) + \dot{q}(t)dt$$

where q is some vector representing quantities being simulated. We use the Euler method for our model because it is simple, and we take small enough increments of dt such that the error in the simulation will be very small or negligible. For our model, the vector q is described as:

$$q(t) = \begin{cases} x(t) \\ y(t) \\ \dot{x}(t) \\ \dot{y}(t) \\ \alpha(t) \\ \omega(t) \end{cases}$$

where x is the position the ball has traveled forward, y is the height of the golf ball's flight, α is the current launch angle, and ω is the spin rate. The derivative of q is described by:

$$\dot{q}(t) = \begin{cases} \dot{x}(t) \\ \dot{y}(t) \\ \ddot{x}(t) \\ \ddot{y}(t) \\ tan^{-1}\left(\frac{\dot{y}(t)}{\dot{x}(t)}\right) \\ 0 \end{cases}$$

In our model the spin rate does not change in flight, so the angular acceleration term is always zero.

In order to simulate the flight of the golf ball, we continuously loop performing the Euler method for small time steps until the ball hits the ground. For aesthetic purposes, we simulate the ball bouncing after it hits the ground. To simulate the bounce, we multiply the velocity in the x and y direction by a small constant (e.g. 0.3) each time the ball hits the ground until the ball stops all together.

Once the flight of the golf ball is simulated, we display the real-time flight of the golf ball in our virtual representation of the world. Figure 24 displays screen shots of the simulation.

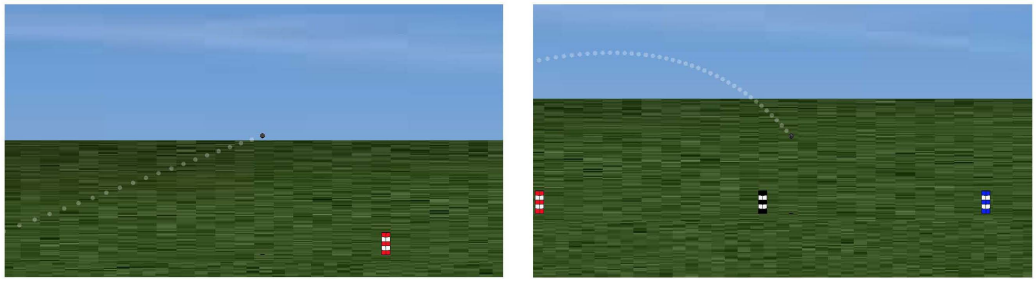
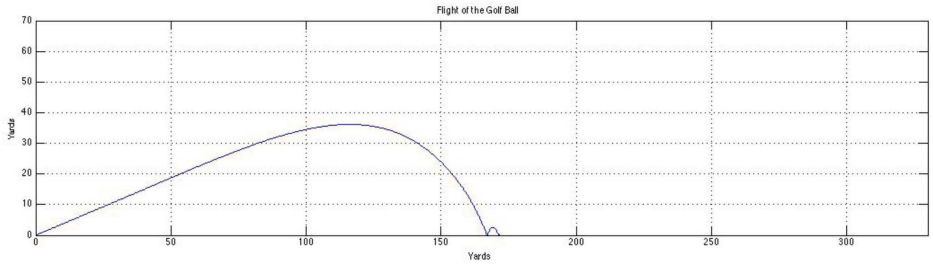


Figure 24: Images of the golf ball flight simulation.

As seen in Figure 24, the flight of the golf ball is captured from a side view. The colored cylinders in the picture represent yardages. For example, the red marker is 100 yards, black is 150 yards, and the blue is 200 yards. This gives golfers a good indication of where the ball is going to land while watching the simulation. Once the simulation is complete, it plots the entire flight path of the golf ball along with important statistics about the shot. Figure 25 depicts this part of the simulation.



Ball Speed (mph): 120.0
Spin Rate (rpm): 7000.0
Launch Angle (degrees): 20.0
Carry Distance (yards): 167.0
Total Distance (yards): 171.6

Figure 25: Sample End Screen of Simulation.

Figure 25 displays the three initial conditions, the carry distance, and the total distance. The carry distance is the measurement of how far the ball flies before it hits the ground. The total distance is the measurement of how fall the ball travels, which includes the bounce after the ball hits the ground. Typically, golfers are more concerned with carry distance because the bounce is highly dependent on the conditions of the landing surface, such as if the grass is wet or if the ball lands on a hard surface. As mentioned before, the inclusion of the bounce in the simulation is mainly for aesthetics because most golf shots

will experience some kind of a bounce when they land. Therefore, the carry distance is the distance used to interpret and compare results throughout this paper. Note that the MATLAB code used to produce the simulation is located in Appendix D of the report.

Results

In this section we will present results for the image processing and simulation components of our system individually. Then, we will present results pertaining to how our system worked as a whole.

Image Processing

First, the accuracy of our image processing software was tested using the image in Figure 13, in which a golfer hit a golf ball using a seven iron. Table 1 displays the initial conditions that were produced by the image processing software, and compares them to the average initial conditions for seven irons hit by PGA Tour professional golfers.

Comparison of Golfer to PGA Tour Golfer		
	Figure 13 Golfer	**PGA Tour Average**
Ball Speed	121 mph	120 mph
Spin Rate	6870 rpm	7097 rpm
Launch Angle	22.4 degrees	16.3 degrees

Table 1: Seven iron initial conditions of ball hit in Figure 13 vs. PGA Tour average.

