Car Eye for the Drunk Guy

Final Report

By

Esteban Bowles-Martinez (estebanb@andrew.cmu.edu)
Michael Mishkin (mmishkin@andrew.cmu.edu)
Tim Kral (tkral@andrew.cmu.edu)

Of

Carnegie Mellon University
Department of Electrical and Computer Engineering
Digital Communications and Signal Processing Systems Design
18-551 Spring 2004 Group 4

Table of Contents

1. Introduction

Every year, thousands of accidents are caused by drunk, sleepy, or elderly drivers who don't realize when their car is drifting out of their lane. Car Eye for the Drunk Guy is a system intended to help these drivers to realize when they are driving out of their lane with a warning indicator similar to the rumble strip on the side of many high-ways.

1.1 The Problem

The goal of the project is to analyze video from a car-mounted camera to determine if the car is driving safely. Our software package reads in these videos and maps out the road on each frame. The map keeps track of the location of road lines and determines whether these lines are being crossed. In order to make this determination our project must first locate these lines and then parameterize them so that their location can be analyzed. If these lines are within a certain range then a lane change has been detected and the driver is alerted.

1.2 Previous Work

No work has previously been done in 18-551 along these lines however CMU has been working with the National Automated Highway System for 20 years on developing autonomous driving systems in a project called Navlab which are the autonomous cars, vans, and busses that were designed by Carnegie Mellon students. There were a few projects done in the robotics department using Navlab that are very similar to ours[1,2] in that they focus on mapping the road and locating lines on the road. These projects both use algorithms and technologies which are based on a generalized Hough Transform as ours is. Another focus of these projects tracking of vehicles on the road with a 2d bound box located with a Kalman filter. One of these projects[2] implements a robust road follower called RALPH which keeps track of the edge of the road and a center line for guidance in the vehicle's steering. RALPH resamples a trapezoidal shape from the road in order to eliminate perspective then uses template based matching techniques to find parallel features. This process is essentially a more specific form of a Hough Transform with a very specific template for the road.

A major difference between our project and these projects is that ours is done using the C67 which is not on board the car so our video feeds were recorded prior to

their processing. As a consequence, the processing can not be done in real time and we could only approach our goal of modeling real time processing.

2. Data Collection/Test Set

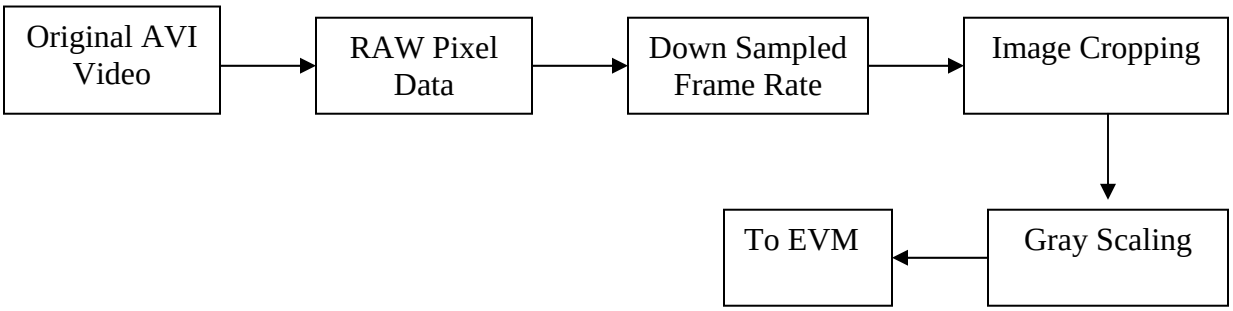
We performed our own video collection by recording video from a moving vehicle while it traveled down a highway in clear weather conditions. The camera used was a Logitech QuickCam® Pro 4000 USB 2.0 digital video camera which was a relatively cheap option (approximately $75.00 on http://www.amazon.com) and was praised on numerous web sites for producing high quality video even in poor lighting conditions. The camera was mounted just above the center of the windshield of a Toyota Camry (approximately 1.75 meters off the ground) and at an angle of approximately 20 degrees from the horizontal. Several one- to two-minute videos were recorded on Interstate 376 outside the city of Pittsburgh to accommodate different driving situations: changing lanes, driving around a curve, passing other vehicles, tailgating and combinations of these. The same situations were also recorded on video at night on the same stretch of road.

