6.111 Final Project
Autonomous RC Car

Kevin Chan · David Gomez · Battushig Myanganbayar

December 9, 2015

Abstract

The goal of this project is to use a Nexys4 FPGA to guide a small, off the shelf RC car around a track defined by the user. The user uses electrical tape to draw a closed loop on a white surface like a whiteboard and then issues a command to the FPGA to recognize the track through a VGA camera. The user then places an IR bandpass filter over the camera and commands the FPGA to start driving the car. The car itself has IR LED's on its roof which are recognized by video processing modules to determine the car's location and heading. This information is relayed to a car controller module which makes the appropriate commands to the RC car through its controller to keep the car on track.

Contents

List of Figures

List of Tables

1 Motivation

This project was largely inspired by the latest work in autonomous cars and Stanford's autonomous Audi RS7 which is designed to race around real racetracks on the limit of its performance. We wanted to pick a project that really pushed the limits of an FPGA and took advantage of an FPGA's strengths to do things that are difficult to do with a standard microcontroller.

The fast and live video processing, image recognition, and control we do in this project really filled that requirement for our team. We also wanted to learn how to interface the FPGA with more sophisticated peripherals like the VGA camera. The VGA camera uses a proprietary I^2C style communication protocol to initialize its configuration registers and returns data over a parallel bus. Learning how to control advanced peripherals over commonly used communication protocols was a valuable lesson.

Two members of our project are on the Formula SAE team at MIT and really love racecars. The ultimate joke of this project was that at the end of the project it should be possible to port this entire project to MIT's electric formula car.

2 Physical Setup

As shown in Figure 1, our setup consists of an FPGA and VGA camera mounted to the ceiling to look down at a whiteboard on the ground.

In Figure 2 you can see the Nexys 4 FPGA assembly clamped to the ceiling. Also attached to the ceiling are the VGA camera and the remote control that came with the RC car we purchased. As shown in Figure 3a, the VGA camera is clamped to the ceiling with its own mount to ensure that it is looking straight down at the track. We have a velcro strip that allows us to easily attach and remove the IR filter as shown in Figure 3b.

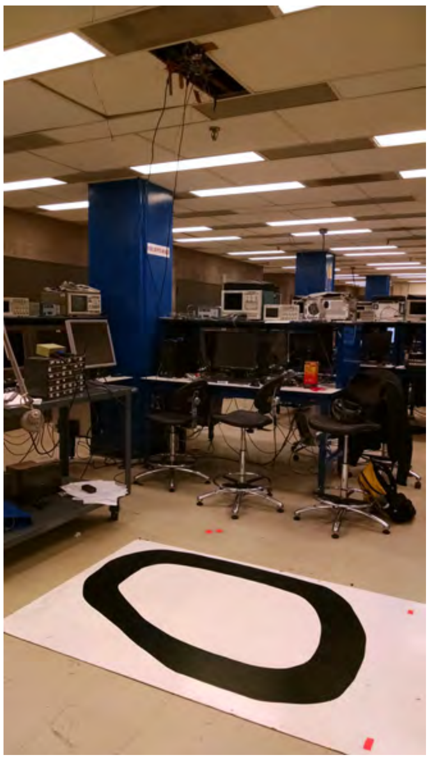


Figure 1: Overview of the whole system

The camera looks straight down at the track and sees an image like the one in Figure 4.

Our car is an off the shelf RC car from Amazon. We have modified its controller to bypass the normal push buttons with FETs which allow us to control the car from the FPGA. The RC Car itself has a breadboard with two IR LEDs attached to it which allow the camera to easily find the car.

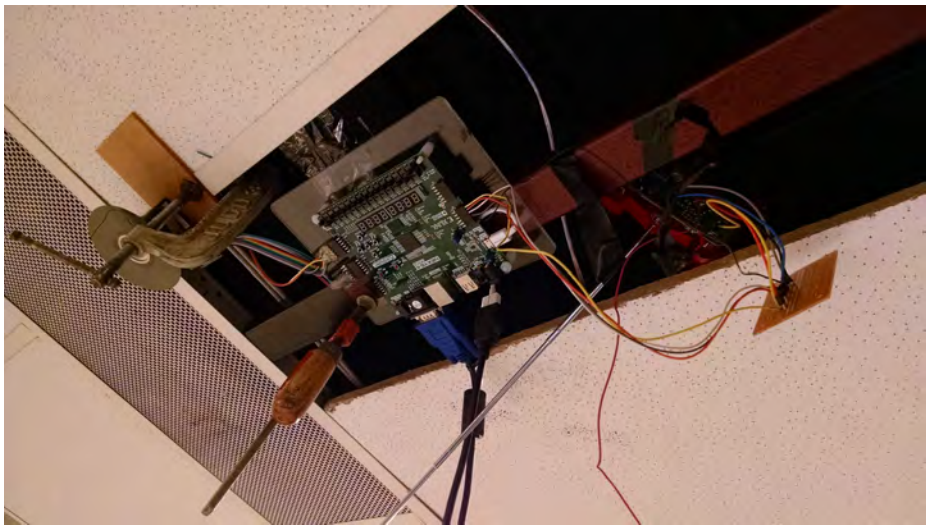
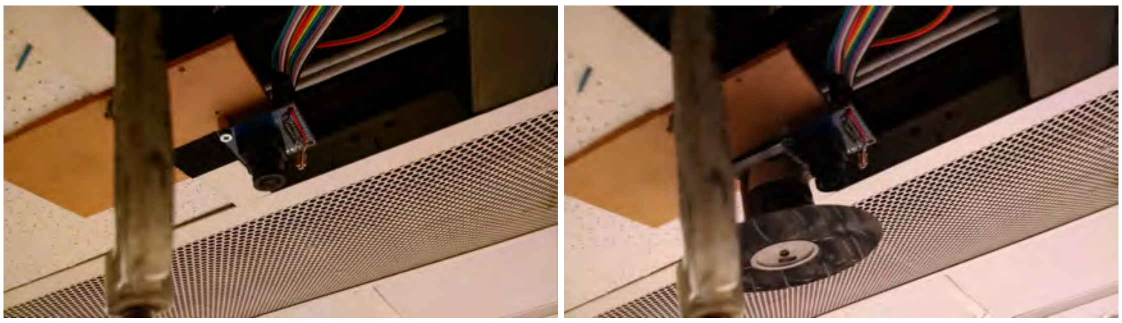