The golfer in Figure 13 is known to be an accomplished player and would expect to have initial conditions that are similar to the average PGA Tour professional, as seen here. If anything, our image processing software produced a slight overestimate of the golfer in Figure 13's ball speed and launch angle; while he is an accomplished golfer, he is still not as accomplished as a PGA Tour professional, and would expect to have slightly lower ball speed and launch angle. However, we concluded that this error was most likely due to poor image quality, and not our image processing software. Ultimately, we concluded that the image processing component of our system was fully capable of providing accurate initial conditions given a high quality image (i.e. there are two clear balls in the image).

Simulation

Next, we tested the accuracy of our simulation code. We took the average initial conditions for a variety of clubs hit by PGA Tour professionals and ran them through the simulator. We then compared the simulated carry distance to the known average PGA Tour carry distance for each specific club. The data is displayed in Table 2.

Club	Ball Speed (mph)	Spin Rate (rpm)	Launch Angle (degrees)	Simulated Distance (yards)	Average Distance (yards)
Driver	167	2686	10.9	269	275
3 Iron	142	4630	10.4	215	212
5 Iron	132	5361	12.1	193	194
7 Iron	120	7097	16.3	169	172
9 Iron	109	8647	20.4	147	148

Table 2: Comparison of PGA Tour Distances to Simulated distances.

The simulated distances using the average PGA Tour initial conditions are very close to the average PGA Tour distances. Note that these average PGA Tour initial conditions did not directly produce the average PGA Tour distances. This is due to the fact that the average carry distances come from data that is actually measured in person, not simulated. We would not expect our simulated results to perfectly match a set of measured data because the measured data is subjected to factors like human error and abnormal weather conditions that our model cannot account for. Based off the results in Table 2, we determined that our simulation code is able to accurately predict the ball's carry distance when hit with a variety of clubs and a wide range of initial conditions.

Entire System

After we determined that the image processing code provided accurate initial conditions (given that it received a high quality image) and that the simulation code accurately predicted the golf ball's carry distance (given it was provided with accurate initial conditions), we measured the accuracy of the image acquisition hardware, the image processing code, and the simulation code working together as a system. First, we hit a number of shots with a variety of clubs to determine the accuracy of the simulation. A sample of simulations that were qualitatively determined to have captured high quality images were hand selected, and the results were compiled in Table 3.

Club	Ball Speed (mph)	Spin Rate (rpm)	Launch Angle (degrees)	Carry Distance (yards)	Expected Carry Distance (yards)	Percent Difference (Actual and Simulated)
3 Wood	151	9,567	8	239	235	1.7
5 Iron	125	10,345	15	185	190	2.6
7 Iron	121	6,867	22.4	165	170	2.9
9 Iron	109	7,785	22	148	150	1.3

Table 3: Sample Simulations using Entire System, with the percent difference between simulated and expected carry distance calculated.

The projected carry distance for each of these simulations was close to the expected carry distance value, which the golfer hitting the shots provided (all proficient golfers know how far they carry each of their clubs, to within a yard or two). Most of the simulated initial conditions were reasonable, except for the spin rates of the 3 wood and 5 iron. We would expect the spin rates of the 3 wood and 5 iron to be much lower than the 7 iron and the 9 iron. However, the images for the 7 and 9 iron were of high quality (e.g. as in Figure 13), while the images for the 3 wood and 5 iron were of lesser quality.

Next, we tested the consistency of our system to see how often the simulator produced reasonable simulations. To test the consistency, we hit 45 shots and recorded whether an image of the ball was captured, whether the image was capable of producing a simulation (i.e. there were at least two balls and they were visible enough such that they were not thresholded out), and whether the simulation was accurate (i.e. the ball flew with 20% of expected carry distance). The results of this test are listed in Table 4.

Qualifier	Percent
Caught an image of the ball.	93%
Able to run the simulation.	78%
Produced a reasonable simulation.	29%

Table 4: Results of the consistency test.

Based on the results from Table 4, we concluded that our system is able to consistently catch an image of the golf ball midflight, and that it is able to produce a simulation based on the captured image a majority of the time. However, we only produce a reasonable simulation a little less than a third of the time. This indicates that the quality of the images needs to be improved because the success of the simulation is highly dependent on the quality of the image.

Discussion

As seen in the results section of this report, the individual system components worked well when they were given quality images and data. The system was able to accurately predict the carry distance of a golf ball traveling up to nearly 170 mph with 10,000 rpm of spin when it was able to capture a high quality image. In fact, all of the percent difference values from Table 3 are less than 3 percent. Moreover, the golf simulator we built could be easily transported and set up by a single individual, and it was affordable. The combined cost of all of the hardware components (including the black cotton sheets used in the setup) was just under $500. However, our system had one major flaw: it was only able to capture high quality images on 29 percent of the attempted simulations. In the following paragraphs, we will discuss some of the issues that kept our simulator from producing accurate results more often.

To provide some context, we begin by analyzing a high quality image. Figure 26 below qualifies as a high quality image for a few reasons. First and foremost, two adjacent balls are clearly visible in the image. Secondly, the red hemispheres are very prominent in the image. This is essential because the more visible the red is in the image, the easier it is for the image processing code to isolate and identify the balls. Finally, the equator is aligned in the image such that it is nearly perpendicular to the camera's line of sight. This is important because the equators are difficult to detect when they are misaligned, as they often appear curved.

Figure 26: Example of High Quality Image.

Most of our inability to accurately simulate (or at times simulate at all) came from not having high quality images. The low quality images that we captured suffered from one or more of the following three problems: poor alignment of the ball's equator, not enough light from the LED flashes hitting the ball, or too much light from the LED flashes hitting the ball.