After recording, our test set consisted of 24 daytime videos and 16 nighttime videos. However, the nighttime videos proved to be worthless because of the low amount of light provided by the headlights of the vehicle. It was impossible to see road and lane lines 5 feet beyond the front of the vehicle and this would have made it very difficult to process these videos and find such lines in the road. Thus we discarded the 16 nighttime videos and had the test set stand at 24 daytime videos.

3. Video Preprocessing

The original recorded video, which is stored in AVI file format, is 320 by 240 pixels, 24-bit color and runs at a speed of 30 frames per second. Preprocessing this video on the PC is necessary for two reasons. One, the video needs to be broken up into individual frames so that the EVM can effectively process one frame of the video at a time and return its results. Second, we realized that processing such a large amount of data (~415 Megabytes for a one minute video) would render the system incredibly slow and thus preprocessing is needed to cut down on the amount of data being sent to the EVM for processing.

With these two reasons in mind, the video preprocessing takes an AVI video and splices it into individual frames. After getting a frame, the PC code takes several steps to reduce the amount of data actually sent to the EVM which includes reducing the frame rate, cropping out areas of the frame that are not needed and reducing the bit depth. In the end the individual frames that are sent to the EVM are 120 pixels high by 320 pixels wide with 8 bits per pixel. The video processing follows the steps shown below:



3.1 Isolating Individual Frames

The AVI files are broken up into individual frames using the AVIFile interface that comes with Microsoft Visual C++ 6.0. Using this code required that vfw.h be included as a header and that the static library vfw32.lib be included in the project. The process required several steps. First the AVI library is initialized with a call to the `AVIFileInit()` function and immediately after this an AVIStream is opened from the target file using the `AVIStreamOpenFromFile(...)` function. Now the length of the video (in number of frames) can be determined by the `AVIStreamLength(...)` function. Running a loop from the first frame to the final frame, the PC code opens a frame stream with the `AVIStreamGetFrameOpen(...)` function, gets the next frame with the `AVIStreamGetFrame(...)` function and releases the frame stream with the `AVIStreamGetFrameClose(...)` function. Information on all of these functions and more can be found on the MSDN web site at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_multimedia_functions.asp. Code for this process looks similar to the following:

```
#include <vfw.h>
```

```
PAVISTREAM pavi;          /* AVIStream pointer */
PGETFRAME pgf;            /* Frame Stream pointer */
LPBITMAPINFOHEADER lpbi;  /* Bitmap Header Info pointer */
LPSTR pixelData;          /* Pointer to actual pixel data */

/* Initialize AVI library */
AVIFileInit();

/* Open AVIStream from target AVI file
   pavi - AVIStream pointer into which stream info is read
   filename - Name of target AVI
   streamtype - Type of stream to be processed (in this case
                streamtypeVIDEO processes a video stream)
   lparam - 0 here indicates that this is the first call to
            this function*/
   mode - The mode under which this AVI should be opened
   handler - NULL here indicates that the registry will
             choose the defauilt handler */
if(!AVIStreamOpenFromFile(&pavi, FileName, streamtypeVIDEO,
                          0, OF_READ, NULL))
{
error...
}

Loop i = 0 through AVIStreamLength(pavi)
     /* Open the frame stream
        pavi - AVIStream pointer initialized by
               AVIStreamOpenFromFile
        LpbiWanted - NULL here indicates that the system
                     Uses the default format */
     if(!(pgf = AVIStreamGetFrameOpen(pavi, NULL)))
     {
     error......
     }

     /* Get the next frame
        pfg - Pointer to frame stream initialized by
              AVIStreamGetFrameOpen
        FrameNumber - The number of the frame that is to be
                      Retrieved */
     lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(pgf, i);


     /* The pixel data is contained in memory right after
        the bitmap header info.  So we can point to the
        pixel data by jumping forward the size of the
        bitmap header info structure. */
     pixelData = (LPSTR)(*lpbi) + *(LPDWORD)(*lpbi);

     ...process pixel data...

     /* Close the frame stream */
     AVIStreamGetFrameClose(pgf);
```

```
EndLoop

/* Release the AVIStream */
AVIStreamRelease(pavi);
/* Close the AVI library */
AVIFileExit();
```

Notice that after the loop finishes that the code must release the AVIStream pointer using the `AVIStreamRelease(...)` function and must close the AVI library using the `AVIFileExit(...)` function. Also notice that after the call to `AVIStreamGetFrame(...)` that the actual pixel data is stored right the bitmap header info structure in memory. Thus the code points to this data by skipped over the size of this structure in memory. Once this is done, the pixel data can be manipulated and/or processed. It is at this point that we begin to cut back on the amount of data sent to the EVM.