Figure 2: The Nexys 4, VGA Camera, and RC Controller clamped to the ceiling


(a) Without filter

(b) With IR filter

Figure 3: The VGA camera mounted to the ceiling

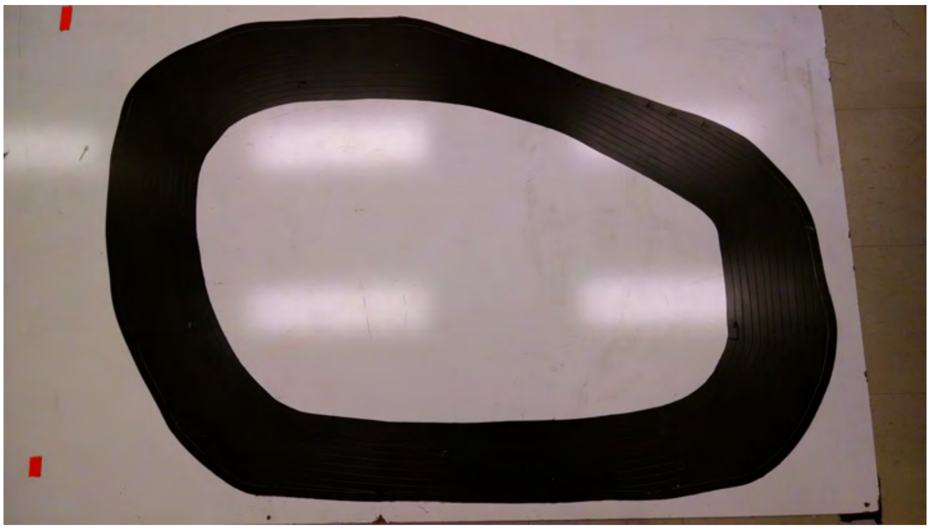


Figure 4: View of the track from the VGA camera

3 System Architecture

The core of the project can be broken into three main subassemblies, one corresponding to each member of our group. These critical assemblies are the Track Processing block, Car Position Processing block, and Car Controller. The structure is shown in Figure 5.

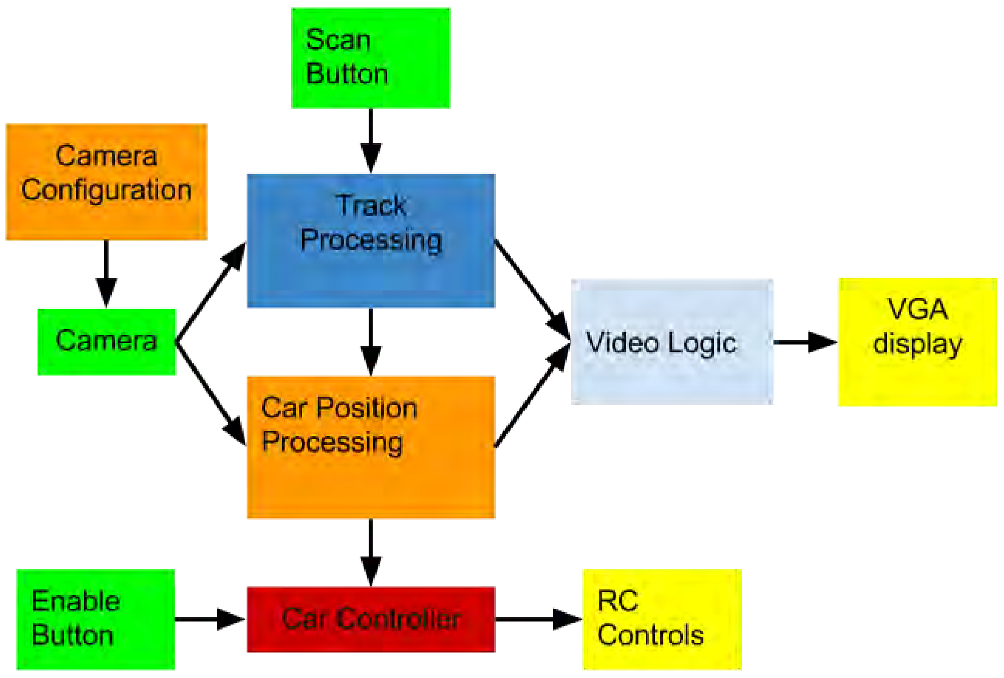


Figure 5: High level block diagram

3.1 Camera Configuration and Reading

3.1.1 Purpose

Our system uses one OV7670 VGA camera mounted to the ceiling pointed downwards at our track and car. The camera is used first to capture the shape of the drawn track. After the track has been processed, the camera data is used to provide visual feedback on the car's position to the car controller.

Because both Kevin's and Battushig's modules for Car Position Finding and Track Recognition required reliable access to camera data, they worked together to build modules that configured the camera on startup and recorded the camera's outputted data to block RAM (BRAM) on the FPGA.

3.1.2 Camera Configuration

This module writes the camera's internal configuration registers upon startup of the camera. These configuration registers change camera parameters such as resolution, frames per second, scaling, white balance, and data output format. The camera is configured using the SIOC and SIOD pins on the camera which are the differential lines on the I^2C bus used only for initial configuration. SIOC and SIOD are driven from one of the JX IO ports on the FPGA.

The configuration module works by reading values that are set in a ROM and sending them sequentially over the serial bus to the camera.

The camera configuration code had been shared with us by the 6.111 staff. We had to make some critical modifications to the provided code to ensure the camera was configured properly for our applications. Because we want to process black and white images only, we instructed the camera to return YUV values. Later, this would allow us to extract gray scale images from the camera by only extracting the intensity (Y) values. Setting the camera up in this grayscale mode involved changing the values being written to several of the configuration registers.

Our largest issues with this module were caused by unclear documentation for the OV7670 camera. We had trouble understanding what each configuration register was actually controlling and what values to write to each register to achieve our desired setup. Finding the values we were supposed to write to each register consisted mostly of trial and error. We will be releasing our code to the 6.111 staff so that it can be used as a reference for setting up the VGA camera in grayscale mode.