Figure 27 depicts an example of an image that was heavily affected by poor alignment of the ball's equator. The border between the red and white hemispheres appears curved in the image because the ball was not aligned correctly before the golfer struck it; the equator was not perpendicular to the camera's line of sight. It is important that the equators appeared straight in the images we captured because the Hough transform only detects straight lines. If the balls were clear in the image but their equators appeared curved, the Hough transform had a tendency to choose a different part of the ball to be a straight line. When this happened, the angle between the lines identified by the Hough transform was very different than the angle between the actual equators, which in turn drastically affected the spin rate. Interestingly, the ball speed and the launch angle were not altered very much by misidentification of equators. Since the calculation of the ball speed and the launch angle only depended on the locations of the equator midpoints, and the midpoints of the misidentified equators tended to be very close to the midpoints of the balls' actual equators, misidentifying the equators had very little effect on the ball speed and launch angle. This likely explains the spin rate discrepancy for the 3 wood and 5 iron in Table 3 of the results section, where the spin rate was too high for those clubs even though the launch angle and ball speed were accurate.

Figure 27: Poor Image Quality due to Poor Alignment.

Figure 28 below depicts an example of an image that was taken with too little light from the LED flashes hitting the ball. The red hemispheres of the ball are hardly visible, and they blend in with the black background. The image is so dark that even the white hemispheres appear blue. These types of images were very difficult to accurately simulate because the pixel component values are so low that all or most of the red hemispheres are thresholded out by the image processing software. The simulation was unable to run because it did not detect two balls in the image. Even if the image processing software was able to pick up parts of the ball, the entire equator was not usually identified because only small patches of the hemisphere showed up in the thresholded image.

Figure 28: Poor image quality due to not enough light.

Figure 29 below depicts an example of a low quality image that was taken with too much light from the LED flashes hitting the black background. In these cases, too much light was reflected back to the camera aperture, and the image of the golf ball was washed out of the frame. This was problematic for our image processing code because the balls' pixels become mixed with the background's pixels, making it difficult to identify the red of the golf ball. When these types of images were thresholded, the red hemispheres appeared very patchy, making the Hough transform more prone to identify lines that are not the equators in the image. Ultimately, this error produced discrepancies similar to the misalignment images: the spin rates tended to be too high but the ball speed and launch angles were reasonably accurate.

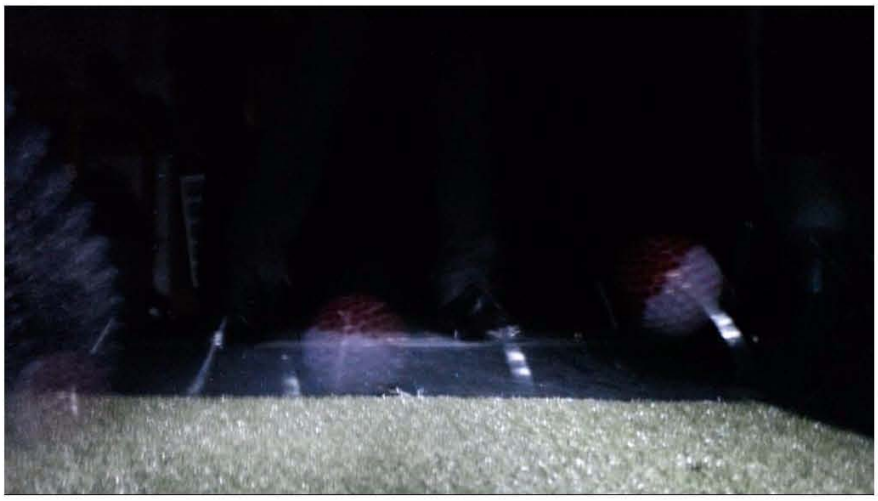


Figure 29: Poor image quality due to too much light.

Poor alignment of the ball's equator, not enough light from the LED flashes hitting the ball, and too much light from the LED flashes hitting the ball were all attributed to a poor setup. Each time the simulator was used we had to setup the light fixture, and re-place the camera and switch. Inevitably, there were slight discrepancies in the setup procedure and our image quality tended to vary a considerable amount. Even when we tried to make fine adjustments in the orientation of the lights or the camera position after our initial set up, it was very difficult to capture high quality images.

It is also important to note that our golf simulator failed to fulfill one of our initial goals; it was not able to account for the ball's curvature from right to left or from left to right. The single camera system that we implemented only gave us data on the ball's first few inches of flight in a single plane. We were not able to find any creative way to extract initial conditions that would allow us to calculate the ball's lateral movement from this limited data set.

Future Work

There are a number of ways in which the golf simulator design presented here could be improved in the future. First and foremost, future versions of this golf simulator should be capable of predicting the golf ball's curvature. To do this, at least one more

camera should be added to the system to detect the ball's side spin. Adding a second camera would not require much extra work; it could be triggered by the same MSP430 signal as the first camera, and it could be mounted over the golfers head. However, adding another $130 webcam to the system would require an increase in the project's budget.

Changes in future designs could also be implemented to increase the quality and consistency of the images captured by our system. A more permanent fixture could be made to house the LEDs, camera, and switch so that the hardware setup is more consistent. This would decrease the amount of variation in image quality due to golfers setting up the system differently. Also, new methods of creating flashes could be explored to create brighter, faster flashes. Or, the placement and orientation of the eight LED flash panels currently in use could be fine-tuned to reduce the amount of washout due to over exposure.

Also, it would be ideal if our simulator could be used in settings that include much more ambient light so that golfers can practice in conditions that more closely resemble what they experience playing on a real golf course in open daylight. Reducing the camera's shutter speed and increasing the intensity of the flash may accomplish this. If the camera had a faster shutter speed, it would reduce the amount of light that enters the aperture, limiting the need to have dark ambient conditions. As long as the strobe light flashes are significantly brighter than the ambient light conditions, imprints of the golf ball may still be captured. However, reducing the camera shutter speed would introduce its own set of problems: the system's timing would become much more important, and would have to be very accurately controlled to ensure that the lights and camera work in synch.