3.2 Reducing the Frame Rate/Down Sampling

The usefulness of Car Eye for the Drunk Guy depends heavily on the driver's reaction time. Processing video at the original rate of 30 frames per second assumes that the driver can react in $1/30^{th}$ of a second, which is much quicker than anyone can be expected to react. In talking with Carlos Reverte of the Carneige Mellon Robotics Institute, we learned that 10 frames per second is more suited for a reasonable driver reaction time. Thus we chose 10 frames per second as our target down sampled rate. Within the PC code, down sampling takes place by isolating every third frame. In terms of the code above, this means that the loop takes steps of three instead of one.

After looking at the output videos at 10 frames per second we saw that the frame rate was still faster than what one would be able to react to in traffic, so we decided that an even lower frame rate would be sufficient. Through experimentation, it was found that at 6 frames per second, the delay between frames is sufficiently noticeable that drivers would be able to begin to react in the time between frames. At 6 frames per second, the PC only sends every fifth frame to the EVM for processing.

3.3 Image Cropping

The height and angle of the camera on the car was engineered such that the car's hood is just below the bottom of the image frame. This maximizes the visible area of the

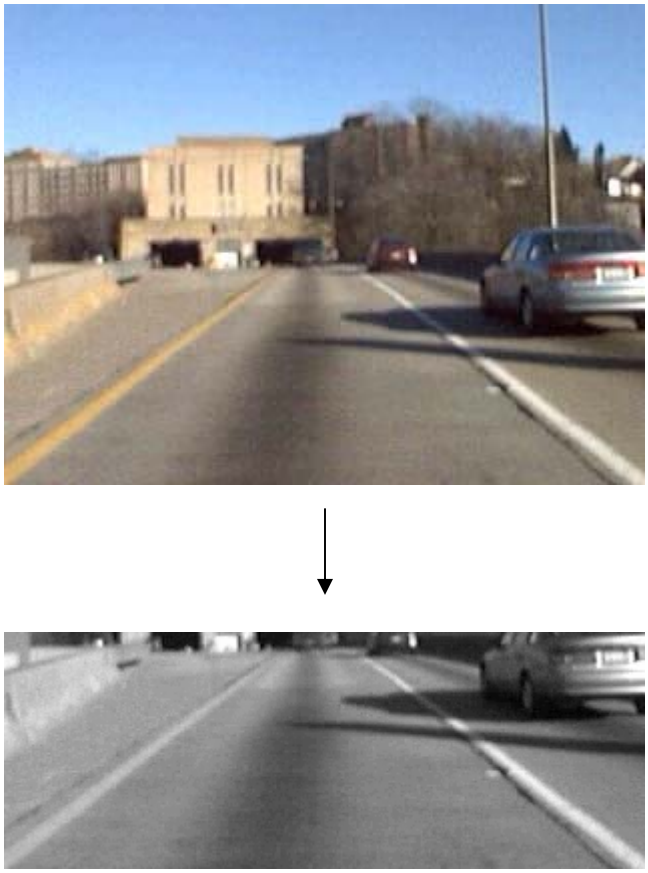
road, giving more useful data with which to work. Still, the top half of the video is above the horizon and since we are only interested in the part of the image that contains road, we crop out the top half of the image before sending it to the EVM.

Such cropping serves two purposes. The first is that it makes the Hough transform on the EVM more accurate by eliminating signs, buildings, trees, sky and other objects which could be falsely identified as possible road lines. The second is that it halves the amount of data needed to be stored on the EVM thus saving memory.

Perhaps the greatest thing about image cropping is that it actually reduces the amount of processing done by the PC. Image cropping takes place at the same time as gray scaling, which is discussed in the next subsection. In order to gray scale a new array of pixel data is created which stores gray data pixel by pixel. The loop that does this processing only runs through half of the image which effectively chops the image in half by only storing half of the image data in the new array.