3.1.3 VGA Camera Read and BRAM

After the camera has been configured, it starts outputting data from its D0-7, VSYNC, PCLK, and HREF outputs. This module stores the data from data bus by interpreting it with the timing provided by the other outputs. The VSYNC signal is used to indicate the beginning of a new frame. The data output is timed on PCLK. HREF indicates valid output on the data bus. This module takes the valid data outputs and stores them appropriately in a dual port BRAM to be accessed by other modules.

The camera read module uses state machine logic to keep track of transitions through the timing logic of the camera's outputs. The timing is depicted in Figure 6.

The module begins by waiting for a falling edge of VSYNC which indicates the beginning of a new frame. The camera reports pixel values in order beginning from the top left corner of the frame and then moving right and down. At a falling edge of VSYNC, we anticipate valid data whenever HREF goes high. When we see HREF high, we begin recording bytes from the data bus at each posedge of the PCLK line. Based on our camera's configuration, we know that the order of outputted bits is U, Y1, V, Y2 where Y1 and Y2 are the intensity values for two adjacent pixels and U and V are the shared chroma values for those pixels. This pattern of outputted bits is repeated for the entire frame. Our state machine is set up to ensure that we capture only all the Y values. To store the values as they are being outputted, we increment a data address line for our BRAM which

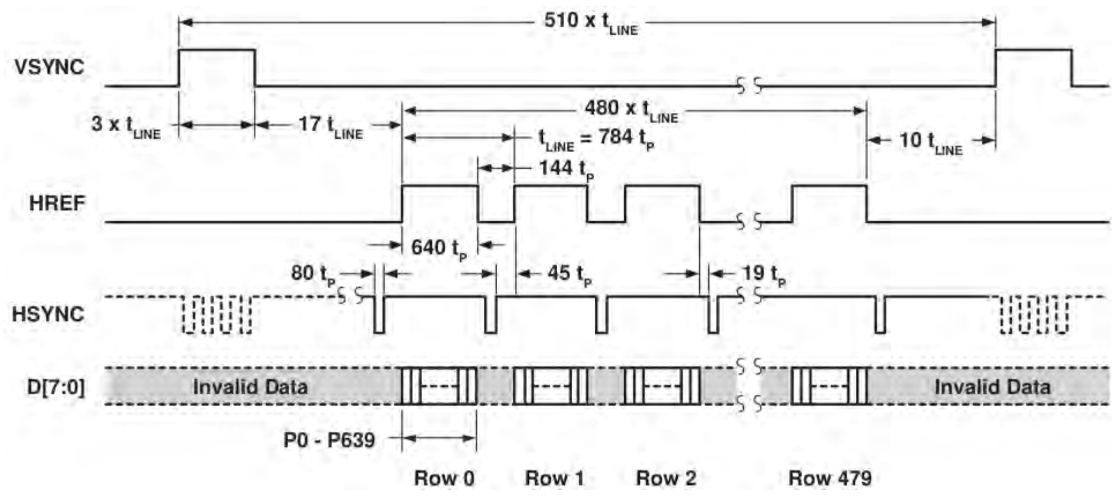


Figure 6: VGA camera output timing diagram

is clocked at the same PCLK output from the camera. Our BRAM is structured to store two pixels' Y values in each row of the BRAM. We had to do this because the depth limit of the BRAM IP on the Nexys4 FPGA is smaller than the total number of pixels we have to store ($480*640 = 307200$). Because we write two pixel values per address in the BRAM and we write the first pixel's data in address 0, the intensity information for a pixel at (x, y) is located at address $(y*640+x)/2$. Whether the pixel's intensity information is located in the most significant half or least significant half of the BRAM row is determined by the parity of the sum x + y. We transition back to a waiting state and reset the write address register for our BRAM when we see a rise in VSYNC which indicates the end of the frame.

Originally, a camera read module had been provided by the course staff along with the camera configuration module; however, After a few hours of trying to get the provided code to work, Kevin and Battushig decided to re-implement the camera read module from scratch nearly a week into the project. This early decision in the project was risky as we were abandoning code that had been demonstrated as working to build our own camera read module. This proved to be the right decision as we ultimately were able to extract data from the camera sooner than other groups that continued to work with the provided code. It also was a greater learning opportunity as we learned more details about the timing of the VGA output and extended us greater flexibility in how our code wrote to memory.

3.2 Track Processing

3.2.1 Purpose

The purpose of this module is to create an internal map of the track that will be used to correlate the car's current and predicted locations to regions of the track. We defined three regions of the track: outside of the track, on the track, and inside the track. Given the region the car is currently on and where it is heading, we can send appropriate commands to the car to keep it driving on the track.

3.2.2 Filtering

To make the recognition process easier, we apply a thresholding filter to the intensity values, Y, stored in the BRAM. The thresholding filter assigns a pixel to white if the intensity value for that pixel is above the thresholding value and black if the intensity is below that value. We experimented with the threshold values, and found that a threshold of 0x30 resulted in the the best noise rejection on our whiteboard without loss in the elements we care about.