Lastly, the simulation code used in our project could be improved in the future. Spin decay (decrease in spin rate over time due to friction) and other aerodynamic factors could be accounted for to more accurately predict ball flight. Also, some simple additions could be made to the simulation code so that it outputs more data to the user, like the shot's apex height or descent angle.

Conclusion

We were able to design and build a vision based golf simulator capable of accurately predicting the carry distance of a golf ball to well within 20 percent of expected carry distance. Provided the simulator captured high quality images, it was capable of producing accurate results for balls traveling up to nearly 170 mph with 10,000 rpm of backspin. The entire project cost less than $500, an amount that would be affordable for most avid amateur golfers. However, our simulator struggled to consistently capture images of high enough quality to regularly produce accurate results; only 29 percent of attempted simulations produced accurate results. Future work and a small increase in the budget will be required before this design can be used as a viable golf simulator.

Acknowledgements

We would like to thank Professor Erik Cheever for his guidance and support throughout the duration of this project. His knowledge and expertise was instrumental in both the design and construction portions of our golf simulator. Without him, this project would not have been possible. We would also like to thank Ed Jaoudi for his assistance in obtaining parts for our project and his help in coming up with creative wiring solutions. Lastly, we would like to thank the Faculty, staff, and our peers in the Swarthmore Engineering department who have played a major role in our academic development over the last four years.

References

Bearman, P. W., and J. K. Harvey. "Golf Ball Aerodynamics." *The Aeronautical Quarterly* 27.112 (1976): n. pag. Web.

Davidhazy, Andrew. "Introduction to Digital Stroboscopic Motion Photography." N.p., 9 Apr. 2002. Web. 5 Nov. 2013. <http://people.rit.edu/andpph/text-digital-stroboscopy.html>.

Gardiner, Martin. "GSA Advanced Golf Simulators: Club Track Camera." N.p., n.d. Web. 13 Nov. 2013. <http://www.golf-simulators.com/Future.html>.

"TrackMan Golf." *TrackMan Golf*. N.p., n.d. Web. 18 Apr. 2014. <http://trackmangolf.com/>.

Appendices

Appendix A:

This is the C code used to program the MSP430.

```
#include <msp430.h>

void init(void) {
    WDTCTL = WDTPW + WDTHOLD;

    BCSCTL2 = SELM_0 + DIVM_0 + DIVS_0;
    if (CALBC1_8MHZ != 0xFF) {
        /* Adjust this accordingly to your VCC rise time */
        __delay_cycles(100000);
        // Follow recommended flow. First, clear all DCOx and MODx bits. Then
        // apply new RSELx values. Finally, apply new DCOx and MODx bit values.
        DCOCTL = 0x00;
        BCSCTL1 = CALBC1_8MHZ;      /* Set DCO to 8MHz */
        DCOCTL = CALDCO_8MHZ;
    }

    /* Basic Clock System Control 1, XT2OFF -- Disable XT2CLK,
     * ~XTS -- Low Frequency, DIVA_0 -- Divide by 1 */
    BCSCTL1 |= XT2OFF + DIVA_0;

    /* Basic Clock System Control 3, XT2S_0 -- 0.4 - 1 MHz, XCAP_1 -- ~6 pF
     * LFXT1S_2 -- If XTS=0, XT1=VLOCLK ; If XTS=1, XT1=3 - 16-MHz Xtal or resonator */
    BCSCTL3 = XT2S_0 + LFXT1S_2 + XCAP_1;

    P1OUT = 0;
    P1SEL = BIT1 + BIT2; /* Port 1 Port Select Register */
    P1SEL2 = BIT1 + BIT2; /* Port 1 Port Select 2 Register */
    P1DIR = BIT0 + BIT6; /* Port 1 Direction Register */
    P1IES = 0; /* Port 1 Interrupt Edge Select Register */
    P1IFG = 0; /* Port 1 Interrupt Flag Register */
    P2IES = 0; /* Port 2 Interrupt Edge Select Register */
    P2IFG = 0; /* Port 2 Interrupt Flag Register */

    UCA0CTL1 |= UCSWRST; /* Disable USCI */
    /* Control Register 1,  UCSSEL_2 -- SMCLK
     * ~UCRXEIE -- Erroneous characters rejected and UCAxRXIFG is not set
     * ~UCBRKIE -- Received break characters do not set UCAxRXIFG
     * ~UCDORM -- Not dormant. All received characters will set UCAxRXIFG
     * ~UCTXADDR -- Next frame transmitted is data
     * ~UCTXBRK -- Next frame transmitted is not a break
     * UCSWRST -- Enabled. USCI logic held in reset state  */
    UCA0CTL1 = UCSSEL_2 + UCSWRST;
    /* Modulation Control Register,UCBRF_0 -- First stage 0
     * UCBRS_2 -- Second stage 2, ~UCOS16 -- Disabled     */
    UCA0MCTL = UCBRF_0 + UCBRS_2;
    UCA0BR0 = 65; /* Baud rate control register 0 */
    UCA0BR1 = 3;     /* Baud rate control register 1 */
    UCA0CTL1 &= ~UCSWRST; /* Enable USCI */

}


void main(void) {
        init();

    //The following code sets the MSP430's clock frequency to 1MHz
    BCSCTL2 = SELM_0 | DIVM_0 | DIVS_0;

    if (CALBC1_1MHZ != 0xFF) {
        DCOCTL = 0x00;
        BCSCTL1 = CALBC1_1MHZ;      /* Set DCO to 1MHz */
        DCOCTL = CALDCO_1MHZ;
```

```
    }

    BCSCTL1 |= XT2OFF | DIVA_0;

    BCSCTL3 = XT2S_0 | LFXT1S_2 | XCAP_1;

    // This section of code controls the flashing of the LED lights
        P1DIR = P1DIR | 0x01;             // Set P1.0 to output direction
        P2DIR = P2DIR | 0x00;                       // Set P2.0 to input direction
        // Note: 0x01 = 0000 0001 binary (the LSB is set to determine direction).
        int i;
    while (1) {
        if (P2IN == 0x01){
                UCA0TXBUF = 0x41;
                for (i = 0;i<100;i++) {               //Do this forever
                        P1OUT = 0x01;         // Turn on LEDs using P1.0
                        __delay_cycles(60);  //delay for 60 clock cycles
                        P1OUT = 0x00;
                        __delay_cycles(2340);  //delay for 2340 clock cycles
                }
        }
    }
}
```

