Computer Vision and its Application in Self-Driving Cars

Benjamin Cutilli
Haverford College (Advisor: Professor Wonnacott) / Bryn Mawr College (Advisor: Professor Eaton)

Computer Vision and its Application in Self-Driving Cars

Benjamin Cutilli
Haverford College (Advisor: Professor Wonnacott) / Bryn Mawr College (Advisor: Professor Eaton)

1. PREFACE

For quite some time, I have been interested in performance cars and racing such as Formula 1, and seeing the progress that has been made with self-driving cars, it is interesting to think about if it is possible to build a self-driving race car. Based on various challenges that have been set up and news going around about Google's self-driving car, it seems like today's self-driving cars focus strictly on getting from point to point safely, which is why these self-driving cars have been made. But why not push the limits, where these cars are getting from point A to point B very quickly without any speed limits or obstacles in the way except for the distance from the start to the finish line? Conventional driving algorithms seem to stay in the middle of the road, mimicking a person walking down the stairs with their hands on the rail. He or she can only move parallel to the rail. So what happens when the fastest way down the stairs is not following the outer rail? What if there is a bend in the opposite direction, and the fastest way to get down the stairs is to cut across the staircase, clip the apex, and swing wide to maintain grip and speed? This is starting to sound like a Formula 1 race, where the drivers do not necessarily hug the corners as much as they can, as they will lose enough speed to destroy the benefits minimizing the distance around the corner. Instead, we see Formula 1 (F1) drivers swinging wide before a turn to maintain speed during the corner, and in most cases, the higher speed outweighs the extra distance the car may have to travel. So far, most of the challenges have been similar to the DARPA challenge where the car has to stay safely within natural barriers to ensure the safest route for the car (Thrun, et al., 2006), or the DARPA Urban challenge, where the natural barriers are replaced with road lines and obstacles like pedestrians (Montemerlo, et al., 2008).

Companies like BMW have cars that drive race-style, but it is mostly based on GPS devices and a track is programmed into the car's memory with the fastest route to travel (Bailey, 2011). This is not computer vision. I believe that it would be interesting to see if it is possible (and it should be) to create a computer vision system that seeks the apex in a corner, calculates its distance, and generates a F1-style line around the bend. This gives way to an interesting take on self-driving cars, a view seemingly eschewed for the sake of safe travelling. However in emergency situations, it can be very vital. Notions like edge detection and perspective can be used to make these distance calculations. The goal is to use the data presented by the camera about a turn, and use it to find the critical points so that the car may properly navigate the bend. All of this sounds simple, but since the camera on the car is not overhead, or bird's-eye view, but at an angle to the road, the computations can become fairly complex.

2. SELF-DRIVING CARS

Self-driving cars such as those that participate in the DARPA Challenge and Google's self-driving car use a host of sensors to determine the best path to take. The team that won the DARPA Challenge, which built a car to successfully navigate a desert, used a combination of laser range finders and visual sensors. The range-finding sensors served in developing a three-dimensional map of the car's surroundings. This helps to track objects that are on the level of the rangefinder, such as walls and other cars (specifically, in the desert, it helped to keep the car in the center of a trail). Thrun states, "[the car] integrates laser data over time into a 3-D point cloud…[which is] then analyzed for vertical obstacles, resulting in 2D maps" (2010, p. 101). Rangefinder techniques do not usually work for long distances, which is where visual support comes into play. Through optical flow and pattern recognition, it was possible for Thrun's team to program the car to look ahead and see what terrain quickly approaching is terrain capable of driving on. In the case for the DARPA Urban challenge, rangefinder techniques were crucial to determine positions of vertical objects such as cars and pedestrians on the road, but, as far as navigating the road itself, range-finding is not reliable in finding the bounds of the roads, and does not help at all in determining the positions of the lines on the road. In the urban challenge, Thrun's team relied on infrared values obtained by the laser to determine where the lines of the road were. GPS helps to determine how far the car is to a far-away object, and since range-finding techniques are very depth-limited, but accurate, the error potential of a meter for GPS is reduced to a few centimeters with the laser range-finder. This technique is called localization (Thrun, 2010).

Software Structure

This team's car, named Stanley, was structured, with respect to software, in four different levels. The most basic level is the sensor interface, which is simply the code written to gather data from all sensors and put it into a useable data structure. To clarify, no data analysis is taking part in this layer, only sensor surveying. The data gathered in the sensor layer is abstracted in the perception layer, which, as Thrun puts it in one of his video lectures, is where are the "magic" happens. The data is analyzed for positions of obstacles and the car's current state is determined through positional filters, such as a Kalman Filter, which determines speed, orientation, and position from sensor data, constructs two-dimensional and three-dimensional maps, and, as explained previously, uses these maps "to find the boundary of a road, so that the vehicle can center itself laterally" (Thrun, et al., 2006, p. 667). Using this data, the planning and control layer plan the route for the vehicle to take, and, knowing that route, determines the inputs for the throttle, steering, and brake. This layer also has to take in car data, such as the gear that is selected. The control portion of the car is simply a finite state automaton, though its construction is not explained. The two last layers are the user interface, which is where humans interface with the planning and control layer, and the vehicle interface, where the control layer interfaces with the car's controls directly (which is the throttle, steering, and brake) (Thrun, et al., 2006, pp. 666-668).

Hardware Structure

In order to determine the best route to take, sensors that gather data describing the terrain ahead, lasers in Stanley's case, are oriented in a sweeping pattern ahead of the car to determine distance and depth. The three-dimensional data from the laser is transformed into a two-dimensional grid, which is the "drivability map" (Thrun, et al., 2006, p. 669). Furthermore, the three-dimensional data is used when detecting obstacles. If two close points from the laser have a vertical difference (vertical being from the ground to the sky) that exceeds a certain value δ:

$$(1) \qquad |Z^i - Z^j| > \delta$$

then those two points seem to represent a "tall" object, which is classified as not drivable. The x-y locations of these points are marked as not drivable on the two-dimensional drivability map, or, as Professor Eaton calls it, an occupancy map. This method is not always reliable, as the car has a tendency to pitch and roll, so a Markov model is applied to the data, which uses data over time to "smooth" out errors (Thrun, et al., 2006, pp. 669-671). In the DARPA Urban challenge, Thrun's team developed a car, named Junior, to navigate simulated public streets, which is closer in concept to this paper's goal, but not very much further away from how Stanley determines its bounds. Essentially, Junior has a set of lasers that send out beams parallel and close to the ground, so that the distance to the curb can be calculated and a proper route avoiding the curbs can be determined (Montemerlo, et al., 2008, p. 7).

Path Planning

Once this data has been determined, a proper path to take needs to be calculated, and, surprisingly, simple search algorithms are the methods by which a path is planned. Dynamic programming is used typically for what are called "structured" environments, such as a road. However, in order to navigate "unstructured" environments, Thrun's team used the A* algorithm, which is a search algorithm comprised of a simple cost determination based on a previous path plus a heuristic function that selects the next node to be searched by underestimating the distance of that node to the goal (Thrun, 2010, p. 103). Essentially, the robot searches for a path, only following the one with the least amount of cost, and added to that cost is a heuristic function that is admissible (which means that the heuristic never overestimates the cost of a node's succeeding paths).

In order to plan these paths, it is clearly necessary to somehow gather pertinent information about the car's surrounding environment, and to do so, we are going to take a look at computer vision, or essentially, obtaining environmental data using cameras.

3. COMPUTER VISION

The basics of computer vision start with defining an image. An image to humans is extremely complex. It involves color, contrast, shading, focus, and many other factors. As humans, we see these elements as one, yet we still analyze a picture based on those specific features. Color and contrast helps to characterize an object and to define an object's bounds, and shading and focus gives three-dimensional perspective. So it makes sense to try to map a human's vision system to computer code, as we are most familiar with this system. Ballard and Brown break down the important segments of an image in their book, *Computer Vision*. They describe computer vision as a range of representations that connect input and output, in four parts: iconic (what we as humans see), segmented (finding the edges of an image), geometric (three-dimensional data), and relational (the position of objects relative to others). Typically in computer vision, we try to map the iconic perspective to one of the three other perspectives (or even as much as all three in some applications). The relational perspective is arguably the most important perspective out of all three of these non-iconic perspectives. It requires a lot of information in order to associate certain objects in a picture with other objects. For example, a picture of a house with trees surrounding it should not be analyzed in a way such that the trees are seen as part of the house. In order to do this properly, a computer needs to be able to tell where objects start and stop, and where they are in a three-dimensional plane, something that can be achieved knowing the information gathered in the segment image and the geometric image (Ballard & Brown, 1982, pp. 6-9).

Edge Detection

So, in order to derive at least the segmented image, we need to be able to find the edges. Inherent to nature, we cannot see objects that are of the same color as its background. For example, a chameleon changes its color to blend in with its background so that it may avoid its predators. Runners, on the other hand, wear reflective gear and neon clothes to make themselves stand out while running on roads. It is the drastic difference in color and contrast that allows humans to recognize different objects. As it turns out, applying that kind of method to computer code is not all that difficult. Ignoring color for now, a computer can take a grayscale image and see where the differences lie. In order to do that, it is best to find where a change in color happens rapidly, as an edge would be where an object "ends" rapidly. This is not such a complex computation as it turns out. Specific to computers, images are typically arranged in values that represent a pixel's red intensity, blue intensity, green intensity, and sometimes alpha intensity, or transparency, associated with the pixel. For our purposes, we are currently interested in the values of the primary colors. In edge-detection, though, we will not be interested in them at all, as it is oftentimes standard practice to map those values onto one grayscale value. Going across horizontally (and eventually vertically) on an image, a computer can observe the rate at which the grayscale values change. A slow change is of no interest to the algorithm, but a quick change, or a high-magnitude grayscale derivative, past a certain threshold (to be defined by the programmer) signifies an area where an edge may be. Specifically, the equation for the derivative, or change in intensity from one pixel to another, is:

$$\Delta I = I_n - I_{n+1} \quad , \tag{2}$$

where I is the intensity of a pixel, and n is a pixel (Russell & Norvig, 2010, p. 936).