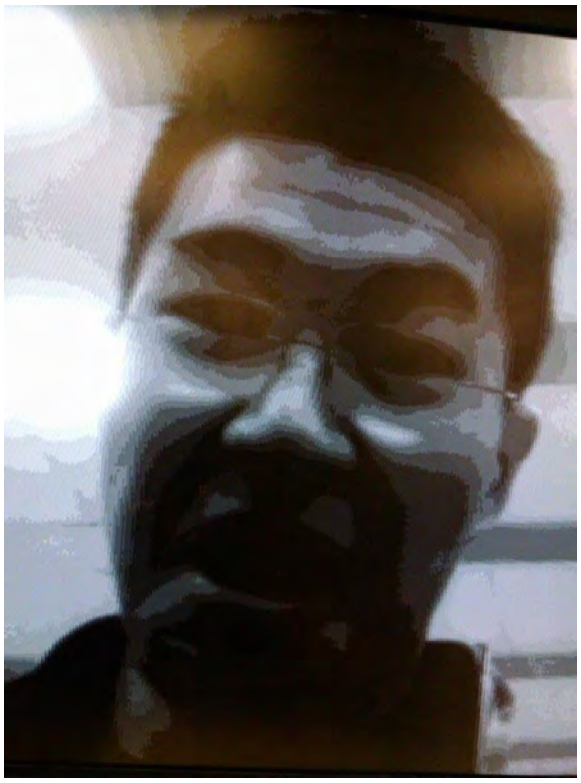


Figure 7: Raw grayscale output from VGA camera

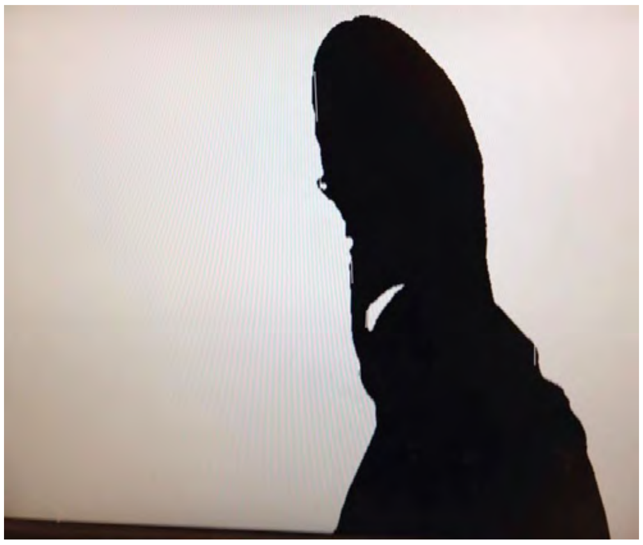
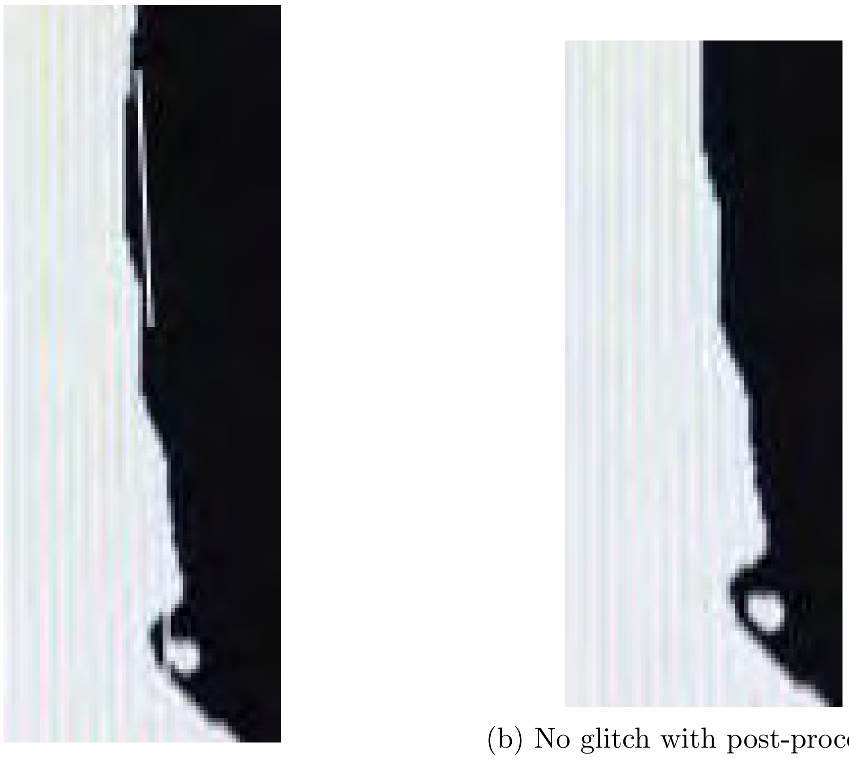


Figure 8: Output from thresholding filter applied to picture of shoe

Figure 7 shows the raw grayscale output from the VGA camera. Figure 8 shows what effect the thresholding filter has when applied to a picture of a person's foot.

In our testing with the thresholding filter, we noticed noise in the transition between black regions and white regions as can be seen in Figure 9a. These small glitches no greater than 2 pixels wide caused massive errors in our track processing algorithm. We changed our algorithm to tag all neighboring pixels around a black pixel as black to eliminate this effect. With that extra processing enabled, we were able to obtain a sharper, cleaner image like the one shown in Figure 9b.


(a) Thresholding filter only

(b) No glitch with post-processing

Figure 9: Thresholding glitch processing

As you can see in the Figure 9a, there was a vertical white line encroaching on the black region. When we assert that neighbors of black pixels are also black, we obtain the cleaner image seen in Figure 9b.

There were risks associated with this post processing since we increased the size of actual track by 2 pixels on the transitioning edges from black to white. However for our application, 2 pixels of error is tolerable and hasn't caused any observably wrong behavior in the overall control of the car.

3.2.3 Track Region Detection and Storage

With the filtered image, the track processing algorithm can safely assume that any solid black in the middle of the white background is the track. The challenge was identifying spaces that were within the boundaries of the track and others that are outside the boundary of the track as both of these areas are just white. We store a number in another BRAM (which we will refer to as the region BRAM) using the same addressing scheme as our camera's BRAM that represents each pixel's region data.

For each row in the frame, we first perform a horizontal scan from left to right until we find a white to black transition. All pixels from the left edge of the frame to this transition point are classified as outside the track, and we assign those pixels a value of 4'b1010 in the region BRAM to represent this. We do a similar scan except starting from the right edge of the screen, again marking all pixels between the right edge and the first white to black transition as outside the track and assigning those pixels a value of 4'b1010 in the region BRAM.