Appendix B:

This is the top level MATLAB code used to automate the image acquisition process and perform the simulation.

```
%% Run this function once to set up the camera.

imaqreset;

% Connect to the webcam which is connected by USB.
vid = videoinput('winvideo', 1);

% Create a preview
preview(vid);

% Set the exposure to the highest exposure time
obj = getselectedsource(vid);
obj.Exposure = -2;

%% Run this section to take the image and run the simulation
addpath('Simulation', 'Computer Vision')

s=instrfind; % Find any serial links (we can have only 1)
delete(s); %... and delete.

% Create a new serial communications link
s=serial('COM4','Baudrate',115200);
fopen(s); %... and open it

% Scan in the message sent from the MSP430
% This blocks till MATLAB receives the message
[x, count] = fscanf(s,'%c',1)

% Always throw out the first image as the ball is
% most often caught in the second image after the signal is received.
img_1 = getsnapshot(vid);
img = getsnapshot(vid);

fclose(s);

% Extract the initial conditions using the image.
[speed, spin_rate, launch_angle] = find_init_cond(img, 24000);

% Simulate the ball flight using the initial conditions.
graphics(speed, spin_rate, launch_angle)
```

Appendix C:

This is the MATLAB code used to perform the image processing.

```
%% This function is the top level program that controls the image processing
%% section of the system
function [speed, spin_rate, launch_angle] = find_init_cond(img, rate)

[m, n , c] = size(img);
% Perform the thresholding on the image.
new_img = threshold(img, 1, 40, 1.4);

% Determine the straight lines in the image
lines_orig = find_lines(new_img, 2);
lines = pick_lines(lines_orig,2);

% Find the centers of the hemispheres in the image.
centers = find_centers(new_img, lines);

% Pick which balls to use.
[first,second] = pick_balls(centers);

% Sort the balls in order of where they show up in image.
[lines, centers] = sort_balls(lines, centers, m);

% Calculate the intial conditions
speed = calc_ballspeed(lines(first:second), rate);
launch_angle = calc_launch_angle(lines(first:second));
spin_rate = calc_spin_rate(lines(first:second), centers(first:second,:), rate);

end



%% This function performs the thresholding in the image.
function done_image = threshold(img, color, val, ratio)
    [m, n, c] = size(img);
    % Zero out the entire new image to start.
    new_image = zeros(m,n);
    for i = 1:3
        if (i == color)
            % This threshold ensure that the particular pixel component
            % is larger than a particular value.
            temp = img(:, :, i) >= val;
            new_image(temp) = new_image(temp) + 1;
        else
            % This ensures that the particular pixel component ratio to
            % the other pixel components is larger than a specific value.
            temp = (double(img(:, :, color)) ./ double(img(:, : , i))) >= ratio;
            new_image(temp) = new_image(temp) + 1;
        end
    end

    done_image = zeros(m,n);
    % If a particular pixel has a value of three, that means it met all
    % of the thresholding criteria so set its value to 1 in the final
    % image.
    done_image(new_image == 3) = 1;
end
```

```
%% This function finds the straight lines in the image.  We assume the
%% that the straight lines are the equators of the golf ball.
function lines = find_lines(BW, num_balls)
    % Erode the image to remove specks in the background.
    sel = strel('square', 3);
    BW = imerode(BW, sel);

    % Dilate the image to make the edges more pronounced and fill any
    % gaps in the ball
    sel = strel('square', 6);
    BW = imdilate(BW, sel);

    % Perform edge detection and enlarge the lines via dialation.
    BW = edge(BW, 'canny');
    se2 = strel('square', 2);
    BW = imdilate(BW, se2);

    % Perform the hough transformation
    [H, theta, rho] = hough(BW, 'RhoResolution', 20);

    % Set the NHoodSize to control how close maximums can be in
    % order to get lines on seperate balls.
    scaling = 2;
    kf = floor(size(H)/scaling);
    sizeN = kf - (1 - mod(kf,2));

    % Find the peaks and the lines in the image using the hough transform.
    h_peaks = houghpeaks(H, num_balls,'Threshold', 0.05*max(H(:)), 'NHoodSize', sizeN);
    lines = houghlines(BW, theta, rho, h_peaks);

end



%% This function picks the longest straight lines given how many balls are desired.
function new_lines = pick_lines(lines, num_balls)

for i = 1:num_balls
    max = -1;
    max_length = -1;
    for j=1:length(lines)
        line = lines(j);
        dis = dist(line.point1, line.point2);
        if (dis>max_length)
          max = j;
          max_length = dis;
        end
    end
    new_lines(i) = lines(max);
    lines(max) = [];
end
end
```

```
%% Determines the centers of the hemispheres in the image.
function centers = find_centers(img, lines)
    % Dilate the image to start to make sure the hemispheres are whole.