Computing ΔI is easy to do, as it only requires subtraction. However, as camera technology is not perfect (and neither are the surfaces of objects), it is not quite as simple as that. Noise in the image is inevitable. One way of smoothing out the image is by using a Gaussian probability distribution. It is possible to smooth out an image, one pixel at a time, based on taking the average of all the pixels around this one pixel, picking the closest neighbors as the most significant, the next closest as less significant, and so on and so forth. The Gaussian function for two dimensions, x and y, is as follows:

$$N_\sigma(x, y) = \frac{1}{\sqrt{2\pi\sigma}} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \quad , \tag{3}$$

where σ is the standard deviation, and assuming mean is 0. Convolving the intensity and the Gaussian function, I and N_σ, respectively, (in order to use the Gaussian distribution), we arrive at the function:

$$(4) \qquad h(x) = (I * N_\sigma)(x,y) = \sum_{u=-i}^{+i} \sum_{v=-j}^{+j} I(u,v) N_\sigma(x-u, y-v) \quad ,$$

and this equation is for two dimensions. The variables i and j are the standard deviation. In this case, the standard deviation refers to the number of pixels that are used for smoothing, starting from the origin of the pixel and going out, in each direction, to the standard deviation. The convolution of the intensity and Gaussian equations weights each pixel within the area of the Gaussian distribution based on how far away the considered pixel is from the pixel we are trying to calculate the intensity for. Therefore, the double summation above averages all of the surrounding pixels by summing up each pixel's intensity multiplied by its Gaussian weight. One more important thing to consider is the orientation of the edge relative to the lens. Since we have two dimensions, the intensity can have a x and a y value. In this case, the direction of the gradient needs to be computed, and that can be done with these equations:

$$(5) \qquad \nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$$

$$(6) \qquad \frac{\nabla I}{\| \nabla I \|} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \quad ,$$

where θ is the orientation of the edge for a specific pixel at (x, y) (Russell & Norvig, 2010, pp. 937-939).

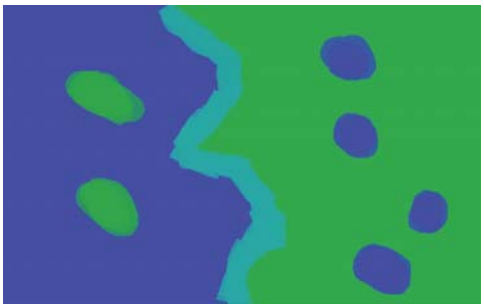
K-Means

One alternative to recognizing, or at least separating, different surfaces, that Professor Eaton had described, is called K-Means. In this application we have the grass, the road, and the sky. In order to separate the grass from the road, and the sky from the other two, we can assign what are called "centroids" to put each pixel into a group of pixels, or a segment. The pixels within these segments all share similar values, specifically location and color values. In a two-dimensional picture, the location information is simply x and y coordinates, and the color information is simply red, green, and blue values (although, it is worth noting, and probably preferable to use hue to detect one surface that may be partially covered by a shadow; red, green, and blue values do not account for these sometimes drastic differences). K-Means simply employs centroids to find the "closest" pixels with respect to location and color, tags each one of these pixels with its respective centroid, and changes its "location" based on the average of the pixels' "locations" ("location(s)" is in quotation marks because in this specific application, a location in an image is five-dimensional: x and y values, and red, green, and blue values, r, g, and b, respectively). In order to determine the location of a pixel relative to a centroid, we just use Euclidean distance:

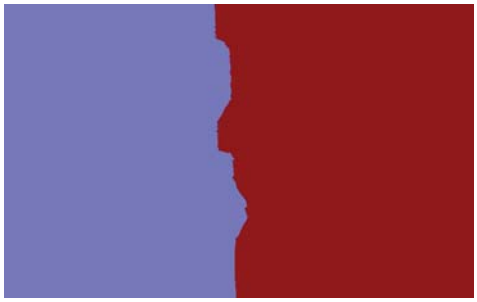
$$(7) \qquad Distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

As stated before, the centroid that is closest to a pixel using this formula is the centroid assigned to that pixel. After each pixel has been assigned a centroid, the centroid's five-dimensional information is updated to an average of its pixels' five dimensional locations, centering the centroid in its pixel "cloud", and the process repeats, with each pixel getting re-assigned a centroid. Slowly, the image becomes more and more segmented into similar areas in the image. The more iterations the segmentation algorithm goes through, usually, the more the image is smooth in color. Below is the result of applying K-Means to an image, using two centroids:

(a) Image to be analyzed



(b) Image after segmentation



It is important to note that, if the number of values for a color (from 0 to 255 in many color spaces) is much bigger than the x and y values may be, more preference will be given to matching that color because more of the Euclidean distance in the equation above will be dedicated to that color. Vice versa, if the x and y values are much bigger than the number of values for a color, the position of each pixel will have more weight than the color of the pixel, because more of the Euclidean distance in the equation above will be dedicated to the location of the pixel. As a result, each dimension should be normalized to be between two values, like [0, 1]. Also, randomization of the centroids' "locations" helps to reduce the occurrences of degenerate cases where two or more centroids have the same distance value to a pixel but their dimensional values are different, and it simply helps K-Means perform better.

Texture

Continuing with the basics of computer vision, we arrive at texture, which is the "visual feel of a surface". A good example of texture is wallpaper. Some wallpapers are plain, but most of them have some sort of pattern to them, like flowers or stripes. The texture is the repeating pattern of flowers or stripes on the wallpaper. These textured objects throw off edge detection because the varying design of textured surfaces can cause heavy fluctuation in the grayscale values. The key to correctly detecting texture in these situations is to find the orientation of the brightness of each pixel and comparing them via histograms (Russell & Norvig, 2010, pp. 939-940). Russell and Norvig state, using an analogy of spots on a tiger and spots in the grass, "A patch on a tiger and a patch on the grassy background will have very different orientation histograms, allowing us to find the boundary curve between them" (Russell & Norvig, 2010, p. 939). It would not be surprising if some of the computer vision techniques described in previous sections can be used for texture recognition as well.

Stereo Vision

Next is binocular stereopsis, where two cameras or other sorts of optical sensors are placed horizontally next to each other, facing in about the same direction. Let us assume we are using cameras. As a result, an item's location on one of the cameras' sensors is slightly different than on the other camera's sensor. This disparity can be measured and correlated to some sort of distance. Usually, this disparity is described by angle. For instance, if one object is further away from another one, where the first object is at a distance Z and the second object is δZ away from the first, typically the optical sensors have to be more outward facing in order to fixate on the further the object. As a result, there is an angular difference between the closer object and the further object. For the left and right images from the cameras, we have the angular disparity between the two objects, P_L and P_R, for the left camera and the right camera, respectively. Letting θ be the angle between the center line from the middle of the optical sensors to the closer object and the line from one of the sensors to the object, the angular displacement for both P_L and P_R is $\delta\theta/2$ with respect to the object further away, and the total displacement between the left and right views is $\delta\theta$ ($P_L + P_R = \frac{\delta\theta}{2} + \frac{\delta\theta}{2} = \delta\theta$).

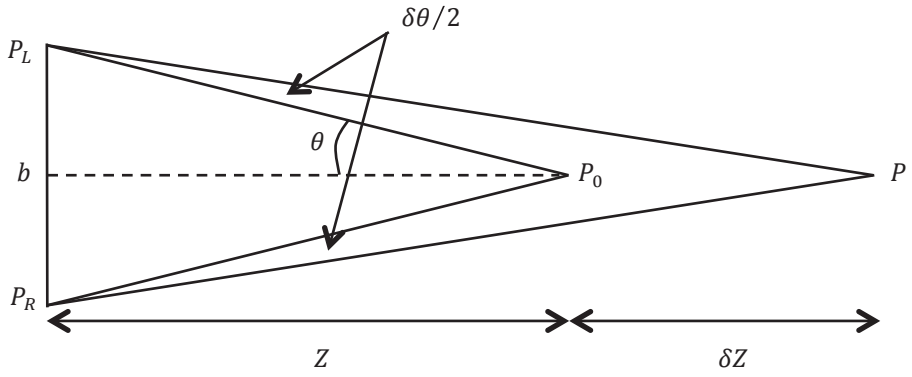


Figure adapted from Figure 24.17 in Russell & Norvig, pp. 950

Now, if we are given the distance between the cameras, b, that are fixated on the object P_0, and we want to find the relationship between depth and disparity then we have the equation

$$(8) \qquad \tan\theta = \frac{b/2}{Z}$$

and

$$(9) \qquad \tan(\theta - \delta\theta/2) = \frac{b/2}{Z + \delta Z} \ ,$$

because θ corresponds to the displacement with respect to the close point, and $\delta\theta/2$ corresponds to the displacement with respect to the further point, then we add δZ to the denominator (because we are computing the tangent). Given a small angle θ, $\tan\theta \approx \theta$, and as a result:

$$(10) \qquad \theta - \delta\theta/2 = \frac{b/2}{Z + \delta Z}$$

$$(11) \qquad \Rightarrow \delta\theta/2 = \frac{b/2}{Z} - \frac{b/2}{Z + \delta Z}$$

$$(12) \qquad \Rightarrow \delta\theta/2 = \frac{b/2}{Z} - \frac{b/2}{Z + \delta Z} \approx \frac{b\delta Z}{2Z^2} \ .$$

However, since the real disparity is $\delta\theta$, twice the equation above, then the angular disparity between P_0 and P is $\frac{b\delta Z}{Z^2}$ (Russell & Norvig, 2010, pp. 949-950).