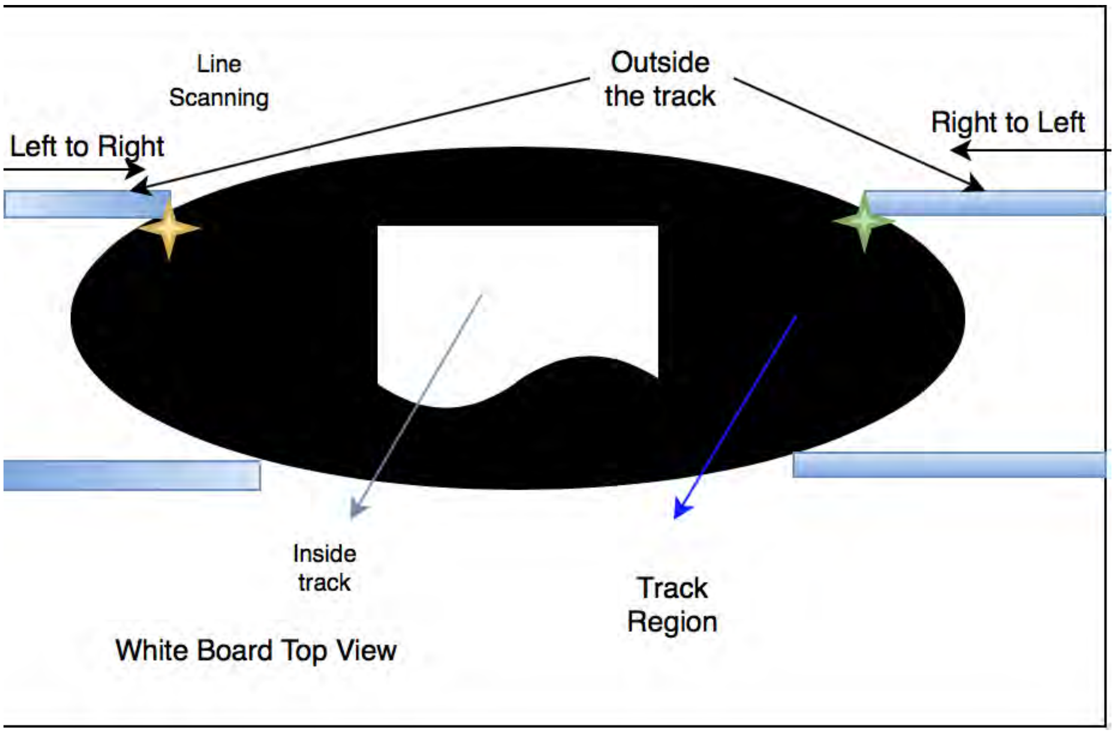


Figure 10: Visual representation of the region detection algorithm

The illustration in Figure 10 shows that, one one specific row, the pixels between the left edge of the screen and the yellow star and the pixels between the right edge of the screen and the green star are marked as outside the track.

After we have identified the outer region of the track, distinguishing between the track and the region within the track is very easy as the track is black and the remaining white pixels are the inner track. Thus, after we have finished assigning the pixels outside the track the appropriate value in the region BRAM, we just assign the rest of the pixels in the row to their color. This means that in our region BRAM, pixels that belong on the track have a value of 4'b0000 and pixels inside the track have a value of 4'b1111.

To be more memory efficient, the filtering and track tagging is done in place in the same BRAM. This also prevents the module from becoming overly complicated and requiring access to multiple scratchpad memory blocks and final output memory blocks.

One issue we ran into during integration was the need for the video module to the display the region data on the screen while simultaneously allowing the car controller module to read the region data for select points on the screen. This problem was easily solved by simply creating a copy of the final region BRAM, allowing the display module and car controller module to access region data independently in their own copy of the region BRAM.

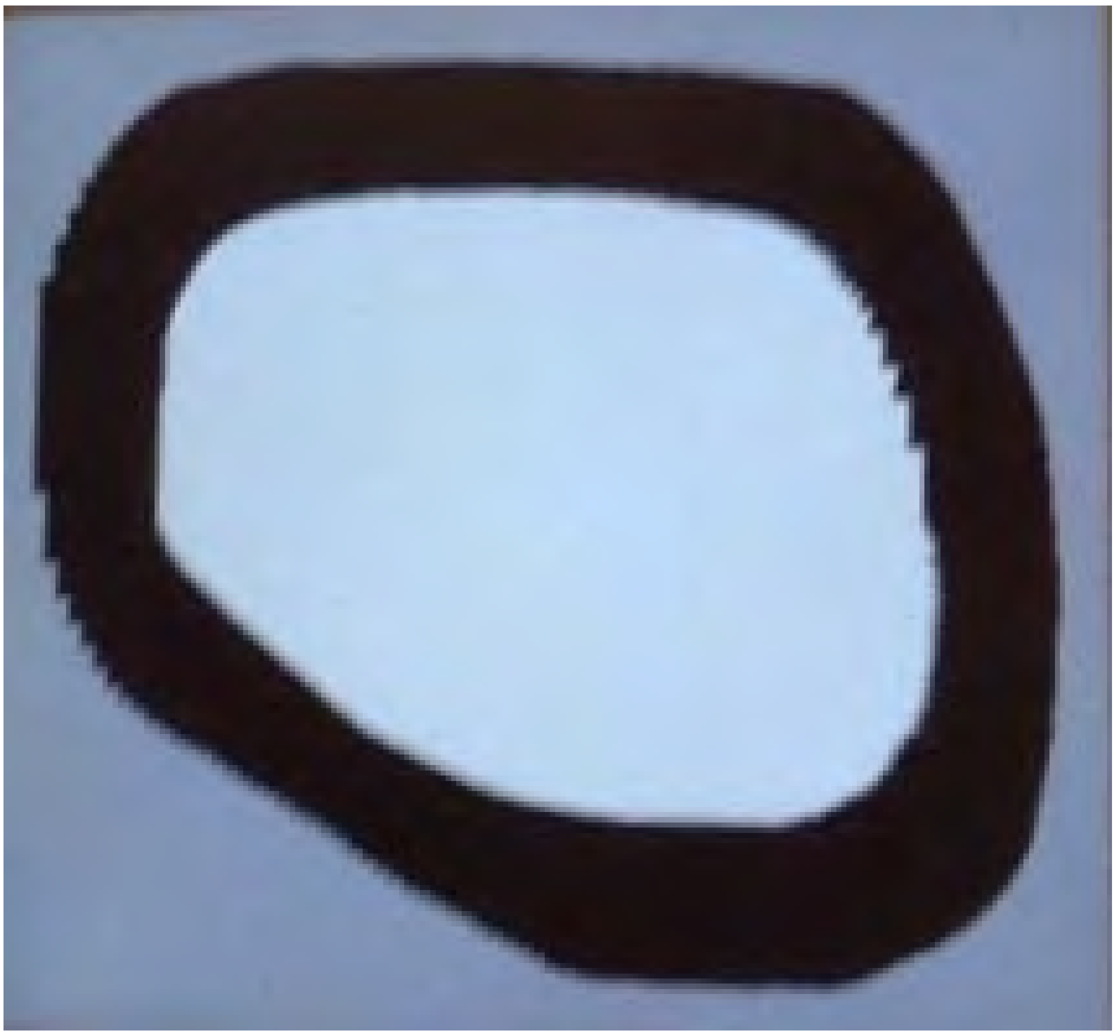


Figure 11: The track regions as processed with our algorithm displayed on a monitor. Grey is the outer track, black is the track, white is the inner track

3.3 Car Position Processing

3.3.1 Purpose

This module processes the live video coming from the camera while the car is driving to determine its current location and generate a next predicted position. We pass these coordinates to the Region Manager in the car controller to determine where the car is currently and where it will be relative to the track.