3.4 Gray Scaling

When the project was started, the idea to gray scale seemed obvious because it would save memory on the EVM. The original 24-bit images take up three times as much memory as 8-bit grayscale. The original gray scaling algorithm called for averaging the values of each color channel. Later we realized that everything in the road is gray, therefore each channel would have essentially the same value. Using this fact, we decided that we could simply take one of the color channels from the original image and use it as our grayscale. We chose the blue channel because yellow shows up brightest in this channel, so yellow road lines will have a higher contrast against the road surface. Below is an example of an individual AVI frame before and after preprocessing:

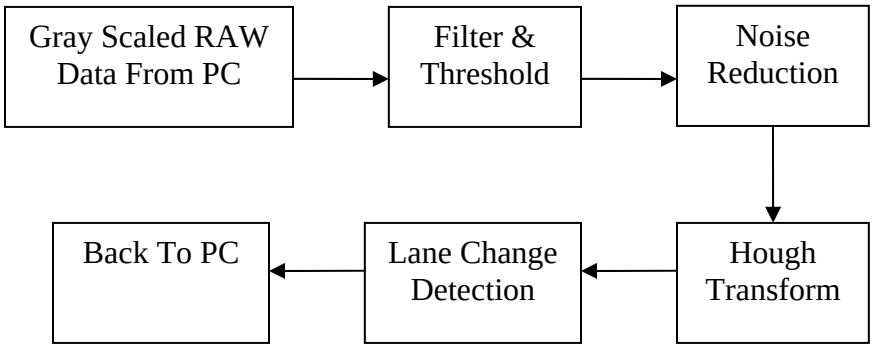


3.5 Debugging

While debugging the preprocessing code, it was useful to save the individual AVI frames as bitmaps to disk in order to view them and make sure that the code was working properly. This also turned out to be useful in debugging the EVM code because we could load an individual frame from disk and test it instead of processing an entire video worth of frames. C code that saved bitmaps to disk was found at Jeff Heaton's web site, http://www.jeffheaton.com/source/sbitmap.c.

4. Algorithm/EVM Processing

After an individual frame is preprocessed, the gray scaled pixel data is sent to the EVM for processing. EVM processing is completed in four steps: filtering/thresholding, noise reduction, Hough Transform and lane change detection as illustrated below:



4.1 Filtering/Thresholding

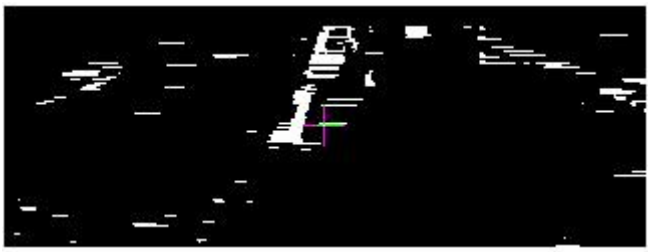
One very important thing that our program takes into consideration when identifying lines is contrast between the road and the lines painted on it. When we started the project, we knew we would have to deal with the problem of shadows in the road. Our program assumes that the brightest things on the road are the lines.

Our first approach to addressing this was to find the mean brightness of the image, and adjust the brightness of every pixel so that every frame, regardless of the presence of shadows, would have the same mean brightness. This uniform brightness across frames is necessary because we need to use the same threshold values in every frame. We then black out any pixels that are less bright than our threshold value, which we determined through experimentation. The results of this method are usually very good, even in images with many patchy shadows from trees directly over the road. However, it fails when an entire road line is completely in shadow and the rest of the image is shadow-free. We decided to go with another approach.

We came up with our second approach when we noticed one frame with a car's shadow completely covering a road line. The line was not detected because the brightness of the whole region in the shadow was below the threshold value. We needed an approach that looks at contrast between nearby pixels to find bright areas instead of contrast between a pixel and the average brightness of the image.

Our program uses a more localized approach to determining which portions of the road are the lines. It looks for changes in pixel brightness along each horizontal row. A [1, 0, -1] filter is used along with a threshold to find when a significant change in brightness has occurred. Through experimentation, we found that a change in brightness of 10 levels out of 256 is significant. If this change in brightness is found, the next

twenty pixels are examined to find if the row has darkened again. If such a decrease is found then this segment of the row is considered to be a possible line and is marked as white in the output image. This same procedure is applied to every row in the input image until a black and white output image is generated which consists of white pixels wherever potential lines are detected. Essentially, the filter looks for regions of brightness less than 20 pixels wide. This keeps wide bright areas, such as white cars, from being detected as possible road lines.