Optical Flow

This is one way to compute the distance from the camera's perspective to the object itself, though not the only way. Another way is through optical flow, which is observing the way that pixels associated with an object shift across the frame over a certain amount of time. As a result, this method only requires one camera. Optical flow is generally depicted with vector arrows laid over an image, and each arrow represents a pixel patch's motion within the frame. Typically, vectors for optical flow are represented with two components, $v_x(x, y)$ and $v_y(x, y)$, where v_x

represents the velocity of a pixel patch in the x direction and v_y represents the velocity of the same pixel patch in the y direction. In order to find where the pixel patch, which is over (x, y), has moved, it makes sense to find pixel patches of similar intensity. The pixel patches are, at time $t + D_t$, at $(x + D_x, y + D_y)$. In order to see if a pixel patch is similar, it is possible to use what is called the sum of squared distances formula, which is as follows:

$$(13) \qquad SSD(D_x, D_y) = \sum_{(x,y)} [I(x, y, t) - I(x + D_x, y + D_y, t + D_t)]^2 \quad ,$$

where I is the intensity function for a pixel. The (D_x, D_y) that minimizes the sum of squared differences function is the most likely displacement from the pixel patch of the successor of the pixel patch, because it is closest to the intensity of the pixel patch we are trying to track. Clearly, however, this does not work for surfaces that are uniform in color and texture, as the differences from pixel to pixel are all the same that it is hard to minimize the sum of squared differences function (Russell & Norvig, 2010, pp. 939-940).

If we know what pixel patch is associated with the previous pixel patch, we can calculate how fast the pixels in the image are moving over time, assuming the pixel is "stationary" within the three-dimensional environment (oncoming cars, for instance, will seem to be optically moving faster than other stationary objects in their nearby surroundings because they are heading towards the camera). Because closer objects move faster in a frame than objects further away, with the vectors determined with optical flow, we can determine how far "away" a pixel is in an image. Furthermore, smooth surfaces tend to have a continuous change in vector, but a patch of pixels that change vectors abruptly tend to be an object in the area in front of the camera. This allows for distance calculation as well as opens up a way to avoid obstacles in a path (Russell & Norvig, 2010, pp. 940-941).

So now that it is clear how to calculate distances, and, knowing the angle, it is possible to translate an image at an angle to a bird's-eye view of the road, how can one pick the best possible route around obstacles, specifically around turns? Typically, observing cars programmed to drive themselves yields the idea that cars should hug the turn and stay in the middle of the lane, going the speed limit. But what happens if the car wants to get from point A to point B around a turn as fast as possible? Staying a certain distance away from the curb will not allow for maximum speed as the car is not minimizing the angle enough, and as a result, the tires will slip. However, if one were to drive a car Formula 1 style, swinging out on the road, and cutting into the turn to just hit the apex of the turn, and swinging out wide again for the exit to maintain speed, the benefit of maintaining the speed would most likely outweigh the benefit of reducing the distance around the turn by driving more slowly.

Shortest Path

A common concept in all of artificial intelligence, but rather straightforward, is shortest path planning. Because calculating distance is so crucial for my application, it makes sense to discuss algorithms that handle shortest path planning and distance calculation based on a binary image. Distance transformation is often used to calculate the minimum distance an object is from the background based on a binary image. In order to define a distance function, three criteria need to be met:

1. Positive definite: $d(p, q) \geq 0$, if and only if $p = q$, for all $p, q \in S$
2. Triangular: $d(p, r) \leq d(p, q) + d(q, r)$, for all $p, q, r \in S$
3. Symmetric: $d(p, q) = d(q, p)$, for all $p, q \in S$

where S is in a binary image and d is a function that maps S to a non-negative integer matrix. In creating this function d, three kinds of algorithms are used: Manhattan distance, chessboard, and Euclidean. The Manhattan distance between two points is:

$$(14) \qquad d_4(P, Q) = |x - u| + |y - v|$$

where $P = (x, y)$ and $Q = (u, v)$. Chessboard distance is very similar to Manhattan distance:

$$(15) \qquad d_8(P, Q) = \max(|x - u|, |y - v|)$$

and Euclidean distance is as follows:

$$(16) \qquad d_e(P,Q) = \sqrt{(x-u)^2 + (y-v)^2}$$

In order to perform three-dimensional distance calculations, one simply modifies these formulae to include one more dimension (Shih, 2010, pp. 179-183).

4. DIFFERENCES IN TRADITIONAL SELF-DRIVING CARS AND THIS SPECIFIC APPLICATION

So, as can be seen, traditional self-driving cars utilize a host of technologies, which include cameras, but are not completely dependent on cameras. In this specific application, we are going to use cameras, as, in most cases, the boundaries of the road are not detectable via a system of lasers or other equipment. Rangefinding hardware is usually good with determining distance to an object that has some vertical substance, such as pedestrians or curbs. However, on race courses, these vertical barriers do not exist. Instead, they are replaced with alternating red and white flat curbs or just standard colored lines. However, other than the type of technology used to gather data, many of the different techniques in utilizing this data remain the same. Depth information is still used, and objects are recognized, allowing an optimal path to be taken based on the car's surrounding environment. As a result, the software structure, such as the interface to the car and general algorithms remain the same. The point is to realize that even though different sensors may be used, the same depth and texture information is accumulated, and, as a result, the same algorithms still apply.

Also, it is important to note that, while many of the computer vision techniques described in the previous sections are obviously legitimate ways to gather information about a car's environment, many of them do not have a place in this thesis' specific application, such as texture recognition or optical flow, as they are computationally expensive and gives the car more information than it actually needs.

5. CALCULATING A FAST PATH

So now we have the tools necessary to start heading towards the problem, which is finding a way to drive quickly through some sort of course. In order to do that, we need to lay out the basics of driving. The car has throttle, the brakes, and steering. If a car goes around a turn too fast, it either has a tendency to understeer or oversteer, (where understeer is when the car's tires push away from the turn, as they lose grip and start to slide, and where oversteer is when the back tires slip out and the car goes into a "powerslide"). The goal is to keep the car neutral around the turn, minimizing understeer and oversteer (Klein, 2003). The options available to us to calculate a proper course that manages under- or oversteer include a range-finding technique through radar and other sorts of range finding technologies and visual techniques including mono and stereo vision. For this paper's purpose, visual systems are the focus in solving this problem. In order to navigate a certain terrain, depth is essential for painting a virtual environment for the car to analyze. If driving on the road, in most cases, the car is parallel to the road in front of it, so analyzing an image from a single camera for depth is not very hard to do, and may be worth the computational and space saving benefits.

Car Properties

It is clear at this point that it is possible to define an equation, specific to a car, relating the speed of the car and the angle at which to turn in before the car begins to spin due to lack of grip. The equation is part downforce, which is how much force the car places on the ground (directly downward) at a certain speed, and part centripetal force, the force competing against downforce for grip. Centripetal force is dependent on the speed and direction of turn-in of the car, which is described in the formula

$$(17) \qquad F_c = m\left(\frac{v^2}{r}\right) \ ,$$

where m is the mass of the car (sprung and unsprung), v is the velocity of the car, and r is the radius of the car's turning circle (Wolfson & Pasachoff, 1999, p. 81). Ideally, we want $F_c = F_d$, where F_d is the downforce, so that the car does not slip. In a typical situation, the car does not change mass, and the velocity aims to be the same. As a

result, the radius needs to change depending on how fast the car is moving, or the car needs to slow down in order to maintain grip around the turn radius. In order to pick the best entry into a turn, the proper radius needs to be calculated, and from that, have the center point, called the turning circle center, of the turn be perpendicular to the line followed through the turn at the apex, about r distance away. The proper place to enter the turn is r distance away from the turning circle center towards the beginning of the turn, where a spot on the circumference of the circle has a tangent equivalent to the line that the car is currently driving on. Since this line is tangent to the turning circle, the car may need to be put on a driving line that is tangent to the circle, but that is not the main focus of this subject. Furthermore, if this radius is outside of the road, it is necessary for the car to slow down enough such that the radius is within the bounds of the road. The exit process for the car will be the same as the entry process, just at the end of the turn. Using the methods of calculation for the turn-in, it is possible to develop some sort of function that maps a certain speed with a certain maximum turning angle. Turn too much, and the driver risks understeer or oversteer; turn in too little, and the driver is not maximizing the speed permitted by the centripetal force and the force of the grip of the tires.

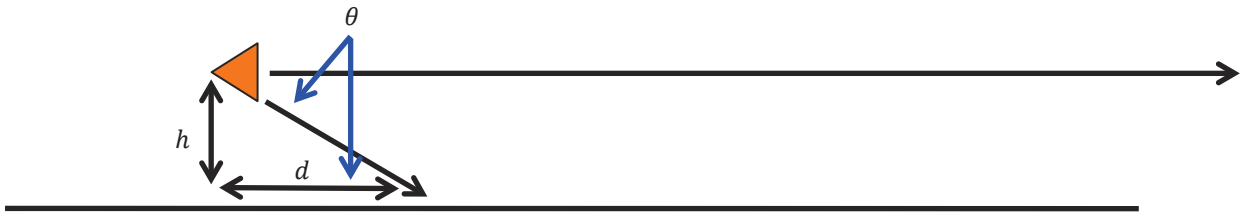
Critical Points

As stated previously, instead of hugging the side of the road, fastest way around the turn is to start out wide, and "clip" the apex of the turn, swinging out wide again while exiting the turn. It is usually the best to swing out to the very edge of the road for turn entry and exit, so those entry and exit points are not too difficult to determine. However, determining the apex of the turn is rather complex. Ideally, the apex should be on the point of the curve of the turn that is changing the "fastest", or the maximum magnitude of the derivative along the curve. This requires some calculation of the curve's respective formula, or linear approximation equation, which is correct, but overly complex. If we were to simply draw a line from the entry point to the exit point, the line drawn will essentially cut the turn. If we then shift this line up by a constant, the last point of the turn that was contained in that line is the apex, or close enough to be sufficient for the apex. Because the image inherently compresses the road segments that are further away, performing this method would not be as accurate as performing it on a map of the turn. A more accurate surface to perform this analysis on would be an inverse perspective projection of the image into a three dimensional environment (where the surface would be a two-dimensional ground) (Grimm & Goldman). Using these three points, we can perform a linear approximation for them to derive a smooth curve for the car to follow around the turn.