3.3.2 Center of Mass Detection

The car has two infrared (IR) LED's on its roof. Through the IR bandpass filter that we use on our camera, the LEDs appear as white circles on a black background. We have to find the center of mass of each circle corresponding to an LED so that we have exact coordinates to work with in determining the car's location and heading. The first step of finding the center of mass is to pass the bytes coming from the camera through a thresholding filter to make each pixel a binary black or white value. This filtering was described above for the track processing module.

The traditional approach to finding the center of masses of these pure white dots would be to find the mean of the x and y coordinates for "blobs" of adjacent white pixels. This method is fairly inefficient and difficult to implement in hardware as it requires division by the number of adjacent white pixels which is not constant because of noise in the camera and ambient lighting. We had to create a more lightweight approach that took advantage of certain properties of our application.

The center of mass detection algorithm we developed takes advantage of the fact that we are finding the center of masses of small circles on the screen. Our algorithm scans row by row through the filtered image starting at the top of the frame. As soon as it finds one pixel that is white, it accepts that point as the center of mass of the first LED. It then asserts that white pixels that are within some window drawn around that first point cannot be the center of mass of the second LED. This window is drawn large enough so that it covers all the white pixels associated with the first LED but not so large as to cover part of the second LED's white circle. The algorithm continues its scan down the image until it finds another white pixel that is outside of the window. It then assumes that this is the center of mass of the second LED and stops its search because it has found both center of masses. This is illustrated clearly in Figure 12.

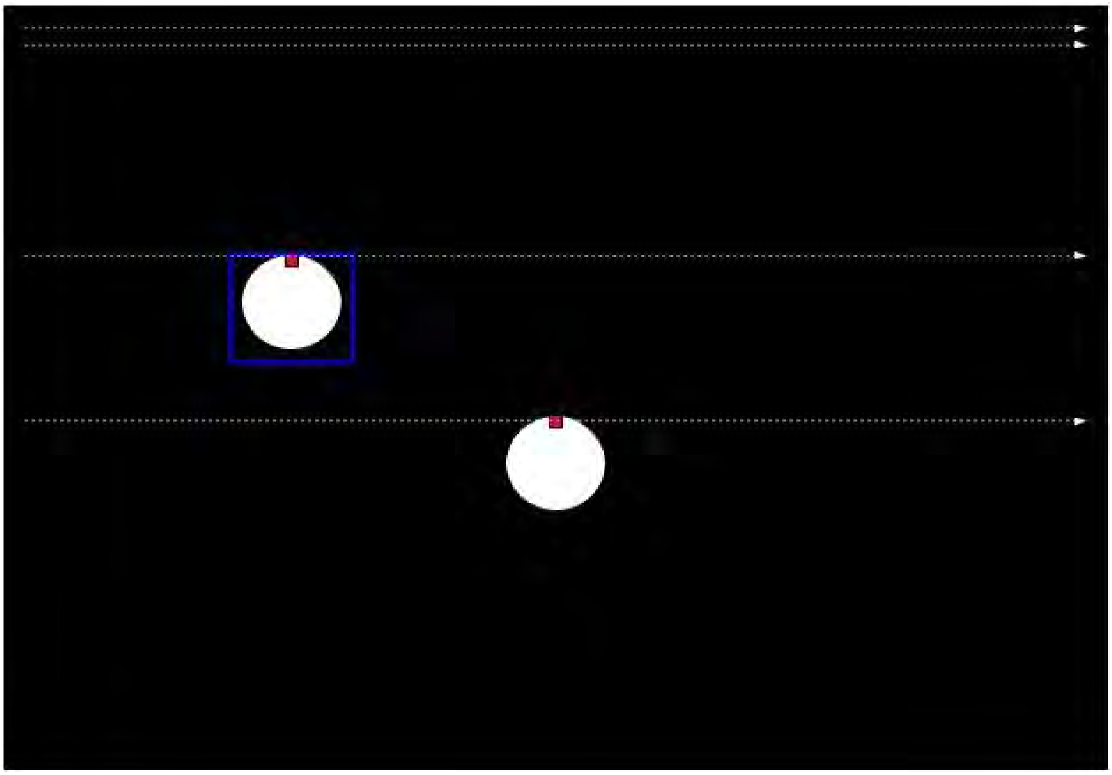


Figure 12: Center of mass tracking algorithm

In Figure 12, the black background with the white dots is the filtered input to the center of mass detector. The dotted white lines show how we scan down the image looking for white pixels. On the top left dot, we detect a single white pixel and draw the window, shown in blue, around the first dot. We continue scanning down the image ignoring white pixels that are inside the window. The second center of mass is found lower in the image. We return the pixel coordinates of the two red squares as the center of masses of each dot.

Our algorithm actually returns the top most pixels of each circular blob as the center of mass. This means that the center of mass we report is actually a few pixels further up the screen than the real center of mass. We are willing to accept this error as the circles are only about 20 pixels in diameter, meaning we expect only an error of about 10 pixels which does not represent a large distance when the screen resolution is scaled to the size of our track.

Overall this algorithm was highly successful. It is very fast and was simple to implement. When we built this algorithm, we acknowledged that if the reported center of masses were not accurate enough, we could simply shift them down by some constant value that was approximately equal to the radius of the white circles and obtain a point much closer to the actual center of mass of each dot. We found that this was not necessary in our final build though.

3.3.3 Car Location and Next Location Calculation

With the center of masses of both LEDs, finding the current location of the car is very easy to obtain as it is just the mean of the center of masses' x and y coordinates. Because we have two center of masses, the divide by two operation to find the mean is effectively free if we use a 1 bit right shift operation.