4.2 Noise Reduction

The resulting image contains the correct road lines, but there is also noise scattered about from a few areas of the road that also happened to fit our road line criteria of bright and narrow. Because our filter only scans horizontally, the noise tends to be white horizontal lines one pixel tall. These lines are removed by a [1; 2; 1] vertical blur filter, which takes out any white pixels with black pixels above and below them. The resulting image is a much cleaner representation of where the road lines are expected to be. It is now ready to be passed into the Hough transform for parameterization of the detected lines.

4.3 Hough Transform

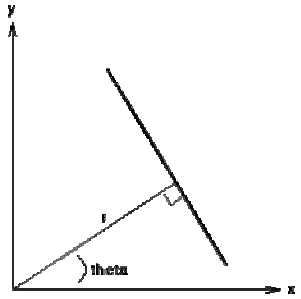
Parameterization of the lines is done with a Hough transform, which we use for detecting straight lines in an image. Typically, the input to the Hough transform is a binary black and white edge-detected image, however, this is not the best input for our

application. Since the road lines themselves are detected rather than the edges of objects, the image passed to the Hough transform is likely a fairly clean image in which the road lines are white pixels and everything else as black pixels.

The Hough transform scans through the image looking for white pixels. When it finds one, it plots a curve in the Hough space representing all of the lines that could possibly pass through the image. This process is called accumulation.

4.3.1 Line Parameterization

There are a few ways that the road lines can be parameterized. Any equation for a line can be parameterized by setting the constants in the equation as parameters. Probably the best-known linear equation is slope-intercept form in which a line is described by the equation y = m*x + b. If a Hough plane were to be drawn for a line in this form then the two axes of the coordinate system would represent the variables m and b such that every point in the Hough plane represents a different line in the x-y plane. The coordinate system used for our Hough plane is based on the polar equation for a line, r = x*cos(theta) + y*sin(theta). This equation describes a line at a distance of r from the origin, oriented at an angle theta as shown below. This coordinate system is better for finding lines at evenly spaced angles, whereas the slope-intercept method would be less effective as lines approach a vertical slope of infinity.



4.3.2 Determining the Best Line

When the accumulator is passed a set of coordinates describing a point in the x-y plane of the road image, the accumulator plots a point in the Hough plane for each angle of line that could possibly pass through that point. The accumulator is run on every white pixel in the line-detected image so that once accumulation is completed, any points in the Hough plane with values greater than a threshold value are considered good approximations of the road lines. As the accumulated Hough plane is being searched for

maximums, the current maximums are stored in a linked list and any points above a relatively low threshold are compared to the current maximums such that in the end the linked list contains two nodes, one for the strongest line on each side of the image. These are the lines we use to represent the road lines in the final image.

4.3.3 Optimizations and Assumptions

We know that we are looking for one road line on each side of the lane, so we split the image and look at each half independently. This ensures that noise from one side of the image won't affect line detection on the other side. It also allows us to reduce the number of angles our accumulator looks at.

We also consider the angles that road lines can have when viewed from the camera's perspective. Points on the left half of the images are only accumulated for angles between 25 and 90 degrees and angles on the right half are only accumulated between 90 and 155 degrees. Angles from 0 to 25 and from 155 to 180 never need to be accumulated since these angles are too sharp to be road lines. Narrowing down the angles in our search increases our program's efficiency, this way CPU cycles are not wasted on angles that we are not interested in.

One additional optimization that we tried out was to only accumulate the bottom portion of the image. This approach assumed that the lines near the bottom of the image were so much more prominent than those at the top of the image that we should only search for lines going through the bottom of the image. Once lines are detected in that portion, we checked the rest of the image for points along those lines. Although this would use fewer cycles, it turned out to be much less accurate in most cases. Since the road lines are often dotted lines, there was not always enough of the line in the bottom portion of the image to get a high enough Hough value to indicate a road line. We decided to sacrifice efficiency for accuracy and stick with an exhaustive search of the Hough plane.