    sel = strel('square', 15);
    BW = imdilate(img, sel);
    % Perform blob detection on the image.
    L = bwlabel(BW);
    blob_measurements = regionprops(L, 'all');
    % Pick the balls that best match up with the given lines.
    for i=1:length(blob_measurements)
        best = 0;
        best_dist = Inf;
        for j=1:length(lines)
            line = lines(j);
            midpoint = [(line.point1(1) + line.point2(1))/2,
                        (line.point1(2) + line.point2(2))/2];
            if (dist(blob_measurements(i).Centroid, midpoint) < best_dist)
                best = j;
                best_dist = dist(blob_measurements(i).Centroid, midpoint);
            end
        end
        centers(best, :) = blob_measurements(i).Centroid;
    end
end



%% Pick two balls based on closest proximity of hemispheres.  We
%% assume that the closest balls will be adjacent ones.
function [first, second] = pick_balls(centers)
    best_dist = Inf;
    for i=1:length(centers)
        for j=i:length(centers)
            if (i~=j)
                if (dist(centers(i, :), centers(j, :)) < best_dist)
                    first = i;
                    second = j;
                    best_dist = dist(centers(i, :), centers(j, :));
                end
            end
        end
    end
end



%% Sort the balls in order of when they show up in image.
function [new_lines, new_centers] = sort_balls(lines, centers, m)
    origin = [0 m];
    for i=1:length(centers)
        distances(i) = dist(origin, centers(i,:));
    end
    distances = sort(distances);
    for i=1:length(distances)
        for j=1:length(centers)
            if distances(i) == dist(origin, centers(j,:))
                new_lines(i) = lines(j);
                new_centers(i,:) = centers(j,:);
            end
        end
    end
end
```

```
%% Calculate the ball speed.
function speed = calc_ballspeed(lines, flash_rate)
    % Diameter of golf ball in miles.
    diam = 2.65152*10^(-5);

    % Determine length of golf ball by averaging length of equators of
    % balls.
    average_length = 0;
    for i=1:2
        average_length = average_length + dist(lines(i).point1,lines(i).point2);
    end
    average_length = average_length/2;

    % Get the conversion from pixels to miles.
    miles_per_pixel = diam / average_length;

    midpoint1 = [(lines(1).point1(1) + lines(1).point2(1))/2,
                 (lines(1).point1(2) + lines(1).point2(2))/2];

    midpoint2 = [(lines(2).point1(1) + lines(2).point2(1))/2,
                 (lines(2).point1(2) + lines(2).point2(2))/2];

    % Find distance between midpoints.
    pixels_traveled = dist(midpoint1, midpoint2);
    % Find amount of miles traveled
    miles_traveled = miles_per_pixel * pixels_traveled;

    % Determine the ball speed.
    miles_per_minute = miles_traveled * flash_rate;
    speed = miles_per_minute * 60;

end


%% Calculate the launch angle.
function launch_angle = calc_launch_angle(lines)
    % Get the midpoints of the equators.
    midpoint1 = [(lines(1).point1(1) + lines(1).point2(1))/2,
                 (lines(1).point1(2) + lines(1).point2(2))/2];

    midpoint2 = [(lines(2).point1(1) + lines(2).point2(1))/2,
                 (lines(2).point1(2) + lines(2).point2(2))/2];
    % Find the vector between the equator midpoints.
    v = abs(midpoint2 - midpoint1);
    u = [1 0];
    v(3) = 0;
    u(3) = 0;
    % Get the angle between the horizontal and that vector.
    angle = atan2(norm(cross(v,u)),dot(v,u));
    launch_angle = angle * 180/pi;
end
```

```
%% Calculate the spin rate.
function spin_rate = calc_spin_rate(lines, centers, flash_rate)

 for i = 1:2
    y = lines(i).point2(2) - lines(i).point1(2);
    x = lines(i).point2(1) - lines(i).point1(1);

    midpoint = [(lines(i).point1(1) + lines(i).point2(1))/2,
                (lines(i).point1(2) + lines(i).point2(2))/2];
    % Find the vector from midpoint to center of hemisphere.
    center_vector = midpoint - transpose(centers(i,:));


     % There are two possible normals
    normal1 = [y, -x];
    normal2 = [-y, x];

    % Pick the normal closest to the midpoint-center vector.
    if(dist(center_vector, normal1) < dist(center_vector, normal2))
        normal(i,:) = normal1;
    else
        normal(i,:) = normal2;
    end
    % Normalize the vector.
    v(i, :) = normal(i,:)/norm(normal(i,:));
    v(i, 2) = -v(i, 2);
    % Calculate the angle between the two.
    temp_angle = atan2(v(i,2), v(i,1)) * 180/pi;

    % Account for negative angles.
    if (temp_angle < 0)
        angle(i) = temp_angle + 360;
    else
        angle(i) = temp_angle;
    end
 end
 % Add the angle together to get the difference.
 % Account for overlapping nature of angles on unit circle.
 if (angle(1) > angle(2))
     total = (360 - angle(1)) + angle(2);
 else
     total = angle(1) + angle(2);
 end

 % Get the spin rate.
 spin_rate = total/360 * flash_rate;

end
```

Appendix D:

This is the MATLAB code used to perform the simulation.

```
%% This function is the top level function that performs the
%% simulation and produces the graphics
function [carry, total_dist] = graphics(v0, omega, alpha2)

fig = figure(1);
% Maximize the figure on the screen.
set (fig, 'Units', 'normalized', 'Position', [0,0,1,1]);

% define the size of the graphics in the simulation.
r = 0.5; % 1.5;define radius of golf ball animation
xmin = -300; % max number of units to the left.
xmax = 500;  % max number of units to the right.
ymin = -100; % max number of units behind.
ymax = 700; % max number of units in front.
zmin = 0;
zmax = 100;

grass = imread('grass2.jpg');%read picture of grass from file
sky = imread('sky.jpg');

hold on

%plot picture of grass on planar surface z = 0
xgrid = linspace(xmin,xmax);
ygrid = linspace(ymin,ymax);
[xgrid,ygrid]=meshgrid(xgrid,ygrid);
f = zeros(size(xgrid));
surf(xgrid,ygrid,f,'CData',flipdim(grass,1),'FaceColor','texturemap','EdgeColor','none')
axis([xmin/2 xmax/2 ymin/2 ymax/2 0 50])

% Plot the sky in the simulation
ygrid = linspace(ymin,ymax);
zgrid = linspace(zmin, zmax);
[ygrid,zgrid]=meshgrid(ygrid,zgrid);
xgrid = ones(size(ygrid))*(xmin+5);
surf(xgrid,ygrid,
zgrid,'CData',flipdim(sky,1),'FaceColor','texturemap','EdgeColor','none')

% Extend the grass
ygrid = linspace(ymin,ymax,ymax-ymin);
zgrid = linspace(-50, 0);
[ygrid,zgrid]=meshgrid(ygrid,zgrid);
xgrid = ones(size(ygrid))*(xmin + 100);
surf(xgrid,ygrid,
zgrid,'CData',flipdim(grass,1),'FaceColor','texturemap','EdgeColor','none')

% Make yardage markers.
cyl_rad = 1;
n_cyl = 5;
[xcyl, ycyl, zcyl] = cylinder(cyl_rad, n_cyl);
ycyl = ycyl + 100;

% Make the red (100 yard) marker.
for i=1:5
  if (mod(i,2) == 0)
    surf(xcyl,ycyl,zcyl,'Facecolor','white');
  else
    surf(xcyl,ycyl,zcyl,'Facecolor','red');
  end
  zcyl = zcyl + 1;
end

% Make the black (150 yard) marker.
[xcyl, ycyl, zcyl] = cylinder(cyl_rad, n_cyl);
ycyl = ycyl + 150;

for i=1:5
  if (mod(i,2) == 0)
```

```
    surf(xcyl,ycyl,zcyl,'Facecolor','white');
  else
    surf(xcyl,ycyl,zcyl,'Facecolor','black');
  end
  zcyl = zcyl + 1;
end

% Make the blue (200 yard) marker.
[xcyl, ycyl, zcyl] = cylinder(cyl_rad, n_cyl);
ycyl = ycyl + 200;

for i=1:5
  if (mod(i,2) == 0)
    surf(xcyl,ycyl,zcyl,'Facecolor','white');
  else
    surf(xcyl,ycyl,zcyl,'Facecolor','blue');
  end
  zcyl = zcyl + 1;
end

% Make the yellow (250 yard) marker.
[xcyl, ycyl, zcyl] = cylinder(cyl_rad, n_cyl);
ycyl = ycyl + 250;

for i=1:5
  if (mod(i,2) == 0)
    surf(xcyl,ycyl,zcyl,'Facecolor','white');
  else
    surf(xcyl,ycyl,zcyl,'Facecolor','yellow');
  end
  zcyl = zcyl + 1;
end

%create grid over which to evaluate sphere (function of two variables phi
%and theta)
phi=linspace(0,pi,12);
theta=linspace(0,2*pi,12);
[phi,theta]=meshgrid(phi,theta);

axis equal
axis off
view(90,5)
% How zoomed in on the ball should it be.
zoomInFactor = 0.03;
% Controls when to plot the ball.
skip_factor = 7;

% Run the simulation to extract flight of the ball.
[x, y, carry, total_dist] = simulate_with_bounce(v0, omega, alpha2);

for i=1:length(x)
     %do some cool plotting stuff
     x2=r*sin(phi).*cos(theta);
     y2=r*sin(phi).*sin(theta)+x(i);
     z2=r*cos(phi)+y(i);
     %more cool plotting stuff
     PP = zeros(size(x2));
     if (mod(i,skip_factor) == 0)
       % Plot the ball.
       ball = surf(x2,y2,z2,'Facecolor','white');
       shadow = surf(x2,y2,PP,'FaceColor','k');
       trail = surf(x2,y2,z2,'Facecolor','white','EdgeColor','none');
       % Add to the tail of the ball.
       alpha(trail,.15)
       % Look at the ball and then zoom out.
       camlookat(ball)
       camzoom(zoomInFactor)
       pause(.000001)
       % Ensure we do not get weird results from deleting the ball at the
       % end of the simulation
       if (i < length(x) - skip_factor)
         camzoom(1/zoomInFactor)
         delete(ball)
```

```
        delete(shadow)
      end
    end

end

% Plot the results of the simulation
clear fig
subplot(2,1,1)
plot(x,y)
grid on
title('Flight of the Golf Ball')
xlabel('Yards')
ylabel('Yards')
xlim([0 330])
ylim([0 70])

mTextBox = uicontrol('style','text');