Approximate Inverse Perspective Projection

All three-dimensional information about an environment is lost when a picture is taken of the environment. This is because the light goes through the lens, and is bent while in the lens so that each pixel on the flat image sensor receives light. This is perspective projection, where the three-dimensional perspective of the camera is projected onto two dimensions. Therefore, when using a single camera to gather data about the surrounding environment, specifically to build a map of the environment, an inverse perspective projection is needed to change the two-dimensional camera projection back to our best guess for the environment's three-dimensional layout. For this specific application, we luckily have some assumptions that we can make. The first assumption is that we know how far the camera is from the ground (although, we will see later on that this does not make a difference in terms of building a map). We also know that the surface the car is driving on is completely flat, which allows us to safely calculate the angle of the camera relative to the ground. Using pure trigonometry, we can recover spatial data knowing the two-dimensional data and these assumptions. Note, however, that this may not complete inverse perspective projection, as Professor Eaton suspects, but it is a trigonometric simulation of it (and therefore, may not as accurate because of different camera types). This notion makes sense, as it seems that perspective projection may require complex matrix calculations (Grimm & Goldman).

Consider the following figure:



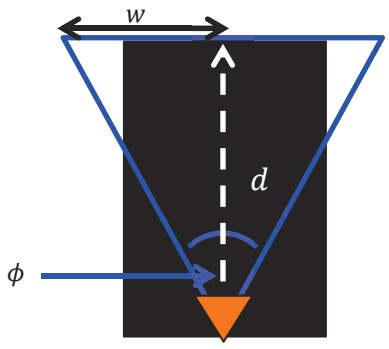
We have the height, h, from the ground to the camera, and we want to find the distance, d, from the point where the camera is over the ground to the portion of land that is represented by a pixel. Knowing the camera's field-of-view, or the angle at which the camera takes in light (for instance, a fisheye lens has a wide field of view, nearly 180 degrees, because it takes in more of its surroundings), θ, and, assuming that the camera is parallel to the ground, the angle between the camera's lowest angle and the line drawn straight ahead from the camera's lens is $\theta/2$ because half of the camera's view is above the light drawn straight ahead, and half is below. Since we know this angle, and the line drawn from the camera is parallel to the ground, simple geometry says that the angle between the camera's lowest angle and the ground is also $\theta/2$. With this angle, we have this relationship:

$$(18) \qquad tan\left(\frac{\theta}{2}\right) = \frac{h}{d} \ ,$$

and solving for d, we arrive at this equation:

$$(19) \qquad d = \frac{h}{tan\left(\frac{\theta}{2}\right)} \ .$$

So with this equation, we are able to calculate the distance to a point in an image from the point of the ground that the camera is over. However, since the camera is a perspective projection, we also need to compensate for the shrinking effect that perspective has on an image. For instance, if there were a road stretching from the camera's position to the horizon, the road would appear to shrink as it approached the horizon. In order to figure out the width of the road, we just need to use more trigonometry. Like the camera's vertical field of view, as described above, the camera also has horizontal field-of-view:



Letting this field-of-view be ϕ, we know that the angle between the midline of the horizontal field of view and an edge of the field of view is $\phi/2$. From the previous calculation, we also know the distance between the camera and the point we are looking at, d. We want to find the distance between the point that we are considering and the midline, so call this distance w. Therefore, we are left with the equation

$$(20) \qquad tan\left(\frac{\phi}{2}\right) = \frac{w}{d}$$

and solving for w, we are left with

$$(21) \qquad w = d\ tan\left(\frac{\phi}{2}\right) \ .$$

With this width information and depth information, it is possible to construct a map which represents an orthogonal bird's-eye view of the track the car is on, scaling the parameter for the tangent functions in both equations for each pixel that is being considered. With this map, the next step is to pick out critical points in the turn.

Path Based on Linear Approximation

In order to calculate the path to go around a turn, one simply needs three points: the entry point, the apex, and the exit point. The apex and the road bounds of the beginning of the turn in and the exit point are all determined with the computer vision algorithms described above. So, now that the three crucial points have been obtained, it is necessary to draw a line between them that best fits the turn around the curve. Clearly, a line of best fit is required, and to create such a line, a Lagrange polynomial interpolation is one of the best ways to do so, as it provides a smooth, consistent-derivative line so that the maximum centripetal force around the turn is minimized. If we are given a set of points, $(x_1, y_1)\ (x_2, y_2)\ (x_3, y_3)$, then, using the Lagrange interpolation, a polynomial that best fits this data is given by

$$(22) \qquad P(x) = \sum_{i=1}^{n} P_j(x) \ ,$$

where $P_j(x)$ is

$$(23) \qquad P_j(x) = y_j \prod_{\substack{k=1 \\ k \neq j}}^{n} \frac{x - x_k}{x_j - x_k} \ .$$

So when $P(x)$ is expanded out explicitly, the result is

$$(24) \qquad P(x) = \frac{(x - x_2)(x - x_3) \dots (x - x_n)}{(x_1 - x_2)(x_1 - x_3) \dots (x_1 - x_n)} y_1 + \frac{(x - x_1)(x - x_3) \dots (x - x_n)}{(x_2 - x_1)(x_2 - x_3) \dots (x_2 - x_n)} y_2 + \cdots + \frac{(x - x_1)(x - x_2) \dots (x - x_{n-1})}{(x_n - x_1)(x_n - x_2) \dots (x_n - x_{n-1})} y_n \ ,$$

where n is the number of points to make a line through. This polynomial is an equation to draw a line through the points around a curve so that the car may follow (Archer & Weisstein, 2012). Ideally, this curve will be "painted" onto the surface that the car is driving on so that the car may follow it with its camera. In summation, this technique can provide a nice line for a car to follow.

6. IMPLEMENTATION

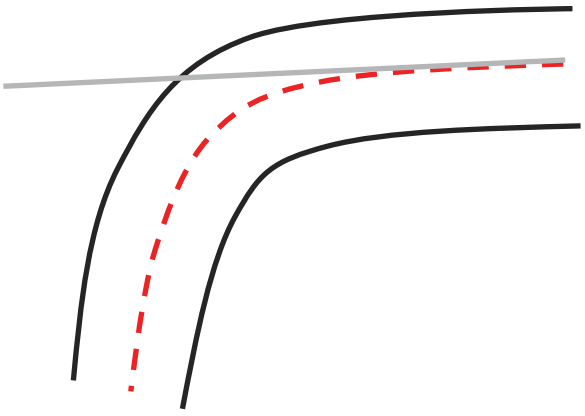
From the start of the thesis, I had my eyes set on implementing a self-driving race car within some sort of physics simulator or other software that can simulate enough physics to make this possible, and Epic Games' Unreal Development Kit (UDK) is well documented and has a vehicle class that simulates vehicle physics. Furthermore, an add-on called USARSim, which is built specifically for robotics simulation, supports UDK, and has a featured called the Image Server, which, through DirectX APIs, is able to retrieve the image buffer generated from the Unreal Engine. This means that accessing screenshots through dynamically linked libraries tied into UDK's Unrealscript code allows a screenshot to go straight from DirectX to the car-driving code, which saves file IO operations as well

as makes it easy to retrieve images. I was able to create a track, and download a car free for use, which simulates physics very well, but makes it a lot simpler, by limiting the possible steering angle of the car based on how fast the car is going. Since we know how limited our steering angle is, it is possible to pick an entry point that allows for a limited turning radius while maintaining speed, or some sort of compromise of the two. The car needed to be calibrated slightly so that engine braking, when the engine naturally slows down the car, needed to be reduced, the top speed increased, the acceleration decreased, and the turning angle curve (over various speeds) expanded so that the wheels turned more smoothly over a range of speed.

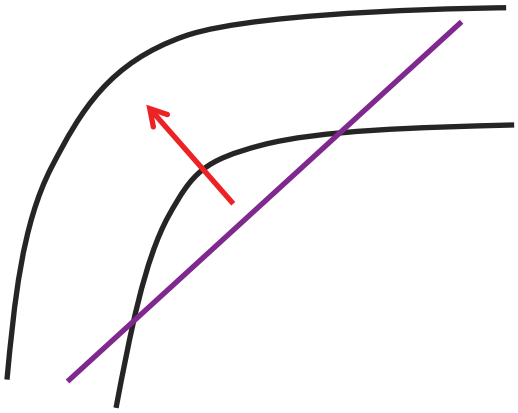
Unfortunately, due to time constraints, I only was able to implement a very rough version of path plotting for turns in the track. Screenshots were taken from the track in UDK at certain turns that weren't any more than 90 degrees, and put through segmentation and turn analysis. The image was first segmented using K-Means, which produced very good results when the centroid's locations we randomized. Also, I could not get the normalization code for each dimension to work properly, so I instead increased the weight of the color values by multiplying them by a constant to simulate normalization. Furthermore, the sky and the ground that were far away in the image were turned blue to avoid noise problems in the path finding. Then, I used the path finding method described in section 4 to determine where the critical points were. I simplified the problem so that the starting point of the turn is right in front of the car, and, as a result, the entry point is at the bottom of the image, halfway across. The exit point was determined by finding the outermost edge of the road near the horizon, which depended on which way the road was turning.