Finding the next location is somewhat more complicated than finding the current location of the car. To find the next predicted point, we take the vector from the LED at the rear of the car to the LED at the front of the car and add that vector 6 times to the current position of the car. The number of times the vector is added changes how early our car reacts to changes in the road ahead and was found through thorough experimentation. The most challenging part of the calculation is determining which center of mass corresponds with the LED at the front of the car and which is associated with the LED at the rear of the car. This is particularly difficult as the LEDs at the front and rear of the car are identical. Determining which LED is in front is a critical part of this module as arriving at the wrong answer will make the next predicted region behind the car instead of in front of it. This would almost always result in complete loss of control of the car.

The final algorithm updates pointers that keep track of which LED is leading and which is trailing. At the start of the program, the user places the car on the track so that it will go forwards in a clockwise direction and so that it points left on the screen. The user presses a hardware button on the FPGA that asserts to the algorithm that the leading LED is the leftmost one. This provides the initial conditions required for tracking the leading LED. Then, once per frame, the module updates the location of the leading LED and trailing LED by comparing the new center of masses to the last predicted position. The center of mass with the smaller Manhattan distance to the last predicted position is assumed to be the leading LED and we calculate the new predicted position based on this.

This module required a lot of revisions to obtain the final working version. Originally, the module determined which LED was the leading one based on the velocity of the car given by changes in the car's current location. Unfortunately, this method was highly susceptible to noise, and did not work consistently when smoothing of the velocity data was used. The predicted position would often flip to a coordinate behind the car instead of in front. We changed the approach of the module in the last week of the project to use the Manhattan distance heuristic to determine the leading LED. This proved to be a much more robust method. This was a critical decision and was the final piece that allowed us to drive the car around the track repeatedly and efficiently.

3.4 Car Control

3.4.1 Purpose

Once the track image is processed to identify regions, and the car's position is identified the next step is being able compare these two data points and determine which commands must be sent to the car in order to keep it driving around the track. The car control section of the project does this with two main modules. The first is Region Manager which responsible for translating car position coordinates into track regions. The second is the Car Controller which is responsible for determining the necessary corrective action needed to stay on the track and then generating the necessary signal pulses needed to send commands to the car.

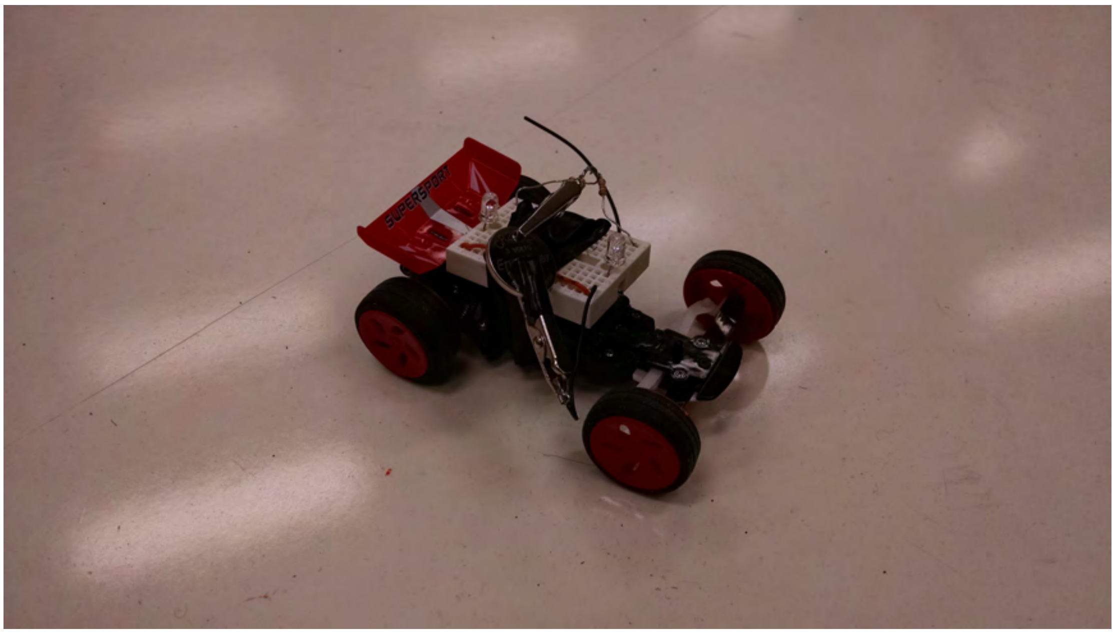


Figure 13: The RC car with IR LEDs mounted on top

3.4.2 Region Manager

The Region Manager's goal is to take a coordinate in the pixel space of the track given by the Car Position Processing modules and translate that to the track region value for that pixel stored in the region BRAM. With this, we can determine if the coordinates representing the car's current and predicted locations are outside of, inside of, or on the track and therefore decide whether a left, right, or straight movement is needed to follow the track.

The Region Manager is implemented with a simple state machine that converts the car coordinates to regions as quickly as possible. Since the region information is stored in BRAM, only one coordinate can be converted to a region at a time so the current position information is processed first followed by the predicted position.

The steps to translate both coordinates are essentially the same. First, a check is done to ensure the coordinate given will actually correspond to a valid BRAM address. If a given coordinate is outside the camera's field of view, the returned region will be outside of the track. This is mostly an issue with the car's predicted coordinate which can be outside of the view of the camera. If the check shows the coordinate is on the screen, then the formula $(x + y * 640) >> 1$ is used to convert it into the BRAM address holding the region data of the coordinate of that pixel. The 1 bit shift is used as an efficient division by two. This is necessary because of our storage of two pixels in each 8 bit wide memory address. We also store the last bit of the operation $(x + y * 640)$ which represents the parity of that address and allows us to determine if the desired 2 bit region value is stored in bits [1:0] or [5:4] of the BRAM address.

Once this process is completed for both the current and next coordinates, the region values for these two are latched into two output registers that will be fed to the car controller. At the stage where the region data is latched into the output

Table 1: Car Controller Decision Making

Current Car Region	Predicted Car Region	Desired Command
Outer Track	Outer Track	Right
Outer Track	Track	Forward
Outer Track	Inner Track	Left
Track	Outer Track	Right
Track	Track	Forward
Track	Inner Track	Left
Inner Track	Outer Track	Right
Inner Track	Track	Forward
Inner Track	Inner Track	Left

registers, we have also implemented a fix for the occasional problem of the camera losing sight of a single LED. When this happens, the car position module raises a flag that causes the Region Manager to automatically set both regions to outside the track. This is interpreted as a right turn command in the Car Controller which helps the car find it's way back onto the track if sight of an LED is lost by going out of view or prevents the car from speeding off the track if the LED happens to be at an angle that prevents the camera from seeing it.