4.3.4 Other Possible Methods

There happens to be another way to accumulate for the Hough transform. Our method goes to each pixel and checks every possible angle through that pixel. Another way to do this is to project lines at every angle across the image. The difference between

the two methods is that they go at the problem from different direction. Our method goes through the image pixel-by-pixel only accumulating for white pixels thus only accumulating where accumulation is necessary. The other method goes through the Hough space angle-by-angle looking at each possible line and finding a total for the number of white pixels along that line in the original image. With this second method many false lines will be searched and although the search will be in constant time it is still less efficient than our method. Because our images have large regions of 0 pixels, this method would require many unnecessary calculations. Additionally, our line-finding filters do a good job of eliminating parts of the image that are not road lines, so most of the pixels that the accumulator looks at give us useful information about where the lines are. The projected angle method would be more suitable if we wanted to find lines in a more randomly drawn image.

4.4 Lane Change Detection

Since the Hough transform traces out the lines of a lane, it is easy to determine when the car is changing lanes. Due to the perspective of the camera, when the car is centered in the lane the road lines appear to lie at the two bottom corners of the image and converge at the center of the top of the image. As the car leaves the center of the lane, one of the lines will become more and more vertical and the bottom of it will move towards the center of the image. Thus a lane change is detected when the bottom of a line intersects the bottom of an image between the 80^{th} and 240^{th} pixels. When this occurs, a rectangle appears in the upper left corner of the image to show the user that a lane change is detected.

5. Video Postprocessing

After a frame is processed, the EVM draws the two best lines for either side of the lane and a possible warning square in white on a black background. Below are two images which show the two cases returned by the EVM:

Centered in lane

Changing Lanes

This simple image is then sent back to the PC for postprocessing. Before a frame is gray scaled during preprocessing, its color data is saved in a buffer to await Hough lines from the EVM. Once these lines are returned to the PC, they are placed over the saved color data to produced a new video frame with Hough Transform lines. This is done by ORing the Hough Lines with the blue channel of the original color data. In this way all data that appears in the black regions of the Hough Lines will pass through unchanged while the white lines will appear as blue lines in the color image.

The PC code then uses the CAviFile class obtained from http://www.codeproject.com/bitmap/createmovie.asp to string the new frames together and create a new AVI file that can be viewed by the user. CAviFile has an extremely simple interface that has only two methods: a constructor and an append frame function making it easy to create new AVI files from bitmaps.

6. Synchronization of EVM and PC

One of the most difficult steps involved in designing fully functional code is synchronization of communications between the EVM and the PC. Essentially, the job of the PC through each iteration of operation is to find the next frame from the video and send it to the EVM. The EVM then processes the frame and returns an image containing only the road lines and possibly the warning indicator. While the EVM is calculating this image, the PC is preparing to transfer the next frame to the EVM. Once the EVM is done

processing, the image that it returns is received by the PC and immediately the EVM is sent the next frame. The PC then inserts the returned road line image into the previous frame of the video and inserts this frame into the output video. The PC then moves on to finding the next frame of the input image. Throughout this process, the PC sends and receives images from the EVM using HPI transfers while PCI transfers are used on the EVM side.

7. Speed and Memory

Our program requires approximately six seconds and up to 400 million cycles to process some of the more complex frames. 300 million of these cycles are spent in the accumulation phase of calculating the Hough transform. Most of our attempts to improve efficiency in the Hough transform were merely ways of calling the accumulator fewer times but since in doing this accuracy was sacrificed, we decided to do exhaustively accumulate all points that are potential lines. The accumulation process was improved slightly in that the range of angles that was accumulated was roughly halved but this step still seems to take an exorbitant amount of time.

One of our methods for improving efficiency was switching the memory locations of certain variables from SDRAM to SBSRAM since SBSRAM is significantly faster. In order to do this we had to eliminate variables such that all of the malloced data would fit on SBSRAM which has a smaller memory size than SDRAM. With these variables gone we had to shuffle data back and forth between the remaining variables which turns out to work rather well since at any given time there is only one input image and one output image.

8. Results

8.1 Successes

Our program is able to find the lane boundary lines in almost every case. Several things that we were concerned about early on in the project ended up working out fine. These include issues with shadows, road curves, traffic, and road surface imperfections.

Because of the dynamic way that our program handles shadows and contrast, it is able to find the lines even when it is difficult to see the lines with human eyes. As long as the road lines are identified correctly, it is easy to determine when the car is out of its

lane, since this condition depends on the angle of the line as found in the Hough transform.