In order to determine which way the road was turning, I projected back a line from the top of the center of the road back, and if the average location of the center of the road were to the left of this line, the road was turning left. Otherwise, it was turning right. With this information, the side of the road that contains the apex is known, so the algorithm knows where to find the apex. Using a line that goes through the entry and exit point, I decremented the constant (because the pixels are indexed from the top of the picture, down), which moved the line up (typically diagonally) until the last point of the turn touches the line, which is the apex. Therefore, the apex is quickly found.

a. Finding the direction of the turn



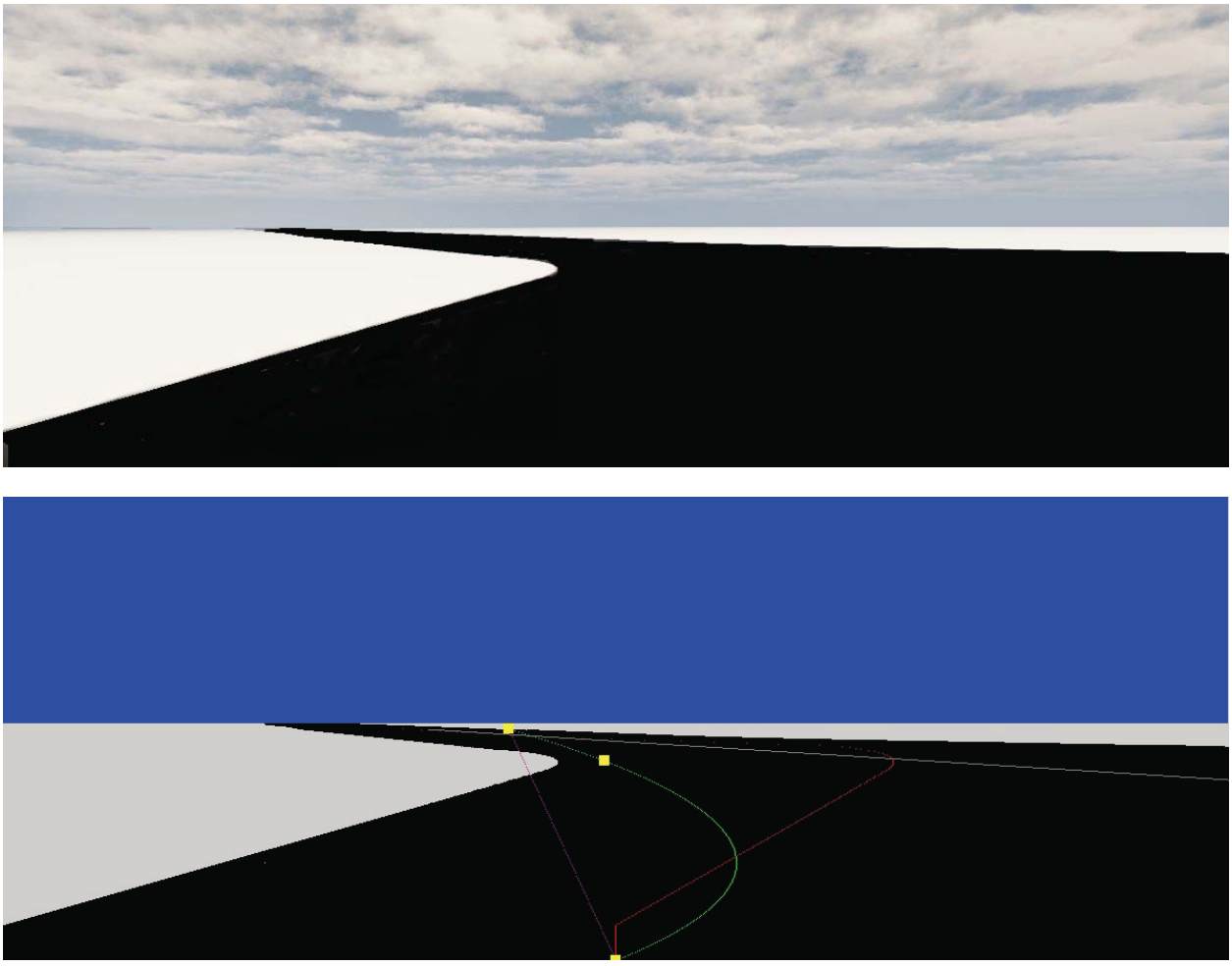
b. Finding the apex



From that point, a simple Lagrange interpolation is used to connect the entry, apex, and exit points, and that interpolation line is colored onto the image. Because the turn goes "up" the image, the y values of the critical points would cross often, and, as a result, the interpolation between the critical points would not be a function (because of the two possible y values for one x value, so the interpolation would give a bad line. In the code, I switched the x and y values of the pixels to be colored on the line so that the interpolation plotted a better line. With further implementation into UDK, the car will try to keep the interpolated line in the center of its camera, so it effectively follows the interpolated line. Furthermore, the car will be performing this image processing, ideally, at each frame, so it is updating its path very frequently.

7. RESULTS

Here is the result of the turn analysis, with the source image on the top, and the segmented and analyzed image below:



In the segmented image, the road is colored black, the grass is colored gray, and the sky is colored blue. The yellow dots are the entry, apex, and exit points, and the green line represents the Lagrange interpolation between those points. The red line is the center of the road, the gray line is the projection used to determine whether the road is turning left or right, and the purple line is the line that is drawn between the entry and exit point and is pushed, in this case, to the right, until it hits the apex. The dot that denotes the apex is pushed out further away from the side of the road, because the center of the car is aiming for the apex, while the inside tire of the car is closer to the side of the road, so it is necessary to make room on the road for that tire. We can measure success based off of visual evidence, as shown in these pictures. In order to measure success in a race environment, it would be best to record the time the car takes to go around the track non-race style and race style, and compare these times to see if this thesis' method works well. There are a number of assumptions made, and many variables unaccounted for if put into practice. These were explained previously, and some will be elaborated on in the next section.

8. IMPROVEMENTS

Given more time, I would have liked to fully implement the car in UDK so that it drives by itself around the track, but I did not have enough time to pursue that goal. Furthermore, improvements the algorithm for path finding also exist. One, stated previously, is that my implementation does not create a bird's eye view of the road for

accurate path mapping, so it just uses the image. This may not be as accurate because of the distortion in the lens and the fact that perspective makes everything shrink as it goes into the horizon, However, the path is updated at frequent, regular intervals, and the mapping is more accurate for closer segments of the road, so this type of analysis may be good enough.

Also, given that noise can be so prevalent, I would like to find a more reliable way of segmenting images, because many of the screenshots I took were subject to glare that messed with the segmentation. I believe that using hue instead of red, green, and blue values, as stated previously, may solve this problem. Furthermore, because these images are subject to noise, it would be interesting to see if implementing a Markov chain with respect to path history would smooth out the car's actions and potentially make it more reliable.

Another concern is a racetrack with varying elevation. Because my application of self-driving cars assumes that the racetrack is uniform in elevation, other factors crucial to the calculation, such as the horizon, are also assumed. These factors, however, change when the plane of the track is not parallel to the direction of the car. I do believe, however, that with the Markov chain described above, plus a little bit more programming to determine where the horizon is, that my method for path calculation will remain almost as accurate as when under these assumptions. Another important assumption that I programmed this path finding with that is that the car will not be dealing with turns that are greater than 90 degrees. However, it would be interesting to see how the car handles turns greater than 90 degrees without modifying the code. I believe it will still perform very well, because the car is updating its path at frequent, regular intervals.

Finally, it is important to note that my implementation does not recognize other cars on the track, so these cars are only good for time-trialing. Optical flow and texture recognition may be necessary for the car to recognize and avoid other cars on the track. Using these techniques to identify cars, the car would give priority to avoiding the cars over following the optimal path, or possibly use a larger, with respect to points, linear interpolation to find a path around the car but closest to the path taken if the car were not there.

9. CONCLUSION

As can be seen, the idea of autonomous race cars is a very interesting one, and poses quite a different challenge than those tackled by current self-driving cars, that only have the goal of getting from point A to point B safely. Unlike traditional self-driving cars, autonomous racecars need to pick the path that optimizes their speed without losing traction and spinning out or running off the track. It is possible to do this with a single camera and some assumptions about the car's environment. There are many technologies used in self-driving cars, such as RADAR, LIDAR, and other range finding technologies, but that hardware is useful for terrain that varies in height, which a track with painted lines might not be. Therefore, a camera seems to make more sense, using color values for feature recognition. While image analysis techniques, like optical flow, texture recognition, and stereo vision are useful in current applications of self driving cars, most of that is not necessary for self-driving race cars. Image segmentation using K-Means proves to be very useful to pick out important features. Clever thinking makes it easy to determine which way the road bends, and, using this knowledge, finding the apex is made much easier. Linear approximation gives the car a smooth path to follow so that speed can be maximized around a turn. Some implementation of these techniques show that they can be successful, and more careful programming will make these methods even more accurate. I believe that this is something worth pursuing further, as fast driving is something seemingly under-researched but very critical to driving.

References

Archer, B., & Weisstein, E. W. (2012). Lagrange interpolating polynomial. *Wolfram Mathworld*. Retrieved February 20, 2012, from http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html

Bailey, S. (2011). BMW track trainer - special report: How a car can teach you to drive. *Road &* Track. Retrieved April 15, 2012, from http://www.roadandtrack.com/auto-news/tech/bmw-track-trainer

Ballard, D. H., & Brown, C. M. (1982). *Computer vision.* Englewood Cliffs, New Jersey: Prentice Hall. Retrieved 2012, from http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/Ballard__D._and_Brown__C._M.__1982__Computer_Vision.pdf

Klein, B. (Director). (2003). *Top gear: Series 2 episode 5* [Television series episode]. BBC.

Grimm, C., & Goldman, K. (n.d.). *Perspective projection*. Retrieved April 20, 2012, from Washington University in St. Louis, Computer Science website: http://www.cs.wustl.edu/~kjg/cse131/modules/arrays/PerspectiveProjection.pdf

Microsoft Visual C++ 2010 Express [Computer software]. (2010). Redmond, WA: Microsoft.