The region manager worked exactly as expected with the exception of one bug which we were never able to explain, the output of the Region Manger corresponding to the next region actually represents the current region and vice a versa. This was fixed by simply flipping the outputs into the Car Controller.

3.4.3 Car Controller

The Car Controller is the final module in our control scheme. It receives the inputs of the current and next region of the track the car is in and based on that controls the output pins on the FPGA that are connected to MOSFETs which are in turn connected to the button inputs of the RC transmitter that the RC car came with. That scheme essentially allows us to electronically control the car with the FPGA.

The Car Controller had two main challenges. First was figuring out what action should be taken based on the current regions of the car. Second was generating the proper timing command pulses to make the car move correctly.

Determining which action to take was simple. The Car Controller simply uses the scheme shown in table 1. These controls result in the car following a clockwise path around the track. Astute readers may notice that the current region is actually unnecessary with this setup. This is indeed true; however, the ability to enable control based on the current position of the car as well was included to allow us to test different styles of control so if we decided that a scheme where the current position mattered we would be able to easily change the operation of the car.

The second challenge of getting the car to obey our commands was much more difficult. The first problem we encountered was working within the constraints of the command protocol used by the RC car. The RC car transmitter works by detecting the state of all the buttons connected to it at the current time and then comparing those states to a table with a number that represents all possible states. This number corresponds to the number of 27MHz “W1” (1KHz 50% Duty Cycle) pulses that the transmitter will then send along with 4 “W2” (500Hz 75% Duty Cycle) start and 4 “W2” stop sequence pulses. A full data frame can be seen in Figure 14. The car receiver is constantly looking for and counting the number of pulses it receives. Based on that number, it decodes a specific command that generates some action, a few of which can be seen in table 2.

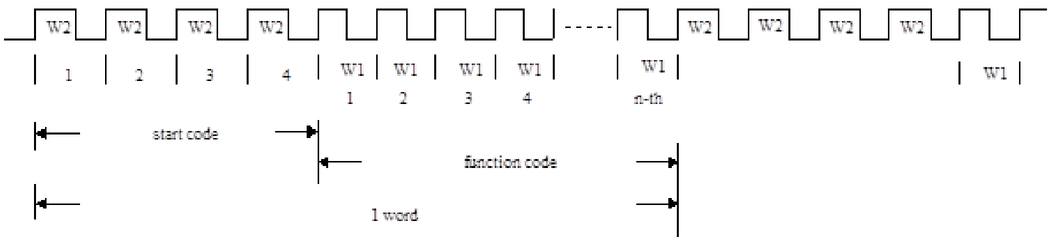


Figure 14: Command Protocol of the Controller

Table 2: Transmitter Command to Pulse Mapping

Number of W1 pulses	Transmitter Buttons	Car Action
4	None	End Code
10	Forward	Forward
28	Forward and Left	Forward and Left
34	Forward and Right	Forward and Right
40	Backward	Backward
58	Left	Left
64	Right	Right

The main challenge that resulted from this is that since different commands required a different number of pulses, different commands actually took different lengths of time to properly send. This not only limited the rate certain commands could be sent at, but also, required the car controller to have some precise hard coded timing values. In order to actually send the different command pulses, a series of different states that held the outputs high for different amounts of time depending on the command were used. Problems were also encountered with the FPGA being able to simulate button presses at a rate that the transmitter, which was designed for the slow and clumsy fingers of children, was able to handle. This problem was solved by adding a command delay state that had to run it’s full duration before another command could be sent.

Another large challenge with the car was its very rapid acceleration, mainly with the forward command (left and right with forward commands resulted in much less power being send to the motors). We found that these rapid accelerations were difficult to control and would often result in the car flying off the track before a corrective command could be issued. Unfortunately, we had very little electrical control over this since the amount of power sent the the motors upon the receipt of a forward command was all contained in integrated circuits on the RC car. Fortunately, Newton's Second Law, $F = M \cdot A$, was quite helpful here as we realized by adding mass to the car, (in this case some shaft collars and a AA battery) we could reduce the acceleration caused by the constant force from the motor. As physics does indeed work, this solution was quite effective at reducing the uncontrollable acceleration caused by forwards commands. However, as effective as Newton's Second Law was at stopping rapid acceleration, Newton's First Law of Inertia then immediately attempted to derail us. We found that successive forwards commands resulted in the car quickly speeding out of control as the car did not have to overcome its initial resting inertia. Our simple solution here was to add a special delay only for forwards commands. This delay was long enough that by the time a second forward command could be issued the car had already slowed down enough for things to stay under control.

One of the key lessons learned for future students considering similar projects is to ensure that whatever hardware you are controlling is actually easily controllable. If we had had a car that allowed us a control the speed and turning angle with more variability, we would have been able to control the car much faster with more perscison.

4 Integration

The modules of this project were designed from the beginning with final integration in mind. Our group knew to expect challenges in the final stages of the project when bringing all of our modules together, so in the beginning of the project we were careful to define very specific interfaces between the three key modules. We wanted to have as little data flowing between modules as possible to prevent contention issues or misunderstandings. For example, the car location and heading module passes just two pixel coordinate pairs to the car controller module. Keeping module interfaces as simple as possible made it possible for our team to stitch together our modules in a matter of minutes rather than days.

5 Conclusion

Overall, our team is very pleased with our final result. We hit our major goal of having the car be able to drive around the track consistently and fairly quickly. The final product is very close to our original vision for the project. We made some changes to our original plan along the way, like using only one camera for both track processing and car tracking instead of two and requiring the user define a solid region on the whiteboard for the track instead of just an outline, but the final product still meets the spirit of our original goals.

There is still work that could be done for this project. With a reliable foundation, we could now implement lap timing to gamify the system, improve the car's control algorithms to make it drive faster, and improve the track recognition algorithm to process more sophisticated track topologies.

In all, this project was an excellent learning experience for our whole team. Our project demonstrates how even at a small scale, autonomous cars with centralized processing require a great deal of sensing and computation to implement even the simplest "bang-bang" control scheme.