Another concern when we began this project was how it would handle curves in the road. This turned out not to be an issue, since the curves in the freeway are very gradual so they still appear straight enough for the Hough transform to find them without difficulty. There is no noticeable difference in how well our program identifies lane changes when the car is changing lanes while the road curves.

Since we deal with real driving conditions, we have to deal with traffic. Since our program only looks for narrow, bright objects when looking for road lines, most cars do not affect our program's effectiveness. Occasionally part of a white or shiny car will be identified as a possible line, but usually the actual lines are more prominent in the Hough transform. Another issue with traffic is cars blocking the road lines. For a car to take up enough area in the image to have the possibility of blocking road lines, we have to tailgate it extremely closely. Still, it would need to be changing lanes in front of us to completely block one of the road lines. Such a situation is very rare in real driving.

The roads in the Pittsburgh area are in horrible condition. We originally thought that road surface imperfections such as potholes, oil marks, and road repair lines would be difficult to deal with. However, these all show up as darker areas of the road, so when our line-finding filter scans the rows, they are always below the brightness threshold. Basically they are treated the same way as shadows.

8.2 Failures

While our program usually finds the road lines accurately, it does make occasional mistakes. Sometimes our program will choose one of the lane boundary lines for another lane or the edge of the road as the best line. This can be solved by comparing the location of the current lines to the locations of the previous frame's lines and requiring that the current line is close to the previous line. Also there are some errors where a line at a completely wrong angle is identified as a road line. This can be solved by requiring that the current line's angle has not changed very much since the previous frame. By requiring these two conditions our program will almost never identify the wrong line.

Another type of failure is when the program does not find any lines that are strong enough to be identified as road lines. This is usually because the paint has faded so much that the line has all but vanished, or because the line has been covered by black road repair marks. When there is no significant dark-to-light contrast where the line should be, our program will not find a line. The only way to address this is to reduce the line-finding filter's threshold, which will reduce our program's accuracy in finding real lines.

9. Conclusion

We are pleased with the results of our project. It works superbly in nearly every situation. Sometimes it sees the lines even better than a human driver. However, we are disappointed with our program's speed. It takes a very long time to process each frame. It would be useless in an application that requires real-time performance. It would be interesting to see this run on faster hardware with more memory. It would also be interesting to see this system integrated into a real car driven by real drunk drivers.

9.1 Future Improvements

There are several ways our program could be improved if only we had more time. One way is to change how we generate our grayscale images. For our grayscaling, we only take one color channel into account. It may be possible to get better contrast by mixing the channels with different levels of each color to find the mix that brings out the most contrast between the road and the lines. Another possible way to improve the speed of our program would be to use a sine look-up table instead of doing trig calculations from within the accumulator for example which is called sometimes thousands of times for a frame. This would save a huge amount of processing time because we could simply perform the trig calculations once when we first start our program then refer to the results when we need them without having to recalculate the values.

10. References

1) F. Dellaert and C. Thorpe, “Robust car tracking using Kalman filtering and Bayesian templates," in Proceedings of SPIE: Intelligent Transportation Systems, vol. 3207, October 1997.

2) F. Dellaert, D. Pomerleau and C. Thorpe, “Model-Based Car Tracking Integrated with a Road-Follower," International Conference on Robotics and Automation, May 1998.

Both of these references were useful in terms of general knowledge about current ongoing research about automated driving systems. Neither provided any information on Hough Transforms or the other algorithms used in this system. Furthermore, both focused on the issue of car tracking while road and lane detection were more of a side note.

3) AVIFile Interface, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/multimed/htm/_win32_multimedia_functions.asp

This web site provides extensive information about the AVIFile interface and its functions. Each function is thoroughly explained with arguments (which themselves are thoroughly explained), return values and header files. Other places on the MSDN site also include extremely useful code examples that use the various functions mentioned. One can also read about these function by looking at the help from Microsoft Visual C++ 6.0.

4) Saving bitmaps to disk, http://www.jeffheaton.com/source/sbitmap.c

This web site provides a C file that will save a bitmap to disk. Such code was useful for debugging the preprocessing code on the PC and the processing code on the EVM.

5) Creating a movie from bitmaps, http://www.codeproject.com/bitmap/createmovie.asp

The Code Project provides a lot of useful information about creating movies from bitmaps. It gives a detailed description of the code and how to use it and provides links to download the necessary libraries to get the code working in VC++ 6.0.