str = sprintf('Ball Speed (mph): %4.1f\nSpin Rate (rpm): %5.1f\nLaunch Angle (degrees):
%3.1f\nCarry Distance (yards): %4.1f\nTotal Distance (yards): %4.1f', v0, omega, alpha2,
carry, total_dist);
set(mTextBox,'String',str);
set(mTextBox, 'Position', [500 150 300 150]);
set(mTextBox, 'FontSize', 20);
set(mTextBox, 'HorizontalAlignment', 'left');

end


%% This function adds bounce to the simulation
function [xfinal,yfinal, carry_distance, final_distance] = simulate_with_bounce(v0,
omega, alpha)
    % Some constants and conversions to begin.
    x = 0;
    y = 0;
    alpha = alpha*pi/180;
    v0 = 0.44704*v0;
    m_to_yd = 1.09361;
    dt = 0.01;

    % Perform the simulation for when the ball is in the air
    q = simulate(x,y,v0*cos(alpha),v0*sin(alpha),omega,alpha,dt);

    qfinal = q;
    carry_distance = q(1,end) *m_to_yd;
    bounce_factor = 0.3;
    % Make the ball bounce forward unti it stops.
    while (q(3,end)>= 0.01)
        x = q(1,end);
        y = 0;
        dx = bounce_factor*(q(3,end));
        dy = -bounce_factor*(q(4,end));
        q = simulate(x,y,dx,dy,omega, alpha,dt);
        qfinal= [qfinal,q];

    end

    % Append the bounce to the simulation.
    xfinal = qfinal(1,:)*m_to_yd;
    yfinal = qfinal(2,:)*m_to_yd;
    final_distance = xfinal(end);
end
```

```
%% A wrapper function that simulates the ball until it hits the ground.
%% Such that the y value is below zero.
function records = simulate(x, y, dx, dy, omega, alpha, dt)
q = [x; y; dx; dy; omega; alpha; dt];
records = q;
count = 1;
while 1
    q = ball_flight(q);
    count = count + 1;
    records(:,count) = q;
    if (q(2)<=0)
      break;
    end
end
end


%% This function performs the actual simulation of the ball in flight.
function dq = ball_flight(q)
% Extract the values from the vector
x = q(1);
y = q(2);
dx = q(3);
dy = q(4);
omega = q(5);
alpha = q(6);
dt = q(7);

m = 0.0453; % mass of the ball
r = 0.021; % radius of ball
d = 2*r; % diameter of ball

% Get coefficients of lift and drag
[Cl, Cd] = get_coeff(sqrt(dx^2 + dy^2), omega);

S = pi*d^2/4; % Cross Sectional area of ball
rho = 1.2; % density of air, kg/m^3

% Calculate the accelerations in the x and y direction
ax = -rho*S/2/m*(dx^2 + dy^2)*(Cd*cos(alpha)+Cl*sin(alpha));
ay = rho*S/2/m*(dx^2 + dy^2)*(Cl*cos(alpha) - Cd *sin(alpha)) - 9.8;

% Return then new q vector based on the increment of time.
dq = [x + dx*dt;
      y + dy*dt;
      dx + ax*dt;
      dy + ay*dt;
      omega; alpha; dt];
end
```

```
%% This function finds the coefficients of lift and drag
%% given the ball speed and the spin rate
function [Cl, Cd] = get_coeff(ball_speed, spin_rate)
ball_speed_lines = [14, 21.9, 30.5, 39,47.2, 55.5, 64, 72.8,81.1, 89];

% Determine what line the ball speed best represents
diff = abs(ball_speed_lines - ball_speed);
[val, i] = min(diff);

% The spins from the table
spins = [0, 1100, 1900, 2800, 3800, 4700, 6300];

% Determine what spin rate is best represented.
diff = abs(spins - spin_rate);
[val, j] = min(diff);

% The coefficients of drag based on speed and spin rate.
cD = [0.5, 0.47, 0.37, 0.405, 0.41, 0.49, 0.52;
      0.35, 0.29, 0.3, 0.35, 0.36, 0.38, 0.39;
      0.22, 0.22, 0.23, 0.25, 0.29, 0.31, 0.33;
      0.22, 0.22, 0.22, 0.22, 0.24, 0.27, 0.29;
      0.22, 0.22, 0.22, 0.22, 0.23, 0.25, 0.26;
      0.21, 0.21, 0.21, 0.21, 0.23, 0.25, 0.26;
      0.21, 0.21, 0.21, 0.21, 0.23, 0.23, 0.24;
      0.21, 0.21, 0.21, 0.22, 0.23, 0.23, 0.24;
      0.21, 0.21, 0.21, 0.21, 0.23, 0.23, 0.24;
      0.21, 0.21, 0.21, 0.21, 0.23, 0.23, 0.24;];

Cd = cD(i,j);

% The spins from the table
spins = [1100, 1900, 2800, 3800, 4700, 6300];

% Determine what spin rate is best represented.
diff = abs(spins - spin_rate);
[val, j] = min(diff);

% The coefficients of lift based on speed and spin rate.
cL = [-0.1, 0.2, 0.28, 0.39, 0.42, 0.45;
       0.15, 0.23, 0.27, 0.34, 0.36, 0.41;
       0.12, 0.2, 0.23, 0.28, 0.31, 0.35;
       0.11, 0.16, 0.18, 0.23, 0.26, 0.3;
       0.105, 0.15, 0.16, 0.2, 0.22, 0.23;
       0.10, 0.14, 0.15, 0.19, 0.20, 0.22;
       0.10, 0.13, 0.16, 0.15, 0.18, 0.21;
       0.10, 0.11, 0.15, 0.16, 0.18, 0.2;
       0.10, 0.11, 0.15, 0.16, 0.18, 0.2;
       0.10, 0.11, 0.15, 0.16, 0.18, 0.2];


Cl = cL(i,j);

end
```