Montemerlo, M., Becker, J., Bhat, S., Dahlkamp, H., Dolgov, D., Ettinger, S., et al. (2008). *Junior: The Stanford entry in the Urban Challenge.* Retrieved 2012, from Sebastian Thrun's website: http://robots.stanford.edu/papers/junior08.html

Russell, S., & Norvig, P. (2010). *Artificial intelligence: A modern approach* (3rd ed.). Upper Saddle River, New Jersey: Prentice Hall.

Shih, F. Y. (2010). *Image processing and pattern recognition.* Hoboken, New Jersey: John Wiley & Sons.

Thrun, S. (2010). Toward robotic cars. *Communications of the ACM*, 99-106. doi:10.1145/1721654.1721679

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., et al. (2006). Stanley: The robot that won the DARPA Grand Challenge. *Journal of Field Robotics*, 661-692. Retreived 2012, from http://robots.stanford.edu/papers/thrun.stanley05.html

Unreal Development Kit [Computer software]. (2011). Cary, NC: Epic Games.

USARSim [Computer software]. (2011).

Wolfson, R., & Pasachoff, J. M. (1999). *Physics for scientists and engineers* (3rd ed.). Reading, Massachusettes: Addison-Wesley.

Acknowledgements:

I would like to acknowledge Professor Eaton, for advising and suggesting different techniques, such as the apex finding technique, K-Means segmentation, and discussing perspective projection; Professor Wonnacott, for advising and suggesting different techniques, such as hue in segmentation, and correcting me on perspective projection; Professor Dougherty, Ian Burnette, and Takumi McAllister for reading over my thesis and giving helpful input; Professor Lindell, Lucas Van Meter, Eric Arnold, Pat Haneman, Faraz Sohail, Aaron Buikema, Andrew Sturner, and Ben Siqueiros, and Chelsea Thorsheim for letting me bounce ideas off of them to make sure that these ideas are sound; Microsoft, for their online help pages for Microsoft Visual C++ 2010 Express and Microsoft Visual C++, the language in which my code for this thesis is written; Epic Games, developers of the Unreal Development Kit (UDK), which I used to create a track, as well as the person, who I do not remember, who drew a layout of the Top Gear test track which I outlined and imported that outline into UDK to create a test track, and the person who developed the car that I modified for UDK; and the developers of USARSim.

APPENDIX (CODE)

```
// segmentation.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include "math.h"
#include <time.h>

#using <mscorlib.dll>
#using <presentationcore.dll>
#using <WindowsBase.dll>
#using <System.dll>


using namespace std;
using namespace System;
using namespace System::Windows::Media;
using namespace System::Windows::Media::Imaging;
using namespace System::IO;
using namespace System::Collections::Generic;



ref struct pixel {
	unsigned char red;
	unsigned char green;
	unsigned char blue;
	int centroid;
} ;

ref struct centroid {
	int x;
	int y;
	int red;
	int green;
	int blue;
	int xsum;
	int ysum;
	int redsum;
	int greensum;
	int bluesum;
	int owned;
	int index;
} ;

bool segmentation();
array<pixel^, 2>^ edgeDetection(array<pixel^, 2>^ pixelMatrix, int imageHeight, int imageWidth);
array<pixel^, 2>^ inversePerspective(array<pixel^, 2>^ pixelMatrix, int imageHeight, int
imageWidth);
array<pixel^, 2>^ criticalPoints(array<pixel^, 2>^ pixelMatrix, int imageHeight, int imageWidth);

int _tmain(int argc, _TCHAR* argv[])
{
	segmentation();
	system("pause");
	return 0;
}



bool segmentation() {
	//DECODING SOURCE JPEG
	Stream^ imageStream = gcnew FileStream("C:\\UDK\\UDKCapture2.jpg", FileMode::Open,
FileAccess::Read, FileShare::Read);
	JpegBitmapDecoder^ decoder = gcnew JpegBitmapDecoder(imageStream,
BitmapCreateOptions::PreservePixelFormat, BitmapCacheOption::Default);
	BitmapSource^ bitmapSource = decoder->Frames[0];

	PixelFormat format = bitmapSource->Format;
```

```
        int imageWidth = bitmapSource->PixelWidth;
        int imageHeight = bitmapSource->PixelHeight;
        int pixelSize = format.BitsPerPixel;
        array<unsigned char>^ pixels = gcnew array<unsigned
char>(imageHeight*imageWidth*pixelSize);
        bitmapSource->CopyPixels(pixels, imageWidth*pixelSize, 0);









        array<pixel^, 2>^ pixelMatrix = gcnew array<pixel^, 2>(imageHeight, imageWidth);

        //PUTTING PIXELS INTO EASY-TO-USE MATRIX
        for (int i = 0; i < imageHeight; i++) {
                int x = 0;
                for (int j = 0; j < (imageWidth*(pixelSize/8)); j = j+3) {
                        pixelMatrix[i, x] = gcnew pixel;
                        pixelMatrix[i, x]->blue = (int)pixels[(i*imageWidth*pixelSize)+j];
                        pixelMatrix[i, x]->green = (int)pixels[((i*imageWidth*pixelSize)+j)+1];
                        pixelMatrix[i, x]->red = (int)pixels[((i*imageWidth*pixelSize)+j)+2];
                        x++;
                }
        }



        cout << "Performing segmentation...";
        int centroidNumber = 2;
        array<centroid^>^ centroids = gcnew array<centroid^>(centroidNumber);
        srand((unsigned)time(NULL));
        for (int i = 0; i < centroidNumber; i++) {
                centroids[i] = gcnew centroid;
                centroids[i]->x = rand() % (imageWidth);
                cout << centroids[i]->x << ", ";
                centroids[i]->y = rand() % (imageHeight);
                cout << centroids[i]->y << endl;
                centroids[i]->red = rand() % 255;
                centroids[i]->green = rand() % 255;
                centroids[i]->blue = rand() % 255;
                centroids[i]->xsum = 0;
                centroids[i]->ysum = 0;
                centroids[i]->redsum = 0;
                centroids[i]->greensum = 0;
                centroids[i]->bluesum = 0;
                centroids[i]->owned = 1;
                centroids[i]->index = i;
        }


        //SEGMENTATION ALGORITHM
        int iterations = 20;
        for (int t = 0; t < iterations; t++) {
                float colorWeight = 2.5;
                for (int i = 0; i < imageHeight; i++) {
                        for (int j = 0; j < imageWidth; j++) {
                                int closestCentroid = 0;
                                float minimumDistance = 10000000.0;
                                for (int c = 0; c < centroidNumber; c++) {
                                        float tempDistance = sqrt(pow((float)(centroids[c]->x -
j),2) + pow((float)(centroids[c]->y - i),2) + pow((float)(colorWeight*(centroids[c]->red -
pixelMatrix[i, j]->red)),2) +  pow((float)(colorWeight*(centroids[c]->green - pixelMatrix[i, j]-
>green)),2) + pow((float)(colorWeight*(centroids[c]->blue - pixelMatrix[i, j]->blue)),2));
                                        if (tempDistance < minimumDistance) {
                                                closestCentroid = c;
                                                minimumDistance = tempDistance;
```

```
                                }
                        }

                        pixelMatrix[i, j]->centroid = closestCentroid;
                        centroids[closestCentroid]->xsum += j;
                        centroids[closestCentroid]->ysum += i;
                        centroids[closestCentroid]->redsum += pixelMatrix[i, j]->red;
                        centroids[closestCentroid]->greensum += pixelMatrix[i, j]->green;
                        centroids[closestCentroid]->bluesum += pixelMatrix[i, j]->blue;
                        centroids[closestCentroid]->owned += 1;
                }
                if (i % 300 == 0) cout << ".";
        }


        for (int c = 0; c < centroidNumber; c++) {
                centroids[c]->x = (int)(centroids[c]->xsum/centroids[c]->owned);
                centroids[c]->y = (int)(centroids[c]->ysum/centroids[c]->owned);
                centroids[c]->red = (int)(centroids[c]->redsum/centroids[c]->owned);
                centroids[c]->green = (int)(centroids[c]->greensum/centroids[c]->owned);
                centroids[c]->blue = (int)(centroids[c]->bluesum/centroids[c]->owned);

                centroids[c]->xsum = 0;
                centroids[c]->ysum = 0;
                centroids[c]->redsum = 0;
                centroids[c]->greensum = 0;
                centroids[c]->bluesum = 0;
                centroids[c]->owned = 1;
        }
    }

    cout << endl;
    //ASSIGNING COLORS TO PIXELS FOR SEGMENTATION SHADING
    for (int i = 0; i < imageHeight; i++) {
        for (int j = 0; j < imageWidth; j++) {
            pixelMatrix[i, j]->red = centroids[pixelMatrix[i, j]->centroid]->red;
            pixelMatrix[i, j]->green = centroids[pixelMatrix[i, j]->centroid]->green;
            pixelMatrix[i, j]->blue = centroids[pixelMatrix[i, j]->centroid]->blue;
        }
    }



    //FOR DEBUGGING PURPOSES
    for (int i = 0; i < imageHeight; i=i+15) {
        for (int j = 0; j < imageWidth; j=j+20) {
            cout << pixelMatrix[i, j]->centroid;
        }
        cout << endl;
    }

    pixelMatrix = edgeDetection(pixelMatrix, imageHeight, imageWidth);
    //inversePerspective(pixelMatrix, imageHeight, imageWidth);
    pixelMatrix = criticalPoints(pixelMatrix, imageHeight, imageWidth);

    //CREATING 1D ARRAY FOR JPEG CONVERSION
    array<Byte>^ pixelSave = gcnew array<Byte>((imageWidth*(pixelSize)*imageHeight));
    for (int i = 0; i < imageHeight; i++) {
        int x = 0;
        for (int j = 0; j < (imageWidth*(pixelSize/8)); j = j+3) {
            pixelSave[(i*imageWidth*pixelSize)+j] = pixelMatrix[i, x]->blue;
            pixelSave[((i*imageWidth*pixelSize)+j)+1] = pixelMatrix[i, x]->green;
            pixelSave[((i*imageWidth*pixelSize)+j)+2] = pixelMatrix[i, x]->red;
            x++;
        }
    }


    List<System::Windows::Media::Color>^ colors = gcnew
List<System::Windows::Media::Color>();
    colors->Add(System::Windows::Media::Colors::Blue);
    colors->Add(System::Windows::Media::Colors::Green);
    colors->Add(System::Windows::Media::Colors::Red);
```

```
	BitmapPalette^ imagePalette = gcnew BitmapPalette(colors);
	BitmapSource^ imageSource = BitmapSource::Create(imageWidth, imageHeight, 96, 96,
PixelFormats::Bgr24, imagePalette, pixelSave, imageWidth*(pixelSize));

	System::IO::FileStream^ imageSaveStream = gcnew
System::IO::FileStream("C:\\UDK\\resultImage.jpg", FileMode::Create);
	JpegBitmapEncoder^ encoder = gcnew JpegBitmapEncoder();
	encoder->FlipHorizontal = false;
	encoder->FlipVertical = false;
	encoder->QualityLevel = 100;
	encoder->Rotation = Rotation::Rotate0;
	encoder->Frames->Add(BitmapFrame::Create(imageSource));
	encoder->Save(imageSaveStream);

	cout << "\nCompleted, Exiting..." << endl;

	return true;
}



array<pixel^, 2>^ edgeDetection(array<pixel^, 2>^ pixelMatrix, int imageHeight, int imageWidth) {
	for (int i = 1; i < imageHeight; i++) {
		for (int j = 1; j < imageWidth; j++) {
			if (i <= 220) {
				pixelMatrix[i-1, j-1]->red = 0;
				pixelMatrix[i-1, j-1]->green = 0;
				pixelMatrix[i-1, j-1]->blue = 254;
			} //else if ((pixelMatrix[i, j]->red) != (pixelMatrix[i-1, j-1]->red) ||
(pixelMatrix[i, j]->green) != (pixelMatrix[i-1, j-1]->green) || (pixelMatrix[i, j]->blue) !=
(pixelMatrix[i-1, j-1]->blue)) {
				//pixelMatrix[i-1, j-1]->red = 254;
				//pixelMatrix[i-1, j-1]->green = 254;
				//pixelMatrix[i-1, j-1]->blue = 254;
			//} else if ((pixelMatrix[i, j]->red) != (pixelMatrix[i, j-1]->red) ||
(pixelMatrix[i, j]->green) != (pixelMatrix[i-1, j]->green) || (pixelMatrix[i, j]->blue) !=
(pixelMatrix[i-1, j]->blue)) {
				//pixelMatrix[i-1, j]->red = 254;
				//pixelMatrix[i-1, j]->green = 254;
				//pixelMatrix[i-1, j]->blue = 254;
			//} else if ((pixelMatrix[i, j]->red) != (pixelMatrix[i, j-1]->red) ||
(pixelMatrix[i, j]->green) != (pixelMatrix[i, j-1]->green) || (pixelMatrix[i, j]->blue) !=
(pixelMatrix[i, j-1]->blue)) {
				//pixelMatrix[i, j-1]->red = 254;
				//pixelMatrix[i, j-1]->green = 254;
				//pixelMatrix[i, j-1]->blue = 254;
			//} else {
				//pixelMatrix[i-1, j-1]->red = 0;
				//pixelMatrix[i-1, j-1]->green = 0;
				//pixelMatrix[i-1, j-1]->blue = 0;
			//}
		}
	}


	return pixelMatrix;
}



array<pixel^, 2>^ inversePerspective(array<pixel^, 2>^ pixelMatrix, int imageHeight, int
imageWidth) {
	//INVERSE PERSPECTIVE CODE - DOES NOT WORK
	double verticalCameraAngle = 50*(3.14159/180);
	double horizontalCameraAngle = 112.0*(3.14159/180);
	float roadVerticalCameraAngle = (float)verticalCameraAngle/2;
	float roadHorizontalCameraAngle = (float)horizontalCameraAngle/2;
	float perPixelVerticalCameraAngle = roadVerticalCameraAngle/(imageHeight/2);
	float perPixelHorizontalCameraAngle = roadHorizontalCameraAngle/imageWidth;
```

```
	float furthestPixelCameraAngle = roadVerticalCameraAngle -
perPixelVerticalCameraAngle*((imageHeight/2)-1);
	float furthestPixelPosition = (1/tan(furthestPixelCameraAngle));
	array<pixel^, 2>^ inversePerspectiveMap = gcnew array<pixel^,
2>((int)ceil(furthestPixelPosition), imageWidth);

	for (int i = 0; i < ceil(furthestPixelPosition); i++) {
		for (int j = 0; j < imageWidth; j++) {
			inversePerspectiveMap[i, j] = gcnew pixel;
			inversePerspectiveMap[i, j]->red = 0;
			inversePerspectiveMap[i, j]->green = 0;
			inversePerspectiveMap[i, j]->blue = 0;
		}
	}

	for (int i = 0; i < imageHeight; i++) {
		for (int j = 0; j < imageWidth; j++) {
			float currentPixelVerticalCameraAngle = roadVerticalCameraAngle -
perPixelVerticalCameraAngle*(imageHeight-i);
			float currentPixelDistance = abs(1/tan(currentPixelVerticalCameraAngle));

			float distanceFromCenterline = abs(((float)imageWidth/2)-j);
			float angleDistanceFromCenterline = perPixelHorizontalCameraAngle *
distanceFromCenterline;
			if (j % 100 == 0) cout << "angleDistanceFromCenterline is " <<
angleDistanceFromCenterline << endl;
			float currentPixelWidth = distanceFromCenterline;


			if (pixelMatrix[i, j]->red == 254 && pixelMatrix[i, j]->green == 254 &&
pixelMatrix[i, j]->blue == 254 && (int)ceil(currentPixelDistance) <
(int)ceil(furthestPixelPosition) && (int)ceil(currentPixelWidth) < imageWidth) {
				inversePerspectiveMap[(int)ceil(currentPixelDistance),
(int)ceil(currentPixelWidth)]->red = 254;
				inversePerspectiveMap[(int)ceil(currentPixelDistance),
(int)ceil(currentPixelWidth)]->green = 254;
				inversePerspectiveMap[(int)ceil(currentPixelDistance),
(int)ceil(currentPixelWidth)]->blue = 254;
			}
		}
	}




	int pixelSize = 24;
	array<Byte>^ pixelSave = gcnew
array<Byte>((imageWidth*(pixelSize)*(int)ceil(furthestPixelPosition)));
	for (int i = 0; i < (int)ceil(furthestPixelPosition); i++) {
		int x = 0;
		for (int j = 0; j < (imageWidth*(pixelSize/8)); j = j+3) {
			pixelSave[(i*imageWidth*pixelSize)+j] = inversePerspectiveMap[i, x]->blue;
			pixelSave[((i*imageWidth*pixelSize)+j)+1] = inversePerspectiveMap[i, x]-
>green;
			pixelSave[((i*imageWidth*pixelSize)+j)+2] = inversePerspectiveMap[i, x]-
>red;
			x++;
		}
	}



	List<System::Windows::Media::Color>^ colors = gcnew
List<System::Windows::Media::Color>();
	colors->Add(System::Windows::Media::Colors::Blue);
	colors->Add(System::Windows::Media::Colors::Green);
	colors->Add(System::Windows::Media::Colors::Red);
	BitmapPalette^ imagePalette = gcnew BitmapPalette(colors);
	BitmapSource^ imageSource = BitmapSource::Create(imageWidth,
(int)ceil(furthestPixelPosition), 96, 96, PixelFormats::Bgr24, imagePalette, pixelSave,
imageWidth*(pixelSize));
```

```
	System::IO::FileStream^ imageSaveStream = gcnew
System::IO::FileStream("C:\\UDK\\inversePerspective.jpg", FileMode::Create);
	JpegBitmapEncoder^ encoder = gcnew JpegBitmapEncoder();
	encoder->FlipHorizontal = false;
	encoder->FlipVertical = false;
	encoder->QualityLevel = 100;
	encoder->Rotation = Rotation::Rotate0;
	encoder->Frames->Add(BitmapFrame::Create(imageSource));
	encoder->Save(imageSaveStream);

	return inversePerspectiveMap;
}



int round(float number) {
	if (number + 0.5 < number + 1) return (int)number;
	else return (int)number + 1;
}

float interpolation(int x, int x1, int x2, int x3, int y1, int y2, int y3) {
	//LAGRANGE INTERPOLATION
	float part1 = (float)y1*((float)((x-x2)*(x-x3))/(float)((x1-x2)*(x1-x3)));
	float part2 = (float)y2*((float)((x-x1)*(x-x3))/(float)((x2-x1)*(x2-x3)));
	float part3 = (float)y3*((float)((x-x1)*(x-x2))/(float)((x3-x1)*(x3-x2)));

	return part1 + part2 + part3;
}

array<pixel^, 2>^ criticalPoints(array<pixel^, 2>^ pixelMatrix, int imageHeight, int imageWidth)
{
	int entryPoint[] = {imageHeight, (int)((1.0/2.0)*imageWidth)};
	cout << "EntryPoint at beginning: " << entryPoint[0] << ", " << entryPoint[1] << endl;


	//FIGURING OUT WHICH WAY THE ROAD TURNS
	List<int>^ centerOfRoad = gcnew List<int>();
	int pointFurthestRight = 0;
	for (int i = (int)(imageHeight/2); i < imageHeight; i++) {
		bool leftEdgeFound = false;
		int leftEdge = 0;
		for (int j = 0; j < imageWidth && !leftEdgeFound; j++) {
			if (pixelMatrix[i, j]->red < 50 && pixelMatrix[i, j]->green < 50 &&
pixelMatrix[i, j]->blue < 50) {
				leftEdge = j;
				leftEdgeFound = true;
			}
		}

		bool rightEdgeFound = false;
		int rightEdge = 0;
		for (int j = imageWidth-1; j > 0 && !rightEdgeFound; j--) {
			if (pixelMatrix[i, j]->red < 50 && pixelMatrix[i, j]->green < 50 &&
pixelMatrix[i, j]->blue < 50) {
				rightEdge = j;
				rightEdgeFound = true;
			}
		}

		int centerPoint = ((leftEdge + rightEdge)/2);
		if (centerPoint >= pointFurthestRight) pointFurthestRight = centerPoint;
		centerOfRoad->Add(centerPoint);
	}


	//DETERMINING THE TANGENT-TO-THE-CENTER-OF-THE-ROAD LINE
	float projectionSlope = ((float)(((imageHeight/2)+30) -
(imageHeight/2)))/((float)(centerOfRoad[30] - centerOfRoad[0]));
	int projectionConstant = (int)(((imageHeight/2)+30) - projectionSlope*centerOfRoad[30]);
	int averageYDiff = 0;
```

```
	//DRAWING THE TANGENT LINE
	for (int i = 0; i < centerOfRoad->Count; i++) {
		averageYDiff += (int)(i - ((projectionSlope*centerOfRoad[i]) +
projectionConstant));
		pixelMatrix[i+(int)(imageHeight/2), centerOfRoad[i]]->red = 254;
		pixelMatrix[i+(int)(imageHeight/2), centerOfRoad[i]]->green = 0;
		pixelMatrix[i+(int)(imageHeight/2), centerOfRoad[i]]->blue = 0;
	}

	//DETERMINING IF THE AVERAGE LOCATION OF THE CENTER OF THE ROAD IS TO THE LEFT OR TO THE
RIGHT OF THE TANGENT LINE
	averageYDiff = averageYDiff/centerOfRoad->Count;
	cout << "Average: " << averageYDiff << ", Last Point: " << centerOfRoad[(centerOfRoad-
>Count)-1] << endl;
	bool turningLeft = false;
	if (averageYDiff <= 0 && projectionSlope > 0) turningLeft = true;
	if (averageYDiff >= 0 && projectionSlope <= 0) turningLeft = true;

	cout << "Turning Left: " << turningLeft << endl;
	cout << "Projection Slope: " << projectionSlope << endl;



	//FINDING THE EXIT POINT
	cout << "FINDING THE EXIT POINT" << endl;
	int exitPoint[] = {(int)(imageHeight/2), 0};
	if (!turningLeft) {
		bool exitPointFound = false;
		for (int j = 0; j < imageWidth && !exitPointFound; j++) {
			if (pixelMatrix[(int)(imageHeight/2), j]->red < 50 &&
pixelMatrix[(int)(imageHeight/2), j]->green < 50 && pixelMatrix[(int)(imageHeight/2), j]->blue <
50) {
				exitPoint[1] = j;
				exitPointFound = true;
			}
		}
	} else {
		bool exitPointFound = false;
		for (int j = imageWidth-1; j >= 0 && !exitPointFound; j--) {
			if (pixelMatrix[(int)(imageHeight/2), j]->red < 50 &&
pixelMatrix[(int)(imageHeight/2), j]->green < 50 && pixelMatrix[(int)(imageHeight/2), j]->blue <
50) {
				exitPoint[1] = j;
				exitPointFound = true;
			}
		}
	}


	//FINDING THE APEX
	cout << "FINDING THE APEX" << endl;
	float slope = ((float)(exitPoint[0]-entryPoint[0]))/((float)(exitPoint[1]-
entryPoint[1]));
	int constant = (int)(exitPoint[0]-(slope*exitPoint[1]));
	int oldConstant = constant;

	cout << "Slope: " << slope << ", " << "Constant: " << constant << endl;


	int apex[] = {0, 0};

	//COUNTING THE NUMBER OF POINTS OF THE GRASS OF THE TURN THAT INTERSECT THE LINE BETWEEN
THE ENTRY AND EXIT POINTS
	int lineCount = 100000;
	while (lineCount > 1) {
		lineCount = 0;
		int lastPoint[] = {-1, -1};
		for (int i = (int)(imageHeight/2); i < imageHeight; i++) {
			for (int j = 0; j < imageWidth; j++) {
				if (pixelMatrix[i, j]->red > 50 && pixelMatrix[i, j]->green > 50 &&
pixelMatrix[i, j]->blue > 50 && abs(i - round((slope*j) + constant)) < 1 && (turningLeft ? ((j -
centerOfRoad[i-(int)(imageHeight/2)]) <= 0) : ((j - centerOfRoad[i-(int)(imageHeight/2)]) > 0)))
{
```

```
                                        lineCount++;
                                        lastPoint[0] = i;
                                        lastPoint[1] = j;
                                }
                        }
                }

                //IF WE FOUND THE LAST POINT
                if (lineCount == 1) {
                        apex[0] = lastPoint[0];
                        apex[1] = (int)((6.0/7.0)*lastPoint[1] +
(1.0/7.0)*centerOfRoad[lastPoint[0]-
(int)(imageHeight/2)]);//(lastPoint[1]+centerOfRoad[lastPoint[0]-(int)(imageHeight/2)])/2;
                        cout << "Found last point" << endl;
                } else if (lineCount == 0) {
                        apex[0] = (lastPoint[0] == -1 && lastPoint[1] == -1) ?
(int)(2*imageHeight/3) : lastPoint[0]; //FIRST CONDITION HANDLES NO APEX
                        apex[1] = (lastPoint[0] == -1 && lastPoint[1] == -1) ?
(int)(abs(entryPoint[1]+exitPoint[1])/2) : (int)((6.0/7.0)*lastPoint[1] +
(1.0/7.0)*centerOfRoad[lastPoint[0]-(int)(imageHeight/2)]); //FIRST CONDITION HANDLES NO APEX
                        cout << "Found no points" << endl;
                }
                constant--;
        }

        //PAINTING LINE BETWEEN ENTRY AND EXIT POINTS
        constant = oldConstant;
        for (int x = 0; x < imageWidth; x++) {
                int y = (int)(slope*x + constant);
                if (y > (int)(imageHeight/2) && y < (imageHeight)) {
                        pixelMatrix[y, x]->red = 200;
                        pixelMatrix[y, x]->green = 0;
                        pixelMatrix[y, x]->blue = 200;
                }
        }

        cout << "Entry point: (" << entryPoint[0] << ", " << entryPoint[1] << "), Apex: (" <<
apex[0] << ", " << apex[1] << "), Exit point: (" << exitPoint[0] << ", " << exitPoint[1] << ")"
<< endl;



        int xMax;
        if (entryPoint[1] >= apex[1] && entryPoint[1] >= exitPoint[1]) xMax = entryPoint[1];
        if (apex[1] >= entryPoint[1] && apex[1] >= exitPoint[1]) xMax = apex[1];
        if (exitPoint[1] >= entryPoint[1] && exitPoint[1] >= apex[1]) xMax = exitPoint[1];

        int xMin;
        if (entryPoint[1] <= apex[1] && entryPoint[1] <= exitPoint[1]) xMin = entryPoint[1];
        if (apex[1] <= entryPoint[1] && apex[1] <= exitPoint[1]) xMin = apex[1];
        if (exitPoint[1] <= entryPoint[1] && exitPoint[1] <= apex[1]) xMin = exitPoint[1];


        //PAINTING THE PROJECTED LINE
        for (int x = 0; x < imageWidth; x++) {
                int y = (int)(projectionSlope*x + projectionConstant);
                if (y > (int)(imageHeight/2) && y < (imageHeight)) {
                        pixelMatrix[y, x]->red = 128;
                        pixelMatrix[y, x]->green = 128;
                        pixelMatrix[y, x]->blue = 128;
                }
        }

        //PAINTING THE INTERPOLATION
        for (int i = (int)(imageHeight/2); i < imageHeight; i++) {
                for (int j = 0; j < imageWidth; j++) {
                        if (j == round(interpolation(i, entryPoint[0], apex[0], exitPoint[0],
entryPoint[1], apex[1], exitPoint[1]))) {
                                pixelMatrix[i, j]->red = 0;
                                pixelMatrix[i, j]->green = 254;
                                pixelMatrix[i, j]->blue = 0;
                        }
                }
```

```
        }


        //PAINTING ENTRY POINT
        for (int i = entryPoint[0]-5; i < entryPoint[0]+5; i++) {
                for (int j = entryPoint[1]-5; j < entryPoint[1]+5; j++) {
                        if (j >= 0 && j < imageWidth && i >= 0 && i < imageHeight) {
                                pixelMatrix[i, j]->red = 254;
                                pixelMatrix[i, j]->green = 254;
                                pixelMatrix[i, j]->blue = 0;
                        }
                }
        }

        //PAINTING EXIT POINT
        for (int i = exitPoint[0]-5; i < exitPoint[0]+5; i++) {
                for (int j = exitPoint[1]-5; j < exitPoint[1]+5; j++) {
                        if (j >= 0 && j < imageWidth && i >= 0 && i < imageHeight) {
                                pixelMatrix[i, j]->red = 254;
                                pixelMatrix[i, j]->green = 254;
                                pixelMatrix[i, j]->blue = 0;
                        }
                }
        }

        //PAINTING APEX
        for (int i = apex[0]-5; i < apex[0]+5; i++) {
                for (int j = apex[1]-5; j < apex[1]+5; j++) {
                        if (j >= 0 && j < imageWidth && i >= 0 && i < imageHeight) {
                                pixelMatrix[i, j]->red = 254;
                                pixelMatrix[i, j]->green = 254;
                                pixelMatrix[i, j]->blue = 0;
                        }
                }
        }

        return pixelMatrix;
}
```